ROLLBACK PROPAGATION DETECTION AND PERFORMANCE EVALUATION OF FTMR²M -- A FAULT-TOLERANT MULTIPROCESSOR

Yann-Hang Lee and Kang G. Shin

Electrical, Computer, & System Engineering Dept.
Rensselaer Polytechnic Institute, Troy, NY 12181

ABSTRACT

In this paper we consider the rollback propagation and the performance of a fault-tolerant multiprocessor with a rollback recovery mechanism (FTMR²M)[1], which was designed to be tolerant of hardware failure with minimum time overhead. Rollback propagation between cooperating processes is usually required to ensure correct recovery from failure. To minimize the waste of processor time and storage overhead required for handling sophisticated rollback propagations, the FTMR²M always keeps one recoverable state. Approaches for evaluating the recovery overhead and analyzing the performance of FTMR²M are presented. Two methods for detecting rollback propagations and multi-step rollbacks between cooperating processes are also proposed.

1. INTRODUCTION

Due to the increasing demand for reliability and survivability in the modern computing arena, it is highly desirable to have a recovery scheme which enables the computing system to automatically recover from various faults. The requirements of fault recovery speed as well as storage size are known to be the most difficult problems faced by the system designer, especially in real-time applications. To meet these requirements, an approach based on a multi-microprocessor system, which has inherently reliable LSI components and system reconfigurability, has been considered as a solution. The Fault-Tolerant Multiprocessor with a Rollback Recovery Mechanism (FTMR²M) [1] was designed to be tolerant of hardware faults with minimum time overhead. In this paper we investigate the performance of FTMR²M and the treatment of multi-step rollbacks.

All correspondence should be directed to Prof. Kang G. Shin, ECSE Department, Rensselaer Polytechnical Institute, Troy, NY 12181.

The organization of FTMR²M is based on the backward recovery method and the recovery block. The recovery block, proposed by Randell [2], is a program structure that is composed of recovery points, acceptance tests, and alternative processes for a given task. The alternative processes may be different algorithms started from the same state. If a process fails the acceptance test or if an error is detected during execution, the system will roll back to the previously recorded state and try one of the other alternatives. Several aspects in this area have been studied; for example, the technique for evaluating and minimizing overhead developed by Chandy and Ramamoorthy [3], the strategies for inserting recovery points by O'Brien [4], the conditions for avoiding the domino effect and purging the old recovery block by Kant and Silberschatz in [5], and other designs, with emphasis on programmer transparency, have been proposed by Meraud and Browaeys [6], and Kim [7, 8].

The concept of the recovery block is also useful for tolerating hardware faults in a reconfigurable system. In general, process states are defined by the internal registers and the variables in memory at the end of each instruction execution. To generate the recovery block, we attempt to record these states consecutively and concurrently during the execution of the process with a special state-save mechanism. After a fault is detected and isolated, the system will be reconfigured to replace the failed processor module (PM). By loading the program code and migrating the recorded states into the replacement PM, the original process can be resumed. Thus every interval between two consecutive state-savings can be regarded as a recovery block.

To resume the execution of a task comprised of cooperating processes from a fault-free and consistent state, the rollback of the failed processor may have to propagate to other processors or to a further recorded state. The worst case is when an avalanche of rollback propagations, namely the domino effect, occurs. We may be able to eliminate the domino effect by restricting interactions and by making an a

0149-7111/82/0000/0171$00.75 © 1982 IEEE

priori analysis of interactions, which bring about higher overhead in storage and execution time. If the probability of the domino effect is small, it seems better to restart the whole task when the domino effect occurs than to prevent the domino effect. This seems true even for the case when rollback propagates more than one step (note that the domino effect is a special case of this). Consequently we adopted a method called automatic rollback recovery which constructs the recovery blocks automatically and allows the task to roll back only a single step; if a multiple-step rollback is required, then the task has to be restarted. In this method an additional measure to detect the domino effect or multiple-step rollback must be taken prior to recovery.

This paper is organized as follows. Section 2 briefly reviews the architecture of FTMR²M, and Section 3 discusses the detection of rollback propagation and domino effect. Section 4 deals with the estimation and analysis of performance in terms of the expected execution time. A conclusion is presented in Section 5.

2. REVIEW OF FTMR²M ARCHITECTURE

The FTMR²M architecture has been introduced in our earlier work [1], but for convenience the basic organization of this multiprocessor is presented in Fig. 1. It is similar to the Cm* system [9, 11]. The major aspects are briefly described below.

2.1 Processor Module and State Saving

A basic processor module in the multiprocessor system under consideration consists of a processor, a local memory, a local switch, state-save memory units and a switch monitor. Each PM can execute a process of the given task and can communicate with other processes allocated in other PMs. The processor module saves its states (i.e. variables and status) at various stages of execution, called a state-save. At regular intervals of duration, T_{ss}, an external clock stimulates every processor module to save its states as soon as it completes the current instruction. Then the processor executes a validation test. If the processor survives the test, the saved state would be regarded as the recovery point for the next interval. If the processor fails the validation test, it will roll back to the previously saved state. The detailed operations in the rollback recovery are shown in Fig. 2.

During a state-save interval, besides the normal execution of instructions, certain operations are automatically executed; for example, a parity check is done whenever the buses are used. Some redundant error detection units are accompanied with the processor module [10], e.g. dual-redundancy comparison, address-in-bound check, etc. These units are expected to detect a malfunction as soon as the corresponding function unit is used. An additional validation process refreshes the shelters to guarantee the saving of a fault-free state and thus ensures safeguarding against fault propagation.

To minimize the time overhead required for state-save, the saving is done concurrently with process execution. Every update of variables in the local memory is recorded in the state-save memory unit simultaneously. Two such state-save memory units, called SSU_1 and SSU_2, are used for saving states at two consecutive intervals. Thus each PM always keeps one valid state saved in one unit and stores the currently changing state in the other. The two SSU's that save the old and current states respectively will be switched following the completion of every state-save. The monitor switch is used to route the current state to one SSU and to manage the state-switches.

Since the update of dynamic elements is copied in only one SSU, the other SSU is ignorant of it. This fact may bring about a serious problem: the newly updated variable may be lost. In order to avoid this, it is necessary to make the contents of two SSU's identical at each state-switching instant or to copy the variables that have been changed in the previous interval into the current state-save unit. A solution to this problem has been proposed in the previous paper [1]. At each state-switch instant, the current SSU contains not only the currently updated variables but also the previously updated variables. So, the contents of the current SSU always represent the newest state of the PM.

2.2 System Organization

The state-save of a task that is distributed over processor modules PM_i ($i=1,\ldots,N$) implies the state-save of each PM. The resumption of a failed process may need cooperation from other processes due to the concurrent execution of processes and interprocessor communication. Fault recovery in the task level could involve the rollback of the failed process and other related processes. Since (1) cooperating PMs can have arbitrary interactions and (2) each PM saves its states asynchronously with others, it is possible to require multi-step rollback. If the probability of a multi-step rollback is small, it may be pratical to simply restart the whole task when a multi-step rollback occurs, instead of constructing more complex mechanism to allow multi-step rollbacks for questionable return. To make the fault recovery successful in this case, the system should check the rollback propagation and multi-step rollback, as well as the migration of failed process to the replacement PM.

The system contains three data paths that connect PMs at two-level hierarchy. The intra-cluster bus, a time-shared parallel bus, groups PMs into a cluster. The inter-cluster link forms a modular computer network which permits expansion. Another data path, formed by the DMA controller and DMA channel, is used to transfer the program code and process states such that the loading and migration of a process can be handled without interfering with the intra-cluster bus.

The cluster node is composed of a switch and a monitor. This switch element, called the cluster switch, handles the external references and records these references to analyze rollback propagations. The cluster monitor manages the PM pool in the cluster. It receives the request from the host and loads a process via the DMA data path to a PM. It also receives reports from PMs in the cluster as to the state-save operations and PMs' conditions. Once a failure is detected, the cluster monitor will signal "retry" to the PM in question. If the same failure is detected again, a permanant fault is declared and the following steps are taken by the cluster monitor and cluster switch:

1. Stop all other PMs that are executing processes of the same task.
2. Make a decision on the rollback propagation.
3. Resume the execution of processes that are not affected by rollback propagation.
4. Find a free PM to replace the failed PM.
5. Migrate the process in the failed PM to the replacement PM.
6. Roll back the processes affected by the rollback of the failed PM.
7. Any interaction directed to the failed PM must wait for the resumption of the process in the new PM. Old and unserviced interactions issued by these rolled-back PMs, which are still queued in the cluster node, are cancelled.

There are several tables that will be involved in the rollback recovery. First, the processor-task table (PTT) is used for associating the processes being executed in the PMs with a task and identifying the PMs' conditions. Second, the task-processor table (TPT) groups all PMs executing processes of a task. Both tables can provide the information of the relations between physical PMs and a logical task. The third table is the address mapping table for external references. To avoid updating the whole mapping table after the reconfiguration, an associated rerouting table (RT) is used to map addresses directed to faulty PMs into their replacements.

3. PROPAGATION AND DETECTION OF ROLLBACK

In FTMR²M, each interaction between cooperating processes is regarded as a reference to shared variables. These shared variables, a portion of dynamic data, can be located in an arbitrary PM. The state of process P_i should include the shared variables that are accessed by process P_i. To roll back a failed process, it is necessary to provide the consistent contents of the shared variables as well as those of the process's local variables and internal states. Since the shared variables may reside outside the associated PM, two related aspects, namely the rollback propagation and the multi-step rollback, must be considered.

3.1 Rollback Propagation and Multi-step Rollback

Suppose a shared variable X resides in PM_j which executes process P_j, and process P_i has accessed X in its current state-save interval. One can consider the following three ways in which the rollback may affect other processes.

1. If P_j rolls back and if P_i has already changed X, the reexecution of P_j requires X to contain the old value prior to the change by P_i. Hence P_i should also roll back.
2. If process P_i rolls back after it has written to variable X, this new value of X should be discarded since the write operation may be faulty and is cancelled by the rollback of P_i. The process P_j has to roll back so as to recover the previous value of X.
3. If P_i reads X during a state-save interval when P_j fails, we can not ensure the correctness of the value read by process P_i. To avoid a possible fault propagation, process P_i has to roll back.

Even if the rollback does not propagate in the case that process P_i has read variable X and then fails, this case is treated the same as above to simplify the detection of rollback propagation. We assume that the cooperating processes must roll back if there exists an interaction during the current state-save interval of one of the involved processes. This propagation is denoted as $P_i \rightarrow P_j$ if rollback of P_i induces rollback of P_j. An example is presented in Fig. 3, where process P_1 fails at time t(f). Since there is an interaction between P_1 and P_2 during the time interval $(t(n_1), t(f))$, process P_2 must roll back to enable the interaction in the reexecution of P_1, denoted by $P_1 \rightarrow P_2$. The rollback of process P_2 will propagate further to other processes, in this example, $P_2 \rightarrow P_4$, $P_1 \rightarrow P_3$, and $P_3 \rightarrow P_2$.

Because of asynchronous state-save nature, the rollback may propagate further to the state

beyond the previous state. In the above example, P_2 must roll back to the state at $t(n_2-1)$ because of the propagation P_3--->P_2; namely this failure requires a multi-step rollback. In the worst case, this fact may introduce unbounded rollback propagations (eventually to the beginning of the processes). In the automatic rollback recovery mechanism, we only accommodate one saved state in each PM to minimize the storage overhead. Hence the whole task has to be restarted if a multi-step rollback is required.

3.2 The Detection of Rollback Propagation

Since every interaction is managed by the cluster node, the cluster node should take responsibility for detecting rollback propagation and deciding if a multi-step rollback is needed. Let a given task be decomposed into N cooperating processes that are assigned to N PMs and executed simutaneously. Two detection methods are proposed as follows:

A. Method 1

Let the array element K(i,j) represent the number of interactions between P_i and P_j during the current state-save interval of P_i. Because of the time disparity in saving states for P_i and P_j, the values of K(i,j) and K(j,i) may be different. The cluster node counts the number of interactions and checks the array when failure occurs as follows.

1. When process P_i saves its state and moves to the next state-save interval, reset K(i,j)=0 for j=1,2...N.
2. When the cluster directs a reference issued by process P_i to P_j, K(i,j) and K(j,i) are incremented by 1.
3. When P_i fails or is affected by another rollback, the cluster node examines K(i,j) for j=1,2...N. If and only if K(i,j)=0, there is no direct propagation from P_i to P_j. If K(i,j)≠0 and K(i,j)=K(j,i), then P_j has to roll back one step. If K(i,j)≠0 and K(i,j)≠K(j,i), the rollback will propagate more than one step.

The condition under which rollback propagates more than one step occurs when (1) there is an interaction across the different state-save intervals of P_i and P_j (see an example in Fig. 3, where P_3 interacts with P_2 in the interval $(t(n_3),t(n_2))$), and (2) P_i and P_j both need to roll back.

From steps 1 and 2, K(i,j) represents the number of interactions between P_i and P_j in the current state-save interval of P_i. To prove the correctness of this method, all possible cases are considered in Fig. 4. For cases (a) and (b), it is obvious that P_j does not have to roll back. For case (c), P_j must roll back, leading to the fact that the interactions during the present state interval of P_i and P_j are reproduced. For the cases in (d), (e) and (f), there are interactions between the state-save instants of P_i and P_j. The single-step rollback of P_i or P_j is not sufficient to cover the all interactions needed in the reexecution. In the above example of Fig. 3, the array of K_{4x4} at time t(f) is

$$\begin{bmatrix} 0 & 2 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

If the processor executing P fails at t(f), the examination of array K_{4x4} concludes that P_2, P_3 and P_4 have to roll back. Since K(3,2)≠K(2,3) and K(3,2)≠0, a multi-step rollback is required.

The time required for deciding if the processes have to roll back or not, called decision delay, depends on the number of comparisons in Step 3. The number of comparisons is determined by the interaction patterns among processes. Since the checking of K(j,i) is not necessary if P_i--->P_j, the maximum number of comparisons would be N(N-1)/2 for N cooperating processes. The memory space for storing the K array increases proportionally to N^2. If N is large, the storage overhead and the decision delay may become a burden to system performance.

B. Method 2

Two condition array elements KC(i,j) and KP(i,j) are used to represent the interactions between processes P_i and P_j. Both arrays are composed of N*N bits, each array element consisting of a single bit. If an interaction occurs from P_i to P_j during the current state-save interval of both P_i and P_j, then KC(i,j)=1. If this interaction occurred in P_j's previous state-save interval, then KP(i,j)=1. The steps for setting these array elements and checking the rollback propagation are as follows:

1. When an interaction is issued by P_i and directed to P_j, then KC(i,j) and KC(j,i) are set to 1.
2. If process P_i saves its states and moves to the next state-save interval, then for j=1,2...N
   (a). KP(j,i):=KP(j,i) + KC(i,j)
        KC(j,i):=0
   (b). KC(i,j):=0
        KP(i,j):=0
   where + is logical OR operation and step (a) is done before step (b).
3. When P_i rolls back, the cluster node checks two rows in each array, namely KC(i,j) and KP(i,j) for j=1,2...N. There are three possibilities: 1). If KP(i,j)=1, then a multi-step rollback occurs. 2). If KP(i,j)=0 and KC(i,j)=1, process P_j has to roll back a single step. 3). If KP(i,j)= KC(i,j)=0, then there is no direct rollback propagation from P_i to P_j.

The proof of Method 2 can be done similarly to the first one. Although the value of KC(i,j) is always equal to KC(j,i), we added this redundancy to speed up the detection process and make the hardware implementation easy. Two arrays are arranged as Fig. 5 such that the checking of a row can be performed at one time. Hence the maximum number of row checkings would be N for N cooperating processes.

4. ESTIMATION AND ANALYSIS OF PERFORMANCE

4.1 Performance Estimation

To evaluate the performance of the automatic rollback recovery mechanism, the expected execution time of a given task has to be determined. Since certain additional processes are inserted into the normal processes, and since multi-step rollbacks are intentionally avoided, the time overhead and the risk of restart should be studied. Suppose a task is partitioned into N processes which are allocated to N PMs and executed simultaneously. Let T_{nd} represent the time spent to complete this task under a fault-free condition. The real execution time, T_t, includes T_{nd}, as well as the time overhead for generating recovery blocks, T_{ov}, and the time required to recover from failures, T_{rec}. An expression of expected execution time, $E(T_t|T_{nd})$, is derived under the following assumptions:

1. The time interval between two consective failures can be described by an exponentially distributed random variable with a mean of $1/\lambda$. For simplicity an error is assumed to be discovered immediately whenever it occurs. Therefore the occurrence of recovery can be modelled as Poisson process with a mean time between recovery or restart that equals $1/\lambda$.
2. The system has a sufficient number of processor modules so that the task may be executed continuously from start to completion. The time needed for fault-free task execution, T_{nd}, is assumed to be independent of system reconfiguration. This is true if the failed process is migrated to the replacement PM in the same cluster as the failed PM.
3. In addition to the set-up time for the newly configured system, a further failure may occur when the task is reexecuted following the recovery from a failure.

For purpose of comparison, the model proposed by Castillo and Siewiorek [12] is introduced for a system without a rollback mechanism. In this model the processes must be restarted whenever failure occurs. A typical sequence of processing is shown in Fig. 6b. The expected recovery time, $E(T_{rec})$, for a given fault-free execution time T_{nd} is expressed by

$$E(T_{rec})=(t_{su}+1/\lambda)(\exp(\lambda T_{nd})-1)-T_{nd} \qquad ---(1)$$

where t_{su} is the time required to set up the restart operation. Since there is no time overhead for generating recovery blocks (i.e. $T_{ov}=0$), the expected execution time can now be expressed by

$$E(T_t|T_{nd})=E(T_{nd})+E(T_{rec})=T_{nd}+E(T_{rec})$$
$$=(t_{su}+1/\lambda)(\exp(\lambda T_{nd})-1) \qquad ---(2)$$

For a system with a rollback recovery mechanism, an additional overhead is introduced into the normal process due to the validation process and state-save operation. Since the state-save operation is done in a "nearly" concurrent manner, and some error detections are embedded in the execution of processes, the significant factor should be the execution of validation process. If there is an erroneous state that is not uncovered by the embedded error detection mechanism, the rollback mechanism may not work for time-critical applications. The validation process is used to ensure the correctness of saved state and thus prevent error propagation to subsequent recovery blocks. The validation process can be simple and short if the embedded error detection mechanism covers most sources of error.

Assuming that a total time T_t is required to complete a given task and T_{ss} is the duration between two consecutive state-save invocations, we can represent the time overhead as follows:

$$T_{ov}\leq(T_t/T_{ss})(t_v+t_s) \qquad ---(3)$$

where T_{ov} is the time overhead for the construction of recovery blocks, t_v is the time required to execute the validation process, and t_s is the time used for state-save. Both sides are equal when there is a state-saving immediately following each state-save invocation. t_s can be expressed as the sum of the time needed to save the internal state of a processor(t_{is}) and the time needed for the transfer or update of state-save units (t_{ssu}). With the scheme described in Section 2, the transfer of updated elements is executed in parallel with non-memory-update operations. Note that the number of non-memory-updates is generally greater than the number of memory-updates. Therefore, t_{ssu} can be assumed negligible. Then the time overhead becomes

$$T_{ov}\leq T_t(T_{sv}/T_{ss}) \qquad ---(4)$$

where $T_{sv}=t_v+t_s$.

In FTMR²M, processes may roll back to the previously saved state or restart the whole task after a failure. Fig. 6a shows the sequence of processing in which the processes may have to restart or roll back after a failure. Let the

probabilities of rollback and restart be P_b and P_s, respectively where $P_b+P_s=1$. Let the set up time of a process after each failure be constant and equal to t_{sb} for rollback (t_{su} for restart). The amount of computation loss in rollback recovery is equal to the operation that has been done between the previous state-save instant and the time instant of failure. This duration, called rollback distance, is a random variable distributed from 0 to $T_{ss}+T_{sv}$. To simplify the derivation of the expected execution time, the rollback distance for each failure is assumed to be constant, denoted as r, and the completion of the task will be delayed by r as a result of each rollback recovery (according to our simulation, this assumption has little effect on performance if $T_{nd} >> T_{ss}$). The time T_{ss} is assumed to be so large that there is always a state-saving for each state-save invocation. If r is set to $T_{ss}+T_{sv}$, the result should be the upper bound of the total execution time. The results derived in the Appendix are given in the following:

$$E(T_t|T_{nd})=(t_{su}+1/(\lambda P_s))(\sum_{m=1}^{\infty} P_r(m) \exp(\lambda P_s(T_{nd}+m(r+t_{sb}))/(1-\nu))-1) \quad ---(5)$$

$$P_r(m)=(1/m!)(\lambda P_b T_{nd})^m(1+mr/T_{nd})^{m-1} \exp(-\lambda P_b T_{nd}) \quad ---(6)$$

where $\nu=T_{sv}/T_{ss}$, and $P_r(m)$ is the probability of occurrence of having m rollbacks before completion of the task.

4.2 Discussion of Performance Models.

From the above estimation, several dependent relations are studied. In Fig. 7, the probability density function of $P_r(m)$ is shown. These curves are similar to those of a Poisson process except for the slight difference due to the recursive occurrences of failure following the recovery from a failure. Figures 8 and 9 express the relation between the expected time wasted for the recovery from faults and the time needed for fault-free execution. The major effect of the rollback recovery mechanism seems to be that the the mean time between failure is enlarged from $1/\lambda$ to $1/(\lambda P_s)$.

In Eq. (5) the length of the state-save invocation interval, T_{ss}, has two mutually conflicting effects. First, an increase of T_{ss} will decrease the percentage of time overhead. On the other hand, the average computation loss by rollback is proportional to the state-save duration because the occurrence of failure is distributed throughout the state-save interval. Futhermore, the amount of interaction within one state-save interval increases with the length of this interval. This increase implies a high possibility of rollback propagations in case of failure. Therefore the increase of T_{ss}, which invokes longer state-save intervals, will introduce more computation loss and higher probability of restart. The optimal value of T_{ss} can be found in Fig. 10, where the percentage of the time lost for fault recovery vs. the percentage of time overhead for generating recovery blocks is plotted. The curves seem to be a straight line if T_{ss} is small (namely T_{sv}/T_{ss} is large) until they reach a minimum which shifts to the right with increasing failure rate. After this minimal point, an increase of rollback distance makes the recovery time increase considerably.

The performance of the rollback recovery mechanism is significantly dependent upon the probability of restart. The probability of restart after a failure, P_s, is a random variable. The distribution of P_s depends on three factors:

1. the hit ratio within each process,
2. the length of the state-save invocation interval (T_{ss}), and
3. time disparities in the actual state-saves among cooperating processes following each state-save invocation.

Apart from the assignment of optimal T_{ss}, the probability of restart can be improved in three ways:

1. Allocate a shared variable to a PM in which the resident process refers to this shared variable most frequently. Since the reference to a shared variable that resides in the same PM can not be across different state-save intervals, the probability of having a multi-step rollback should decrease.
2. Decompose the task into cooperating processes such that the amount of interaction is minimized. This aspect is one of the major unsolved issues in any multiprocessor system. A suitable decomposition will certainly enhance the hit ratio and decrease P_s.
3. Accommodate a third state-save unit to provide two saved states for rollback. In this arrangement a process is allowed to roll back more than one step. If the probability of restart in the single-step rollback enviroment is P_s, then the probability of restart with three state-save units would be clearly less than P_s. To detect a two-step rollback, additional arrays are needed to identify the interactions during the previous state-save interval. When a rollback propagates beyond the current state-save interval, these additional arrays will be examined by the same methods in Section 3 to decide whether the rollback propagates further or not.

5. CONCLUSION

Emphasis in the design of FTMR²M has been placed on the fast state-save scheme that allows rollback recovery with little time overhead. To permit processes to be general and to ensure programmer-transparency, recovery points are established automatically and regularly. This approach does not require high-level insertion strategies or limitations in the setting of recovery points [4, 6, 7], and also does not require synchronization of state-save operations of different PMs as does the COPRA system [5]. But due to the lack of synchronization among state-savings of processes, the task may be required to restart after a failure. This risk would result in a substantial influence on the performance (particularly in case of heavy interactions).

Because of the reconfigurability of a multiprocessor system, the failure rate of a processor can not represent the reliablity of the system. Besides the soft-fail capability, the system should be characterized by its performance when a failure occurs. The expected execution time is useful to indicate performance in this case. In the above analysis, an upper bound of expected execution time is provided. Precise analysis is difficult because: (1) The rollback distance is a random variable that depends on the occurrence of error and the error detection processes. Some errors can not be discovered immediately following their occurrence, in which case the rollback can not be modelled as the Poisson process with the same parameter, λ, as the occurrence of failure. (2) Since certain processes of a given task are not affected by the rollback of a failed process, the increase of execution time by each rollback is not equal to the rollback distance.

One major concern in the implementation of an automatic rollback recovery mechanism is modularity and simplicity. The individual rollback mechanism associated with each processor module offers system modularity and simplicity. Hence the present fault-tolerant multiprocessor has a high potential use for critical real-time applications such as aircraft or industrial control, among others.

REFERENCE

1. A. M. Feridun and K. G. Shin, "A Fault-Tolerant Multiprocessor System with Rollback Recovery Capabilities", Proc. 2nd Int'l Conf. on Distributed Computing System, April 1981.
2. B. Randell, "System Structure for Software Fault Tolerance", IEEE Trans. on Software Eng., Jun. 1975, pp. 220-232.
3. K. M. Chandy and C. V. Ramamoorthy, "Rollback and Recovery Strategies for Computer Program", IEEE Trans. on Comp., June 1972, pp. 546-556.
4. F. T. O'Brien, "Rollback Point Insertion Strategies", Proc. of the 6th Int'l Symp. on Fault-Tolerant Computing, Pittsburg, 1976, pp. 138-142
5. K. Kant and A. Silberschatz, "Error Recovery in Concurrent Processes", Proc. COMPSAC 80,, Fall 1980, pp. 608-614.
6. C. Meraud and F. Browaeys, "Automatic Rollback Techniques of the COPRA Computer", Proc. of 6th Int'l Conf. on Fault-Tolerant Computing, 1976, pp. 23-29.
7. K. H. Kim, "An Approach to Programmer-Transparent Coordination of Recovering Parallel Processes and its Efficient Implementation Rules", Proc. 1978 Int'l Conf. on Parallel Processing, Aug. 1978, pp. 58-68.
8. K. H. Kim, "An Implementation of a Programmer-Transparent Scheme for Coordinating Concurrent Processes in Recovery", Proc. COMPSAC 80, Fall 1980, pp. 615-621.
9. R. J. Swan, S. H. Fuller, and D. P. Siewiorek, "Cm*: a Modular Multi-Microprocessor", AFIPS Conf. Proc., Vol. 46, 1977, pp. 637-644.
10. K. H. Kim, "Error Detection, Reconfiguration and Recovery in Distributed Processing System", Proc. Int'l Conf. on Distributed Computing Systems, Oct. 1979, pp. 284-295.
11. S. H. Fuller, J. K. Ornstein, L. Raskin, P. I. Rubinfeld, P. J. Swan, "Multi-Microprocessors: An Overview and Working Example", Proceedings of the IEEE, Vol. 66, No. 2, pp. 216-228, Feb. 1978.
12. X. Castillo, D. P. Siewiorek, "A Performance-Reliability Model for Computing Systems", 10th Int'l Conf. on Fault-Tolerant Computing, 1980, pp. 187-192.

APPENDIX

1. The Derivation of Probability of m Rollbacks, $P_r(m)$

Suppose the rollback distances are the same for every rollback, denoted by r. The time between two successive rollbacks is assumed to be a random variable with the exponential distribution. The parameter of the distribution would be λP_b, the average rate of rollback recovery. For a small time interval h, the probability of having no rollback would be $1-\lambda P_b \cdot h$. The probability of having m rollbacks from time 0 to t is governed by the following difference equation.

$$P_r(m,t+h)=P_r(m,t)(1-\lambda P_b h)+P_r(m-1,t)\lambda P_b h P_r(0,r)+$$
$$\cdots\cdots\cdots +P_r(0,t)\lambda P_b h P_r(m-1,r)$$

then $\frac{d}{dt}(P_r(m,t))=\lambda P_b P_r(m,t)+\lambda P_b P_r(m-1,t)P_r(0,r)$
$$+\cdots\cdots\cdots+\lambda P_b P_r(0,r)P_r(m-1,r)$$

From $P_r(0,t)=\exp(-\lambda P_b t)$, we can get

$$P_r(1,t)=(\lambda P_b t)\exp(-\lambda P_b(t+r))$$
$$P_r(2,t)=2(\lambda P_b t)(1+2r/t)\exp(-\lambda P_b(t+2r))$$
$$\vdots$$
$$P_r(m,t)=(1/m!)(\lambda P_b t)^m(1+mr/t)^{m-1}$$
$$\exp(-\lambda P_b(t+mr))$$

2. The Derivation of the Expected Execution Time

Let $T_t=T_{real}+T_{res}$, where T_{res} is the time spent for the restart and T_{real} is the sum of T_{nd}, the time spent for the rollback recovery, and the time overhead for a state save.

Then $E(T_t)=E(T_{real}+E(T_{res}|T_{real}))$. From 14,

$E(T_{res}|T_{real})=(t_{su}+1/\lambda_s)(\exp(\lambda_s T_{real})-1)-T_{real}$

Where $1/\lambda_s$ is the mean time between restarts and $\lambda_s=\lambda P_s$. For the case of having m rollbacks before completion,

$$T_{real,m}=T_{nd}+m(r+t_{sb})+((T_{real,m}/T_{ss})T_{sv})$$

where r represents the rollback distance.
Let $T_{sv}/T_{ss}=\nu$, then

$$T_{real,m}=(T_{nd}+m(r+t_{sb}))/(1-\nu)$$

Then $E(T_t|T_{nd})=(t_{su}+1/\lambda_s)(E(\exp(\lambda_s T_{real}))-1)$
$$=(t_{su}+1/\lambda_s)(\sum_{m=1}^{\infty}P_r(m)\exp(\lambda_s T_{real,m})-1)$$

P = Processor
M = Local Memory
S = Local Switch
MS = Monitor Switch
SSU = State-save Memory Unit
CS = Cluster Switch
CM = Cluster Monitor
DMAC = DMA Controller

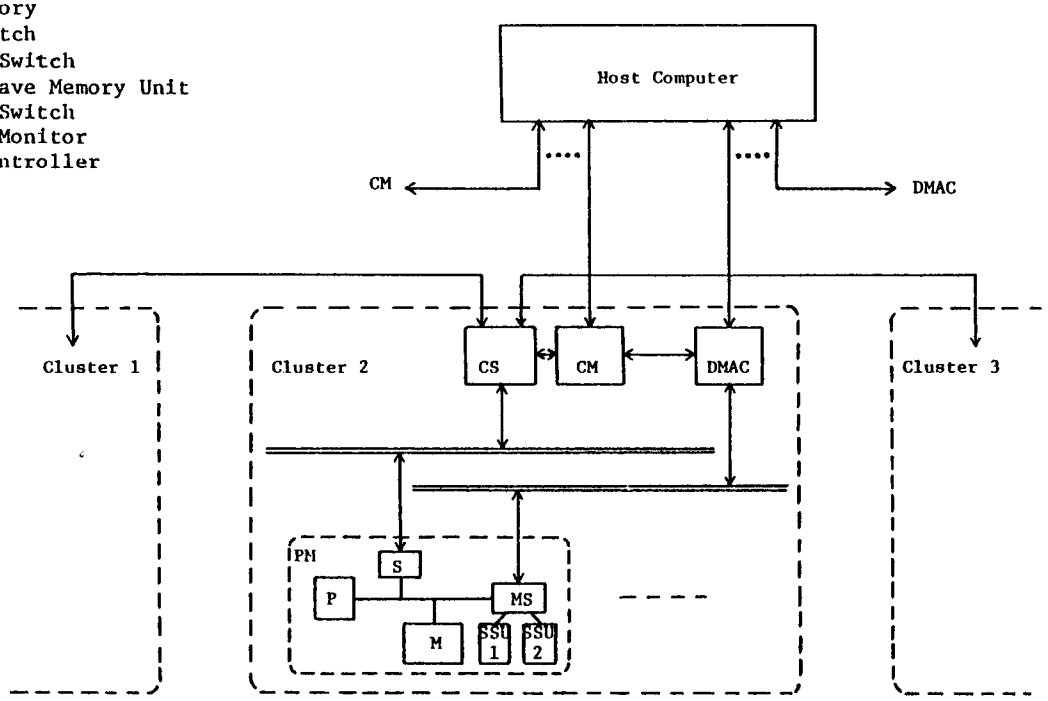


Fig. 1. Rollback Recovery Multiprocessor Structure

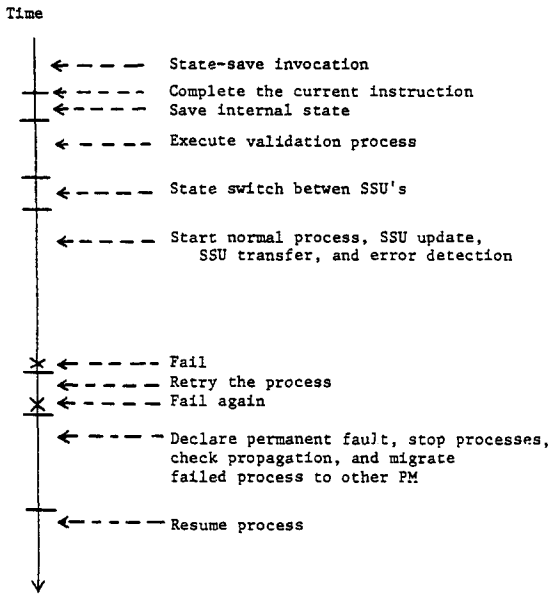


Fig. 2. Sequence of a Rollback Recovery

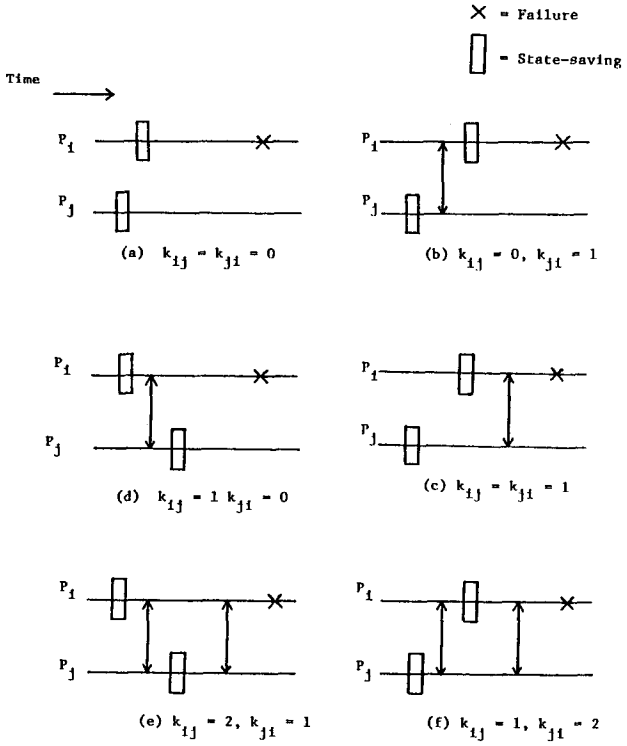


Fig. 4. Various Conditions of Occurrence of Interaction

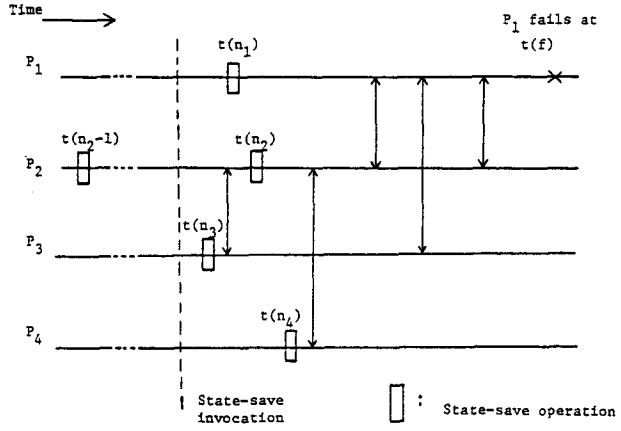
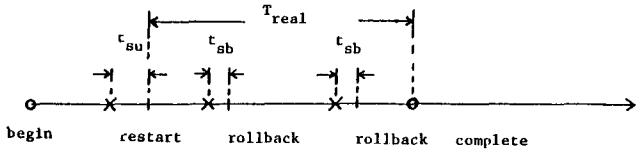


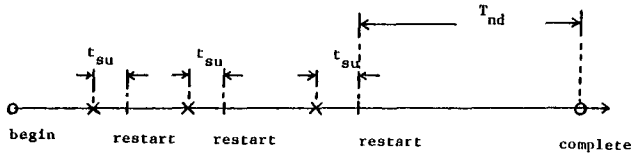
Fig. 3. Example of Rollback Propagation

		P_1	P_2	P_3	.....	P_N
P_1	KC(1,j):	0	1	1	.....	1
	KP(1,j):	0	0	1	.....	0
P_2	KC(2,j):	1	0	1	.....	1
	KP(2,j):	0	0	0	.....	0
	⋮					⋮
P_N	KC(N,j):	1	0	0	.....	0
	KP(N,j):	0	0	1	.....	0

Fig. 5. The Arrangement of Array KC_{NxN} and KP_{NxN}



(a) The case with rollback capability



(b) The case without rollback capability

Fig. 6. The Sequence of Execution in Two Structure

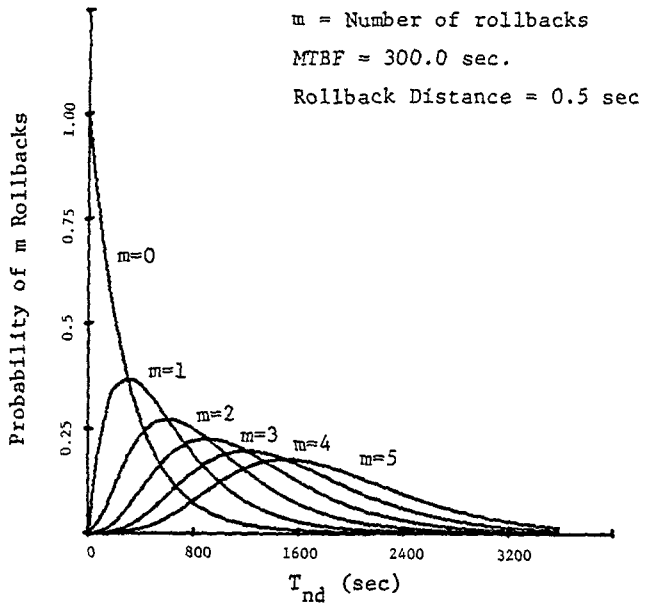


Fig. 7. Probability of m Rollbacks

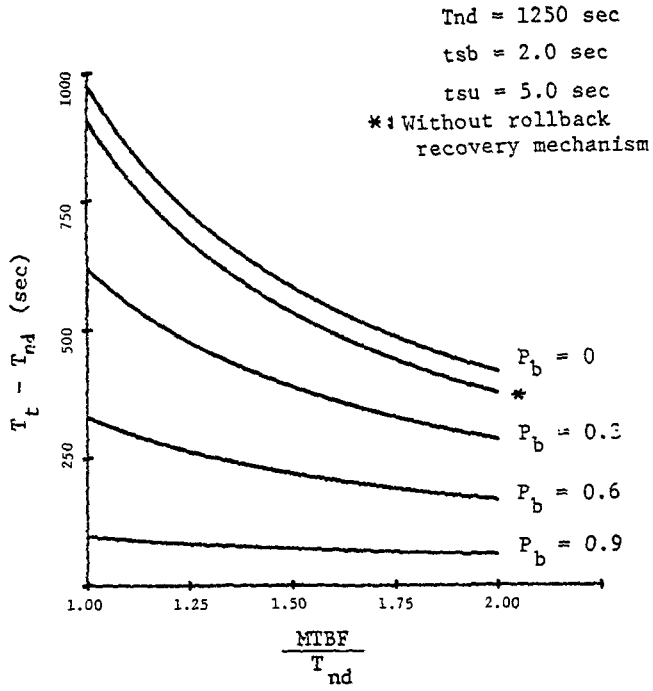


Fig. 9. Effect of Automatic Rollback Recovery (2)

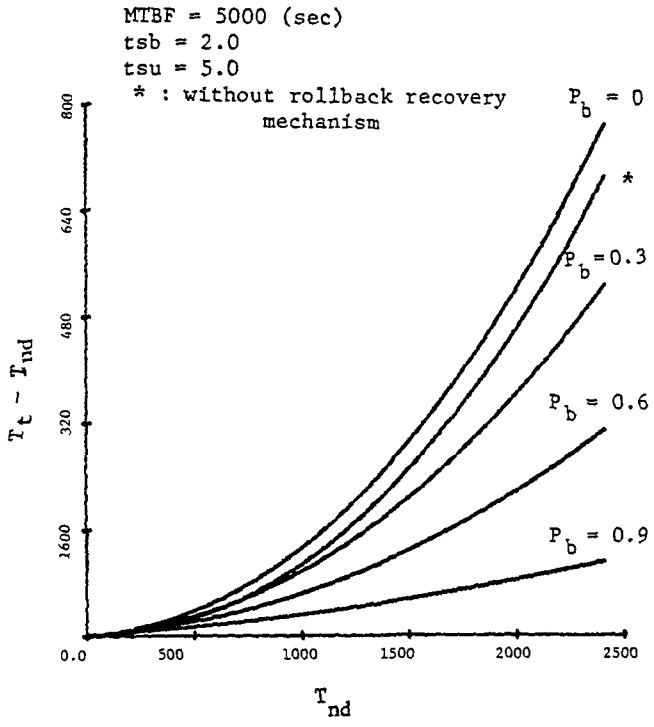


Fig. 8. Effect of Automatic Rollback Recovery (1)

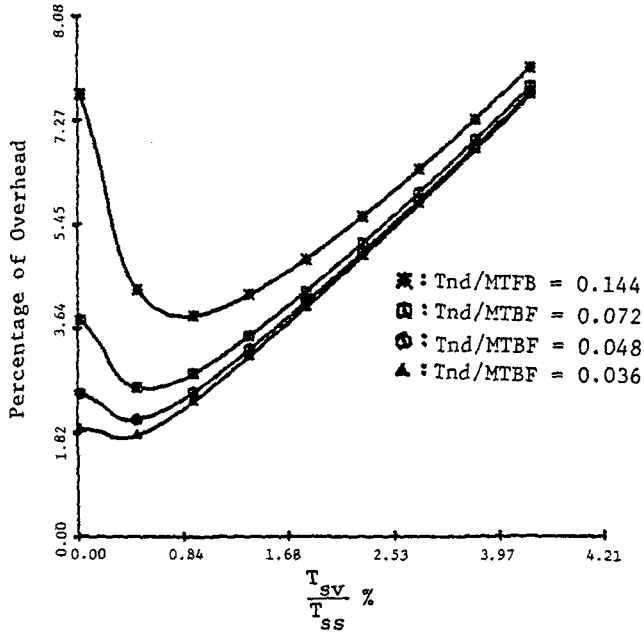


Fig. 10. Relative Overhead vs. State-save Invocation Interval