Scheduling Message Processing for Reducing Rollback Propagation

Yi-Min Wang and W. Kent Fuchs

Coordinated Science Laboratory
University of Illinois at Urbana-Champaign

Abstract

Traditional checkpointing and rollback recovery techniques for parallel systems have typically assumed the communication pattern is specified by program behavior. In this paper we exploit the property that the communication pattern can often be changed at run-time without affecting program correctness. A scheduling algorithm for message processing and its implementation for reducing rollback propagation are described. The algorithm incorporates a user-transparent prioritized scheme based upon the run-time communication and checkpointing history. Communication trace-driven simulation for several parallel programs written in the Chare Kernel language demonstrates that the probability of rollback propagation can be reduced at the cost of slight additional performance degradation.

1 Introduction

Numerous checkpointing and rollback recovery techniques have been proposed in the literature for parallel systems. *Rollback propagation* and the associated *domino effect* [1] have been the primary issue of concern in many of these techniques. Checkpointing for parallel and distributed systems can be classified into three primary categories. *Coordinated checkpointing* schemes synchronize computation with checkpointing by coordinating processors during a checkpointing session in order to maintain a consistent set of checkpoints [2-4]. Rollback propagation is avoided at the cost of potentially significant performance degradation during normal execution. *Loosely-synchronized checkpointing* schemes [5,6] reduce the overhead for coordination by taking advantage of the loosely-synchronized checkpointing clocks and by bounding the message transmission delay. *Independent checkpointing* schemes replace the checkpoint synchronization by dependency tracking and possibly message logging [7-11] in order to preserve process autonomy. Rollback propagation in case of a fault is managed by searching for a consistent system state based on the dependency information. Lower run-time overhead during normal execution is achieved by allowing slower recovery and maintaining multiple checkpoints. Our paper considers independent checkpointing schemes for possibly nondeterministic execution.

Research on rollback recovery in multiple-processor systems has typically assumed that the communication pattern is determined by program behavior and is not otherwise controllable. Our approach is based on the observation that the communication pattern (message sending and processing) can often be determined by the run-time support system in a user-transparent way as well as by program behavior. This observation has been used by others to reduce cache thrashing by means of array subscript analysis in nested parallel loop constructs for dynamic thread scheduling [12]. In a message-passing system, since the order in which the messages arrive at a processor can not be assumed, changing the order of message processing will typically not affect program correctness. We will show that the probability of rollback propagation in a message-passing system can often be greatly reduced by reordering the processing of messages.

Another contribution of this paper is the measurement of actual rollback propagation for several parallel programs. Analysis of checkpointing and rollback recovery protocols has typically been performed by means of theoretical models. We examine the degree of rollback propagation in parallel programs with independent checkpointing and three message scheduling algorithms.

[1]This research was 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 Grant NASA NAG 1-613, in cooperation with the Illinois Computer Laboratory for Aerospace Systems and Software (ICLASS).

The outline of the paper is as follows. Section 2 describes the system model; the message scheduling algorithm is presented in Section 3; and Section 4 gives our evaluation, the measurement of rollback propagation and comparisons. Section 5 discusses the limitations of our approach.

2 System Model and Checkpoint Consistency

The system model considered in this paper is a *message-driven system* consisting of a number of concurrent processes for which all process communication between processors is through message passing. Processes on the same processor share a single message queue and belong to the same fail-stop recovery unit [8, 13]. Processes can be dynamically generated and the request for the creation of a new process is sent out as a *job message* to some processor according to the load balancing strategy. When a processor is ready to process a new message, it can pick any one in the queue, depending on the scheduling algorithm, and then invoke or create the appropriate process as requested by the message. Although this model is usually applied to distributed-memory multicomputers, recent work on parallel environments has shown that the message processing model can also be efficiently used on shared-memory multiprocessors [14].

During normal execution, the state of each processor is occasionally saved as a *checkpoint* on stable storage. Let CP_{ik} denote the kth checkpoint of processor p_i with $k \geq 0$ and $0 \leq i \leq N-1$, where N is the number of processors. A *checkpoint interval* is defined to be the time between two consecutive checkpoints on the same processor and the interval between CP_{ik} and $CP_{i(k+1)}$ is called the kth checkpoint interval. Each message is tagged with the current checkpoint interval number and the processor number of the sender. Each processor takes its checkpoint independently and updates the *communication information table*, or *input table* [9], as follows: if at least one message from the mth checkpoint interval of processor p_j has been processed during the previous checkpoint interval, the pair (j, m) is added to the table. A *checkpoint space reclamation algorithm* [15] can be periodically invoked by any processor to reduce the space overhead.

When processor p_i detects an error, it starts a two-phase centralized recovery procedure [9]. First, a *rollback-initiating* message is sent to every other processor to request the up-to-date communication information. Each surviving processor takes a *virtual checkpoint* upon receiving the *rollback-initiating* message so that the communication information during the most recent checkpoint interval is also collected. After receiving the responses, p_i constructs the *extended checkpoint graph* [7] and executes the *rollback propagation algorithm* [15] to determine the *local recovery line*. A *rollback-request* message is sent to each processor which then rolls back and restarts according to the local recovery line.

There are two important situations concerning the consistency between two checkpoints. In Fig. 1(a), if processors p_i and p_j restart from CP_{ik} and CP_{jm} respectively, message m is recorded as "received but not yet sent". In a general model without the assumption of deterministic execution [16], message m becomes an *orphan message* [6] and results in inconsistency between CP_{ik} and CP_{jm}.

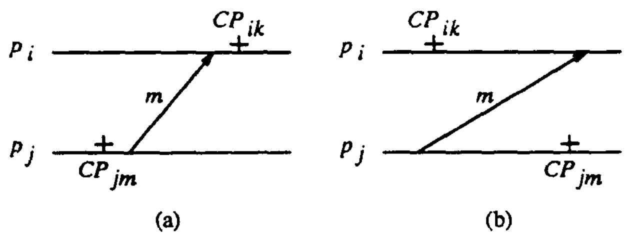

Figure 1: Checkpoint consistency (a) message received but not yet sent; (b) message sent but not yet received.

Fig. 1(b) illustrates the second situation. The message m becomes a *lost message* [6] according to the system state containing CP_{ik} and CP_{jm}. By defining the *state of the channels* to be the set of messages sent but not yet received, it has been proved [2, 5] that checkpoints like CP_{ik} and CP_{jm} can be considered consistent if the corresponding state of the channels is also recorded. Koo and Toueg [3] assumed such a state is recorded at the sender side by some end-to-end transmission protocol. Another way of recording the channel state is through message logging. Pessimistic logging protocol [17, 18] can ensure such a state is properly recorded at the receiving end[2]. As a result, we consider the situation in Fig. 1(b) as consistent.

[2]The recovery protocol described above can also be modified and applied to systems with optimistic logging [11].

Statement A per telecom
Dr. Clifford Lau
ONR/Code 1114
Arlington, VA 22217-5000 NWW 6/23/92

3 Scheduling Message Processing

3.1 Problem description

In communication-induced checkpointing [19-22] a checkpoint is taken on the sender side whenever communication between two processors occurs. Because the rollback of a processor does not affect any other processor, rollback propagation is avoided. Our message scheduling algorithm is motivated by the above schemes. The difference is, instead of inserting the checkpoints based on a given communication pattern, we control the communication pattern whenever possible according to the fixed checkpoint pattern. The receiver of each message tries to delay the processing of the message until the sender passes its next checkpoint based on the predetermined checkpoint interval.

For example in Fig. 2(a), message m_0 enters the queue of processor p_1 at point A earlier than the message m_2 at point B. By using a simple first-in-first-out (fifo) scheduling algorithm, p_1 will process m_0 first when it is ready to process a new message at point C. However, since the order of processing these two messages does not affect program correctness in our model, m_2 is a better candidate at point C because its sender p_2 has passed its next checkpoint. If indeed m_2 is first processed and is finished at point D (Fig. 2(b)), the sender of message m_0 will also have passed its next checkpoint. When all the messages can be processed in this way, there will be no rollback propagation.

The following situations need to be considered for modifying the simple scheme described above.

1. If the processing of m_2 completes at point D' instead of D, the sender of m_0 has not passed the checkpoint yet. If p_1 has to process m_0 in order to maintain performance, the communication pattern will no longer be free of rollback propagation. However, the delayed processing of message m_0 does contribute to reducing the rollback propagation probability because it reduces the chance that an error in p_0 requires the rollback of p_1 because of m_0.

2. Consider the case shown in Fig. 3. Suppose p_1 is forced to process m_0 at point A. It is clear that any rollback initiated by p_0 between A and CP_{01} will rollback m_0 and therefore m'_0. It then becomes irrelevant whether the message m'_0 is processed before CP_{01} or after.

3. Fig. 4 illustrates the situation where the rollback propagation involves more than two processors.

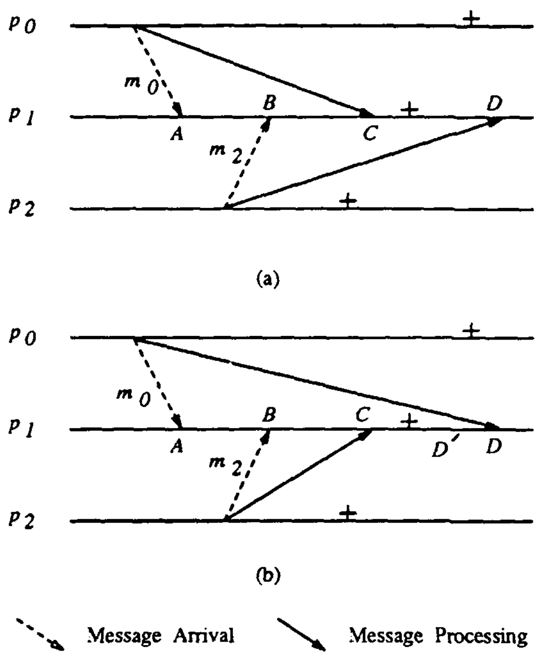

Figure 2: Checkpointing and message processing (a) normal fifo processing; (b) reordered processing for reducing rollback propagation.

Although m_1 is processed after p_1 has passed CP_{11}, a rollback of p_0 initiated between A and CP_{01} can still propagate the rollbacks to p_1 and p_2 through m_0 and m_1. Such a situation is similar to the *indirect potential recaller (IPR)* relationship described by Kim, et. al [23,24].

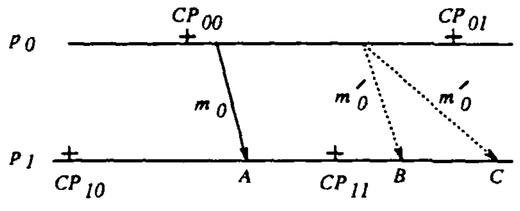

Figure 3: Processing irrelevant messages in terms of rollback propagation.

3.2 The scheduling algorithm

Based on the above observations, we give the following definition:

Definition: A message m in the queue of a processor p is *safe* if the immediate processing of m by p does not increase the probability of rollback propagation.

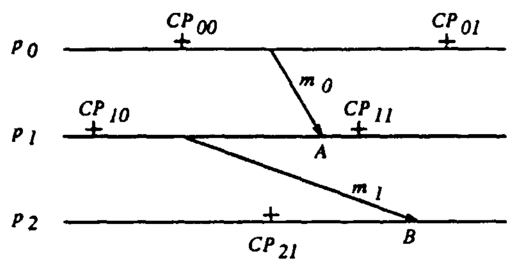

Figure 4: Rollback propagation involving more than two processors.

There are two types of safe messages:

Type 1: when at least one message from the ith checkpoint interval of p_s is processed by the receiver p_r, all the messages from the jth checkpoint interval, $j \leq i$, of p_s are safe with respect to p_r (Fig. 3).

Type 2: messages sent to another process on the same processor are always safe because the recovery unit is the processor and so the rollback of the sender of such messages automatically rolls back the receiver.

Definition: The *hazard index* associated with each message m with respect to the receiver p is a measure of the increased probability of rollback propagation resulting from the immediate processing of m by p.

By definition, the hazard indices of all safe messages are zero. For each *unsafe message*, the exact hazard index might depend on the communication between other processors, for example the message m_1 in Fig. 4, and is in general impossible or difficult to calculate. Since the purpose of our work is to "reduce" rollback propagation instead of "eliminating" it, we are interested in finding a feasible, low-overhead approximation to the hazard index. In this paper, we approximate the hazard index of an unsafe message by "the time to the next checkpoint of its sender" In particular, when the sender of a message m takes its next checkpoint after m was sent, the hazard index of m reduces to zero and m is treated as a safe message. The actual implementation will be described later. Experimental results in Section 4 support the use of such an approximation.

Our message scheduling algorithm is a prioritized scheme based upon the hazard index which encapsulates the run-time communication and checkpointing information. The higher the hazard index for a message, the lower priority it is assigned. Therefore, when a processor is ready to choose a new message for processing, it will first choose one of the safe messages if there is one; otherwise, the unsafe message with the lowest hazard index will be chosen. With this scheduling algorithm, the probability of rollback propagation is not increased until it is necessary to keep the processors from idling.

3.3 Implementation

In order for the receiver to maintain the hazard indices for the received messages, an additional piece of information needs to be piggybacked on each message: the time to the next checkpoint of the sender when the message is sent. The receiver can then properly manage its message queue based on this information.

Instead of keeping messages from different processors in the same queue, each processor maintains an array of sub-queues, one for each processor. Each message enters its corresponding sub-queue if its hazard index is not zero and is added to the highest-priority *safe queue* if it is a safe message. Three additional data structures are needed for proper queue management:

1. *Last_Update_Time* records the time at which the most recent update of the time-to-next-checkpoint information was completed. It is needed for the aging operation described later.

2. *Last_Known_CP_Num*$[N]$ is an array, with one entry for each processor, recording the most recent checkpoint interval number of every processor that is known to the local processor based on the communication history.

3. *Last_Processed_CP_Num*$[N]$ is an array recording the highest checkpoint interval number of the processed messages from each processor. It is used for identifying Type-1 safe messages.

The updates of hazard indices and priorities take place only when a new message arrives (***enqueueing***) or when the processor is about to process the next message (***dequeueing***). The operations performed on the message queue for enqueueing and dequeueing are outlined in Fig. 5 and Fig. 6, respectively. The *aging* operation updates the time-to-next-checkpoint information of the last message in each non-empty unsafe sub-queue by the amount of the difference between current time and *Last_Update_Time*. If the time to the next checkpoint of a message becomes negative, all the messages in the same sub-queue are moved to the safe queue.

```
/* message $m$ from the $i$th checkpoint interval of $p_s$ arrives at the message queue $Q$
   on $p_r$ */
perform *aging* operation on $Q$;
if ($i < Last\_Known\_CP\_Num[p_s]$)
    add $m$ to safe queue;
else {
    if ($i > Last\_Known\_CP\_Num[p_s]$)
        $Last\_Known\_CP\_Num[p_s] = i$;
    if ($i \leq Last\_Processed\_CP\_Num[p_s]$)
        add $m$ to safe queue;
    else
        add $m$ to unsafe sub-queue[$p_s$];
}
```

Figure 5: Operations for enqueueing.

```
/* $p_r$ is about to choose a message from
   queue $Q$ */
perform *aging* operation on $Q$;
if (safe queue is non-empty)
    choose a message from safe queue;
else {
    choose the unsafe message $m$ with the
    smallest hazard index;
    move the remaining messages in the same
    sub-queue to the safe queue;
    /* if $m$ is from the $i$th checkpoint interval of
       $p_s$ */
    $Last\_Processed\_CP\_Num[p_s] = i$.
}
```

Figure 6: Operations for dequeueing.

4 Experimental Evaluation

Our message scheduling algorithm is implemented in the *Chare Kernel* which has been developed as a medium-grain, machine-independent parallel language [14]. A program written in the Chare Kernel language can run on both shared-memory and distributed-memory machines such as Encore Multimax, Sequent Symmetry, and the Intel iPSC/2 hypercube. Our experiments are on an eight-processor Multimax 510.

A Chare Kernel program is structurally similar to a C program. It contains a superset of the C language without global or static variables. Two Chare Kernel calls resulting in communication are:

1. *CreateChare*() sends a message to request the creation of a small process, called *chare*, and the processing of the enclosed data by one of the subroutines, called *entry codes*, inside the created chare.

2. *SendMsg*() sends a message to an existing chare and requests the enclosed data to be processed by one of the entry codes.

There is no *receive_message* statement in a Chare Kernel program. All messages are sent to the copy of the Kernel running on the destination processor and managed by the Kernel according to the scheduling algorithm selected by the user. Messages sent by *CreateChare*() calls are kept in *job queues* and those sent by *SendMsg*() calls enter the *message queues*. Each message queue has a higher priority than the corresponding job queue. We apply our scheduling algorithm, referred to as *PRIoritized Message Process Scheduling* (*PRIMPS*), to both types of queues and force the Kernel to dequeue a message from the job queue when there is no safe message left on the message queue. In this section, we will compare our PRIMPS algorithm with two alternatives supplied by the Kernel: first-in-first-out (fifo) scheduling and last-in-first-out (lifo) scheduling.

The four programs used in this study are Matrix multiplication, Circuit extraction, Knight tour and N queen. The execution and checkpoint parameters for each program are listed in Table 1. Each checkpoint interval is arbitrarily chosen to be approximately 10 percent of the total execution time. Offsets between corresponding checkpoints on different processors are introduced to study the rollback propagation resulting from asynchronous checkpointing. The ith checkpoint of p_j is taken at time $i * T - j * \Delta$ where T is the checkpoint interval and Δ, the offset per processor, is arbitrarily set to $T/10$. Our implementation of periodic checkpointing utilizes the interrupt service routine for UNIX *alarm(T)* system call as the checkpointing routine. Each checkpointing action is simulated by inserting a constant delay (2 seconds). We assume a technique for detecting the messages which do not require logging [11] is employed so that the overhead for message logging is negligible.

We define several rollback statistics associated with each communication pattern which encapsulate various costs for rollback recovery. They allow quantitative comparison between different scheduling algorithms.

Table 1: Execution and checkpoint parameters of the Chare Kernel programs.

Benchmark programs	Matrix multiplication	Circuit extraction	Knight tour	N Queen
Number of processors	4	4	6	6
Execution time (sec)	290.07	252.51	280.15	1507.78
Checkpoint interval (sec)	30	30	30	150
Offset per processor (Δ)	3	3	3	15

Table 2: Rollback statistics.

Benchmark programs		Matrix multiplication	Circuit extraction	Knight tour	N Queen
rbcp	lifo	5.42	1.25	30.50	30.61
	fifo	2.54	1.15	31.72	32.19
	PRIMPS	1.17	1.13	1.62	1.28
rbpe	lifo	2.83	1.19	5.73	5.43
	fifo	2.09	1.12	5.78	5.40
	PRIMPS	1.13	1.10	1.53	1.23
ckp1	lifo	0.537	0.045	0.906	0.841
	fifo	0.224	0.019	0.906	0.833
	PRIMPS	0.039	0.016	0.044	0.023
Execution time (sec)	lifo	290.07	252.51	280.15	1507.78
	fifo	299.50	256.88	282.50	1635.18
	PRIMPS	304.56	254.34	278.50	1526.92
Performance	degradation	4.99%	0.72%	0.00%	1.27%

1. The average of *the total number of rolled-back checkpoints (rbcp)* due to the rollback initiation of a processor.

2. The average of *the total number of rolled-back processors (rbpe)* due to the rollback initiation of a processor.

3. *The probability of rolling back at least one processor to some checkpoint before the most recent checkpoint (ckp1).*

Communication traces were collected by intercepting the *CreateChare*() and *SendMsg*() calls, and recording the time each message was dequeued. Communication trace-driven simulation was then performed on the traces to obtain the rollback statistics shown in Table 2 and Fig. 7. All the reported numbers were computed by averaging over five runs.

The results show that the degree of rollback propagation varies widely between different programs and different offsets. In all cases, our PRIMPS algorithm reduces the cost of rollback recovery and the probability of rollback propagation at the expense of less than 5 percent performance degradation. Note that the performance degradation reported here is measured against the execution with lifo scheduling and with simulated checkpointing overhead. Therefore, it represents the overhead required for performing the prioritized scheduling above the normal overhead for supporting checkpointing.

One potential disadvantage of independent checkpointing schemes is that slower recovery due to possible rollback propagation may make it unsuitable for real-time applications. Table 2 shows that the rollback cost (*rbcp*) and restarting cost (*rbpe*) for the PRIMPS algorithm are not only greatly reduced but also reduced to numbers less than two. Since the minimum value for both *rbcp* and *rbpe* is one (the rollback of the faulty processor), the statistics in Table 2 imply

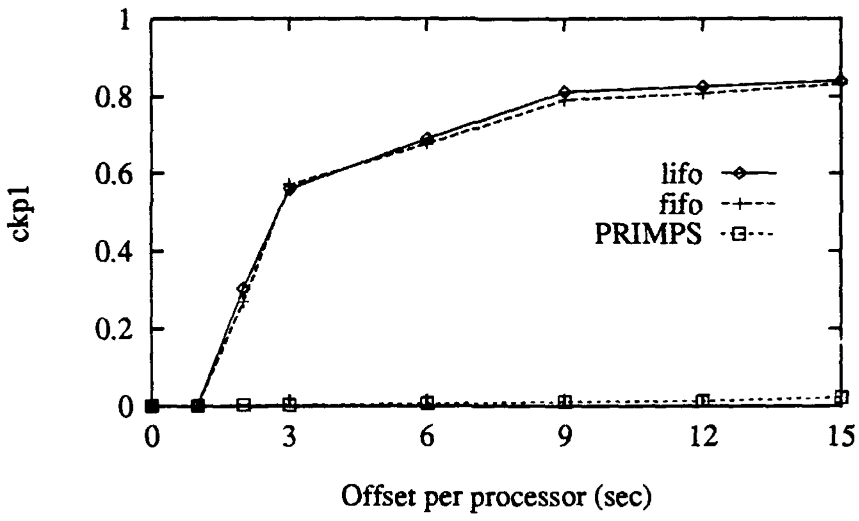

Figure 7: Comparison of rollback statistics as a function of the offset per processor Δ for the N Queen program.

that rollback recovery for independent checkpointing can be made faster by using the PRIMPS algorithm.

The small percentage (less than 5%) of potential rollbacks contributing to the non-zero *ckpl*'s for the PRIMPS algorithm exist mostly at the starting phase or the ending phase of the program execution where the number of messages is small. Although the checkpoints are not explicitly synchronized, this result shows the set of most recent checkpoints still forms a consistent recovery line with high probability (higher than 0.95) for the PRIMPS algorithm.

Fig. 7 shows that the degree of rollback propagation becomes worse as the offset Δ increases for all three algorithms. However, it is less sensitive to the offset for the PRIMPS algorithm. Therefore, our approach is particularly attractive for applications where such offsets may exist and the synchronization cost for always maintaining a consistent set of checkpoints is prohibitively high.

5 Limitations

Although the actual implementation of logging and checkpointing will have little impact on our comparison of rollback statistics, it does affect the performance degradation. For some applications, the PRIMPS algorithm might tend to start more jobs than is necessary to keep the system busy in order to delay the processing of unsafe messages. Therefore, the memory usage and the overhead for checkpointing and message logging may be higher.

Sometimes priorities are attached to the messages in order to ensure correctness or achieve better load balancing. Since our message scheduling algorithm can not schedule messages across different priority groups, there are fewer messages available for scheduling in each group and the algorithm will be less effective.

6 Concluding Remarks

In this paper, we have exploited the property that reordering the message processing can change the communication pattern to reduce the occurrence of rollback propagation without affecting program correctness. The concepts of safe messages and hazard indices are introduced as the basis for our run-time prioritized scheduling algorithm. Experimental results based on the communication trace-driven simulation for several parallel programs show that the probability of rollback propagation can be greatly reduced at the cost of reasonable performance degradation.

Acknowledgement

The authors wish to express their sincere thanks to Balkrishna Ramkumar for his insight and help with the Chare Kernel, to L. V. Kalé for access to the Chare

Kernel, to Prith Banerjee for his parallel programs, and to the referees for their valuable comments.

References

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

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

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

[4] K. Li, J. F. Naughton, and J. S. Plank, "Checkpointing multicomputer applications," in *Proc. IEEE Symp. on Reliable Distributed Systems*, pp. 2–11, 1991.

[5] F. Cristian and F. Jahanian, "A timestamp-based checkpointing protocol for long-lived distributed computations," in *Proc. IEEE Symp. on Reliable Distributed Systems*, pp. 12–20, 1991.

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

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

[8] R. E. Strom and S. Yemini, "Optimistic recovery in distributed systems," *ACM Trans. on Computer Systems*, Vol. 3, No. 3, pp. 204–226, Aug. 1985.

[9] B. Bhargava and S. R. Lian, "Independent checkpointing and concurrent rollback for recovery - An optimistic approach," in *Proc. IEEE Symp. on Reliable Distributed Systems*, pp. 3-12, 1988.

[10] D. B. Johnson and W. Zwaenepoel, "Recovery in distributed systems using optimistic message logging and checkpointing," *J. of Algorithms*, Vol. 11, pp. 462–491, 1990.

[11] Y. M. Wang and W. K. Fuchs, "Optimistic message logging for independent checkpointing in message-passing systems." Technical Report, Center for Reliable and High-Performance Computing, Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, 1992.

[12] J. Fang and M. Lu, "A solution of cache ping-pong problem in RISC based parallel processing," in *Proc. Int. Conf. on Parallel Processing*, pp. I–238–I–245, 1991.

[13] R. D. Schlichting and F. B. Schneider, "Fail-stop processors: An approach to designing fault-tolerant computing systems," *ACM Trans. on Computer Systems*, Vol. 1, No. 3, pp. 222–238, Aug. 1983.

[14] W. Shu and L. V. Kalé, "Chare kernel - A runtime support system for parallel computations," *J. Parallel and Distributed Computing*, Vol. 11, pp. 198–211, 1991.

[15] Y. M. Wang, P. Y. Chung, I. J. Lin, and W. K. Fuchs, "Reducing space overhead for independent checkpointing," Tech. Rep. CRHC-92-06, Center for Reliable and High-Performance Computing, Coordinated Science Laboratory, University of Illinois at Urbana–Champaign, 1992.

[16] D. B. Johnson and W. Zwaenepoel, "Transparent optimistic rollback recovery," *Operating Systems Review*, pp. 99–102, Apr. 1991.

[17] M. L. Powell and D. L. Presotto, "Publishing: A reliable broadcast communication mechanism," in *Proc. 9th ACM Symp. on Operating Systems Principles*, pp. 100–109, 1983.

[18] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle, "Fault tolerance under UNIX," *ACM Trans. on Computer Systems*, Vol. 7, No. 1, pp. 1–24, Feb. 1989.

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

[20] K. L. Wu and W. K. Fuchs, "Recoverable distributed shared virtual memory," *IEEE Trans. on Computers*, Vol. 39, No. 4, pp. 460–469, Apr. 1990.

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

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

[23] K. H. Kim, J. H. You, and A. Abouelnaga, "A scheme for coordinated execution of independently designed recoverable distributed processes," in *Proc. Symp. on Fault-Tolerant Computing*, pp. 130–135, 1986.

[24] K. H. Kim and J. H. You, "A highly decentralized implementation model for the Programmer-Transparent Coordination (PTC) scheme for cooperative recovery," in *Proc. Symp. on Fault-Tolerant Computing*, pp. 282–289, 1990.