Coherence-Based Coordinated Checkpointing for Software Distributed Shared Memory Systems

Angkul Kongmunvattana, Santipong Tanchatchawal, and Nian-Feng Tzeng

Center for Advanced Computer Studies
University of Louisiana at Lafayette

Abstract

Fault-tolerant techniques that can cope with system failures in software distributed shared memory (SDSM) are essential for creating productive and highly available parallel computing environments on clusters of workstations. In this paper, we propose a new, efficient coordinated checkpointing technique, called coherence-based coordinated checkpointing (CCC), for SDSM. Our CCC minimizes both the checkpointing overhead during failure-free execution and the cost of recovery from failures by leveraging existing coherence information maintained by SDSM. In the presence of system failures, it allows SDSM to recover from the most recent checkpoint, saving the re-computation time.

We have performed experiments on a cluster of eight Sun Ultra-5 workstations, comparing our CCC technique against both simple coordinated checkpointing (SCC) and incremental coordinated checkpointing (ICC) techniques by actually implementing these techniques in TreadMarks, a state-of-the-art SDSM system. The experimental results demonstrate that our CCC technique consistently outperforms both SCC and ICC techniques. In particular, our technique increases the execution time slightly by 0.5% to 4% for a 2-minute checkpointing interval during failure-free execution, while SCC and ICC techniques result in the execution time overhead of 4% to 100% and 3% to 64%, respectively, for the same checkpointing interval.

1. Introduction

Software distributed shared memory (SDSM) has become one practical technology for creating productive parallel computing environments on workstation clusters. It simplifies parallel programming tasks by creating a globally coherent memory address space across interconnected workstations. 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. Performance of SDSM 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. Consequently, for good performance, a recent SDSM system often adopts a relaxed memory consistency model [1], which makes various optimizations possible. As the system size grows, however, the probability of SDSM system failures increases. This vulnerability is unacceptable, especially for long-running applications and high-availability situations. Fault-tolerant techniques are therefore essential for achieving recoverable SDSM systems.

To cope with system failures, several checkpointing and message logging techniques for SDSM have been proposed [5, 7, 9, 13, 15, 16, 17, 18, 20, 24]. Coordinated checkpointing is an effective technique for crash recovery support in SDSM. It creates a checkpoint during the synchronization process where a globally consistent state of execution is established to save all computation that SDSM has performed, up till prior to checkpoint creation. Message logging was originally proposed for solving unbounded rollback-recovery associated with an independent checkpointing technique by logging messages exchanged among processors. Most of previous work on coordinated checkpointing for SDSM has assumed a sequential consistency model and a directory-based memory coherence enforcement protocol. Although it is possible to apply those earlier coordinated checkpointing techniques to recent SDSM systems under a relaxed memory consistency model, our experimental results indicate that those earlier techniques are undesirable because they incur excessive overheads and keep lots of data unnecessary for SDSM recovery.

Coherence-based coordinated checkpointing (CCC) is a new, efficient technique we propose in this paper for providing SDSM with crash recovery support, aimed at reducing both the checkpointing overheads and the cost of recovery from failures. Our CCC technique relies on memory coherence enforcement information to select the data truly necessary for correct crash recovery, minimizing the checkpoint size. To assess the impacts of a checkpoint creation overhead on SDSM performance, we have implemented our CCC technique, the simple coordinated checkpointing

(SCC) technique, and the incremental coordinated checkpointing (ICC) technique separately in TreadMarks, a state-of-the-art SDSM. Experimental results on eight Sun Ultra-5 workstations using four parallel application programs under the checkpointing interval of 2 minutes show that our CCC technique causes very low overhead during failure-free execution, ranging from 0.5% to 4% of the execution time when checkpointing is disabled.

This paper is organized into six sections. Section 2 provides basic background pertinent to this paper, including related work. Our coherence-based coordinated checkpointing technique is introduced in Section 3. In Section 4, we describe our experimental environment, including the SDSM system, the hardware platform, and the application benchmarks. Section 5 presents the experimental performance results, and Section 6 concludes the paper.

2. Pertinent Background

2.1 SDSM

SDSM creates a shared memory image on top of parallel systems with physically distributed processing nodes, such as distributed memory multicomputers and workstation clusters. 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 [12] (i.e., one of the least restrictive relaxed memory consistency models) is preferred 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 a shared variable, and *release* for a process to relinquish an acquired variable, permitting another process to acquire the variable.

Lazy release consistency (LRC) [14] 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 a lock grant message sent to the process at which an 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. As SDSM relies on virtual memory traps, its memory coherence is maintained at the OS page granularity. The multiple-writer (MW) protocol and lazy diff creation [6] are designed to alleviate false sharing and to reduce the data transfer amount of SDSM when fine-grain application programs are run. MW allows multiple processing nodes to modify different portions of the same shared memory page at the same time, whereas lazy diff creation permits each processor to send only a summary of modifications for updating an invalid shared memory page, instead of sending the whole page. Our experiments are conducted on TreadMarks, an LRC-based SDSM which employs both the MW protocol and the lazy diff creation technique.

2.2 Related Work

Checkpointing has its root in message-passing systems and can be classified into two distinct approaches: coordinated and independent [10]. Coordinated checkpointing [8, 11, 19, 22] requires all processes to synchronize their checkpoints so that the state contained in the union of all checkpoints is a consistent global state. Upon a failure, restoring the system to a globally consistent state of execution is achieved by gathering data from the most recent checkpoint, and therefore, checkpoints older than the most recent one can be discarded. Its simplicity has attracted several researchers to study techniques for reducing both the checkpoint size and the overhead of checkpoint coordination. In particular, Chandy and Lamport presented the concept of a globally consistent state of execution in distributed systems [8]. Subsequent investigation into distributed snapshots resulted in algorithms for reducing the number of messages required for synchronization under coordinated checkpointing [19, 22]. Elnozahy $et\ al.$ evaluated two techniques for tolerating checkpoint latency and for reducing the checkpoint size [11]. Independent checkpointing [4, 23, 25] allows each process to create a checkpoint individually at any time. It does not guarantee bounded rollback-recovery, so previous checkpoints cannot be discarded and garbage collection is necessary to limit the number of checkpoints stored. We focus on coordinated checkpointing in this study.

Most of the previous work on coordinated checkpointing for SDSM is targeted at sequentially consistent SDSM systems, dubbed SC-based SDSM, with directory-based memory coherence enforcement employed. Specifically, Carter $et\ al.$ [7] estimated coordinated checkpointing overhead of SDSM using results published separately [11]. Janakiraman and Tamir [13] demonstrated that some coordinated checkpointing overheads due to message dependency may be eliminated. Cabillic $et\ al.$ [5] incorporated coordinated checkpointing in barrier synchronization of SC-based SDSM to reduce its checkpointing overhead due to coordination. Recently, Costa $et\ al.$ [9] partially implemented

coordinated checkpointing in LRC-based SDSM, but they only presented the results of an independent checkpointing technique on a cluster of four workstations. No prior work has ever completely implemented coordinated checkpointing in an LRC-based SDSM system to assess its overhead and performance. In this study, we have introduced an efficient coherence-based coordinated checkpointing technique for LRC-based SDSM, incorporating it in TreadMarks with experimental results gathered and demonstrated.

3. Proposed Technique

The idea behind checkpointing is to store a state of execution on reliable storage periodically as a checkpoint during failure-free execution, and then, to use it for reconstructing a state of execution at the beginning of a crash recovery process, saving re-computation time during recovery. In particular, we make use of transparent checkpointing, where SDSM automatically creates a checkpoint and no modification to application codes is required.

3.1 Concept and Description

Our coherence-based coordinated checkpointing (CCC) is tailored specifically for a state-of-the-art SDSM system developed under the notions of lazy release consistency, multiple-writer, and write-invalidate protocols. It relies on coherence-related information to select data truly necessary for correct recovery. Specifically, it uses write-notice records to select a summary of modifications needed for updating shared memory pages in the most recent checkpoint and follows the SDSM internal memory management routine to identify portions of a local data segment actually required to be added to a new checkpoint. Other techniques, such as simple coordinated checkpointing (SCC) and incremental coordinated checkpointing (ICC), create much larger checkpoints since SCC flushes an entire memory address space of each process as a checkpoint, and ICC tracks only the shared memory data segment where it has a permission to write-protect memory pages for detecting the modifications. Next, we explain different segments of the memory address space in each process of SDSM in details.

The memory address space of each process in SDSM can be classified into four categories, as follows: (i) a data segment, (ii) a shared memory segment, (iii) a stack segment, and (iv) a text segment. Both stack and text segments are very small in comparison to other segments and have little impact on checkpointing overheads. A data segment consists of the memory address space allocated through memory allocation routines, such as malloc() and sbrk(). SDSM uses this data segment for its internal data structures and for all local variables of the application program. There are four types of internal data structures in SDSM, including: (i) diffs pools, (ii) write-notice pools, (iii) time-interval pools, and (iv) twins pools. Diffs pools are allocated for storing a summary of modifications made to each shared memory page during every execution interval. Write-notice pools are used for bookkeeping a list of dirty pages, and the list is updated at such synchronization points as locks and barriers. Time-interval pools keep the interval time stamps for events ordering in SDSM. Twins pools are available for each shared memory page to create its pristine copy before performing any write operation, and therefore, a summary of modifications can be obtained later by comparing a shared memory page with its twin. Since all these data structures and local variables are resided in a data segment, their memory protection modes are managed by the operating systems, making it impossible to track the changes of these data through a write-protection mechanism. Consequently, the incremental coordinated checkpointing (ICC) technique, which relies on adding the modifications to a previously created checkpoint, has to flush the entire data segment at every checkpoint creation; so does the simple coordinated checkpointing (SCC) technique. In contrast, our CCC technique consults the write-notice, and then, flushes only local variables and diffs to their corresponding shared memory pages in the most recent checkpoint, reducing checkpointing overheads.

A shared memory segment is allocated by SDSM using a memory mapping operation, i.e., mmap(). SDSM utilizes this segment solely for shared memory data and manages it at the OS page granularity. Consequently, the memory protection mode is regulated on per-page basis and classified according to the existence of a pristine copy of each shared memory page (i.e., twin) and to the shared memory page status, such as (i) private, (ii) valid, (iii) invalid, and (iv) empty. A shared memory page under the private status is set to the read-/write-able mode, and so is a valid page with a twin; a valid page without a twin is set to the read-only mode, whereas the invalid and empty pages are set to the inaccessible mode. At the beginning of execution, SDSM sets every shared memory page on all processors to the read-only mode using an mprotect() function, and therefore, the modifications of each shared memory page can be tracked. Under the SCC technique, each process simply flushes the entire shared memory segment as part of its checkpoint, while ICC reduces the checkpoint size by selecting only shared memory pages that have been modified since the last checkpoint. Our CCC technique does not flush any part of the shared memory segment after the first checkpoint has been performed, since diffs are sufficient for updating the shared memory pages from the most recent checkpoint.

While it is possible to migrate the data resided in the (local) data segment into the shared memory segment (allowing ICC to track an entire data set using a write-protection mechanism), doing so has several drawbacks, as follows: (i) requiring modifications to the application source codes,

Barrier

```
if (barrier_manager)
   wait for all other processes
        to check-in;
   invalidate dirty pages;
   if (the_first_checkpoint)
     checkpoint the shared memory segment;
   endif
   save the stack environment;
   checkpoint the data segment;
   checkpoint the stack segment;
   send barrier release message
        piggybacked with list of
        dirty pages to all other
        processes;
else
   send check-in message to
        barrier manager;
   wait for barrier release message;
   invalidate dirty pages;
   if (the_first_checkpoint)
     checkpoint the shared memory segment;
   endif
   save the stack environment;
   checkpoint the data segment;
   checkpoint the stack segment;
endif
```

Checkpoint Restoration

```
restore the shared memory segment;
restore the text and data segment;
restore the stack segment;
restore the stack environment;
```

Figure 1. Pseudo Code for Implementing Our Coherence-Based Coordinated Checkpointing Technique in SDSM.

(ii) exhibiting degradation of memory management efficiency as SDSM cannot allocate/deallocate memory space (for diffs, twins, etc.) on demands, (iii) increasing the complexity of memory coherence enforcement in SDSM, and (iv) hindering SDSM performance. Therefore, it is not a viable approach.

In summary, SCC creates each checkpoint by flushing almost an entire address space of every process to stable storage at the synchronization point, whereas the ICC and our CCC techniques attempt to reduce the checkpoint size respectively by tracking the modifications of shared memory pages and by leveraging existing coherence-related information in SDSM. Our experimental results on a cluster of eight workstations using TreadMarks with four parallel benchmark programs reveal that the SCC technique indeed writes a large-sized checkpoint and incurs long latency consistently, hampering SDSM performance. The ICC technique, on the contrary, reduces the size of shared memory data necessary to be added to the checkpoint, lowering the checkpointing overhead. Our CCC technique further decreases the checkpoint size significantly, making checkpointing in SDSM more affordable and attractive.

3.2 Implementation

Figure 1 shows the pseudo code of our coherence-based coordinated checkpointing technique for implementing recoverable SDSM systems.

3.2.1 Checkpoint Creation Process

At a barrier, the barrier manager waits for all other processes to check-in. When all check-in messages have arrived, the barrier manager invalidates its copy of shared memory pages according to the write-notice (piggybacked with check-in messages) received, and then, performs a checkpoint creation as follows:

- **Checkpoint the shared memory segment** by first determining whether this is the first checkpoint; if so, calculating the range of shared memory address space actually used by the application, and then, bookkeeping the status of each shared memory page resided in that range. Next, setting the memory protection mode of all shared memory pages to the read-only mode, and checkpointing the contents of shared memory pages to stable storage before restoring the correct memory protection mode of all shared memory pages according to their page status kept earlier. If the checkpoint has already been performed at least once, then there is no need to checkpoint the shared memory segment, but only a summary of modifications (which resides in the data segment) is sufficient for updating the contents of shared memory pages in the most recent checkpoint.
- **Save the stack environment** by using setjmp() instruction.
- **Checkpoint the data segment** by first consulting the write-notice records as to whether there are modified shared memory pages since the last checkpoint; if so, applying those diffs to their corresponding shared memory pages in the stored checkpoint, and then, flushing the entire address space of the local variables as a part of the new checkpoint.
- **Checkpoint the stack segment** by first discovering its starting address and its size, and then, checkpointing its content to the stable storage as a checkpoint.

After a checkpoint has been created, the barrier manager sends barrier release messages to all other processes.

Each barrier release message is piggybacked with a write-invalidation notice specifically compiled for each process. For any process other than the barrier manager, upon arriving at a barrier, it sends out a check-in message to the barrier manager and waits for a barrier release message. When a barrier release message has arrived, it invalidates its copy of shared memory pages according to the write-notice (piggybacked with a barrier release message), and then, performs checkpoint creation using the same procedure as shown above.

3.2.2 Crash Recovery Process

The crash recovery process for our CCC technique is straightforward. The failed process starts the recovery procedure by performing a checkpoint restoration, as follows:

- **Restore the shared memory segment** by first reading a starting address and the size of the segment from the checkpoint, and then, setting an entire shared memory segment to the read-only mode. Next, it reads a record (i.e., page address, page status, and page contents) of each shared memory page from the checkpoint, changes the memory protection mode of that page to the read-/write-able mode, restores the page contents, and changes memory protection mode of that page according to its page status. These steps are repeated till the end of the checkpoint.

- **Restore the text and the (local) data segment** by first reading the starting and ending addresses of each segment from the checkpoint, and then, using them to set the size of allocated space for each segment. Next, it copies the contents of both segments to the corresponding addresses.

- **Restore the stack segment** by first reading the top of stack address from the checkpoint, and then, recursively calling the function that compares the top of stack address obtained from the checkpoint with current top of stack address until they are matched. Next, it reads the starting address and the size of the stack segment and uses them to restore the stack contents.

- **Restore the stack environment** by using the longjmp() instruction.

As the checkpoint is created at a barrier synchronization point, the longjmp() instruction returns the recovery process to the barrier routine. The recovery process then proceeds beyond the barrier as if the failure has never occured. Next, we briefly describe our experimental platform, software systems, and application benchmarks.

4. Experimental Setup

Coordinated checkpointing techniques are built on top of TreadMarks [2] for our experiments. TreadMarks implements the LRC protocol [14] with several optimization techniques, including multiple-writer and lazy diff creation [6], to lower the amount of data movement and interprocess communication. It adopts the UDP/IP protocol for message exchange, uses the SIGIO signal for delivering request messages, and relies on a virtual memory trap (SIGSEGV) for invoking the memory coherence enforcement mechanism. All checkpoints and message logs are stored in non-volatile storage (i.e., local disk) for recovery. 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. We allocated 2 GB of each local disk for virtual memory paging (i.e., swap space). The size of virtual memory pages is 8 KB. The network for connecting these machines is a full-duplex fast Ethernet switch (100 Mbps).

Program	Problem Size	Synchronization	Exec. Time (sec.)
3D-FFT	$2^5 \times 2^5 \times 2^5$	barriers	10680
MG	$64 \times 64 \times 64$	barriers	2100
SOR	3000×3000	barriers	8040
Water	1331 molecules	locks and barriers	5520

Table 1. Applications' Characteristics.

In this study, we employed four parallel applications from different sources as benchmark programs, among which 3D-FFT and MG are from the NAS benchmark suite [3], Water from the SPLASH benchmark suite [21], and SOR from the TreadMarks distribution. These applications have been selected in several previous studies of SDSM (e.g., [9, 17, 24]). Table 1 lists application characteristics, including the problem sizes, the synchronization types, and the execution times on a cluster of eight workstations with checkpointing and message logging disabled.

5. Experimental Results

5.1 Overhead Per Checkpoint

Table 2 presents overheads of coordinated checkpointing (per checkpoint, after the first checkpoint) under the simple coordinated checkpointing (SCC) technique, the incremental coordinated checkpointing (ICC) technique, and our coherence-based coordinated checkpointing (CCC) technique during failure-free execution. For each application, it lists a breakdown of the checkpoint data components and the checkpoint creation time. Every checkpoint consists of a shared data segment, a (local) data segment, and a stack segment. The checkpoint creation time corresponds to the time used by each process to create a checkpoint in coordination. Note that SDSM creates a checkpoint during its

execution when virtual memory paging and swapping continue to take place, so our results cannot be compared with those data gathered by simulating checkpoint creation on an idle disk.

Coordinated checkpointing techniques provide SDSM with crash recovery capability, but they involve different performance penalty amounts. From Table 2, it is apparent that our CCC consistently results in lower failure-free overhead than other techniques (which create larger checkpoints). The checkpoint size is smaller with our CCC than SCC, by as much as 73% for 3D-FFT, 78% for MG, 97% for SOR, and 81% for Water. Our CCC also creates a smaller checkpoint than ICC, by as much as 72% for 3D-FFT, 75% for MG, 96% for SOR, and 80% for Water. This results directly from taking advantages of coherence information existing in SDSM, allowing our CCC technique to identify data truly necessary for correct recovery. According to coherence information, our CCC technique can separate the address space of diff pools from that of local variables (both resides in the local data segment), and therefore, it can manage to flush diffs and local variables to update the shared memory pages and local variables, respectively, only from the latest checkpoint. Without such coherence information, ICC cannot incrementally checkpoint this portion of process address space because it has no permission to write-protect memory pages for detecting the changes made to a (local) data segment, and thus a whole (local) data segment is kept for a checkpoint, leading to an unnecessarily large checkpoint size. As a result, the checkpoint creation time is much smaller with our CCC technique (after the first checkpoint) than with SCC or ICC. Specifically, our CCC reduces the checkpointing time overhead from 86% to 98% when compared with SCC, and from 83% to 96% when compared with ICC.

Further examination of our experimental results reveals that the overhead of checkpoint creation corresponds to not only the checkpoint size but also the placement of data used for creating a checkpoint. One may easily notice that the checkpoint sizes of 3D-FFT in Table 2(a) and Water in Table 2(d) are not drastically different, but the checkpoint creation times of these applications differ largely. This is caused by the fact that most of the (local) data segment dedicated for the diff pools in 3D-FFT is used for storing local diffs, whereas the diff pools in Water are mainly designated for remote diffs. While we have to gather both local and remote diffs during checkpoint creation, local diffs cause much higher overhead than remote ones. This is because local diffs reside on disk and need to be paged-in for access by a virtual memory mechanism, whereas remote diffs are copied directly from the network to local memory. This side-effect further exaggerates the checkpointing time overhead of the SCC technique under 3D-FFT since the paging of local diffs interferes with the placement of

Checkpointing Technique	Checkpoint Size				Checkpoint Creation Time (Seconds)
	Shared Data (MB)	Data Segment (MB)	Stack Segment (KB)	Total (MB)	
SCC	2.31	50.63	2.42	52.94	109.81
ICC	0.14	50.63	2.42	50.77	76.30
CCC	0.00	14.24	2.42	14.24	5.06

(a) 3D-FFT

Checkpointing Technique	Checkpoint Size				Checkpoint Creation Time (Seconds)
	Shared Data (MB)	Data Segment (MB)	Stack Segment (KB)	Total (MB)	
SCC	5.20	27.35	2.74	32.55	4.40
ICC	0.68	27.35	2.74	28.03	3.57
CCC	0.00	7.08	2.74	7.08	0.60

(c) MG

Checkpointing Technique	Checkpoint Size				Checkpoint Creation Time (Seconds)
	Shared Data (MB)	Data Segment (MB)	Stack Segment (KB)	Total (MB)	
SCC	36.09	40.20	2.43	76.29	120.55
ICC	4.52	40.20	2.43	44.72	59.81
CCC	0.00	2.01	2.43	2.01	2.77

(d) SOR

Checkpointing Technique	Checkpoint Size				Checkpoint Creation Time (Seconds)
	Shared Data (MB)	Data Segment (MB)	Stack Segment (KB)	Total (MB)	
SCC	1.03	47.03	2.56	48.06	22.60
ICC	0.02	47.13	2.56	47.15	22.93
CCC	0.00	9.31	2.56	9.31	0.85

(b) Water

Table 2. Overhead Details under Different Checkpointing Techniques (per checkpoint).

shared memory pages in local memory (i.e., possibly forcing the OS to page-out some shared memory pages to make room for local diffs). Hence, checkpointing is much more complex in SDSM than in message-passing systems, and it is important to have an efficient coordinated checkpointing technique that leverages internal coherence information of SDSM, because such a technique may lower the overhead significantly during failure-free execution.

Checkpointing Technique	Checkpoint Size				Checkpoint Creation Time (Seconds)
	Shared Data (MB)	Data Segment (MB)	Stack Segment (KB)	Total (MB)	
SCC	39.56	31.46	2.78	71.02	144.50
ICC	4.32	31.46	2.78	35.78	4.95
CCC	0.00	7.23	2.78	7.23	0.88

Table 3. Overhead Details (per checkpoint) of MG using Problem Size $128 \times 128 \times 128$.

Another interesting observation is the effect of the problem size on the overhead of checkpoint creation. For example, the checkpoint creation times of MG are far smaller than those of 3D-FFT, even though their checkpoint sizes are not drastically different, according to Table 2. This is because the data set of MG (both shared and local data) is easily fit in the main memory of our testbed. Note that while our workstation has 64 MB of memory, only about

one-half of it is available to SDSM and its application, since the operating systems use anywhere from 23 MB to 40 MB, depending on the memory pressure. To illustrate the effect of the problem size on checkpointing overhead, Table 3 shows the overhead details (per checkpoint, after the first checkpoint) of MG under different coordinated checkpointing techniques using the problem size of $128 \times 128 \times 128$. It is evident from the results that when the data set needed for checkpoint creation is no longer fit into the main memory, the overhead of checkpoint creation is increased drastically. In particular, the checkpoint creation time of MG under SCC jumps from 4.40 seconds to 144.50 seconds when the problem size grows from $64 \times 64 \times 64$ to $128 \times 128 \times 128$, whereas the checkpoint creation times increase moderately under ICC and CCC, as the sets of checkpoint data still remain within the physical main memory bound.

5.2 Total Execution Time

Figure 2 illustrates the impacts of different coordinated checkpointing techniques on SDSM performance in terms of the total execution time (normalized for easy illustration) under the checkpointing interval of 2 minutes. This 2-minute assumption is very conservative since a longer checkpoint interval is generally used in practice, and thus, we actually overestimate the cost of checkpoint creation in SDSM because longer checkpoint intervals tend to decrease overheads during failure-free execution. For comparison, the total execution time of SDSM without any coordinated checkpointing incorporated serves as the performance baseline. From the results, our CCC technique leads to negligible execution time overhead, ranging from 0.5% to 4% only. This low overhead results directly from a small checkpoint size. On the contrary, SCC and ICC increase the execution time drastically on most applications, ranging from 4% to 100% for SCC and from 3% to 64% for ICC. Such high overheads associated with SCC and ICC are due to checkpointing excessive amounts of data unnecessarily for correct recovery of SDSM. Note that the checkpointing overheads of MG under SCC and ICC are unusually low since their data amounts for checkpoint creation are small and fit into the main memory.

5.3 Checkpoint Restoration Performance

Table 4 shows the cost of checkpoint restoration after system failures. The process of restoring a globally consistent state of execution of a failed process consists of four phases, as we explained in Section 3. The overheads come from not only the disk read operations but also the memory management operations, since the correct memory protection mode of each shared memory page and its contents have to be restored correctly. The experimental results indicate that the cost of recovery is small. This is due to the fact that the recovery process has an exclusive access to the disk during crash recovery, reducing disk access latency. Note that while we may lower checkpoint creation overhead by limiting information added on to the most recent checkpoint, the size of the checkpoint on disk either remains the same or is enlarged as the address space of SDSM grows.

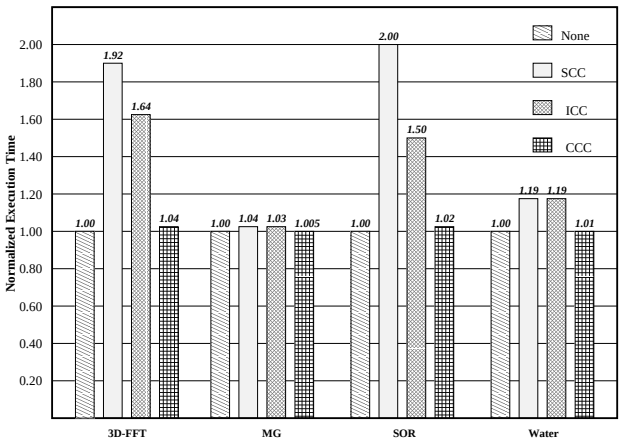

Figure 2. Impacts of Checkpointing on Execution Time (2-minute checkpoint interval).

Program	Checkpoint Size (MB)	Checkpoint Restore Time (Seconds)
3D-FFT	52.94	10.87
MG	32.55	4.73
SOR	76.29	31.18
Water	48.06	5.72

Table 4. Overhead of Checkpoint Restoration.

6. Conclusions

We have proposed an efficient coordinated checkpointing technique, called coherence-based coordinated checkpointing (CCC), for SDSM in this paper. The experimental outcomes demonstrate that our CCC technique with the checkpointing interval of 2 minutes incurs fairly low failure-free overhead, roughly 0.5% to 4% of the SDSM normal execution time, whereas the simple coordinated checkpointing and the incremental coordinated checkpointing techniques result in the execution time overhead of 4% to 100% and 3% to 64%, respectively. This is due to the small checkpoint size under our CCC technique, which leverages coherence-related information maintained by SDSM to select data truly necessary for correct recovery. Consequently, SDSM with our CCC technique incorporated can recover from system failures much faster due to a significant savings in the re-computation time. Our CCC technique makes the implementation of recoverable SDSM systems feasible and attractive.

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, January 1991.

[4] B. Bhargava and S. R. Lian. Independent Checkpointing and Concurrent Rollback Recovery for Distributed Systems - An Optimistic Approach. In *Proc. of the 7th IEEE Symp. on Reliable Distributed Systems*, pages 3–12, October 1988.

[5] G. Cabillic, G. Muller, and I. Puaut. The Performance of Consistent Checkpointing in Distributed Shared Memory Systems. In *Proc. of the 14th IEEE Symp. on Reliable Distributed Systems (SRDS-14)*, 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] J. B. Carter, A. L. Cox, S. Dwarkadas, E. N. Elnozahy, D. B. Johnson, P. J. Keleher, S. Rodrigues, W. Yu, and W. Zwaenepoel. Network Multicomputers using Recoverable Distributed Shared Memory. In *Proc. of the 38th IEEE Int'l Computer Conf. (COMPCON Spring'93)*, pages 515–523, February 1993.

[8] K. M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Trans. on Computer Systems*, 3(1):63–75, February 1985.

[9] 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.

[10] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. Technical Report CMU-CS-99-148, Carnegie Mellon University, June 1999.

[11] E. N. Elnozahy, D. B. Johnson, and W. Zwaenepoel. The Performance of Consistent Checkpointing. In *Proc. of the 11th IEEE Symp. on Reliable Distributed Systems (SRDS-11)*, pages 39–47, October 1992.

[12] 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.

[13] G. Janakiraman and Y. Tamir. Coordinated Checkpointing-Rollback Error Recovery for Distributed Shared Memory Multicomputers. In *Proc. of the 13th IEEE Symp. on Reliable Distributed Systems (SRDS-13)*, pages 42–51, October 1994.

[14] 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.

[15] 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.

[16] A. Kongmunvattana and N.-F. Tzeng. Coherence-Centric Logging and Recovery for Home-Based Software Distributed Shared Memory. In *Proc. of the 1999 Int'l Conf. on Parallel Processing (ICPP'99)*, pages 274–281, September 1999.

[17] 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.

[18] A. Kongmunvattana and N.-F. Tzeng. Logging and Recovery in Adaptive Software Distributed Shared Memory Systems. In *Proc. of the 18th IEEE Symp. on Reliable Distributed Systems (SRDS-18)*, pages 202–211, October 1999.

[19] T. H. Lai and T. H. Yang. On Distributed Snapshots. *Information Processing Letters*, 25:153–158, May 1987.

[20] 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.

[21] 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.

[22] M. Spezialetti and P. Kearns. Efficient Distributed Snapshots. In *Proc. of the 6th Int'l Conf. on Distributed Computing Systems (ICDCS-6)*, pages 382–388, May 1986.

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

[24] 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.

[25] Y.-M. Wang and W. K. Fuchs. Optimistic Message Logging for Independent Checkpointing in Message-Passing Systems. In *Proc. of the 11th IEEE Symp. on Reliable Distributed Systems (SRDS-11)*, pages 147–154, October 1992.