(NASA-CR-169560) DESIGN AND EVALUATION OF A FAULT-TOLERANT MULTIPROCESSOR USING HARDWARE RECOVERY BLOCKS (Michigan Univ.) 54 p HC A04/MF A01 CSCL 09B

N83-13819

Unclas
G3/60 02114

THE UNIVERSITY OF MICHIGAN

COMPUTING RESEARCH LABORATORY

THE UNIVERSITY OF MICHIGAN

COMPUTING RESEARCH LABORATORY[1]

DESIGN AND EVALUATION OF A FAULT-TOLERANT MULTIPROCESSOR USING HARDWARE RECOVERY BLOCKS

Yann-Hang Lee and Kang G. Shin

CRL-TR-6-82

AUGUST 1982

Room 1079, East Engineering Building
Ann Arbor, Michigan 48109
USA
Tel: (313) 763-8000

[1]This work was supported in part by NASA grant No. NAG 1-296. All correspondence should be sent to Professor Kang G. Shin. 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 agencies.

DESIGN AND EVALUATION OF A FAULT-TOLERANT MULTIPROCESSOR USING HARDWARE RECOVERY BLOCKS

Yann-Hang Lee and Kang G. Shin

ABSTRACT

In this paper we consider the design and the 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 block is constructed by consecutive state-save operations and several state-save units in every processor and memory module. When a fault is detected, the multiprocessor reconfigures itself to replace the faulty component and then the process originally assigned to the faulty component 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 multiple-step rollbacks may thus become necessary. 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 multi-step rollback recovery and the risk of restart. The performance evaluation in terms of the mean and variance of execution time of a given task is also presented.

Index Terms - Fault-tolerant multiprocessor, rollback recovery, hardware/software recovery block, rollback propagation, coverage of recovery.

1. INTRODUCTION

There are numerous benefits to be gained from a multiprocessor. In addition to the decreasing of hardware cost and the inherent reliability of LSI components, the capacity of reconfiguration makes the multiprocessor 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 error recovery without having to restart the whole task when an error occurs.

In general, the tolerance of failure during system operation is realized 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 failure of the system. There are two design approaches in error detection: (1) detect an error immediately, 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, the execution of validation routines, or acceptance test are typical methods for the second approach. After the detection of an error, the faulty components, which are the source of error, are localized and replaced so as to enable the system to be operational again. To recover from an error, the rollback recovery method or the re-initialization 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 the error 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 an error is detected is lost, 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 (these may lead to the restarting of the whole task and result in high re-initialization overhead). The rollback recovery method at the software level is also difficult to implement and may not be effective, especially for tightly coupled processes, since (1) the software recovery points 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. (This could become a heavy burden on the program designers). Several alternatives have been proposed: for example, the conversation scheme [4], the interprocess communication primitives in producer-consumer system [5], the programmer-transparent scheme [6,7], the system defined checkpoints [8], etc. These methods could lead to a loss of efficiency in the absence of error, the accumulation of a 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], 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 <u>processor or memory failures</u>. In general a process state can be regarded as the status of internal registers of the assigned processor and the process variables stored in memory. In order to resume a failed process, an error-free process state should be restored. The hardware recovery block is constructed in a quasi-synchronized manner which 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 non-local states when a check-point is encountered. Moveover, instead of the assertions in the check-point of the software recovery block, the hardware resources are tested by embedded checking circuits and self-test 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 by transferring the recorded states into the replacement module, the original process can be resumed.

The multiprocessor with a hardware recovery block scheme takes advantage of the large number of processor units available to facilitate fast recovery from hardware failures. Furthermore, the system minimizes the time required to establish every recovery block that would significantly affect system performance.

For both hardware and software recovery blocks, the rollback of the failed process 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 such limitations, several consecutive states are saved so that the processes are allowed to roll back multiple steps in case of rollback propagation. The coverage of a multi-step rollback, which indicates the probability of having a successful rollback recovery when the 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 2. The detailed description can be found in [9,10]. In this section, we also extend the previous design to a general multiprocessor on which our hardware fault recovery can be implemented. Section 3 presents an algorithm to detect rollback propagations among cooperating processes and also proposes a model to evaluate the coverage of multi-step rollback recovery. Section 4 uses the results of Section 3 and deals with the analysis and estimation of performance in terms of the mean and variance of the task completion time. The conclusion follows in Section 5.

2. 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. First, the basic state-save mechanism associated with every processor module and common memory is briefly presented. Then we discuss the rollback recovery operations of a task for which the following two multiprocessors can be used: in one, there is no common memory, but local memory of one processor module is accessible by other processor modules (e.g., Cm* system [11]); in the other, the system is equipped with separated common memory modules [12] and restricts the access of local memory only to the resident processor.

2.1 Processor Module, Common Memory, and State-save

A basic processor module (PM) in the multiprocessor comprises a processor, a local memory, a local switch, state-save memory units (SSUs) 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 separated common memory or local memories depending upon the multiprocessor structure. Thus each process in a PM can communicate with other processes (allocated to other PMs) through the shared variables. PMs save their states (i.e. process local variable and processor status) in an SSU 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 PMs, 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 PMs a state-save invocation signal at a regular interval, Tss. 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 a 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 SSUs are used to record the updates of CM only. The access requests of CM are managed by an access queue on the basis of 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 served.

During a state-save interval, besides the normal memory reference or instruction execution, certain operations are automatically executed; for example, a parity check is done whenever a bus/memory is used. Some redundant error detection units also accompany the processor module [13], e.g., dual-redundancy comparison, address-in-bound check, etc. These units are expected to detect a malfunction whenever the corresponding function units are used. An additional validation process which could be the execution of self-test routine refreshes the shelters 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 SSU_1, SSU_2, ... 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 SSUs and record the currently changing state in the remaining SSU. As shown in Fig. 3, the SSU_1, SSU_2, .. SSU_N are so arranged to record the states for consecutive state-save intervals T(i),T(i+1), ... 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 SSUs are exhausted. The monitor switch is used to route the updates to SSUs and to manage the aging of SSUs. Therefore the state-save mechanism of each PM or CM provides an N-step rollback capability. However, in Section 3, we will show that only a small number of SSUs are sufficient to establish high coverage of rollback recovery for a given task.

Since the update of dynamic elements is recorded in only one SSU, the other SSUs 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 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 [9]. 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.

2.2 Rollback Recovery Operations of a Task

As described in the above section, each processor module and common memory has its own rollback mechanism with several saved states. With these individual rollback recovery capabilities, the rollback recovery of a task is described as follows.

Suppose a task is partitioned and then allocated to M modules i (i=1,2,...,M). These modules include PMs and CMs 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. Moveover, 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 multi-step rollback (a detail of this will be discussed in the next section). In order to make 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 multi-step rollback. The system monitor receives the task execution command and then allocates PMs and CMs to perform the task. Both devices are defined in a logical sense. They could be a host computer, 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 about 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 PMs that are executing processes of the task in question.
2. Make a decision as to rollback propagation.
3. Resume the execution of 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 addresses directed to the faulty module into its replacement.

6. Restore the previous states of the processes affected by the rollback of the 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 PMs, which are still queued in the access queues, are cancelled.

3. ROLLBACK PROPAGATION AND MULTI-STEP 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 SSUs 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 arbitrary processes 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. Therefore, a multi-step rollback is required.

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 when 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) [14]:

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

(2). If D ≥ the total computation done then restart
else if D ≥ U then done
else go to step (1).

3.1 Rollback Propagation and Multi-step Rollback

In general rollback propagation can not be avoided if the processes interact with each other arbitrarily. For the organization of multiprocessor in the previous section, a process will be located 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. Each interaction between cooperating processes is implemented as a memory reference to a shared variable. It is also regarded as a memory reference across the modules. To avoid having to trace 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 it during the current state-save interval. Let P_i-->P_j denote the rollback propagation in which the rollback of process

P_i induce the state restoration in more than one modules and than induce the rollback of process P_j. An example is presented in Figure 4, where process P_1 fails at time t_f. Since the interactions between P_1 and P_2 exist during the time interval (t^1_n, t_f), process P_2 must roll back to enable the interaction for the resumption of P_1. The rollback of P_2 will propagate further to other processes; in this example, P_2-->P_4, P_1-->P_3, and P_3-->P_2 .

In the above example, we can find that the rollback of P_3 and P_2 to their most recently saved state still cannot provide a consistent state. (This requires a multi-step rollback). The reason that a single step rollback can not recover the process states is mainly due to the occurrence of references between the asynchronous state savings of interacting processes. Consider the cases in Figure 5. Suppose P_i rolls back because of failure or rollback propagation from another process. In case (a), the single step rollback of P_i is sufficient to recover its state if there is no other rollback propagated to P_j. In cases (b), (c), and (d), both P_i and P_j have to roll back. Since there exists an interaction between the state-savings of P_i and P_j, rollback to further state is necessary. A property related to the necessary condition for a successful rollback can be stated as follow:

Property: When process P_i rolls back to the beginning of state-save interval $T_i(m)$, (process P_i may rolls back n steps to reach this point, $n \leq N$) if there is no interaction with P_j across different state-save intervals $T_i(m)$ and $T_j(m-1)$ for all j, where $j=1,2,\ldots,M$ and $j \neq i$, then the state of P_i can be restored by this rollback.

This property implies that the rollback of a task TK where $TK=\{P_i|i=1,2,..M\}$ will be recovered from a failure if P_i for any i is not affected by the rollbacks of P_j for all $j \neq i$ and if P_i rolls back $n_i \leq N$ steps at which P_i's state is restored.

3.2 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 multi-step rollbacks. Suppose there are (N+1) SSUs at each module, then the maximum possible rollback step 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 $T_i(k-n+1)$. For state-save interval n ($n=1,2,3,\ldots,N$), we assign two matrices $KC_n(M*M)$ and $KP_n(M*M)$ to represent the interaction during the state-save interval $T_i(k-n+1)$. 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 $KP_n(i,j)=1$. The steps for setting these elements and checking the rollback propagation are listed as follows:

1. Reset both matrices to zero at the beginning of the task.
2. 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.
3. If module i saves its state and moves to the next state-save interval, then for $j=1,2,\ldots,M$

   (a). $KP_1(j,i)=KP_1(j,i)+KC1(i,j)$ (where + is logical OR operation)
   $KC_1(j,i)=0$

   (b). $KC_n(i,j)=KC_{n-1}(i,j)$,
   $KP_n(i,j)=KP_{n-1}(i,j)$ for $n=N,N-1,\ldots,2$

   (c). $KC_1(i,j)=0$, $KP_1(i,j)=0$
4. When module i rolls back n steps, the switch controller checks the corresponding two rows in matrices KC_n and KP_n, namely $KC_n(i,j)$ and $KP_n(i,j)$ for $j=1,2,\ldots,M$. There are three possible conditions: 1). if $KP_n(i,j)=1$ then module j has to roll back (n+1) steps, 2). if $KP_n(i,j)=0$ and $KC_n(i,j)=1$, then module j has also to roll back n steps. 3). 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.

Let us define $RB_i(n)$, $n=1,2,\ldots,N$, to indicate the rollback step of module i. If module i rolls back n steps, then $RB_i(n)=1$, otherwise $RB_i(n)=0$. So, if $RB_i(n)=0$ for all n, then module i does not have to roll back. From the above conclusions and definitions, the condition of having a successful rollback recovery for a task can be expressed as follows:

The rollback of a task will be successful if one of the following two conditions is satisfied for all modules:

1. $RB_i(n)=0$, for all n.
2. If there is an integer n such that $RB_i(n)=1$, then either $KP_n(i,j)=0$ for all $j=1,2,\ldots,M$, or there exist integers j and w such that $KP_n(i,j)=1$, $RB_j(w)=1$, and $w > n$.

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

3.3 The Evalution of Multi-step Rollback

If a module i fails at time t_f during the k-th state-save interval, $T_i(k)$, then a single step rollback of module i is examined to see if it is sufficient to recover from the failure. The result may lead to rollback propagations and thus to multi-step 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 SSUs are exhausted. In this section a probability model is derived to evalute the coverage of the multi-step rollback recovery which indicates the effectiveness of present fault-tolerant mechanism. Suppose every module has (N+1) SSUs and the task is allocated to M modules including PMs and CMs. 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. 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 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. $b_{ij1}=b_{ij2}=\ldots=b_{ijN}=b_{ij}$. The exact value of b_{ij} is difficult to solve. 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 issued by 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 propagation if module i rolls back. Note $r_{ii}=1$ for all i.

E: The matrix $[e_{ij}]$, i,j=1,2,...,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 k-th 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. $N_t=\lfloor T_{ef}/(T_{ss}-T_{sv})\rfloor$.

u_{ijk}: The average memory reference rate from module i to module j during the k-th 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 of processes can be divided into different phases which have a constant reference rate [15,16]. If N_t is moderately large, u_{ijk} could be assumed to be a constant during a state-save interval.

To derive the coverage of a multi-step 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:

$$f_{ij} = f_{ji} = g_{ij} + g_{ji} - g_{ij}*g_{ji} \qquad \text{------(1)}$$

where $g_{ij} = (1/N_t) \sum_{k=1}^{N_t} (1-\exp(-(u_{ijk})*T_{ss}))$ represents the average probability of having an interaction issued by module i and directed 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 $\sum_{k=1}^{N_t}(u_{ijk})*T_{ss}$, we have the following relationship:

$$\sum_{k=1}^{N_t} u_{ijk}=(N_t*a_{ij})/(\sum_{m=1}^{M} a_{im}*e_{im}) \qquad \text{------(2)}$$

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

$$1/e_{ij} \geq u_{ijk} \geq 0 \qquad \text{------(3)}$$

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 $u_{ij,1}=u_{ij,2}=\ldots=u_{ij,N_t}$.

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

To solve for r_{ij} from f_{ij}, a fully connected network is drawn as Figure 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 [17] can be used to solve for r_{ij}:

$$r_{ij} = \bigcup_{q} (D_{ij,q}) \qquad \text{----(4)}$$

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

$$C(1) = (1/M) \sum_{i=1}^{M} \prod_{j=1}^{M} (1-r_{ij}(1-\sum_{k=1}^{M} b_{jk})) \qquad \text{-----(5)}$$

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

$$C(h) = C(1)(1-C(h-1))+C(h-1) \qquad \text{------(6)}$$

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

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

This example has the access localities 0.85 and 0.9 for processes which correspond to the experimental results obtained from Cm* [18]. The numerical results are presented in Table 1 and are also plotted in Figure 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 SSUs is enough to achieve a satisfactory coverage of rollback recovery. It should be particularly noted that the requirement of a small number of SSUs is mandatory for actual implementation.

4. THE PERFORMANCE OF ROLLBACK RECOVERY MECHANISM

Several methods for analyzing the rollback recovery system have been proposed [19 - 22]. They in general deal with a transaction-oriented database system and compute the 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 [23]. All of these approaches either assume the state restoration is obtainable by a single checkpoint or do not include the rollback capability at all. In this section, we explicitly take into account the problem of multi-step rollback and the risk of restart for the rollback recovery mechanism.

4.1 Notations and Assumptions

The following notations will be used in the sequel:

T_t:	The total execution time to complete the giv [illegible] sk with occurrence of errors. It includes the requir [illegible] tion time under error-free condition, the time lost due to rollbacks and restarts, and the time overhead for generating recovery blocks.
T_{real}:	The total execution time to complete the task without restart (i.e., all failures are recovered by rollbacks).
$T^j_{roll,m}$:	The time lost due to the j-th rollback in module m which consists of the set up time for resumption, tsb, and the computation undone by rollback.

T^i_{rst}:	The time lost due to the i-th restart which includes the set up 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 k-th rollback when the task can be completed without restart.
X^j_r (X^i_s):	The duration between the (j-1)-th and the j-th rollbacks (the (i-1)-th and the i-th restarts).
C(i):	The accumulated coverage of rollback recovery from a single step to i steps. This value is calculated by the Equations (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 the failure is recovered by the rollback.
Pr(m):	The probability of having m rollbacks during the time interval, T_{real}.
$Z_r(z)$, $Z_{st}(z)$:	The probability generating functions of Pr(m), $P_{st}(h)$ respectively.

$\phi_t(s)$, $\phi_{real}(s)$: 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. Suppose 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 j-th error occurs, module m spends $T^j_{roll,m}$ to recover from this error if the error is recoverable by a rollback. Otherwise, the whole task has to restart. $T^j_{roll,m}$ consists of the set up 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 time is postponed by max($T^j_{roll,m}$) where m=1,2,.. M for the rollback recovery of the j-th error. The resultant completion time will be the upper bound because of the following reasons: (1) $T^j_{roll,m}$ 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 $\sum_{m=1}^{M} T^j_{roll,m}$. Since the completion of task is regarded as the completions of all its processes, the time lost from the task's point of view could be max($T^j_{roll,m}$) but not larger than this maximal value. (2) The true delay impacted 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 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 [24]. Since $\lambda(t)$ is slowly time-varying (for example with a period of one day), it is 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 whenever it occurs (see Section 3 for a brief description on relaxing this assumption). From the definitions of P_s, P_b, and $P_{st}(h)$, we have $P_s=1-C(N')$ where N' is the number of states saved and $N' \leq N$, and each module has (N+1) SSUs. 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,.. N, and $P_{st}(1)=C(1)/P_b$. The occurrence of rollback and restart can be modelled as Poisson processes with means $\lambda_b=\lambda P_b$ and $\lambda_s=\lambda P_s$, respectively.

4.2 The Performance Model

The total task execution time, T_t, can be divided into several phases as shown in Figure 8. The last phase is always ended with the completion of task. Other phases are followed by a restart. So the amount of effective computation at the beginning of each phase is zero. During each phase, rollback recoveries are allowed so that the effective computation between rollbacks are 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 having R restarts, and the distribution of the durations of other phases which are defined as T^i_{rst} for i=1,2,..R.

In the last phase, the task will be executed from the beginning to the completion without any restart. Let T_{sv} denote the time overhead for generating a recovery block. The effective computation in a state-save interval under the error-free condition is $T_{ss}-T_{sv}$. 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 $h*T_{ss}$. The effective computation between two consecutive rollbacks becomes $(X_r-h*T_{ss})^+$ when a module rolls back h steps where $(X)^+=\max(0,X)$ is the positive rectification function. With the probability having an h-step rollback, $P_{st}(h)$, two functions are presented:

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

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

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

$$Pr(k)=P(TE_{k+1} > T_{ef})-P(TE_k > T_{ef})$$
$$=H(T_{ef},k)-H(T_{ef},k+1) \qquad \text{-----(9)}$$

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

$$\phi_{real} = \exp(-sT_{ef})(Z_r(\exp(-st_{sb})Z_{st}(\exp(-sT_{ss}))) \qquad \text{----(10)}$$

From Figure 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:

$$\phi_t(s) = \sum_{l=0}^{\infty} \exp(-slt_{su})\left(\frac{\lambda_s}{\lambda_s+s}\right)^{\ell}\left[\sum_{j=0}^{\ell} \binom{\ell}{j}(-1)^j \phi_{real}((j+1)(\lambda_s+s))\right] \qquad \text{----(11)}$$

This equation shows a general expression of the total execution time. For the system without rollback recovery mechanism, we can substitute P_s=1, P_b=0, and then $\phi_{real}(s)$ becomes $\exp(-s*T_{ef})$. The result obtained from the above equation is the same as that in [23]. The mean and variance of the total execution time can be obtained from $-\frac{\partial \phi_t}{\partial s}\Big|_{s=0}$ and $\frac{\partial \phi_t}{\partial s}\Big|_{s=0}$ respectively. In Figure 9, the mean execution time for the example in Section 3 is plotted. It is obvious that the overhead of generating recovery block 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. Since the embedded checking circuits are not cost-effective and complex [25], the overhead of generating recovery block can be reduced with a completely self-checking mechanism. Figure 10 expresses the variance of execution time for the previous example. It suggests that the prediction of the total execution time could be more accurate if 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 a larger and more uncertain recovery overhead is involved.

Another interesting parameter is the duration of state-save invocation, T_{ss}. The interval has two mutually conflicting effects. Figure 7 points out the increasing of T_{ss} will induce more rollback propagations and degrade the coverage (a 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 Figure 11. The figure expresses that there exists a linear relationship between T_t and T_{ss} when T_{ss} is small (where the overhead of generating recovery block 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.

5. CONCLUSION

We considered the design of a hardware recovery mechanism for a fault-tolerant multiprocessor with emphasis 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 automatically and regularly. This approach does not require high-level insertion strategies or limitations for setting up recovery points [6,7,8,26] and also does not require synchronization of state-save operations among different processors as does the COPRA system [27]. We derived mathematically the probability of multi-step 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.

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 should be considered to achieve the overall system reliability. 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 [28]) or by using reliable switches (e.g., 2X2 fault-tolerant switching element proposed in [29]). However, the faults occurred in the supplementary resources, like SSUs and monitor switches, do not cause damages to the computation itself but will change the recovery capability. Although the performability [30] 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 on 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 capabality may have no serious consequence to the completion of a given task. Moreover, the quality of the recovery procedure largely determines the distribution of the task completion time. The traditional reliability measures, such as reliability, availability, and computation capacity, taken separately, thus can not reflect the characteristics of this fault-tolerant system. However, the overhead required to treat an error, the contamination of error, and the effect on the task execution time, should be included to represent the effectiveness of fault-tolerance. In this paper, we achieved the fast treatment of failure by the automatic rollback recovery mechanism, and estimated the mean and variance of the completion time for a given task under moderate assumptions. We also point out that the assumption of no latency between error detection and error occurrence can be relaxed if we know the confident rollback distance or the distribution of this latency.

One major concern in most real-time applications,such as aircraft or industrial control, etc., is whether the required task can be completed prior to a given deadline or not. The rollback mechanism associated with each module not only offers system modularity and simplicity, but 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.

REFERENCE

1. J. A. Rohr, "Starex Self-repair Routines: Software Recovery in the JPL-STAR Computer," Proc. of the 3th Int'l Symp. on Fault-Tolerant Computing. 1973, pp. 11-16.

2. F. E. Heart, S. M. Qrnstin, W. R. Crowther, and W. B. Barher, "A New Minicomputer/Multiprocessor for the ARPA network," Proc. 1973 AFIPs Natl. Computer Conf., 1973, Vol 42, pp 529-537.

3. B. Randell, P. A. Lee, and P. C. Treleaven, "Reliablity Issues in Computing System Design," Computing Surveys, June 1978, pp.123-165.

4. B. Randell, "System Structure for Software Fault Tolerance," IEEE Trans. on Software Eng., Vol SE-1, Jun. 1975, pp. 220-232.

5. D. L. Russell, "Process Backup in Producer-Consumer Systems," Proc. of 6-th ACM Symp. on Operating System Principles, Nov. 1977, pp. 151-157.

6. 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.

7. K. H. Kim, "An Implementation of a Programmer-Transparent Scheme for Coordinating Concurrent Processes in Recovery," Proc. COMPSAC 80, Oct. 1980, pp. 615-621.

8. K. Kant and A. Silberschatz, "Error Recovery in Concurrent Processes," Proc. COMPSAC 80, Oct. 1980, pp. 608-614.

9. 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, pp. 283-298.

10. Y. H. Lee, and K. G. Shin, "Rollback Propagation Detection and Proformance Evaluation of FTMR^{2}M -- A Fault-Tolerant Multiprocessor," 9th Annual Symp. on Computer Architecture, April 1982, pp. 171-180.

11. R. J. Swan, S. H. Fuller, and D. P. Siewiorek, "Cm*: a Modular Multi-Microprocessor," Proc. 1977 AFIPS Natl. Computer Conf., Vol. 46, 1977, pp. 637-644.

12. P. H. Enslow, "Multiprocessor Organization - A Survey," Computing Surveys, Vol. 9, No. 1, March 1977, pp. 101-129.

13. K. H. Kim, "Error Detection, Reconfiguration and Recovery in Distributed Processing Systems," Proc. 1st Int'l Conf. on Distributed Computing Systems, Oct. 1979, pp. 284-295.

14. J. J. Shedletsky, "A Rollback Interval for Networks with an Imperfect Self-Checking Property," IEEE Trans. on Computer, Vol. c-27, No. 6, June 1978, pp. 500-508.

15. A. W. Madison and A. P. Batson, "Characteristics of Program Localities," Comm. of ACM, Vol. 19, May 1976, pp. 285-294.

16. A. P. Batson, "Program Behavior at the Symbolic Level," Computer, Nov. 1976, pp. 21-26.

17. S. Rai, and K. K. Aggarwal, "An Efficient Method for Reliability Evaluation of a General Network," IEEE Trans. on Reliability, Vol. R-27, No. 3, Aug. 1978, pp. 206-211.

18. S. H. Fuller et al., "Multi-Microprocessors: An Overview and Working Example," Proc. IEEE, Vol. 66, No. 2, Feb. 1978, pp. 216-228.

19. K. M. Chandy, J. C. Browne, C. W. Dissly anb W. R. Uhrig, "Analytic Models for Rollback and Rocovery Strategies in Data Base Systems," IEEE Trans. of Software Eng., vol. SE-1, no. 1, March 1975, pp. 100-110.

20. K. M. Chandy and C. V. Ramamoorthy, "Rollback and Recovery Strategies for Computer Programs," IEEE Trans. on Comp., Vol. C-21, No. 6, June 1972, pp. 546-556.

21. E. Gelenbe and D. Derochette, "Proformance of Rollback Recovery Systems under Intermittent Failures," Comm. of the ACM, Vol. 21, No. 6, June 1978, pp. 493-499.

22. J. W. Young, "A First Order Approximation to the Optimum Checkpoint Interval," Commu. of the ACM, Vol. 17, No. 9, Sep. 1974, pp. 530-531.

23. X. Castillo and D. P. Siewiorek, "A Performance-Reliability Model for Computing Systems," Proc. of the 10th Int'l Symp. on Fault-Tolerant Computing, 1980, pp. 187-192.

24. X. Castillo and D. P. Siewiorek, "Workload, Performance, and Reliability of Digital Computing Systems," Proc. of the 11th Int'l Symp. on Fault-Tolerant Computing, 1981, pp. 84-89.

25. W. C. Carter et al., "Cost Effectiveness of a Self Checking Computer Design," Proc. of the 7th Int'l Symp. on Fault-Tolerant Computing, 1977, pp. 117-123.

26. F. J. O'Brien, "Rollback Point Insertion Strategies," Proc. of the 6th Int'l Symp. on Fault-Tolerant Computing, 1976, pp. 138-142.

27. C. Meraud and F. Browaeys G. Germain, "Automatic Rollback Techniques of the COPRA Computer," Proc. of 6th Int'l Conf. on Fault-Tolerant Computing, 1976, pp. 23-29.

28. G. B. Adams and H. J. Siegel, "A Fault-Tolerant Interconnection Network for Supersystems," IEEE Trans on Computer, Vol. C-31, No. 5, May 1982, pp. 443-454.

29. W. Lin and C. L. Wu, "Design of a 2x2 Fault-Tolerant Switching Element," 9th Annual Symp. on Computer Architecture, April 1982, pp.181-189.

30. J. F. Meyer, "On Evaluating the Proformability of Degradable Computing Systems," IEEE Trans. on Computer, Vol. C-29, No. 8, Augest 1980, pp. 720-731.

Appendix A. Calculation of the probability of having k rollback within T_{real}

From the difinition of $P_{st}(h)$, the task will roll back h steps with probability $P_{st}(h)$ after a failure within the last phase T_{real}. Let the rollback distance for the j-th rollback recovery is T^j_{roll} which is approximately equal to hT_{ss} with a probability $P_{st}(h)$. Thus the accumulated effective computation time before the k-th rollback, TE_k, is given by

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

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

$$f_\alpha(t) = \sum_{h=1}^{N} P_{st}(h)(1-\exp(-\lambda_b hT_{ss}))\delta(t) + \exp(-\lambda_b t) \sum_{h=1}^{N} P_{st}(h)\exp(-\lambda_b hT_{ss}) \qquad \text{-----(A.2)}$$

where $\delta(t)$ is impluse function. Let Z represent $\sum_{h=1}^{N} P_{st}(h)\exp(-\lambda_b hT_{ss})$. Thus f_α is simplified by

$$f_\alpha(t) = (1-Z)\delta(t) + \exp(-\lambda_b t)Z \qquad \text{----(A.3)}$$

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

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

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

$$P(TE_k \le t) = \int_0^t f_{te,k}(\tau)d\tau$$

$$= \sum_{i=0}^{k-1} \binom{k}{i}(1-Z)^i(Z)^{k-i}G_{k-i}(t)+(1-Z)^k \quad \text{----(A.5)}$$

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

Appendix B. Calculation of the characteristic function of total execution time, ϕ_t

From Fig. 8, the total execution time T_t is the sum of T_{real} and T_{rst}, where $T_{rst} = \sum_{i=1}^{\ell} T^i_{rst}$ if there are ℓ restarts. With the conditional probability of T_t, we have the following equation:

$$E(T_t|T_{real}) = E(T_{real})+E(T_{rst}|T_{real}) \quad \text{----(B.1)}$$

It is assumed that the time interval between the (i-1)-th and the i-th restarts X_s^i is exponential distributed with mean $1/\lambda_s$. Thus, for a given T_{real}, the time lost due to the i-th restart is randomly distributed between t_{su} to $T_{real}+t_{su}$ with the density function, $f_{rst,i}$, which is given in the following:

$$f_{rst,i}(t+t_{su}) = \frac{\lambda_s \exp(-\lambda_s t)}{1-\exp(-\lambda_s T_{real})} \qquad \text{for } 0 \le t \le T_{real} \qquad \text{----(B.2)}$$

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

$$P_{rs|T_{real}}(\ell) = (\exp(-\lambda_s T_{real}))(1-\exp(-\lambda_s T_{real}))^{\ell} \qquad \text{----(B.3)}$$

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

$$\phi_{t|T_{real}}(s) = (\exp(-sT_{real})) \sum_{\ell=1}^{\infty} P_{rs|T_{real}}(\ell)(\phi_{rst|T_{real}}(s))^{\ell} \qquad \text{---(B.4)}$$

where $\phi_{rst|T_{real}}(s)$ is the characteristic function of the time loss due to a restart for a given T_{real}, i.e., the Laplace transformation of $f_{rst,i}(t)$. By substituting $P_{rs|T_{real}}(\ell)$ and $\phi_{rst|T_{real}}(s)$ into equation (B.4) and integrating with the density function of T_{real}, the characteristic function of T_t is obtained as Equation (11).

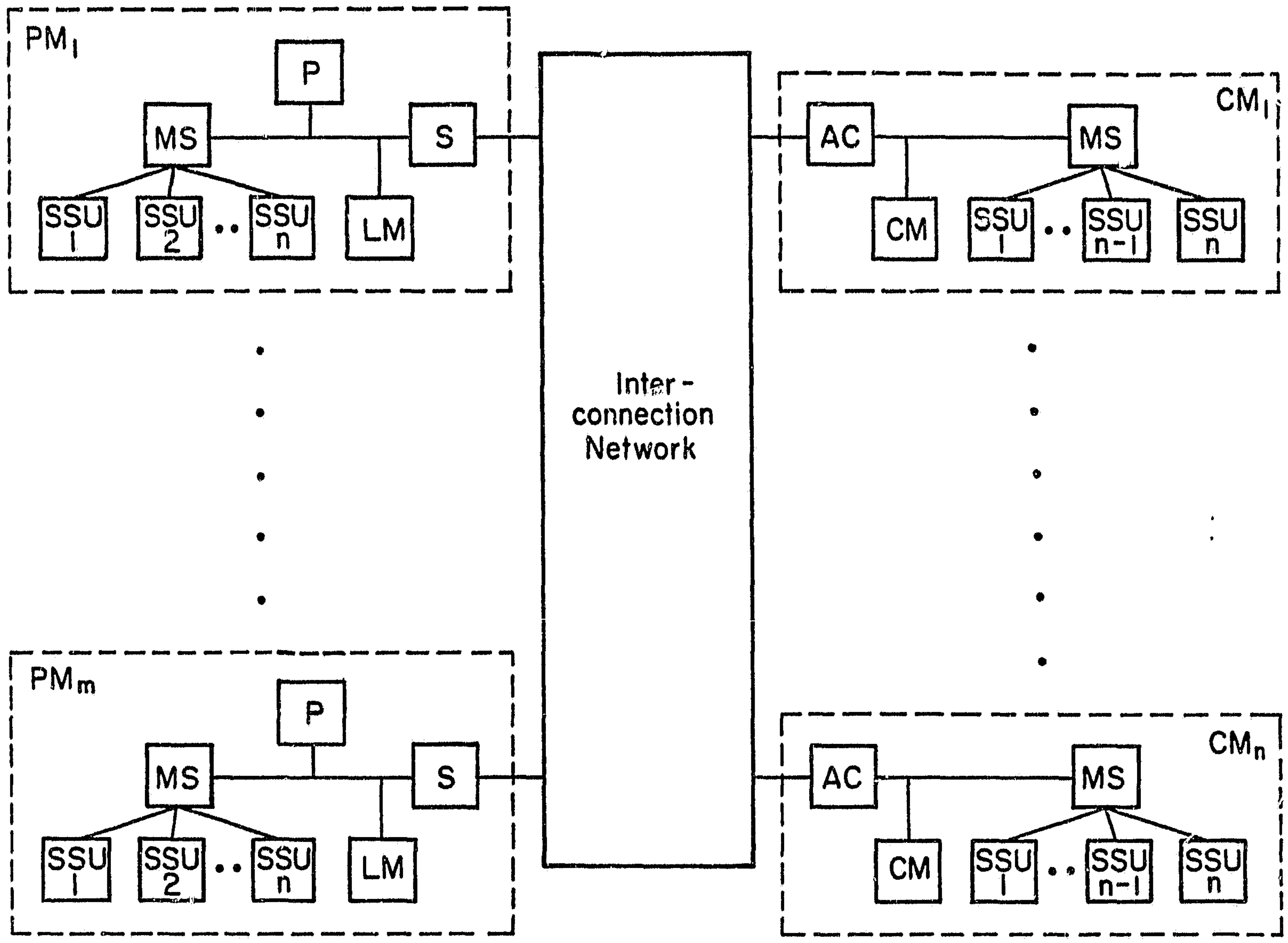
i \ C(i)	case 1	case 2	case 3
1	0.75067	0.68610	0.44713
2	0.93783	0.90147	0.69433
3	0.98449	0.96907	0.83100
4	0.99612	0.99029	0.90656
5	0.99902	0.99695	0.94834

case 1: with minimum f_{ij} and N_t=100

case 2: with minimum f_{ij} and N_t=10

case 3: with maximun f_{ij}

Table 1. A Numerical Example for the Coverage of Multi-step Rollbacks



P = processor	CM = common memory
S = switch	AC = access controller
MS = monitor switch	SSU = state-save unit
LM = local memory	

Figure 1. The Organization of a Fault-Tolerant Multiprocessor using a Rollback Recovery Mechanism

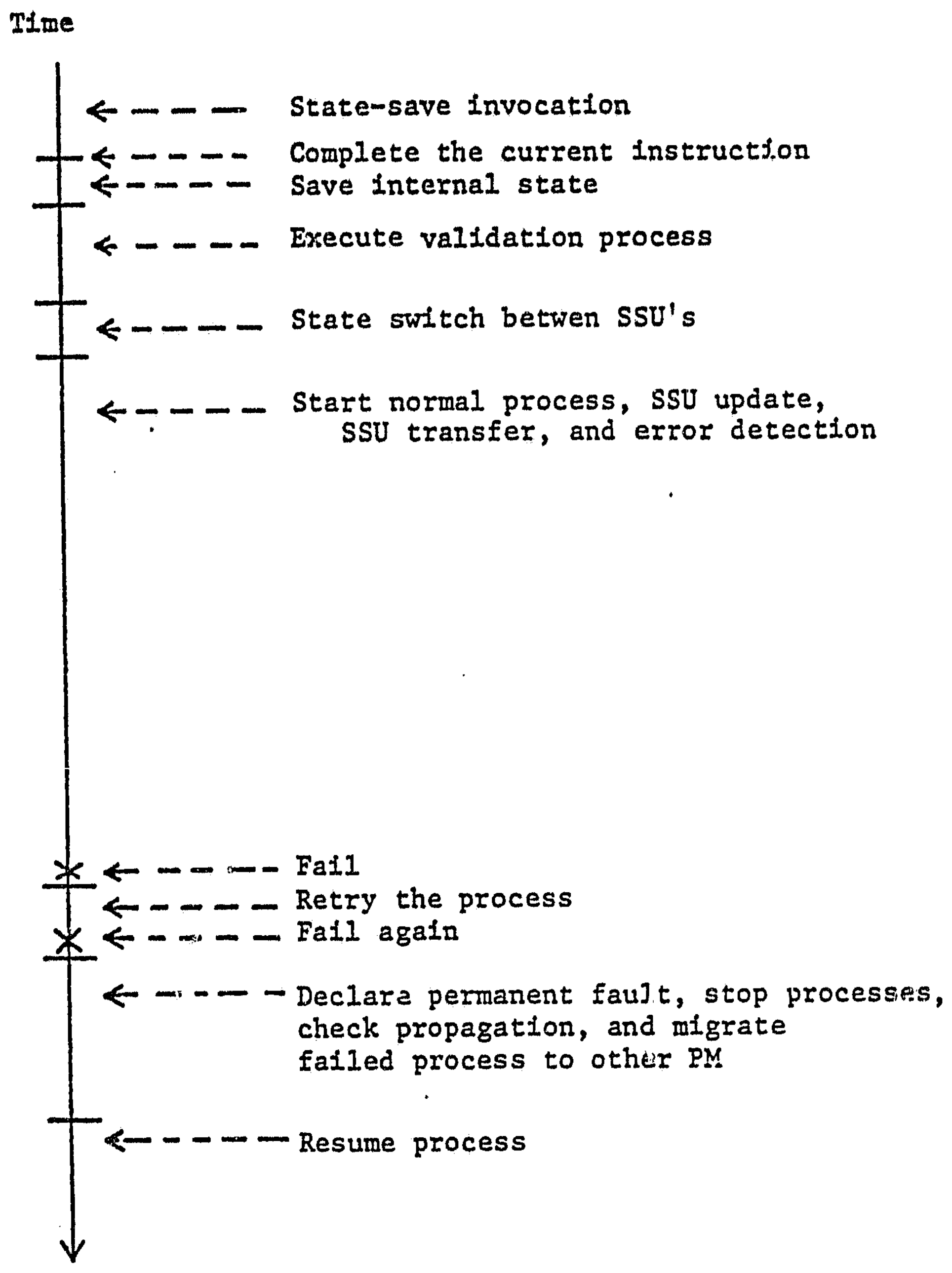

Time
State-save invocation
Complete the current instruction
Save internal state
Execute validation process
State switch betwen SSU's
Start normal process, SSU update, SSU transfer, and error detection
Fail
Retry the process
Fail again
Declare permanent fault, stop processes, check propagation, and migrate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

Fig. 2. Sequence of a Rollback Recovery

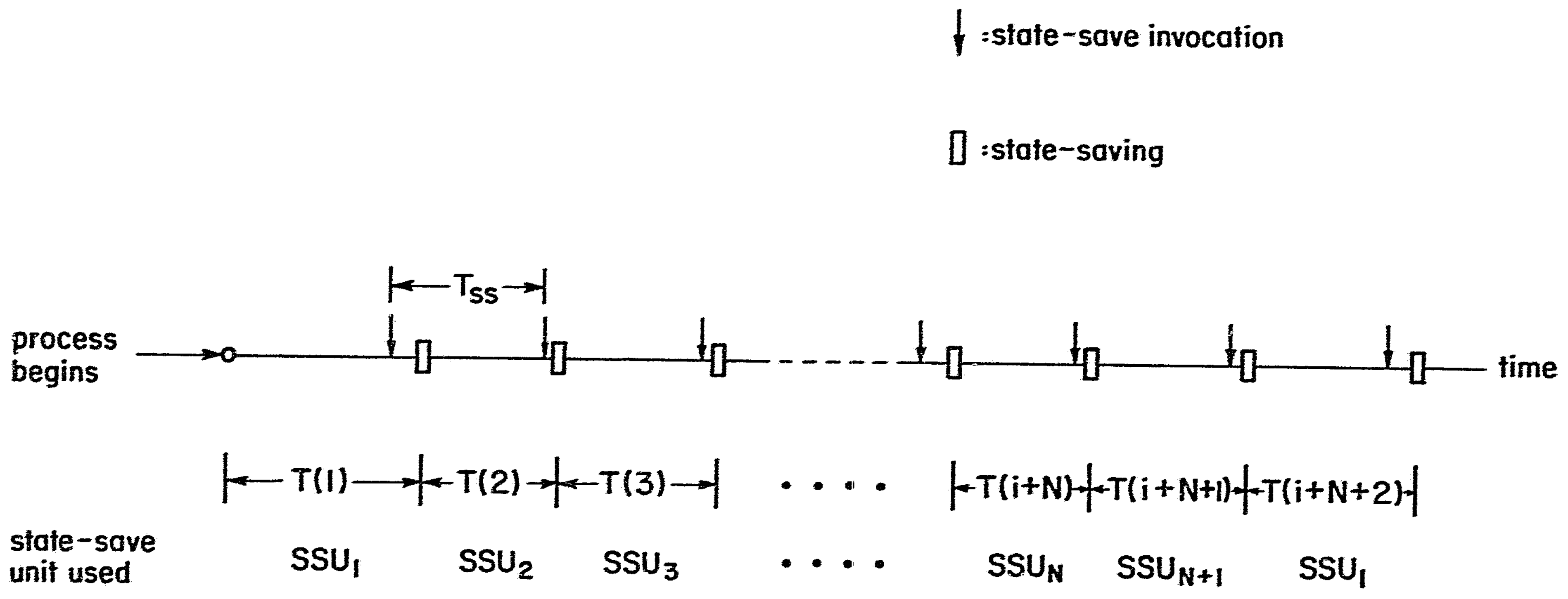
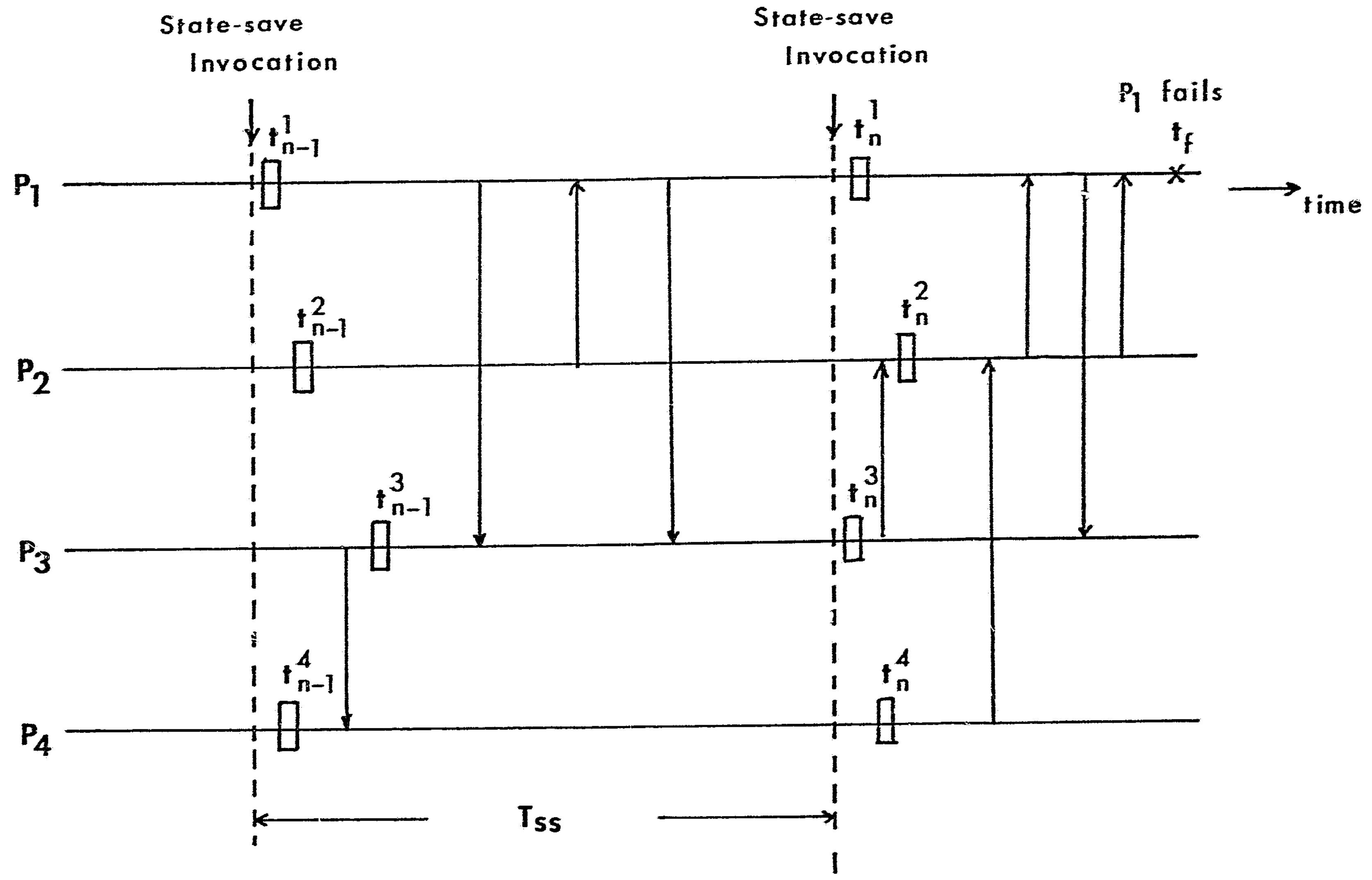


Figure 3. State-save Operations in One Module


State-save Invocation
State-save Invocation
P1 fails
t_f
t^1_{n-1}
t^1_n
t^2_{n-1}
t^2_n
t^3_{n-1}
t^3_n
t^4_{n-1}
t^4_n
P_1
P_2
P_3
P_4
time
Tss

(a)

$$KC_{n-1} = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \qquad KC_{n} = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{bmatrix}$$

$$KP_{n-1} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \qquad KP_{n} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

(b)

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

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

(c)

Figure 4. An Example of Rollback Propagation and Multi-step Rollback

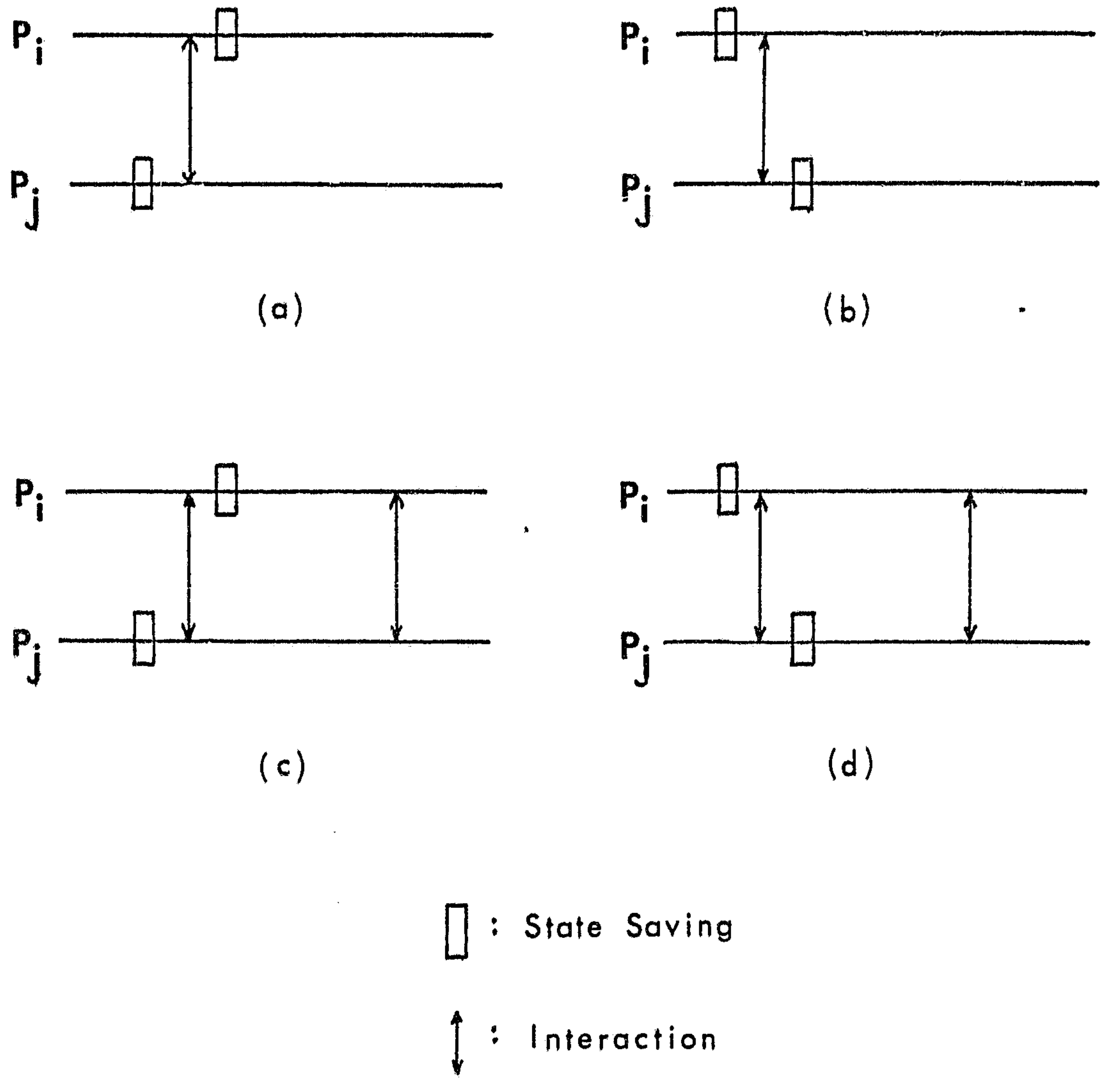
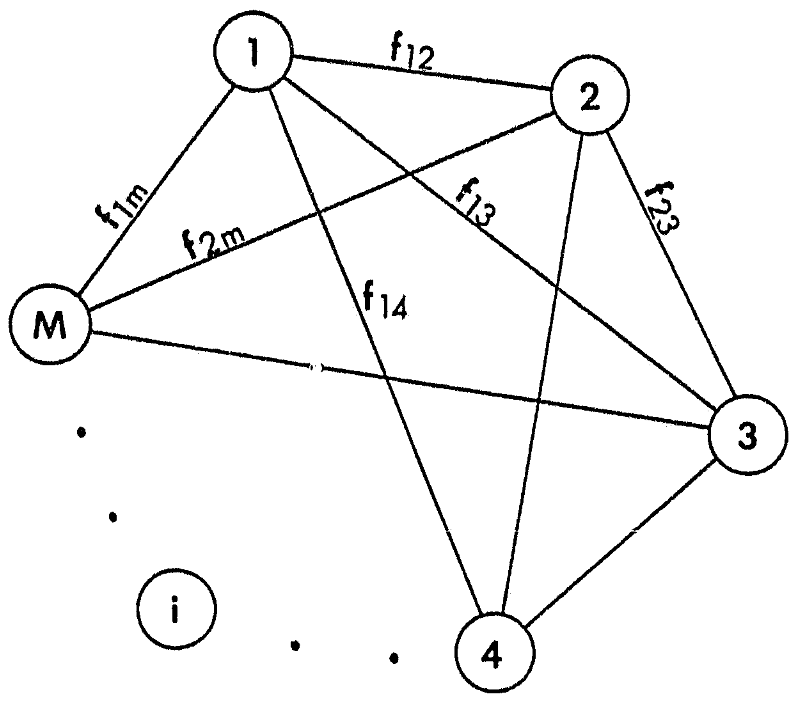


Figure 5. Interaction Patterns Related to Rollback Propagation



(i) : Module i

Figure 6. The Rollback Propagation Network

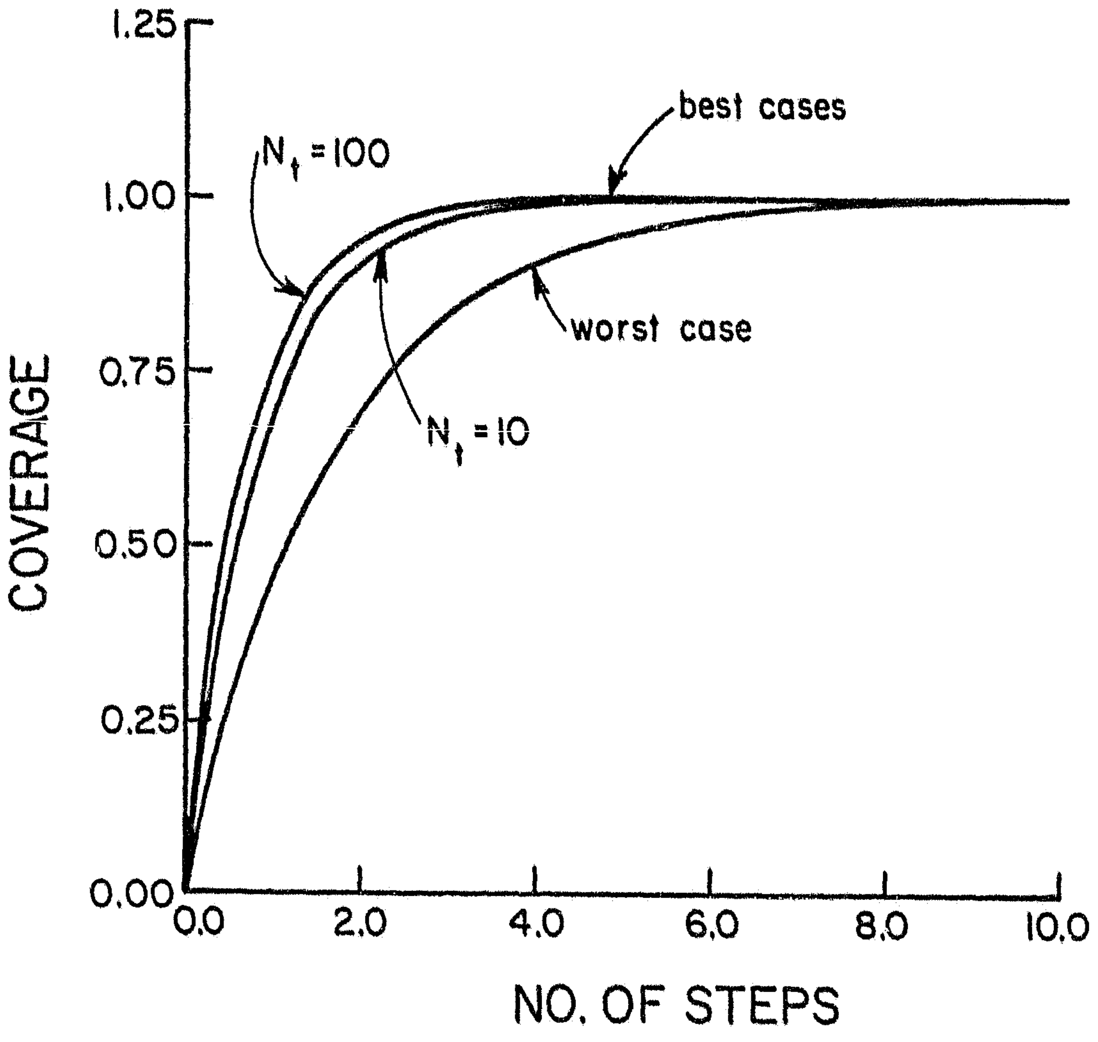


Figure 7. Rollback Coverage vs. No. of Rollback Steps

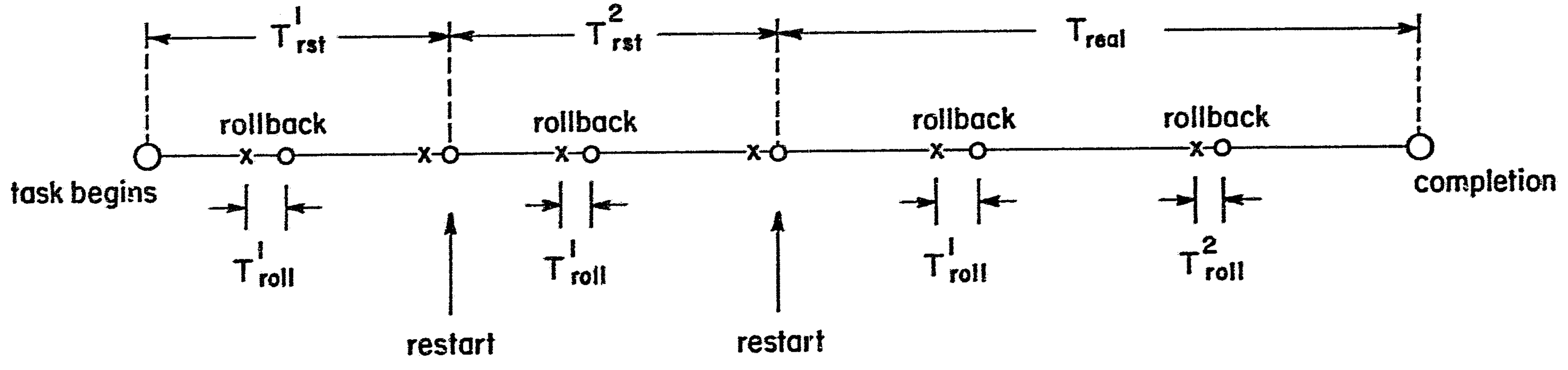


Figure 8. Task Execution Phases

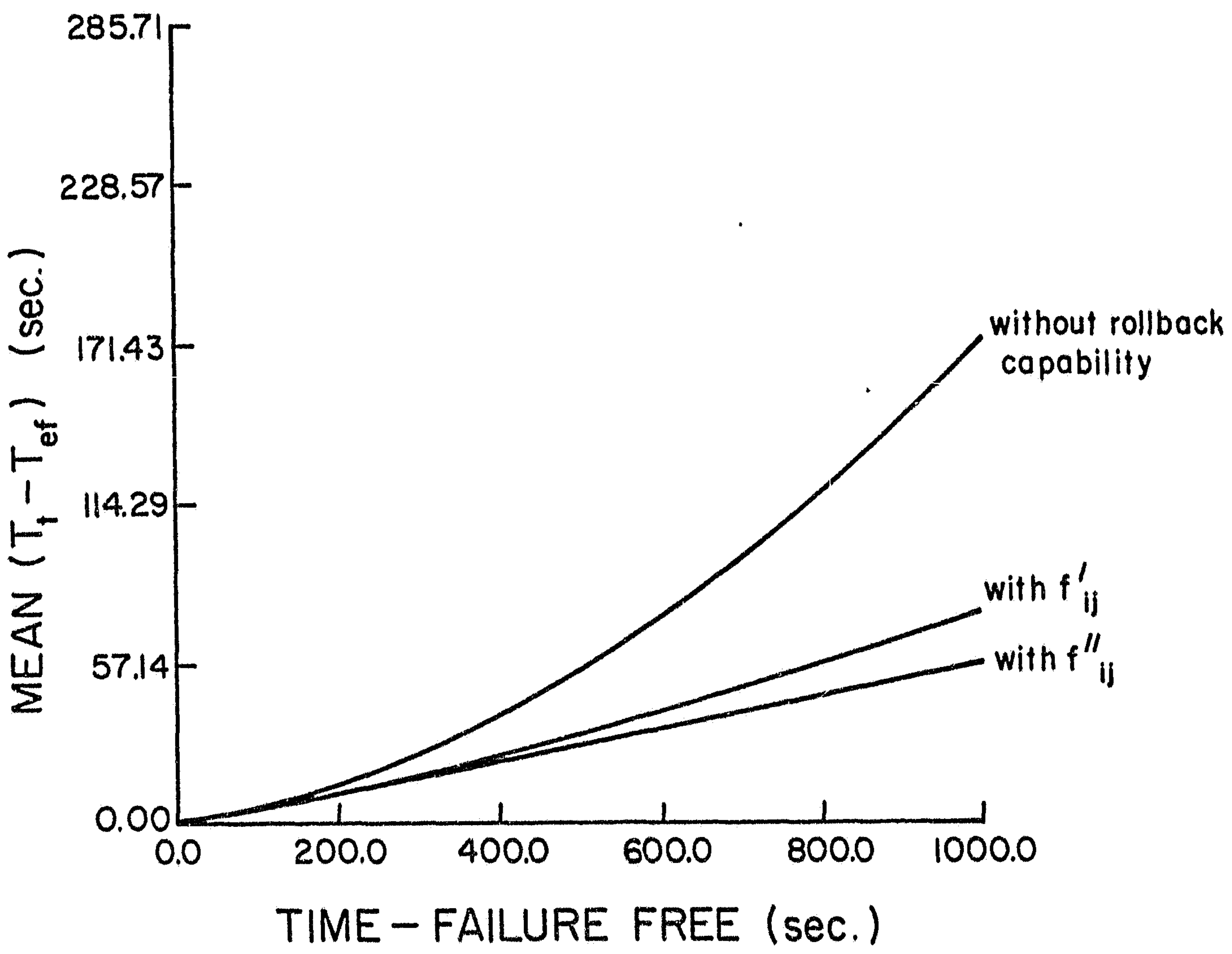


Figure 9. Mean Time-Overhead vs. Error-Free Execution Time

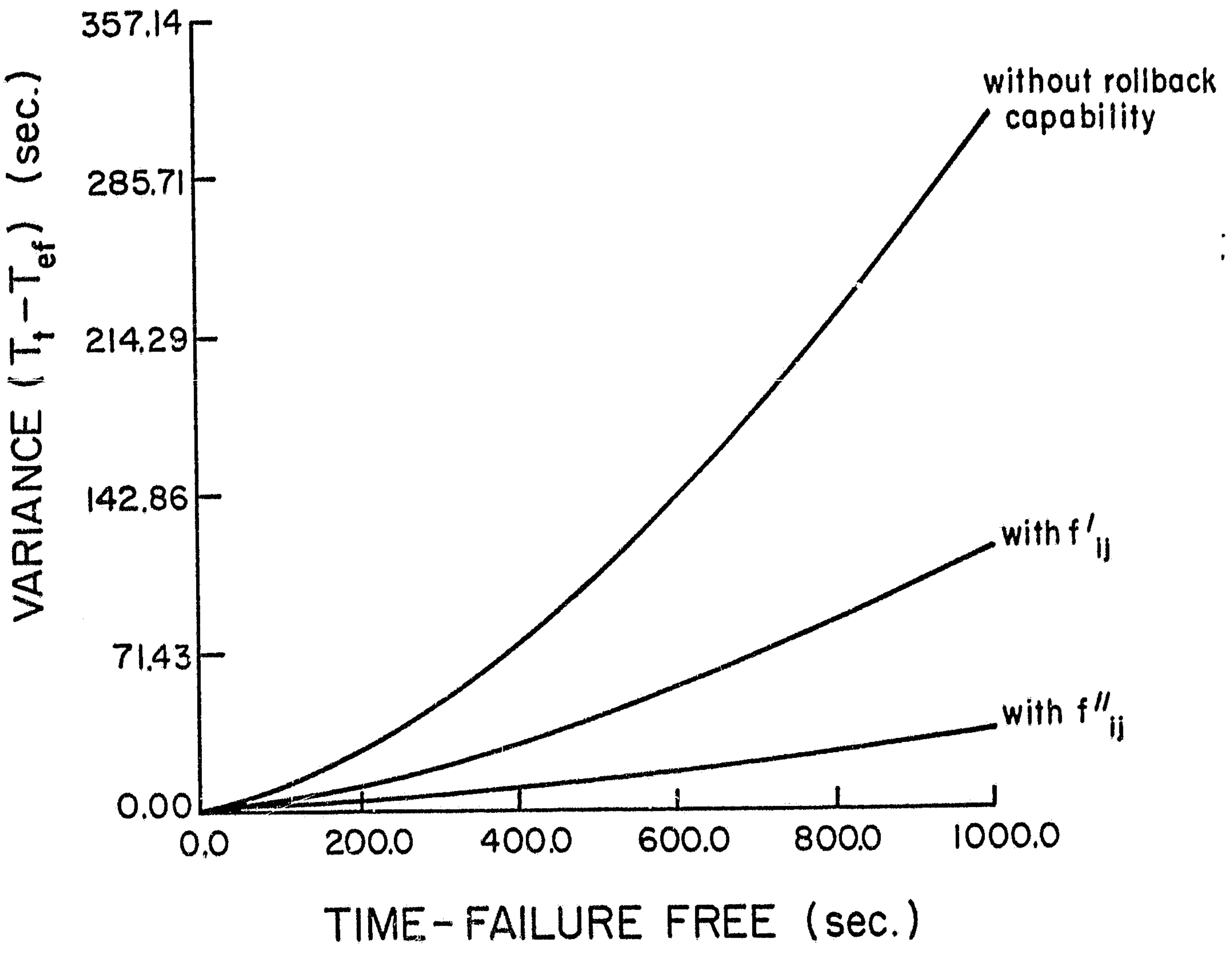


Figure 10. Variance of Time-Overhead vs. Error Execution Time

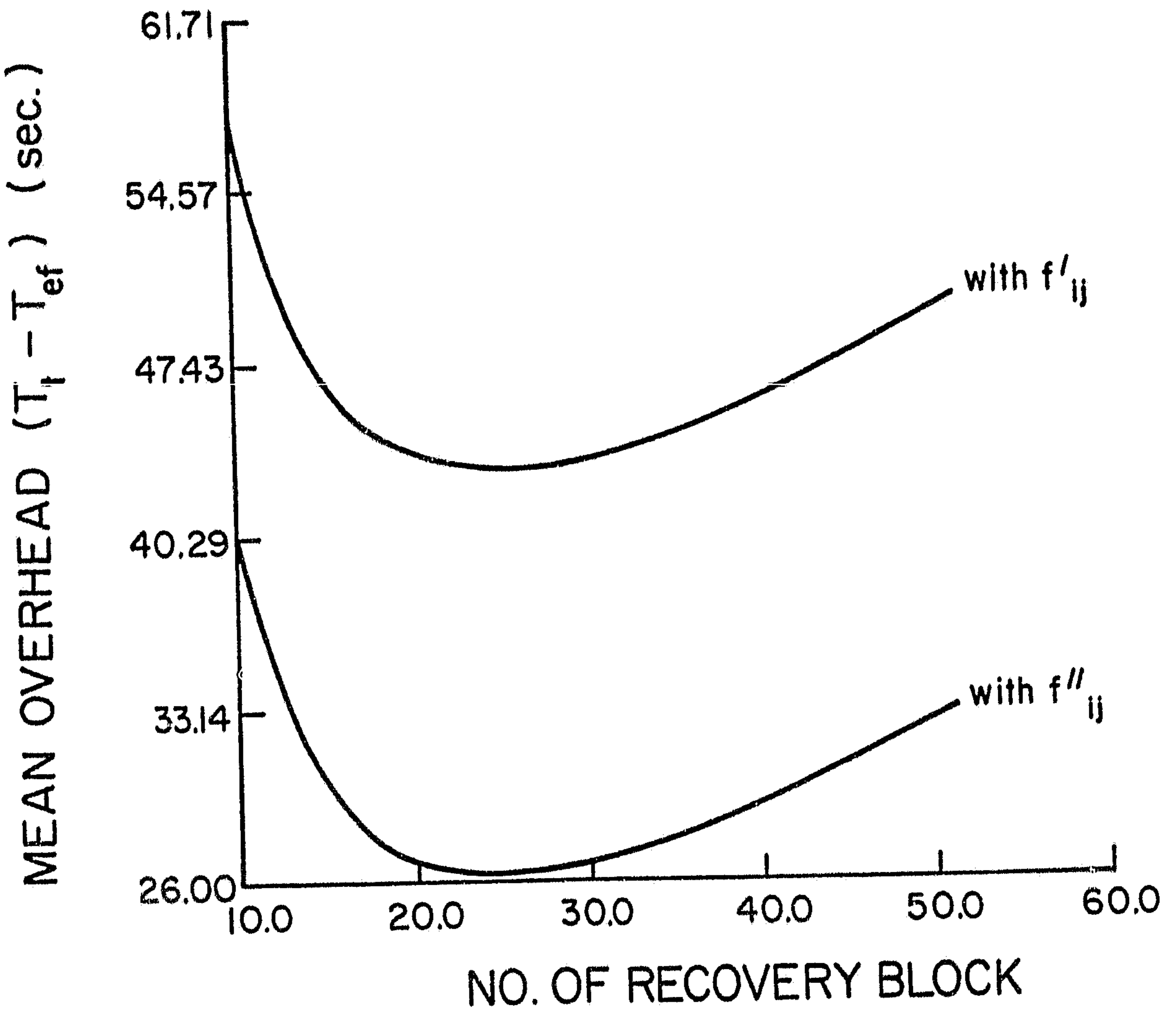


Figure 11. Mean Time-Overhead vs. Total Number of Recovery Blocks for a Given Task