Adaptive Independent Checkpointing for Reducing Rollback Propagation

Jian Xu — jx@cs.brown.edu
Robert H. B. Netzer — rn@cs.brown.edu
Dept. of Computer Science
Brown University
Box 1910
Providence, RI 02912

Abstract

Independent checkpointing is a simple technique for providing fault tolerance in distributed systems. However, it can suffer from the *domino effect*, which causes the rollback of one process to potentially propagate to others. In this paper we present an *adaptive* checkpointing algorithm to practically eliminate rollback propagation for independent checkpointing. Our algorithm is based on proofs of the conditions necessary and sufficient for a checkpoint to belong to some consistent global checkpoint, previously an open question. We characterize these conditions with a generalization of Lamport's happened-before relation called a *zigzag path*. Our algorithm tracks zigzag paths on-line and checkpoints when certain paths are detected. Experiments on an iPSC/860 hypercube show that our algorithm reduces the average rollback required to recover from any fault to less than one checkpoint interval per process, and checkpoints only 4% more often than traditional periodic checkpointing algorithms. We thus eliminate rollback propagation without the run-time overhead of coordinated checkpoints or other schemes that attempt to reduce rollback propagation.

1. Introduction

Checkpointing and rollback recovery are techniques to provide fault-tolerance in distributed systems[1, 2, 3, 4, 5, 6, 12, 13, 14, 16]. With *independent* checkpointing, each process checkpoints its state independently during normal execution, and when a fault is detected, the execution is rolled back and resumed from earlier checkpoints[1, 4, 12, 13, 16]. Because processes do not coordinate checkpoints during normal execution, independent checkpointing is simple and has low run-time overhead. However, it can have potentially high recovery overhead because of the *domino effect*[11]. To recover from a fault, the execution must be rolled back to a consistent state, but rolling back one process could result in an avalanche rollback of other processes to find a consistent state. In the worst case, all processes have to be restarted from the execution's beginning. Such unbounded rollback can prevent forward progress in the presence of faults and must be avoided. In this paper we present an *adaptive* checkpointing scheme that determines at run-time when each process should checkpoint to avoid rollback propagation. Our scheme adapts to the execution's message-passing patterns to checkpoint when certain causal paths between checkpoints are detected. Experiments on an iPSC/860 hypercube show that our scheme reduces the average rollback required to less than one checkpoint interval per process, and checkpoints only 4% more often than past independent checkpointing schemes. Our adaptive checkpointing technique virtually eliminates rollback propagation, providing a new technique to efficiently support fault-tolerant computing.

Our main theoretical result is a proof of the necessary and sufficient conditions for an arbitrary set of checkpoints to belong to the same consistent global checkpoint (a consistent checkpoint is a set of checkpoints, one per process, where none has seen a message received that another has not yet sent). Using these conditions we characterize *exactly* when a checkpoint is useful in the sense that such a consistent set exists containing it. If all checkpoints taken during an execution are useful, then we can recover from any fault without rollback propagation. Until now, the exact conditions that allow checkpoints to belong to a consistent set have been unknown. A known sufficient condition for precluding checkpoints from the same consistent set is that one *happens before* the other (in the sense that a message is sent after one checkpoint but received before the other). However, this condition is not necessary. We prove the necessary and sufficient conditions by defining the notion of a *zigzag* path, a generalization of Lamport's happened before relation.

Our adaptive checkpointing algorithm is based on our zigzag result. The algorithm tracks certain zigzag paths on-line to make run-time decisions about when to checkpoint. Each process independently makes these

This research was partly supported by ONR Contract N00014-91-J-4052 (ARPA Order 8225) and NSF grant CCR-9309311.

decisions by determining at each receive whether the incoming message completes a zigzag path, and if so, takes a new checkpoint. Adapting in this way effectively prevents zigzag paths from forming, ensuring that checkpoints are useful. Our experimental results show that with our adaptive algorithm, less than *one* checkpoint interval per process needs to be rolled back to recover from any fault. In addition, the run-time overhead of our algorithm is usually within 4% of traditional independent checkpointing algorithms (which suffer from higher rollback propagation). Thus, we achieve the benefits of no rollback propagation but without the run-time overhead of coordinated checkpoints or other schemes that attempt to reduce rollback.

This work is part of our larger effort toward adaptive strategies for replaying parallel programs[7, 8, 10].

2. Motivation and Related Work

Checkpointing and rollback recovery are techniques for recovering from faults during execution. A drawback is the domino effect, the unbounded restoration of states among the processes. Previous work either makes simplifying assumptions to avoid this effect, or incurs run-time overhead by checkpointing unnecessarily often (or coordinating checkpoints). Our goal is to study when unbounded restoration occurs, and develop an efficient checkpointing algorithm to reduce it.

Some systems assume the execution between message receives is deterministic[4, 13]. Rolling back process p (after a fault) then requires no other processes to be rolled back. When p is resumed, messages recomputed by p during recovery will be identical to those computed originally and can be ignored. When p finally reaches the point at which it originally failed, the entire execution can be restarted. We can therefore start recovery from any checkpoint and execute to a consistent state where normal execution is resumed.

However, ensuring that the reexecution of checkpoint intervals is identical to their original execution can be costly or impossible. Interactions with the external environment (such as querying an X server) must be traced so they can be reproduced during recovery. Complex interactions with the physical environment are sometimes impossible to reproduce. These problems can be avoided by always rolling back to consistent checkpoints: if when the sender of a message is rolled back, the corresponding receiver is also rolled back, normal execution can resume immediately. In general, when restarting from a consistent checkpoint, no process will be in a state that has recorded the receipt of a rolled back message. Our goal is to always rollback to a consistent checkpoint.

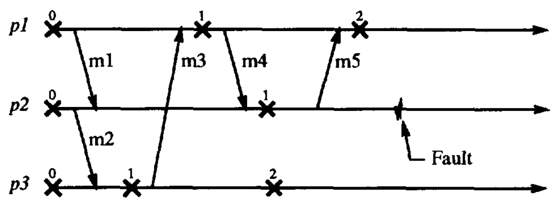

Figure 1. Domino effect when recovering from a fault. The "x"s indicate checkpoints. To recover, all processes must be rolled back to their initial checkpoints.

Two approaches exist to achieving consistent checkpoints: *independent* and *coordinated* checkpointing. With independent checkpointing, each process checkpoints independently without coordinating with others[1, 12, 16]. The consistent checkpoint for starting a recovery is computed after a fault. However, such a checkpoint does not always exist, so rolling back could incur the domino effect. For example, to recover from the fault in Figure 1, all processes have to be rolled back to their start. To roll back $p2$ to $C_{2,1}$, we have to roll back $p1$ to $C_{1,1}$ because of the undone message $m5$. We must then roll back $p2$ further to $C_{2,0}$ (because of the undone message $m4$) and so on. Russell calls this phenomenon *cyclic restoration*, when rolling back some process p to point x causes further restoration of p to a point preceding x[12].

Coordinated checkpointing eliminates cyclic restoration by guaranteeing that consistent checkpoints always exist[2, 3, 5, 6, 14]. By coordinating processes in setting up checkpoints, every checkpoint belongs to a consistent checkpoint, so these systems are domino-free. To coordinate checkpoints, some systems use explicit control messages. Other systems piggyback checkpoint markers on regular messages[2, 3, 5, 14]. Piggybacked markers are appended to outgoing messages after a process decides to initiate a new checkpoint, and processes receiving a marker take a new checkpoint (to ensure it is consistent with the process initiating the checkpoint). Since these systems depend on markers for coordination, there is no guarantee on how often a process receives markers and checkpoints. Some processes may take only a few checkpoints, and most computation performed by that process then has to be undone if that process faults. In contrast, if we allow each process to decide when to checkpoint independently, unnecessary checkpoints could be taken. In the worst case, if N processes each decide to take a new checkpoint at about the same time, the system could take $O(N^2)$ checkpoints.

Our goal is to develop an *adaptive* checkpointing algorithm for reducing rollback propagation for independent checkpointing. Independence is attractive since the complexity and overhead of coordination is not required. Our approach differs from coordinated checkpointing in that we make no guarantee that *every* checkpoint will belong to some consistent checkpoint. Instead, we concentrate on reducing rollback propagation instead of eliminating it (which could incur high overhead). Although in the worst case the domino effect could still occur, experiments with our algorithm show that this does not happen.

Wang and Fuchs have developed a transparent message scheduling technique for reducing rollback propagation for independent checkpointing[15]. They observe that the order of processing received messages can be manipulated to reduce rollback propagation. By delaying the processing of a received message until the sender passes its next checkpoint, rollback propagation can be reduced. Although this strategy is effective, message scheduling cannot always be manipulated. Instead, our approach observes that by minimizing the number of checkpoints that cannot belong to any consistent checkpoint, we can reduce rollback propagation.

3. Model

We model the execution of a message-passing program with a *program execution*, $P = \langle E, \xrightarrow{HB} \rangle$, where E is a finite set of *events* and $\xrightarrow{HB}$ is the *happened-before* relation defined over E. An event represents the execution instance of a send, receive, or checkpoint operation. We assume a fixed number of processes exist during execution, and that each process periodically checkpoints its state. We denote the i^{th} checkpoint in process p as $C_{p,i}$, and assume each process p takes an initial checkpoint $C_{p,0}$ immediately after execution begins. Also, the *checkpoint interval* $S_{p,i}$ is all the computation in process p between $C_{p,i}$ and $C_{p,i+1}$ (and includes $C_{p,i}$ but not $C_{p,i+1}$).

The happened-before relation, $\xrightarrow{HB}$, shows how events potentially affect one another, and is defined as the irreflexive transitive closure of the union of two other relations: $\xrightarrow{HB} = (\xrightarrow{XO} \cup \xrightarrow{M})^+$. The $\xrightarrow{XO}$ relation shows the order events in the same process execute. The i^{th} event in any process p (denoted $e_{p,i}$) always executes before the $i+1^{st}$ event: $e_{p,i} \xrightarrow{XO} e_{p,i+1}$. The $\xrightarrow{M}$ relation shows the message deliveries: $a \xrightarrow{M} b$ means that a sent a message b received. An event a is said to happen before b iff a could affect b because they belong to the same process or a sequence of messages was sent from a (or a following event) to b (or a preceding event).

A *global checkpoint* is a set of local checkpoints, one per process. A *consistent global checkpoint* is a global checkpoint in which no message is recorded received but not yet sent (no two checkpoints in such a checkpoint happen before one another). In a consistent global checkpoint C, we say that messages recorded sent but not received are *lost relative to C*. If we always log during execution the contents of lost messages, they can be retrieved during recovery from the log, so they do not preclude recovery. Since message logging is necessary for compensating for lost messages, we assume that it is always used. In addition, we do not assume the execution between message receives is deterministic (e.g., because of interactions with the external environment, such as querying the real-time clock). Thus, as illustrated in Section 2, we always have to restart from a consistent checkpoint to recover from a fault.

4. Zigzag Paths and Useful Checkpoints

We now present our results on *zigzag paths*. We first introduce zigzag paths as a generalization of Lamport's happened-before relation, and then use them to characterize exactly when an arbitrary set of checkpoints can all belong to the same consistent global checkpoint. We also use the zigzag condition to characterize exactly when an arbitrary checkpoint is useful in the sense that it can belong to some consistent checkpoint. In Section 5 we use these results to develop an adaptive checkpointing algorithm for reducing rollback propagation.

4.1. Zigzag Paths

A global checkpoint is consistent iff none of its component checkpoints happen before one another (i.e., they are all mutually *unordered*). However, having one checkpoint $C_{p,i}$ happen before another $C_{q,j}$ is not necessary for precluding them from the same consistent global checkpoint. They may never be able to belong together in a consistent way even if they are unordered; they can be precluded indirectly via their relationships to other processes' checkpoints. For example, in Figure 1, no checkpoint of $p2$ can be combined with both $C_{1,0}$ and $C_{3,1}$ to form a consistent checkpoint, even though $C_{1,0}$ and $C_{3,1}$ are unordered. We define the notion of a *zigzag path* to capture this situation. A zigzag path can be thought of as a generalization of Lamport's happened-before relation, which defines causal paths between checkpoints.

Definition 1

A *zigzag path* exists from $C_{p,i}$ to $C_{q,j}$ iff there are messages $m_1, m_2, \cdots, m_n$ ($n \geq 1$) such that

(1) m_1 is sent by process p after $C_{p,i}$,

(2) if m_k ($1 \leq k < n$) is received by process r, then m_{k+1} is sent by r in the same or a later check-

point interval (although m_{k+1} may be sent before *or* after m_k is received), and
(3) m_n is received by process q before $C_{q,j}$.

Checkpoint C is involved in a *zigzag cycle* iff there is a zigzag path from C to itself.

It is important to note the difference between a zigzag path and a *causal path*. In the above definition, we require either that m_{k+1} is sent after m_k is received, or that m_{k+1} is sent before m_k is received but in the same checkpoint interval. Thus, if $C_{p,i}$ happens before $C_{q,j}$ ($C_{p,i} \xrightarrow{HB} C_{q,j}$) then a causal path exists from $C_{p,i}$ to $C_{q,j}$, and by Definition 1 a zigzag path also exists. However, because we allow message m_{k+1} to be sent in a zigzag path *before* message m_k is received (condition (2) in Definition 1), the zigzag path may not be a causal path (and thus may not cause one checkpoint to happen before the other). Figure 1 shows a zigzag path from $C_{1,0}$ to $C_{3,1}$ (formed by messages $m1$ and $m2$, the receive of $m1$ happening after the send of $m2$) which is not a causal path.

Another difference between zigzag paths and causal paths is that zigzag paths do not always represent causality, so it is possible for a zigzag path to exist from a checkpoint back to itself (called a zigzag cycle). In contrast, causal paths can never form cycles. A zigzag cycle can exist because in a zigzag path we allow a message to be sent before the previous message in the path is received (as long as the send and receive are in the same checkpoint interval). Figure 1 also illustrates two zigzag cycles involving $C_{2,1}$ and $C_{3,1}$, respectively. The one involving $C_{2,1}$ is formed by messages $m5$ and $m4$; the one involving $C_{3,1}$ is formed by messages $m3$, $m1$ and $m2$.

Zigzag paths exactly capture both the necessary and sufficient conditions for an arbitrary set S of checkpoints to belong to the same consistent global checkpoint, as described in the follow theorem (proofs appear elsewhere[9] due to limited space).

Theorem 1 (Consistency Theorem)

A set of checkpoints, S, from different processes can belong to the same consistent global checkpoint iff no checkpoint in S has a zigzag path to any other checkpoint (including itself) in S.

Intuitively, if a zigzag path exists between a pair of checkpoints, and that zigzag path is also a causal path, then the checkpoints are ordered. Because consistency requires checkpoints to be unordered, they cannot be consistent. However, even if the zigzag path is not a causal path, no consistent checkpoint can be formed that contains both checkpoints. The zigzag nature of the path causes any global checkpoint that includes the two checkpoints to be inconsistent. More specifically, there is at least one message in the zigzag path that crosses the global checkpoint, causing one component checkpoint to happen before another.

Conversely, if no zigzag path exists between any pair of checkpoints in S (including a zigzag path from a checkpoint to itself), then it is *always* possible to construct a consistent checkpoint containing all checkpoints in S. By including the first checkpoint in each process that has no zigzag path to any checkpoint in S, a consistent checkpoint is formed. The checkpoint is guaranteed to be consistent since none of its component checkpoints can happen before one another (otherwise, at least one of its component checkpoints has a zigzag path to one of the checkpoints in S).

4.2. Useful Checkpoints

If a checkpoint $C_{p,i}$ cannot belong to any consistent global checkpoint, then if a fault occurs during its interval (i.e., interval $S_{p,i}$), we cannot recover by simply rolling back process p to $C_{p,i}$. Instead, a cyclic restoration is required, since we must roll back p to find an earlier checkpoint that can belong to a consistent global set of checkpoints[12]. If we could ensure that *all* checkpoints are useful in the sense that they belong to such a consistent set, then we can always roll back a faulty process to its most recent checkpoint, avoiding cyclic restoration (and guaranteeing faster recovery). Thus, checkpoints that cannot belonging to consistent checkpoints are the direct cause of cyclic restoration.

The following corollary states that a checkpoint is useful iff it is not involved in any zigzag cycle. Our adaptive checkpointing algorithm presented in the next section is based on this corollary — we reduce rollback propogation by minimizing the number of checkpoints that cannot belong to any consistent set.

Definition 2

A checkpoint is *useful* iff it belongs to some consistent global checkpoint.

Corollary 1

Checkpoint $C_{p,i}$ is useful iff it is involved in no zigzag cycle (i.e., there is no zigzag path from $C_{p,i}$ to itself).

5. Adaptive Checkpointing Algorithm

Corollary 1 shows that if we could eliminate all zigzag cycles, all checkpoints would be useful. We could then always roll back a faulty process to its most recent checkpoint, avoiding cyclic restoration. Our approach for reducing rollback propagation is to attempt to minimize the number of checkpoints that are not useful. In this section, we describe an on-line approximation for detecting zigzag cycles, and use this approximation in an adaptive checkpointing algorithm. Our adaptive algorithm directs a process to checkpoint whenever it receives a message that is about to complete a zigzag cycle. In Section 6, we present experimental results showing that our adaptive checkpointing algorithm effectively reduces rollback distance to less than one checkpoint interval per process.

5.1. Detecting Zigzag Cycles

First we discuss how to dynamically detect zigzag cycles. Ideally, we wish to perform a check at each receive operation to determine whether the incoming message completes a zigzag cycle. However, exact zigzag path detection may require knowledge of future as well as past execution. Figure 2 shows an example zigzag path from $C_{1,0}$ to $C_{1,1}$ that cannot be detected within bounded time; the path is finally formed only when message $m3$ is received, which can be arbitrarily far ahead of the path's endpoints. Our strategy is to instead detect *causal* paths, which can be easily tracked on-line (discussed later).

To detect zigzag cycles using causal tracking, we check each received message to determine if its sender is involved in a zigzag cycle. If process p receives a message during checkpoint interval $S_{p,i}$, and we find there is a causal path from $C_{p,i}$ to a checkpoint *preceding* the send, then that message completes a zigzag cycle involving the sender's checkpoint. For example, in Figure 3a there is a zigzag path from $C_{2,1}$ to itself (formed by messages $m2$ and $m1$). We can determine at the receive of $m2$ that a causal path exists from $C_{1,1}$ to $C_{2,1}$, so the zigzag cycle (completed when $m2$ is received) involving $C_{2,1}$ can be detected at this receive. Although this approximation does not detect all zigzag cycles (e.g., the zigzag cycle involving $C_{3,1}$ in Figure 1 is undetectable by our approximation), it is effective because most zigzag paths are also causal paths. The results in Section 6 imply that zigzag paths that are not causal paths (because they contain messages that are received after the next in the path is sent) are rare.

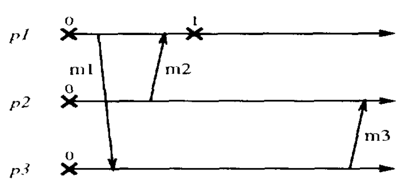

Figure 2. Zigzag path from $C_{1,0}$ to $C_{1,1}$ is not formed until $m3$ is received, arbitrarily far in the future.

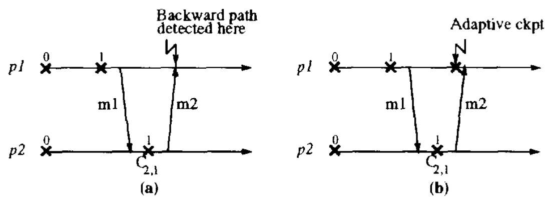

Figure 3. (a) zigzag cycle (involving $C_{2,1}$) is detected at receipt of $m2$, (b) an adaptive checkpoint is taken to prevent the cycle from forming, so the *sender's* checkpoint ($C_{2,1}$) is useful.

5.2. Algorithm

We now present our adaptive checkpointing algorithm (shown in Figure 4). This algorithm employs the approximation described above to determine (at each receive operation) whether an incoming message is about to complete a zigzag cycle that involves the sender process. Upon detecting such a cycle. our algorithm directs the receiver process to take a new checkpoint *before* it processes the message, preventing the zigzag cycle from forming (thus ensuring the sender's checkpoint is useful).

To implement the approximation, we track causal dependences among checkpoints by having each process maintain a *dependency vector* (DV). Each process keeps count of its current checkpoint interval, and its DV is a vector of checkpoint numbers, one per process. In the current DV for process p, the i^{th} entry is the index of the latest checkpoint of process i that has a causal path to the current point of p. Process p's DV entry for p itself always contains its current checkpoint number. To maintain these vectors, each process appends its DV to every outgoing message, and upon receiving a message, updates its DV by a component-wise maximum with the vector appended to the incoming message. To allow detection of zigzag cycles, each process also copies its DV to another vector ZV whenever it takes a new checkpoint. Then, when process p sends a message to a process q, it also appends the q^{th} entry of its ZV (called the *Zid*) to the message. This index shows the latest checkpoint in q (the destination of the message about to be sent) that has a causal path to $p's$ (the sending process) current checkpoint. The DV is similar to a *vector timestamp*, a common technique for tracking causality in distributed systems; the DV differs from vector timestamps in that it contains checkpoint numbers instead of event numbers.

```
1: Zid = the index piggybacked on Msg;
2: if (Zid = current checkpoint number of p) then
3:     /* Msg completes a zigzag cycle */
4:     take a new checkpoint;
5: update this process' DV from the DV piggybacked
6:     on Msg (by taking a component-wise maximum);
```

Figure 4. Adaptive checkpointing algorithm, invoked just before processing *Msg* in process *p*. *Msg* has appended to it the sender's *DV* and the index of *p*'s last checkpoint (*Zid*) with a causal path to the sender.

To locate messages that complete zigzag cycles, the algorithm is invoked at each receive operation. After process p has received a message, but before the message is processed, the algorithm checks whether there is a causal path from its current checkpoint to a checkpoint preceding the message send. As discussed earlier, such a causal path implies that a zigzag cycle exists involving the sender's checkpoint. This check is performed by comparing the *Zid* to the receiver's current checkpoint number (line 2 in Figure 4). If they are equal, then a causal path exists from the receiver's most recent checkpoint to a checkpoint that precedes the send (such as message $m2$ in Figure 3a). In this case, our algorithm directs the receiving process to take a new checkpoint (line 4) before it processes the incoming message. Otherwise, the message can be processed without first taking a checkpoint.

For example, in Figure 3b a checkpoint is taken in $p1$ after $m2$ is received but before it is allowed to be processed. This adaptively taken checkpoint prevents the zigzag cycle involving $C_{2,1}$ from forming, ensuring that it is useful. Note that taking a checkpoint in $p1$ ensures that the checkpoint already taken in $p2$ remains useful.

Our adaptive checkpointing can be integrated with other independent checkpointing algorithms. For example, we can combine it with the traditional *periodic checkpointing* algorithm. In the traditional algorithm, each process periodically takes a checkpoint every T seconds, where T is appropriately chosen to balance checkpointing overhead and rollback time (e.g., small T's incur high overhead but keep checkpoint intervals short). When we combine our algorithm with the traditional one, we obtain an *adaptive periodic checkpointing* algorithm, where a checkpoint is taken every T seconds *or* when a zigzag cycle is detected (whichever comes first). Such a combined algorithm keeps the time required to rollback a checkpoint interval to the desired level (since T becomes an upper bound on the time between checkpoints), and checkpoints at moments that attempt to keep them useful (avoiding cyclic restoration). In Section 6 we present experimental results showing that this simple algorithm virtually eliminates rollback propagation.

Another possible modification to the algorithm is to also impose a *lower* bound on the checkpoint period (e.g., to prevent from checkpointing more often than every L seconds, for a given L). However, the experimental results in Section 6 show that even if we do not impose such a lower bound, our algorithm checkpoints only 4% more frequently than the traditional algorithm. Thus, although in the worst case our algorithm could checkpoint frequently, this worst case does not occur in practice. This behavior is in contrast to some domino-free systems [2, 14] that take (potentially) many checkpoints to guarantee domino-free recovery. We virtually avoid the domino effect with little increase in checkpoint overhead.

6. Experimental Results

To measure the effectiveness of our adaptive checkpointing algorithm at reducing rollback propagation, we analyzed executions of six message-passing programs. First, we measured the average rollback required to recover from a fault. We compared the average rollback when using checkpoints taken by our adaptive algorithm to those taken by the traditional periodic checkpointing algorithm. This comparison shows how effectively our adaptive algorithm reduces rollback propagation. Second, we measured the increase in the number of checkpoints taken by our adaptive algorithm (over the traditional algorithm). This increase indicates the run-time overhead of our algorithm. We found that our adaptive algorithm reduces rollback distance to less than one checkpoint interval per process and it requires only 4% more checkpoints than the traditional algorithm. These results suggest that our adaptive checkpointing technique virtually eliminates rollback propagation, providing a new technique to support fault-tolerant computing.

Our six test programs (provided by colleagues) are developed on a 128-node Intel iPSC/860 hypercube. Each program was run once on a 16-node subcube with an instrumented message-passing library to provide traces for simulation. These programs ran between 72 and 1,135 seconds and passed between 13,560 and 370,000 messages. Since the checkpoint frequency has an impact on the performance of checkpointing algorithms, we varied the inter-checkpoint period from 10% of the total execution time to 30% (other checkpoint periods are less interesting since they represent the cases where there are either too few or too many checkpoints in a process). Varying the period allows us to fully understand the behavior of our algorithms. For example, when only a few checkpoints are taken in each process, many messages are sent in each checkpoint interval, increasing the number of zigzag paths we can detect.

6.1. Effectiveness of Reducing Rollback Propagation

Our first experiment measured the effectiveness of our checkpointing algorithm at reducing rollback propagation. We measured the average rollback (per process) required to recover from a fault when using the checkpoints our algorithm takes. We compared this average (shown in Figure 6) to the rollback incurred by the traditional periodic checkpointing algorithm (shown in Figure 5). The average rollback distance is the total number of rolled-back checkpoint intervals required to recover from a fault, averaged over all possible fault points in the execution. This distance shows how far back we must go to find a consistent checkpoint to recover from a fault. The distance also indicates how many checkpoint intervals need to be reexecuted during recovery and is directly proportional to the recovery time.

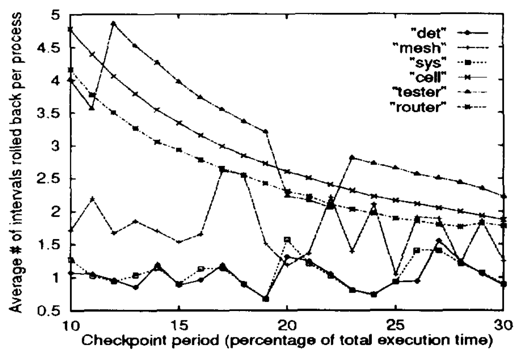

Figure 5. Average number of intervals (per process) that must be rolled back to recover from a fault when using the traditional periodic checkpointing algorithm.

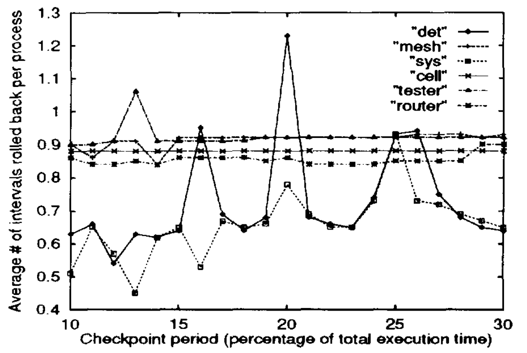

Figure 6. Average number of intervals (per process) that must be rolled back to recover from a fault when using our adaptive checkpointing algorithm.

Figure 5 shows that with traditional periodic checkpointing, the domino effect occurs; the average rollback was typically around 2.5 checkpoint intervals per process. In our implementation of the traditional algorithm, each process checkpointed at about the same time (because each process checkpoints every T seconds), but even these nearly synchronized checkpoints did not eliminate domino effects. This effect became more severe when the checkpoint period was small, since the execution was then divided into many checkpoint intervals (increasing the opportunity for zigzag cycles). In contrast, with our adaptive period checkpointing algorithm, the domino effect virtually disappears. Not only is the average rollback (per process) greatly reduced, but it is reduced to the point where less than one checkpoint interval (per process) needs to be rolled back. This performance is comparable to coordinated checkpointing (where rollback is never greater than one) but is achieved without the run-time overhead of coordinating interacting processes.

6.2. Run-Time Overhead of Adaptive Checkpointing

Our second experiment examined our algorithm's run-time overhead. There are two sources of overhead. First, each process maintains a dependence vector, and checks at each receive for zigzag cycles. Second, overhead is incurred by checkpointing. Since there is evidence that maintaining vector timestamps and performing simple checks introduce low overhead[7], we concentrate on the checkpointing overhead. We measured the percentage increase in the number of checkpoints taken by our adaptive algorithm over the traditional periodic checkpointing algorithm (for the same checkpoint period). For a checkpoint period of T, the traditional algorithm checkpoints each process every T seconds, and our algorithm checkpoints at least every T seconds. The percentage increase shows the additional run-time overhead we incur. Figure 7 shows that except for the *tester* program, our adaptive algorithm checkpoints less than 4% more often than the traditional periodic checkpointing algorithm. Since in practice the checkpoint periods are usually large, we consider this overhead acceptable.

7. Conclusions

In this paper we presented an adaptive checkpointing algorithm that practically eliminates rollback propagation for independent checkpointing schemes. We based our algorithm on proofs of the conditions necessary and sufficient for showing that a checkpoint can belong to a consistent global checkpoint, previously an open question. Our algorithm checkpoints a process whenever a zigzag cycle is about to be formed, preventing it from forming. Experiments show that our adaptive checkpointing elimi-

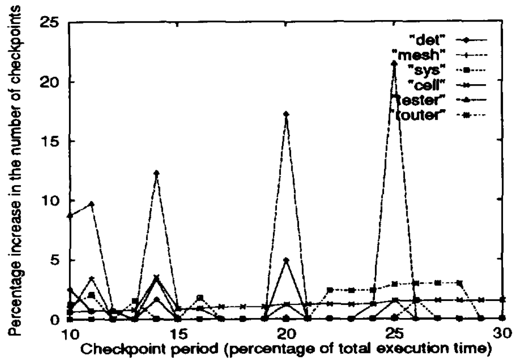

Figure 7. Percent increase in number of checkpoints taken by our adaptive checkpointing algorithm over the traditional periodic checkpointing algorithm.

nates rollback propagation at only a small increase in checkpoint frequency.

Acknowledgements

We thank Yi-Min Wang for his detailed comments on an earlier draft of this paper, Willy Zwaenepoel for helpful discussions, Prith Banerjee for the CAD programs, Mark Hill for other test programs, and the anonymous referees for their valuable comments.

References

[1] B. Bhargava and S. R. Lian, "Independent checkpointing and concurrent rollback for recovery - An optimistic approach," *Proc. IEEE Symp. on Reliable Distr. System*, pp. 3-12 (1988).

[2] D. Briatico, A. Ciuffoletti, and L. Simoncini, "A Distributed Domino-Effect Free Recovery Algorithm," *4th IEEE Symp. on Reliability in Dist. Software and Database Syst.*, pp. 207-215 (1984).

[3] K. M. Chandy and L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems," *ACM Trans on Comp Syst* 3(1) pp. 63-75 (Feb, 1985).

[4] D. B. Johnson and W. Zwaenepoel, "Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing," *Journal of Algorithms* **11** pp. 462-491 (1990).

[5] K. H. Kim, J. H. You, and A. Abouelnaga, "A Scheme for Coordinated Execution of Independently Designed Recoverable Distributed Processes," *Proc. IEEE Fault-Tolerant Computing Symposium*, pp. 130-135 (1986).

[6] R. Koo and S. Toueg, "Checkpointing and rollback-recovery for distributed systems," *IEEE Trans. on Soft. Eng.* **13**(1) pp. 23-31 (Jan 1987).

[7] R.H.B. Netzer and B. P. Miller, "Optimal Tracing and Replay for Debugging Message-Passing Parallel Programs," *Supercomputing '92*, pp. 502-511 Minneapolis, MN, (November 1992).

[8] R.H.B. Netzer, "Optimal Tracing and Replay for Debugging Shared-Memory Parallel Programs," *ACM/ONR Workshop on Parallel and Distributed Debugging*, pp. 1-11 San Diego, CA, (May 1993).

[9] R.H.B. Netzer and J. Xu, "Necessary and Sufficient Conditions for Consistent global Snapshots," *Brown University Computer Science Dept. Technical Report CS-93-32*, (July 1993).

[10] R.H.B. Netzer and J. Xu, "Adaptive Message Logging for Incremental Replay of Message-Passing Programs," *Supercomputing '93*, Portland, OR, (November 1993).

[11] B. Randell, "System Structure for Software Fault Tolerance," *IEEE Trans. on Software Engineering* **SE-1**(2) pp. 220-232 (June 1975).

[12] D. Russell, "State restoration in systems of communicating processes," *IEEE Trans. on Software Engineering* **SE-6**(2) pp. 183-194 (Mar 1980).

[13] R. E. Strom and S. Yemini, "Optimistic Recovery in Distributed Systems," *ACM Trans. on Computer Systems* **3** pp. 204-226 (August 1985).

[14] K. Venkatesh, T. Radhakrishnan, and H. F. Li, "Optimal Checkpointing and Local Recording for Domino-Free Rollback Recovery," *Information Processing Letter* **25** pp. 295--303 (July 1987).

[15] Y. M. Wang and W. K. Fuchs, "Scheduling Message Processing for Reducing Rollback Propagation," *Proc. IEEE Fault-Tolerant Computing Symposium*, pp. 204-211 (July 1992).

[16] Y. M. Wang and W. K. Fuchs, "Optimistic message logging for independent checkpointing in message-passing systems," *IEEE Symp. on Reliable Distributed Syst.*, pp. 147-154 (Oct 1992).