Coherence-Centric Logging and Recovery for Home-Based Software Distributed Shared Memory

Angkul Kongmunvattana and Nian-Feng Tzeng

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

Abstract

The probability of failures in software distributed shared memory (SDSM) increases as the system size grows. This paper introduces a new, efficient message logging technique, called the coherence-centric logging (CCL) and recovery protocol, for home-based SDSM. Our CCL minimizes failure-free overhead by logging only data necessary for correct recovery and tolerates high disk access latency by overlapping disk accesses with coherence-induced communication existing in home-based SDSM, while our recovery reduces the recovery time by prefetching data according to the future shared memory access patterns, thus eliminating the memory miss idle penalty during the recovery process. To the best of our knowledge, this is the very first work that considers crash recovery in home-based SDSM.

We have performed experiments on a cluster of eight SUN Ultra-5 workstations, comparing our CCL against traditional message logging (ML) by modifying TreadMarks, a state-of-the-art SDSM system, to support the home-based protocol and then implementing both our CCL and the ML protocols in it. The experimental results show that our CCL protocol consistently outperforms the ML protocol: Our protocol increases the execution time negligibly, by merely 1% to 6%, during failure-free execution, while the ML protocol results in the execution time overhead of 9% to 24% due to its large log size and high disk access latency. Our recovery protocol improves the crash recovery speed by 55% to 84% when compared to re-execution, and it outperforms ML-recovery by a noticeable margin, ranging from 5% to 18% under parallel applications examined.

1. Introduction

Clusters of workstations, PCs, or SMPs represent cost-effective platforms for parallel computing. Programming on such clusters with explicit message exchange, however, is known to be difficult. Software distributed shared memory (SDSM) creates a shared memory abstraction (i.e., a coherent global address space) on top of the physically distributed processing nodes (i.e., processors) to simplify parallel programming tasks. While faster processors and higher network bandwidth continue to improve SDSM performance and scalability, the probability of system failures increases as the system size grows. This existence of vulnerability is unacceptable, especially for long-running applications and high-availability situations. Hence, a mechanism for supporting fast crash recovery in SDSM is indispensable.

Home-based SDSM [18] is one type of SDSM developed under the notion of relaxed memory consistency model [1]. While it relies on a virtual memory trap as other SDSM systems [2, 5], home-based SDSM assigns a *home* node for each shared memory page to collect updates from all writers of that page. This home node offers some advantages to SDSM as follows: (i) a read/write to a page on its home node does not cause any page fault nor requires any summary of write modifications, (ii) it takes only one round-trip message to bring a remote copy of any shared memory page up-to-date, and (iii) no garbage collection is needed. Due to these advantages, the home-based SDSM protocol has been a focus of several recent studies [7, 10, 14, 18]. Unfortunately, no prior work has ever been attempted on crash recovery in home-based SDSM. This paper is the very first one to deal with crash recovery in such SDSM.

Message logging is a popular technique for providing *home-less* SDSM with fault-tolerant capability [6, 11, 12, 13, 17]. This technique is attractive because it guarantees bounded rollback-recovery of independent checkpointing and improves the crash recovery speed of coordinated checkpointing [8]. While those earlier logging protocols work well under home-less SDSM, they cannot be applied to home-based SDSM, where each shared memory page has a home node at which updates from all writers are collected. This is because home-less SDSM maintains its coherence enforcement data differently from home-based SDSM. Specifically, home-less SDSM always keeps a summary of modifications made to each shared memory page (known as *diff*), while home-based SDSM creates diffs only for remote copies of shared memory pages and discards them after a home copy is updated.

In this paper, we propose a new, efficient logging protocol, called *coherence-centric* logging (CCL), for home-based SDSM. CCL minimizes logging overhead by overlapping disk accesses (for flushing) with coherence-induced communication in home-based SDSM, and by selecting to record only information indispensable for recovery. To assess the impacts of our CCL protocol on home-based SDSM performance, we have modified TreadMarks (a state-of-the-art home-less SDSM) to support the home-based protocol, with traditional message logging and then with our CCL incorporated in it. The experiment has been conducted on a cluster of eight Sun Ultra-5 workstations. Experimental results under four parallel applications demonstrate that our CCL protocol leads to very low failure-free overhead, ranging from 1% to 6% of the normal execution time. For fast recovery, we also introduce an efficient recovery scheme, which is specifically tailored for home-based SDSM. Our recovery scheme reads coherence enforcement data from the local disk and fetches the updates from the logged data on remote nodes (i.e., other writers) at the beginning of each time interval, reducing disk access frequency and eliminating the memory miss idle time. Because of these optimization techniques and lighter network traffic during crash recovery, our scheme shortens the recovery time substantially, ranging from 55% to 84%, when compared with re-execution.

The outline of this paper is as follows. Section 2 describes home-based SDSM [18] and its coherence enforcement protocol. Our coherence-centric logging and recovery are proposed in Section 3, along with an overview on the traditional message logging and its recovery procedure. Section 4 first states our experimental platform and the parallel applications used, and then presents the performance results of our proposed protocol. Related work is discussed in Section 5, followed by conclusion in Section 6.

2. Home-Based SDSM

Coherence-centric logging and recovery we propose aim at home-based SDSM. They take advantage of properties associated with home-based SDSM to lower failure-free overhead. In this section, we describe home-based SDSM and its coherence enforcement protocol, known as home-based lazy release consistency (HLRC), before explaining optimization steps adopted by HLRC.

2.1 Overview

SDSM simplifies parallel programming tasks 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 coherence enforcement of a shared memory image, which spans across memories at the interconnected processors. Such hardware support is readily available in most, if not all, commodity microprocessors. While versatile and attractive, SDSM has to minimize its coherence enforcement overhead for achieving good performance. To this end, the implementation of SDSM usually adopts a relaxed memory consistency model [1] because various optimization steps can then be applied to arrive at efficient SDSM systems.

Release consistency (RC) [1] is one of the least restrictive relaxed memory consistency models; it does not guarantee that shared memory is consistent all of the time, but rather it makes sure consistence only after synchronization operations (i.e., locks and barriers). In essence, RC ensures a synchronized program to see 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 also allows to update a local copy of shared data multiple times, but all the updates have to be completed before the release is performed. Home-based SDSM employs a variant of RC implementations called home-based lazy release consistency (HLRC) [18] and also incorporates several optimization steps, as described in sequence below.

2.2 Home-Based Lazy Release Consistency

Home-based lazy release consistency (HLRC) [18] is among the efficient software implementations of RC. It assigns a home node for each shared memory page as a repository of updates from all writers, simplifying memory management of coherence enforcement mechanism. HLRC also combines the advantages of write-update and write-invalidate protocols to make a shared memory page at the home node (i.e., a home copy) up-to-date at the end of each writer time interval, and to bring other copies (i.e., remote copies) up-to-date on demands. Specifically, writers of remote copies flush updates (i.e., a summary of modifications) to its home node at the end of each time interval (i.e., via a release operation), updating its home copy. Consequently, no persistent state has to be kept in memory. All other remote copies are invalidated at the beginning of the subsequent interval (i.e., by an acquire operation), according to the write-invalidation notice piggybacked with a lock grant or barrier release message. Each remote copy is updated using only a single round-trip message to the home node, where updates from all writers are collected. Hence, HLRC minimizes memory consumption for supporting coherence enforcement via a write-update protocol and reduces coherence-induced traffic via a write-invalidate protocol, leading to good performance and scalability.

2.3 Optimization

SDSM utilizes the virtual memory trap to invoke coherence enforcement mechanisms, and therefore, coherence granularity in SDSM is of the OS page size. This coarse-grain coherence enforcement often leads to a false-sharing scenario, where several processes request to modify different portions of the same shared memory page simultaneously, resulting in a thrashing problem or ping-pong effect. A multiple-writer protocol alleviates this problem by allowing multiple writable copies of a shared memory page to coexist [5]. This is permissible under the definition of RC, and the correct execution of a program is guaranteed as long as it is data-race free. The use of a multiple-writer protocol optimizes the performance of home-based SDSM. Another optimization step taken by home-based SDSM is through diff-based write propagation [5]. It uses the summaries of modifications (i.e., diffs) for updating the home copy of a shared memory page, reducing the amount of data transfer when the write granularity is small.

3. Recoverable Home-Based SDSM

This section first gives an overview on the traditional message logging protocol and then proposes our coherence-centric logging and recovery. The proposed logging and recovery protocol takes advantage of the properties of home-based SDSM to yield low overhead. It is the first attempt to deal with fault tolerance in home-based SDSM.

3.1 Overview on Traditional Message Logging

Message logging (ML) has its root in message-passing systems [4, 8, 9, 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 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 most recent checkpoint and the logged messages are replayed to reconstruct a consistent state of execution before the failure. When this traditional ML protocol is applied to home-based SDSM, it keeps in its local memory the updates (i.e., diffs) sent from writer processes, the up-to-date copy of shared memory pages delivered from their home nodes, and write-invalidation notices. These volatile logs are flushed to stable storage (i.e., a local disk) at the subsequent synchronization point. Should a failure occur, recovery starts from the most recent checkpoint and generates the execution by replaying the logged data from non-volatile storage at each synchronization point and at each memory miss. While this logging protocol is straightforward to implement and its recovery process is easy to follow, it has high overhead due to its large log size and frequent disk accesses. To make fault-tolerant SDSM more affordable, we propose in the next subsection a new, efficient coherence-centric logging and recovery protocol for home-based SDSM.

3.2 Proposed Logging and Recovery

Our coherence-centric logging (CCL) protocol is tightly integrated into the coherence enforcement protocol employed by home-based SDSM, namely HLRC (refer to Section 2 for details). It records only information indispensable to correct recovery, minimizing the amount of logged data, and overlaps its disk accesses with coherence-induced communications already present in HLRC, reducing the adverse impacts of high disk access latency. In particular, CCL keeps in its local disk, the incoming write-invalidation notices, the records of incoming updates together with a writer process id number, and the summary of modifications (i.e., diffs) produced at the end of each time interval. Diffs are created by comparing locally a remote copy of a shared memory page with its twin (i.e., a pristine copy created before a write operation). The disk accesses (i.e., flushing operations) are performed during the release operation (i.e., right after the diffs are sent to the respective home nodes of pages), overlapping high disk access latency with inter-process communication. Hence, CCL allows HLRC to discard a diff as soon as the diff has been applied to its corresponding home copy, whereas ML needs to keep such a diff in the memory until the next synchronization point, where the disk flushing operation will be performed; additionally, when a page fault occurs at an invalid remote copy, CCL does not keep a received copy of a shared memory page that HLRC has fetched from its home node because such an up-to-date copy can be reconstructed during recovery. As a result, the total log size of CCL is only about 10% of that created by ML, as will be detailed in Section 4. This log size reduction and the disk access latency-tolerant technique save both space and time in the logging process, leading to significant failure-free overhead reduction.

To limit the amount of work that has to be repeated after a failure, checkpoints are created periodically. A checkpoint consists of all local and shared memory contents, the state of execution, and all internal data structures used by home-based SDSM. All this information is needed at the beginning of a crash recovery process to avoid restarting from the (global) initial state. The first checkpoint flushes all shared memory pages to stable storage, and then only those pages that have been modified since the last checkpoint will be included in a subsequent checkpoint, reducing the checkpoint creation overhead. While our logging protocol allows each process to create its checkpoints independently and guarantees bounded rollback-recovery, it is applicable to coordinated checkpointing as well (i.e., 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, however, is beyond the scope of this paper.

Recovery

As mentioned earlier, the recovery process starts from the most recent checkpoint, followed by the execution replay of logged data. At the beginning of each time interval, the recovery process fetches the corresponding logs of updates (i.e., diffs) for its home copy from the writer process(es), as provided in the records of incoming updates. Our recovery brings a remote copy up-to-date by first requesting for an up-to-date page from its home node; if such a page is not available (i.e., a home copy has been advanced to another time interval by diffs from other writers), a remote copy has to be reconstructed by fetching both the home copy from the most recent checkpoint of its home node and logs of updates from the writer process(es), corresponding to information provided by the logged write-invalidation notices. Obviously, in the worst case, the home node must rollback to the most recent checkpoint in order to recreate its modification. While it is evident that this recovery scheme is inspired by our previous work, where an efficient approach was introduced for crash recovery in home-less SDSM [11], prefetching is more complicated in recoverable home-based SDSM than in home-less SDSM. In particular, prefetching in home-less SDSM simply relies on the diffs and write-invalidation notices maintained by a coherence enforcement protocol of home-less SDSM, whereas prefetching in home-based SDSM requires a home copy, logs of write-invalidation notices, and logs of updates to reconstruct a remote copy. Despite its higher complexity, our recovery for home-based SDSM substantially improves the crash recovery speed, ranging from 55% to 84% when compared with re-execution, under all applications examined. This reduction in recovery time results from the smaller total log size of CCL, the elimination of page faults, and lighter traffic over the network during recovery.

Example

Figure 1 shows a snapshot of coherence-centric logging and recovery in home-based SDSM. In this example, home nodes of pages *x, y,* and *z* are processes P_1, P_2, and P_3, respectively. During the failure-free execution, P_1 acquires the lock (for interval A), writes on pages *x, y,* and *z*, and then releases the lock. At the time of lock release, P_1 flushes diff of page *y* to P_2 and diff of page *z* to P_3, following the HLRC protocol. In addition, P_1 also stores those diffs in its local disk, as required by our CCL. When P_2 and P_3 receive asynchronous updates (i.e., diffs from P_1), they apply

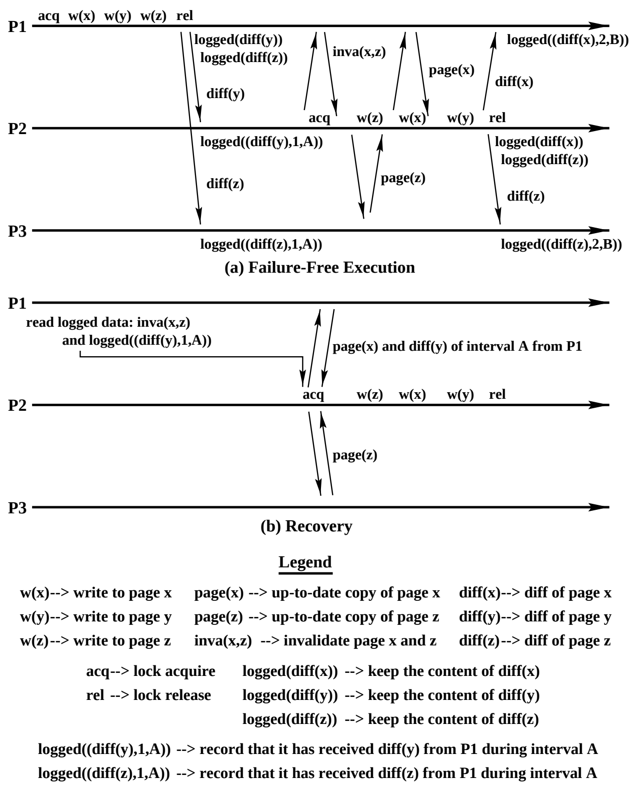
P1
acq w(x) w(y) w(z) rel
logged(diff(y))
logged(diff(z))
diff(y)
inva(x,z)
page(x)
logged((diff(x),2,B))
diff(x)
P2
acq w(z) w(x) w(y) rel
logged((diff(y),1,A))
page(z)
logged(diff(x))
logged(diff(z))
diff(z)
diff(z)
P3
logged((diff(z),1,A))
logged((diff(z),2,B))
(a) Failure-Free Execution
P1
read logged data: inva(x,z)
and logged((diff(y),1,A))
page(x) and diff(y) of interval A from P1
P2
acq w(z) w(x) w(y) rel
page(z)
P3
(b) Recovery
Legend
w(x)--> write to page x
w(y)--> write to page y
w(z)--> write to page z
page(x) --> up-to-date copy of page x
page(z) --> up-to-date copy of page z
inva(x,z) --> invalidate page x and z
diff(x)--> diff of page x
diff(y)--> diff of page y
diff(z)--> diff of page z
acq--> lock acquire
rel --> lock release
logged(diff(x)) --> keep the content of diff(x)
logged(diff(y)) --> keep the content of diff(y)
logged(diff(z)) --> keep the content of diff(z)
logged((diff(y),1,A)) --> record that it has received diff(y) from P1 during interval A
logged((diff(z),1,A)) --> record that it has received diff(z) from P1 during interval A

Figure 1. A Snapshot of Coherence-Centric Logging and Recovery in Action.

incoming diffs to their respective home copies and record this asynchronous update event. Later on, P_2 acquires the lock (for interval B), and invalidates its remote copies of pages *x* and *z*, according to the write-invalidation notices piggybacked with a lock grant message. When P_2 attempts to write on pages *z* and *x*, it causes virtual memory traps, which in turn fetch up-to-date copies of pages *z* and *x* from their home nodes. Accessing page *y* on P_2 causes no page fault because the home copy is always valid. At the time of lock release, P_2 flushes diff of page *x* to P_1 and diff of page *z* to P_3. It also stores those diffs in its local disk. When P_1 and P_3 receive asynchronous updates (i.e., diffs from P_2), they apply incoming diffs to their respective home copies and record this asynchronous update event.

Figure 1(b) assumes that P_2 crashes a certain time after the volatile logs of this interval are flushed to the local disk, but before the next checkpoint is created. After a failure is detected, the recovery process starts from the last checkpoint and replays the logged data. During this snapshot interval, the recovery process P_2 reads its local logged data and discovers that it has to update remote copies of pages *x* and *z*, and that it also receives asynchronous updates from P_1. Consequently, P_2 fetches the up-to-date copy of page *z* from its home node, P_3, and retrieves the up-to-date copy of page *x* along with a diff of page *y* from process P_1, following our recovery scheme.

3.3 Implementation

Figures 2 and 3 show the procedure of our logging and recovery for implementing recoverable home-based SDSM.

Lock

During the failure-free execution, a lock release operation not only creates and then flushes (to the home node) the summary of modifications, called updates or diffs, made to shared memory pages during the previous time interval, but also stores such a summary into its local disk (along with the write-invalidation notices it received at the beginning of this interval, i.e., at the lock acquire, and the records of incoming updates from other writers to its home copy). In essence, on a lock acquire operation, the acquiring process sends a message to the lock manager requesting for the ownership of the lock. When the the lock grant message (piggybacked with write-invalidation notices) arrives, the acquiring process invalidates its remote copies of shared memory pages accordingly and keeps the write-invalidation notices in its memory, waiting to be flushed to the local disk at the subsequent lock release operation. Whenever the home node receives asynchronous updates from writer processes, it records the event of such updates (not its contents) in its memory, consisting of the interval number, page id of a home copy, and the writer process id. At the lock release, a process flushes its summary of modifications to the home node(s) of corresponding pages and also stores such a summary in its local disk, along with the write-invalidation notices and the records of incoming updates.

Lock Acquire

```
if (in_recovery)
  read logs of incoming update events from its local disk;
  update home copies with logs of diffs from the writer process(es);
  read logs of write-invalidation notices from its local disk;
  reconstruct each remote copy with a home copy from its home node
       and logs of diffs from the writer process(es);
  create twin for each remote copy that will be written in the next
       time interval;
else
  send lock request to lock manager;
  wait for lock grant message (piggybacked with write-inv. notices);
  invalidate remote copies of pages according to write-inv. notices;
  keep those write-invalidation notices in memory;
endif
```

Lock Release

```
if (in_recovery)
  release the lock;
else
  create diffs and flush them to home nodes;
  flush those diffs, write-invalidation notices, and records of
       incoming updates events to local disk;
  wait for acknowledgements and then discard those diffs;
endif
```

Asynchronous Update Handler

```
apply received diffs to home copies;
send an acknowledgement back to the writer process;
records this incoming updates event in memory;
discard those diffs;
```

Figure 2. Procedure for Lock Synchronization in Coherence-Centric Logging and Recovery.

Barrier

```
if (in_recovery)
  read logs of incoming update events from its local disk;
  update home copies with logs of diffs from the writer process(es);
  read logs of write-invalidation notices from its local disk;
  reconstruct each remote copy with a home copy from its home node
       and logs of diffs from the writer process(es);
  create twin for each remote copy that will be written in the next
       time interval;
else
  if (barrier_manager)
    create diffs and flush them to home nodes;
    flush those diffs to local disk;
    wait for acknowledgements and then discard those diffs;
    wait for all other processes to check in;
    invalidate dirty remote copies of pages;
    send barrier release message piggybacked with write-inv.
        notices to all other processes;
  else
    create diffs and flush them to home nodes;
    flush those diffs to local disk;
    wait for acknowledgements and then discard those diffs;
    send check-in message to barrier manager;
    wait for barrier release message;
    invalidate dirty remote copies of pages;
  endif
endif
```

Figure 3. Procedure for Barrier Synchronization in Coherence-Centric Logging and Recovery.

When a failure is detected, the *in_recovery* flag is set. On a lock acquire, the home copy is prepared for the next time interval by updating itself with logs of diffs retrieved from the writer process(es), as recorded in the logs of incoming updates events. To get the correct remote copies of pages for the next time interval, the recovery process fetches an up-to-date copy of those pages from their respective home nodes. Remote copies that need to be updated are identified by the logs of write-invalidation notices in that time interval. If the home copy is already in a more advanced time interval (i.e., diffs from other writers have been applied to the home copy), then the recovery process has to fetch those diffs from the writer process(es) to reconstruct its remote copy to the correct time interval before proceeding beyond the lock acquire.

Barrier

During a failure free period, on arrival at a barrier, each process that has performed a write operation to shared memory page(s) creates and flushes to the respective home node(s) a summary of modifications (i.e., diff) of each remote copy. Such a process also logs those diffs into its local disk while waiting for an acknowledgement from each home node. Upon receiving asynchronous updates from a remote process, the home node applies received diffs to its home copy, sends an acknowledgement back to the writer process, records this incoming updates event, and discards the diffs. Each process also discards its local diffs after an acknowledgement has arrived, and sends a check-in message (piggybacked with the write-invalidation notice) to the barrier manager. The barrier manager waits until it received all check-in messages, before invalidates its remote copies of shared memory pages in accordance with the write-invalidation notice, and then sends back a check-out message (piggybacked with an up-to-date write-invalidation notice) to each process, allowing the recipient of a check-out message to invalidate its remote copies of shared memory pages accordingly and to proceed beyond the barrier.

When a failure is detected, the *in_recovery* flag is set. On arrival at the barrier, the home copy is prepared for the next time interval by updating itself with logs of diffs retrieved from the writer process(es), as recorded in the logs of incoming update events. To get the correct remote copies of pages for the next time interval, the recovery process fetches an up-to-date copy of those pages from their respective home nodes. The remote copies that need to be updated are identified by the logs of write-invalidation notices in that time interval. If the home copy is already in a more advanced time interval (i.e., diffs from other writers have been applied to the home copy), the recovery process has to fetch those diffs from the writer process(es) to reconstruct its remote copy to the correct time interval before proceeding beyond the barrier.

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 protocol for recoverable home-based SDSM. We first compare the performance results of our coherence-centric logging (CCL) with those of traditional message logging (ML). Since the goal is to examine logging overhead, no checkpoint is taken in our experiments. Subsequently, we evaluate and contrast the crash recovery speeds of our recovery and ML-recovery.

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) switch. We allocated 2 GB of the local disk at each workstation for virtual memory paging and left 1.2 GB of local disk space available for logged data.

Program	Data Set Size	Synchronization
3D-FFT	100 iterations on $2^7 \times 2^7 \times 2^7$ data	barriers
MG	200 iterations on $2^7 \times 2^7 \times 2^7$ grid	barriers
Shallow	5000 iterations on 25000^2	barriers
Water	120 iterations on 512 molecules	locks and barriers

Table 1. Applications' Characteristics.

We modified TreadMarks [2] to support the HLRC protocol [18], and then incorporated our proposed logging and crash recovery for quantitative evaluation. For measuring the failure-free overhead and the crash recovery speed, four parallel applications were used in our experiments, including 3D-FFT, MG, Shallow, and Water. 3D-FFT and MG are originally from the NAS benchmark suite [3], with the former computing the 3-Dimensional Fast Fourier Transform, and the latter solving the Poisson problem on a multigrid. Shallow is a weather prediction kernel from NCAR, and Water is a molecular dynamics simulation from the SPLASH benchmark suite [15]. The data set size and the synchronization type of each application used in this experimental study are listed in Table 1.

4.2 Performance Results of Logging Protocols

Table 2 presents the failure-free overhead of different logging protocols. It lists total execution time, the mean log size, the total log size, and the number of times the volatile logs are flushed to stable storage. Since no logging exists in the home-based TreadMarks, its execution time serves as a performance baseline. Both our coherence-centric logging (CCL) and traditional message logging (ML) protocols provide home-based SDSM with crash recovery capabilities, but they involve different performance penalty amounts.

Logging Protocol	Execution Time (sec.)	Mean Log Size (KB)	Total Log Size (MB)	# of Flushes
None	363.24	-	-	-
ML	450.36	1760	359	204
CCL	385.14	221	45	204

(a) 3D-FFT

Logging Protocol	Execution Time (sec.)	Mean Log Size (KB)	Total Log Size (MB)	# of Flushes
None	300.96	-	-	-
ML	354.56	43	276	6404
CCL	307.10	4	24	6404

(b) MG

Logging Protocol	Execution Time (sec.)	Mean Log Size (KB)	Total Log Size (MB)	# of Flushes
None	744.33	-	-	-
ML	848.60	29	865	30000
CCL	765.22	2.4	71	30000

(c) Shallow

Logging Protocol	Execution Time (sec.)	Mean Log Size (KB)	Total Log Size (MB)	# of Flushes
None	139.64	-	-	-
ML	152.27	1.1	44	38842
CCL	141.60	0.05	2	38842

(d) Water

Table 2. Overhead Details under Different Logging Protocols.

From Table 2, it is apparent that our CCL consistently results in lower failure-free overhead than ML, which induces a larger mean log size and is affected significantly by high disk access latency. CCL keeps a far less amount of logged data because it stores only coherence-related information that cannot be retrieved or recreated after a failure, whereas ML simply records all incoming messages. Consequently, the total log size of our protocol is only a small

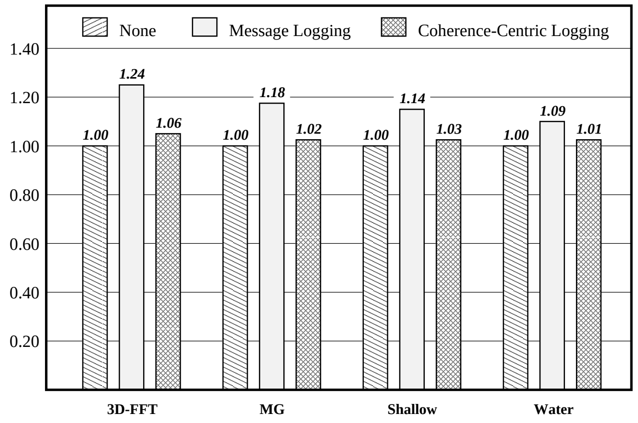

Figure 4. Impacts of Logging Protocols on Execution Time.

fraction of that of the ML protocol; namely, it is only 12.5% for 3D-FFT, 8.7% for MG, 8.2% for Shallow, and 4.5% for Water. A larger mean log size of ML also increases the disk access time during flushing operations, lengthening the critical path of program execution and hampering home-based SDSM performance. To minimize the adverse impact of disk access latency, our CCL *overlaps* flushing operations with coherence-induced communication already existing in home-based SDSM. Figure 4 depicts the impacts of different logging protocols on home-based SDSM performance using the normalized execution time. It demonstrates that our CCL protocol adds very little overhead to execution time, ranging from 1% to 6%. This low overhead results directly from a small log size and our disk access latency-tolerant technique. By contrast, ML increases the execution time by as much as 24% for 3D-FFT, 18% for MG, 14% for Shallow, and 9% for Water; it directly corresponds to the large mean log size and high disk access latency.

4.3 Performance Results of Crash Recovery

Since there is no crash recovery protocol incorporated in the original home-based 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 with no time saving. This normal execution time is employed as a baseline for crash recovery performance comparison. Under our crash recovery scheme, the recovery time is shortened substantially when compared with re-execution, by as much as 84% for 3D-FFT, 73% for MG, 55% for Shallow, and 62% for Water (see Figure 5). This directly results from the avoidance of page fault via prefetching in our recovery scheme, the minimized disk access time due to a small mean log size of CCL, and lighter traffic over the networks during recovery. On the other hand, message logging recovery (ML-recovery) eliminates the need of message transmission, but suffers from memory miss idle time and high disk

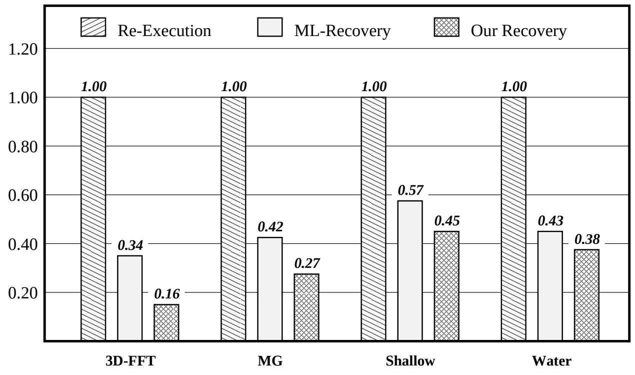

Figure 5. Impacts of Logging Protocols on Execution Time.

access latency in reading large logged data, which lengthen the recovery time. From Figure 5, ML-recovery leads to recovery time reduction of 66% for 3D-FFT, 58% for MG, 43% for Shallow, and 57% for Water, in comparison with re-execution. Hence, our recovery scheme outperforms ML-recovery by a noticeable margin, ranging from 5% to 18% under parallel applications examined.

5. Related Work

The study of message logging for SDSM was attempted first by Richard and Singhal [13]. They considered logging and independent checkpointing protocols for sequentially consistent home-less 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 home-less SDSM under relaxed memory consistency models [17], realized by logging all 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 [6], based upon the idea of sender-based message logging protocols [9], in which the logged data are kept in volatile memory of the sender process. Unfortunately, the protocol cannot handle multiple-node failures. Park and Yeom [12] described a logging protocol known as reduced-stable logging (RSL), 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 RSL utilized less disk space for the logged data than that of [17]. Recently, a superior logging protocol, dubbed lazy logging, which significantly reduces the log size and the number of accesses to stable storage, has been devised [11]. Experimental results have shown that lazy logging induces lower execution overhead than

RSL. While these earlier logging protocols work well for home-less SDSM under relaxed memory consistency models, they do not recognize the notion of home node and cannot be applied efficiently to home-based SDSM. Our new, efficient coherence-centric logging and recovery protocol is the first one to deal with logging and recovery for home-based SDSM. Its experimental results have been gathered, demonstrated, and discussed in Section 4.

6. Conclusions

We have proposed a new, efficient logging and recovery protocol for recoverable home-based SDSM implementation in this paper. The experiment outcomes reveal that our coherence-centric logging protocol incurs very low failure-free overhead, roughly 1% to 6% of the normal execution time, and that our crash recovery scheme improves crash recovery speed by 55% to 84% when compared with re-execution. This results directly from our coherence-centric logging, which keeps only information necessary for recovery to a consistent execution state, and from our disk access latency-tolerant technique, which overlaps the disk flushing operation with inter-process communication. Our proposed recovery scheme totally eliminates the memory miss idle time and obviates the need of memory invalidation by prefetching up-to-date data before they are to be accessed in a subsequent time interval, therefore giving rise to fast crash recovery. Our coherence-centric logging and recovery is readily applicable to arrive at recoverable home-based 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. 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, January 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] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proc. of the 13th ACM Symp. on Operating Systems Principles (SOSP'91)*, pages 152–164, October 1991.

[6] M. Costa $et\ al.$ Lightweight Logging for Lazy Release Consistent Distributed Shared Memory. In *Proc. of the 2nd USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pages 59–73, October 1996.

[7] A. L. Cox, E. de Lara, Y. C. Hu, and W. Zwaenepoel. A Performance Comparison of Homeless and Home-Based Lazy Release Consistency Protocols in Software Shared Memory. In *Proc. of the 5th IEEE Symp. on High-Performance Computer Architecture (HPCA-5)*, pages 279–283, January 1999.

[8] 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.

[9] 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.

[10] P. J. Keleher. Update Protocols and Iterative Scientific Applications. In *Proc. of the 12th Int'l Parallel Processing Symp. (IPPS'98)*, pages 675–681, March 1998.

[11] 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.

[12] 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.

[13] G. G. Richard III and M. Singhal. Using Logging and Asynchronous Checkpointing to Implement Recoverable Distributed Shared Memory. In *Proc. of the 12th IEEE Symp. on Reliable Distributed Systems*, pages 58–67, Oct. 1993.

[14] R. Samanta, A. Bilas, L. Iftode, and J. P. Singh. Home-Based SVM Protocols for SMP Clusters: Design and Performance. In *Proc. of the 4th IEEE Symp. on High-Performance Computer Architecture*, pages 113–124, February 1998.

[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] Y. Zhou, L. Iftode, and K. Li. Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory Systems. In *Proc. of the 2nd USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pages 75–88, October 1996.