JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING 29, 211–218 (1995)

Ensuring Correct Rollback Recovery in Distributed Shared Memory Systems[1]

BOB JANSSENS[2] AND W. KENT FUCHS

Center for Reliable and High-Performance Computing, Coordinated Science Laboratory, University of Illinois, 1308 West Main Street, Urbana, Illinois 61801

Distributed shared memory (DSM) implemented on a cluster of workstations is an increasingly attractive platform for executing parallel scientific applications. Checkpointing and rollback techniques can be used in such a system to allow the computation to progress in spite of the temporary failure of one or more processing nodes. This paper presents the design of an independent checkpointing method for DSM that takes advantage of DSM's specific properties to reduce error-free and rollback overhead. The scheme reduces the dependencies that need to be considered for correct rollback to those resulting from transfers of pages. Furthermore, in-transit messages can be recovered without the use of logging. We extend the scheme to a DSM implementation using lazy release consistency, where the frequency of dependencies is further reduced. © 1995 Academic Press, Inc.

1. INTRODUCTION

Distributed shared memory (DSM) provides the programming advantages of a shared-memory image in scalable parallel systems. Checkpointing and rollback are commonly used to recover from detected processor errors in environments where high reliability is essential. Recent trends toward using workstation clusters for parallel scientific computing make recoverability useful even when reliability demands are less critical. In a workstation network, a node may kill a process or completely reboot, either due to a system exception, or due to direct action by a user. With checkpointing and rollback, an application can recover from such an event without restarting computation from the beginning. Checkpointing is also useful for process migration to reduce adverse impact on other users [18].

In parallel systems, dependencies between processing nodes can cause the overall system state to be incorrect when one node rolls back. The problem of rolling back to a *consistent global state* has been widely investigated for message-passing systems. It is possible to directly apply this research to shared memory, by modeling the system in terms of message passing. However, previous work in shared-memory recovery has used a laxer model of dependencies, using the intuition that only messages that transfer actual application data should cause dependencies. This assumption simplifies the implementation of a recoverable DSM, since many control dependencies can be ignored. Furthermore, the performance overhead of handling dependencies is reduced, and the potential for rollback propagation is decreased.

This paper presents the design of an independent checkpointing method for DSM that takes advantage of DSM's specific properties to reduce error-free and rollback overhead. By using periodic checkpointing, by ensuring that a node's interaction with other nodes is atomic, and by recovering the pagetable independently through *ownership timestamps*, the number of dependencies that can cause rollback propagation is reduced. Additionally, the remaining dependencies are unidirectional, allowing the algorithm to handle in-transit messages without the use of message logging. By using a *passive server* model of DSM we show that the dependencies in our recovery algorithm can be derived from the traditional message-passing model. We extend our checkpointing method to a DSM with lazy release consistency [7, 15], further reducing dependencies to only synchronization interactions. As described, our schemes are designed for software-implemented DSM (shared virtual memory) and independent checkpointing. However the ideas presented can also be extended to hardware implementations and any other distributed system checkpointing method [14].

To ensure correct recovery in a parallel system, a rollback needs to result in a consistent global state [6]. If the execution is partially deterministic, logging and message replay can be used, albeit at a high cost [9]. The simplest recovery method in general nondeterministic parallel systems is *coordinated checkpointing* [6, 8], where nodes synchronize both to checkpoint and to roll back. To avoid coordination overhead, *independent checkpointing* can be used [4, 23]. Its main disadvantages are the need to track all dependencies, the need to implement logging to enable recovery of in-transit messages, and the potential for rollback propagation.

[1] This paper contains material previously presented at the 13th SRDS [13]. This research was supported in part by the Office of Naval Research under Contract N00014-91-J-1283, and by the National Aeronautics and Space Administration (NASA) under Grant NASA NAG 1-613, in cooperation with the Illinois Computer Laboratory for Aerospace Systems and Software (ICLASS).

[2] E-mail: janssens@uiuc.edu.

0743-7315/95 $12.00

Various distributed system recovery techniques have been applied to shared memory. Earlier techniques have used communication-induced checkpointing [2, 5, 12, 24]. Schemes using coordinated checkpointing have also been developed, both in bus-based systems [2, 3] and in DSM systems [10, 11]. Various DSM schemes based on logging and deterministic replay have also been designed [19, 20, 22].

The distributed system model, where every message causes a dependency between nodes, is too strict for shared-memory parallel programs. A more relaxed dependency model can be used for rollback recovery in shared memory only if there is no possibility of deadlock due to nodes waiting for messages that may never arrive. Recovery schemes designed for bus-based shared memory systems have generally used the relaxed model. In these systems deadlock is avoided by the bounded transmission delay of the bus. In DSM systems, other measures have to be taken to avoid messaging deadlock if the relaxed dependency model is used. A dependency pattern and an atomic interaction model similar to the one described in this paper has been used to design a coordinated checkpointing scheme [11].

2. DESIGN OF A RECOVERABLE DSM

Our recoverable DSM algorithm uses a fixed distributed manager (FDM) protocol for maintaining coherence [17]. In this protocol, every node maintains ownership information for a fixed subset of shared pages. A page fault to a shared page causes a request to its manager, which forwards it to the owner. A node's page table indicates that it has either exclusive write access (W), read access (R), or invalid access (I) to a page. A copyset of nodes that have a copy of a page is maintained to allow their invalidation when a node obtains exclusive write access.

Pseudo-code for our checkpointing and rollback recovery algorithm as integrated into the DSM algorithm is given in Fig. 1. A page fault initiates an interaction, where a local fault handler is called, which then consults a request server on the owner via the manager. Every nodes calls its checkpoint routine every T seconds. Every checkpoint on a node starts a new checkpoint interval by incrementing `ckp_interval`. This value is appended to every message that transfers a page of data between nodes. The receiving node uses the value to update its *dependency table*, which is used for dependency tracking [4, 23]. When an error is detected, the rollback initiation routine constructs a consistent global checkpoint by requesting every other node's current dependency table. It then sends appropriate rollback commands to the nodes. These nodes may need to roll back multiple checkpoint intervals. Unless a garbage collection algorithm [23] is used, all checkpoints are saved until the end of program execution. Unlike message-passing recovery algorithms [4, 23], in-transit messages do not need to be replayed from a log during reexecution.

2.1. Maintaining Atomicity of Server Events

The recovery algorithm ensures that a node's part of an interaction is executed atomically. This avoids deadlock and spurious messages caused by partially completed interactions. When the checkpoint routine is called, checkpointing is delayed if an interaction is in progress. It is not possible to delay a rollback if an error is detected during an interaction. Consider the situation in Fig. 2, where a request for read access from node A has been forwarded, by the page's manager on node M, to node B. If node A rolls back while waiting, it will receive the reply from node B unexpectedly after rollback. To handle such spurious replies, a sequence number uniquely identifying the interaction is attached to each message. In the example, node A assigns a unique number to the request message, and B attaches this number to its reply. When node A rolls back, it will have no record of the sequence number sent by B, so it rejects the message.

If node B or node M rolls back to while handling the request from A, node A will not receive a reply from node B. To handle such cases of potential deadlock, a node waiting for a reply that receives a request for dependency information waits for a fixed amount of time, and if no reply is received, indicates that it must roll back by setting the `must_rb` flag in its dependency table. In the example, if node B decides to roll back before sending the reply, all other nodes in the system that receives B's request for dependency information while waiting for a reply set a timer. In the absence of other rollbacks, all nodes except A probably receive the reply before timing out. Node A does time out, sending its dependency table to B with `must_rb` set. Node B then constructs its consistent global checkpoint taking into account that node A must roll back.

2.2. Page Table Recovery with Ownership Timestamps

Our design uses an ownership timestamp scheme where every node keeps track of the last time it became owner of a page. The scheme allows all directory information beside the ownership timestamps to be lost after rollback without affecting correct execution. Every time ownership is transferred, the old owner sends its current value of the page's ownership timestamp to the new owner. Upon receiving the value, the new owner increments it and then stores it as its ownership timestamp for the page. Periodically, when the timestamp overflows, all nodes need to synchronize to reset their timestamps to ensure correct ordering. Ownership timestamps are saved together with the state of the user appliation during checkpointing.

After a node rolls back, all the page table information except for the ownership timestamps is unknown. The first access to a page after rollback causes a fault. If the manager did not roll back, the protocol proceeds as usual, restoring access rights to the requester. If the manager did roll back, it has no ownership information, and queries all nodes for their ownership timestamps for the page. It can then determine the owner of the page by comparing all the

Read Fault Handler

```
send read request to manager of page;
receive page and ckp_interval from owner;
update dependency table;
access = R;
```

Write Fault Handler

```
send write request to manager of page;
receive page, copyset, ownership
 timestamp and ckp_interval from owner;
update dependency table;
send invalidates to all members of copyset;
increment and save ownership timestamp;
access = W;
```

Read Request Server

```
access = R;
add requester to copyset;
send page and ckp_interval to requester;
```

Write Request Server

```
access = I;
send page, copyset, ownership timestamp
 and ckp_interval to requesting node;
```

Manager

```
if (owner == unknown)
  request ownership timestamps from
   all nodes;
  owner = node with largest timestamp;
endif
forward request to owner;
if (request == write) owner = requester;
```

Checkpoint

```
if (in_interaction)
  wait for completion;
endif
save dependency table;
save ownership timestamps;
save user state;
increment ckp_interval;
```

Rollback Initiation

```
request dependency info from other nodes;
determine recovery line;
send rollback request to other nodes;
```

Rollback Server

```
if (waiting)
  set timeout timer;
  if (timer expires) must_rb = 1;
endif
send dependency table to initiator;
receive rollback request;
if (rb_interval != current)
  restore user state for rb_interval;
  reset pagetable;
  restore ownership timestamps;
endif
```

FIG. 1. Pseudo-code for independent checkpointing algorithm.

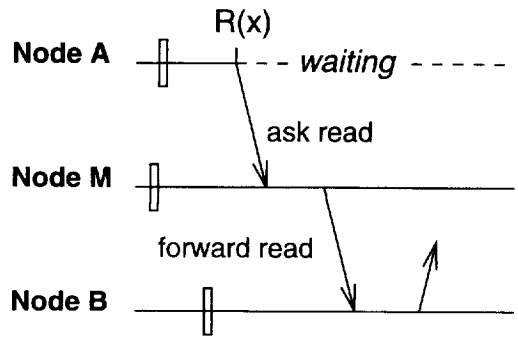

FIG. 2. Situation resulting from an incomplete interaction.

ownership timestamps received. Due to rollbacks on other nodes, it is possible that the manager has incorrect ownership information. Therefore every node keeps track of the pages it owns. If a node receives a request to a page it does not own, it rejects the request, and ownership timestamps are used to find the correct owner.

2.3. *Performance Impact*

Our DSM recovery scheme reduces the two main drawbacks of independent checkpointing techniques: the high error-free overhead of message logging and dependency tracking, and the potentially high overhead of recovery due to rollback propagation. Dependencies between

TABLE I
Address Trace Characteristics

Program	Description	Total number of references	Data reads		Data writes	
			Total	Shared	Total	Shared
gravsim	N-body simulator	92,178,814	33,266,880	12,484,455	6,392,078	251,694
fsim	Fault simulator	149,918,375	50,950,933	39,326,911	3,958,919	999,127
tgen	Test generator	101,264,382	32,613,809	16,550,450	4,461,889	642,796
pace	Circuit extractor	87,861,165	23,266,576	1,286,787	7,842,338	348,524
phigure	Global router	132,998,231	38,244,233	4,281,207	11,530,981	1,876,400

checkpoint intervals can cause a domino effect, where cascading rollbacks force reexecution of a large part of the program. To determine the reduction in dependencies caused by using our scheme we performed trace-driven measurements with multiprocessor address traces from five parallel scientific programs running on an Encore Multimax. The traces were generated by the TRAPEDS address tracer from execution on seven processors. Each trace contains at least 10 million memory references per processor [21]. Table I describes the characteristics of the traces used.

Figure 3 presents simulation results for the frequency of messages in the DSM applications. There are about 10,000 messages per million memory references, all of which cause dependencies in a traditional message-passing approach to independent checkpointing. The frequency of dependency-carrying messages is decreased by a factor of about 3.5. This means the overhead of dependency tracking is decreased. More importantly, the potential for rollback propagation and the domino effect is reduced. Since our scheme uses periodic checkpointing, with approximately the same period on each node, only a small percentage of dependencies occur between different checkpoint intervals and cause rollback propagation. The lower the frequency of dependency-carrying messages, the higher the probability that the latest set of checkpoints represents a consistent global state and can be used for recovery. In traditional message-passing independent checkpointing schemes, runtime analysis can be used to avoid logging all but 1% of these messages [23]. Our scheme does not need to perform the runtime analysis, nor does it need to incur logging overhead for any messages.

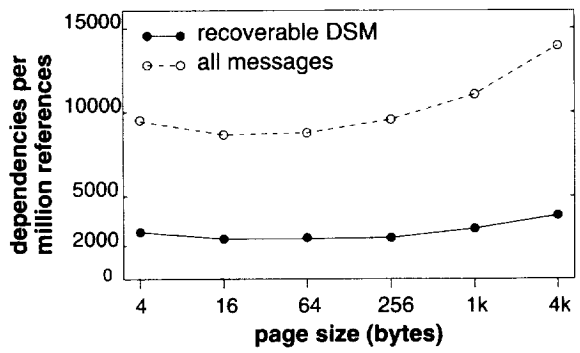

FIG. 3. Frequency of dependency comparison.

3. MODELING DSM DEPENDENCIES

In order to reason about the correctness of the DSM recovery scheme, it is necessary to extend the traditional message-passing model of execution. We describe our *passive server* model for DSM execution with rollbacks, and then analyze our recovery algorithm for DSM to show that only page-transfer dependencies remain.

3.1. The Passive Server Model

Program execution in a message-passing distributed system is modeled as a set of processes and a set of reliable channels. Program execution is represented by a pair, $P = (E, \overset{D}{\longrightarrow})$, where E is a set of events and $\overset{D}{\longrightarrow}$ is the *dependence* relation defined over E. Events within a process are ordered by the $\overset{XO}{\longrightarrow}$ (execution order) relation. Events on different processes are ordered by the $\overset{M}{\longrightarrow}$ (message) relation where $a \overset{M}{\longrightarrow} b$ means event a sent a message and event b received it. The $\overset{D}{\longrightarrow}$ relation is the union of the other two: $\overset{D}{\longrightarrow} = \overset{XO}{\longrightarrow} \cup \overset{M}{\longrightarrow}$. Every event represents an atomic action which may change the state in one of the processes. A special checkpoint event can be inserted between two events to record the current state of the process.

When a process needs to roll back, it communicates with all other processes to determine a consistent set of checkpoints. Upon receiving notification of a rollback, a process may either need to roll back to a checkpoint, or it may continue operation. If it continues, we can treat the current volatile state as a virtual checkpoint [23]. A global checkpoint is a set of real and virtual checkpoints, one per process. Consider two events a and b, where b occurs in the execution order before the global checkpoint and a occurs in the execution order after the global checkpoint. A global checkpoint is consistent if there are no two such events such that $a \overset{M}{\longrightarrow} b$ or $b \overset{M}{\longrightarrow} a$. A global checkpoint is also consistent if lost messages can be retrieved during reexecution and there are no two events such that $a \overset{M}{\longrightarrow} b$.

To simplify reasoning about consistency of global checkpoints it is useful to treat the $\overset{M}{\longrightarrow}$ relation as bidirectional.

To do this we replace every dependency $a \xrightarrow{M} b$, by a causal dependency $a \xrightarrow{C} b$, and a backward dependency $b \xrightarrow{B} a$. Consider again two events a and b, where b occurs in the execution order before a global checkpoint and a occurs in the execution order after a global checkpoint. The requirements for consistency are now that there are no two such events such that $a \xrightarrow{C} b$ and there are no two such events such that $b \xrightarrow{B} a$ and the message between a and b is unlogged.

Our passive server model for DSM systems is derived from the message-passing model. We model program execution in DSM systems as a set of client processes which run the application program and a set of passive server processes which provide a shared-memory image to the clients. Events in the servers are always triggered by the receipt of a request message, either from a client, or another server. A write or read event in a client, together with the events it causes in the servers may be collectively called an interaction. The passive server model differs from the message-passing model in that it collects all the events in a process during an interaction into one single event. Figure 4 illustrates the interactions in the FDM algorithm in terms of the passive server model. For every causal dependency between processes, there is a backward dependency which is not shown.

3.2. *Eliminating Dependencies*

We now analyze the FDM recovery algorithm to verify that all but causal page transfer dependencies can be ignored. Consider the role of the manager in an interaction. On a read interaction, the state on the manager's node (node M) is the same before the interaction as afterwards. If node M rolls back, the ownership information it maintains is lost, but it can be recovered by using ownership timestamps. So all dependencies involving node M in a read interaction can be eliminated. In a write interaction, node M changes state; it records the new owner of the page. If the new owner rolls back, node M may contain erroneous ownership information. Any request that is routed to the wrong owner by M will be rejected however, and the timestamps will be used to find the correct owner. So again we can ignore all dependencies with node M. Therefore, by using ownership timestamps, the function of the manager has been made redundant and does not have to be considered for rollback to a consistent state.

Having eliminated the dependencies with the node that contains a page's manager, we can now analyze interactions solely in terms of the dependence between the local (L) and remote (R) nodes. In a read interaction to a clean page (Fig. 4a, when node L rolls back, the state of the recovery line is the same as if node R also rolled back, except for the extra member of the copyset. Since the copyset is allowed to be a superset of all the nodes that have readable copies, the recovery line is consistent. So the backward dependency $L \xrightarrow{B} R$ can be eliminated. When the read interaction involves a dirty page (Fig. 4b), a rollback of node L will cause a situation where node R has lost write permission without guaranteeing that a copy of the dirty page has been saved on another node. However, node R is still the owner, so any further requests will be supplied from its copy of the page. Therefore the dependency $L \xrightarrow{B} R$ can again be eliminated. So, in a read interaction, there remains only the causal dependency, $R \xrightarrow{C} L$, from the remote node to the local node.

Next, we consider a remote write access (Figs. 4c and 4d). Ignoring invalidations, the interactions for a clean and dirty write are identical, with the access permission of the page on the remote node changing from W to I. If node L rolls back and reexecutes the write access, the request is directed by the manager to node R. Node R rejects the request because it has given up ownership. This rejection will cause the ownership timestamps to be used to find the correct copy of the page. Therefore, the dependency $L \xrightarrow{B} R$ is eliminated. The causal dependency $R \xrightarrow{C} L$ remains since it transmits a block of data.

If the block is readable by more than one remote node when the local node asks for write access, all the copies in the remote nodes will be invalidated. A node L can safely roll back past an interaction in which it invalidated node R'. In the global state after rollback, it will appear as if node R' has been invalidated spontaneously and at

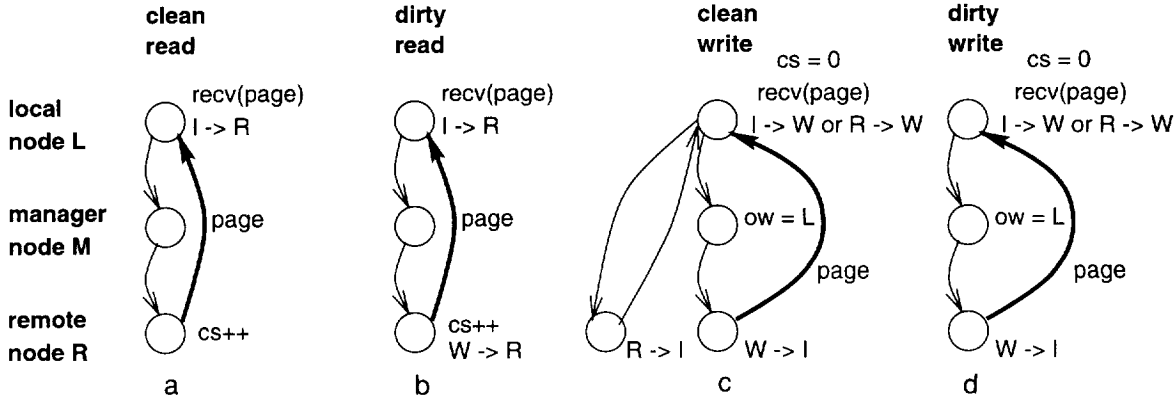

FIG. 4. FDM algorithm interactions in terms of the passive server model.

the next access node R' can ask the owner for a new copy of the block. Therefore there is no dependency $L \rightarrow R'$. If node R' rolls back past the interaction, the access rights of all its blocks are set to unknown. Therefore any access to a block that was invalidated before rollback will ask the owner for a new copy, just as if the block had been invalidated. So the remaining $R' \rightarrow L$ dependency is eliminated, resulting in a dependency-free invalidation interaction.

4. RECOVERABLE DSM WITH LAZY RELEASE CONSISTENCY

In software DSM systems, false sharing due to large page sizes, and high per-message overhead can make generating speedup with traditional protocols difficult. Lazy release consistency (LRC) successfully overcomes these drawbacks, approaching the performance of a bus-based multiprocessor on a high-speed workstation network [7, 15]. We

Write Fault Handler

```
create twin of page;
access = W;
```

Interval Creation

```
create write notices for every
 twinned page;
```

Acquire

```
send acquire request, vt, and ckp_interval
 to lock manager;
receive node id and last_acq timestamp
 from lock holder;
increment and save last_acq timestamp;
receive vt, write notices, all diffs,
 and ckp_interval from last acquirer;
update dependency table;
apply diffs;
```

Acquire Server

```
send node id and last_acq timestamp to
 requester;
wait until lock is released;
update dependency table;
send vt, write notices, all diffs,
 and ckp_interval to requester;
```

Lock Manager

```
if (last_acq == unknown)
  request last_acq timestamp
   from all nodes;
  last_acq = node with largest timestamp;
endif
forward request to last acquirer;
set last_acq to requester;
```

Checkpoint

```
if (in_interaction)
  wait for completion;
endif
save dependency table;
save node state;
```

Rollback Initiation

```
request dependency info from other nodes;
determine recovery line;
send rollback request to other nodes;
```

Rollback Server

```
if (waiting)
  if (lock holder == unknown)
    set timeout timer;
    if timer expires must_rb = 1;
  else if (lock holder == requester)
    must_rb = 1;
  endif
endif
send dependency table to initiator;
receive rollback request;
if (rb_interval != current)
  restore user state;
  set last_acq records to unknown;
endif
```

FIG. 5. Pseudo-code for recoverable LRC algorithm.

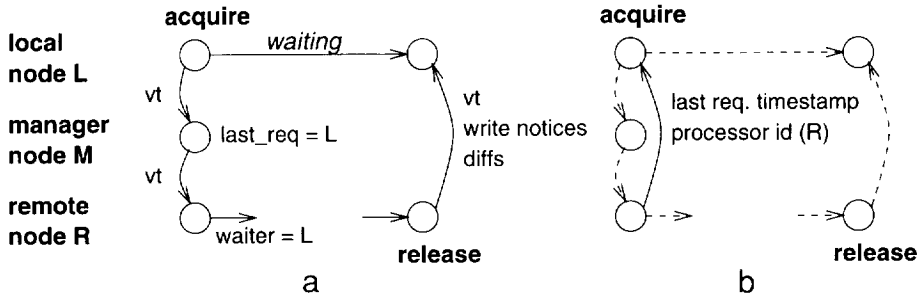

FIG. 6. (a) Acquire interaction in LRC algorithm. (b) Extra message to support rollback recovery.

use the low number of messages transmitted in a DSM with lazy release consistency to develop a recovery algorithm that further reduces the number of dependencies that need to be considered.

Our algorithm uses a modified version of the lazy update version of LRC [7]. Instead of using the traditional sequential consistency ordering of shared memory accesses [16], LRC only enforces ordering between intervals in the computation delineated by acquire and release synchronization accesses. As long as the programmer introduces enough synchronization in the program to avoid data races, the system is indistinguishable from a sequentially consistent system [1].

Pseudo-code for our recoverable LRC algorithm is given in Fig. 5. A multiple writer protocol is used, where a twin of a page is created locally on a write fault and updates are propagated to other nodes by comparing a page and its twin and encoding the result in a *diff*. Ordering is guaranteed by using vector timestamps (`vt`) [15]. Execution on processors is divided into intervals by synchronization accesses. Every processor keeps track of which intervals it is aware by updating its vector timestamp on any interaction. On an acquire, the vector timestamps are used to propagate *write notices* of all modifications to memory locations that occurred before the acquire. To limit the dependencies to acquire interactions, unlike the original LRC algorithm [7], our algorithm also sends all diffs together with the write notices. Periodic garbage collection deletes write notices have been propagated to all nodes. There is no concept of ownership of memory blocks; all the information on the contents of pages is transferred directly from the releaser to the acquirer of a lock. Locks are implemented separately from data pages. Every lock has a manager which keeps a record of its last acquirer in the `last_acq` variable.

During checkpointing, all state of the nodes is saved. However, at recovery, the `last_req` records in the lock managers, and any record of a waiter at a lock in a node are set to unknown. A `last_req` timestamp, analogous to the ownership timestamp used for pages in the previous algorithm, is used to recover unknown `last_req` records. The only interaction in our algorithm occurs during an acquire, as illustrated in Fig. 6. To make the interaction atomic, the node holding the lock replies with its id and `last_acq` timestamp when it first receives the acquire request (see Fig. 6b). When a rollback request is received, any node waiting for a lock compares the id of the rollback initiator with the id of the node holding its lock. If they are equal, it sets `must_rb`, guaranteeing that it will roll back out of the partially completed interaction.

The `last_req` timestamp scheme ensures that all dependencies with the lock manager can be ignored. The only dependency occurs when a node succeeds in acquiring a lock. However, since our LRC algorithm does not have an independent mechanism to recover all coherence information, the backward dependency cannot be eliminated. Rather than logging messages, our algorithm records a bidirectional dependency on an acquire. Figure 7 shows the results of simulations with the shared-memory address traces for our sequential consistency and LRC algorithms. The LRC algorithm reduces the dependency frequency by about a factor of 3.

5. CONCLUSIONS

Checkpointing and rollback recovery algorithms for message-passing systems are grounded in well-established theory. Research on recoverable shared memory has generally assumed a looser dependency relation for data transfers only. By using a passive server model, our work shows that this dependency pattern for shared memory can be derived from the dependency pattern for message passing, and therefore can be used in architectures where a shared-

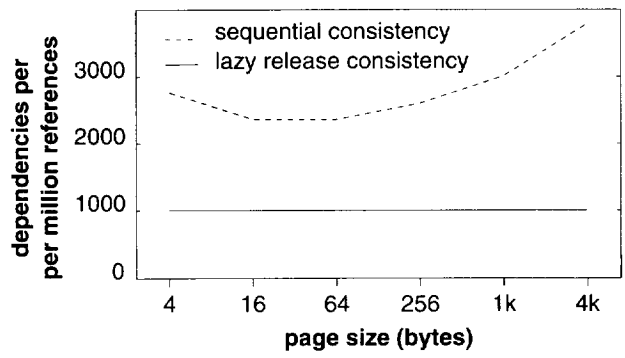

FIG. 7. Dependency frequencies with different memory consistency models.

memory image is provided via physically distributed memory. The model allows the implementation of efficient recoverable DSM algorithms. Since the need for logging is eliminated, and the potential for rollback propagation across a set of checkpoints is decreased, our method is especially applicable to periodic independent checkpointing. We applied our technique in the design of periodic checkpointing algorithms for both sequential consistency and lazy relaxed consistency memory models.

ACKNOWLEDGMENTS

Our work benefited from discussions with Alain Gefflaut at IRISA and Gaurav Suri, Yi-Min Wang, Nuno Neves, and Sujoy Basu at Illinois.

REFERENCES

1. Adve, S. V., and Hill, M. D. A unified formalization of four shared-memory models. *IEEE Trans. Parallel Distrib. Systems* **4,** 6 (June 1993), 613–624.
2. Ahmed, R. E., Frazier, R. C., and Marinos, P. M. Cache-aided rollback error recovery (CARER) algorithms for shared-memory multiprocessor systems. *Proc. 20th Int. Symp. Fault-Tolerant Comput.*, 1990, pp. 82–88.
3. Banâtre, M., *et al.* An architecture for tolerating processor failures in shared-memory multiprocessors. Tech. Rep. 707, IRISA, Rennes, France, Mar. 1993.
4. Barghava, B., and Lian, S.-R. Independent checkpointing and concurrent rollback for recovery in distributed systems—an optimistic approach. *Proc. 7th Symp. Reliable Distributed Syst.*, 1988, pp. 3–12.
5. Bernstein, P. A. Sequoia: A fault-tolerant tightly coupled multiprocessor for transaction processing. *Computer* **21,** 2 (Feb. 1988), 37–45.
6. Chandy, K. M., and Lamport, L. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Systems* **3,** 1 (Feb. 1985), 63–75.
7. Dwarkadas, S., *et al.* Evaluation of release consistent software distributed shared memory on emerging network technology. *Proc. 20th Int. Symp. Comp. Arch.*, May 1993, pp. 144–155.
8. Elnozahy, E. N., Johnson, D. B., and Zwaenepoel, W. The performance of consistent checkpointing. *Proc. 11th Symp. Reliable Distributed Syst.*, 1992, pp. 39–47.
9. Elnozahy, E. N., and Zwaenepoel, W. On the use and implementation of message logging. *Proc. 24th Int. Symp. Fault-Tolerant Comput.*, June 1994, pp. 298–307.
10. Gefflaut, A., Morin, C., and Banâtre, M. Tolerating node failures in cache only memory architectures. *Proc. Supercomputing '94.*, Nov. 1994.
11. Janakiraman, G., and Tamir, Y. Coordinated checkpointing-rollback error recovery for distributed shared memory multicomputers. *Proc. 13th Symp. Reliable Distributed Syst.*, Oct. 1994, pp. 42–51.
12. Janssens, B. and Fuchs, W. K. Relaxing consistency in recoverable distributed shared memory. *Proc. 23rd Int. Symp. Fault-Toerant Comput.*, Jun. 1993, pp. 155–163.
13. Janssens, B., and Fuchs, W. K. Reducing interprocessor dependence in recoverable distributed shared memory. *Proc. 13th Symp. Reliable Distributed Syst.*, Oct. 1994, pp. 34–41.
14. Janssens, B., and Fuchs, W. K. Recoverable distributed shared memory under sequential and relaxed consistency. Tech. Rep. CRHC-95-10, Center for Reliable and High-Performance Computing, Univ. of Illinois, Urbana, IL, May 1995.
15. Keleher, P., *et al.* Distributed shared memory on standard workstations and operating systems. *Proc. Winter Usenix Conf.*, Jan. 1994, pp. 115–131.
16. Lamport, L. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.* **C-28,** 9 (Sep. 1979), 690–691.
17. Li, K., and Hudak, P. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Systems* **7,** 4 (Nov. 1989), 321–359.
18. Litzkow, M., and Solomon, M. Supporting checkpointing and process migration outside the UNIX kernel. *Proc. Usenix Winter Conf.*, 1992.
19. Neves, N., Castro, M., and Guedes, P. A checkpoint protocol for an entry consistent shared memory system, *Proc. 13th ACM Symp. Principles Distributed Comput.*, Aug. 1994.
20. Richard, G. G., III, and Singhal, M. Using logging and asynchronous checkpointing to implement recoverable distributed shared memory. *Proc. 12th Symp. Reliable Distributed Syst.*, 1993, pp. 58–67.
21. Stunkel, C. B., Janssens, B., and Fuchs, W. K. Address tracking of parallel systems via TRAPEDS. *Microprocessors Microsystems* **16,** 5 (1992), 249–261.
22. Suri, G., Janssens, B., and Fuchs, W. K. Reduced overhead logging for rollback recovery in distributed shared memory. *Proc. 25th Int. Symp. Fault-Tolerant Comput.*, June 1995.
23. Wang, Y.-M., and Fuchs, W. K. Optimistic message logging for independent checkpointing in message-passing systems. *Proc. 11th Symp. Reliable Distributed Syst.*, 1992, pp. 147–154.
24. Wu, K.-L., and Fuchs, W. K. Recoverable distributed shared virtual memory. *IEEE Trans. Comput.* **39,** 4 (Apr. 1990), 460–469.

Received November 1, 1994; revised April 27, 1995; accepted May 8, 1995

BOB JANSSENS is a Ph.D. degree candidate in electrical and computer engineering at the University of Illinois at Urbana–Champaign. He is a research assistant in the Coordinated Science Laboratory, focusing on recoverable distributed shared-memory systems. His research interests and experience include computer architecture, operating systems, parallel and distributed computing, and fault-tolerant computing. He received B.S. and M.S. degrees in 1987 and 1991, respectively, from the Department of Electrical and Computer Engineering at the University of Illinois. He has held a summer research position at IBM T. J. Watson Research Center in Yorktown Heights, New York, and a guest scientist position at Siemens Research in Munich, Germany.

W. KENT FUCHS received the B.S.E. degree from Duke University in 1977. In 1984 he received the M. Div. degree from Trinity Evangelical Divinity School in Deerfield, Illinois, and in 1985 he received the Ph.D. degree from the University of Illinois. He is currently a professor in the Department of Electrical and Computer Engineering and the coordinated Science Laboratory at the University of Illinois. He has received several awards for his research in dependable computing. He has edited special issues of *Computer* and *IEEE Transactions on Computers.* He is currently a member of the editorial board for *IEEE Transactions on Computers* and *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.* He is a fellow of the IEEE.