The Performance of Consistent Checkpointing in Distributed Shared Memory Systems *

Gilbert Cabillic, Gilles Muller, Isabelle Puaut
IRISA, Campus de beaulieu, 35042 Rennes Cédex FRANCE
e-mail: cabillic/muller/puaut@irisa.fr

Abstract

This paper presents the design and implementation of a consistent checkpointing scheme for Distributed Shared Memory (DSM) systems. Our approach relies on the integration of checkpoints within synchronization barriers already existing in applications; this avoids the need to introduce an additional synchronization mechanism. The main advantage of our checkpointing mechanism is that performance degradation arises only when a checkpoint is being taken; hence, the programmer can adjust the trade-off between the cost of checkpointing and the cost of longer rollbacks by adjusting the time between two successive checkpoints. The paper compares several implementations of the proposed consistent checkpointing mechanism (incremental, non-blocking, and pre-flushing) on the Intel Paragon multicomputer for several parallel scientific applications. Performance measures show that a careful optimization of the checkpointing protocol can reduce the time overhead of checkpointing from 8% to 0.04% of the application duration for a 6 mn checkpointing interval.

1 Introduction

Distributed Shared Memory (DSM) [1] implements a shared memory programming interface on systems without hardware support for shared memory (e.g., distributed memory multicomputers, networks of workstations). The DSM abstraction is now recognized as an efficient alternative to message passing for the programming of scientific applications. Number-crunching applications often run for a long duration and are thus highly sensitive to system crashes (hardware or operating system). Moreover, the large number of nodes from which modern high performance multicomputers are made proportionally increases the probability of a failure. In order to be widely useful, DSM must tolerate system crashes, for instance by the provision of a checkpointing scheme. Checkpointing mechanisms consist in saving the states of processes (*checkpoints*) in *stable* (crash-proof) storage. In the event of a failure, processes are rolled back to their last checkpoint and are then restarted. Checkpointing has been widely studied in message passing environments. Studies split into two main classes: *independent checkpointing* and *consistent checkpointing*. In independent checkpointing, each process saves its own checkpoints independently, without any coordination with the others. When a process fails, all the processes must coordinate together to compute a consistent collection of checkpoints. This can lead to a *domino effect* while determining the recovery line [2, 3, 4]. Message logging has been proposed to avoid such a problem: when a process fails, it locally recovers by restarting from its last checkpoint and by replaying the messages from the log [5, 6, 7].

In the consistent checkpointing approach, the checkpointing action of individual processes is synchronized so that the set of checkpoints represents a *consistent state* of the whole system [8]. After a failure, failed processes, as well as surviving processes are rolled back to their last checkpoint. Consistent checkpointing techniques can furthermore be divided into two sub-groups: *blocking* and *non-blocking* techniques. In blocking techniques [9, 10, 11], processes synchronize together when saving a checkpoint and are halted during the whole checkpointing protocol. In non-blocking techniques [12, 13, 14], each process takes a temporary checkpoint and resumes its execution. Later on, temporary checkpoints are made definitive when it is known that all processes have saved their temporary checkpoint, and that no more message is in transit. It should be noticed that non-blocking checkpointing is more complex to implement than blocking checkpointing, as it requires message logging during the execution of the checkpointing protocol.

Recovery schemes for DSM systems must address the same issues as message passing systems. However, checkpointing mechanisms that require message logging should be avoided, since DSMs often generate a higher message traffic than message passing systems. Moreover, the logging of messages, which often contain virtual memory pages, increases the consumption of RAM memory, which is a scarce resource for many scientific applications. As a consequence, many independent checkpointing based DSMs require each pro-

*Appeared in the 14th Symposium on Reliable Distributed Systems, Bad Neuenahr, Germany, September 1995 (pages 96–105). Also published as IRISA research report number 924, available via anonymous ftp at irisa.irisa.fr under techreports/

cess to save a checkpoint at each communication with another process [15, 16, 17]. Such solutions require an unnecessarily high checkpointing frequency and checkpoint traffic, both of which are sensitive to the applications' inter-process communication frequency. This leads to a high overhead during normal operation.

This paper presents a consistent checkpointing based error recovery scheme for DSM systems. The originality of our approach relies on the integration of checkpoints within application synchronization barriers which are common in scientific applications. This technique avoids the introduction of an extra synchronization mechanism and permits to dilute the checkpoint synchronization overhead within existing synchronizations. The main advantage of our scheme is that performance degradation due to checkpointing arises only when a checkpoint is being taken, since it requires no special recovery-related actions (e.g., message logging, dependency tracking) when processes communicate. Hence, the frequency of checkpointing can be tuned to applications' needs instead of being determined by the frequency of inter-process communication.

Our proposal is implemented on a 56 nodes Intel Paragon multicomputer by extending the MYOAN DSM [18]. Its performance is measured for scientific applications. The performance analysis includes a comparison of the performance of a basic checkpointing protocol with several optimizations of this algorithm (incremental, non-blocking, and page pre-flushing). These optimizations focus on reducing the overhead of checkpointing on failure-free executions of the applications.

The remainder of this paper is organized as follows. Section 2 presents the consistent checkpointing scheme and proposes several implementations of this scheme. We report and analyze performance measurements of these implementations in Section 3. In Section 4, our research is compared with related work. Finally, conclusions are given in Section 5.

2 A Consistent Checkpointing Scheme for DSM Systems

2.1 System and Fault Model

We consider a system composed of a high speed network of *fail-stop* nodes: either a node works according to its specification, or stops working (i.e., *crashes*) without corrupting data. A simple error detection mechanism, based on the periodic use of watchdog timers, is used to detect node failures. Regardless of the physical topology of the interconnection network, the logical topology is fully connected. A (reliable) communication channel exists between every pair of nodes. Nodes share a common reliable distributed file system, based on the RAID technology [19].

The checkpointing mechanism is based on K. Li's fixed distributed manager scheme [1] (the actual algorithm [18] is an extension of Li's scheme with multiple consistency protocols). The shared memory space is divided into a set of fixed-size pages. Consistency is maintained by an invalidation-based protocol: each page has either a single read-write copy or several read-only copies; before a node writes to a read-only page, it first obtains the write privilege by invalidating all the copies of the pages on the other nodes. Information related to pages' status (access mode, copyset, owner) are kept, by a node called the page's *manager*, in a *page descriptor*. The set of page descriptors on a node is called a *directory*. Given the identifier of a page, every node knows the page's manager by applying a mapping function.

2.2 Design Overview

This paragraph gives our main motivations for the design of our checkpointing mechanism. Although it was initially designed for a multicomputer (the Intel Paragon), it can also be used for networks of workstations. However, the multicomputer target architecture influenced our design, particularly in the choice of consistent checkpointing. Furthermore, the performance of the checkpointing mechanism on networks of workstations, which is not dealt with in this paper, has to be considered.

Checkpointing scheme

When designing a checkpointing mechanism, one must choose between existing classes of algorithms the most appropriate solution. Our main design choice has been to use a consistent checkpointing scheme. Several reasons led us to this decision. Firstly, the time overhead only comes from the computation of consistent checkpoints: there is no need for logging messages, nor for tracking inter-process communications during normal operation. The frequency of checkpointing can thus be tuned to the application's needs, instead of being determined by the frequency of inter-process communications, which is hardly predictable by the application programmer. Furthermore, logging is efficient when messages are small and inter-process communications low. This is not true for many DSM based applications. Secondly, independent checkpointing techniques reduce the amount of computation lost, but require nodes to be failure-independant and to restart independantly. This assumption is not verified on most today multicomputers, for which a node crash leads to the shutdown and restart of the whole machine.

The difficulty of consistent checkpointing is to guarantee that the saved state is consistent. In blocking schemes, this is addressed by halting processes while in non-blocking schemes, inter-process communications are logged until it is known that the state is consistent (i.e., no messages are in transit). Our solution, which is based on blocking checkpointing, takes advantage of the behavior of many parallel applications in which processes regularly synchronize through barriers, thus obtaining a consistent state. By ensuring that checkpoints are always saved within barriers, we reuse these *natural* consistent states and hence avoid the overhead of computing new ones. In addition, the checkpointing scheme is made transparent to the programmer by requiring him/her to specify only the application's checkpointing interval; the processes' states are saved at the next synchronization barrier following the expiration of the checkpointing interval.

Saving DSM state

Two kinds of data must be saved when taking a process checkpoint in DSM systems: *private* data, such as the process' stack, and data *shared* with other processes (e.g., DSM pages). For the sake of simplicity, both processes' private and shared data are allocated in DSM. This permits to use the same checkpointing mechanism for both shared and private data.

When computing a consistent checkpoint, shared DSM pages must be written to stable storage. It must be chosen whether directories must also be saved or not. As directories contain the pages's owners, not saving directories implies to initialize at restart the pages' owners with arbitrary values. This generates an unnecessary high page traffic on the network when processes reconstruct their working set. Since directories are quite small, saving them add little time overhead. Consequently, we have chosen to save the directories together with the shared pages they describe.

Storage for checkpoints

Stable storage for checkpoints is provided by a file system, based on the RAID technology, which can be accessed by all nodes through the communication network. Storing checkpoint files to disk is performed by a distinguished process, called the *checkpoint server*.

There is at any time a single *permanent* consistent checkpoint, which is identified by an increasing *Consistent Checkpoint Number* (CCN). The checkpoint representation is made of a set of files, each file containing either a DSM page or a directory. There is also an *identification file* per node, which stores the current value of CCN, and is used to identify the files making the permanent checkpoint. It thus permits to discard files belonging to a checkpoint under construction (if any). Identification files are written atomically by the checkpoint server. A checkpoint under construction becomes permanent if and only if all node identification files have been written to disk.

Recovery

Thanks to the choice of a consistent checkpointing scheme, recovery is simple. After a crash, all nodes reboot the operating system. The application processes are then restarted: the permanent checkpoint is identified from the identification files contents, and the contents of DSM pages and directories are restored from the checkpoint files.

The following paragraph describes a basic checkpointing algorithm. Then, we introduce several optimizations that can be made to possibly reduce the time overhead of the checkpointing protocol during failure-free application executions.

2.3 Basic Checkpointing Algorithm

The computation of a consistent checkpoint is performed by every process within a synchronization barrier. During the checkpointing protocol, each process saves its participation to a *tentative* checkpoint. The checkpointing protocol proceeds as follows:

- Application processes synchronize with each others using a synchronization barrier;
- Each process increments its local value of CCN, and then requests the saving of its part of the tentative checkpoint to the checkpoint server. The data saved by a process P contains the pages currently owned by P (even if they have not been modified since the last checkpoint), as well as P's directory. The application process is blocked until the checkpoint server has written data to disk.
- Each process asks the writing of its identification file by sending a message containing the current value of CCN to the checkpoint server.
- Processes synchronize again in order to avoid any change to DSM pages before the end of the checkpointing protocol.

Optimizing the basic checkpoint algorithm can be performed using three strategies: (i) by reducing the amount of data saved in a checkpoint, (ii) by reducing the delay during which processes are blocked, and (iii) by using processes blocking time for beginning flushing shared pages. The following three paragraphs detail these optimizations.

2.4 Incremental Checkpointing

The basic checkpointing algorithm saves all shared pages on disk. A first optimization consists of saving only the pages that have been modified since the last checkpoint. This reduces the amount of data written to disk, and thus reduces the processes blocking time. The implementation of this scheme requires to identify which pages have been modified since the last checkpoint. As our target operating system (see section 3) does not provide a primitive for consulting the pages *dirty bits*, two mechanisms (*accurate* and *estimate*) where experimented for detecting modified pages. Both schemes add a flag DSC (*Dirty Bit Set*) to each page descriptor that indicates if the page is modified.

- In the *accurate* scheme, the DSC bit of a page indicates (accurately) if the page has been modified since the last checkpoint. The implementation of this scheme relies on the operating system ability to trap access privilege violations. When a process takes a checkpoint, only the pages with their DSC bit set are stored on disk. Their DSC bit is then reset and their access privilege is changed to read-only. The DSC bit of the page will be set again if a privilege violation occurs later. The main drawbacks of this scheme are the overhead of the system calls required for restricting the access privilege on each page written to disk, as well as the cost of detecting privilege violations.
- The *estimate* scheme has been designed to avoid the performance drawbacks of the accurate scheme. When taking a checkpoint, a process saves all its owned read-write pages, but only the owned read-only pages with their bit DSC set. The DSC bit of a page is set when the owner of the page changes, and is reset when the page has a read-only access privilege and is written to disk. Thus, the set of pages saved during the checkpointing protocol is a superset of the pages modified since the last checkpoint.

2.5 Non-blocking Page Flushing

In the basic checkpointing protocol, an application process resumes only when its directory and the pages it owns are written to disk. Two solutions, *pre-copying* and *copy-on-write*, were considered to reduce the delay during which application processes are blocked.

- In the *pre-copying* scheme, an application process is not blocked until data (DSM pages and directories) is transferred to disk. It does not even wait for an acknowledgment of the messages' receipt. A simple implementation of message passing, in which data is immediately copied across address spaces is used for communicating between application processes and the checkpoint server.
- The *copy-on-write* scheme differs from the pre-copying scheme by the way data is transferred between the application processes and the checkpoint server. Instead of using a simple implementation of message passing, this scheme uses an implementation of message passing which relies on the *copy-on-write* mechanism[1]. On the sender process, the pages to be transferred are protected against writes until the receiver physically reads them; if the sender attempts to modify a page, the kernel makes a copy of the page permitting the sender to continue its execution. This scheme is similar to the one used in [20] and [14].

[1] This mechanism is provided on the Paragon by the Norma interprocess communication facility.

In order to guarantee the consistency of checkpoint files, for both pre-copying and copy-on-write schemes, the checkpoint server is modified so as to ensure that identification files are written to disk only when the other files (containing DSM pages and directories) have been written to disk.

2.6 Page Pre-Flushing

The motivation for our last optimization, named *page pre-flushing*, is to take benefit of the behavior of irregular parallel applications, for which there is a large variation of the delay during which processes are blocked at synchronization barriers. In this scheme, the first processes arrived at a synchronization barrier can spend a portion of their waiting time requesting the flush of the pages they currently own. The page pre-flushing scheme works as follows:

- Each application process, without synchronizing with the others, increments its local value of CCN, and then sends requests to the checkpoint server, containing the pages it (currently) owns.
- When all its pages have been sent to the checkpoint server, each application process synchronizes with the others through a first synchronization barrier. Note that while the process is blocked, other application processes may still be running, and ownership of pages that were written to disk by the process may change. This implies that a page can be flushed to disk several times during the checkpointing protocol.
- Each process sends its directory to the checkpoint server, asks the checkpoint server for the writing of its identification file and then synchronizes with the other processes through a second synchronization barrier. As all processes have synchronized before saving their respective directories, they are consistent with each others. Thus, the checkpoint server is able to detect the last owner of each page, and to discard the older copies of the page.

The difference between the non-blocking page flushing schemes presented in paragraph 2.5 and the page pre-flushing scheme is that in the former schemes, processes synchronize before saving their state, thus obtaining a consistent system state, while in the latter scheme, processes save their state before every process has reached the synchronization point. The main benefit of the *pre-flushing* scheme is to possibly reduce the burst in the use of both the communication links and the checkpoint server just after processes have synchronized, which can cause a degradation of the applications performance.

3 Performance

Performance of our consistent checkpointing scheme is analyzed below. First, an overview of the software and hardware environment used for the experiments is given. The performance of the checkpointing mechanism is then analyzed.

3.1 Overview

The performance measurements were done by extending the MYOAN shared virtual memory [18], running on the Intel Paragon multicomputer [21]. MYOAN is a shared virtual memory implementing both sequential consistency, through an invalidation based protocol similar to K. Li's static distributed scheme [1], and relaxed consistency protocols suited to the applications' memory access patterns. The hardware configuration used consists of 56 compute nodes, 3 input/output nodes, and 3 RAID level 3 (bit-interleaved parity) disks. Each node includes two i860 processors and 16 Mb of memory, of which nearly 8 Mb are consumed by the operating system. Communication links between nodes have a grid topology. Nodes have access to a common clock with a microsecond precision. The Paragon run the Paragon/OSF1 operating system, based on the Mach OSF micro-kernel. We have measured a transfer rate of 3Mb/s for the parallel file system (PFS) of the machine. The checkpoint server relies on PFS and is distributed on 4 compute nodes in order to avoid the bottleneck of a centralized server.

3.2 The application programs

The experiments were done using sixteen nodes with a single application process per node. The performance of the checkpointing protocol was measured on four parallel applications: *Mp3d*, *Matmult*, *MGS* and *Radix*. *Mp3d* is an application from the SPLASH benchmark [22] that solves a problem in rarefied fluid flow simulation. The main shared data structures of the application are two large arrays; the first one stores the state information for each particle and the second one stores the properties of the space where particles move. The experiment was ran for a system of 40000 particles for 11 iterations. False sharing occurs when accessing the array of particles. *Matmult* is made of 25 loops of multiplications between two square matrices of 512x512 doubles. There is no false sharing in this application; each node fills exactly sixteen pages of the result matrix. *MGS* (Modified Gram-Schmidt) [23] is an algorithm producing, from a set of vectors, an orthonormal basis of the space generated by these vectors. The application loops on calls to the MGS algorithm for a 512x1024 problem's size. Finally, *Radix*, from the SPLASH2 benchmark, implements an integer radix sort. A summary of the applications' characteristics is given below.

Program name	Running time (Seconds)	Shared memory (Kb)
Mp3d	1487	1640
Matmult	1214	6144
MGS	575	4096
Radix	348	2272

The applications running times range from 6 mn for Radix to 25 mn for Mp3d. The total amount of data stored in DSM ranges from 1.6 Mb for Mp3d to 6 Mb for Matmult. For space considerations, results for MGS and Radix will be given only for the basic checkpointing algorithm and for the most optimized checkpointing algorithm (incremental, non-blocking).

3.3 Checkpointing Overhead

Basic checkpointing protocol

Table 1 presents a comparison between the running times of applications when run without checkpointing and when run with the basic checkpointing protocol. The checkpointing time includes: (i) the cost of network transmission to the checkpoint server, (ii) the cost of saving checkpoints on disk and (iii) the cost of communication between processes required to ensure that a consistent state is recorded. We chose checkpointing intervals ranging from 3 mn for Radix to 7 mn for Matmult. In fact, we expect application programmers in a real environment to choose longer checkpointing intervals, thus leading to a lower checkpointing overhead than the one shown in table 1.

Program name	Check. Interval (Sec)	Without checkpt. (Sec)	With checkpt. (Sec)	Difference (Sec)	Difference %
Mp3d	370	1487	1608	121	8.14
Matmult	438	1314	1459	145	11.04
MGS	240	515	707	192	22.87
Radix	174	348	512	164	47.16

Table 1: Running time with and without checkpointing (basic algorithm)

Results show that even with a non optimized checkpointing protocol, the time overhead of checkpointing is reasonably low (8% for Matmult and 11% for Mp3d). It is higher for Matmult than for Mp3d as Matmult has a higher ratio between shared data size and computation duration. The checkpointing overhead for MGS and Radix is more important that the one for Mp3d and Matmult since the checkpointing interval for these two applications is much lower.

Incremental checkpointing

The time overhead of the two incremental checkpointing protocols, estimate and accurate, have been measured for Mp3d and Matmult. Results are given in table 2.

Program name	Basic (%)	Estimate (%)	Accurate (%)
Mp3d	8.14	7.33	8.71
Matmult	11.04	6.42	6.52

Table 2: Performance of incremental checkpointing

Compared with the basic checkpointing protocol, the estimate scheme lowers the checkpointing overhead from 8% to 7% for Mp3d and from 11% to 6% for Matmult. The performance gain of the estimate scheme is higher for Matmult, where the introduction of incremental checkpointing avoids the saving of the two source matrices (2/3 of the application data). The performance gain is lower for Mp3d, for which only three shared pages are not modified between two successive checkpoints.

For both applications, the estimate scheme gives better results than the accurate scheme. This is due to the memory access patterns of the two applications, which modify their whole working space (except the source matrices for Matmult and three pages for Mp3d) between two consecutive checkpoints. The two optimizations detect the same set of pages as being modified. The additional overhead in the accurate scheme comes from the protection violations needed to accurately keep track of modified pages.

Figure 1 details the average timing of the checkpointing protocol for the basic checkpointing algorithm and the two incremental algorithms. The checkpointing time is an average of the checkpointing times measured on the 16 nodes. The figure indicates for each application process an average value of (i) the time during which the process is blocked, (ii) the time required for flushing DSM pages and (iii) the time required for flushing directories. For the accurate scheme, time (ii) includes the time required for restricting the access privilege of pages written to disk.

The figure shows that compared with the basic protocol, the time required for flushing pages to disk in the estimate scheme decreases of about 7% for both applications. When the accurate scheme is used we observe smaller gains: 6% is gained for Matmult; 20% is lost for Mp3d due to the large number of individual access privilege restrictions that have to be done (202).

An important decrease of the average process blocking time is also observed (about 13% for Mp3d and 66% for Matmult with both incremental checkpointing protocols). This can be explained by a large variation of the number of pages owned by each node. For both applications, a particular node owns a large percentage of DSM pages. This node requires more time than the others for flushing its pages, thus increasing the average process blocking time. Since for the selected applications most pages owned by this node are not modified between two consecutive checkpoints, the average process waiting time decreases.

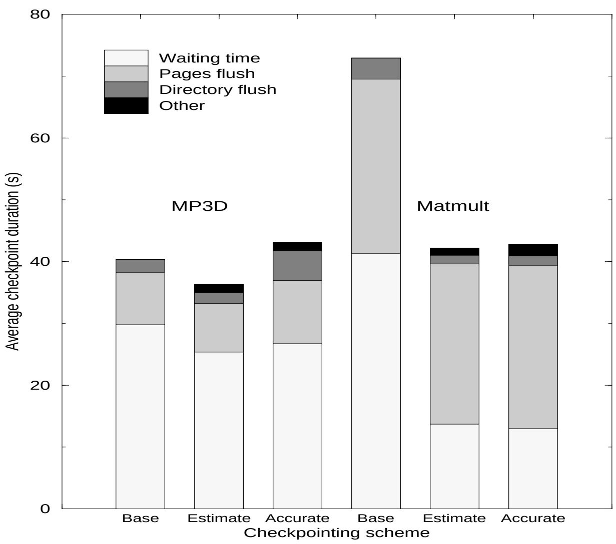

Figure 1: Detailed timing of the incremental checkpointing protocols

We observed for the accurate scheme an execution time overhead of 39 s for Mp3d (2.5% of the application's running time), and a execution time overhead of 0.8 s for Matmult (0.07% of the application's running time). This overhead comes from the treatment of protection violations, required for the accurate detection of modified pages. This overhead is higher for Mp3d than for Matmult, since more pages are saved during each checkpoint. Note that this overhead would not exist if the accurate scheme was implemented on top of an operating system allowing to read the pages dirty bits.

In summary, incremental checkpointing slightly reduces the checkpointing overhead of the basic checkpointing protocol for the considered applications. However, better gains would be achieved with an appropriate support of the operating system (e.g., ability to act on sets of pages or to consult the pages dirty bits).

Non-blocking page flushing

Table 3 gives the performance of the two non-blocking protocols: pre-copying and copy-on-write. These two schemes are implemented within the estimate incremental scheme studied in the last paragraph. The cost of checkpointing for these two optimizations only includes: (i) the cost of sending messages to the checkpoint server and (ii) the cost of synchronization (compared to the basic checkpointing scheme, the cost of network transmission and disk access are not included).

Table 3 shows that an important reduction of the checkpointing overhead is obtained by both non-blocking schemes. In average, the checkpointing overhead is divided by a factor of 35 for Mp3d and 55 for Matmult compared to the blocking incremental checkpointing

Program name	Blocking (%)	Pre-copying (%)	Copy-on-write (%)
Mp3d	7.33	0.21	0.18
Matmult	6.42	0.11	0.04
MGS	22.87	0.11	0.06
Radix	47.16	0.89	0.82

Table 3: Performance of non-blocking page flushing (incremental estimate scheme)

protocol. If we compute average values between pre-copying and copy-on-write schemes, the time required for flushing pages and directories is divided by 146 for Mp3d and by 212 for Matmult; the average process waiting time is divided by 26 for Mp3d and by 40 for Matmult. As for the incremental schemes discussed in the previous paragraph, the decrease of the processes blocking time comes from the variation of the number of pages owned by each node; a single process increases the average process blocking time since it has more pages to flush than the others; an important reduction of this process page flushing time thus leads to a decrease to the other processes waiting time.

Let us now compare the pre-copying and copy-on-write schemes on the detailed average checkpoint timing shown in figure 2.

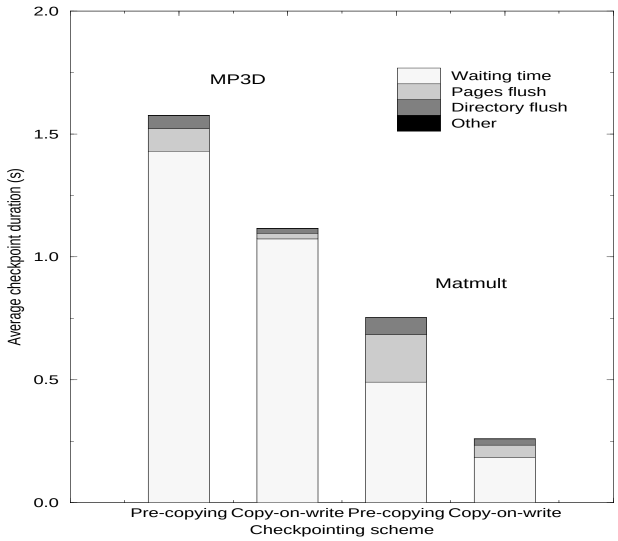

Figure 2: Detailed timing of the non-blocking checkpointing protocols

For Matmult, compared to the pre-copying scheme, the copy-on-write scheme divides the checkpointing overhead by a factor of 3. The time required for flushing pages and directories is divided by 3.4 and as explained before, due to the application memory access pattern, this results in a reduction of the processes average waiting time. An overhead of less than 0.1% is added to the application running time due to the use of copy-on-write. This means that most copy-on-write faults occur during the processes waiting time. A similar behavior can be observed for Mp3d.

Page pre-flushing

The influence of the page pre-flushing mechanism on the checkpointing overhead is shown in table 4. The checkpointing overhead is given for an incremental (estimate) copy-on-write scheme with and without the page pre-flushing mechanism.

Program name	Without pre-flushing (%)	With pre-flushing (%)
Mp3d	0.21	0.24
Matmult	0.04	0.04

Table 4: Performance of page pre-flushing (incremental, copy-on-write algorithm)

Results show that for regular applications like Matmult, the page pre-flushing algorithm has (as expected) no influence on the checkpointing overhead. For the irregular application Mp3d, the use of the page pre-flushing mechanism leads to an increase of 12% of the checkpointing overhead. This unexpected result comes from the fact that Mp3d exhibits false sharing; even if one process that is arrived at a synchronization barrier has pre-flushed a page, there is a high probability that the page will be modified by another process before the end of the checkpointing protocol. Hence, the page will be saved several times, which increases the checkpointing overhead.

Influence of checkpointing interval

Figure 3 shows the influence of the checkpointing interval on the checkpointing overhead for Matmult. The protocol used for this experiment is the most efficient for this application: incremental (estimate scheme) and non-blocking (copy-on-write scheme).

The figure shows that when a single checkpoint is taken, all pages are written to disk, leading to a checkpointing overhead of 0.18%. For additional checkpoints, only modified pages are saved. The checkpointing overhead then grows almost linearly with the number of checkpoints taken during the application. This shows that except for the first checkpoint saved, the overhead of checkpointing only depends of the number of checkpoints taken. Hence, the application programmer can adjust the checkpointing interval to the needs of its application.

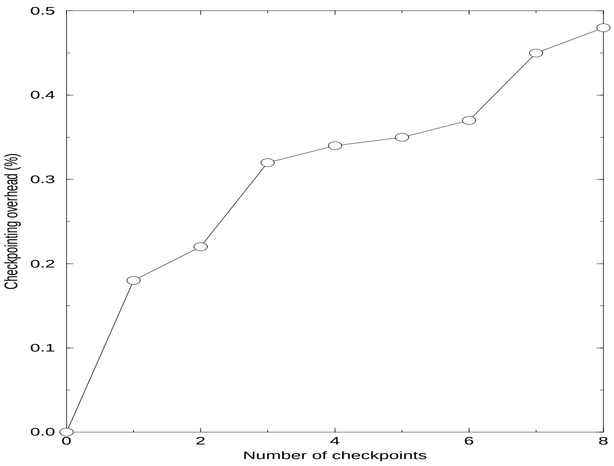

Figure 3: Influence of the number of checkpoints on the checkpointing overhead

4 Related Work

Numerous recovery mechanisms for software DSMs have been proposed. A key difference of our proposition compared to most other recoverable DSMs its that its performance was measured on an implementation rather than using trace-driven simulation. Our paper follows the same approach as [14]. Both papers compare several implementations of consistent checkpointing, but [14] focus on message passing systems, although we deal with DSM systems. Except [24], which is suited to collaborative design applications and in which recovery is provided through transactions, most recovery schemes for DSMs are aimed at parallel applications, and use either independent or consistent checkpointing. They are compared below with our work.

4.1 Recoverable DSMs based on independent checkpointing

Most recoverable DSMs are based on independent checkpointing. However, their efficiency is reduced by the fact that each process takes a checkpoint at each communication with other processes [16, 15, 17, 25].

In Wu and Fuchs's proposal [15], domino effect is avoided by requiring each process to take a checkpoint when it communicates with another process (i.e., when it reads a page that has been modified by another process). A single consistent checkpoint is maintained on a reliable twin-page disk. The main advantage of this recoverable DSM as well as many other recoverable DSMs based on independent checkpointing, is that recovery after a crash only involves one node. However, a page is transferred from one node to another only after all the dirty pages of the source node are flushed to disk, which introduces a high time overhead on failure-free executions. In contrast, our checkpointing mechanism requires a synchronization between all nodes at recovery time, but introduces an overhead on failure-free executions only when processes take a checkpoint. Consequently, our scheme permits the time overhead of error recovery to be adjusted to the applications needs, while it is imposed by the applications data sharing rate in [15].

Tam and Hsu focus in [16] on the recovery of the DSM data structures (directories). The set of directories is considered as a distributed database. When a page migrates from one node to another, the database is updated through an optimized transactional scheme. Since our scheme requires that processes synchronize before saving their directories, the directories saved on disk are mutually consistent. Thus we do not need as in [16] complex mechanisms for ensuring the consistency of data.

Stumm and Zhou propose in [17] four algorithms for building fault tolerant DSMs. The fourth (and most sophisticated) one, replicates each shared page on failure independent nodes in order to support the crash of one node. When a dirty page has to be transferred from one node to another, a copy of the page is left on the source node. In addition, in order to maintain sequential consistency, all dirty pages of the source node must then be transferred atomically on the destination node. As copies of pages are generated each time shared pages are transferred between nodes, a garbage collector, using timestamps, detects and frees old copies of shared pages. Unlike the algorithms proposed in [17], no garbage collection is required in our scheme, as only two different values of each shared pages exist in our system: the up-to-date value, which is stored in the nodes memories, and the recovery value, which is stored on disk in the permanent checkpoint. In addition, our scheme does not introduce any overhead when transferring shared pages across nodes.

Brown and Wu describe in [25] a recoverable DSM based on the use of *integrated-snoopers*. Each node embeds both an application process and a *snooper* process. The snooper of a node maintains for a subset of the shared pages their owner, copyset and last value. The snooper of a page can respond on behalf of a failed owner. The responsibility for snooping a page migrates from node to node (the snooper of a page is dynamic). Although snoopers can be implemented efficiently on broadcast networks, messages must be sent explicitly to the snooper process on other network topologies, which then increases the network traffic. The main benefit of [25] is that after a node crash, computation can be restarted without waiting for the faulty node to be repaired. However, like most proposals, an expensive operation (here, a communication with the snooper) is required at each migration of a dirty page.

4.2 Recoverable DSMs based on consistent checkpointing

Few recoverable DSMs are based on consistent checkpointing [26, 27]. In [26], the nodes' main memories are used to store both current and recovery data. In addition to the most recent copies of a shared page, at least two recovery copies of the page, stored in different nodes, are required to support a single node failure. The DSM's consistency protocol is extended so as to ensure that the recovery copies of each shared page always exist. The benefit of this solution is that the use of disks for saving recovery data is avoided, thus leading to a performance gain when using a high speed communication network. In addition, recovery pages that have not been modified since the last checkpoint can be read by application processes. Compared to our scheme, the recoverable DSM proposed in [26] requires important modifications of the consistency protocol, which may require inter node communications for always having two recovery copies of each shared page. Moreover, the use of the nodes memory to store recovery data reduces the size of user available memory, which leads to an increasing amount of swapping, and decreases the performance of large parallel applications. The recoverable DSM described in [26] exhibits a checkpointing overhead that ranges from 5% to 35%, for a short checkpointing interval of 3 seconds. These results can hardly be compared with our scheme as the application memory requirements in [26] are much lower than our test applications.

The recovery scheme closest to the one proposed in this paper is described in [27]. In order to limit the number of processes that must synchronize when saving a checkpoint, the DSM's consistency protocol is modified so as to track dependencies between processes. When a process takes a checkpoint, only dependent processes must synchronize and save their state; but all nodes must synchronize when recovering from a crash. The directories are not saved when processes save a checkpoint; they are reconstructed at recovery time. This reduces the amount of data to be written to disk but makes the recovery algorithm more complex. Unlike [27], the several schemes described in this paper require a synchronization of all the processes, both at checkpointing and recovery; in addition directories are written to disk when computing a consistent checkpoint. These two points have greatly simplified the implementation of the checkpointing protocol, leading thus to an efficient implementation. In addition, it was not shown in [27] that tracking dependencies between processes actually increases the performance of the checkpointing protocol.

5 Concluding Remarks

This paper has described the design and implementation of a consistent checkpointing mechanism for DSM systems. The main interest of the paper relies on the study in a real environment of several optimizations of consistent checkpointing (incremental checkpointing, non-blocking page flushing and page pre-flushing). The performance of these optimizations was measured on four parallel applications: fluid flow simulation (Mp3d), matrix multiplication (Matmult), modified Gram-Schmidt algorithm (MGS), and integer radix sort (Radix). While the interest of the third optimization appeared to be limited for the considered applications, the first two optimizations have permitted to reduce the time overhead of checkpointing from 8.14% to 0.04% for Mp3d, from 11.10% to 0.04% for Matmult, from 22.87% to 0.06% for MGS and from 47.16% to 0.82% for Radix. In average, for the selected applications, implementing both incremental checkpointing and non-blocking page flushing divides the checkpointing overhead by a factor of 80. In addition, it was shown that the time overhead due to checkpointing increases almost linearly with the number of checkpoints saved in an application. This permits the application programmer to choose the checkpointing interval according to its applications needs of reliability. As already stated in [28, 11], we found during the implementation of the checkpointing protocol that today's operating systems, even based on the micro-kernel technology, do not offer enough support for implementing incremental checkpointing, since they do not offer primitives for reading the pages dirty bits or acting on sets of pages.

Acknowledgments

This paper has benefited from discussions with M. Banâtre, whose comments are gratefully acknowledged. Thanks to C. Morin and B. Dupin for having read earlier versions of this paper. The design of Myoan is supported by Intel SSD under an External Research and Development Program (INRIA contract number 193C214313180120).

References

[1] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–357, November 1989.

[2] B. Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, 1(2):220–232, 1975.

[3] W. G. Wood. A decentralized recovery control protocol. In *Proc. of 11th International Symposium on Fault-Tolerant Computing Systems*, pages 159–164, Portland (OR), June 1981.

[4] G. Bhargava and S. R. Lian. Independent checkpointing and concurrent rollback for recovery in distributed systems - an optimistic approach. In

Proc. of the 7th Symposium on Reliable Distributed Systems, pages 3–12, Colombus (OH), October 1988.

[5] R. E. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, August 1985.

[6] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle. Fault-tolerance under Unix. *ACM Transactions on Computer Systems*, 7(1):1–24, 1989.

[7] E.N. Elnozahy and W. Zwaenepoel. Manetho: Transparent rollback-recovery with low overhead, limited rollback and fast output commit. *IEEE Transactions on Computers*, 41(5):526–531, May 1992.

[8] K. M. Chandy and L. Lamport. Distributed snapshots : Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[9] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering*, 13(1):23–31, January 1987.

[10] P. Leu and B. Bhargava. Concurrent robust checkpointing and recovery in distributed systems. In *Proc. of 4th International Conference on Data Engineering*, pages 154–163, Los Angeles (CA), February 1988.

[11] G. Muller, M. Hue, and N. Peyrouze. Performance of consistent checkpointing in a modular operating system: Results of the FTM experiment. In K. Echtle, D. Hammer, and D. Powell, editors, *First European Dependable Computing Conference - EDCC1*, volume 852 of *LNCS*, pages 491–508, Berlin, October 1994. Springer Verlag.

[12] K. Li, J.F. Naughton, and J.S. Plank. Checkpointing multicomputer applications. In *Proc. of the 10th Symposium on Reliable Distributed Systems*, pages 1–10, September 1991.

[13] L.M. Silva and J.G. Silva. Global checkpointing for distributed programs. In *Proc. of the 11th Symposium on Reliable Distributed Systems*, pages 155–162, Houston (TX), October 1992.

[14] E. L. Elnozahy, D. B. Johnson, and W. Zwaenepoel. The performance of consistent checkpointing. In *Proc. of the 11th Symposium on Reliable Distributed Systems*, pages 39–47, October 1992.

[15] K. L. Wu and W. K. Fuchs. Recoverable distributed shared memory: Memory coherence and strorage structures. *IEEE Transactions on Computers*, 34(4):460–469, April 1990.

[16] V. O. Tam and M. Hsu. Fast recovery in distributed shared virtual memory systems. In *Proc. of 10th International Conference on Distributed Computing Systems*, pages 38–45, Paris, France, May 1990.

[17] M. Stumm and S. Zhou. Fault tolerant distributed shared memory algorithms. In *Proc. of 2nd IEEE Symposium on Parallel and Distributed Processing*, pages 719–724, Dallas, Texas, December 1990.

[18] G. Cabillic, T. Priol, and I. Puaut. MYOAN: an implementation of the KOAN shared virtuel memory on the Intel Paragon. Research Report 812, IRISA, March 1994.

[19] P. M. Chen, E. K. Lee, A. Gibson, R. H. Katz, and D. A. Patterson. Raid: High-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2):145–185, 1994.

[20] K. Li, J.F. Naughton, and J.S. Plank. Real-time concurrent checkpoint for parallel programs. In *Second ACM SIGPLAN Symposium on Principles and Practice Parallel Programming (PPOPP), SIGPLAN notices*, volume 25, pages 79–88, 1990.

[21] Intel Corporation. *Paragon User's Guide*, 1993.

[22] J.P. Singh, W.D. Weber, and A. Gupta. Splash : Stanford parallel applications for shared-memory. Technical Report CSL-TR-91-469, Computer Systems Laboratory, Stanford University, April 1991.

[23] G. H. Golub and C. F. V. Loan. *Matrix Computation.* The Johns Hopkins University Press, 1990. Second edition.

[24] M. J. Feeley, J. S. Chase, V. R. Narasayya, and H. M. Levy. Integrating coherency and recoverability in distributed systems. In *Proc. of the First Symposium on Operating Systems Design and Implementation*, November 1994.

[25] L. Brown and J. Wu. Dynamic snooping in a fault-tolerant distributed shared memory. In *Proc. of 14th International Conference on Distributed Computing Systems*, pages 218–226, Poznan, Poland, June 1994.

[26] A. Kermarrec, G. Cabillic, A. Gefflaut, C. Morin, and I. Puaut. A recoverable distributed shared memory integrating coherence and recoverability. In *Proc. of 25th International Symposium on Fault-Tolerant Computing Systems*, Pasadena, CA, June 1995.

[27] G. Janakiraman and Y. Tamir. Coordinated checkpointing-rollback error recovery for distributed shared memory multicomputers. In

Proc. of the 13th Symposium on Reliable Distributed Systems, pages 42–51, Dana Point, CA, October 1994.

[28] A. W. Appel and K. Li. Virtual memory primitives for user programs. In *Proc. of 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 96–107, April 1991.