ON PARALLEL AND DISTRIBUTED SYSTEMS

2

AN EVOLUTIONARY APPROACH TO CONCURRENT CHECKPOINTING [1]

Junsheng Long, Bob Janssens, and W. Kent Fuchs

Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
1101 West Springfield Ave.
Urbana, IL 61801

Principal contact: Junsheng Long

long@crhc.uiuc.edu
(217) 333-8294
FAX: (217) 244-5686

Abstract

This paper describes an evolutionary approach to establishing a consistent global recovery line for concurrent processes. Unlike globally synchronized schemes, our approach uses no agreement protocols and thus no rounds of messages to decide upon a recovery line. Unlike logging-based schemes, our approach neither stores the messages exchanged between concurrent processes, nor constructs message dependence graphs to determine a recovery line. In contrast to communication-synchronized schemes, our technique reduces overhead by not always synchronizing computation with checkpointing and by allowing a potentially inconsistent recovery line temporarily.

Evolutionary concurrent checkpointing periodically starts a checkpointing session by checkpointing each process locally. As the checkpointing session progresses, the initial checkpoints are updated according to the communication between the concurrent processes. This local checkpoint updating guarantees that the recovery line evolves into a consistent line. Evolutionary concurrent checkpointing can be applied to message-based multicomputer systems, shared virtual memory systems, and shared memory multiprocessors. We evaluate the performance of our approach using execution traces from a hypercube multicomputer and a shared-memory multiprocessor.

Key Words: *fault tolerant computing, checkpointing, and rollback error recovery.*

[1]This research is supported in part by the Department of the Navy and managed by the Office of the Chief of Naval Research under Contract N00014-91-J-1283, and in part by the National Aeronautics and Space Administration (NASA) under Contract NAG 1-613, in cooperation with the Illinois Computer Laboratory for Aerospace Systems and Software (ICLASS).

I. Introduction

Rollback using checkpoints is the classic backward recovery method used in fault-tolerant computer systems. In concurrent computation, independent checkpointing of individual processes is inadequate since communication between processes may cause propagation of rollbacks, potentially resulting in a domino effect [1–3]. To avoid rollback propagation, it should always be possible to roll back to a consistent recovery line. A consistent recovery line is a set of checkpoints, one for each process, across which there is no interprocess communication.

One approach to obtaining a consistent recovery line is to stop computation and synchronize the concurrent processes at an agreed-upon point in time [3–5]. In some tightly coupled systems it is possible to synchronize processors instantaneously [5]. However, typically this global synchronization requires rounds of message exchanges. An alternative is to synchronize checkpointing with communication [5–8]. Whenever two processes communicate, checkpointing can be invoked in both processes. The recovery line is always consistent since there is no communication across the corresponding checkpoints. During recovery, only the individual process encountering the error rolls back, because the faulty process has not communicated since its last checkpoint. In the communication synchronized approach, checkpointing frequency is fixed and is dependent on communication patterns.

Message logging is often used to reduce the cost of checkpoint operations [6, 9, 10]. Instead of resending past messages during recovery, message logs are replayed to produce them. Optimistic logging [11–15], which can be viewed as communication synchronized checkpointing with deferred logging operations, is often used. Deferred logging often requires complex methods to keep message dependence information for uncommitted message logs and to manage interleaving message retries and message replays during recovery

Chandy and Lamport have shown that the global state of a distributed system consists of both the states of individual processes and communication channels [16]. They proposed to save individual process states independently and log all messages sent by the processes before checkpointing their states. A special marker message is broadcast to all other processes after the local process makes its checkpoint. Provided a FIFO channel, all messages before the marker message from the process are the ones that need to be logged. n broadcasting messages are required. This approach has been applied to concurrent checkpointing in distributed systems [17–19]. With a bounded communication latency and loosely synchronized clocks, the special marker messages can be eliminated [20,21]. However, restoring the original message order after rollback often requires a mechanism to determine when to replay from message logs and when to retry messages due to the interleaving of logged messages and normal messages during checkpointing.

This paper describes an evolutionary approach to concurrent checkpointing. In this approach computation periodically enters a checkpoint session, where a consistent recovery line evolves. A checkpoint session can be initiated at any computation point. Upon receiving notification of the start of a checkpoint session, each process independently takes a local checkpoint. The initial recovery line, made up of the local checkpoints, may be inconsistent since no attempt has been made to prevent communication across it. As computation progresses the local checkpoints are updated whenever there is a communication between processes, as in the communication synchronized approach. This local checkpoint updating causes the recovery line to evolve into a consistent recovery line. At the end of the checkpoint session a consistent recovery line is guaranteed and its checkpoints can be committed. The resulting global recovery line requires that all processes roll back to their previous checkpoint if an error occurs. The frequency of checkpoint sessions can be controlled, depending on the performance and reliability requirements of the system.

Our approach does not specify the mechanism by which individual checkpoints are taken. It attempts to reduce the overhead in coming to an agreement about a consistent recovery line. Therefore, it is useful only in systems where the overhead of synchronization between processors dominates the overhead of taking individual checkpoints. Other limitations to our approach are the requirements that communication is synchronized between processors and that communication latency is bounded. Many systems conform to these requirements, and ones that do not can usually be modified to conform.

The following section describes the assumptions, key ideas, and techniques of the evolutionary checkpointing algorithm. The subsequent two sections discuss the correctness and performance considerations. Section V describes application to rollback recovery in both shared-memory and distributed memory computer systems.

II. Evolutionary Checkpointing Algorithm

A. Computation Model

The computation considered in this paper consists of a number of concurrent processes that communicate through messages over a network. This model is extended later to a cache-based shared-memory system by viewing a memory access to nonlocal data as a message from the source processor that provides the data to the destination processor that receives the data.

In our communication model, messages are assumed to be synchronized: the sender is blocked until an acknowledge message is received from the receiver. Most lower layers of network models naturally provide and implement acknowledge mechanisms (e.g., Ethernet). Reliable communication requires acknowledge messages even at high levels. In distributed memory systems and network file servers, the read/write requests are in fact implemented with remote procedure calls (*RPC*) or synchronized messages [22]. Multiprocessor systems also meet this assumption since read/write accesses are atomic and synchronized. The assumption provides two advantages. First, checkpointing of a message sender can be requested by the message receiver during a checkpoint session if necessary. Second, this checkpointing request can be piggybacked on the acknowledgment at low additional cost.

Christian, and Tong *et al.* use a bounded communication latency to remove the special checkpointing marker messages in Chandy and Lamport's checkpointing scheme [16,20,21]. In this paper, we use a similar bounded communication latency for our evolutionary checkpoint scheme to determine a consistent recovery line without exchanging extra messages. We denote the communication upper bound as Δ in the paper.

In general, communication latency is nondeterministic at the user level due to message size, processes that are not ready to communicate, and underlying network characteristics. A two layer approach can be used to achieve bounded communication latency. A message server can be inserted below the user level process. The user process sends and receives messages only through its message server. The user level messages can be asynchronous and unbounded in communication latencies. However, the message server divides messages into packets to remove the uncertainty in communication latency due to message size. In many networks, proper techniques, such as priority preemptive scheduling, can guarantee a deterministic communication response for the message server [23,24]. In some systems, the message server is a natural component, such as the cache controller in shared-memory multiprocessors, and the pager in distributed memory systems [8,25]. Another approach that can be used to obtain a bounded communication latency is the timeout mechanism. Even if communication latency is unbounded, messages are delivered within a small threshold with a high probability [21]. The messages with a communication delay larger than the timeout threshold can be detected and treated as a *performance failure* [21].

A computation is divided into alternating checkpoint-free and checkpoint sessions. A checkpoint is a snapshot of the process state at the time of checkpointing. The operation of our scheme does not depend on the manner in which the checkpoints are taken, as long as the computation state at the checkpoint can be restored. Since in a checkpoint session only the last checkpoints taken on each processor are guaranteed to form a consistent recovery line, the intermediate checkpoints can be generated in local memory and do not have to written out to a stable or backup storage. Thus, checkpoint updating can be accomplished quickly by marking the process state unmutable [5,8,26]. The final checkpoints still need to be copied to stable storage. If the overhead of waiting for this copying to occur is too high, another process can be scheduled to do the copy, without blocking the computation [26]. We therefore assume the checkpoint operation time during a checkpoint session to be negligible compared to the communication delay upper bound.

A checkpointing coordinator broadcasts a *ckp_start* message to initiate a checkpoint session and a *ckp_end* message to terminate this checkpoint session. This checkpointing coordinator can be one of the participating concurrent processes. Our recovery algorithm can handle errors that cause missing ckp_start or ckp_end messages. The need to broadcast ckp_end can be eliminated by a local timer at each process. If the local clocks are loosely synchronized with a small shift, using local clocks to signal ckp_start and ckp_end is possible, similar to other schemes in the literature [20,21].

The point of time at which a process enters a checkpoint session is its entry point to the checkpoint session. Similarly, the time at which a process exits a checkpoint session is its exit point to the checkpoint session. The set of the entry points for a checkpoint session form the checkpoint session entry line, and the set of exit points for a checkpoint session form the checkpoint session exit line. The reception points of ckp_start and ckp_end form the initial entry line and exit line for the checkpoint session.

B. Approach

In order to obtain a consistent recovery line, we want to eliminate the messages that cross the recovery line by achieving three goals:

1. There is at least one local checkpoint for each process, and thus a recovery line, during a checkpoint session.
2. There are no messages exchanged across the entry line or exit line.
3. Inside the checkpoint session, messages do not cross the current potentially consistent recovery line.

To fulfill these requirements, our evolutionary checkpointing scheme makes use of the following techniques:

- Upon entering a checkpoint session, every process immediately takes a checkpoint. This guarantees that there always exists a recovery line from the beginning of the session.
- In order to eliminate messages crossing the checkpoint session entry line, the initial entry points are adjusted to *include* crossing messages in the checkpoint session.
- To remove messages crossing the exit line, the initial exit points are adjusted to *exclude* crossing messages from the checkpoint session.
- Inside the checkpoint session, checkpointing is synchronized with communication. Both the receiver and sender of a message take a new local checkpoint immediately after the communi-

```
-------------------------------------------------------------------
local variables and operation for each node:

    ckp_num:      checkpoint number (time stamp);
    ckp_session:  checkpointing in session flag
                     0 - not in a checkpointing session
                     >0 - ckp_num currently in session
    checkpoint(n): make a local checkpoint with checkpoint number n
    enter_ckp_session()  // enter a checkpoint session
        {    ckp_num++; ckp_session = ckp_num;
             checkpoint(ckp_num);
        }

Augmented message format:

    message : <ckp_num, ckp_session, normal message>;
    ack:      acknowledge: <ckp_num, normal acknowledge>
                  ckp_num - 0 : no need to checkpoint
                           >0: makes a checkpoint with
                               checkpoint number ckp_num.
-------------------------------------------------------------------
```

Figure 1. Local Variables and Operations at Each Process Node.

cation. This communication synchronized checkpoint updating leaves the exchanged message behind the recovery line and makes the recovery line evolve towards a consistent line.

- The ckp_end message is signaled 2 Δ after ckp_start. We will show that this condition prevents messages from completely bypassing the checkpoint session.

C. Detailed Description

Figure 1 describes the local data structures needed to implement evolutionary checkpointing [2]. Figures 2 and 3 describe the detailed algorithms for the message sender and receiver.

[2]The appended checkpointing information such as the checkpoint number can be eliminated if the delivery of ckp_start and ckp_end is reliable, since using mismatches in checkpoint numbers to detect the missing ckp_start and ckp_end messages is not necessary.

```
ack = send_message(msg);
if (ack.ckp_num > 0) { // need to make a local checkpoint
    if (ckp_num + 1 == ack.ckp_num) {
        // receiver already passed the entry line
        // advance local checkpoint entry point to now
        enter_ckp_session();
    } else if (ckp_num == ack.ckp_num) {
        // both sender and receiver in checkpointing session
        checkpoint(ckp_num);
    } else { // detect performance fault or missing ckp_start
             // or ckp_end msgs (Lemma 3).
        error();
    }
} else if (ack.ckp_num == 0) { // no need for local checkpointing
    if (ckp_session != 0) {
        // receiver already exits its session
        // adjust its checkpoint exit point to now.
        ckp_session = 0;
    }
} else  // impossible by the ack format
    error();
```

Figure 2. Sender Algorithm.

```
-------------------------------------------------------------------------

When a ckp_start is received,

    if (ckp_start.ckp_num == ckp_num+1 && ckp_session == 0)
        enter_ckp_session(); // a new ckp_session
    else if (ckp_start.ckp_num == ckp_num && ckp_session == 1)
        // ignore it; its entry point has been adjusted before.
    else error(); // detect missing ckp_start/end msgs.

When a ckp_end is received,

    if (ckp_end.ckp_num == ckp_num) {
        if (ckp_session == 1) ckp_session = 0; // exit the session
        // else ignore it; its exit point has been adjusted before.
    } else error(); // detect missing ckp_start/end msgs.

when a message is received,

    if (ckp_session) { // checkpointing in session
        if (msg.ckp_num + 1 == ckp_num) {
            // sender yet to enter checkpointing session;
            ack_back(ckp_num); // ask sender to checkpoint
            checkpoint(ckp_num);
        } else if (msg.ckp_num == ckp_num) {
            // both sender and receiver in the ckp session;
            if (msg.ckp_session == ckp_num) {
                // sender still in checkpointing session;
                // both update their local checkpoints.
                ack_back(ckp_num);
                checkpoint(ckp_num);
            } else {
                // sender exited the session; no checkpoint update.
                ack_back(0);
            }
        } else // detects missing ckp_start/end msgs
            error():
    } else { // out of the checkpointing session
        if (ckp_num == msg.ckp_num) {
            // both sender and receiver out of the session;
            // no local checkpointing asked for the sender.
            ack_back(0);
        } else if (ckp_num+1 == msg.ckp_num) {
            // receiver yet to enter the checkpointing session;
            // advance the local entry point to now.
            ckp_num++;
            ckp_session = ckp_num;
            ack_back(ckp_num);
            checkpoint(ckp_num);
        } else // performance fault or ckp_start/end missing:
              // msg crosses the ckp session (lemma 3).
            error();
    }
-------------------------------------------------------------------------
```

Figure 3. Receiver Algorithm.

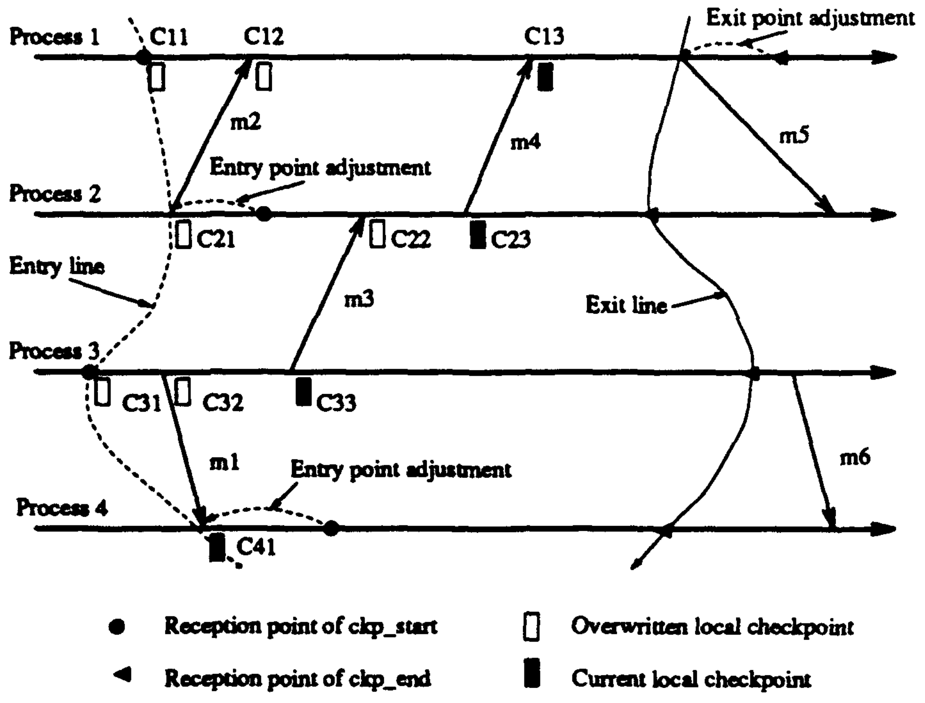

Figure 4. Checkpoint Session and Recovery Lines.

C.1. Entering a Checkpoint Session

Upon receiving the ckp_start signal from the checkpointing coordinator, a process enters the checkpoint session and takes a local checkpoint by saving its process state. The different session entry points of the processes form the session entry line. The initial set of local checkpoints provides a potentially inconsistent initial recovery line. For example, the initial recovery line {C11, C21, C31, C41} in Figure 4 is not consistent since if process 2 is restarted from C21, it will not resend message m2, while if process 1 is restarted from C11, it will wait for this message.

C.2. Adjusting the Entry Points

If a process that has not entered the checkpoint session exchanges a message with a process already in the checkpoint session, it will not wait for the ckp_start to enter the checkpoint session. Instead it marks its entry point as if it has received the ckp_start before the message exchange and takes its initial checkpoint right after the exchange. When ckp_start is subsequently received, it is ignored. The adjustment of an entry point from the ckp_start reception point is demonstrated in Figure 4 where message m1 crosses the original entry line (the ckp_start reception line). By moving the entry point of process 4 to the point of communication, m1 is included in the checkpoint session. Therefore, process 4 makes its initial checkpoint C41 while process 3 updates its local checkpoint C31 with C32 at the request piggybacked on the acknowledge from process 4.

C.3. Updating Local Checkpoints

If a message is exchanged between two processes inside a checkpoint session, the receiver updates its local checkpoint to the current state. Meanwhile, it also piggybacks on the acknowledge of the message a request to the sender to update its local checkpoint. In Figure 4, the message m3 between processes 2 and 3 leads to the updating of C21 and C32, to C22 and C33, respectively. This checkpoint updating makes the recovery line evolve to consistency by including the exchanged message in the checkpointed state. For example, when process 2 updates C21 with C22 and process 3 updates C32 with C33, they include m3 in the checkpointed state. The new recovery line {C12, C22, C33, C41} is consistent.

In our scheme, a sender takes a local checkpoint when the acknowledge from the receiver requires it to. During a checkpoint session, local checkpointing is synchronized with the computation, as in communication synchronized checkpointing schemes [5,8,27,28]. Our approach can be viewed as a scheme that samples, during checkpoint sessions, a small fraction of checkpoints made by the communication synchronized schemes.

C.4. Adjusting the Exit Points

When a process is in a checkpoint session and receives a ckp_end, it exits the checkpoint session. If the process exchanges a message with a process that has exited the session, the process marks its exit point and ignores the subsequently received ckp_end. No checkpoint updating is performed. In this manner, the message exchange is *excluded* from the checkpoint session. It can be shown that when the processes reach the exit line, the current local checkpoints form a consistent recovery line. Message m5 in Figure 4 illustrates a case of exit point adjustment. Message m5 crosses the original exit line (the ckp_end reception line). When process 1 is notified through the message acknowledgement that the receiver of m5 is already outside the checkpoint session, it immediately moves its exit point to the point of communication and exits the checkpoint session. In this manner, m5 is excluded from the checkpoint session. When the last process leaves the checkpoint session, the exit line is complete, and the set of current local checkpoints ({C13, C23, C33, C14} in the example) comprises a consistent recovery line.

C.5. Avoiding Bypassing Messages

Provided that communication delay is bounded by Δ, broadcasting ckp_end 2Δ after the ckp_start broadcast guarantees that no messages bypass the checkpoint session. That is, there is no message that originates before a checkpoint session and is received after the checkpoint session, such as message m4 in Figure 5. If a message were allowed to bypass the checkpoint session, some checkpoints of the resulting recovery line might be missed. For example, process 3 interacts with process 2 after passing its exit point but before receiving m4 from process 4. Process 3 has already exited the checkpoint session, thus the exit point of process 2 is adjusted and the local checkpoints are not updated to C24 and C33. Even if we let process 3 update its local checkpoint after receiving

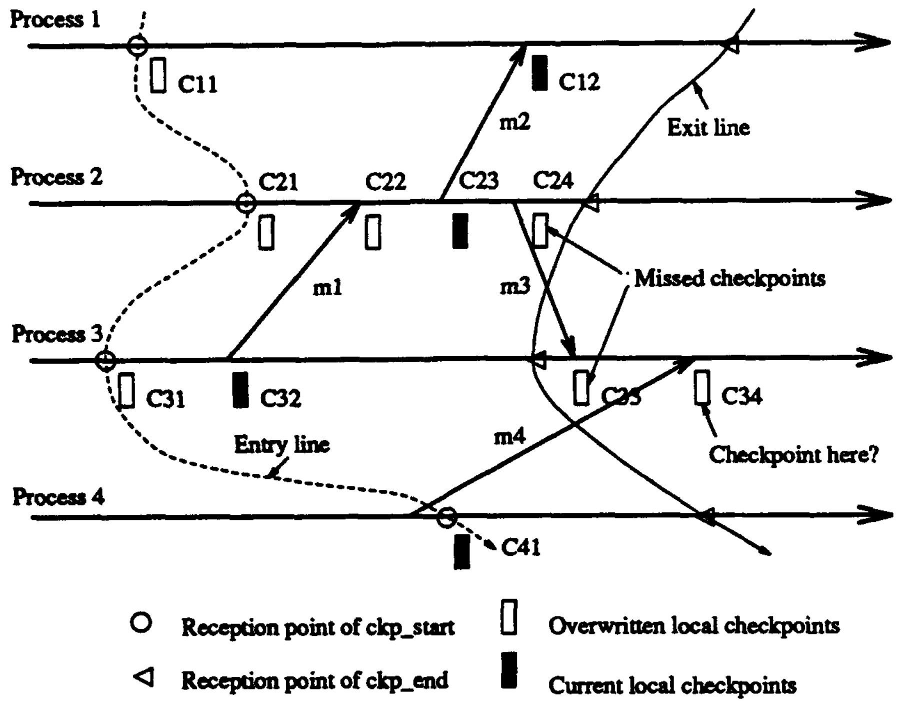

Figure 5. Example of a Message Bypassing a Checkpoint Session.

m4, the missing checkpoints (C24, C33) make the current recovery line ({C12, C23, C34, C41}) inconsistent, since there is a message exchange across the exit line (m3).

C.6. Handling Missing Checkpointing Messages

Missing ckp_start and ckp_end messages can be detected by the evolutionary algorithms in Figures 2 and 3. Suppose process j missed a ckp_start message for a checkpoint session. If it communicates with another process already in the checkpoint session, this missing ckp_start does not affect the checkpointing algorithm, since this message is ignored due to the entry point adjustment. If process j receives a ckp_end message, its local checkpoint number is mismatched with the checkpoint number in the ckp_end message. This detects a missing ckp_start. In general, mismatches in the local checkpoint number and the checkpoint number in messages detects errors in message delivery in the evolutionary scheme.

III. Correctness

There exist an entry line and an exit line for each checkpoint session. Every process receives a ckp_start message for each checkpoint session. The only time the entry point is not the reception point of the ckp_start is when the entry point has been adjusted to an earlier point due to a message exchange between a process that is yet to enter the checkpoint session with another process already in the checkpoint session. So there is always an entry line at or before the ckp_start reception line. Similarly, there is always a session exit line at or before the ckp_end reception line. Since the ckp_end is broadcast 2Δ after the ckp_start, the ckp_end will be received by each process after the ckp_start. If the ckp_end reception point is the exit point for a process, the exit point is behind its corresponding ckp_start reception point and thus its entry point. If the exit point has been adjusted, the process must be in the checkpoint session when the adjustment occurs, and the exit point will not be adjusted ahead its entry point. Thus, the exit line is always behind the corresponding entry line. Therefore,

Lemma 1: Given the algorithm in Figures 2 and 3, there is an entry line followed by an exit line for each checkpoint session.

We will show that there is a recovery line after an entry line. That is, every process will have a local checkpoint after this line. Upon receiving a ckp_start, a process either makes a local checkpoint or ignores this ckp_start. According to the algorithm, the process ignores a ckp_start only when its entry point has been adjusted to an earlier time than the ckp_start reception point. As a part of its entry point adjustment, a local checkpoint is made for this process. This proves the following lemma.

Lemma 2: Given the algorithm in Figures 2 and 3, there is a recovery line after the last process passes the entry line of a checkpoint session.

Lemma 1 and Lemma 2 imply that there is a recovery line when the last process passes the exit line. Before we show that this recovery line is consistent, we first prove a lemma which assures that the minimum time difference between the entry point for one process and the exit point for any other process is at least one Δ. This condition assures that no messages bypass the checkpoint session. That is, a message originated before a checkpoint session will not be received after the checkpoint session and vice versa. Let (s_i, e_j) be the pair of the entry time for process i and the exit time for process j for the same checkpoint session.

Lemma 3: Given the algorithm in Figures 2 and 3, $e_j - s_i > \Delta$ for any (s_i, e_j) and $i \neq j$.

Proof: Let (s_i, e_j) be the pair with the minimum difference among all the possible pairs. There are only two cases possible. (1) e_j is the time of ckp_end reception by process j. According to Lemma 1, s_i is either the reception time of the ckp_start at process i or an earlier point than the reception time due to an entry point adjustment. Therefore, we need only to prove that the time difference between e_j and the reception time of the ckp_start at process i is greater than Δ. Since any message will be delivered within Δ and the ckp_end is broadcast 2Δ after the ckp_start, no process will receive a ckp_start later than one Δ after the broadcast of ckp_start and no process will receive a ckp_end before 2Δ after the broadcast of ckp_start. Therefore, $e_j - s_i > \Delta$. (2) e_j is not the time of ckp_end reception by process j. This case occurs only when process j receives a message from a process (e.g., process k) that passed the exit line when process j was still in the checkpoint session (i.e., it is yet to receive the ckp_end). According to the algorithm, j will adjust its exit point from its ckp_end reception point. This case is impossible. Otherwise, e_j cannot be in the minimum pair of (s_i, e_j), since process k has passed the exit line before process j $(e_k < e_j)$. This contradicts that (s_i, e_j) is the smallest pair of all the possible pairs that include (s_i, e_k). $\square$

A recovery line is consistent if all messages sent before (after) a consistent recovery line are received before (after) this line. That is, there are no message exchanges across a consistent recovery line. This guarantees that any rollback will not need to cross this line and thus eliminates the domino effect of rollback propagation.

Theorem 1: Given the algorithms in Figures 2 and 3, the set of the current local checkpoints forms a consistent recovery line when the last process exits a checkpoint session.

Proof: According to Lemmas 1 and 2, there is a recovery line when the last process exits a checkpoint session. Suppose a process receives a message after this exit line. The sender cannot be in the state prior to the checkpoint session, since the sender has yet to pass its entry point and thus its exit point. This implies that the exit line is still incomplete. The sender cannot be in the checkpoint session either; otherwise, the algorithm requires the receiver to ask the sender to adjust its exit point to exclude the message exchange from this checkpoint session. Therefore, the sender must be after its exit point. Suppose a process sends a message after the exit line. The receiver cannot be in the state prior to the checkpoint session; otherwise, this gives an incomplete exit line. The receiver cannot be in the checkpoint session either, since the algorithm will adjust the receiver's exit point to exclude the message from this checkpoint session. Thus, the receiver must have passed the exit line. Therefore, there is no message exchange across the exit line, and the recovery line after the exit line is consistent. Since there is no local checkpoint updating after the exit line, this consistent line remains until the next checkpoint session. □

IV. Performance Considerations

A. Convergence Time

We define the convergence time of our evolutionary checkpointing scheme as the time for a potentially inconsistent recovery line to evolve into a consistent recovery line. This parameter determines the minimum length of a checkpointing session. More importantly, it also affects the overhead involved in our scheme since the longer the convergence time, the more local checkpoint updating is likely. The following theorem gives an upper bound on the convergence time of our algorithm.

Theorem 2: Given the algorithm in Figures 2 and 3, the convergence time of the recovery line during a checkpointing session is less than 3 Δ.

Proof: According to the algorithm, the first process enters the checkpointing session upon receiving a ckp_start, which occurs no earlier than the ˋ:p_start broadcasting time. The last process to receive a ckp_end will receive it no later than 3 Δ after the ckp_start broadcast since ckp_end is broadcast 2 Δ after ckp_start, and ckp_end will be delivered to every process within Δ. According to the proof of Lemma 1, the exit line forms before the ckp_end reception line because the exit point is either the reception point of a ckp_end or at an earlier time than the reception point due to the exit point adjustment. Theorem 1 guarantees a consistent recovery line after all processes pass the exit line. Therefore, there is a consistent recovery line no later than 3 Δ after the ckp_start is broadcast. Thus the convergence time is less than 3 Δ. □

B. Run-time Overhead

The expected run-time overhead (C_k) can be simply expressed in terms of the frequency of checkpoint sessions (n), checkpointing time per session (C_s), rollback probability (p_r) and recovery overhead (C_r) as

$$C_k = nC_s + np_rC_r$$

$$C_s = C_{init} + N_{update}C_{update}$$

where C_{init} is the checkpoint cost of the initial checkpoint made at the entry of a checkpoint session; N_{update} and C_{update} are the frequency and the overhead of local checkpoint updating respectively. The first term, nC_s, in C_k represents the checkpointing overhead, while the second term, np_rC_r, is the recovery overhead.

Given the frequency (n) and length (convergence time) of checkpoint sessions, the checkpointing overhead, C_s, depends on the frequency and overhead of local checkpoint updating. The number of times that a local checkpoint is updated is computation specific. Every time a message is sent - received inside a checkpoint session, the local checkpoint has to be updated. Given the limited convergence time, the number of updates is likely to be limited.

To determine the number of checkpoint updates that can be expected in a distributed memory system we traced the communication patterns of eight parallel programs on an 8-node Intel IPSC/2 hypercube (Table 1). We took random snapshots of the computation with lengths varying from 10 to 500 msec. On the IPSC/2, message latency averages about 1 msec/K [29]. Our snapshot lengths therefore represent conservative estimates of the session lengths that could be chosen for the IPSC/2. For every program and snapshot length we performed 1000 random trials.

Table 2 shows the frequency of messages (which corresponds to the number of checkpoint updates) for different session lengths. For the numerical programs (**fft**, **mult**, **gauss**, **qr** and **navier**), the average number of messages transmitted or received is less than 3. However, the number of messages in a particular checkpoint session can be as high as 244 (**qr**). Typically messages in the hypercube occur in bursts when data are distributed to and collected from the nodes. If this

Table 1. Hypercube Program Traces.

Program	Description	Execution Time (msec)	Message # recvs.	Message # sends	Message avg. size (bytes)
fft	fast Fourier transform	51363	110	60	97.7K
mult	matrix multiplication	4160	48	43	18.5K
gauss	Gauss elimination	47222	6764	2706	622.8
qr	QR factorization	3590	4105	4098	508.9
navier	fluid flow simulator	21315	118	118	22.7
tester	circuit test generator	123339	13215	10786	264.3
cell	circuit cell placement	50645	42619	42764	31.7
router	VLSI channel router	435648	371700	371650	18.9

Table 2. Communication Characteristics of Hypercube Traces.

Trace	Session length (msec)	Messages max #	Messages average #	Trace	Session length (msec)	Messages max #	Messages average #
	10	1	0.12		10	4	0.11
fft	50	2	0.12	**navier**	50	4	0.11
	100	2	0.12		100	5	0.12
	500	3	0.13		500	10	0.13
	10	4	0.05		10	15	0.34
mult	50	4	0.06	**tester**	50	44	0.98
	100	4	0.06		100	67	1.57
	500	4	0.07		500	100	4.16
	10	10	0.33		10	32	2.05
gauss	50	40	0.81	**cell**	50	104	7.76
	100	80	1.05		100	149	13.48
	500	243	2.24		500	451	28.21
	10	7	0.90		10	30	2.56
qr	50	30	1.92	**router**	50	123	10.58
	100	58	1.85		100	213	21.10
	500	244	2.34		500	505	100.55

is the case, compiler-assisted techniques that detect communication bursts in programs and plan checkpoint session accordingly could be used to decrease the number of checkpoints in a session [30,31].

The overhead of updating a local checkpoint varies with the checkpointing mechanism used. If a new complete state is saved as the checkpoint update, C_{update} is the same as the initial checkpoint cost, C_{init} [30,32]. If only the change in state since the last checkpoint is needed to update the checkpoint (e.g., flushing dirty pages in a virtual memory system), C_{update} is likely to be smaller than C_{init} [8]. If local checkpoint updating is implemented with logging messages, C_{update} is the cost of message logging. In the above hypercube example, message logging may be appropriate for high message density programs such as **router**.

Recovery overhead, C_r, is related to the reprocessing time after recovery, which on the average is one-half of the checkpoint interval. Studies on checkpoint placement have shown that the rollback probability, p_r, is typically small enough to ensure low recovery overhead (np_rC_r) compared to checkpointing overhead (nC_s), even when the checkpoint interval and/or recovery cost are large. Schemes with an inherent high checkpoint frequency fail to take advantage of the benefits of making checkpoint intervals large. In the evolutionary approach, the checkpoint interval can be chosen as large as necessary to reduce checkpointing overhead [33–36].

C. Memory Overhead

The storage requirement for the evolutionary checkpointing is two global checkpoints, one for the last committed checkpoint and one for the current working buffer for the uncommitted checkpoint. For virtual memory-based systems, the working buffer can be set copy-on-write to the committed checkpoint. The working space is split with the committed checkpoint only when a modification is needed. After the current checkpoint is committed, the space for the old committed checkpoint can be switched to the working space.

V. APPLICATIONS TO SHARED MEMORY SYSTEMS

Recently there has been an active research interest in recoverable shared-memory and shared virtual memory computer systems [5, 8, 27, 28, 37–39]. Both globally synchronized and communication-synchronized approaches have been applied to these systems. The main drawback of these schemes is uncontrollable checkpointing [27]. In this section we will demonstrate how evolutionary checkpointing can be adopted to these situations.

A. Recovery in Cache-Based Multiprocessor Systems

In cache-based systems, cache-based rollback error recovery can be used to recover from transient processor errors [40]. In this recovery scheme, the checkpoint state is kept in the main memory, those dirty cache blocks that have not been modified since the last checkpoint, and the processor registers. A processor takes a checkpoint whenever it is necessary to replace a dirty block in its cache. At a checkpoint, the processor registers are saved, and all dirty cache blocks are marked unchangeable. Unchangeable lines may be read, but have to be written back to memory before being written. Rollback is accomplished by simply invalidating all cache lines except the unchangeable lines, restoring the processor registers, and restarting the computation.

Wu et al. proposed a cache-based recovery method for shared-memory multiprocessor systems using the communication-synchronized approach [8]. A communication is an access to a dirty cache block from the private cache of another processor. Communication between processors induces a checkpoint on the source processor. The destination processor does not need to be checkpointed, since if it rolls back it can always acquire a new copy of the transmitted data from the source processor. The effect is similar to message logging, in that the data received are available again after an eventual rollback. Ahmed *et al.* have proposed a globally synchronized checkpointing strategy for cache-based error recovery in multiprocessors [5]. They assume that a checkpoint operation can be synchronized among all processors and takes only one cycle.

These cache-based schemes have the disadvantage that the frequency of unavoidable checkpoints, due to replacement of dirty lines, is high [27]. However, the overhead in taking a checkpoint is very low. Therefore cache-based recovery is applicable to updating the checkpoints during the checkpoint session in our evolutionary scheme in which checkpointing activities are only for a very short period.

To apply our approach to cache-based recovery, we first map our system model to the shared-memory multiprocessor model. The cache controllers serve as the message servers of our model. Caches behave as the normal caches for checkpoint free computations, and as Wu's caches during checkpoint sessions. A communication is a *read* or a *write* access to a nonlocal cache. Communication in multiprocessors is synchronized since the processor is blocked until data are accessed. The memory access time, and therefore the communication time, is also bounded.

A global interrupt can be used as the ckp_start and ckp_end broadcasting mechanism [3]. This global interrupt sets or clears the local flag ckp_session at each processor as if a ckp_start or ckp_end is broadcast. During checkpoint sessions, the checkpoint operation is synchronized with communication such as in Wu's scheme [4]. The checkpoint session can be short since the convergence time is only 3 times the maximum access time to a block present in another processor's cache. At the end of the session, the checkpoint is committed, and the cache is switched from checkpointing operation to normal operation.

A shadow paged memory is needed because the state changes between checkpoint sessions can not overwrite the committed checkpoint [28]. A copy of the memory space is used for the committed checkpoint and another for the temporary working spacing. The unchangeable cache blocks are written back to the checkpoint pages when they are replaced from the caches. A copy-on-write mapping of the working pages to the checkpoint pages may save memory and avoid unnecessary

[3]Since the global interrupts can usually be assumed to be delivered re'iably and no error detection for mismatching checkpoint session numbers is necessary, the extra checkpointing information appended to each message required by the evolutionary algorithms (Figures 1, 2 and 3) can be eliminated.

[4]In a remote memory access, a source processor that provides data and a destination processor that initiates the request for accessing the data can be distinguished. The checkpoint operation at the destination processor can be eliminated since the source processor backs up the data requested in its local checkpoint and the destination processor can retry the access and acquire the data from the checkpoint.

memory copying. A rollback simply invalidates all cache blocks except unchangeable blocks and restarts all processors from the committed checkpoint.

Five parallel program traces running on seven processors of an 8-processor Encore Multimax 510 were used to evaluate this evolutionary scheme [27]. Program **tgen** is a test generator; **fsim** is a fault simulator; **pace** is a circuit extractor; **phigure** is a global router, and **gravsim** is an N-body collision simulator. Each benchmark program runs for about 10 seconds. At least 80 million references are traced in each applications [27]. The caches used are 64 K two-way set associate caches with 32-byte blocks. To apply our evolutionary scheme, we need to estimate the maximum access time for the Encore Multimax 510. The longest access is the cache miss that acquires the bus last when all processors have a miss. Since a 32-byte block takes 320 nsec (nanosecond) to fetch, the longest access is 8 $\times$ 320 or 1.28 μsec (microsecond) [41]. Thus, $\Delta \cong 1.28$ μsec. The processor is rated at 8.5 MIPS; the maximum number of instructions executed during Δ is about 8.5 $\times$ 8 $\times$ 1.28 or 87.04. Therefore, the convergence time for the Multimax 510 is about 3.84 μsec or 262 instructions. We used the number of references to determine the session length. We simulated five different session lengths of around 262 instructions: 10, 50, 100, 500, and 1000 instructions. The interval between checkpointing sessions for the evolutionary scheme can be set at any value. For our evaluation we set it at one million references. As a comparison, we evaluated the cache-based schemes of Wu et al. and Ahmed et al. The results are presented in Figures 6, 7, and 8.

The number of checkpoint updates that need to be performed during a checkpoint session depends on the amount of communication in the program. Figure 6 presents the average and maximum number of updates observed during a checkpoint session for each of the programs. For all but the largest session lengths in **fsim** and **tgen**, the average number of updates is at most one. For the longer sessions in **fsim** and **tgen**, the average is driven up by a few sessions with many

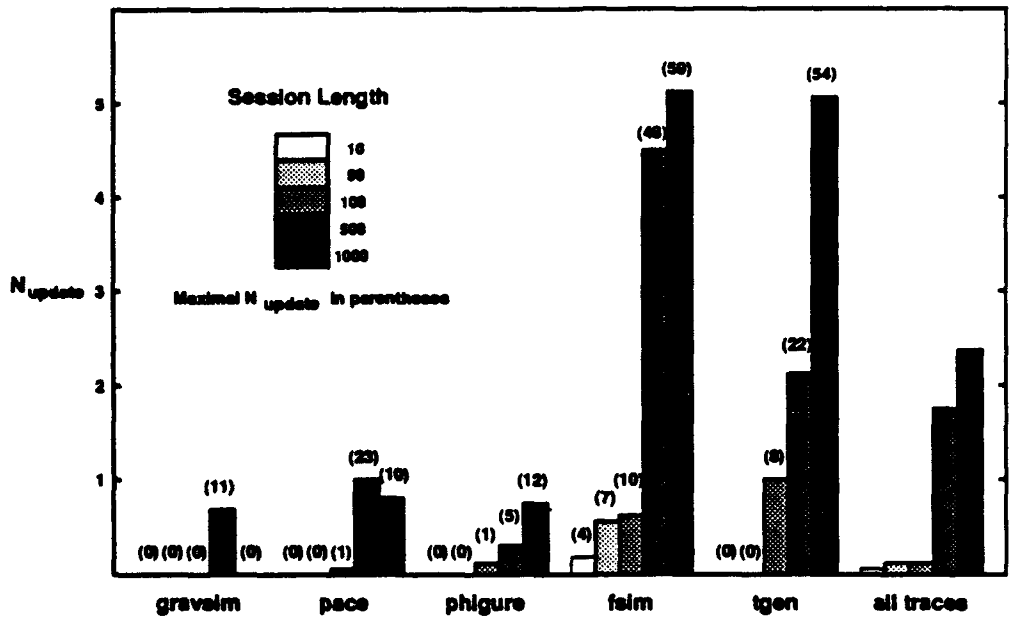

Figure 6. Average Number of Checkpoint Updates per Session.

updates. For all traces combined, however, even with a session length of 1000, the average number of updates is only around 2.5. We also found that the checkpoint size for an checkpoint update is either one or two. This indicates that local checkpoint updating only produces a limited run-time overhead.

A comparison can be made between the average checkpoint frequency for the evolutionary scheme and the other cache-based schemes. It should be noted that the checkpoint frequency for the evolutionary scheme can be controlled by adjusting the interval between sessions, while the checkpointing frequency for the other schemes is predetermined by the communication patterns of the applications we traced. For the evolutionary schemes we consider both the initial checkpoints in the session and the further updates to calculate the average frequency. The checkpoint frequencies are plotted in Figure 7. For a session interval of one million and a session length of 500, the checkpoint frequency varies between 1 and 2.5 per million accesses. On the other hand, the fre-

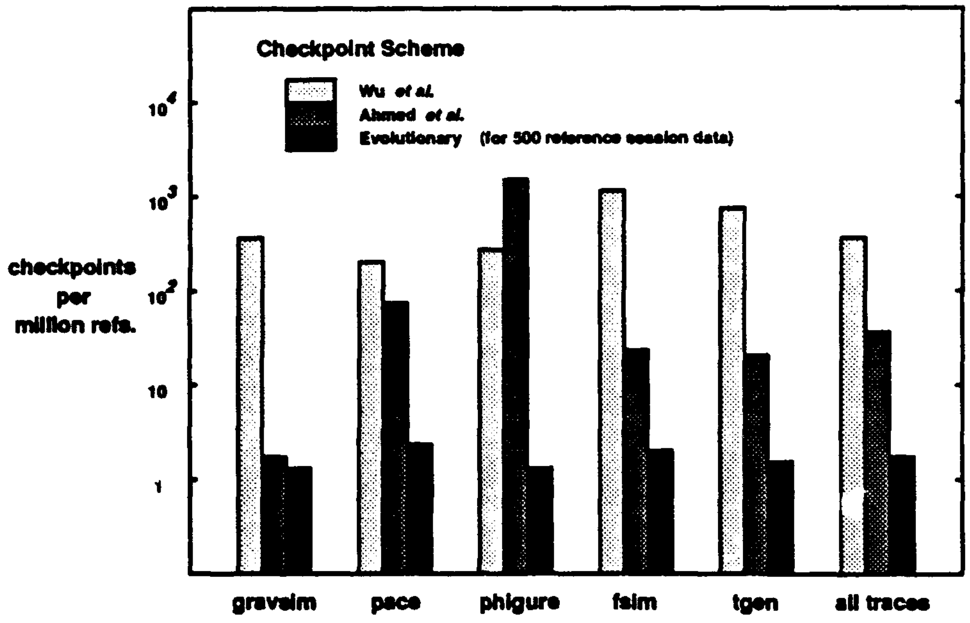

Figure 7. Scheme Comparison: Checkpoint Frequency.

quency for the globally synchronized scheme varies between 1.7 and 1500 per million accesses, and the frequency for the communication-synchronized scheme varies between 200 and 1000 per million references. The overhead of cache-based checkpointing depends on the number of cache blocks that are marked unchangeable (the checkpoint size) since extra cycles are needed to write these blocks back before they can be used. Figure 8 presents the sum of the sizes for all checkpoints during the execution of the program. This total checkpoint size is about an order of magnitude smaller for the evolutionary scheme than for the other schemes [5]. All the data show that the evolutionary scheme can provide checkpointing with a more controllable frequency and at a lower cost than previous schemes.

[5]It may be worth noticing that the total checkpoint size is basically determined by the number of checkpoint sessions and the size of the initial checkpoints in each checkpoint session, since the number and size of checkpoint updates in the evolutionary scheme are limited. The number of checkpoint sessions can be controlled with proper placements of checkpoint sessions.

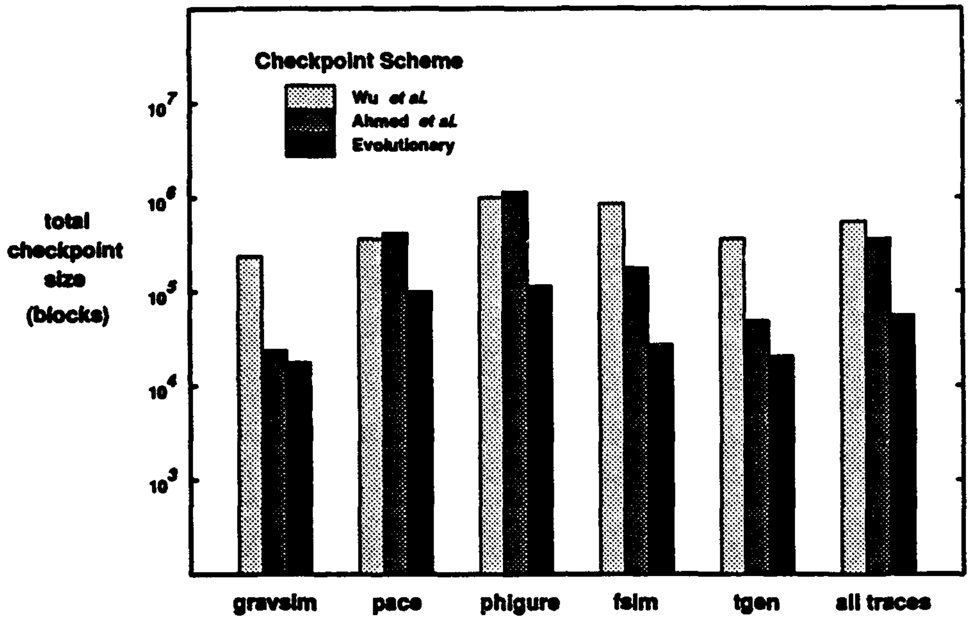

Figure 8. Scheme Comparison: Total Checkpoint Size.

B. Shared Virtual Memory System

A shared virtual memory system supports a shared-memory programming model in a distributed computer environment [22]. An interprocessor memory access may be implemented as an *RPC* (synchronized message) over a network. A communication synchronized checkpointing scheme similar to those for multiprocessor systems was proposed by Wu and Fuchs [28]. In such a system, the virtual memory is shared, cached in the main memory of individual processing nodes, and backed up on a stable storage. A checkpoint operation consists of flushing all dirty pages and saving processor registers to the stable storage. Whenever there is a remote access to a dirty page, the source processor takes a checkpoint. The checkpoint operation at the destination processor is eliminated since the source processor logs the requested page as a part of its checkpoint. If the destination processor rolls back, it can access the logged page from the checkpoint. Since the system is expected to be recoverable after node crashes, a shadow page system is used to accommodate the last committed virtual space (checkpoint) and the working space between checkpoints.

Similar to the multiprocessor case, our evolutionary scheme can be mapped to the distributed virtual memory case to provide controllable checkpointing. In this case, the message server is the pager process, and communication is a remote access to a dirty page. A checkpoint operation is performed at the source node during checkpoint sessions. A remote memory access is synchronized as the result of the *RPC* mechanism. Communication delay is likely to be bounded since page size is limited and the network is usually dedicated to the system. The timeout mechanism in *RPC* will further ensure the communication bound.

A global interrupt for the ckp_start and ckp_end broadcasting is not possible in a distributed-memory system, thus we need to use message broadcasting for ckp_start and ckp_end. To reduce the cost of flushing dirty pages, checkpoint operations may mark the local dirty pages unchangeable in memory. The marked pages are committed after the checkpoint session ends. A recovery simply restarts all processes from the recovery line. Unlike the multiprocessor case, the shadow pages needed for our evolutionary scheme are already used in the Wu and Fuchs scheme. Thus, our scheme incurs no additional memory overhead.

VI. SUMMARY

In this paper we have presented an evolutionary checkpointing strategy for concurrent processes. This checkpointing scheme starts from a potentially inconsistent recovery line by checkpointing individual processes independently. Local checkpoints are updated whenever there is communication during the checkpoint session. This local checkpoint updating makes the recovery line evolve into a globally consistent recovery line. We showed that the convergence time from an inconsistent recovery line to a consistent one is three times the maximal communication latency upper bound.

We verified the low overhead of our evolutionary scheme by measurements on different computer systems. Unlike globally synchronized checkpointing schemes, our evolutionary scheme requires no global synchronization protocols. The evolutionary approach provides controllable checkpointing in contrast to the communication synchronized schemes. The trace-based evaluation has shown that our scheme can achieve low-cost checkpointing at a controllable interval for error recovery in multiprocessors and distributed virtual memory systems. However, our scheme is limited by the requirement of synchronous communication with a bounded latency. In the systems with low overhead synchronization mechanisms, our scheme may not be necessary, since a global synchronization scheme may be simpler to implement than our approach.

REFERENCES

[1] B. Randell, "System structure for software fault tolerance," *IEEE Trans. Software Eng.*, Vol. 1, No. 2, pp. 220–232, June 1975.

[2] P. A. Lee and T. Anderson, *Fault Tolerance: Principles and Practice.* Springer-Verlag/Wien, 1990.

[3] K. Tsuruoka, A. Kaneko, and Y. Nishihara, "Dynamic recovery schemes for distributed processes," *Proc. IEEE 2nd Symp. on Reliability in Distributed Software and Database Syst.*, pp. 124–130, 1981.

[4] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *CACM*, Vol. 21, No. 7, pp. 558–566, July 1978.

[5] R. E. Ahmed, R. C. Frazier, and P. N. Marinos, "Cache-aided rollback error recovery (carer) algorithms for shared-memory multiprocessor systems," *Proc. 20th Int. Symp. Fault-Tolerant Comput.*, pp. 82–88, 1990.

[6] J. F. Bartlett, "A nonstop kernel," *Proc. ACM 8th Symp. Oper. Syst. Principles*, pp. 22–29, Dec. 1981.

[7] A. Borg, J. Baumbach, and S. Glazer, "A message system supporting fault tolerance," *Proc. ACM 9th Symp. Oper. Syst. Principles*, pp. 90–99, Oct. 1983.

[8] K.-L. Wu, W. K. Fuchs, and J. H. Patel, "Error recovery in shared memory multiprocessors using private caches," *IEEE Trans. Parallel and Distributed Syst.*, Vol. 1, No. 2, No. 2, pp. 231–240, 1990.

[9] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle, "Fault tolerance under unix," *ACM Trans. Comput. Syst.*, Vol. 3, No. 1, No. 1, pp. 63–75, Feb., 1985.

[10] M. L. Powell and D. L. Presotto, "Publishing: A reliable broadcast communication mechanism," *Proc. 9th Symp. Oper. Syst. Principles*, pp. 100–109, Oct., 1983.

[11] D. B. Johnson and W. Zwaenepoel, "Recovery in distributed systems using optimistic message logging and checkpointing.," *J. Algorithms*, Vol. 11, No. 3, pp. 462–491, Sept. 1990.

[12] T. T.-Y. Juang and S. Venkatesan, "Efficient algorithms for crash recovery in distributed systems," *Proc. 10th Conf. Foundations of Software Technology and Theoretical Comput. Sci.*, pp. 349–361, 1990.

[13] T. T.-Y. Juang and S. Venkatesan, "Crash recovery with little overhead," *Proc. 11th Int. Conf. Distributed Comput. Syst.*, pp. 454–461, May 1991.

[14] A. P. Sistla and J. L. Welch, "Efficient distributed recovery using message logging," *Proc. 8th Symp. Principles of Distributed Comput.*, Aug. 1989.

[15] R. E. Strom and S. A. Yemini, "Optimistic recovery in distributed systems," *ACM Trans. Comput. Syst.*, Vol. 3, No. 3, pp. 204–226, Aug. 1985.

[16] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM Trans. Comput. Syst.*, Vol. 3, No. 1, pp. 63–75, Feb. 1985.

[17] M. Spezialetti and P. Kearns, "Efficient distributed snapshots," *Proc. 6th Int'l. Conf. Distributed Comput. Syst.*, pp. 382–388, 1986.

[18] R. Koo and S. Toueg, "Checkpointing and rollback-recovery for distributed systems," *IEEE Trans. Software Eng.*, Vol. 13, No. 1, pp. 23–31, Jan. 1987.

[19] K. Li, J. F. Naughton, and J. S. Plank, "Checkpointing multicomputer applications," *Proc. 10th Symp. Reliable Distributed Syst.*, pp. 2–11, 1991.

[20] Z. Tong, R. Y. Kain, and W. T. Tsai, "Rollback recovery in distributed systems using loosely synchronized clocks," *IEEE Trans. Parallel and Distributed Syst.*, Vol. 3, No. 2, pp. 246–251, March 1992.

[21] F. Cristian, "A timestamp-based checkpoint protocol for long-lived distributed computations," *Proc. 10th Symp. Reliable Distributed Syst.*, pp. 12–20, 1991.

[22] K. Li, "IVY: A shared virtual memory systems for parallel computing," *Proc. Int. Conf. Parallel Processing*, pp. 94–101, 1988.

[23] H. Tokuda, C. W. Mercer, Y. Ishikawa, and T. E. Marchok, "Proiority inversions in real-time communication," *Proc. 10th IEEE Real-Time Syst. Symp.*, Dec. 1989.

[24] H. Tokuda and C. W. Mercer, "ARTS: Adistributed real-time kernel," *ACM Oper. Syst. Rev.*, Vol. 23, No. 3, July 1989.

[25] K. Li and P. Hudak, "Memory coherence in shared virtual memory systems," *Proc. 5th ACM Symp. Principles Distributed Comput.*, pp. 229–239, 1986.

[26] K. Li, J. F. Naughton, and J. S. Plank, "Real-time, concurrent checkpoint for parallel programs," *Proc. 2nd ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, pp. 79–88, March 1990.

[27] B. Janssens and W. K. Fuchs, "Experimental evaluation of multiprocessor cache-based error recovery," *Proc. Int. Conf. Parallel Processing*, Vol. I, pp. 505–508, Aug. 1991.

[28] K.-L. Wu and W. K. Fuchs, "Recoverable distributed shared virtual memory," *IEEE Trans. Comput.*, Vol. 39, No. 4, pp. 460–469, April 1990.

[29] J.-M. Hsu and P. Banerjee, "Hareware support for message routing in a distributed memory multicomputer," *Proc. Int. Conf. Parallel Processing*, pp. 508–515, Aug. 1990.

[30] C. C. Li and W. K. Fuchs, "CATCH: Compiler-assisted techniques for checkpointing," *Proc. 20th Int. Symp. Fault-Tolerant Comput.*, pp. 74–81, 1990.

[31] J. Long, W. K. Fuchs, and J. A. Abraham, "Compiler-assisted static checkpoint insertion," *Proc. 22th Int. Symp. Fault-Tolerant Comput.*, 1992.

[32] J. Long, W. K. Fuchs, and J. A. Abraham, "Implementing forward recovery using checkpointing in distributed systems," *Proc. 2nd IFIP Working Conf. Dependable Comput. for Critical Applications*, pp. 20–27, Feb. 1991.

[33] C. M. Krishna, K. G. Shin, and Y.-H. Lee, "Optimization criteria for checkpoint placement," *CACM*, Vol. 27, No. 6, No. 6, pp. 1008–1012, Oct. 1984.

[34] A. Duda, "The effects of checkpointing on program execution time," *Information Processing Letters*, Vol. 16, pp. 221–229, 1983.

[35] E. Gelenbe and D. Derochette, "Performance of rollback recovery systems under intermittent failures," *CACM*, Vol. 21, No. 6, No. 6, pp. 493–499, 1978.

[36] J. W. Young, "A first order approximation to the optimal checkpoint interval," *CACM*, Vol. 17, No. 9, pp. 530–531, Sept. 1974.

[37] P. A. Bernstein, "Sequoia: a fault-tolerant tightly coupled multiprocessor for transaction processing," *IEEE Comput.*, Vol. 21, pp. 37–45, Feb. 1988.

[38] N. S. Bowen and D. K. Pradhan, "Vitual checkpoints: Architecture and Performance," *IEEE Trans. Comput.*, Vol. 41, No. 5, May 1992.

[39] T. P. Ng, "Checkpointing in a virtual shared memory system," Tech. Rep. UIUCDCS-R-91-1700, Department of Computer Science, University of Illinois, Dec. 1991.

[40] D. B. Hunt and P. N. Marinos, "A general purpose cache-aided rollback error recovery (CARER) technique," *Proc. 17th Symp. Fault-Tolerant Comput.*, pp. 170–175, 1987.

[41] Encore Computer Corporation, *Multimax Technical Summary*. Encore Computer Corporation, Jan. 1989.

List of Tables

List of Figures