Design and Evaluation of a Fault-Tolerant Multiprocessor Using Hardware Recovery Blocks

YANN-HANG LEE, STUDENT MEMBER, IEEE, AND KANG G. SHIN, SENIOR MEMBER, IEEE

Abstract—**In this paper we consider the design and evaluation of a fault-tolerant multiprocessor with a rollback recovery mechanism.**

The rollback mechanism is based on the hardware recovery block which is a hardware equivalent to the software recovery block. The hardware recovery blocks are constructed by consecutive state-save operations and several state-save units in every processor and memory module. Upon detection of failure, the multiprocessor reconfigures itself to replace the faulty module and then the process originally assigned to the faulty module retreats to one of the previously saved states in order to resume fault-free execution.

Due to random interactions among cooperating processes and also due to asynchrony in the state-savings, the rollback of a process may propagate to others and thus the need of multiple-step rollbacks may arise. In the worst case, when all the available saved states are exhausted, the processes have to restart from the beginning as if they were executed in a system without any rollback recovery mechanism. A mathematical model is proposed to calculate both the coverage of multistep rollback recovery and the risk of restart. Also presented is the evaluation of mean and variance of execution time of a given task with occurrence of rollbacks and/or restarts.

Index Terms—**Fault-tolerant multiprocessor, hardware/software recovery blocks, performance of rollback recovery mechanisms, rollback propagation.**

I. Introduction

THERE are numerous benefits to be gained from a multiprocessor. In addition to the decreasing of hardware costs and the inherent reliability of LSI components, the capacity of reconfiguration makes the multiprocessor even more attractive when system reliability is important. It is particularly essential to critical real-time applications that the system be tolerant of failure with minimum time overhead and that the task be completed prior to the imposed deadline. Hence, one of the major issues of reliable multiprocessor design is to provide the capability of error recovery without having to restart the whole task in case of failure.

In general, the tolerance of failure during system operation is achieved by three steps: detection of error, reconfiguration of system components, and recovery from error. The purpose of error detection is to recognize the erroneous state and to prevent a consequent system failure. There are two basic design approaches for error detection: 1) detect an error upon its occurrence, and 2) isolate the erroneous information before it is propagated. For the first approach, the most widely used techniques are error detection/correction coding, addition of built-in checking circuits (e.g., voting hardware), etc. Error detection schemes such as consistency test, execution of validation routines, or acceptance test are typical examples for the second approach. Following the detection of error, the faulty components, which are the source of error, are localized and replaced so as to enable the system to be operational again. In order to recover from an error, the rollback recovery method or the reinitialization of a fault-free subsystem is usually invoked in order to resume the failed computation. Both methods consist of state restoration and recovery point establishment. In JPL-STAR system [1] the recovery points are defined by the application program which also takes the responsibility of compensating for the information prior to the recovery point. Hence, its error recovery capability is constructed in the application software level. On the other hand, the strategies used in PLURIBUS [2] are to organize the hardware and software components into reliable subsystems and to mask errors above the interface level of a subsystem. When an error is detected, the subsystem performs backward recovery by restarting the subsystem.

The conventional restart recovery technique could be costly and inept since 1) the computation between the start of task and the time when error is detected has to be undone, and 2) if the task is distributed over different processing units in the multiprocessor, it is difficult to provide a consistent task state and to isolate a subtask to prevent the propagation of erroneous information to others. (This may lead to the restarting of the entire task and may result in high reinitialization overhead.) The rollback recovery method at the software level is devised to tolerate *design faults* but may not be effective for tightly coupled processes since 1) the software recovery points by themselves in each process are not sufficient to recover the task unless they belong to the same recovery line [3], and 2) the program designers have to structure carefully the parallel processes so that the interacting processes establish recovery points in a well-coordinated manner. Several alternatives have been proposed; for example, the conversation scheme [4], the interprocess communication primitives in a producer-consumer system [5], the programmer-transparent scheme [6], [7], the system defined checkpoints [8], the decentralized recovery control protocol [9], etc. These methods could lead to a loss of efficiency in the absence of error, the accumulation of a

Manuscript received July 30, 1982; revised May 15, 1983. This work was supported in part by NASA under Grant NAG 1-296. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the funding agency.

The authors are with the Department of Electrical and Computer Engineering, University of Michigan, Ann Arbor, MI 48109.

large amount of recorded states for heavy interprocess communications, or some undesirable restrictions in communication schemes.

However, the concept of the recovery block, proposed by Randell [3], [4] and Horning [10], can still be useful for tolerating *hardware faults* in the multiprocessor. In this paper, we employ this concept to construct a hardware recovery block which enables the task to survive *processor* or *memory failures*. In order to resume a failed process, an error-free process state — which includes the status of internal registers of the assigned processor and the process variables stored in memory — should be restored. The hardware recovery block is constructed in a quasi-synchronized manner and saves all states of a process consecutively and automatically. This happens in parallel with the execution of the process by using a special state-save mechanism implemented in hardware. The hardware recovery block is different from the software recovery block which only saves nonlocal states when a checkpoint is encountered. Moreover, instead of the assertions in the acceptance test of the software recovery block, the hardware resources are tested by embedded checking circuits and diagnostic routines. After an error is detected and the faulty component is located, the system will be reconfigured to replace the failed hardware module. By loading the program code and transferring the recorded states into the replacement module, the original process can be resumed.

The multiprocessor with the hardware recovery block scheme takes advantage of the fact that a large number of processors and memory units can be made available at inexpensive costs for fast recovery from hardware failures. Furthermore, it only requires a minimal amount of time for establishing recovery blocks, which in turn significantly improves system performance.

For both hardware and software recovery blocks, the rollback of the failed process alone to the previous state is not sufficient for concurrent processing. The rollback of one process may propagate to other processes or to a further recorded state. (This is called *rollback propagation*.) The worst case is when an avalanche of rollback propagations, namely the *domino effect*, occurs. The domino effect is impossible to avoid if no limitation is placed on process interactions [8]. Instead of placing any of such limitations, several consecutive states are saved so that the processes are allowed to roll back multiple steps in case of rollback propagation. (This method is here termed the *automatic rollback recovery*.) The coverage of a multistep rollback — the probability of having a successful rollback recovery when cooperating processes roll back multiple steps — should be examined to decide the effectiveness of this method. Both the recovery overhead and the computation loss resulted from this automatic rollback recovery mechanism should also be studied carefully. Furthermore, since the time interval between two consecutive state savings is related to the final performance figure of this method, the optimal value of this interval has to be determined.

This paper is divided into five sections. Since the construction of hardware recovery blocks in the multiprocessor plays a basic role, we review it briefly in Section II. The detailed description can be found in [11], [12]. In this section, we also extend our previous design to a general multiprocessor on which our hardware fault recovery can be implemented. Section III presents an algorithm to detect rollback propagations among cooperating processes and also proposes a model to evaluate the coverage of multistep rollback recovery. Section IV uses the results of Section III and deals with the analysis and estimation of performance in terms of the mean and variance of the task completion time. The paper concludes with Section V.

II. Automatic Rollback Mechanism for a Multiprocessor

The multiprocessor under consideration has a general structure and consists of processor modules, interconnection network, and/or common memory modules. To benefit from the locality of reference, every processor module owns its local memory which is accessible via a local bus. Every processor module can also access the shared memory through the interconnection network. The rollback recovery operations of a task can be applied to two types of multiprocessors: in one, there is no common memory, but local memory of one processor module is accessible by other processor modules (e.g., Cm* system [13]); in the other, the system is equipped with separate common memory modules [14] and restricts the access of local memory only to the resident processor. These two types are representatives of contemporary general-purpose multiprocessors.

A. Processor Module, Common Memory, and State-Save Mechanism

A basic processor module (PM) in the multiprocessor comprises a processor, a local memory, a local switch, state-save memory units (SSU's), and a monitor switch as shown in Fig. 1. It is assumed that a given task is decomposed into processes each of which is then assigned to a processor module. The shared variables among these cooperating processes are located in the shared memory which is either separate common memory or local memories depending upon the multiprocessor structure discussed above. Thus, each process in a PM can communicate with other processes (allocated to other PM's) through the shared variables. Each PM saves its states (i.e., process local variables and processor status) in SSU's at various stages of execution; this operation is called a *state-save*. Ideally, it would be preferable to save states of all processes at the same instant during the execution of task. Because of the indivisibility and asynchrony of instruction execution in PM's, it is difficult to achieve this ideal case without forced synchronization and the consequent loss of efficiency. In order to alleviate this problem, we employ a quasi-synchronized method in which an external clock sends all PM's a state-save invocation signal at a regular interval, T_{ss}. This invocation signal will stimulate every PM to save its states as soon as it completes the current instruction and then to execute 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 or an error is detected during execution of

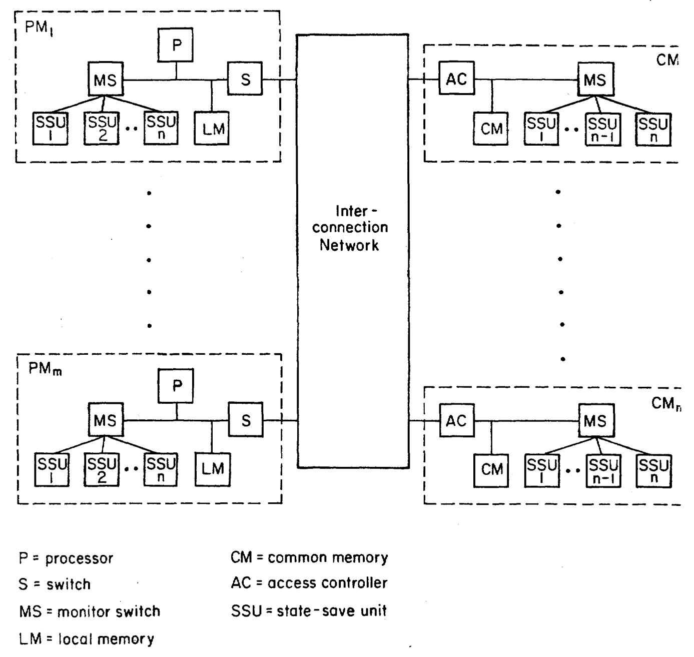


Fig. 1. The organization of a fault-tolerant multiprocessor using a rollback recovery mechanism.

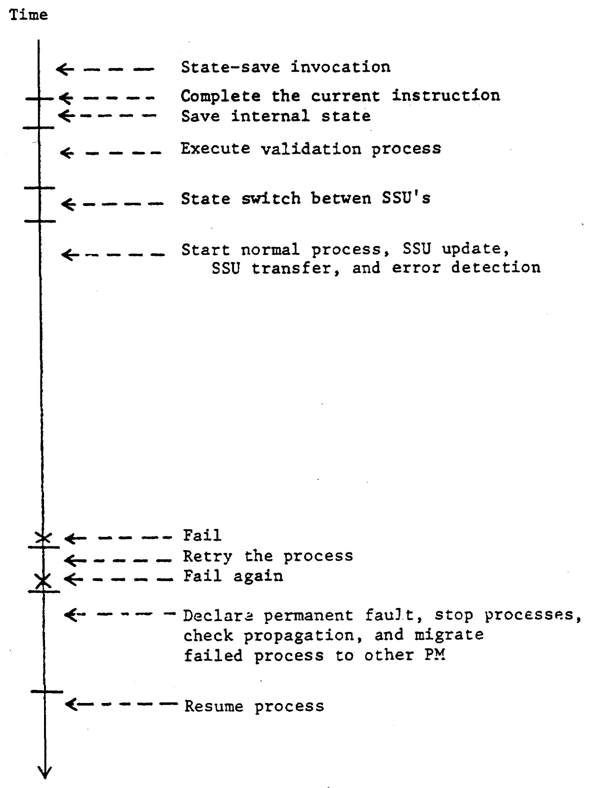

Time
State-save invocation
Complete the current instruction
Save internal state
Execute validation process
State switch betwen SSU's
Start normal process, SSU update, SSU transfer, and error detection
Fail
Retry the process
Fail again
Declare permanent fault, stop processes, check propagation, and migrate failed process to other PM
Resume process

Fig. 2. Sequence of a rollback recovery.

the resident process, the system will be reconfigured to replace the faulty component and the associated process will roll back to one of the previously saved states. The detailed operations of state saving and rollback recovery are shown in Fig. 2.

Similarly to a processor module, each common memory module (CM) also contains state-save memory units and a monitor switch. These SSU's are used to record the updates of CM only. The access requests of CM are managed by an access queue on the basis of the first-come-first-serve discipline. When a PM refers to a variable resident in a CM, an access request is sent to the destination CM through the interconnection network and enters the access queue associated with the CM. When all the preceding requests to this CM are completed, the access request will be honored and a reply will be sent back to the requesting PM. When a state-save invocation is issued, a state-save request is placed at the tail of every access queue. Thus, the state-save in CM is performed when the requests made prior to the state-save invocation have been completely serviced.

During a state-save interval, besides the normal memory reference or instruction execution, certain operations are automatically executed; for example, an error correcting code is used whenever a data is retrieved from memory. Some redundant error detection units also accompany the processor module [15], e.g., dual-redundancy comparison, address-inbound check, etc. These units are expected to detect malfunctions whenever the corresponding function units are used. An additional validation process which could be the execution of a diagnostic routine is used to guarantee that the saved state be correct, and thus guards against the existing fault extending to the next state-save interval.

Suppose there are $(N + 1)$ state-save units for every PM (and every CM), called $\mathrm{SSU}_1, \mathrm{SSU}_2, \cdots \mathrm{SSU}_{N+1}$. These units are used for saving states at $(N + 1)$ consecutive state-save intervals. Thus, each PM or CM is able to keep N valid states saved in N SSU's and record the currently changing state in the remaining SSU. As shown in Fig. 3, the SSU_1, $\mathrm{SSU}_2, \cdots \mathrm{SSU}_N$ are so arranged to record the states for consecutive state-save intervals $T(i + 1), \cdots T(i + N)$ and the SSU_{N+1} is used to record the updates in the current state-save interval, $T(i + N + 1)$. To minimize the time overhead required for state-saving, the saving is done concurrently with process execution. Every update of variables in the local memory is also directed to the current SSU. When a PM or CM moves to the next state-save interval, each used SSU will age one step and the oldest SSU will be changed to the current position if all SSU's are exhausted. The monitor switch is used to route the updates to SSU's and to manage the aging of SSU's. Therefore the state-save mechanism of each PM or CM provides an N-step rollback capability. In Section III, we will show that only a small number of SSU's are sufficient to establish high coverage of rollback recovery for typical multiprocessor applications.

Since the update of dynamic elements is recorded in only one SSU, the other SSU's are ignorant of it. This fact may bring about a serious problem: the newly updated variables may be lost. In order to avoid this, it is necessary to make the contents of the currently updated SSU identical with that of the memory or to copy the variables that have been changed in the previous intervals into the current SSU. A solution to this problem has been discussed in our previous paper [11]. At each state-switching instant, the current SSU contains not only the currently updated variables, but also the previously updated variables. Consequently, the contents of the current SSU always represents the newest state of the PM or CM.

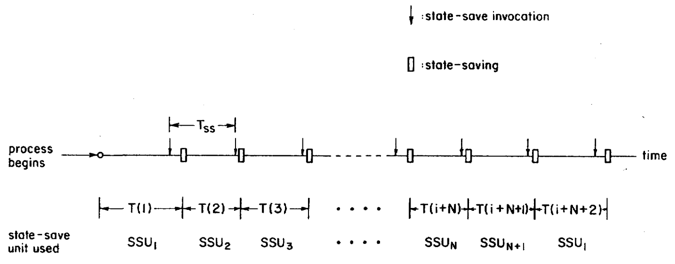


Fig. 3. State-save operations in one module.

B. Rollback Recovery Operations of a Task

Suppose a task is partitioned and then allocated to M modules ($i = 1, 2, \cdots, M$). These modules include PM's and CM's and will be dedicated to this task until its completion. The state saving of a task implies the state savings of these modules. The rollback of a process is equivalent to the state restoration of the associated modules. Since the process state includes the internal hardware states, local variables, and global variables, the resumption of a failed process may need cooperation from common memory and/or other processes. Moreover, due to arbitrary interactions between cooperating processes and the asynchrony in state savings among them, the rollback of one process may cause others to roll back and it is therefore possible to require a multistep rollback (a detail of this will be discussed in the next section). In order to make a decision as to rollback propagation and also to perform housekeeping jobs (e.g., task allocation, interconnection network arbitration, reconfiguration, etc.), a system monitor and a switch controller are included in the multiprocessor. The switch controller handles the global variables references and records these references for analyzing rollback propagation and multistep rollback. The system monitor receives the task execution command and then allocates PM's and CM's to perform the task. Both devices are defined in a logical sense. They could be a host computer, or a special monitor processor, or one of general processor modules in the system.

To deal with the error recovery, the system monitor receives reports from each module regarding the state-save operations and its conditions. Once an error is detected, the system monitor will signal "retry" to the module in question. If the error recurs, a permanent fault is declared and the following steps are taken by the system monitor and the switch controller.

1) Stop all PM's that are executing processes of the task in question.

2) Make a decision as to rollback propagation.

3) Resume execution of the processes that are not affected by rollback propagation.

4) Find a free module to replace the failed one.

5) Transfer the process or data in the failed module to the replacement module and reroute the path to map references directed to the faulty module into its replacement.

6) Restore the previous states of the processes affected by the rollback of the resident process in the faulty module.

7) Any interaction directed to a module to be restored must wait for the resumption of the module. Old and unserviced interactions issued by the rolled-back PM's, which are still queued in the access queues, are cancelled.

III. Rollback Propagation and Multistep Rollback

In order to roll back a failed process, the consistent values of the process variables and the internal states of the associated PM should be provided. The local variables and internal states which are saved in the SSU's of a PM are easily obtainable. However, the shared variables—which may be located in any arbitrary PM or CM and may be accessed by any process—bring about a difficult problem: the rollback of a failed process induces the rollback of other processes (i.e., rollback propagation occurs). The rollback propagation might result in another inconsistent state for certain processes of the task, thereby requiring a multistep rollback.

Furthermore, the hardware may have latent faults which are undetectable until they induce some errors. In the following discussion, we assume that an error will be detected immediately as it occurs. So the rollback propagation is used only to obtain a consistent state. However, it can be easily extended to the case in which error latency exists and is bounded by U [16] as follows.

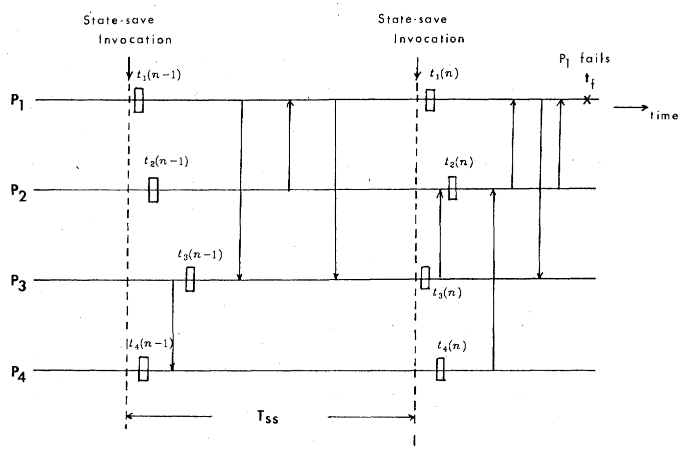
1) First obtain a consistent state which may entail rollback propagations, and calculate the total rollback distance D.

```
2) **If** $D \geq$ the total computation done **then** restart
     **else if** $D \geq U$ **then** done
         **else** go to step 1).
```

A. Rollback Propagation and Multistep Rollback

In general rollback propagation cannot be avoided if the processes interact with each other arbitrarily. For the multiprocessor organization in the previous section, a process is allocated to one PM and/or several CM's and each module has its own rollback recovery mechanism. So each module can be regarded as an object for rollback propagation. An interaction between cooperating processes is implemented as a memory reference to a shared variable, i.e., a memory reference across the modules. To avoid the need of tracing every reference to the shared variables and to simplify the detection of rollback propagation, we assume that the failure of a particular module leads to the automatic rollback of all modules that have interacted with the module during its current state-save interval. Let $P_i \rightarrow P_j$ denote the rollback propagation in which the rollback of process P_i induces the state restoration in one or more modules containing P_j, that is, the rollback of P_i causes P_j to roll back. Let the nth state-save interval of P_i be $T_i(n)$ and the beginning moment of $T_i(n)$ where P_i saved its state be at $t_i(n)$. An example is presented in Fig. 4, where process P_1 fails at time t_f and saves its state at $t_1(n)$ during state-save interval $T_1(n)$. Since interactions between P_1 and P_2 exist during the time interval $[t_1(n), t_f]$, process P_2 must roll back to revive the interactions when P_1 is resumed. The rollback of P_2 will propagate further to other processes; in this example, $P_2 \rightarrow P_4$, $P_1 \rightarrow P_3$, and $P_3 \rightarrow P_2$. When Wood's definitions [9] are used, the state of process P_1 saved at $t_1(n)$ can be regarded as a *potential recovery initiator* of the saved states of P_2, P_3, and P_4.

In the above example, we can find that the rollback of P_3



(a)

$$KC_2 = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \qquad KC_1 = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{bmatrix}$$

$$KP_2 = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \qquad KP_1 = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

(b)

$$RB_1(n) = \begin{cases} 1 & n \leq 2 \\ 0 & \text{otherwise} \end{cases} \qquad RB_2(n) = \begin{cases} 1 & n \leq 2 \\ 0 & \text{otherwise} \end{cases}$$

$$RB_3(n) = \begin{cases} 1 & n \leq 2 \\ 0 & \text{otherwise} \end{cases} \qquad RB_4(n) = \begin{cases} 1 & n \leq 1 \\ 0 & \text{otherwise} \end{cases}$$

(c)

Fig. 4. An example of rollback propagation and multistep rollback.

and P_2 to their most recently saved state still cannot provide a consistent task state. The reason that a rollback of cooperating processes cannot recover the process states is mainly due to the occurrence of references between the asynchronous state savings of interacting processes. For convenience, a restorable state for P_i is defined as follows.

Definition: Suppose process P_i rolls back to the state saved at $t_i(k)$. This state is *restorable* for P_i if either of the following two conditions is satisfied:

C1) P_i has no interaction with other processes during the state-save interval $T_i(k)$.

C2) The rollback of P_i to $t_i(k_i)$ induces the rollback of P_j to $t_j(k_j)$ for $j = 1, 2, \cdots, M$ and $j \neq i$, but there is no interaction needed to be reissued between P_i and P_j during the interval $[t_i(k_i), t_j(k_j)]$ if $t_i(k_i) \leq t_j(k_j)$ or $[t_j(k_j), t_i(k_i)]$ otherwise.

Consider the cases in Fig. 5. Suppose P_i rolls back to $t_i(k)$ because of failure or rollback propagation from another pro-

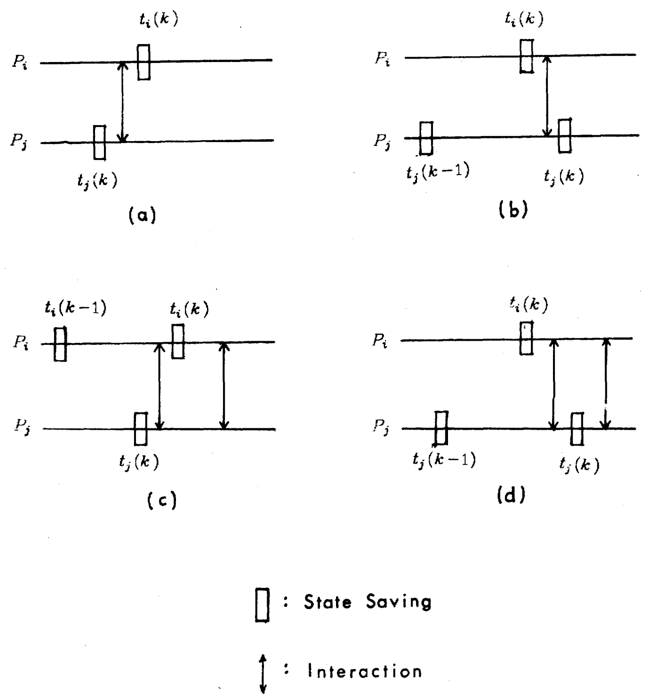


Fig. 5. Interaction patterns related to rollback propagation.

cess. In case a), the state saved at $t_i(k)$ is restorable for P_i only. A single step rollback of P_i is sufficient to recover its state. In cases b) and c), both P_i and P_j have to roll back and the states saved at $t_i(k)$ and $t_j(k-1)$ are restorable for P_i and P_j respectively, while in case d), the states at $t_i(k-1)$ and $t_j(k)$ become restorable.

The necessary condition for recovering a task TK, where $TK = \{P_i \mid i = 1, 2, \cdots, M\}$, with rollback mechanisms can be obtained from the above definition. The task TK is recoverable from a failure if for all i either P_i is not affected by the rollbacks of other processes or P_i rolls back to its most recently restorable state.

B. The Detection of Rollback Propagation

Since every external memory reference is managed by the switch controller, the switch controller should take responsibility for detecting rollback propagation and deciding on multistep rollbacks. Suppose there are $(N+1)$ SSU's in each module, then the maximum possible number of rollback steps is N. Let the current state-save interval of module i be $T_i(k)$, then an n-step rollback will restore the module i to the beginning of interval $T_i(k-n+1)$ [i.e., the state at $t_i(k-n+1)$]. For state-save interval $T_i(k-n+1)$, $(n = 1, 2, 3, \cdots, N)$, we assign two matrices $KC_n(M \times M)$ and $KP_n(M \times M)$ to represent the interactions during $T_i(k-n+1)$. Every element in both matrices consists of *a single bit*. $KC_n(i,j)$ is set to 1 if an interaction occurs between module i and module j during the state-save intervals $T_i(k-n+1)$ and $T_j(k-n+1)$. If an interaction exists between the two during module j's previous state-save interval, $T_j(k-n)$, then $KP_n(i,j) = 1$. We also define $RB_i(k)$, $k = 1, 2, \cdots, N$, to indicate the number of rollback steps for module i. If module i rolls back n steps, then $RB_i(k) = 1$ for all $k \leq n$. So, if $RB_i(k) = 0$ for all k, then module i does not have to roll back. The steps for setting these elements and checking the rollback propagation are listed below.

S1) Reset both matrices to zero at the beginning of the task.

S2) When an interaction is issued from module i and directed to module j, then $KC_1(i,j)$ and $KC_1(j,i)$ are set to 1.

S3) If module i saves its state and moves to the next state-save interval, then for $j = 1, 2, \cdots, M$

a) If P_j has already moved to its new state-save interval, then

$$KP_1(j,i) = KP_1(j,i) + KC_1(i,j)$$

where + is logical OR operation.

$$KC_1(j,i) = 0$$

b) $KC_n(i,j) = KC_{n-1}(i,j)$,

$KP_n(ij) = KP_{n-1}(i,j)$

for $n = N, N - 1, \cdots, 2$,

c) $KC_1(i,j) = 0, \qquad KP_1(i,j) = 0$.

S4) When an error is detected in module i, $RB_i(1)$ is set to one and all other RB's are reset to zero.

S5) If $RB_i(n) = 1$ (i.e., module i rolls back at least n steps), the switch controller checks the corresponding rows in matrices KC_n and KP_n, namely $KC_n(i,j)$, $KC_n(j,i)$, and $KP_n(i,j)$ for $j = 1, 2, \cdots, M$. There are four possible rollback propagations.

i) If $KP_n(i,j) = 1$ then module j has to roll back $(n + 1)$ steps. Set $RB_j(k)$ for all $k \leq (n + 1)$ to 1.

ii) If $KP_n(i,j) = 0, KC_n(i,j) = 1$, and $KC_n(j,i) = 1$, then module j also has to roll back n steps. Set $RB_j(k)$ for all $k \leq n$ to 1.

iii) If $KP_n(i,j) = 0, KC_n(i,j) = 1$, and $KC_n(j,i) = 0$, then module j needs to roll back $(n - 1)$ steps. Set $RB_j(k)$ for all $k \leq (n - 1)$ to 1.

iv) If $KP_n(i,j) = 0$ and $KC_n(i,j) = 0$, then there is no direct rollback propagation from module i to module j.

S1), S2), and S3) are used to record interactions. S4) initiates rollback in module i which may propagate to a farther state in the same module and/or to cooperating modules. S5) deals with the determination of rollback propagations. In the condition i) of S5), there is an interaction which occurred in both the P_i's $(k - n + 1)$th and the P_j's $(k - n)$th state saving intervals. Thus, P_j has to roll back $(n + 1)$ steps to recover this interaction. The conditions ii) and iii) indicate that an interaction occurred in the P_j's $(k - n + 1)$th and $(k - n + 2)$th state saving intervals, respectively. The corresponding bits of RB_j are set for these conditions. Since the rollback of P_j decided in S5) can only provide a restorable state for P_i, recursive checking for every j with $RB_j(k) = 1$ is necessary. S5) can also be easily implemented by a recursive procedure which will cease when no more setting of RB's is needed. The final figure of RB's represents the number of necessary rollback steps for each process.

An example is shown in Fig. 4, where Fig. 4(a) describes memory references, Fig. 4(b) is the current contents of KC and KP matrices, and Fig. 4(c) is the result of rollback propagation.

C. The Evaluation of Multistep Rollback

If module i fails at time t_f during the kth state-save interval $T_i(k)$, then we consider a single step rollback of module i to see if it is sufficient to recover from the failure. The result may lead to rollback propagations, and thus to multistep rollbacks as previously discussed. Since the number of state-save units associated with each module is *finite*, the whole task may have to restart when all the states recorded in SSU's are exhausted. In this section a probability model is derived to evaluate the coverage of the multistep rollback recovery which indicates the effectiveness of the present fault-tolerant mechanism. Recall that a module has $(N + 1)$ SSU's and the task is allocated to M modules including PM's and CM's. To derive the coverage, the following assumptions are made and notations used:

A: The access matrix whose element a_{ij} represents the probability of making a reference from module i to module j. For a memory module i, $a_{ij} = 0$, for all j. The sum of all elements in one row must be equal to 1 for a processor module i, i.e., $\sum_{j=1}^{M} a_{ij} = 1$.

b_{ijn}: The probability that $KP_n(i,j) = 0$, which means no interaction occurs during the disparity between module i's and module j's $(k - n + 1)$th state saving instants. For simplicity b_{ijn} is assumed to be a constant for all n, i.e., $b_{ij1} = b_{ij2} = \cdots = b_{ijN} = b_{ij}$. The exact value of b_{ij} is difficult to obtain. Since the state-saving invocations are synchronized, there is at most one instruction occurred during this disparity. An approximate representation is used, i.e., $b_{ij} = \text{Prob}[(B_{ij} \cap B_{jj}) \cup (B_{ii} \cap B_{ji})]$, where B_{ij} is the event that a memory reference from module i to module j occurs at any arbitrary moment.

f_{ijn}: The average probability of having direct rollback propagation from module i to module j due to an n-step rollback of module i. We also assume f_{ijn} to be a constant, f_{ij}, for all n.

r_{ij}: The probability that module j has to roll back because of the direct or indirect propagations if module i fails and then rolls back. Note $r_{ii} = 1$ for all i.

E: The matrix $[e_{ij}]$, $i, j = 1, 2, \cdots, M$, in which element e_{ij} is the average execution time for memory references issued from module i to module j.

T_{ef}: The total execution time of a given task under an error free condition and without the time overhead for generating recovery blocks.

$T_i(k)$: The duration of the kth state-save interval of module i. Because of the asynchrony between state-save invocation and actual state saving, $T_i(k)$ is a random variable. If T_{ss} is long enough such that there is always a state saving following every state-save invocation, the mean of $T_i(k)$ is equal to T_{ss}. To make the analysis simple, this duration is assumed to be constant and equal to the duration of state-save invocation interval, T_{ss}.

T_{sv}: The time overhead for generating a recovery block.

N_t: The total number of state savings before task completion in error-free condition. $N_t = \lfloor T_{ef}/(T_{ss} - T_{sv}) \rfloor$.

u_{ijk}: The average memory reference rate from module i to module j during the kth state-save interval of module i. Occurrence of these memory references is assumed to be a

Poisson process with a time-varying parameter during the progress of task execution. In general, the memory references by processes can be divided into different phases each of which has a constant reference rate [17], [18]. Thus, if N_t is moderately large, u_{ijk} could be assumed to be a constant during the kth state-save interval.

To derive the coverage of a multistep rollback, the probability of direct rollback propagation, i.e., f_{ij}, should be obtained first. From the above definitions and assumptions, f_{ij} is the average probability that there exists at least one memory reference between module i and module j during one state-save interval. It can be expressed as follows:

$$f_{ij} = f_{ji} = g_{ij} + g_{ji} - g_{ij} g_{ji} \tag{1}$$

where $g_{ij} = (1/N_t)\sum_{k=1}^{N_t}(1 - e^{-u_{ijk}T_{ss}})$ represents the average probability of having an interaction from module i to module j during a single state-save interval. Since the total number of memory references between module i and module j is equal to $a_{ij}[T_{ef}/(\sum_{m=1}^{M} a_{im}e_{im})]$ and $\sum_{k=1}^{N_t} u_{ijk}(T_{ss} - T_{sv})$, we have the following relationship:

$$\sum_{k=1}^{N_t} u_{ijk} = (T_{ef}a_{ij})\Big/\left[(T_{ss} - T_{sv})\sum_{m=1}^{M} a_{im}e_{im}\right]. \tag{2}$$

Also, the maximum value of memory reference rate u_{ijk} must be less than or equal to the reciprocal of e_{ij}, that is,

$$\frac{1}{e_{ij}} \geq (u_{ijk})_{\max} \geq u_{ijk} \geq 0. \tag{3}$$

It is easy to observe that f_{ij} is a monotonically increasing function of g_{ij} and g_{ij} is a bounded concave function of u_{ijk}. With the above two constraints we can get the extrema of f_{ij} as follows:

1) The maximum value of f_{ij}, denoted as f'_{ij}, occurs when $u_{ij,1} = u_{ij,2} = \cdots = u_{ij,N_t}$.

2) The minimum value of f_{ij}, denoted as f''_{ij} occurs when there are i) h intervals $\{h = e_{ij}T_{ef}a_{ij}/[(T_{ss} - T_{sv})\sum_{m=1}^{M} a_{im}e_{im}]\}$ in which $u_{ijk} = 1/e_{ij}$, ii) $(N_t - h - 1)$ intervals in which $u_{ijk} = 0$, and iii) one interval in which $u_{ijk} = \{T_{ef}a_{ij}/[(T_{ss} - T_{sv})\sum_{m=1}^{M} a_{im}e_{im}]\} - h/e_{ij}$.

To solve for r_{ij} from f_{ij}, a fully connected network is drawn as Fig. 6 in which every node represents a module, and the link (i,j) connecting node i and node j denotes the relationship for direct rollback propagation between module i and module j. Then f_{ij} can be considered as the probability of having a directly connected link between node i and node j. The theory of network reliability [19] can be used to solve for r_{ij}

$$r_{ij} = \bigcup_{q} (D_{ij,q}) \tag{4}$$

where $D_{ij,q}$ is the probability that the qth path from node i to node j is connected and $\cup$ is the probability union operation. With an additional assumption that the occurrence of failure is equally distributed over the entire modules in a statistical sense, the coverage of a single step rollback, denoted by $C(1)$, becomes

$$C(1) = (1/M)\sum_{i=1}^{M}\prod_{j=1}^{M}\left[1 - r_{ij}\left(1 - \sum_{k=1}^{M} b_{jk}\right)\right] \tag{5}$$

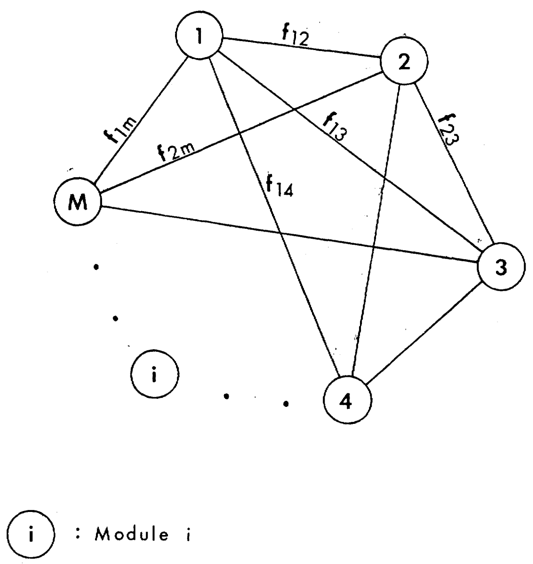


Fig. 6. The rollback propagation network.

and the accumulated coverage from a single step rollback to an h-step rollback can be derived by the following recursive equation:

$$C(h) = C(1)[1 - C(h - 1)] + C(h - 1). \tag{6}$$

The coverage of the multistep rollback recovery is calculated for an example with the following access matrix:

$$\begin{bmatrix} 0.9 & 0.08 & 0.02 & 0. \\ 0.1 & 0.85 & 0.03 & 0.02 \\ 0.03 & 0.03 & 0.9 & 0.04 \\ 0. & 0.02 & 0.08 & 0.9 \end{bmatrix}$$

This example has the access localities 0.85 and 0.9 for processes which correspond to the experimental results obtained from Cm* [20]. The numerical results are presented in Table I and are also plotted in Fig. 7. These results include three cases: the best coverage computed from f''_{ij} for different values of N_t, and the worst coverage computed from f'_{ij}. These results show that only a small number of SSU's is enough to achieve a satisfactory coverage of rollback recovery. It should be particularly noted that the requirement of a small number of SSU's is mandatory for actual implementation. On the other hand, this conclusion must be interpreted in the context of access localities; the number of SSU's required for a given coverage tends to increase with the decrease in access localities (i.e., when there are heavy interactions). This tendency, however, should be understood as an inherent problem associated with multiprocessors rather than with the present fault-tolerant mechanism (see [21] for the dependence of multiprocessor performance on access localities).

IV. The Performance of Rollback Recovery Mechanism

Several methods for analyzing the rollback recovery system have been proposed [22]–[27]. They in general deal with a transaction-oriented database system and compute the

TABLE I
A NUMERICAL EXAMPLE FOR THE COVERAGE OF MULTISTEP ROLLBACKS

i \ C(i)	case 1[a]	case 2[b]	case 3[c]
1	0.75067	0.68610	0.44713
2	0.93783	0.90147	0.69433
3	0.98449	0.96907	0.83100
4	0.99612	0.99029	0.90656
5	0.99902	0.99695	0.94834

[a]case 1: with minimum f_{ij} and $N_t = 100$

[b]case 2: with minimum f_{ij} and $N_t = 10$

[c]case 3: with maximum f_{ij}

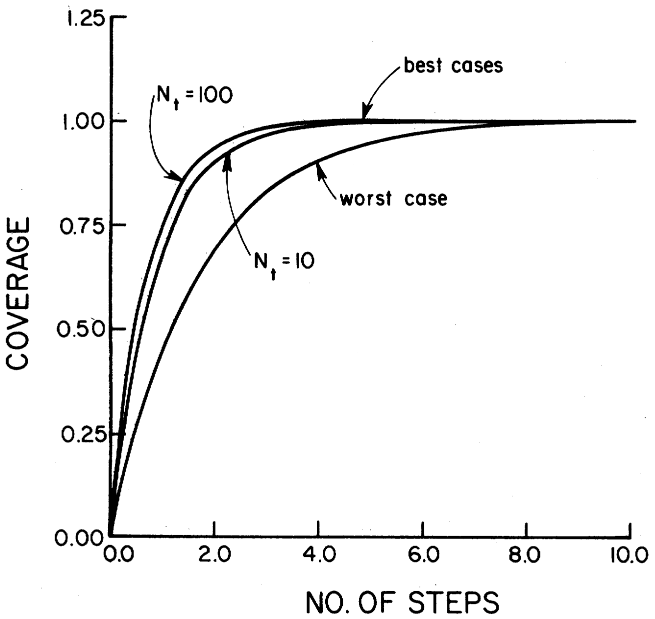


Fig. 7. Rollback coverage versus number of rollback steps.

optimum value of the intercheckpoint interval. Castillo and Siewiorek studied the expected execution time which is required to complete a task with the restart recovery method [28]. All of these approaches either assume the state restoration is obtainable by a single checkpoint or do not include the rollback propagation at all. In this section, we explicitly take into account the problem of multistep rollback and the risk of restart for the rollback recovery mechanism.

A. Notations and Assumptions

The following notations will be used in the sequel:

T_t: The total execution time to complete the given task with occurrence of errors. It includes the required execution time under error-free condition, the time loss due to rollbacks and restarts, and the time overhead for generating recovery blocks.

T_{real}: The total execution time to complete the task when all failures are recovered by rollbacks instead of restarts.

$T^j_{\text{roll},m}$: The time lost due to the jth rollback in module m which consists of the setup time for resumption, t_{sb}, and the computation undone by rollback.

T^i_{rst}: The time lost due to the ith restart which includes the setup time for restart, t_{su}, and the time between the previous start and the moment at which error is detected.

TE_k: The accumulated effective computation before the kth rollback when the task can be completed without restart.

$X^j_r(X^i_s)$: The duration between two consecutive rollbacks (restarts).

$C(i)$: The accumulated coverage of rollback recovery from a single step to i steps. This value is calculated by (5) and (6) presented in the previous section.

$P_b(P_s)$: The probability of rollback (restart) when a failure occurs.

$P_{st}(h)$: The probability of having an h-step rollback given that the failure is recovered by the rollback.

$P_r(m)$: The probability of having m rollbacks during the time interval, T_{real}.

$Z_r(z), Z_{st}(z)$: The probability generating functions of $P_r(m), P_{st}(h)$, respectively.

$\Phi_t(s), \Phi_{\text{real}}(s)$: The characteristic functions of T_t, T_{real}, respectively.

The goal of our analysis is to calculate the mean and variance of the total execution time of a given task, T_t. Recall that the task is decomposed and then allocated to M modules. During the normal operation, the small overhead is required to generate consecutive recovery blocks in each module. When the jth error occurs, module m spends $T^j_{\text{roll},m}$ to recover from this error if the error is recoverable by a rollback. Otherwise, the whole task has to restart. $T^j_{\text{roll},m}$ consists of the setup time which is composed of the decision delay required for examining rollback propagation, the reconfiguration time, and the time used to make up for the computation undone by the rollback. We assume that the task completion be delayed by $\max\{T^j_{\text{roll},m}\}$ where $m = 1, 2, \cdots M$ for the rollback recovery of the jth error. The resultant completion time will be the upper bound because of the following reasons. First, $T^j_{\text{roll},m}$ can be interpreted as the time lost due to the rollback in module m. So, the total time lost in all the concerned modules is $\sum_{m=1}^{M} T^j_{\text{roll},m}$. Since the completion of a task is regarded as the completions of all its processes, the time lost from the task's point of view could be $\max\{T^j_{\text{roll},m}\}$ but not larger than this maximal value. Secondly, the true delay effect on the completion of task by a rollback will be shortened because of the possible reduction in the waiting time of process synchronization. To facilitate system reconfiguration, we also assume the multiprocessor has a sufficient number of standby modules so that the task may be executed continuously from start to end without waiting for the availability of modules. The time needed for error-free execution is regarded as constant and is independent of reconfiguration.

In general, the occurrence of error can be modeled as a Poisson process with parameter $\lambda(t)$ which equals the reciprocal of mean time between failures [29]. Since $\lambda(t)$ is slowly time-varying (for example with a period of one day), it is

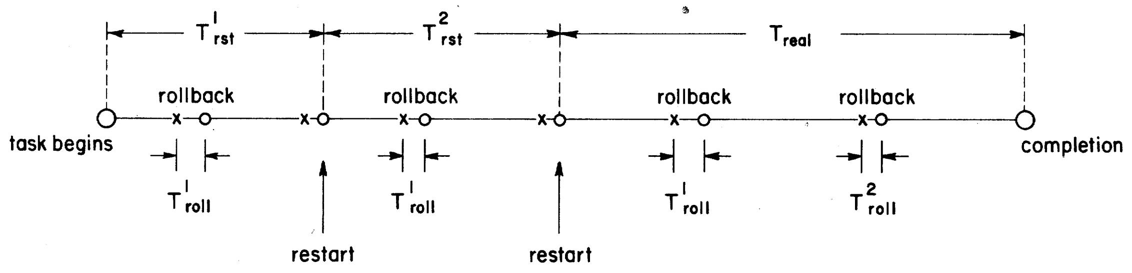


Fig. 8. Task execution phases.

assumed to be constant over the duration of one task execution, i.e., $\lambda(t) = \lambda$. For simplicity an error is assumed to be detected immediately as it occurs (see Section III for a brief description on relaxing this assumption). From the definitions of P_s, P_b, and $P_{st}(h)$, we have $P_s = 1 - C(N)$ when each module has $(N + 1)$ SSU's. Therefore, the probability of rollback, P_b, becomes $C(N)$. $P_{st}(h)$ is equal to $(1/P_b)[C(h) - C(h - 1)]$ for $h = 2, \cdots, N$, and $P_{st}(1) = C(1)/P_b$. After the detection of error, the occurrence of rollback and restart can be regarded as a Bernoulli process, with probability P_b and P_s respectively, and independent of the error generation process. Thus they can be modelled as Poisson processes with parameters $\lambda_b = \lambda P_b$ and $\lambda_s = \lambda P_s$, respectively.

B. The Performance Model

The total task execution time, T_t, can be divided into several phases as shown in Fig. 8. The last phase is always ended with the completion of task. Other phases are followed by a restart. This implies that the amount of effective computation at the beginning of each phase is zero. During each phase, the effective computation between rollbacks is accumulated toward the task completion. To derive the distribution of T_t, we should determine the distribution of the duration of the last phase (which is defined as T_{real}), the probability of having R restarts prior to the last phase, and the distribution of the durations of other phases which are defined as T^i_{rst} for $i = 1, 2, \cdots R$.

In the last phase, the task will be executed from the beginning to the completion without any restart. It is assumed that T_{ef} is much larger than T_{ss} ($T_{ef} >> T_{ss}$) so that the rollback distance of an h-step rollback can be approximated by hT_{ss}. The effective computation between two consecutive rollbacks becomes $(X_r - hT_{ss})^+$ when a module rolls back h steps where $(X)^+ = \max\{0, X\}$ is a positive rectification function. With the probability of an h-step rollback, $P_{st}(h)$, two functions are introduced

$$Z = \sum_{h=1}^{N} e^{-h\lambda_b T_{ss}} P_{st}(h) \tag{7}$$

$$H(t, k) = \sum_{i=0}^{k} \binom{k}{i} (1 - Z)^i (Z)^{k-i} G_{k-i}(t) \tag{8}$$

where $G_{k-i}(t)$ is the $(k - i)$th order gamma distribution function with parameter λ_b for $(k - i) > 0$, and $G_0 = 1$. In Appendix A, we show that the distribution function of the accumulated effective computation after k rollbacks is $\text{Prob}(TE_k \le t) = H(t, k)$. Therefore, the probability of k rollbacks during the time interval T_{real}, $P_r(k)$, is given by

$$\begin{aligned} P_r(k) &= P(TE_{k+1} > T_{ef}) - P(TE_k > T_{ef}) \\ &= H(T_{ef}, k) - H(T_{ef}, k + 1). \end{aligned} \tag{9}$$

T_{real} is composed of T_{ef} and the time lost due to rollbacks which is a sum of identically distributed random variables, $T^j_{\text{roll},m}$, for $j = 1, 2, \cdots k$. Substituting the probability mass functions of $P_r(k)$ and $P_{st}(h)$, we get the characteristic function of T_{real} which is given below:

$$\Phi_{\text{real}}(s) = e^{-sT_{ef}} Z_r[e^{-st_{sb}} Z_{st}(e^{-sT_{ss}})]. \tag{10}$$

From Fig. 8, the total time T_t can be represented as the sum of T_{real} and the random sum of T^i_{rst}. The characteristic function of T_t derived in Appendix B is given in the following:

$$\Phi_t(s) = \sum_{n=0}^{\infty} e^{-nst_{su}} \left(\frac{\lambda_s}{\lambda_s + s}\right)^n \cdot \left\{\sum_{j=0}^{n} \binom{n}{j} (-1)^j \Phi_{\text{real}}[(j + 1)(\lambda_s + s)]\right\}. \tag{11}$$

This equation presents a general expression of the total execution time. For the system without the rollback recovery mechanism, we can use $P_s = 1, P_b = 0$, and then $\Phi_{\text{real}}(s)$ becomes $e^{-sT_{ef}}$. The result obtained from the above equation is the same as that in [28]. The mean and variance of the total execution time can be obtained from $-\partial\Phi_t(s)/\partial s|_{s=0}$ and $\partial^2\Phi_t(s)/\partial s^2|_{s=0}$. In Fig. 9, the mean execution time for the example in Section III is plotted. It is obvious that the overhead of generating recovery blocks has an important effect on the rollback recovery method. Since the state savings are performed in parallel with the normal process execution, the overhead contains only the time required for the validation test. When the embedded checking circuits are not very much cost-effective and complex [30], the overhead of generating recovery blocks can be reduced with a completely self-checking mechanism. Fig. 10 expresses the variance of execution time for the previous example. It suggests that the prediction of the total execution time becomes more accurate when the rollback recovery mechanism is used. This result is expected intuitively since the probability of restart is reduced considerably. In a system with a higher probability of restart, the system contains a larger and more uncertain recovery overhead (i.e., larger mean and variance).

Another interesting parameter is the duration of state-save invocation, T_{ss}. The interval has two mutually conflicting effects. Fig. 7 indicates that the increasing of T_{ss} will induce more rollback propagations and degrade the coverage (a

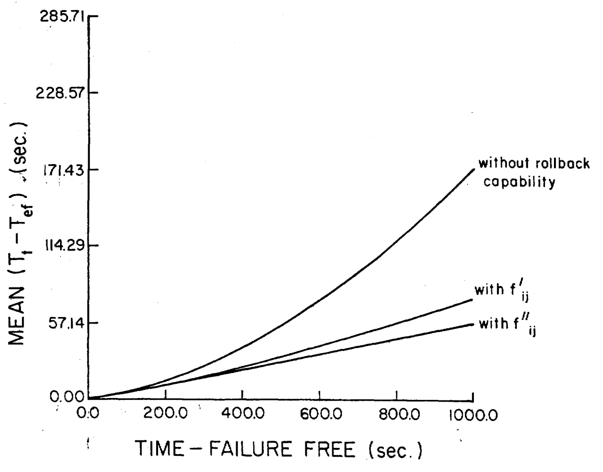


Fig. 9. Mean of time-overhead versus error-free execution time.

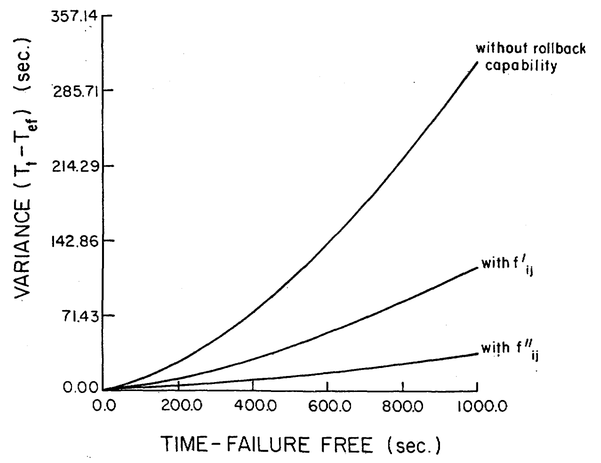


Fig. 10. Variance of time-overhead versus error-free execution time.

larger value of N_t means a shorter state-save interval). Since the occurrence of error is distributed throughout the state-save interval, the average computation loss due to rollbacks is proportional to the state-save duration. Therefore the increase of T_{ss}, which invokes longer state-save intervals, will introduce more computation loss and higher probability of restart. On the other hand, the percentage of the total time overhead for generating recovery blocks is reduced by the increase of T_{ss}. The optimum value which minimizes the expected execution time can be found in Fig. 11. Fig. 11 shows that there exists a linear relationship between T_t and T_{ss} when N_t is larger (i.e., T_{ss} gets smaller), where the overhead of generating recovery blocks dominates the final result. When T_{ss} is greater than the optimum value, the loss due to recovery increases considerably because of the larger time loss in each rollback.

V. Conclusion

We considered the design of a hardware recovery mechanism for a fault-tolerant multiprocessor with emphasis placed on a fast state-save operation which requires little time overhead. To permit processes to be general and to ensure programmer transparency, recovery points are established

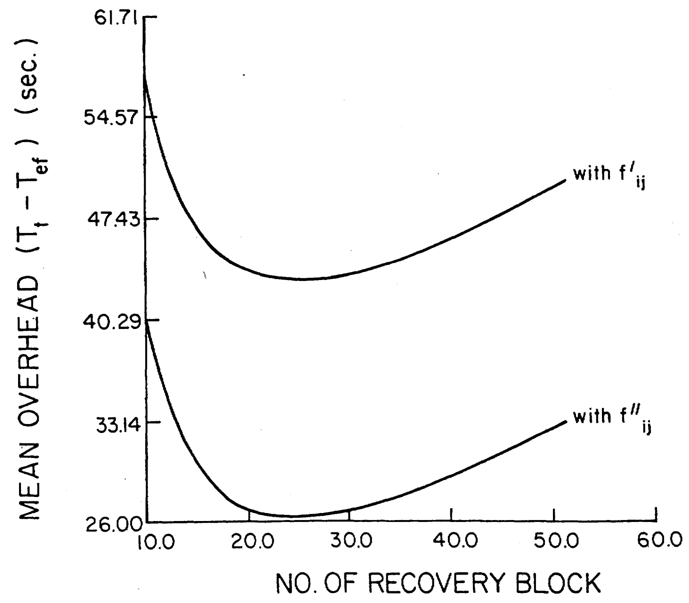


Fig. 11. Mean time-overhead versus total number of recovery blocks for a given task.

automatically and regularly. We also derived analytically the probability of multistep rollback, the coverage of rollback recovery, and the risk of restart which are usually ignored in most existing analyses. The results in this work indicate that the performance of the rollback recovery mechanism is significantly dependent upon the risk of restart which can be minimized by a higher local hit ratio. So, the improvements are related to the partitioning, cooperation, and allocation of processes. This is a common, inherent issue in the design of multiprocessors rather than in that of the present fault-tolerant system [21].

Since the rollback mechanism used here only provides a recovery capability to tolerate the hardware faults in processor modules and common memory modules, further improvements are conceivable to achieve the overall system reliability. In addition to memory assignments many program operations may involve file access and input–output interfaces which also affect the system behavior. These operations can not be simply recovered by a standard rollback procedure. Thus, other special recovery actions, such as execution of *recoverable procedures* [10], should be included (e.g., exception handling for input–output operations). That is, additional recoverable procedures provided by the program designer are needed to take special related recovery actions. With the same concept our hardware recovery scheme can be extended to provide such special recovery actions by, for example, associating separate save units and/or procedures with each of I/O interfaces, file accesses, etc. In addition, the reliability of the interconnection network can be obtained by using redundant hardware to form additional paths (e.g., additional stages in generalized cube network [31]) or by using reliable switches (e.g., 2 × 2 fault-tolerant switching element proposed in [32]). However, the faults which occurred in the supplementary resources, like SSU's and monitor switches, do not cause damages to the computation itself but will change the recovery capability. Although the performability [33] of the system at a single state is not affected by SSU's, etc., the overall lifetime performability is changed because of the degradation of recovery capability. A higher recovery capability can be gained by using hardware redundancy. For instance, an additional standby monitor switch can either test the active monitor switch or replace the active one whenever it malfunctions.

To deal with the performance of a fault recoverable and reconfigurable multiprocessor, the delay in the task completion time due to the errors is an important parameter. In such a system one or more faults which cause the errors in the computation and the loss of a portion of function capability may have no serious consequence to the completion of a given task, but the quality of the recovery procedure largely determines the distribution of the task completion time. Thus, the overhead required to treat the contamination of error, and the effect on the task execution time, should be included to represent the effectiveness of fault-tolerance. In addition, for most real-time applications, such as aircraft or industrial control, etc., one major concern is whether the required task can be completed prior to a given deadline or not. The rollback mechanism proposed in this paper not only offers system modularity and simplicity, but also provides fast recovery and accurate prediction of the task completion time. Hence, the present fault-tolerant multiprocessor has a high potential use for critical real-time applications.

Appendix A
Calculation of the Probability of Having k Rollbacks During the Duration T_{REAL}

From the definition of $P_{st}(h)$, the task will roll back h steps with probability $P_{st}(h)$ following a failure detection within the last phase of duration T_{real}. Let the rollback distance for the jth rollback recovery be T^j_{roll} which is approximately equal to hT_{ss} with probability $P_{st}(h)$. Thus the accumulated effective computation time before the kth rollback TE_k, is given by

$$TE_k = \sum_{j=1}^{k} (X^j_r - T^j_{\text{roll}}). \tag{A.1}$$

Since the occurrence of rollback is a Poisson process with parameter λ_b, the density function of X^j_r is $\lambda_b e^{-\lambda_b t}$. The probability of having $(X^j_r - T^j_{\text{roll}}) = 0$ is $\sum_{h=1}^{N} P_{st}(h)(1 - e^{\lambda_b h T_{ss}})$. The density function of $(X^j_r - T^j_{\text{roll}})$ becomes

$$f_\alpha(t) = \sum_{h=1}^{N} P_{st}(h)(1 - e^{-\lambda_b h T_{ss}})\,\delta(t) + e^{-\lambda_b t} \sum_{h=1}^{N} P_{st}(h) e^{-\lambda_b h T_{ss}} \tag{A.2}$$

where $\delta(t)$ is an impulse function. Let $Z = \sum_{h=1}^{N} P_{st}(h) e^{-\lambda_b h T_{ss}}$. Then f_α is simplified by

$$f_\alpha(t) = (1 - Z)\,\delta(t) + e^{-\lambda_b t} Z. \tag{A.3}$$

The characteristic function of TE_k, which is equal to $[\Phi_\alpha(s)]^k$ where $\Phi_\alpha(s)$ is the characteristic function of $(X^j_r - T^j_{\text{roll}})$, becomes

$$\Phi_{te,k}(s) = \sum_{i=0}^{k} \binom{k}{i} (1 - Z)^i (Z)^{k-i} \left(\frac{\lambda_b}{s + \lambda_b}\right)^{k-i}. \tag{A.4}$$

Taking the inverse Laplace transform, the density function of TE_k [denoted as $f_{te,k}(t)$] is obtained. Then the distribution function of TE_k becomes

$$P(TE_k \le t) = \int_0^t f_{te,k}(\tau)\,d\tau = \sum_{i=0}^{k-1} \binom{k}{i} (1 - Z)^i (Z)^{k-i} G_{k-i}(t) + (1 - Z)^k \tag{A.5}$$

where $G_{k-i}(t)$ is the $(k - i)$th order gamma distribution function.

Appendix B
Calculation of the Characteristic Function of Total Execution Time $\Phi_T(s)$

From Fig. 8, the total execution time T_t is the sum of T_{real} and T_{rst}, where $T_{rst} = \sum_{i=1}^{n} T^i_{rst}$ when there are n restarts. Given the conditional probability of T_t, we can write the following equation:

$$E(T_t \mid T_{\text{real}}) = T_{\text{real}} + E(T_{rst} \mid T_{\text{real}}). \tag{B.1}$$

It is assumed that the time interval between the $(i - 1)$th and the ith restarts, X^i_s, is exponentially distributed with mean $1/\lambda_s$. Thus, for a given T_{real}, the time lost due to the ith restart, T^i_{rst}, is randomly distributed between t_{su} to $T_{\text{real}} + t_{su}$ with density function, $f^i_{rst|T_{\text{real}}}(t)$, given by

$$f^i_{rst|T_{\text{real}}}(t + t_{su}) = \frac{\lambda_s e^{-\lambda_s t}}{1 - e^{-\lambda_s T_{\text{real}}}} \quad \text{for } 0 \le t \le T_{\text{real}}. \tag{B.2}$$

The probability of having n restarts for a given T_{real} is

$$P_{rs|T_{\text{real}}}(n) = (e^{-\lambda_s T_{\text{real}}})(1 - e^{-\lambda_s T_{\text{real}}})^n. \tag{B.3}$$

Since $T_t = T_{\text{real}} + \sum_{i=1}^{n} T^i_{rst}$ if there are n restarts before the task completion, the characteristic function of T_t for a given T_{real} becomes

$$\Phi_{t|T_{\text{real}}}(s) = e^{-sT_{\text{real}}} \sum_{n=0}^{\infty} P_{rs|T_{\text{real}}}(n)\,[\Phi_{rst|T_{\text{real}}}(s)]^n \tag{B.4}$$

where $\Phi_{rst|T_{\text{real}}}(s)$ is the characteristic function of the time loss due to a restart for a given T_{real}, i.e., the Laplace transformation of $f^i_{rst|T_{\text{real}}}(t)$. By substituting $P_{rs|T_{\text{real}}}(n)$ and $\Phi_{rst|T_{\text{real}}}(s)$ into (B.4) and integrating with the density function of T_{real}, the characteristic function of T_t is obtained as (11) in Section IV.

References

[1] J. A. Rohr, "Starex self-repair routines: Software recovery in the JPL-STAR computer," in *Proc. 3rd Int. Symp. Fault-Tolerant Comput.*, 1973, pp. 11–16.

[2] F. E. Heart, S. M. Ornstein, W. R. Crowther, and W. B. Barker, "A new minicomputer/multiprocessor for the ARPA network," in *Proc. 1973 AFIPS Nat. Comput. Conf.*, 1973, vol. 42, pp. 529–537.

[3] B. Randell, P. A. Lee, and P. C. Treleaven, "Reliability issues in computing system design," *Comput. Surveys*, pp. 123–165, June 1978.

[4] B. Randell, "System structure for software fault tolerance," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 220–232, June 1975.

[5] D. L. Russell, "Process backup in producer-consumer systems," in *Proc. 6th ACM Symp. Operating Syst. Principles*, pp. 151–157, Nov. 1977.

[6] K. H. Kim, "An approach to programmer-transparent coordination of recovering parallel processes and its efficient implementation rules," in *Proc. 1978 Int. Conf. Parallel Processing,* Aug. 1978, pp. 58–68.

[7] ——, "An implementation of a programmer-transparent scheme for coordinating concurrent processes in recovery," in *Proc. COMPSAC 80,* pp. 615–621, Oct. 1980.

[8] K. Kant and A. Silberschatz, "Error recovery in concurrent processes," in *Proc. COMPSAC 80,* pp. 608–614, Oct. 1980.

[9] W. G. Wood, "A decentralized recovery control protocol," in *Proc. 11th Int. Symp. Fault-Tolerant Comput.,* 1981, pp. 159–164.

[10] J. J. Horning, H. C. Lauer, P. M. Melliar-Smith, and B. Randell, "A program structure for error detection and recovery," in *Lecture Notes in Computer Science: Operating Systems.* New York: Springer-Verlag, 1974, pp. 171–187.

[11] A. M. Feridun and K. G. Shin, "A fault-tolerant multiprocessor system with rollback recovery capabilities," in *Proc. 2nd Int. Conf. Distributed Comput. Syst.,* pp. 283–298, Apr. 1981.

[12] Y. H. Lee, and K. G. Shin, "Rollback propagation detection and performance evaluation of FTMR^2M — A Fault-Tolerant Multiprocessor," in *9th Annu. Symp. Comput. Arch.,* pp. 171–180, Apr. 1982.

[13] R. J. Swan, S. H. Fuller, and D. P. Siewiorek, "Cm*: a Modular Multimicroprocessor," in *Proc. 1977 AFIPS Nat. Comput. Conf.,* vol. 46, 1977, pp. 637–644.

[14] P. H. Enslow, "Multiprocessor organization — A survey," *Comput. Surveys,* vol. 9, pp. 101–129, Mar. 1977.

[15] K. H. Kim, "Error detection, reconfiguration and recovery in distributed processing systems," in *Proc. 1st Int. Conf. Distribut. Comput. Syst.,* pp. 284–295, Oct. 1979.

[16] J. J. Shedletsky, "A rollback interval for networks with an imperfect self-checking property," *IEEE Trans. Comput.,* vol. C-27, pp. 500–508, June 1978.

[17] A. W. Madison and A. P. Batson, "Characteristics of Program Localities," *Commun. Ass. Comput. Mach.,* vol. 19, pp. 285–294, May 1976.

[18] A. P. Batson, "Program behavior at the symbolic level," *Computer,* pp. 21–26, Nov. 1976.

[19] S. Rai and K. K. Aggarwal, "An efficient method for reliability evaluation of a general network," *IEEE Trans. Reliability,* vol. R-27, Aug. 1978.

[20] S. H. Fuller *et al.,* "Multimicroprocessors: An overview and working example," *Proc. IEEE,* vol. 66, pp. 216–228, Feb. 1978.

[21] K. G. Shin, Y.-H. Lee, and J. Sasidhar, "Design of HM^2p — A hierarchical multimicroprocessor for general-purpose applications," *IEEE Trans. Comput.,* vol. C-31, pp. 1045–1053, Nov. 1982.

[22] K. M. Chandy, J. C. Browne, C. W. Dissly, and W. R. Uhrig, "Analytic models for rollback and recovery strategies in data base systems," *IEEE Trans. Software Eng.,* vol. SE-1, pp. 100–110, Mar. 1975.

[23] K. M. Chandy and C. V. Ramamoorthy, "Rollback and recovery strategies for computer programs," *IEEE Trans. Comput.,* vol. C-21, pp. 546–556, June 1972.

[24] E. Gelenbe and D. Derochette, "Performance of rollback recovery systems under intermittent failures," *Commun. Ass. Comput. Mach.,* vol. 21, pp. 493–499, June 1978.

[25] J. W. Young, "A first order approximation to the optimum checkpoint interval," *Commun. Ass. Comput. Mach.,* vol. 17, pp. 530–531, Sept. 1974.

[26] G. L. Brodetskiy, "Periodic dumping of intermediate results in systems with storage-destructive failures," *Cybernetics,* vol. 15, pp. 685–689, Sept.–Oct. 1979.

[27] ——, "Effectiveness of storage of intermediate results in systems with failures that destroy information," *Eng. Cybernetic,* vol. 16, pp. 75–81, Nov.–Dec. 1978.

[28] X. Castillo and D. P. Siewiorek, "A performance-reliability model for computing systems," in *Proc. 10th Int. Symp. Fault-Tolerant Comput.,* 1980, pp. 187–192.

[29] ——, "Workload, performance, and reliability of digital computing systems," in *Proc. 11th Int. Symp. Fault-Tolerant Comput.,* 1981, pp. 84–89.

[30] W. C. Carter, *et al.,* "Cost effectiveness of a self checking computer design," in *Proc. 7th Int. Symp. Fault-Tolerant Comput.,* 1977, pp. 117–123.

[31] G. B. Adams and H. J. Siegel, "A fault-tolerant interconnection network for supersystems," *IEEE Trans. Comput.,* vol. C-31, pp. 443–454, May 1982.

[32] W. Lin and C. L. Wu, "Design of a 2 × 2 fault-tolerant switching element," in *Proc. 9th Annu. Symp. Comput. Arch.,* Apr. 1982, pp. 181–189.

[33] J. F. Meyer, "On evaluating the performability of degradable computing systems," *IEEE Trans. Comput.,* vol. C-29, pp. 720–731, Aug. 1980.

Yann-Hang Lee (S'81) received the B.S. degree in engineering science and the M.S. degree in electrical engineering from National Cheng Kung University, Taiwan, China, in 1973 and 1978, respectively.

Currently, he is working towards the Ph.D. degree in computer, information, and control engineering at the University of Michigan, Ann Arbor, MI. His research interests include multiprocessor and distributed systems, performance evaluation, and fault-tolerant computing.

Kang G. Shin (S'75–M'78–SM'83) was born in Choongbuk Province, Korea, on October 20, 1946. He received the B.S. degree in electronics engineering from Seoul National University, Seoul, Korea in 1970, and both the M.S. and Ph.D. degrees in electrical engineering from Cornell University, Ithaca, NY in 1976 and 1978, respectively.

From 1970 to 1972 he served in the Korean Army as an ROTC officer and from 1972 to 1974 he was on the research staff of the Korea Institute of Science and Technology, Seoul, Korea, working on the design of VHF/UHF communication systems. From 1974 to 1978 he was a Teaching/Research Assistant and then an Instructor in the School of Electrical Engineering, Cornell University. From 1978 to 1982 he was an Assistant Professor at Rensselaer Polytechnic Institute, Troy, NY. He was also a Visiting Scientist at the U.S. Airforce Flight Dynamics Laboratory in Summer 1979, and at Bell Laboratories, Holmdel, NJ in Summer 1980 where his work was concerned with distributed airborne computing and cache memory architecture, respectively. He also taught short courses for the IBM Computer Science Series in the area of computer architecture. Since September 1982, he has been with the Department of Electrical and Computer Engineering at The University of Michigan, Ann Arbor, MI, where he is currently an Assistant Professor. His current teaching and research interests are in the areas of distributed and fault-tolerant computing, computer architecture, and robotics and automation.

Dr. Shin is a member of the Association for Computing Machinery, Sigma Xi, and Phi Kappa Phi.