Integrating Coordinated Checkpointing and Recovery Mechanisms into DSM Synchronization Barriers

AZZEDINE BOUKERCHE
University of Ottawa
and
ALBA CRISTINA MAGALHAES ALVES DE MELO
University of Brasilia

Distributed shared memory (DSM) creates an abstraction of a physical shared memory that parallel programmers can access. Most recent software DSM systems provide relaxed-memory models that guarantee consistency only at synchronization operations, such as locks and barriers. As the main goal of DSM systems is to provide support for long-term computation-intensive applications, checkpointing and recovery mechanisms are highly desirable. This article presents and evaluates the integration of a coordinated checkpointing mechanism to the barrier primitive that is usually provided with many DSM systems. Our results on some popular benchmarks and a real parallel application show that the overhead introduced during the failure-free execution is often small.

Categories and Subject Descriptors: D.1.3 [**Concurrent Programming**]—*Parallel Programming*; D.4.5 [**Reliability**]—*Checkpoint/restart, Fault tolerance*

General Terms: Algorithms, Design, Performance, Reliability

Additional Key Words and Phrases: Distributed shared memory, barrier synchronization

1. INTRODUCTION

In order to make shared-memory programming possible in distributed architectures, we must create a shared-memory abstraction that parallel processes can access. This abstraction is called distributed shared memory (DSM). The first DSM systems tried to give parallel programmers the same guarantees

A preliminary version of this paper appeared at the 4*th International Workshop on Experimental and Efficient Algorithms* (*WEA 2005).*
This research was supported by Finatec/Brazil, NSERC, Canada Research Chair Programs and CFI, and OIT/Distinguished Researcher Award.
Authors' addresses: Azzedine Boukerche, School of Information Technology and Engineering, University of Ottawa, Canada; Alba C. M. A. de Melo, Department of Computer Science, University of Brasilia, Brasilia, Brazil.

they had when programming uniprocessors. It has been observed that providing such a strong memory consistency model creates a huge coherence overhead, slowing down the parallel application and bringing frequently the system into a thrashing state [Mosberger 1993]. To alleviate this problem, researchers have proposed to relax some consistency conditions, thus creating new shared-memory behaviors that are different from the traditional uniprocessor one.

In the shared-memory programming paradigm, synchronization operations must be used every time processes want to restrict the order in which memory operations should be performed. Using this fact, hybrid memory-consistency models guarantee that processors only have a consistent view of the shared memory at synchronization time. This allows a great overlapping of basic read and write memory accesses that can potentially lead to considerable performance gains. By now, the most popular hybrid memory-consistency models for DSM systems are release consistency (RC) [Gharachorloo et al. 1990] and scope consistency (ScC) [Iftode et al. 1998].

While the memory-consistency model defines when consistency should be ensured, coherence protocols define how it should be done. Most recent DSM systems use sophisticated coherence protocols that are often variations of the lazy-release consistency (LRC) protocol [Keleher et al. 1994]. LRC is a multiple-writer, homeless protocol that uses techniques, such as page twinning, page diffing, and write notices to guarantee that pages at the shared memory segment are consistent at acquire time. In homeless protocols, fetching the up-to-date version of a page usually involves sending messages to many nodes whereas, in home-based protocols, it is sufficient to fetch the up-to-date version of a page from a single node, called home.

For any system running on a distributed platform that aims to be used in large scale, fault tolerance mechanisms must be considered. Usually, fault tolerance is achieved by periodically checkpointing each process that composes the system and, in the case of a failure, recovering the system to a previous consistent state by activating the saved checkpoints.

Checkpointing can be done in a coordinated or in an uncoordinated way. Coordinated checkpointing is achieved by establishing a checkpointing session that captures a global consistent state of the execution and saves it to a stable storage. Usually, all DSM processes stop computing to take their checkpoints in a coordinated way. Rollback/recovery is quite simple and is done by activating the last set of checkpoints.

In an uncoordinated (or independent) checkpointing, there is no need to establish checkpointing sessions and all processes can take their checkpoints whenever they want. However, rollback/recovery, in this case, is unbounded and garbage collection is complex [Elnozhay, et al., 1996]. To overcome this problem, message logging is often associated with uncoordinated checkpointing.

Barrier synchronization is used in DSM systems whenever a global synchronization point needs to be established. Thus, taking checkpoints at barriers is a natural choice to implement coordinated checkpointing in DSM systems.

In this article, we propose and evaluate a coordinated checkpointing/recovery strategy that can be adapted to DSM systems that have barrier primitives. We also opted to take the actual checkpoints with an existing checkpoint library

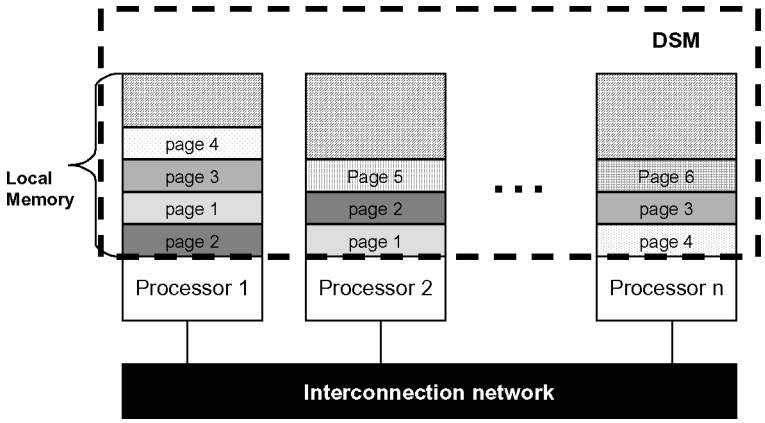

Fig. 1. The DSM Abstraction.

(ckpt [Zandy 2003]) in order to make our approach more portable and to concentrate efforts on the coordinated checkpointing strategy itself. This separation between the checkpointing mechanism and the checkpointing policy is the original part of our approach.

The remainder of this paper is organized as follows. Section 2 presents the main concepts involved in distributed shared-memory systems. Section 3 describes some characteristics of memory-coherence protocols for DSM systems. Section 4 presents an overview of checkpointing/recovery schemes for DSM systems. The JIAJIA software DSM is described in Section 5. Section 6 describes our checkpointing and recovery mechanisms. Some experimental results are discussed in Section 7. Finally, Section 8 concludes the paper and presents future work.

2. DISTRIBUTED SHARED MEMORY

DSM has received a lot of attention in the last few years since it offers the shared memory-programming paradigm in a distributed or parallel environment where no physically shared memory exists.

One way to conceive a DSM system is by the shared virtual memory (SVM) approach (Figure 1). SVM implements a single-paged, virtual address space over a network of computers. It works basically as a virtual memory system. Local references are executed exclusively by hardware. When a nonresident page is accessed, a page fault is generated and the SVM system is contacted. Instead of fetching the page from disk, as do the traditional virtual memory systems, the SVM system fetches the page from a remote node and restarts the instruction that causes the page fault trap.

In order to improve performance, DSM systems usually replicate pages. Replication creates a coherence problem that arises when one of the copies is modified. To solve this problem, one straightforward approach is to send modifications immediately to all processors and, wait for their acknowledgments before the next operation is issued [Mosberger 1993]. While maintaining strong consistency, this approach creates a huge coherence overhead. Relaxed memory

models aim to solve this coherence problem, since they no longer guarantee strong consistency at all times, allowing replicas of the same data to have, for some period of time, different values [Adve 1996]. Relaxed memory models do reduce coherence overhead but provide a programming model that is complex since, at some instants, the programmer must be conscious of replication.

Hybrid memory models are a class of relaxed memory models that postpone the propagation of shared data modifications until the next synchronization point. These models are quite successful in the sense that they permit a great overlapping of basic memory operations while still providing a reasonable programming model. Release consistency (RC) [Gharachorloo et al. 1991] and scope consistency (ScC) [Iftode et al. 1998] are the most popular memory models for software DSM systems.

In release consistency, competing accesses are called special accesses, which are divided into synchronization and nonsynchronization operations. There are two subtypes of synchronization operations: *acquire* accesses and *release* accesses. Read and write memory operations are called ordinary accesses.

Informally, in release-consistent systems, it must be guaranteed that [Gharachorloo et al. 1991] before an ordinary access performs, all previous acquire accesses must be performed; and before a release performs with respect to any other processor, all previous ordinary accesses must be performed. There is also a third condition that requires special accesses to be totally ordered, i.e., synchronization operations must be seen in the same order by all processors. Munin [Bennet et al. 1990] and TreadMarks [Keleher et al. 1994] are examples of release consistent software distributed shared memory systems. DASH [Lenoski et al. 1993] is a hardware-based release consistent DSM system.

The goal of scope consistency (ScC) is to take advantage of the association between synchronization variables and ordinary shared variables they protect. It was proposed by Iftode et al. [1998]. In scope consistency, executions are divided into consistency scopes that are defined on a per-lock basis. Scope consistency orders only synchronization and data accesses that are related to the same synchronization variable. The association between shared data and the synchronization variable that guards them is implicit and depends on program order. Informally, a system is scope consistent if [Iftode et al. 1998] (1) before a new section of a consistency scope is allowed to open at process P, any write previously performed with respect to that consistency scope must be performed with respect to P; and (2) a memory access is allowed to perform with respect to a process P only after all consistency scope sessions previously entered by P (in program order) have been successfully opened. In addition, a global synchronization point can be defined by synchronization barriers. JIAJIA [Hu, et al. 1999] and Brazos [Speight and Bennet 1998] are examples of scope-consistent software DSMs.

3. MEMORY COHERENCE PROTOCOLS FOR DSM SYSTEMS

There are two basic coherence protocols that are traditionally used to solve the cache-coherence problem: write-invalidate and write-update protocols. In general, at consistency time, a write-invalidate protocol guarantees that there

is only one up-to-date copy of the data in the system. If these data are further needed, they are fetched from the only node that has a valid copy. On the other hand, a write-update protocol allows many copies of the same data to be valid at consistency time and generally guarantees that updates will be done atomically and totally ordered. Although most of DSM systems use write-invalidate based protocols (TreadMarks[Keleher et al. 1994], ATMK[Amza et al. 1997]), some more recent DSM systems offer also write-update protocols (ADSM[Monnerat and Bianchini 1998], Brazos [Speight and Bennet 1998]).

Another characteristic that is important in a cache-coherence protocol for DSM systems is the number of simultaneous writers allowed. The most intuitive approach allows only one processor to write data at a given moment (single-writer protocol). However, as the unity of consistency is often a page, false sharing can occur when two or more processors want to access independent variables that belong to the same page. To reduce the effects of false sharing, many DSM systems use multiple-writer protocols, allowing many processors to have write access to the same page simultaneously and then merging the multiple versions of the page when a consistent view is required. Multiple writer protocols are generally implemented for LRC and scope-consistent DSM systems as follows. If a write fault occurs inside a critical section, the original page is copied to a twin before write access is granted. When the lock is released, the pages are compared to their twins and the differences between them (*diffs*) are generated. At acquire time, the lock manager sends to the acquiring process an acquire message containing the identifications of the pages that are no longer valid (*write notices*). These pages are invalidated before the application continues its execution.

There are two basic approaches used to manage the information needed to execute coherence protocols in page-based DSM systems: home-based and homeless. In home-based systems, each page is assigned to a node (*home-node*) that concentrates all modifications made to the page. Every time an up-to-date version of the page is needed, it is sufficient to contact the home node in order to fetch the page. In the homeless approach, each processor that modifies a page maintains such modifications locally. In order to obtain an up-to-date version of the page, a node must collect the modifications that are distributed all over the system. Modifications are kept by each node and garbage collection is required.

Home-based protocols do have some advantages. First, each access fault requires only communication with the home-node. Second, since modifications are eagerly applied at the home-node, there is no need to keep additional control structures, such as twin pages or diffs, after the home node is updated. However, as modifications are eagerly sent to the home node, such protocols generally require additional messages. Also, on an access fault, homeless protocols fetch only the modifications made to the page (*diffs*) while home-based protocols fetch the whole page [Hu, et al. 1999].

4. CHECKPOINTING/RECOVERY MECHANISMS FOR DSM SYSTEMS

For long run applications, fault tolerance mechanisms are very useful since, in the case of a failure, they avoid computation to be restarted from the beginning.

Usually, fault tolerance is achieved by saving periodically the system state to a non-volatile memory. In the case of a failure, the system state is read and computation can be restarted from it. The act of saving the state of a process is known as *checkpointing* and the act of restarting the computation from a checkpoint is *recovery*. In this section, we will discuss checkpointing/recovery protocols for DSM systems.

4.1 General view of Checkpointing Strategies

Checkpointing has been extensively studied in the research area of distributed systems [Elnozhay et al. 1996]. In a generic way, three types of strategies exist for checkpointing an application: coordinated, uncoordinated or communication-induced.

Coordinated checkpointing is achieved by establishing a checkpointing session that captures a global consistent state of the execution and saves it to stable storage. Usually, all processes stop regular message activity to take their checkpoints, in a coordinated way. Rollback/recovey is quite simple and is done by activating the last set of checkpoints. In an uncoordinated (or independent) checkpointing, there is no need to establish checkpointing sessions and all processes can take their checkpoints whenever they want. However, rollback/recovery in this case is unbounded and garbage collection is complex Elnozhay et al. [1996]. To overcome this problem, message logging is often associated with uncoordinated checkpointing.

Communication-induced checkpointing piggybacks dependency related information on the regular messages exchanged by the processes Elnozahy et al. [1996]. Doing that, checkpointing does not need special "checkpoint session" messages but, in some cases, more checkpoints than the necessary are taken. In a generic way, when a process receives a message, it analyzes the dependency-related information contained in it and, in some cases, takes a forced checkpointing before processing the message.

4.2 Examples of Checkpointing/Recovery Strategies for DSM Systems

In DSM systems, research in rollback/recovery is concentrated in adapting either coordinated or uncoordinated checkpointing strategies from message-passing systems to the DSM environment.

Janakiraman and Tamir [1994] proposed a coordinated checkpointing mechanism that keeps track of the DSM processes interactions by building a communication tree in order to reduce the number of processes involved in a checkpointing session. Incremental checkpoints are taken only for this subset of processes and stored to disk. Upon a failure, all processes restore the last set of checkpoints and continue execution from it. Costa et al. [1996] proposed an uncoordinated log-based checkpointing strategy that tolerates one node failure. Logs are kept at the remote memoies and checkpoints are stored in the local disks. It was designed for the Lazy Release Consistency (LRC) coherence protocol and piggybacks checkpoint related information at the protocol messages. Recovery is achieved by restoring the last set of checkpoints and using the logged information to create the pre-failure execution.

Table I. Characteristics of Seven Checkpoint/Recovery Mechanisms for DSM Systems

Mechanism	Failures Tolerated	Logging	Checkpointing Mechanism	Consistency Model
Janakiraman and Tamir [1994]	one	no	coordinated	Sequential
Costa et al. [1996]	one	yes	uncoordinated	Release
Kongmunvattana et Tzeng [1999]	multiple	yes	coordinated/ uncoordinated	Release
Kongmunvattana et al [2000]	multiple	no	coordinated	Release
Sultan et al. [2000]	one	yes	uncoordinated	Release
Badrinath and Morin [2004]	one	no	coordinated/ uncoordinated	Sequential
Katsinis and Hetch [2004]	one	no	coordinated	Sequential

A logging strategy for ADSM is proposed in Kongmunvattana and Tzeng [1999]. Coherence-related information is logged according to the coherence protocol, which is in use. The protocols considered are multiple-writer/write-invalidate and single-writer/write-update. Coordinated and uncoordinated checkpointing can be used. The use of coordinated checkpointing with logging can be justified since, in this case, the size of the logs can be reduced.

A coordinated checkpointing strategy for TreadMarks is presented in Kongmunvattana et al. [2000]. Coherence-related information contained in write-notices are used to decide which data really need to be saved in the checkpoint. Checkpoint is taken on barrier operations, in a transparent way. Recovery is achieved by activating the last set of checkpoints. An uncoordinated checkpointing strategy for home-based release consistent DSM systems is presented by Sultan et al. [2000]. This approach tolerates one node failure. Logs of DSM operations are maintained in remote memories for further use in the recovery mechanism, which will restore the checkpoint of the faulty process, using the logged information to replay the events. When a checkpoint is taken, the logs maintained in local memory are flushed to disk.

In Badrinath and Morin [2004], the impact of locks and barriers in sequential consistent DSM systems is analyzed. Kernel-level support is used to keep track of data dependencies in either coordinated and uncoordinated checkpointing.

Katsinis et. al. [2004] propose that fault-tolerant DSM systems should be supported by a broadcast-based architecture. Consistent incremental checkpoints are saved into the memories of remote nodes by a distributed synchronization barrier mechanism, which is supported by a broadcast architecture (SOME-Bus).

Most of the approaches analyzed in this section (Table I) tolerate only one node failure and rely on lazy release consistency. We can observe in Table I that there is no consensus about which checkpointing strategy to use. All the coordinated checkpointing approaches that were analyzed restore the last set of checkpoints from all processes to recover execution. Using uncoordinated checkpointing in conjunction with logs does not require this, and only the checkpoint of the faulty process is restored.

In the majority of the approaches summarized in Table I, coherence-related information is used to decide which information will be included in the

checkpoint. Also, most of these works deployed their own checkpoint libraries. Modifications made to the operating system kernel must be often taken into account by the checkpoint library and this leads to reduced portability. Modifications into the kernel itself are made in Badrinath and Morin [2004]. Katsinis and Hetch [2004] rely on a specific broadcast architecture to achieve fault tolerance. Like Janakiraman and Tamir [1994], Kongmunvattana et al. [2000], and Katsinis and Hetch [2004], we opted to use coordinated checkpointing. However, unlike them, we integrated an existing checkpointing library to our checkpointing mechanism. Note that we did not use coherence-related information to reduce the size of the checkpoint file since our approach is designed to work in any DSM system that provides barriers as synchronization mechanisms.

5. THE JIAJIA SOFTWARE DSM SYSTEM

JIAJIA is a software DSM system proposed by Hu et al. [1999]. It implements the scope-consistency memory model with a write-invalidate multiple-writer home-based protocol.

In JIAJIA, the shared memory is distributed among the nodes in a NUMA-architecture basis. Each shared page has a home node. A page is always present in its home node and it is also copied to remote nodes on an access fault. There is a fixed number of remote pages that can be placed at the memory of a remote node. When this part of memory is full, a replacement algorithm is executed.

Scope consistency is a memory model that requires that consistency must be guaranteed when a process acquires a lock or when it reaches a synchronization barrier (Section 2). In the first case, consistency is maintained in a per-lock basis, i.e., only the shared variables that were modified on the critical section guarded by lock l are guaranteed to be updated when a process acquires lock l.

On a synchronization barrier, however, consistency is globally maintained and all processes are guaranteed to see all past modifications to the shared data [Iftode 1998].

In order to implement scope consistency, JIAJIA statically assigns each lock to a lock manager. The functions that implement lock acquire, lock release, and synchronization barrier in JIAJIA are jia_lock, jia_unlock and jia_barrier, respectively [Shi 1999].

On a release access, the releaser sends all modifications performed inside the critical section to the home node of each modified page. The home node applies all modifications to its own copy and sends an acknowledgment back to the releaser. When all of the acknowledgments arrive, the releaser will send a message containing the numbers of the pages modified inside the critical section (i.e., write notices) to the lock manager.

On an acquire access, the acquirer sends an ACQ message to the lock manager. When the lock manager decides that the lock can be granted to the acquirer, it responds with a lock-granting message that contains all write notices associated with that lock. Upon receiving this message, the acquirer invalidates all pages that have associated write notices, since their contents are no longer

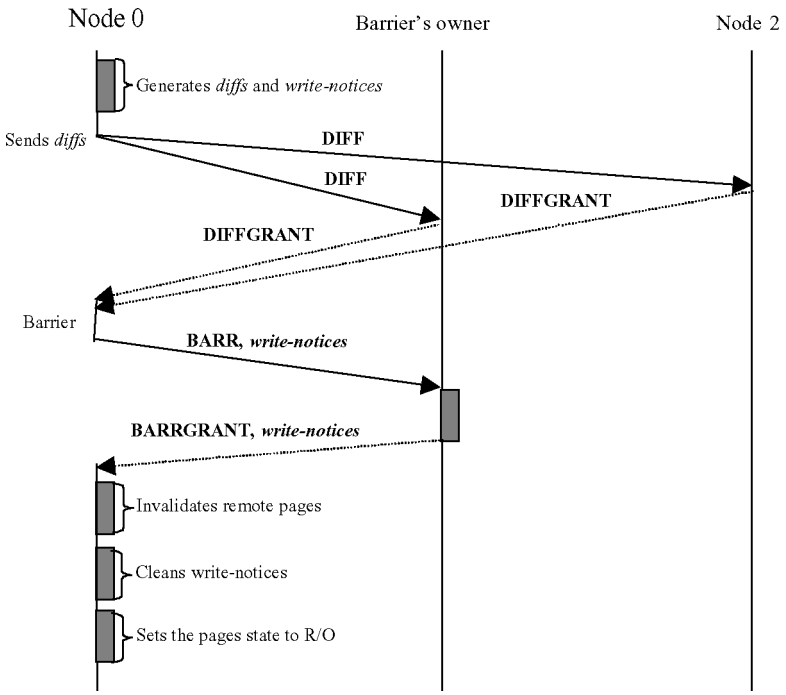

Fig. 2. Barrier synchronization in the JIAJIA software DSM system.

valid. Figure 2 illustrates the barrier operation in JIAJIA. For simplicity, the whole process was only represented for node 0.

On a barrier access, the arriving process generates the diffs of all pages that were modified since the last barrier access. Then, it sends the diffs to the respective home nodes. The home nodes receive the diffs, apply the modifications, and send an acknowledgment to the arriving process. The arriving process then sends a BARR message containing the write notices of all modified pages to the owner of the barrier.

When all processes arrive at the barrier, the owner of the barrier sends back a BARRGRANT message to each process, containing the write notices of all pages modified since the last barrier. Upon receiving this message, the processor invalidates the pages contained in the write notices and continues the execution.

6. PROPOSED CHECKPOINTING/RECOVERY MECHANISMS

6.1 Design Choices

Some assumptions were made when designing our mechanisms. First, the communication network and the stable storage used to store the checkpoints is assumed to be fault-free. Second, only transient faults will be treated. Permanent faults are not supported. Third, processes fail in a fail-stop mode. More complex failures such as byzantine failures are not considered. Fourth, failures that occur during the recovery process are not supported.

The first decision made in the design of our checkpoint/recovery mechanism is whether to use coordinated or uncoordinated checkpointing. Although coordinated checkpointing can introduce nonnegligible overheads in failure-free executions, the overheads that can be introduced by uncoordinated strategies because of complex garbage collection schemes and log management are often very high. Besides, uncoordinated checkpointing makes the recovery process more complex. Thus, we opted to design a coordinated checkpointing scheme, and to overcome the possible problem of considerable overheads introduced to failure-free executions, special care was taken not to increase the number of messages exchanged by the nodes because of checkpointing.

The second design choice concerned the checkpointing mechanism itself. We opted to use an existing checkpointing library, since we wanted to concentrate our efforts on the coordinated checkpointing strategy. We also we wanted to decouple the checkpoint strategy from the checkpointing procedure in order to make it easier to use different libraries for different operating systems, while maintaining the same checkpointing strategy.

In order to choose an appropriate checkpointing library, we evaluated *Libckpt* [Plank et al. 1995], *Libckp* [Wang et al. 1995], and *ckpt* [Zandy 2003]. The following characteristics were analyzed. First, solutions that required modifications in the operating system kernel were discarded, since these modifications introduce portability problems. Libraries that ran always in user-level with no kernel modifications were preferred. Second, open source libraries were preferred. Third, and more important, the recovery mechanism provided by the library should work for general Unix processes and, specially, for Unix processes that use JIAJIA primitives.

Of those, only *ckpt* presented these three characteristics, which is the reason why it was chosen. However, *ckpt* does have some limitations [Zandy 2003]. First, only integral checkpointing is done, i.e., incremental checkpointing is not implemented. Second, checkpoint files are always written to disk and that precludes the use of remote memories as stable storage. Third, the following resources are not saved to the checkpoint file [Zandy 2003]: open files, network connections, Unix interprocess communication (IPC) mechanisms, thread information, and process status.

6.2 Design of the Checkpointing Mechanism

Our coordinated checkpointing mechanism is integrated to the barrier synchronization primitive and requires no additional internode communication to take checkpoints. In DSM systems that implement relaxed memory models such as lazy release consistency and scope consistency, the barrier is the only execution point that captures a consistent global state of the execution. Thus, it is the natural choice for the integration of a coordinated chekpointing mechanism. In this case, there is no need for the checkpointng mechanism to establish a consistent global state, since it is done naturally by the barrier primitive.

The primitive *jia_barrier* illustrated in Figure 2 was modified to include our checkpointing strategy. When a process receives a BARRGRANT message, it knows that all the other processes have already reached the barrier and that

the global consistent state is attained. At this moment, the following actions are taken:

1. the process disables interruptions to guarantee that no messages will be treated while the checkpointing is underway;
2. integral checkpointing is made to the local disk using the command *checkpointHere*, provided by the *ckpt* library. If there is already a checkpoint for this process on the disk, it is replaced by this last one; and
3. interruptions are enabled.

After that, the *jia_barrier* proceeds as in the original implementation (Figure 2): remote pages are invalidated, write-notices are cleaned, and the pages state is set to read/only.

The programmer has the choice to enable/disable the checkpointing mechanism whenever he/she wants. The primitives *jia_config(CKPTREC,ON)* and *jia_config(CKPTREC,OFF)* are provided and they activate/deactivate the checkpointing mechanism at the subsequent barrier primitives. The primitive *jia_barrier_ckpt* is also provided to force checkpointing without considering the value of the variable CKPTREC.

6.3 Design of the Rollback/Recovery Mechanism

Since we designed a coordinated checkpointing strategy, our rollback/recovery mechanism allows the execution to be continued from the last set of saved checkpoints. The whole rollback/recovery mechanism is illustrated in Figure 3, where P0 is the node from where the DSM application was initially launched.

The program *jia_restart <application>* is offered to provide this facility. When the command *jia_restart* is executed at processor P0, the following executable files are copied to the remote machines: *jia_restart, <application>*, and *restart*. The library *librestart.so* and some configuration files are also copied at this point. After that, the remote execution command *rexec "jia_restart <application>"* is issued to all nodes that were running the DSM application when the failure occurred.

When this command starts execution on the remote nodes, it first executes the procedure *initmem* that does the mapping of the DSM area to the memory assigned to the process. After that, the checkpoint file is read from the local disk and the process state is built from this file by the *ckpt* library (program *restart*). Since the *ckpt* library did not save some data that are needed for a JIAJIA process to restart correctly, additional procedures (*initsyn* and *initcomm*) were included at this point.

The procedure *initsyn* is needed to allocate memory for the data structures that represent the locks and barriers. Finally, the procedure *initcomm* allocates and reactivates the sockets that were used by the process, which were not saved by *ckpt*, as explained in Section 6.1. Since the restored JIAJIA processes resume execution from the last saved barrier, there is no need for synchronization at the end of this protocol.

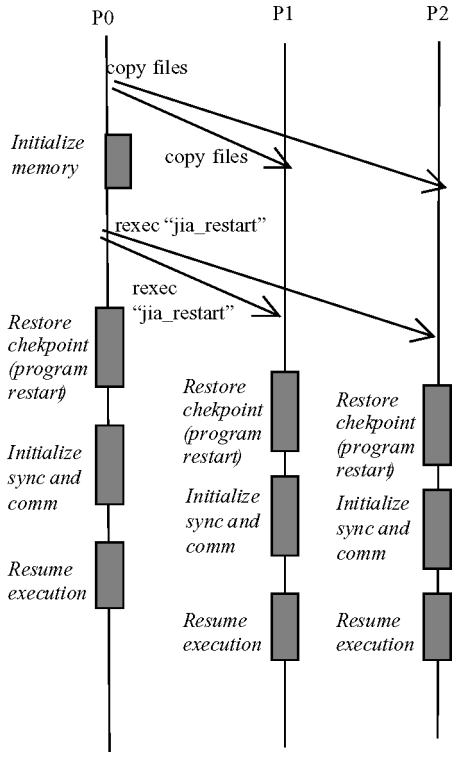

Fig. 3. Proposed rollback/recovery protocol.

7. EXPERIMENTAL RESULTS

To evaluate our mechanism, we ran our experiments on a dedicated cluster of eight 550-MHz, Pentium II with 160 MB RAM connected by a 100 Mbps Ethernet switch. The JIAJIA software DSM system v.2.1 ran on top of Debian Linux 2.2.5. The mechanisms described in Section 6 were implemented in C and integrated to the software DSM JIAJIA. In our experiments, we have selected four popular parallel benchmarks and a real parallel application, i.e., EP from NAS parallel benchmark Bailey et al. [1991], TSP from TreadMarks benchmarks Lu et al. [1997], MM, which is a matrix multiplication program that uses the inner product algorithm, and MMA, which is also an inner-product matrix multiplication, but each value is calculated with additions.

A real parallel application that aligns long DNA sequences (Genome) was also evaluated. This application uses a variant of the algorithm proposed by Smith and Waterman [1981] and has time complexity $O(n^2)$ where n is the size of the sequences. More details about this application can be found in Melo et al. [2003].

The sizes of the problem and the synchronization primitives used by these applications are shown in Table II.

The execution times and speedups of these applications are shown in Figure 4. It must be noted that the Genome application parameters were set in

Table II. Characteristics of the Applications

Program	Problem size	Synchronization
EP	2^{28}	Locks and Barriers
TSP	19	Locks and Barriers
MM1408	1408 × 1408	Barriers
MMA1408	1408 × 1408	Barriers
Genome398	398 × 398	Locks and Barriers
Genome782	782 × 782	Locks and Barriers

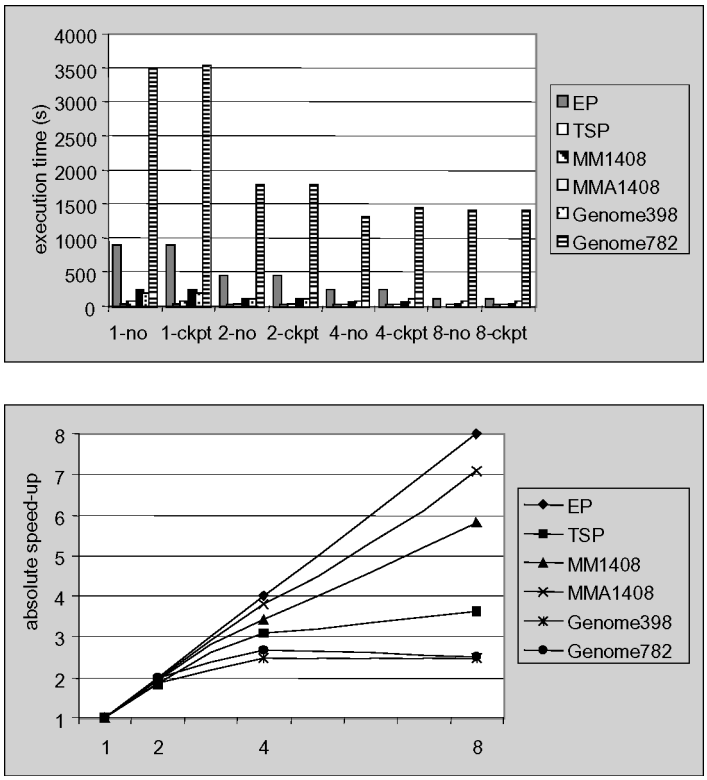

Fig. 4. Execution times (s) and speedups for the applications.

order to reproduce a highly communication-intensive scenario with bad speedups.

Figure 5 shows the average size of the checkpoint file for these applications.

For the *ckpt* library there seems to be a lower bound on the checkpoint file size that relies around 7 MB (Figure 5). For this reason, the sizes of the checkpoint files for EP and TSP do not decrease as we increase the number of nodes. For MM and MMA, the size of the checkpoint file for the one-node execution is 31 MB. As long as we increase the number of nodes, the part of the matrix to be calculated is smaller and so are the checkpoint files.

It must be noted that the size of the checkpoint file at node 0 is larger than in the other nodes. For instance, with 8 processors, the size of the checkpoint file

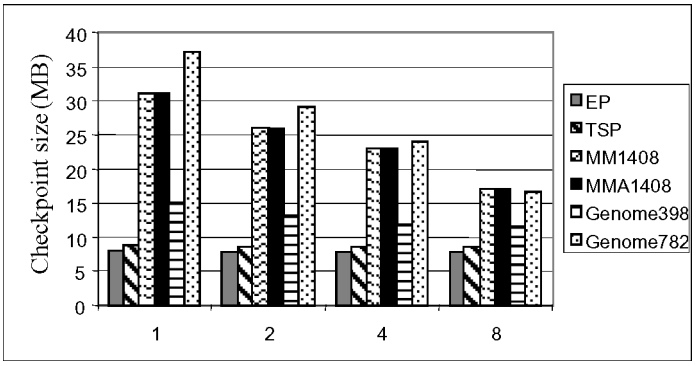

Fig. 5. Average checkpoint sizes for the applications.

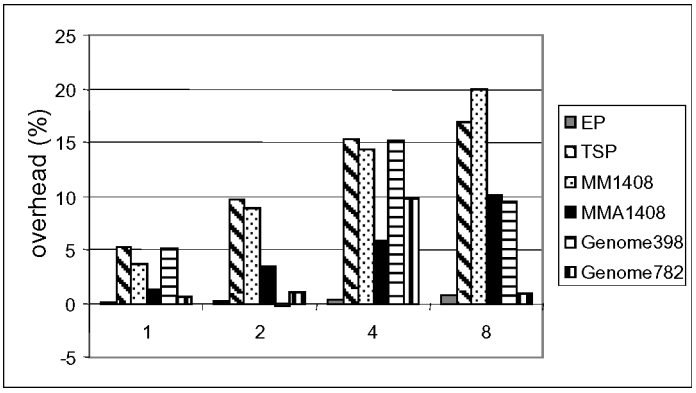

Fig. 6. Failure-free overheads for the six applications.

for MM1408 is 23.1 and 14.3 MB, at node 0 and at the other nodes, respectively. This can be explained, since node 0 is considered the master node and contains data structures that are exclusive to it.

Figure 6 shows the overhead introduced by our checkpointing strategy in failure-free executions. The highest overhead obtained was 19.9%, achieved when executing MM with eight processors. For the four benchmarks analyzed, the overhead introduced by our mechanism increases as long as the number of nodes is increased. This can be explained since, with more nodes, the execution time is smaller, but the size of the checkpointing file does not decrease in the same rate (Figure 5) and 7 MB seems to be a lower bound for the checkpoint file size. Thus, the impact of the overhead on the total execution time increases when more nodes are added.

The genome applications (genome398 and genome782) were of particular interest since they presented really bad speedups (Figure 4) and this could lead to a negative impact in our checkpointing mechanism. For these two applications, the highest overheads (9.7 and 15.1%, respectively) were obtained with four nodes. With eight nodes, the failure-free overhead presented by these applications dropped to 0.8 and 9.4%, respectively.

Nevertheless, even in this challenging and worst case scenario, the overhead values obtained in our experiments we have conducted to evaluate our checkpointing mechanism with the genome applications were close to the ones obtained with three benchmarks (TSP, MM, and MMA). Furthermore, to investigate the performance of our recovery mechanism, we turned the power off while the genome782 application was running, waited for 5 min and then turned the power on. After rebooting, all the nodes restarted execution from the last saved checkpoint and the results produced were correct. The whole recovery process took less than 2 min.

8. CONCLUSIONS AND FUTURE WORK

This article presented the design and evaluation of checkpointing and rollback/recovery mechanisms that can be adapted to DSM systems that use barriers as synchronization mechanisms. The main characteristic of our checkpointing mechanism is that it does coordinated checkpointing at barrier time, without adding messages to the DSM system. We chose to use an existing checkpointing library (ckpt) in order to separate the procedure from the mechanisms and to focus on the checkpointing and recovery mechanisms themselves.

The experimental results obtained in an eight machine cluster with four popular benchmarks and a bioinformatics application presented reasonable failure-free overheads, ranging from 0.6 to 19.9% for eight-node executions. The size of the checkpoint file is also reasonable, ranging from 7 to 17 MB with eight processors.

As future work, we shall investigate further our mechanism using long-run real-parallel applications. We shall also study further on how to integrate an incremental checkpointing strategy to our mechanism.

REFERENCES

ADVE, S. 1996. Shared memory consistency models: A tutorial. *IEEE Computer*, (Dec.). 66–76.

AMZA, C., COX, A., DWARKAKAS, S., AND ZWAENENPOEL, W. 1997. Software DSM protocols that adapt between single writer and multiple writer. In *Proceedings of the 3rd IEEE Symposium on High Performance Computer Architecture,* (Feb.) San Antonio, TX. IEEE Computer Society, Washington, D.C. 261–271.

BADRINATH, R. AND MORIN, C. 2004. Software DSM protocols that adapt between single writer and multiple writer. In *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid,* (Apr.). Chicago, IL, IEEE Computer Society, Washington, D.C. 459–466.

BAILEY, D., ET AL. 1991. The NAS parallel benchmarks. *The International Journal of Supercomputing and Applications 5*, 3, 63–73.

BENNET, J. K., CARTER, J. S., AND ZWAENENPOEL, W. 1990. Munin: Distributed shared memory based on type-specific memory coherence. In *Proceedings of the 2nd ACM SIGPLAN on Principles and Practice of Parallel Programming,* Seattle, WA, ACM Press, New York. 168–176.

COSTA, M., GUEDES, P., SEQUEIRA, M., NEVES, N., CASTRO, N. 1996. Lightweight logging for lazy Release consistency distributed shared memory. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation,* 59–73.

ELNOZHAY, M., ALVISI, L., AND WANG, L. 1996. A survey of rollback/recovery protocols in message-passing systems, Tech. Rep. TR CMU-CS-96–181, Carnegie-Mellon University, Pittsburgh, PA.

GHARACHORLOO, K., LENOSKI, D., LAUDON, J., GIBBONS, P., GUPTA, A. AND HENNESSY, J. 1991. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture* (May). Seattle, Washington, 15–26.

HU, W., SHI, W., AND TANG, Z. 1999. JIAJIA: An SVM system based on a new cache coherence protocol. In *Proceedings of the High Performance Computing and Networking.* Lecture Notes on Computer Science 1593, Amsterdam, Netherlands, (Apr.). 463–472.

IFTODE, L. 1998. *Home-Based Shared Virtual Memory,* PhD Thesis, 1998, Princeton University, Princeton, NJ.

IFTODE, L., SINGH, J. P., AND LI, K. 1998. Scope consistency: Bridging the gap between release consistency and entry consistency. *Theory of Comptuing Systems 31*, 4, 451–473.

JANAKIRAMAN, G. AND TAMIR, Y. 1994. Coordinated checkpointing-rollback error recovery for DSM multicomputers. In *Proceedings of the 13th Symposium on Reliable Distributed Systems*, Dana Point (Oct.), IEEE Computer Society, Washington, D.C. 42–51.

KATSINIS, C. AND HETCH, D. 2004. Fault tolerant distributed shared memory on a broadcast architecture. *IEEE Transactions on Parallel and Distributed Systems 15*, 12, 66–76.

KELEHER, P., COX, A. L., DWARKADAS, S., AND ZWAENEPOEL, W. 1994. TreadMarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the USENIX Winter 1994 Technical Conference* (Jan.). San Francisco, CA. 115–131.

KONGMUNVATTANA, A. AND TZENG, N. 1999. Logging and recovery in adaptative software distributed shared memory. In *Proceedings of the 18th Symposium on Reliable Distributed Systems*, Lausanne, Switzerland, (Oct.). IEEE Computer Society, Washington, D.C. 202–211.

KONGMUNVATTANA, A., TANCHATCHAWAL, S., AND TZENG, N. 2000. Coherence-based coordinated checkpointing for software distributed shared memory. In *Proceedings of the 20th International Conference on Distributed Computing Systems* (April). 556–563.

LENOSKI, D., ET AL. 1993. The DASH prototype: Logic overhead and performance. *IEEE Transactions on Parallel and Distributed Systems 4*, 1, 41–61.

LU, H., SWARKADAS, S. AND COX, A. L. 1997. Quantifying the performance differences between PVM and treadMarks. *Journal of Parallel and Distributed Computation* 43, 65–78.

MELO, R. F., WALTER, M. E. M. T., MELO, A. C. M. A., BATISTA, R. B., SANTANA, M. N., MARTINS, T., AND FONSECA, T. 2003. Comparing two long DNA sequences using a DSM system. In *Proceedings of the 9th International Euro-Par Conference*, Lecture Notes in Computer Science 2790, Klagenfurt, Austria (Aug.). Springer-Verlag, New York. 517–524.

MONNERAT, L. AND BIANCHINI, R. 1998. Eficiently adapting to sharing patterns in software DSMs. In *Proceedings of the 4th International Symposium on High Performance Computing Architecture,* Las Vegas, NV (Feb.). IEEE Computer Society, Washington, D.C. 289–299.

MOSBERGER. 1993. Memory consistency models. *Operating Systems Review 27*, 1, 18–26.

PLANK, J. S., BECK, M., KINGSLEY, G., AND LI, K. 1995. Libckpt: Transparent checkpointing under linux. In *Proceedings of the USENIX Winter 1995 Technical Conference* (Jan.). New Orleans, LA. 213–224.

SHI, W. 1999. Improving the Performance of DSM Systems, PhD Thesis, 1999, Chinese Academy of Sciences.

SMITH, T. F. AND WATERMAN, M. S. 1981. Identification of common molecular subsequences. *Journal of Molecular Biology 147*, 1, 195–197.

SPEIGHT, E. AND BENNET, J. 1998. Reducing coherence-related communication in software distributed shared memory systems. Tech. Rep. ECE-TR-98-03, Rice University, Houston, TX.

SULTAN, F., NGUYEN, T. AND IFTODE, L. 2000. Scalable fault tolerant distributed shared memory. In *Proceedings of the ACM/IEEE Conference on Supercomputing* (Nov.). Dallas, TX. IEEE Computer Society, Washington, D.C.

WANG, Y., CHUNG, P., AND FUCHS, W. 1995. Tight upper bounds on useful distributed systems checkpoints. Tech. Rep. CHRC-95–16, University of Urbana-Champaign, Urbana, IL.

ZANDY, V. 2003. Ckpt: A Checkpoint Library Under Unix. http://www.cs.wisc.edu/~zandy/ckpt.

Received August 2000; revised March 2001; accepted May 2001