Using Logging and Asynchronous Checkpointing to Implement Recoverable Distributed Shared Memory

Golden G. Richard III Mukesh Singhal
Department of Computer and Information Science
The Ohio State University
Columbus, OH 43210
Email: {grichard, singhal}@cis.ohio-state.edu

Abstract

Distributed shared memory provides a useful paradigm for developing distributed applications. As the number of processors in the system and running time of distributed applications increase, the likelihood of processor failure increases. A method of recovering processes running in a distributed shared memory environment which minimizes lost work and the cost of recovery is desirable so that long-running applications are not adversely affected by processor failure. This paper presents a technique for achieving recoverable distributed shared memory which utilizes asynchronous process checkpoints and logging of pages accessed via read operations on the shared address space. The proposed scheme supports independent process recovery without forcing rollback of operational processes during recovery. Our method is particularly useful in environments where taking process checkpoints is expensive (e.g., in some UNIX®[1] environments).

Keywords: Distributed shared memory, crash recovery, checkpointing, logging.

1 Introduction

Traditionally programs running on a distributed system are structured as processes executing on physically distinct processors, with communication being accomplished by exchanging messages. Distributed shared memory [6] provides an attractive alternative with several benefits. These benefits include reduced effort in porting applications written for environments with a physically shared memory (such as tightly-coupled multiprocessors), the ability to easily share complex data structures among processes running on distinct processors, the presence of a very large virtual address space, and ease in performing process migration, whether for purposes of recovery or load balancing [6].

In a distributed shared memory (DSM) system, the shared address space is implemented in software, unlike the physically shared memory in tightly-coupled multiprocessors. The address space is physically distributed over the processors in the system and the software layer implementing DSM provides a shared memory abstraction by transferring pages between processors as needed. Stumm and Zhou [14] examine four algorithms for implementing DSM. The *central-server* algorithm utilizes a central server which controls all access to the DSM. The *full-replication* algorithm assumes that the shared data is replicated on all processors and updates are applied to all copies. The *migration* and *read-replication* algorithms distribute shared data over the processors, with the read-replication algorithm allowing multiple concurrent readers.

As the number of processors in a distributed system and the running time of applications executing on such a system increases, the likelihood of processor failure due to machine malfunction, power failure, user error, etc. increases. To minimize lost work, processes running on a DSM system should be recoverable. It is also desirable that recovery of a process affects as few other processes in the system as possible; that is, we would like to avoid having to undo work performed by operational processes.

Process checkpointing provides a technique for allowing processes to recover from processor failures. When a process takes a checkpoint, the process registers, stack, local memory contents, etc. are saved on stable storage. After a failure, process checkpoints can be used to restore the system to a consistent state. A *consistent* state in a DSM system is a state in which no process has performed a read operation for which the corresponding write operation has not been performed. Two methods for taking checkpoints are commonly used: synchronous

[1] UNIX is a trademark of Novell, Inc.

checkpointing and asynchronous checkpointing. In *synchronous* schemes, processes coordinate to insure that their checkpointed states form a consistent global system state. *Asynchronous* checkpointing requires no interaction between processes, but can result in rollback propagation (called the *domino effect*) if no additional measures are taken for recovery. In the extreme case, this propagation continues until all processes have rolled back to the beginning of execution. Consider processes *i* and *j* in Figure 1. Process *i* has taken process checkpoints C1, C2, and C3 independently and process *j* has taken checkpoints C1' and C2'. Each process has performed the operations indicated on the shared variables *w, x, y*, and *z*. When process *j*'s processor fails, *j* is restored from its most recent checkpoint, C2'. However, the resulting global state, (C3, C2'), is not consistent because the write operation on *x* has been undone, though the value written has been seen by *i*. Process *i* is then rolled back to C2 in an attempt to achieve a consistent system state. The resulting global state (C2, C2') is still not consistent, however, because in the process of rolling back to C2, the write to variable *y* performed by *i* is undone. Since *j* has accessed the value of *y* corresponding to the undone write, *j* must roll back again, this time to C1'. This rollback propagation continues until *i* and *j* have been rolled back to the beginning, thus losing all work they performed before the failure.

This paper presents a technique for recovery of failed processors in a DSM system which is based on asynchronous checkpointing of application processes and logging of pages accessed via read operations on the shared address space. We use a logging technique to avoid rolling back *any* operational process and therefore avoid the domino effect. Processes log pages which are read to volatile storage and the processors upon which the processes are executing commit the logs to stable storage before exposing modified pages to other processors. Process checkpoints can be taken periodically to reduce both log size and the time needed for recovery in the event of a failure. When a processor failure is detected, the failed processes can be recovered by reinstating their most recent checkpoints and then performing independent recovery with the aid of the logs on stable storage. This procedure can be performed on any available processor and does not require operational processes to roll back.

The rest of the paper is organized as follows. Section 2 presents the underlying model of computation. Section 3 presents the proposed technique for implementing recoverable distributed shared memory. Section 4 examines related work on recovery and fault tolerance and compares it to the proposed method. Section 5 concludes the paper.

2 System model

The system we consider is composed of a collection of homogeneous workstations (processors), connected via a local area network, and supported by file servers, which provide a global shared disk environment. The processors share neither physical memory nor a common (global) clock. Processors communicate solely by exchanging messages.

The local area network is assumed to provide fault-free communication, via FIFO channels, between the processors. Disk storage is assumed to be reliable. File servers must not remain down indefinitely after a crash and must resume fault-free operation after restoration. Processors are assumed to be fail-stop [9]; that is, they stop functioning without generating spurious or incorrect messages. A checkpoint/restore mechanism, capable of saving and restoring the state of processes executing on the processors, is assumed to exist. Processor failure is detectable via time-outs or some alternative mechanism and is assumed to cause failure of all processes executing on the processor.

A program running on the DSM system consists of processes which communicate only via access to variables in the DSM (that is, they do not explicitly exchange messages). Multiple processes can exist on a single processor. All processes which make up a program are deterministic.

We consider a DSM model that allows, for each shared page, either multiple reader processes or a single writer process (i.e., we assume a *read-replication* algorithm for DSM). Allowing multiple read-only copies of a page increases the level of concurrent access to shared data, but also makes maintaining memory coherence more difficult. The memory coherence semantics for an implementation of DSM dictate how "current" shared pages read by a process must be. A number of choices for memory coherence semantics have been proposed, including strict, sequential, processor, weak, and release consistency [8]. We focus on *strict* consistency, which insures that the copy of a page *p* read by a process contains the most recently written value. This choice provides a programmer designing an application for a DSM system with the familiar coherence semantics of uniprocessors. DSM with less restrictive memory coherence requirements can be more efficient and is likely to provide greater levels of concurrency, but with potentially greater effort for the application programmer. Cheriton [3] and Ahamad et al. [1] discuss approaches to implementing DSM with relaxed coherence semantics.

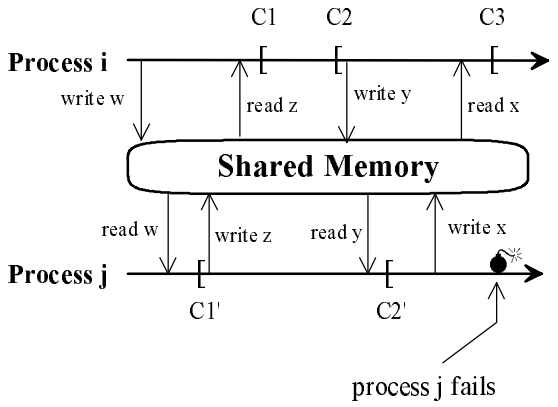


Figure 1. **The domino effect.**

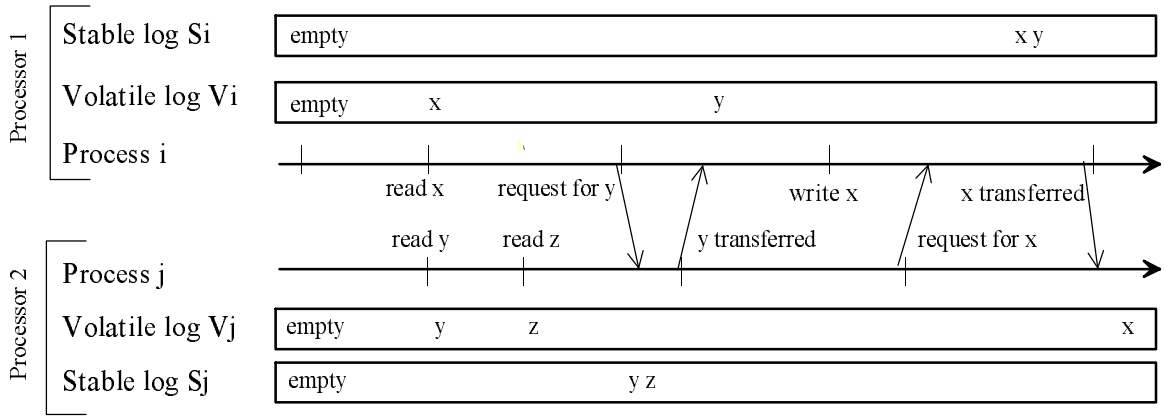
3 A technique for recoverable DSM

Our objective is to develop a technique which allows independent process recovery in a DSM system. By independent recovery we mean the ability to recover a process without rolling back operational processes and with as little interference with operational processes as possible. We also minimize the number of *required* checkpoints, since in some environments taking checkpoints is expensive. Our technique allows processes on distinct processors to take checkpoints asynchronously; the interval between checkpoints can be adjusted to balance the cost of checkpointing vs. the savings in recovery time gained by taking checkpoints often. When a processor failure is detected, process recovery begins immediately on an available processor to ensure that modifications to the shared address space made by the failed processes are not lost and are consistent with respect to other processes. The technique tolerates failure of single processors; if more than one process is executing on a processor that fails, all the failed processes are restarted on a common processor.

The proposed technique is incorporated into the protocol for maintaining memory coherence and is related to the Dynamic Distributed Manager algorithm presented in [6]. The unit of sharing is a page. Multiple concurrent readers (with no writers) for a page are allowed, but only a single writer (with no readers) is allowed. Pages and ownership information are distributed over the processors in the system. No central coordinator is assumed. To gain ownership or read access for a non-local page, a request is sent to the *probable* owner of the page (the probable owner is the processor's best guess at who the owner is, based on local information). If a processor receives a request for a page which it no longer owns, it forwards the request to the processor it granted the page ownership to last, and so on. When the request reaches the owner of the page, the owner transfers the page to the original requesting processor. Strict coherence is maintained by using a *write-invalidate* protocol to invalidate the read-only copies of a page before a write is allowed to proceed.

When a process reads a shared page, it saves a copy of the page in its volatile log. The log contents is tracked to avoid logging duplicate copies of a page, which would result in unnecessarily large volatile logs. Before a processor grants a request for transfer of a page, the volatile logs of processes executing on the processor are flushed to the stable logs, stored on a stable, shared disk. This ensures that the stable logs contain sufficient information to allow failed processes to reexecute at least far enough to have knowledge of all updates they made to the DSM which were seen by other processors. The volatile logs of multiple processes executing on a processor must be flushed to stable storage as an atomic event to ensure consistency when recovery is performed. Figure 2 illustrates the logging operations for two processes *i* and *j*, executing on distinct processors. In the initial state, *i* and *j* have performed no read operations so the volatile and stable logs for both processes are empty, processor 1 (executing *i*) owns page *x*, and processor 2 (executing *j*) owns pages *y* and *z*. When process *i* reads page *x*, a copy of the page is stored in *i*'s volatile log. The reads of *y* and *z* by process *j* result in similar logging. When *i* attempts to read page *y*, a page fault results and a request for *y* is sent to processor 2. Before processor 2 transfers a copy of page *y*, the contents of *j*'s volatile log is flushed to the stable log. When processor 1 receives a copy of page *y*, *i* reads *y* and stores a copy of *y* in its volatile log. Note that the value for *x* created by *i*'s subsequent write is not logged; this value will be reconstructed during recovery should *i* later fail. When process *j* attempts to read *x,* a page fault occurs and a request for *x* is sent to processor 1. Before the page request is honored, *i* flushes the contents of its volatile log to stable storage. Note that when the contents of the volatile log is copied into the stable log, the volatile log is cleared. This is not apparent in the illustration.

When a process fails, it is restarted from its most recent checkpoint and read operations are satisfied from the log on stable storage. During recovery, with the aid of the stable log, a process is able to execute until it has reconstructed all modifications to the DSM which were visible to other processors before its failure. When the log contents is exhausted, the process resumes normal execution.



Volatile logs are stored in local memory while stable logs are stored on stable storage. The contents of a volatile log is flushed to the stable log before a processor grants a page request.

Figure 2. **Volatile and stable logs.**

The rest of the section is organized as follows. Data maintained by each process and processor is explained in section 3.1. In section 3.2, the structure of the volatile and stable logs is explored. Section 3.3 explains how read and write operations on the DSM are performed and how page requests are processed. Section 3.4 addresses the problem of determining which processor owns a page, particularly in the presence of failures which have destroyed the page ownership information stored in a processor's page table. Section 3.5 covers process checkpointing, which is useful in reducing log size and the time required for processor recovery. Process recovery is discussed in more detail in 3.6. Section 3.7 discusses correctness. Process migration and completion are addressed in 3.8.

3.1 Data structures

A processor in the system may execute multiple processes. Processes executing on a single processor share the page table for the processor, but maintain local data of their own for purposes of recovery and logging. Each processor maintains the processor-local data illustrated in Figure 3. *ID* contains a unique identifier for the processor. *lock* is an array of locks used to control access to each page table entry by the processes residing on the processor. Array *pageentry* forms the processor's page table. Each entry of the page table contains four fields: *owner, copyset, ts,* and *data.* If for page *p* the *owner* field is equal to *ID*, then processor *ID* owns page *p*. Otherwise the *owner* field contains the probable owner of the page or $\perp$ (to indicate that the owner is currently unknown). *copyset* contains a (possibly empty) list of the processors which have read-only copies of the page. *copyset* $=\emptyset$ and *owner=ID* implies that processor *ID* has exclusive ownership and may perform writes. *ts* is a vector timestamp which tracks the process states of the processes which have written *p;* it is updated whenever a process writes *p*. *ts* is needed to capture causual dependencies among processes (see the next paragraph for timestamp details and Section 3.3 for more information on how they are updated). *data* contains the page contents or a sentinel value indicating the page data is on disk.

Each process *i* running on the processor maintains the data illustrated in Figure 4. *TSi* is a vector clock for process *i*. A *vector clock* [7] is an array of integers, one integer for each process in the system. The vector clock at process *i* ticks by incrementing the entry corresponding to process *i* whenever a shared memory access is performed by *i*. *TSi* is also updated using the *ts* vector timestamp attached to a page whenever that page is accessed by process *i*. This update sets each entry $TSi(j)$, $j \neq i$, to the maximum of $ts(j)$ and $TSi(j)$ while leaving $TSi(i)$ unchanged. For writes, the updated *TSi* is used to replace the timestamp *ts* for the written page. During recovery, timestamps are used to ensure that a process performing a write does not destroy a more current version of a page. *recovering* indicates whether the process is currently recovering.[1]

```
ID                  /* processor ID    */
lock(1..#pages)     /* pageentry locks */
pageentry(1..#pages) /* page table     */

Each pageentry(p):
owner    /* processor ID of owner      */
copyset  /* set of processors with     */
         /* readable copies            */
ts       /* vector timestamp for page  */
data     /* pointer to page contents   */
```

Figure 3. **Local data for each processor.**

3.2 Log structure

To facilitate quick recovery, the log structure shown in Figure 5 is used. A singly-linked list of copies of *pageentry*(*p*) is maintained for each page *p*. The *pageentry* fields *owner* and *copyset* are not saved in the log. A field *accesscnt* is included with each page entry in the log. *accesscnt* indicates how many times a copy of page *p* in the log should be used to satisfy a read on *p* during recovery. Rather than logging multiple copies of *p* when consecutive reads of identical copies of *p* occur during normal process execution (e.g., when values of variables stored in a common page are read), the *accesscnt* field for the current copy of *p* in the log is incremented. This allows a single copy of *p* to satisfy consecutive reads of *p* in some circumstances, which reduces volatile and stable log space requirements. A pointer to the current head of each list is maintained during recovery. When a process *i* attempts to read page *p* while recovering, the *accesscnt* field for the page pointed to by HEAD is examined. If *accesscnt* for the page is non-zero, then *accesscnt* is decremented and the read is satisfied with the page pointed to by HEAD. Otherwise, HEAD is advanced. If HEAD has not reached the end of the linked list, then *accesscnt* for the copy of *p* pointed to by HEAD is decremented and the read is satisfied using this copy of *p*. If HEAD reaches the end of the linked list for a page then recovery is complete, because the contents of the log have been exhausted. This log structure does not require a search of the list of copies of page *p* in the log at recovery time; the "right" copy of *p* in the log is always the copy pointed to by HEAD or the copy following it. Note that during recovery the stable copy of the log is not modified, since the log may be needed again should another processor crash occur.

```
Process log   /* see Figure 5        */
TSi           /* vector clock        */
recovering    /* process recovering? */
```

Figure 4. **Local data for each process.**

3.3 Access to shared data

Initially we assume that the system is fault-free, to illustrate how processes access shared data and how logging is performed. We relax this assumption shortly. As processes in the DSM system execute, they invoke procedures **read** and **write** to access pages in the shared memory space. Page faults are handled by these procedures by requesting that pages be transferred from remote processors. When a page *p* is transferred, its complete *pageentry*(*p*) entry is forwarded (that is, *owner, copyset, ts,* and the page *data* are transferred as a unit). Each processor executes the **requestcopy** and **requestpage** procedures in response to requests for read-only and write access to pages, respectively.

Reading a page

When process *i* wants to read page *p* in the shared address space, it invokes procedure **read**, which performs the following operations:

1 A lock is placed for entry *p* in the processor page table.

2 The owner for page *p* is determined, then process *i*'s clock ticks by incrementing the *i*th element in the vector *TSi*.

3 If the process is recovering and the read operation can be satisfied from its log, then the appropriate page *plog* is fetched from the log. *TSi* is updated to reflect dependency on the process states associated with *plog*. If process *i*'s current process state number (as indicated by *TSi*(*i*)) is greater than the process state number for *i* attached to *pageentry*(*p*) (as indicated by *pageentry*(*p*).*ts*(*i*)) and *pageentry*(*p*).*owner* = *ID*, then *pageentry*(*p*).*data* is set to *plog.data.* The read request is satisfied using *plog.data*. Steps 4-7 are skipped during recovery.

[1] Table compaction techniques are appropriate for the *lock* and *pageentry* arrays, since maintaining an entry for each page in the virtual address space is likely to be prohibitively expensive. For clarity, however, the algorithm is presented as if *lock* and *pageentry* are directly addressable arrays.

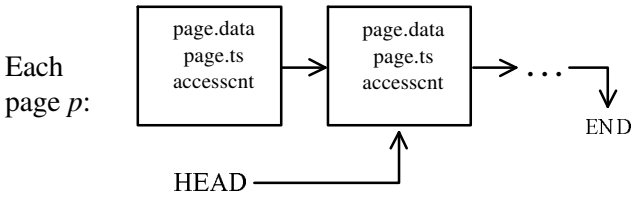


Figure 5. **Log structure.**

4 Otherwise, the process is not recovering. If processor *ID,* upon which *i* is executing, does not have a readable copy of the page, then a request for the page is made. When a copy of the page is received from the owner, the copyset for page *p* is updated to include *ID.*

5 Process *i*'s vector clock *TSi* is updated to reflect dependency on the process states associated with page *p.*

6 If *pageentry*(*p*).*ts* ≠ *ts* of the most recently logged copy of *p*, then page *p* has been modified since the last time it was logged, so a new copy of *p* is written to the process' volatile log. Otherwise, the *accesscnt* field for the logged page is incremented.

7 The read request is satisfied with *pageentry*(*p*).*data*.

8 The lock is released.

Writing a page

When process *i* wants to write to page *p* in the shared address space, it invokes procedure **write**, which performs the following operations:

1 A lock is placed for entry *p* in the processor page table.

2 The owner for page *p* is determined.

3 Process *i*'s clock ticks by incrementing the *i*th element in the vector *TSi*.

4 If processor *ID*, upon which *i* is executing, owns page *p*, then invalidate messages are sent to all sites in the copyset for *p*. When acknowledgements from all copyset sites are received, the copyset for page *p* is set to null. Step 5 is skipped.

5 Otherwise, processor *ID* must obtain exclusive ownership of the page. A request for ownership is sent to the owner of page *p*. When the page is received, *ID* assumes ownership.

6 Process *i*'s vector clock *TSi* is updated to reflect dependency on the process states associated with page *p*.

7 If process *i*'s current process state number (as indicated by *TSi*(*i*)) is greater than the process state number for *i* attached to page *p* (as indicated by *pageentry*(*p*).*ts*(*i*)), then the timestamp for *p* is set to *TSi* and write access to the data for page *p* is allowed. Otherwise, the write is discarded.

8 The lock is released.

Servicing a request for a read-only copy of a page

When a processor receives a request for a read-only copy of a page *p* from processor *IDj*, it executes the procedure **requestcopy**. This procedure performs the following operations:

1 If a process is currently recovering on the processor receiving the request, the request is queued until recovery is complete.

2 A lock for *pageentry*(*p*) is placed.

3 The owner of *p* is determined.

4 If the owner of *p* is not *ID*, then the request is forwarded to the owner. Steps 5-7 are skipped.

5 Otherwise (processor *ID* owns page *p*), the volatile logs of all processes executing on *ID* are flushed to stable storage before processing continues.

6 *pageentry*(*p*) is sent to processor *IDj*.

7 The copyset for page *p* is updated to include processor *IDj*.

8 The lock for *pageentry*(*p*) is released.

Servicing a request for ownership of a page

When a processor receives a request for exclusive ownership of a page *p* from processor *IDj*, it executes the procedure **requestpage**. This procedure performs the following operations:

1 If a process is currently recovering on the processor receiving the request, the request is queued until recovery is complete.

2 A lock for *pageentry*(*p*) is placed.

3 The owner of *p* is determined.

4 If the owner of *p* is not *ID*, then the request is forwarded to the owner. Steps 5-7 are skipped.

5 Otherwise (processor *ID* owns page *p*), messages are sent to all processors in the copyset for *p,* requesting that their copies of *p* be invalidated. When acknowledgements have been received from all processors in the copyset, the contents of the volatile logs of all processes executing on *ID* are flushed to stable storage and the copyset is set to null.

6 *pageentry*(*p*) is sent to processor *IDj*.

7 When an acknowledgement from processor *IDj* verifying the reception of the page is received, *IDj* is marked as the owner of page *p*.

8 The lock for *pageentry*(*p*) is released.

3.4 Determining page ownership

Procedure **findowner**, called by **read**, **write**, **requestcopy**, and **requestpage**, is responsible for determining which processor owns a page, for the purpose of forwarding a page request. If processor *ID* is trying to determine the owner of page *p*, then the following possibilities exist:

- If *ID*'s *pageentry*(*p*).*owner* = *ID*, then processor ID owns page *p* and no further action is required.

- If *ID*'s *pageentry*(*p*).*owner* = *P* and *P* is an operational processor, then *ID* optimistically assumes that the actual owner of the page is *P* and the page request is sent to *P*. If processor *P* no longer owns page *p*, then it will execute the **findowner** procedure to continue the search for ownership.

- If *pageentry*(*p*).*owner* = ⊥, then *ID* has no ownership information for page *p*, either because a failure destroyed the contents of *ID*'s page table and *ID* was later restored or because *ID* never had the information. To determine the owner of page *p*, processor *ID* broadcasts a request for ownership information to all processors in the system. Each processor, in response to a request for ownership information, checks its local page table and responds with one of the following: NO (the processor does not own the page), COPY (the processor has a read-only copy of the page), or OWNER (the processor owns the page). If an OWNER response is received, then *pageentry*(*p*).*owner* is set to the id of the OWNER processor. If a failed processor is encountered, the processes previously executing on that processor must be recovered and the recovery processor queried again after recovery is complete. If after receiving responses from all operational processors, and with no processes currently recovering (except possibly on *ID* itself), an OWNER is not found, then *ID* assumes ownership of the page and updates the copyset by noting which processors responded with COPY.

3.5 Checkpointing

The algorithm as described allows logs to grow without bound. To shrink the logs and reduce the time required for process recovery after a failure, process checkpoints should be taken periodically. To take a checkpoint, a process flushes the contents of its volatile log, the process state, and all dirty pages in the processor's page table to stable storage. Each process takes checkpoints asynchronously with respect to processes executing on other processors. If a process executing on processor *ID* takes a checkpoint, then the volatile logs of other processes executing on *ID* must be flushed to stable storage before process checkpointing is considered complete. The other processes on *ID* do not necessarily need to take checkpoints at this time. This ensures that a consistent state will be restored after recovery if *ID* subsequently fails. Only one checkpoint per process needs to be stored at any time; once the steps above have completed successfully, the previous checkpoint and the stable log for the process are discarded.

3.6 Recovery

When a processor fails, the contents of volatile memory, including the volatile logs, is lost for all processes executing on the processor. Recovery for a process will be performed using its stable log. When the failure of a processor is detected, the system begins recovery of the process(es) by restarting them on an available processor (called the *recovery* processor) from their most recent checkpoint(s). Two possibilities exist for recovery of the process(es): the recovery processor is already executing DSM processes or is a "spare" (and is currently idle). The latter case requires no special action. In the former case, the operational processes on the recovery processor must be blocked if they attempt to access shared data during recovery. This prevents operational processes from accessing stale data which may be temporarily introduced into the page table during recovery. Since the recovery processor resources are finite, the blocking of operational processes will also expedite recovery. Once recovery is

complete, all blocked processes resume normal operation. An alternative to blocking operational processes on the recovery processor is to associate a temporary page table with recovering processes which is coalesced into the processor page table after recovery is complete. We do not explore this option further here. Processes executing on processors other than the recovery processor are blocked only if they attempt to access pages owned by the recovery processor.

For recovery, a process restores its latest checkpoint, sets its *recovery* flag to true, and resumes execution. **read** operations on shared pages are satisfied from the stable log until the log is exhausted (see Section 3.2). This ensures that the process reads the same values as it read during execution before its failure, and therefore writes the same values. A **write** operation during recovery results in an update to a DSM page only if the page being modified has a vector timestamp which is earlier than the current value of the recovering process' vector clock. This ensures that a recovering process does not destroy values of pages updated after the failure of the process. When a **read** operation for the process cannot be satisfied from the stable log, recovery of the process is complete and the *recovering* flag is set to false. Since the contents of the volatile log were flushed to stable storage before page requests were granted by the processor executing the process prior to its failure, the stable log is guaranteed to contain sufficient information to allow the process to recover to a point where it has knowledge of performing all its visible updates to the DSM.

3.7 Correctness argument

We now show that the proposed recovery technique restores the system to a consistent state after recovery is complete and that stale data is not read by operational processes while recovery is underway. Recall that a consistent state in a DSM system is one in which no process has performed a read operation for which the corresponding write operation has not been performed by some process. As processes execute under the proposed technique, they log DSM pages which are read to volatile storage. Before a processor grants a change of ownership for a page or releases a read-only copy of a page, it flushes the volatile logs of the processes it is executing to stable storage. Consequently, if the processor subsequently fails, the system has enough information to restore a consistent state. While recovering, processes satisfy read requests from their stable logs, which provide the same sequence of values to the processes as they read during their previous normal execution. Thus recovering processes generate the same values for write operations as they generated during normal operation, before failure occurred (since the processes are deterministic). After recovery is complete, any modifications the recovering processes made to the DSM which were visible to other processes are reconstructed.

Theorem 1: The global system state after processes on a failed processor are recovered is a consistent state.

Proof: For simplicity, we first assume that each processor in the system executes a single process. We consider the case where each processor executes multiple processes shortly. Let j be a process executing in the DSM system. At some point, j reads a page p which has value V. Suppose, by way of contradiction, that all processes are operational and that the global state reflects a situation where j has read the page p (with value V), but no process has written value V to p. Let i be the process which wrote V to p before failing. At the time of this write, the processor on which i was executing must have owned page p. Before the processor granted a copy of p (containing value V) to j, it flushed the volatile log for i to stable storage. Since all processes are operational, recovery of i has taken place. When i was recovered, it was restarted from its most recent checkpoint and then satisfied all reads during recovery from its stable log. But since i previously wrote V to p when presented with input identical to that contained in its stable log, and since the stable log contained enough information for i to reach execution of the write of V to p, the write must have taken place during recovery. By contradiction, either i has not yet been recovered or the write of V to p, seen by j, has occurred. Therefore, the global system state is consistent.

We now consider the case where i and j may have executed on the same processor. In this case, when i wrote V to p and j subsequently read V from p, the volatile log for i was not necessarily flushed to stable storage. Since the volatile logs for all processes executing on a single processor are flushed to stable storage as an atomic event (as mentioned in Section 3), however, it is impossible that the contents of j's stable log contained a value V for p unless i's stable log contained sufficient information for it to reexecute the write of V to p. Therefore, consistency is maintained even when a processor crash causes multiple processes to fail and recover.

Now we show that stale data is not read by an operational process as a result of recovery. When a process performs a write operation during recovery, it compares its vector clock to the timestamp on the page to be modified. If the timestamp on the page is more recent than the process' vector clock value, then the write is not

performed ("Thomas Write Rule" [16]). This prevents a process from overwriting a more current copy of *p* (and thereby prevents introduction of stale data which might persist even after recovery is complete). If an operational process requests a copy or ownership of a page which is owned by a processor executing recovering processes, the request is queued until recovery is complete. This prevents an operational processor from reading an intermediate (stale) value of a page during recovery.

3.8 Process migration/Process completion

Before a process *i* is migrated or when the process completes execution, all dirty pages owned by the processor upon which *i* is executing must be flushed to shared disk. This is necessary because any pages modified by *i* potentially exist only in the volatile page table on the processor. If the pages are not stored on a stable medium, a subsequent processor crash would cause the modifications made by *i* to be lost, since *i* is either executing on another processor and would not be recovered as a result of the crash or has finished execution and therefore ceased to exist. In either case, a crash of the processor previously executing *i* would not result in *i* being recovered and recreating the modifications.

4 Related work

Recovery in Message-passing Systems

Sistla and Welch [10] present algorithms for processes recovery in a message-passing environment. When a process crashes and restarts, it initiates a recovery phase which involves all processes exchanging information about the extent of their respective logged states; processes then rollback and replay messages to achieve the maximum possible global consistent state. Bhargava and Lian [2] present a recovery technique which uses a two-phase rollback algorithm. It minimizes overhead during normal execution and is therefore appropriate for environments where failure is infrequent. In the worst case, processes may have to roll back to the beginning of execution. Strom and Yemini [12] present a technique based on optimistic message logging, rollback, and message replay which tolerates failure of an arbitrary number of processors. Juang and Venkatesan [5] describe methods for recovering processes which attach very little (or no) additional information to each application message. Establishment of a consistent global state is accomplished via a coordinated effort at recovery time.

We wish to avoid rolling back operational processes in the interest of not having to undo work which has been performed and to avoid the impact of global coordination at the point of recovery. Having operational processes roll back and coordinate to achieve a consistent global state in a DSM system also complicates the management of page ownership information, since ownership of pages changes dynamically as the entire system progresses.

Johnson and Zwaenpoel [4] present an optimistic message logging and checkpointing method for message-passing systems based on a central server which monitors the logging progress of the system and advances the global recovery state as processes log messages to stable storage. In the context of recovery in DSM systems, we are more interested in exploring fully-distributed approaches, to avoid the possible bottleneck that the central server introduces.

Process Recovery Under DSM

Wu and Fuchs [17] present a checkpointing and recovery scheme for DSM which uses process checkpoints and a twin-page disk storage technique. Before a process allows local updates to be seen by other processes (via creation of read-only copies of a page or by transfer of page ownership), all pages modified by the process are flushed to stable storage and a process checkpoint is taken. This ensures that if the process subsequently fails, all pages modified by the process are on stable storage and that the processor will be able to recover in a state where it remembers performing updates to DSM pages. To recover the process, it is restarted from the latest checkpoint. The twin-page disk mechanism allows incremental flushing of modified pages to disk. In some environments, taking process checkpoints frequently can result in a very significant overhead and it is in these environments that the proposed technique provides an attractive alternative, since our technique introduces the ability to tailor the checkpointing interval.

Fault Tolerance for DSM

Stumm and Zhou [13] examine methods for making the central-server, full-replication, migration, and read-replication algorithms single-fault tolerant in the absence of a shared disk. The emphasis here is on fault-tolerance rather than recovery; the concern is to ensure that parts of the shared address space do not become unavailable due to processor failure, not to recover failed processes in a DSM environment. Their methods are interesting because they do not require stable storage, but a potentially high communication cost must be paid (in the case of the migration and read-replication algorithms).

Page Table Recovery

Tam and Hsu [15] present a technique for fast recovery of the volatile page ownership information which is usually lost when a processor fails. Their technique is designed to provide an alternative to broadcast requests for ownership information during recovery.

5 Conclusions and future work

We have presented a technique for performing processor recovery in a DSM system. The technique supports independent recovery and requires no rollback of operational processes. It also requires no global coordination among processors at recovery time to achieve a consistent system state. Processes executing on distinct processors take checkpoints asynchronously. The checkpointing frequency can be adjusted to trade off the overhead of checkpointing processes often (which results in reduced recovery time) vs. the increased performance in the absence of failures gained by checkpointing less frequently.

The technique is particularly useful in environments where taking process checkpoints is expensive, such as under UNIX with no kernel support for checkpointing. With large networks of workstations running UNIX being a popular academic computing environment, large amounts of computing power are available. Using distributed shared memory on such networks eases the difficulty of writing distributed applications and the availability of workable recovery methods is important.

Our technique tolerates failure of single processors. We are investigating techniques for performing recovery in the face of multiple processor failures at a reasonable cost. Reduced consistency models are an interesting direction, because a fair amount of overhead seems to be inherent in recovery schemes for strictly consistent DSM. We are currently working on an implementation of the proposed technique to assess performance and logging overhead. We would like to express our appreciation to the anonymous referees for their useful comments. One suggested logging/checkpointing to the stores of other processors; this seems worth exploring.

References

[1] M. Ahamad, P. W. Hutto, R. John, "Implementing and Programming Causal Distributed Shared Memory," *Proc. of the 10th Intl. Conf. on Distributed Computing Systems*, Paris, June 1990, pp. 274-281.

[2] B. Bhargava, S. R. Lian, "Checkpointing and Rollback Recovery in Distributed Systems -- An Optimistic Approach," *Proc. of the 7th Symp. on Reliable Distributed Systems*, Columbus, OH, pp.3-12, October 1988.

[3] D. R. Cheriton, "Problem Oriented Shared Memory: A Decentralized Approach to Distributed System Design," *Proc. of the 6th Intl. Conf. on Distributed Computing Systems*, May 1986.

[4] D. B. Johnson, W. Zwaenpoel, "Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing," *Journal of Algorithms*, No. 11, pp. 462-491, 1990.

[5] T. Y. Juang, S. Venkatesan, "Crash Recovery With Little Overhead," *IEEE 11th Intl. Conf. on Distributed Computing Systems,* 1991.

[6] K. Li, "Shared Virtual Memory on Loosely Coupled Multiprocessors," Ph.D. dissertation, Technical Report YALEU/DCS/RR-492, September 1986.

[7] F. Mattern, "Virtual Time and Global States of Distributed Systems," *Parallel and Distributed Algorithms*, North-Holland, 1989, pp. 215-226.

[8] B. Nitzberg, V. Lo, "Distributed Shared Memory: A Survey of Issues and Algorithms," *IEEE Computer*, August 1991.

[9] R. D. Schlichting, F. B. Schneider, "Fail-stop Processors: An Approach to Designing Fault-tolerant Computing Systems," *ACM Trans. on Computer Systems*, vol. 1, no. 3, pp. 222-238, August 1982.

[10] A. P. Sistla, J. L. Welch, "Efficient Distributed Recovery Using Message Logging," *Proc. of the 8th Annual ACM Symp. on Principles of Distributed Computing*, August 1989.

[11] S. H. Son, A. K. Agrawala, "Distributed Checkpointing for Globally Consistent States of Databases," *IEEE Trans. on Software Engineering*, vol. 15, no. 10, October 1989.

[12] R. E. Strom, S. Yemini, "Optimistic Recovery in Distributed Systems," *ACM Trans. on Computer Systems*, vol. 3, no. 3, pp.204-226, August 1985.

[13] M. Stumm, S. Zhou, "Fault Tolerant Distributed Shared Memory Algorithms," *Proc. of the 2nd Symp. on Parallel and Distributed Processing*, pp. 719-724, December 1990.

[14] M. Stumm, S. Zhou, "Algorithms Implementing Distributed Shared Memory," *IEEE Computer*, May 1990.

[15] V. Tam, M. Hsu, "Fast Recovery in Distributed Shared Virtual Memory Systems," *IEEE 10th International Conference on Distributed Computing Systems*, 1990.

[16] R. H. Thomas, "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases," *ACM Trans. on Database Systems*, pp. 180-209, June 1979.

[17] K. Wu, W. Fuchs, "Recoverable Distributed Shared Virtual Memory," *IEEE Trans. on Computers*, vol. 39, no. 4, April 1990.