NASA Contractor Report 172571

NASA-CR-172571
19860002439

INTEGRATED ANALYSIS OF ERROR DETECTION AND RECOVERY

Kang G. Shin and Yann-Hang Lee

THE UNIVERSITY OF MICHIGAN
Ann Arbor, Michigan

Grant NAG1-296
October 1985

NASA
National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665

TABLE OF CONTENTS

1. INTRODUCTION 1

1.1. Outline of the Report 2

1.2. Survey of Previous Works 4

2. ERROR DETECTION PROCESS 6

2.1. Classification of Error Detection Mechanisms 9

2.2. The Model of Error Detection Process 11

2.2.1. Model Development 12

2.2.2. Mathematical Description of State Transitions 16

3. ANALYSIS OF THE IMPACTS OF ERROR DETECTION MECHANISMS 20

3.1. Estimation of the Probability of Producing and Unreliable Result 20

3.2. Evaluation of Computation Loss and Execution Cost 22

3.2.1. Recovery Strategies and Their Respective Overheads 25

3.2.2. Calculation of Computation Loss and Execution Cost 32

3.3. Design Consideration for Detection Mechanisms 35

4. EVALUATION OF SOFTWARE RECOVERY BLOCKS 39

4.1. Evaluation of Asynchronous Recovery Blocks 43

4.1.1. Modeling Assumptions 44

4.1.2. A Model for Asynchronous Recovery Blocks 45

4.1.3. The Analysis of Asynchronous Recovery Blocks 48

4.2. Synchronized Recovery Blocks 53

4.3. Implantation of Pseudo Recovery Points 60

N86-11906#

5. DESIGN AND EVALUATION OF HARDWARE RECOVERY BLOCKS 64

5.1. Hardware Recovery Blocks for Multiprocessor 64

5.1.1. Processor Module, Common Memory, and State-Save Mechanism 65

5.1.2. Rollback Recovery Operations of a Task 71

5.2. Rollback Propagation and Multi-step Rollback 72

5.2.1. Rollback Propagation and Multi-Step Rollback 73

5.2.2. The Detection of Rollback Propagation 77

5.2.3. The Evaluation of Multi-Step Rollback 79

5.3. The Performance of Rollback Recovery Mechanism 85

5.3.1. Notations and Assumptions 85

5.3.2. The Performance Model 87

6. CONCLUDING REMARKS 93

7. REFERENCES 98

8. APPENDIX A 105

9. APPENDIX B 106

10. APPENDIX C 108

FIGURES

Figure 2.1.	The Error Detection Process.
Figure 2.2.	The Model for Error Detection Process.
Figure 2.3.	A Cycle of Periodic Diagnostics.
Figure 2.4.	A Markov Chain for the Recovery from an Intermittent Fault.
Figure 3.1.	Max (p_e) versus the Coverage of Single Diagnostic, ξ.
Figure 3.2.	Max (p_e) versus t_n.
Figure 3.3.	The Flow Chart of Recovery Processes.
Figure 3.4.	Probablistic Flow Diagram of Recovery Processes.
Figure 3.5.	The Effects of Periodic Diagnostics.
Figure 3.6.	Design Space for Coverage and Mean Error Latency Subject to Constaints of p_f and $COST$.
Figure 4.1.	A History Diagram of Occurance of Interactions and Recovery Points.
Figure 4.2.	The Model of Asynchronous RB's for 3 Processes. .
Figure 4.3.	The Simplified Model of Asynchronous RB's for n Processes.
Figure 4.4.	The Construction of State $S_2 1$ and $S_2 2$ of Discrete Markov Chain Y_d.
Figure 4.5.	Mean Value of X vs. the Number of Processes.
Figure 4.6.	The Density Function of X_3 $f_x(t)$.
Figure 4.7.	Establishment of Recovery Lines upon Synchronization Requests.
Figure 4.8.	Establishment of Pseudo Recovery Points for Rollback Error Recovery.
Figure 5.1.	The Organization of a Fault-Tolerant Multiprocessor Using a Rollback Recovery Mechanism.
Figure 5.2.	Sequences of a Rollback Recovery.
Figure 5.3.	State-save Operations in One Module.
Figure 5.4.	An Example of Rollback Propagation and Multi-step Rollback.
Figure 5.5.	Interaction Patterns Related to Rollback Propagation.
Figure 5.6.	The Rollback Propagation Network.
Figure 5.7.	Rollback Coverage vs. No. of Rollback Steps.
Figure 5.8.	Task Execution Phases.
Figure 5.9.	Mean Time-Overhead vs. Error-Free Execution Time.
Figure 5.10.	Variance Time-Overhead vs. Error-Free Execution TIme.
Figure 5.11.	Mean Time-Overhead vs. Total Number of Recovery Blocks for a Given Task.

TABLES

Table 3.1. Performance Expressions of Several Parameters in Error Recovery.
Table 4.1. Mean Values of X and L_i for Constant p.

1. INTRODUCTION

Reliability in computing systems can be obtained through two means: fault avoidance and fault tolerance. Fault avoidance aims to prevent the occurrence of faults so that system operations are not disturbed. The typical methods of fault avoidance are the selection of reliable components and design of reliable structures. The second approach, fault tolerance, is to fortify system operations against faults. The basic idea of fault tolerance is to install redundancy in data or hardware components. Such redundancy provides additional information with which to mask or recover from the fault. Fault tolerance is becoming important because as time progresses, more computers can be expected to be involved in critical applications where the repair of faulty components is inhibited and/or human lives could be at stake. For this reason, specifications of the failure probability of computers can be expected to be stringent. Indeed, the overall failure probability of such a computer has to be much less than the failure probability of any one of its components. Simple fault avoidance is not sufficient when extreme reliability is required: one requires the ability to tolerate faults as well.

To evaluate the extent of fault tolerance in any system, two aspects have to be examined. One is the survivability of system which is concerned with the type and number of faults that the system can tolerate before it fails. The other aspect is concerned with how considerably installed techniques of fault-tolerance affect the normal operational characteristics of the system. In most traditional reliability-related models, several measures, like system reliability, availability, computation capacity or performability, etc., are evaluated in dealing with the first problem. These models usually make certain assumptions about failure treatment and lack the capacity to evaluate the impacts of fault-tolerance techniques on normal system performance, and the consequences of mis-handling of errors. It is easy to see that in some cases, a mis-handling of failure may be more serious than the occurrence of failure. In this report, we study the second aspect in detail and establish several models for error detection and recovery.

In the discussions that follow, we adopt the terminology of Randell et. al., [59]. A *fault* means a malfunction in a physical component. It does not affect the system operations before its manifestation which we term the occurrence of an *error*. Thus, a fault is not "visible" before a corresponding error is induced. An error can be generated because of the existence of a fault or through the propagation of other errors. When an error occurs, the system is in an erroneous state, by definition.

1.1. Outline of the Report

To exhibit fault tolerance during normal operation, the first step is to identify the occurrence of a fault which has to be recognized through a process of error detection. By investigating error symptoms or executing a diagnostic, the source of the error is determined. The second step is to isolate the faulty components from system and to reconfigure the hardware/software so that the whole system remains or becomes operational. Finally, the computation processes which have been contaminated by the error have to be recovered. These three steps can be realized serially. However, they depend on each other. For instance, when error detection mechanisms are sluggish, recovery of computation becomes difficult and inefficient since the errors will have propagated over a wide area. Reconfiguration also affects the capability to detect further errors, which ability is necessary especially when the mission life cycle is long and there is insufficient redundancy.

In this report, We assume that there are enough resources such that the system can be reconfigured after an error is detected. It is likely to see that the impacts of detection mechanisms on the system performance and the efficiency of various recovery procedures.

In Chapter 2, the property of fault and detection mechanisms are studied. A fault is *latent* when it exists in the system but does not harm any operations. Once an error is generated, the detection mechanisms are supposed -- in contemporary models -- to identify the error immediately. Nevertheless, some errors may not be captured by these detection mechanisms upon its appearance and may then spread as a result of subsequent flow of information. The time disparity between generation and detection of an error is called as *error latency.* A model is established to incorporate both imperfect coverage and the existence of error latency.

Because of the existence of error latency, the system or task may have to suffer certain defects, like the production of reliable computation results, the delay in task completion, etc. In Chapter 3, as well as the measures of these impacts, the design of detection mechanisms is studied which, in corporated with recovery procedures, will achieve the specified requirements of system performance.

We also model and analyze the recovery of the computation processes that are contaminated by the error. Depending on the detection mechanisms used, corresponding recovery methods will be adopted. The performance of these recovery methods is discussed and evaluated quantitatively. Among these recovery methods, we concentrate on rollback recovery because of its variety and the importance for distributed system. The evaluation of software recovery blocks in cooperating processes is carried out in Chapter 4. Finally, the hardware implemented recovery blocks in multiprocessor system are designed and analyzed in Chapter 5. The report concludes with Chapter 6.

1.2. Survey of Previous Works

There is an extensive literature in the area of error detection, error recovery, and models of fault-tolerant computers. Surveys of error handling were made by Randell et al. [59], Kim [38], Anderson [2], and Siewiorek [68] where procedures of error handling are described. These procedures provided for error detection and recovery considered separately. The relationship between these facts is not considered.

Ever since Ball and Hardie [5] gave the statistical results about the failures in computers, many papers have appeared in the literature concerning the modeling of failures, detections and associated performance. Gunther [33] and Shedletsky [62] assumed that the input signals are independent and then applied the concept of the fault set to estimate the probability of error generation and error detection. Agrawal studied the same problem using information theory [1]. Several experiments and simulations were carried out to measure the probability of error detection and the distribution of detection time by injecting hardware faults [8, 22-23, 48, 50-52, 75, 79]. These measurements determined the interval between fault injection and error detection. However, no direct way is known for estimating the moment of error occurrence which is within the above interval.

About the intermittent faults, their transient behavior are assumed to have Markovian properties by Breuer [11] and then extensively used by Hopkins [34], Malaiya [48-49], Ng [55], etc. Tasar, in a literature survey of intermittent faults [74], discussed the theoretical modeling of intermittent faults, and then pointed out the difficulty of diagnosing intermittent faults. Indeed, run-time detection is more efficient than any kind of diagnostic in the matter of detecting intermittent faults.

There are many studies of error-correcting codes, and self-checking circuits. Several papers examined the impact of partial self-checking on system reliability [23-24,63,70]. Bossen and Hasio [10], and Carter et al. [15] investigated the design problems of self-checking circuits, the placement of these circuits, and their cost effectiveness. However, little work has been done on performance measures for function level detections. Courtois presented experiments on a microprocessor for the detection of error by the mechanisms of timeout and invalid op-code [22]. Andrews discussed the inclusion of executable assertion to provide run-time error detection [3]. The overheads in program size and execution time are measured in the experiments.

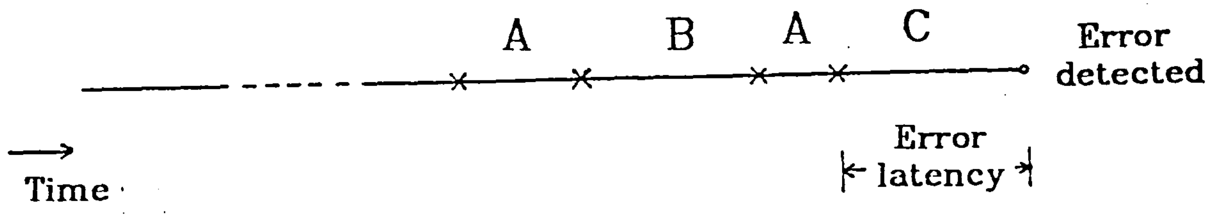
Among error recovery methods, rollback recovery has received more attention than any of the others. The recovery block, proposed by Horning [35] and Randell [58] has been widely used for backward error recovery. For the implementation of recovery blocks in multiprocess systems, Merlin and Randell suggested the conversation scheme in [54,59]. Kim proposed more flexible programming structions in [40]. There are many studies concerning rollback propagation and the domino effect, such as the detection of rollback propagation by Wood [80], the insertion of redundant recovery points by Kim [37,39] and Kant [36], the synchronized rollback schemes for transactions using a two-phase commitment protocol by Gray [32], Kohler [41], and Ferran [27], and the restoration of information for blocking propagation by Russell [60-61]. Although the proposed refinements can be proved to avoid the domino effect, the overheads associated with them are generally high. However, no quantitative estimates of the probability of a domino effect and the associated loss have been made so that it is impossible to compare the relative benefit which we obtain from these respective refinements.

Several methods for analyzing the performance of rollback recovery system have been proposed [4,19-21,30-31,78,80]. They, in general, deal with a transaction-oriented database system and compute the optimum length of the intercheckpoint interval. The assumptions that the checkpoints are completely reliable and no multi-step rollback is involved are adopted to simplify the derivations. Castillo and Siewiorek studied the expected execution time which is required to complete a task with the restart recovery method [16]. Brodetskiy included the problems of single step rollback and restart in [12-14]. These studies, in general, confine to the mono-processing environment. For distributed processes, rollback propagation and multi-step rollback have to be considered.

2. ERROR DETECTION PROCESS

When there exists a fault in a computer system, an input signal may cause the fault to induce some errors, or it may simply be unaffected by this fault and produce a correct output. The fault is said to be *latent* if it does not harm normal operations. The time interval between the moments of fault occurrence and error occurrence is called *fault latency*. For an ultra-reliable system, a latent fault is a considerable threat since it may cause a catastrophe in the event that more than one latent fault becomes active at the same time.

When an error is generated, it is desired that the error detection mechanisms associated with the system identify it immediately. Nevertheless, some errors may not be captured by error detection mechanisms upon occurrence and then spread as a result of the subsequent flow of information. Thus, the damage caused by an error will propagate until it is detected and handled appropriately. See Figure 2.1 for a typical error detection process. The delay between the occurrence of an error and the moment of its detection, called *error latency*, is important to damage assessment, error recovery, and



Duration A: A fault exists and active in the system

Duration B: The fault becomes inactive

Duration C: Errors exist in the system

Figure 2.1 The Error Detection Process.

establishing confidence in the computation results.

The error latency has been defined by Courtois as ***detection time*** [22,23] and by Shedletsky as ***latency difference*** [63]. Courtois also presented results of on-line tests of the M6800 microprocessor that included the distributions of detection time for certain detection mechanisms. Shedletsky proposed a technique to evaluate the error latency based on the "fault set" philosophy and the probability distribution of input signals. Bavuso *et al.* investigated the problem of the latent fault and proposed experiments to measure the time interval between the moments of fault injection and error detection [8]. Their study indicated that a significant proportion of faults is not detected even after many iterations of a process.

When error latency is significant, there is the possibility of the system putting out incorrect computation results, since there may be some undetected errors at the output phase. Also, even if the system detects all errors before the output phase, the computation achieved during the latent period may already have been contaminated and thus be useless. In practice, error latency is never zero, and in the event of an error the whole system is delayed by the more complicated recovery that is required to remove the contamination that is spread during error latency.

To evaluate these two effects -- the probability of producing an unreliable result and the computation loss resulting from error -- it is necessary to examine the error detection mechanisms incorporated in computer systems and their respective capabilities. One may then establish a different recovery strategy for the errors captured by each distinct detection mechanism, thus obtaining the most appropriate possible recovery performance and execution cost. To evaluate error-handling capability including tradeoffs between various detection mechanisms and recovery methods, it is necessary to consider recovery performance and execution cost, taken as a whole.

In this chapter, a model is proposed to describe error detection process. In the following section, the classification, properties, and associated recovery methods of error detection mechanisms are discussed. The detection model is then developed.

It is assumed throughout this report that faults in hardware components are a potential cause of transition to erroneous states during normal operation. An *error* is defined to be the erroneous information/data resulting from fault(s).

2.1. Classification of Error Detection Mechanisms

There are various error detection mechanisms which can be incorporated in a computer system. The basic principle of these mechanisms is the use of redundancy in devices, information, or time. Based on (i) where they are employed, (ii) their respective recovery methods, and (iii) performance measures, error detection mechanisms are divided into the following three categories.

1. Signal level detection mechanisms

Usually, the mechanisms in this category are implemented by built-in self-checking circuits. Whenever an error is generated by a predescribed fault, these circuits detect the malfunction immediately even if the erroneous signal does not have any logical meaning. Typical methods in this category include error detection codes, duplicated complementary circuits, matchers, etc. The performance of these detection mechanisms is measured by the *coverage,* denoted by c, which is the probability of detecting an error induced by an arbitrary fault. It is difficult to have a perfect coverage because (i) it is prohibitively expensive to design detection mechanisms which cover all types of faults, and (ii) physical dependence between function units and detection mechanisms cannot be completely eliminated.

Since this class of detection mechanisms detects an error immediately upon occurrence, there is no contamination through error propagation. This makes the subsequent recovery operations simple and effective. Two kinds of recovery methods are suitable for this category; one is error masking, in which redundant information is used to retain correctness, the other is retry, in which the previous action is re-executed.

2. Function level detection mechanisms

The detection mechanisms in this category are intended to check out unacceptable activities or information at a higher level than the previous category. Unlike the signal level detection mechanisms, they verify system operations by functional assertions on response time, working area, provable computation results, etc. These detection mechanisms can be regarded as "barriers" or "guardians" around normal operations. After an error is generated by a fault, the resulting abnormality may grow very quickly--the "snowball effect" [22], or "error rate phenomenon" [56]--until it hits the barriers. Several software and hardware techniques such as capability checking, acceptance testing, invalid op-code checking, timeout, and the like can be applied.

The important issues for function level detection mechanisms are error isolation and damage assessment. Both issues depend upon system structure as well as on inherent properties of the executed programs or tasks. When there are clear cleavages between sub-systems or sub-tasks, the effective detection assertions can be easily declared, thus permitting greater error isolation and reducing contamination. Usually, rollback and restart recovery methods are used to rescue failed processes. Rollback requires state restoration such that part of the process can be resumed. The restart method purges the old computation and then re-issues the same task to other non-faulty processors.

3. Periodic diagnostics

This method is usually referred to as off-line testing because the processing unit under test cannot perform any useful task. It is composed of a diagnostic program which supplies inputs such that all existing faults are activated and thus generate errors. Several theoretical approaches and simulations have been proposed to determine the probability of finding an error after applying diagnostics for a certain duration (equivalent to the probability of detecting fault as a function of test duration) [11,29,62,75]. All these results have indicated that the effectiveness of the present category is a monotonically increasing function of testing time. Since the time required for complete testing(i.e. ensuring 100% coverage) is in general too long, an appropriate policy of diagnostics is to perform an imperfect test periodically during normal operation and perform a thorough diagnostics when the system is idle.

2.2. The Model of Error Detection Process

For analytical convenience, occurrence of faults is usually modeled as a Poisson process. Let *MTBF* be the mean time between two successive fault occurrences. Also, let F_i and p_i i=1,2,3 denote the event and the probability that the fault is transient, intermittent, or permanent, respectively. Naturally, $p_1+p_2+p_3=1$. When the classification of faults into these three types is independent of occurrence of fault, occurrence of event F_i can be modeled as a Poisson process with rate $p_i/MTBF$. Then, the following model can be used for a separate analysis of the effects of each type of faults.

2.2.1. Model Development

Figure 2.2 shows our model of the error detection process. The model consists of three parts: the occurrence of a fault, the consequent generation of an error, and the detection of that error. Since the probability of having multiple faults at any time is small, they are excluded from the model. There are six states in the model as follows:

(1). **NF** (non-faulty): In this state no fault exists in the system.

(2). **F** (faulty): There is a fault which is active and capable of inducing errors, but there are no errors.

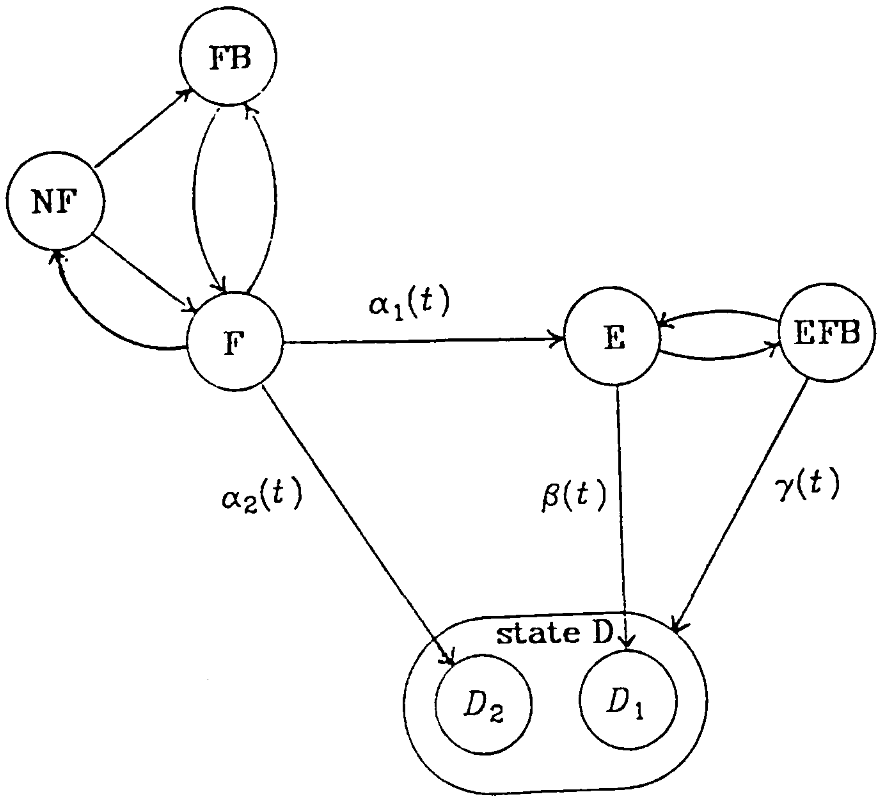
(3). **FB** (fault-benign): There is an inactive intermittent fault.

(4). **E** (error): There is at least one undetected error in the system and the fault which has caused that error is still present.

(5). **EFB** (error-fault-benign): At this state the intermittent fault has become inactive or the transient fault has disappeared after it induced an error.

(6). **D** (detection): At this state, the detection mechanisms have identified the error in the system. To distinguish between whether the system has been contaminated or not, two substates, called $\mathbf{D_1}$ and $\mathbf{D_2}$, are included. The system will enter $\mathbf{D_1}$ when the detected error has contaminated at least part of the system. On the other hand, the system enters $\mathbf{D_2}$ when an error is detected before it begins to propagate through the system. Signal level detection and diagnostics cause transitions from **F** to $\mathbf{D_2}$. In fact, these transitions can be divided into two steps: an existing fault induces an error, and the error is detected immediately following its occurrence.

Let λ denote the rate of occurrence of F_i type faults, i.e., $\lambda = p_i / MTBF$ i=1,2,3 when transient, intermittent and permanent faults are separately considered. Since



Note: The transitions between **NF**, **F**, **FB**, and **E**, **EFB** are dependent on the type of fault.

Figure 2.2 The Model for Error Detection Process.

intermittent faults may become inactive, a benign state has to be included in the model. Several models of intermittent faults have been proposed and used for testing and reliability evaluation [11,48,55,71,77]. In our model, the transitions between **NF**, **F**, **FB**, and between **E**, **EFB** are used to describe the behavior of intermittent faults. For transient and permanent faults, **FB** does not exist, implying that the transition rates between **F** and **FB**, μ and ν, are zero. Similarly, for intermittent and permanent faults the rate of transition from **F** to **NF**, τ, equals zero.

Consider the process of generating errors by a given fault. With the assumption that the signal patterns of successive inputs are independent, Shedletsky treated the period of fault latency as a random variable with a composite geometric distribution for discrete inputs or cycles [62]. Using the concepts of information theory, Agrawal presented a formula to estimate the probability of inducing error [1]. For tractability we have assumed in our model an exponentially distributed fault latency with rate α when a task is executing. While the diagnostic program is running, the transition duration from **F** to $\mathbf{D}_2$ is assumed to be exponentially distributed with parameter ω. If the diagnostic program is executed for period t_p following a normal operation period t_n and a process swapping period t_v as shown in Figure 2.3, the coverage of a single diagnostic, denoted by ξ, is equal to $1-e^{-\omega t_p}$ for each execution of diagnostics.

Once the system enters **E**, the erroneous information starts to spread until function level detection mechanisms identify any unacceptable result. There are two paths to $\mathbf{D}_1$ and they represent transition rates of $\beta(t)$ and $\gamma(t)$, respectively. At state **E**, since the fault still exists, it is possible that the fault is captured by signal level detection mechanisms or diagnostics *prior to* the function level error detection. We exclude this case from

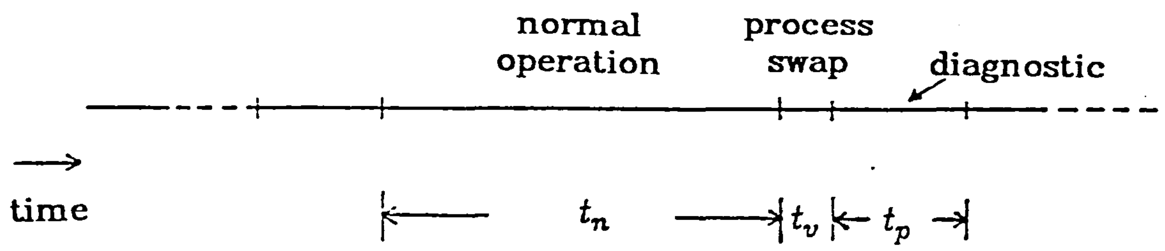


Figure 2.3 A Cycle of Periodic Diagnostics.

the model because the process has already become erroneous, and the subsequent signal level detection has no effect on this error. (Namely, a direct transition from **E** to **D**$_2$ is not included.) It is also possible that there are multiple errors induced by the same fault or by an old, undetected error when the system is in **E** or **EFB**. The function level detection mechanisms will recognize that the system is erroneous regardless of which error is captured. However, the error latency must be measured from the moment that the first error occurs.

2.2.2. Mathematical Description of State Transitions

Let a computer system incorporate the three types of error detection mechanisms discussed above. For notational convenience number the states **NF, F, FB, E, EFB, D**$_1$, **D**$_2$ with i for i=1,2,...,7. Then one can obtain a transition probability matrix $\mathbf{H}_{7\times7}(t)$ by making use of the model in Figure 2.2.

$$H_{7\times7}(t) = \begin{bmatrix} -\lambda & \frac{\lambda\nu}{\mu+\nu} & \frac{\lambda\mu}{\mu+\nu} & 0 & 0 & 0 & 0 \\ \tau & -(\mu+\tau+\alpha_1(t)+\alpha_2(t)) & \mu & \alpha_1(t) & 0 & 0 & \alpha_2(t) \\ 0 & \nu & -\nu & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -(\mu+\tau+\beta(t)) & \mu+\tau & \beta(t) & 0 \\ 0 & 0 & 0 & \nu & -(\nu+\gamma(t)) & \gamma(t) & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad (2.1)$$

Since the diagnostic is invoked periodically, transition rates $\alpha_1(t)$, $\alpha_2(t)$, $\beta(t)$, and $\gamma(t)$ are the following functions of time.

$$\alpha_1(t) = \begin{cases} (1-c)\alpha & \text{if } n(t_n+t_p+t_v) < t \le n(t_n+t_p+t_v)+t_n \\ 0 & \textit{otherwise} \end{cases} \tag{2.2a}$$

$$\alpha_2(t) = \begin{cases} c\alpha & \text{if } n(t_n+t_p+t_v) < t \le n(t_n+t_p+t_v)+t_n \\ \omega & \textit{otherwise} \end{cases} \tag{2.2b}$$

$$\beta(t) = \begin{cases} \beta & \text{if } n(t_n+t_p+t_v) < t \le n(t_n+t_p+t_v)+t_n \\ 0 & \textit{otherwise} \end{cases} \tag{2.2c}$$

$$\gamma(t) = \begin{cases} \gamma & \text{if } n(t_n+t_p+t_v) < t \le n(t_n+t_p+t_v)+t_n \\ 0 & \textit{otherwise} \end{cases} \tag{2.2d}$$

where c is the coverage of the signal level detection; α is the transition rate that a fault generates an error; β and γ represent the transition rates that the function level detection captures errors in states **E** and **EFB**, respectively; and n is a positive integer.

Hence the state probabilities, $\pi(t)=[\pi_1(t),\pi_2(t),...,\pi_7(t)]$, can be obtained by solving the following differential equation:

$$\frac{d\pi(t)}{dt} = \pi(t)\,\mathbf{H}(t); \qquad \pi(0) = \pi_0 \tag{2.3}$$

where $\pi_i(t)$ is the probability that the system is in state i at time t. Because of the absorbing property of $\mathbf{D}_1$ and $\mathbf{D}_2$, one can easily see that $\pi_6(\infty)+\pi_7(\infty)=1$.

Assume the initial state that the system begins with is **NF**. When a transient or a permanent fault occurs, the system will enter the non-faulty state again after either the fault disappears or the system is reconfigured to eliminate the source of the fault.

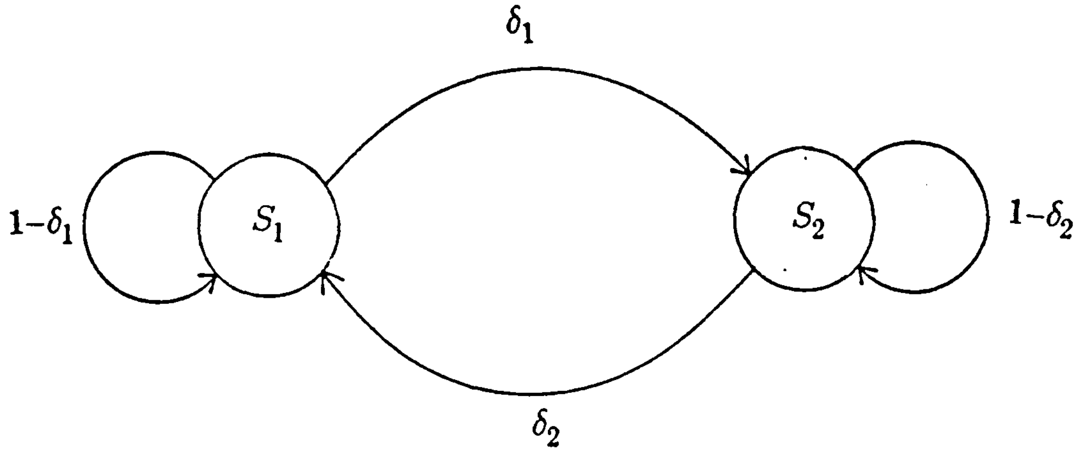
In case of an intermittent fault, it is possible for the system to be in **FB** instead of **NF** even after some recovery procedures are successfully applied. For example, when the fault becomes benign during the retry recovery, the system enters **FB**. Let S_1 (or S_2) be the event that the system is in state **NF** (or **FB**) after recovery from an intermittent fault. This process can be represented by a Markov chain shown in Figure 2.4 and the transition probabilities between S_1 and S_2, denoted by δ_1 and δ_2. These transition probabilities are computed using Eq.(2.3) and the corresponding recovery performance will be discussed in the recovery strategy of next chapter. Note that, under S_2, the same intermittent fault will be detected by the signal level detection with probability one if it induces an error again.

A task may start execution when the system is in any one of **NF**, **F**, **FB**, **E**, **EFB** (but certainly not in $\mathbf{D}_1$, $\mathbf{D}_2$,). Using the Markov model in [18], we can calculate (i) the mean number of visits to state i, $i=1,2,..,5$, before the system is absorbed into $\mathbf{D}_1$ or $\mathbf{D}_2$ for every F_j $j=1,2,3$, and (ii) the mean time interval, $E[X_i|F_j]$ $j=1,2,3$, during which the system stays in state i before transition to $\mathbf{D}_1$ or $\mathbf{D}_2$ takes place. Then, the probability that a task begins execution when the system is in state i, is formulated as follows:

$$\pi_i(0|F_j) = \begin{cases} E[X_i|F_j] \,/\, \sum_{k=1}^{5} E[X_k|F_j] & \text{for } i=1,2,..,5 \\ 0 & \text{for } i=6,7 \end{cases} \tag{2.4}$$

It may be possible that the active duration of an intermittent fault increases every time it becomes active following its first occurrence. This would imply that the transition rates between fault active and fault benign depend on the duration for which an intermittent fault exists. In such a case, the model suggested in [49] can be used in the above system equations.

It cannot be over-emphasized that our modeling of the error detection process is intended to evaluate the effects of various detection mechanisms on task execution. This fact is in sharp contrast to most conventional methods in which models have been developed and then used to estimate the system reliability or to determine the coverage of failure. For example, in CARE III [71,77] the error propagation rate is defined by the



$$\begin{tikzcd}
S_1 \arrow[loop left, "1-\delta_1"] \arrow[r, bend left, "\delta_1"] & S_2 \arrow[loop right, "1-\delta_2"] \arrow[l, bend left, "\delta_2"]
\end{tikzcd}$$

Figure 2.4 A Markov Chain for the Recovery from an Intermittent Fault.

user and the model is applied to determine the coverage. Note that a transition to $\mathbf{D}_1$ represents the detection of error by function level detection mechanisms, whereas $\mathbf{D}_2$ is reachable directly from **F** by signal level detection mechanisms or diagnostics. The impacts of detection mechanisms on task execution will be reflected through Eq. (2.3) and the state distribution $\pi(t|F_j)$.

3. Analysis of the Impacts of Error Detection Mechanisms

In case of imperfect coverage (i.e. $c<1.0$) in the signal level detection and non-zero error latency in the function level detection, the system will suffer from the following two undesirable effects; one is the possibility of putting out potentially erroneous results and error propagations because the system is unaware of the existence of error, and the other is the additional recovery overhead resulting from error propagation through the system during error latency. With the model proposed in the previous section and moderate assumptions regarding error recovery, we will in this section analyze these two effects and then use them to specify the requirements for design of error detection. The error recovery strategy is presented here for studying computation loss. However, detailed discussions concerning rollback recovery method are given in Chapter 4 and 5.

3.1. Estimation of the Probability of Producing an Unreliable Result

The execution of a task consists of parallel and/or serial execution of processes. We can always partition the task into processes in such a way that every process receives all the input data at the beginning of its execution and sends the computation result to its successors at the end of execution. A serious situation, namely the propagation of erroneous information through the system, appears if an error occurs and cannot be discovered before the end of execution. For convenience, let us define an *unreliable result* as follows:

Definition: If there exists at least one error at the moment of process completion and if the system is at that moment still unaware of the presence of that error, the process is said to end with an *unreliable result.*

An unreliable result may even include the cases of producing wrong and/or no outputs. On the other hand, it may yield a correct output despite the presence of error if the computation is not contaminated by the error. However, the result cannot be trusted owing to the presence of an undetected error at the moment of output. (No one would have much confidence in the computation result under this circumstance!) It is therefore important to estimate the probability of producing an unreliable result, denoted by p_e, as a measure of *lack of confidence* in the computation result.

Let T denote the execution time of a process. If T is deterministic, p_e is given by $p_e = \sum_{j=1}^{3} p_j \{\pi_4(T|F_j) + \pi_5(T|F_j)\}$, which is the probability that the system is in **E** or **EFB** at the moment of process completion. When T is a random variable with density function $f_T(t)$, then p_e becomes $p_e = \int_0^\infty \{\sum_{j=1}^{3} p_j [\pi_4(t|F_j) + \pi_5(t|F_j)]\} f_T(t)\, dt.$

When a diagnostic is scheduled periodically for the process, the resulting p_e becomes a function of the time interval between the output moment and the time the previous diagnostic has run. The shorter this time interval, the more reliable the computation result. However, because of the uncertainty of the process execution time, it is difficult to schedule periodic diagnostics so that the system is tested just before the process moves into the output phase. Here, using the proposed model, we can compute the maximum value of p_e, denoted by $max(p_e)$, which occurs when the time interval between the process completion and the last diagnostic is equal to t_n. Observe that $1-max(p_e)$ represents the lower bound of confidence (or *sure confidence)* in the computation results and thus can be used for design specifications. Some simulation results are graphed in Figures 3.1 and 3.2. In Figure 3.1, $max(p_e)$ starts to decrease sharply only when each diagnostic has a higher coverage ($\xi \geq 0.95$). In Figure 3.2, we compare three different cases: (i) with periodic diagnostics and $c=0.6$, (ii) with periodic diagnostics and $c=0.8$, and (iii) with periodic diagnostics, $c=0.6$, and doubled function level detection rates. From the model, we can observe that $max(p_e)$ is linearly related to the coverage of the signal level detection and varies exponentially with respect to the function level detection capability. However, perfect coverage and zero error latency are impossible to attain in practice. Thus, the combination of both the signal level and the function level detection mechanisms have to be used to reduce p_e.

3.2. Evaluation of Computation Loss and Execution Cost

The designer needs to evaluate the expected computer performance if the properties of detection mechanisms such as coverage and error latency are known. To this end, we have considered here two important parameters to represent the impact of these detection mechanisms on computer performance; *computation loss* and *execution cost.* Computation loss -- a system oriented view -- is represented by the amount of time used for error handling, whereas execution cost -- a task-oriented view -- shows the effect of error detection and recovery on a particular task in the event that an error is detected during its execution. After the detection of an error, one may use one of several recovery methods to rescue the executing process. Recovery strategies usually depend on the detection mechanisms and the fault/error types.

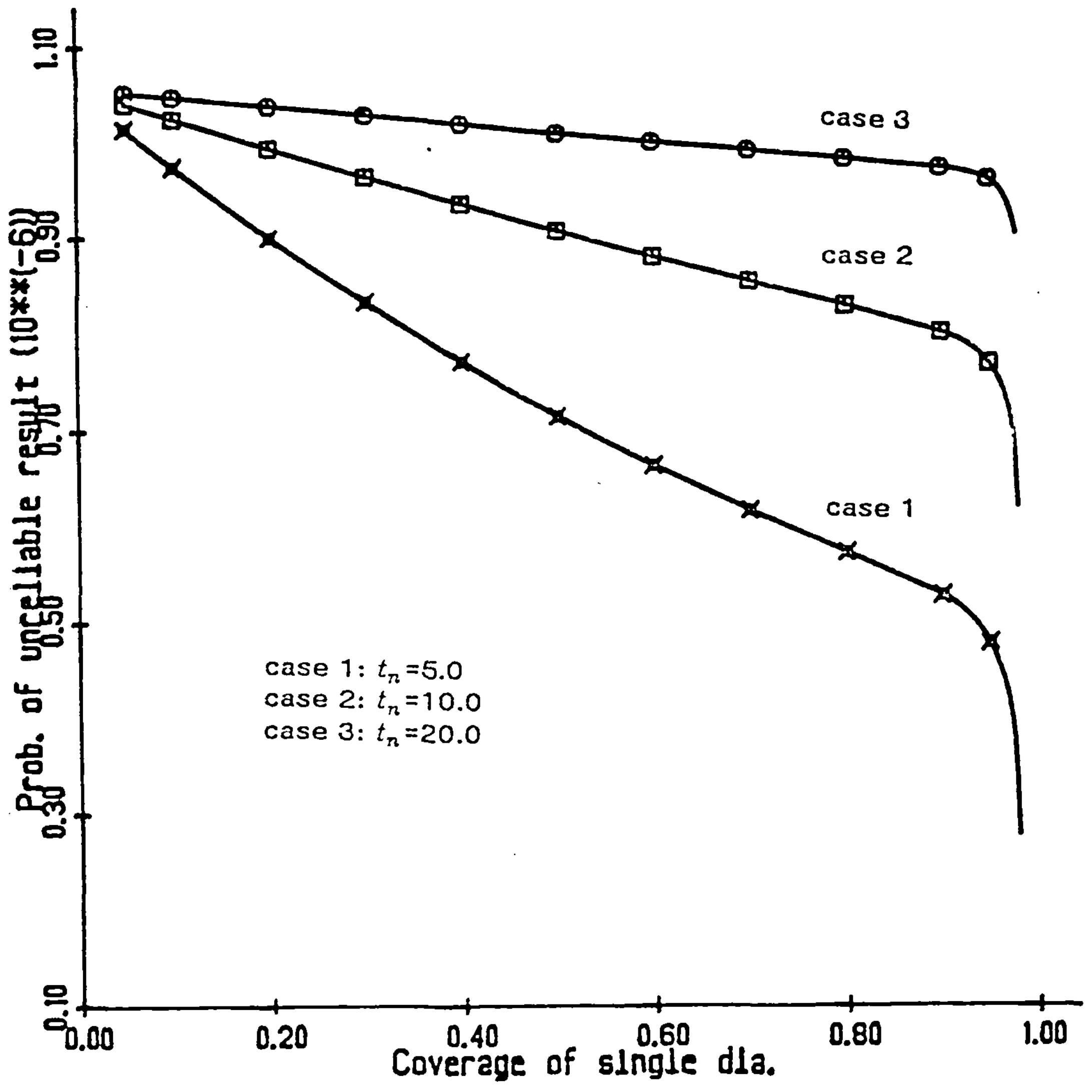


Figure 3.1 Max(p_e) versus the Coverage of Single Diagnostic, ξ.
($\lambda=10^{-6}$,$\mu=0.2$,$\nu=0.1$,$\tau=0.2$,$\alpha=0.2$,$\beta=0.5$,$\gamma=0.1$,$\omega=20.0$,$c=0.6$,$T=100.0$).

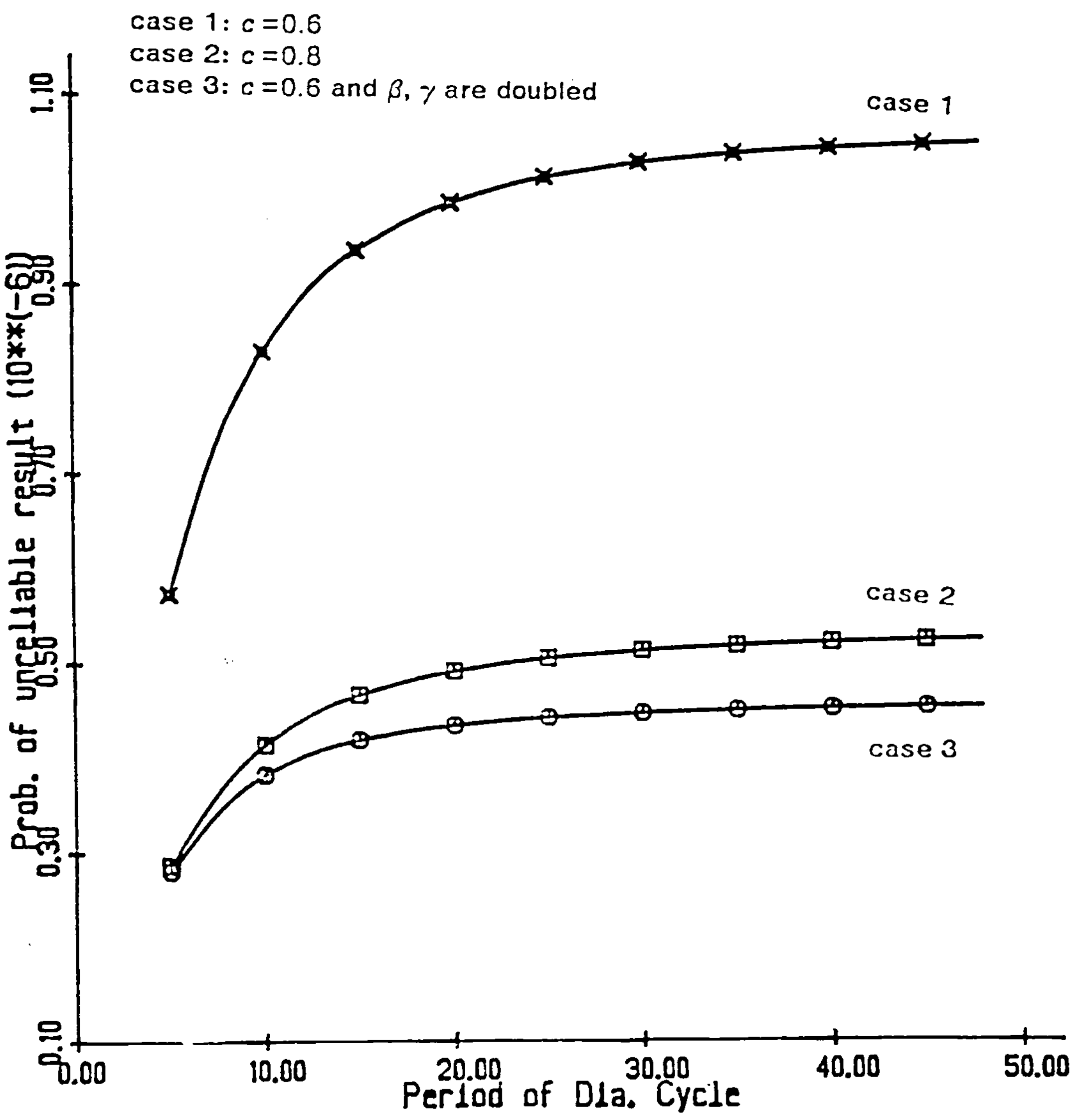


Figure 3.2 Max(p_c) versus t_n.
($\lambda = 10^{-6}, \mu = 0.2, \nu = 0.1, \tau = 0.2, \alpha = 0.2, \beta = 0.5, \gamma = 0.1, \omega = 20.0, \xi = 0.8, T = 100.0$).

The overhead and efficiency associated with these recovery methods are briefly discussed in the sequel. This discussion is not intended to present the details of error recovery. More detailed accounts of rollback and restart recoveries are given in Chapter 4 and 5.

3.2.1. Recovery Strategies and Their Respective Overheads

If an error is detected by some detection mechanism, rollback or restart can always be applied to recover the process from the error. It is, however, possible to use masking or retry if the error is captured by signal level detection mechanisms. Figure 3.3 illustrates four recovery strategies, their applications and their application precedence when multiple strategies are used to recover from a single error. In Figure 3.4, a probabilistic flow diagram between these recovery methods is presented.

Note that a transient fault may not induce any error before its disappearance. The probability of having an error, given the occurrence of fault, is $P(E)=\frac{\alpha p_1}{(\alpha+\tau)}+p_2+p_3$. Let $R_{i,j}$ and $\rho_{i,j}$ represent respectively the mean overhead and the probability that the i-th recovery method is applied to recover from an error which is generated by F_j, where i=1,2,3,4 for masking, retry, rollback and restart, respectively. We also define $\theta_{i,j}$ as the conditional probability that the process is recovered, given that the i-th method is used when F_j occurs. Let p_j' be the probability of F_j given an error is detected, which are listed in Table 3.1. We can use Figure 3.4 to represent the mean total overhead of recovery $RT=\sum_{j=1}^{3} p_j' \left(\sum_{i=1}^{4} \rho_{i,j} R_{i,j}\right)$ for every error detection.

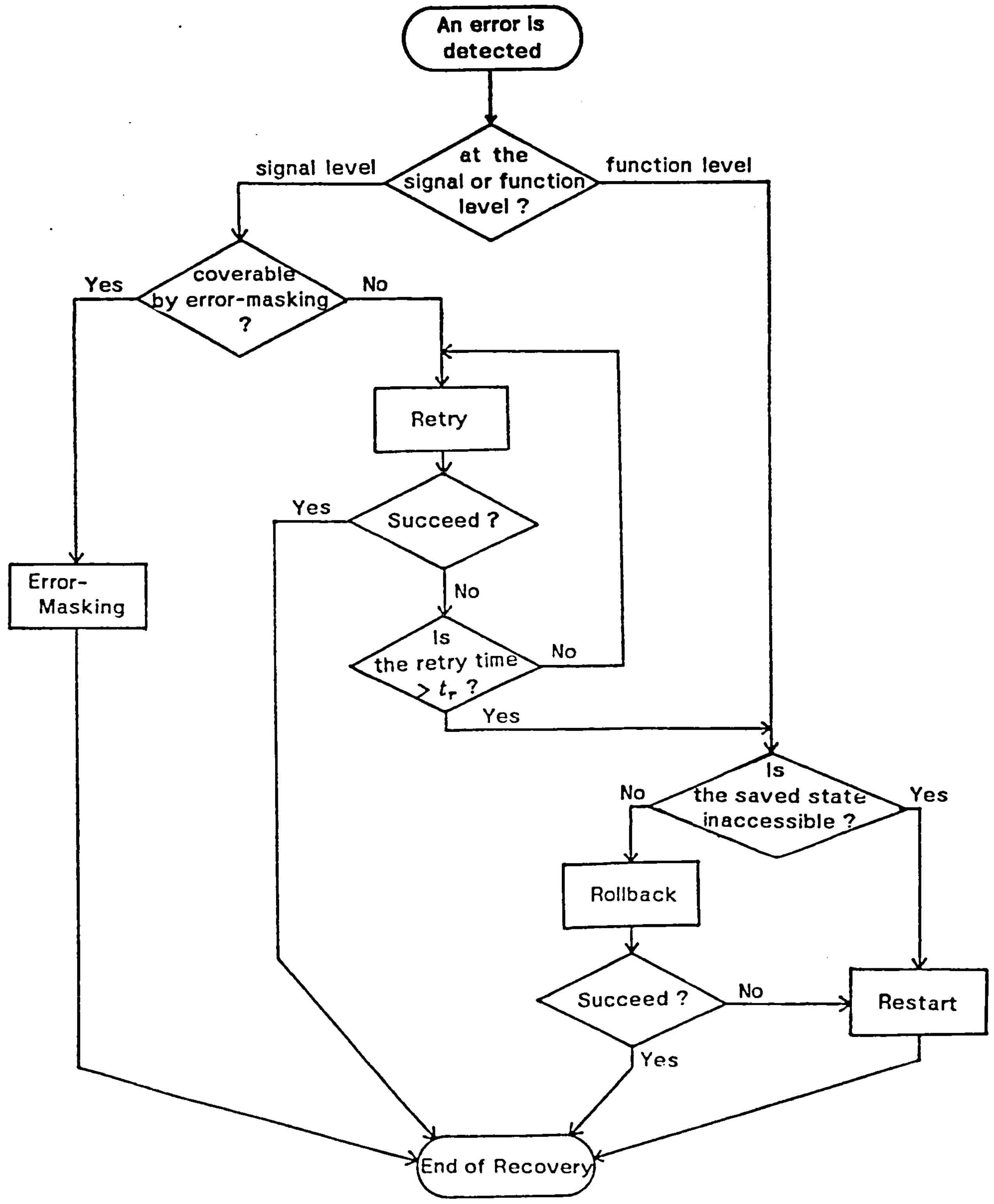
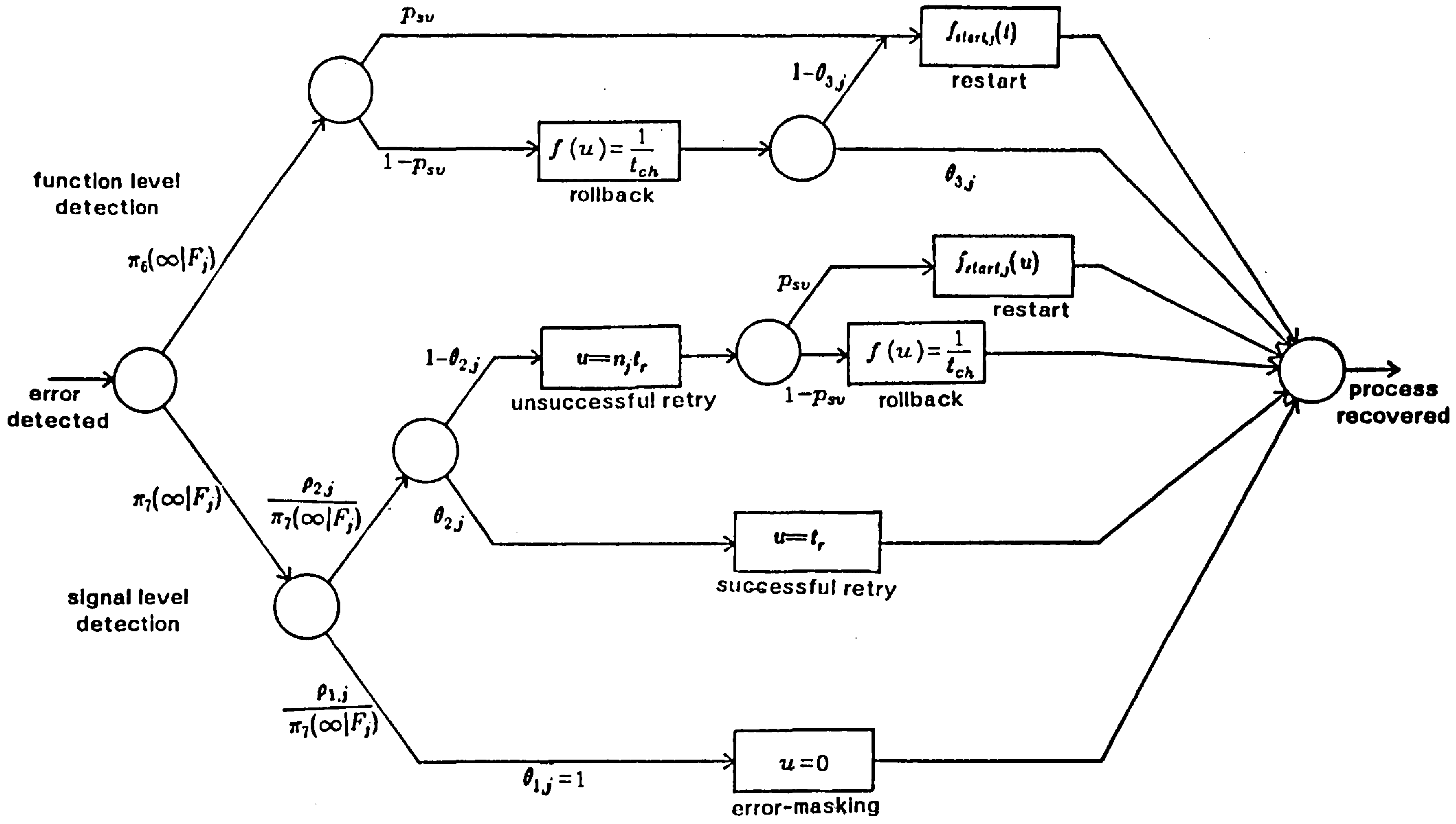


Figure 3.3 **The Flow Chart of Recovery Processes.**



u: recovery delay
$f(u)$: density function of u
$n_1=n_3=1$, and $\text{Prob}(n_2=n)=\delta_2(1-\delta_2)^{n-1}$

Figure 3.4 Probabilistic Flow Diagram of Recovery Processes.

	F_1 (transient fault)	F_2 (intermittent fault)	F_3 (permanent fault)
p_j'	$\frac{p_1 PE_1}{P(E)}$	$\frac{p_2}{P(E)}$	$\frac{p_3}{P(E)}$
$\rho_{2,j}$	$(1-\rho_1)\frac{\pi_7(\infty\|F_1)}{PE_1}$	$(1-\rho_1)\pi_7(\infty\|F_2)$	$(1-\rho_1)\pi_7(\infty\|F_3)$
$\theta_{2,j}$	$1-e^{-rt_r}$	0	0
$R_{2,j}$	t_r	$\frac{t_r}{\delta_2}$	t_r
$\rho_{3,j}$	$(1-p_{sv})\{\frac{\pi_6(\infty\|F_1)}{PE_1}+\rho_{2,1}(1-\theta_{2,1})\}$	$(1-p_{sv})\{\pi_6(\infty\|F_2)+\rho_{2,2}(1-\theta_{2,2})\}$	$(1-p_{sv})\{\pi_6(\infty\|F_3)+\rho_{2,3}(1-\theta_{2,3})\}$
$\theta_{3,j}$	$1-\frac{\pi_6(\infty\|F_1)}{PE_1}D_1$	$1-\pi_6(\infty\|F_2)D_2$	$1-\pi_6(\infty\|F_3)D_3$
$R_{3,j}$	$t_b+\frac{t_{ch}}{2}$	$t_b+\frac{t_{ch}}{2}$	$t_b+\frac{t_{ch}}{2}$

where $D_j = \int_0^{t_{ch}} \frac{1-p_{46}(t|F_j)}{t_{ch}}\,dt - \int_{t_{ch}}^{\infty} p_{46}(t|F_j)\,dt$, and $PE_1=\pi_6(\infty|F_1)+\pi_7(\infty|F_1)$.

Table 3.1 Performance Expressions of Several Parameters in Error Recovery.

1. Error Masking

Most error masking methods employ error-correcting code in data transfer, memory, and arithmetic units. Error masking is the most efficient recovery method when it can be applied successfully. In fact, we can regard in this case that the error has never occurred since the system still provides correct results despite the existence of error. Thus, one can assume $R_{1,j}=0$, i.e., zero recovery overhead, and $\theta_{1,j}=1$. The probability that error-masking is used, $\rho_{1,1}=\rho_1\pi_7(\infty|F_1)(\alpha/(\alpha+\tau))$ and $\rho_{1,j}=\rho_1\pi_7(\infty|F_j)$ for all j=2,3, depends on the conditional probability that error occurs due to the faults in the units with error-correcting code and can be corrected by the error-correcting code, given that the error occurs.

2. Retry Recovery

Retry can be attempted at various levels, e.g., at the levels of micro-instruction, instruction, or I/O operations. Retry is useful when error has not propagated yet at the time of detection. Re-executions of the same operation can produce a correct result only if the related fault is transient or intermittent and disappears during retry. Ideally, the system should apply retry recovery until the fault disappears if it is transient with a short active duration. For permanent faults, retry recovery is not helpful. However, after the detection of error by signal level detection mechanisms, it is very difficult, if not impossible, to tell the type of fault. Moreover, if it is transient, it is impossible to predict when the fault will disappear.

Due to above reasons, assume the system will retry automatically for a fixed duration t_r upon detection of an error by the signal level detection. Then, we can obtain mathematical expressions of $\rho_{2,j}$, $\theta_{2,j}$, and $R_{2,j}$ for j=1,2,3 as listed in Table 3.1. Note that the second subscript j represents here the fault type j.

Recall that if an error generated by an intermittent fault is recovered successfully by retry, the same fault will be detected again by the signal level detection when it becomes active and induces error again. Thus, there are $1/\delta_2$ retries on the average among which application of the last retry will be unsuccessful. In case of intermittent faults the transition probabilities, δ_1, δ_2, between S_1 and S_2 are expressed as follows: (See Section B for the definition of S_1 and S_2.)

$$\delta_1 = \pi_7(\infty|F_2)\,(1-\rho_1)\,(1-e^{-\mu t_r}) \tag{3.1}$$

$$\delta_2 = e^{-\mu t_r} \tag{3.2}$$

From Eqs. (3.1) and (3.2), it is easy to see that though it is simple and practical, the above retry method is not intelligent. It may be more desirable to design a retry mechanism which can recognize the intermittent nature of the fault following several consecutive, successful retries for the same fault. Since retry mechanism observes the active duration of a fault and the detection mechanism gives the duration of fault occurrence, it is possible to measure the fault characteristics. However, the information obtained is not complete because the failure of retry. It should be attractive to find a combined retry and estimation methodology.

3. Rollback Recovery

Rollback recovery can be regarded as a type of retry which needs to save process states during normal operation. When an error is detected, the process rolls back to one of the previously saved states. The original idea of rollback recovery is accommodated with acceptance tests for software reliability [35,56]. Here, for rollback recovery we assume periodic insertion of checkpoints such that the process can be resumed at any one of these checkpoints. Let t_{ov} and t_{ch} be respectively the overhead for saving states and the interval between two adjacent checkpoints. Then, the percentage of the overhead for establishing checkpoints is $t_{ov} / (t_{ov}+t_{ch})$. Note that rollback recovery fails if the states saved are destroyed by a fault, or if the states are contaminated by error (e.g. due to the presence of error during the state saving).

The time lost in rollback recovery is the sum of the computation undone and the setup time[1] for rollback, t_b. When we consider the re-occurrence of error during recovery, it is extremely difficult to determine this time loss. However, when the fault occurrence rate is very small (typically 10^{-6} per second for the IC's manufactured today), we can assume no error occurrence during rollback. We also assume that only the most recently saved state is kept in order to minimize the storage requirements for checkpoints. Then, the time loss in computation simply becomes the interval between the moment of the last state saving and that of the error occurrence which cannot be recovered by error-masking or retry. Since the *MTBF* is in general much greater than the inter-checkpoint interval t_{ch}, one can assume that the occurrence of rollback recovery is uniformly distributed within the inter-checkpoint interval, given that it is applied. Let p_{sv} be the probability that the saved state becomes inaccessible or unusable and $p_{46}(t|F_j)$ be the probability distribution function of error latency for fault type F_j, i.e. the probability that the system is in $\mathbf{D}_1$ at time t when the system starts from $\mathbf{E}$. $p_{46}(t|F_j)$ is equal to $\pi_6(t|F_j)$ in Eq. (2.3) when $\pi(0)=[0,0,0,1,0,0,0]$. Then, we obtain $\rho_{3,j}$, $\theta_{3,j}$ and $R_{3,j}$ as listed in Table 3.1.

[1]The setup times for both rollback and restart recoveries are needed for hardware reconfiguration and software initialization. The hardware reconfiguration is to eliminate the source of error (i.e. fault(s)) for the resident process in the faulty module.

4. Restart Recovery

When restart recovery is applied, the whole process is re-executed from the beginning to recover from an error. Since the system can be reconfigured to replace the faulty component, restart recovery will eventually succeed as long as there are enough resources to replace faulty components. Hence, we have $\theta_{4,j}=1$ and $\rho_{4,j}=1 - \rho_{1,j} - \rho_{2,j}\theta_{2,j} - \rho_{3,j}\theta_{3,j}$. The time wasted in each restart is the sum of the setup time for reconfiguration and reinitialization, and the time of error detection, T_d, measured from the beginning of process execution. For simplicity, we assume that the moment of restart recovery is uniformly distributed within the task execution period. Thus the density function of the overhead involved in restart, $f_{start,j}(t)$ is equal to $1/T$ for $t_s \leq t \leq T+t_s$, and $R_{4,j} = t_s + T/2$ where t_s is the setup time for restart. Details of the effects on task execution time by successive restarts can be found in [45].

3.2.2. Calculation of Computation Loss and Execution Cost

Now with the preceding overhead analyses, consider the computer time that is used for actual computation instead of error handling. The average computation loss due to a single error detected, denoted by CL, has to include the overheads due to periodic diagnostics, periodic insertion of checkpoints, and recovery in the event of error. Define η as the percentage of the average computation loss for each error detection, which is expressed by:

$$\eta \approx \frac{CL}{1/(\lambda\ P(E))} = \sigma + \lambda P(E) \sum_{j=1}^{3} p_j' \ (\sum_{i=1}^{4} \rho_{i,j} R_{i,j}) \tag{3.3}$$

where $1/(\lambda\ P(E))$ is an approximate mean time between two successive error detections, and σ is the percentage loss due to periodic diagnostics and insertion of checkpoints and is given by:

$$\sigma = \frac{t_p+t_v}{t_n+t_p+t_v} + \frac{t_{ov}}{t_{ov}+t_{ch}}$$

The above equation indicates that the time wasted for executing periodic diagnostics and checkpointing is a dominating factor in the total computation loss when the system is highly reliable (i.e. the system has a small λ). In Figure 3.5, plotted are the simulation results for the percentage of the total computation loss, η, and the mean loss in recovery, RT. The reduction in recovery loss by periodic diagnostics is small because (i) the diagnostic is useful only if it can capture faults before they induce errors, and (ii) the diagnostic is incapable of detecting an intermittent fault when the fault is inactive. (iii) even if the diagnostic identifies a fault, the system still has to reconfigure or retry to eliminate this fault. Detection mechanisms other than on-line diagnostics are more advantageous due to their favorable effects and overheads on the computer performance.

Observe that this time loss is related to the system, not to tasks to be executed on the system. One can therefore regard this as the *system overhead*. On the other hand, tasks executing on the system may suffer from delays in execution due to error detection and recovery overhead which are *task-specific*. When a task is time-critical, the delay in its execution may cause a catastrophe (e.g. loss of human lives, economic and social disaster, etc.) if the execution is not completed within a specified time limit called *hard deadline,* denoted by t_{dead}. This was termed *dynamic failure* in [43,65]. Also, the running cost -- the cost for use of computer as well as controlling an actual system which uses the computed results -- will certainly go up with the increase of the execution delay. In case of error, based on Figure 3.4, we can write the probability density function of the execution delay due to the recovery from an F_j type fault, $f_r(t|F_j,T)$, where T is the

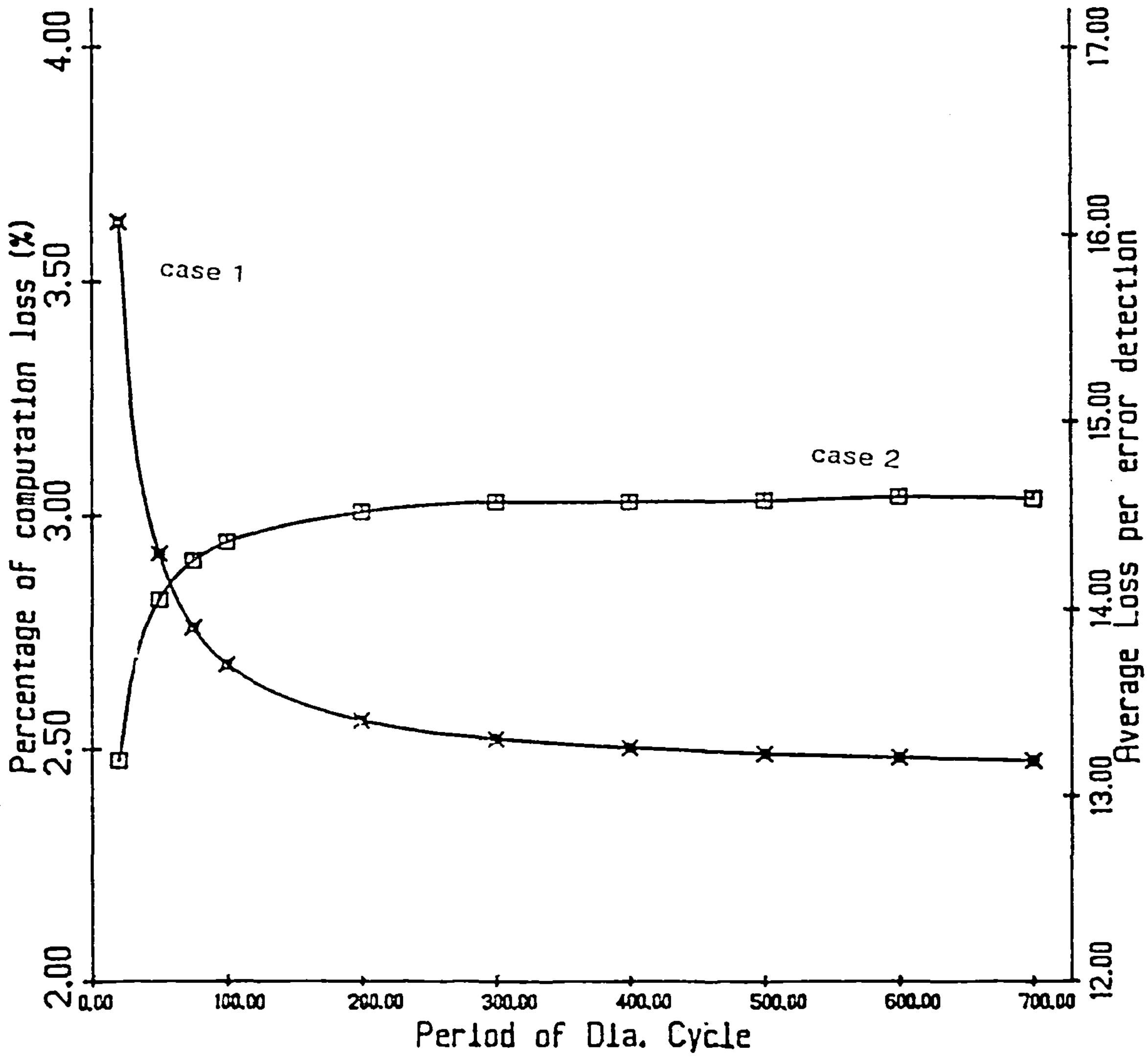


Figure 3.5 The Effects of Periodic Diagnostics on Percentage of Total Loss (case 1), η, and Total Recovery Loss (case 2), RT.
($\lambda=10^{-6}, \mu=0.2, \nu=0.1, \tau=0.2, \alpha=0.2, \beta=0.5, \gamma=0.1, \omega=20.0, \xi=0.8, c=0.6$, $T=100.0$).

needed time for task completion under a fault-free condition. These density functions are listed in the Appendix A. Note that for intermittent faults the task may be completed with successful retries. In the expression for $f_r(t|F_2,T)$ given in the Appendix A, for simplicity we used the upper bound of error handling delay; that is, whenever an error occurs, the task completion is achieved with rollback or restart recovery.

Since the overhead associated with checkpointing and diagnostics has to be included, the time needed for task execution under the fault-free condition becomes $\hat{T}=(1+\sigma)T$. For any computation process, the delay in execution may induce an extra cost. For example, in real-time applications this cost may be the additional energy or fuel used for the controlled system, the consequence of longer response time. etc. Given a cost function for the execution time t, $C(t)$, which is a monotonic non-decreasing function (see [43,65] for an example of its detailed derivation), we can obtain the total execution cost, $COST$, and the probability of dynamic failure, p_{dyn}, as below.

$$COST = \sum_{j=1}^{3} p_j \int_{T}^{\infty} C(t)\, f_r(t|F_j,\hat{T})\, dt \tag{3.4}$$

$$p_{dyn} = \sum_{j=1}^{3} p_j \int_{t_{dead}}^{\infty} f_r(t|F_j,\hat{T})\, dt \tag{3.5}$$

3.3. Design Consideration for Detection Mechanisms

Consider the performance and reliability measures, p_e, p_{dyn}, and $COST$. These measures quantitatively represent the consequences of imperfect detection mechanisms and then reflect the effects of detection mechanisms on the system performance. In this section, these measures are used to address problems in the design of detection mechanisms.

Suppose that the specifications of performance requirements and application tasks are now given. To provide the required fault-tolerance in the design, we have to answer the following two questions: (i) what kinds of detection mechanisms should be incorporated in the computer system to be designed?, and (ii) what are their properties in meeting the specifications? In other words, we need to know the coverage by signal level mechanisms, the error latency in function level mechanisms, and the period of diagnostics. Suppose for instance that the real-time operations and time-critical processes are now our major design concern. The specifications must include the limit for the probability of failure as well as the maximum allowable extra cost caused by shortcomings of detection mechanisms.

According to our simulation results in Figure 3.5, the avoidance of error by diagnostics appears useful only if the cycle time of diagnostics is not much greater than the fault's active period, which is usually small for transient and intermittent faults. This implies that a frequent application of diagnostics is needed. However, in such a case, the computation time wasted for executing diagnostics as well as the total execution cost increases prohibitively, making the periodic use of diagnostics during normal operation less useful. It also indicates that the probability of capturing intermittent faults and the improvement of loss in recovery by diagnostics are small. Consequently, on-line diagnostics are not useful for time-critical applications.

As a conservative measure, the probability of failure due to imperfect detection mechanisms, denoted by p_f, can be represented by the sum of p_e and p_{dyn}. From the model, one can see that p_e is dependent exponentially on error latency but linearly on coverage, c. That is, the decreasing of error latency has a greater impact on p_e than does the increasing of the coverage. However, an improvement in the coverage will decrease the probability of error propagation and thus reduce the recovery overhead. In Figure 3.6, curves with constant p_f and constant $COST$ are plotted, where $C(t)$ is assumed to be $(t-\hat{T})^2$ for $t \geq \hat{T}$. It shows the combination of the coverage and the mean error latency required to attain p_f and $COST$, below the specified values. The area under both the constant p_f and constant $COST$ lines indicates the design space for selecting the coverage and the mean error latency.[2] It is clearly that perfect signal level detection is within the design space, though it is impractical. By contrast, the combination of small error latency and zero signal level detection may not satisfy the specifications. This can be seen easily from the fact that with a zero signal level detection, every recovery must require rollbacks and/or restarts. The use of rollbacks and/or restarts for recovery is more time-consuming than error-masking and retry which are available only to signal level detection mechanisms. Hence, signal level detection mechanisms must be included in the design. The curves with constant $COST$ show that the average execution cost is insensitive with respect to the coverage of the signal level detection mechanism. This is due to the fact that all errors induced by intermittent or permanent faults have to be recovered by rollback or restart whatever the nature of the error detection process, and that because of the overheads imposed on saving states, recovery points have to be placed relatively far apart. It is important to recognize that an effective recovery method will have severe impact on the delay of task execution.

The feasible design space indicated in Figure 3.6 will provide the requirements in detection mechanisms for certain system performance specifications. However, it is very difficult to objectively determine an optimal combination of signal level and function level detection mechanisms. The main reason for this is that (i) the coverage has to be

[2] The mean error latency is equal to the mean time needed from state $\mathbf{E}$ to state $\mathbf{D}_1$ and will reflect the capability of function level detection mechanisms.

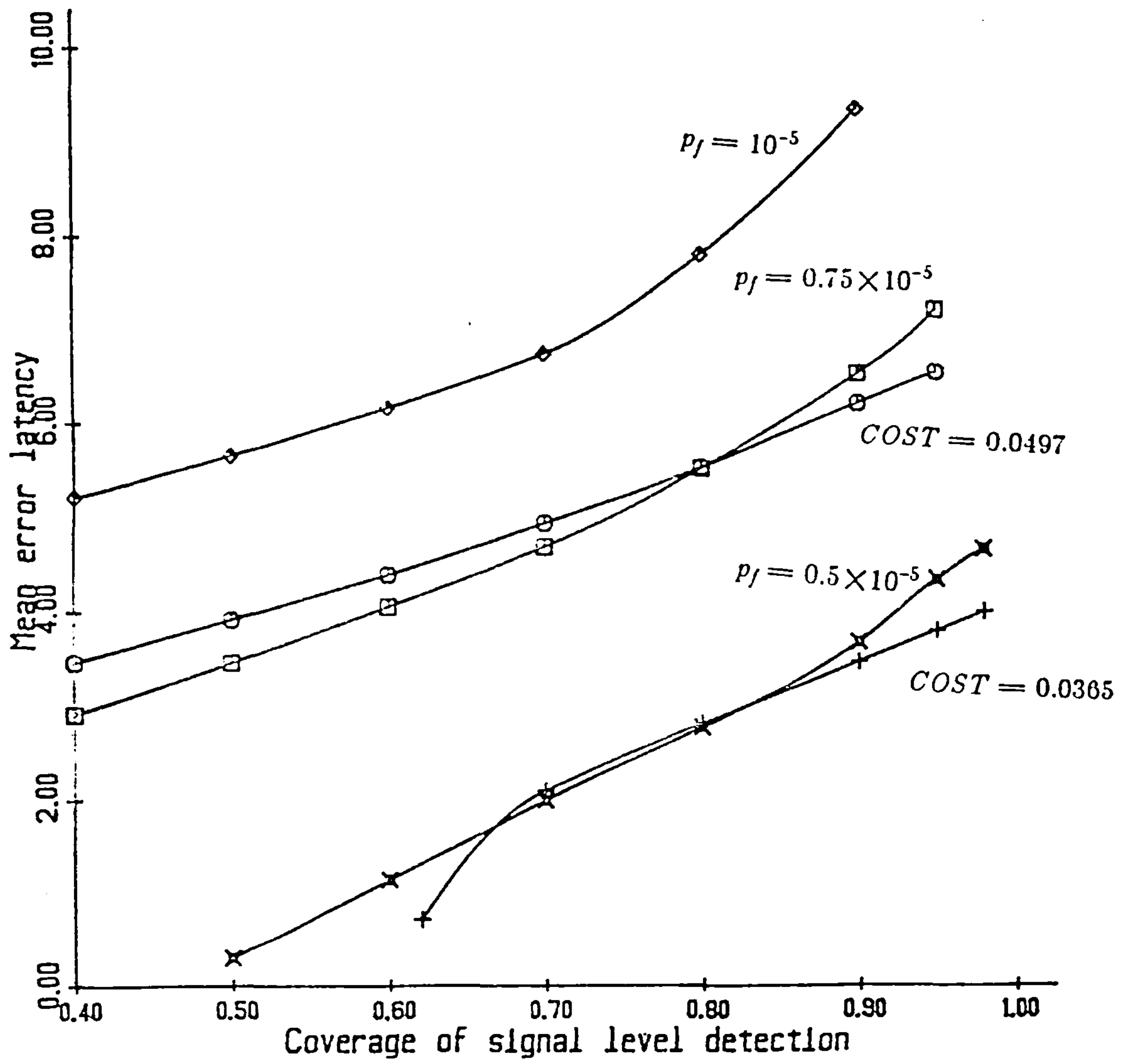


Figure 3.6 Design Space for Coverage and Mean Error Latency Subject to Constraints of p_f and *COST*.
(With the same system parameters as in Figure 3.5 and T_{dead}=150, p_{sv}=0.2).

related to actual hardware costs, (ii) error latency and performance of function level detection mechanisms are application-dependent, and (iii) the cost of function level detection mechanisms, especially, software checking, is neither well structured nor well understood at present.

4. EVALUATION OF SOFTWARE RECOVERY BLOCKS

The *recovery block* (RB), proposed by Horning [35] and Randell [58], has been widely used for backward error recovery. It is a sequential program structure that consists of an acceptance test, a recovery point(RP), and alternative algorithms for a given process. A process saves its state at a recovery point and then enters a recovery region. At the end of a recovery block, the acceptance test is executed to check correctness of the computation results. In case an error is detected during the normal execution or the computation results fail to pass the acceptance test, the process rolls back to an old state saved at the previous RP and executes one of the other alternatives.

Unfortunately, however, for cooperating concurrent processes the rollback of a process may cause other processes to roll back (this phenomenon is called *rollback propagation*) because of interprocess communications and imperfect checking of global correctness. Moreover, rollback may propagate to further RP's since recovery points of individual processes may not provide globally consistent states for all processes involved. This rollback propagation continues until it reaches a *recovery line* at which globally consistent states for all involved processes do exist. In the worst case, an avalanche of rollback propagation, called the *domino effect,* can push the processes back to their beginnings, thus resulting in loss of the entire computation done prior to the occurrence of error.

A detailed description of the domino effect can be found in [59]. For convenience let us consider Figure 4.1 to visualize rollback propagations. Process P_1 begins to roll back because of unsuccessful acceptance test AT_4^1. Due to interprocess communications the rollback P_1 propagates to the other two processes P_2 and P_3. Eventually, the whole system has to restart from recovery line RL_2, undoing the entire computation between RL_2 and AT_4^1. The time interval between the restart point following an error recovery and the time point at which an error is detected or the acceptance test fails, called the *rollback distance,* can be used to represent the computation loss in rollback error recovery. The rollback distance may be unbounded in the case of the domino effect.

The domino effect is the major obstacle in implementing the recovery block scheme for concurrent processing. The process designer is able to predict neither the time of the occurrence of process interactions nor that of the appearance of recovery lines. In addition, it is not desirable to randomly place recovery points and acceptance tests without considering process characteristics. Thus, it is impossible to avoid the domino effect only by appropriate placement of recovery blocks and it is possible to have a disaster such as unbounded rollback propagation, a large rollback distance, and a great number of largely useless recovery points occupying large amounts of memory space, etc. Furthermore, detection of rollback propagation and determination of recovery lines will become more complex though they can be made in a centralized [44-45] or decentralized manner [54,78,80].

Several refinements have been proposed to overcome the drawbacks in the recovery block scheme. One approach is to put concurrent processes into a controlled scope, either to synchronize the occurrence of acceptance tests or to direct process interactions. For the former, Randell [58] has suggested the *conversation scheme* which requests every

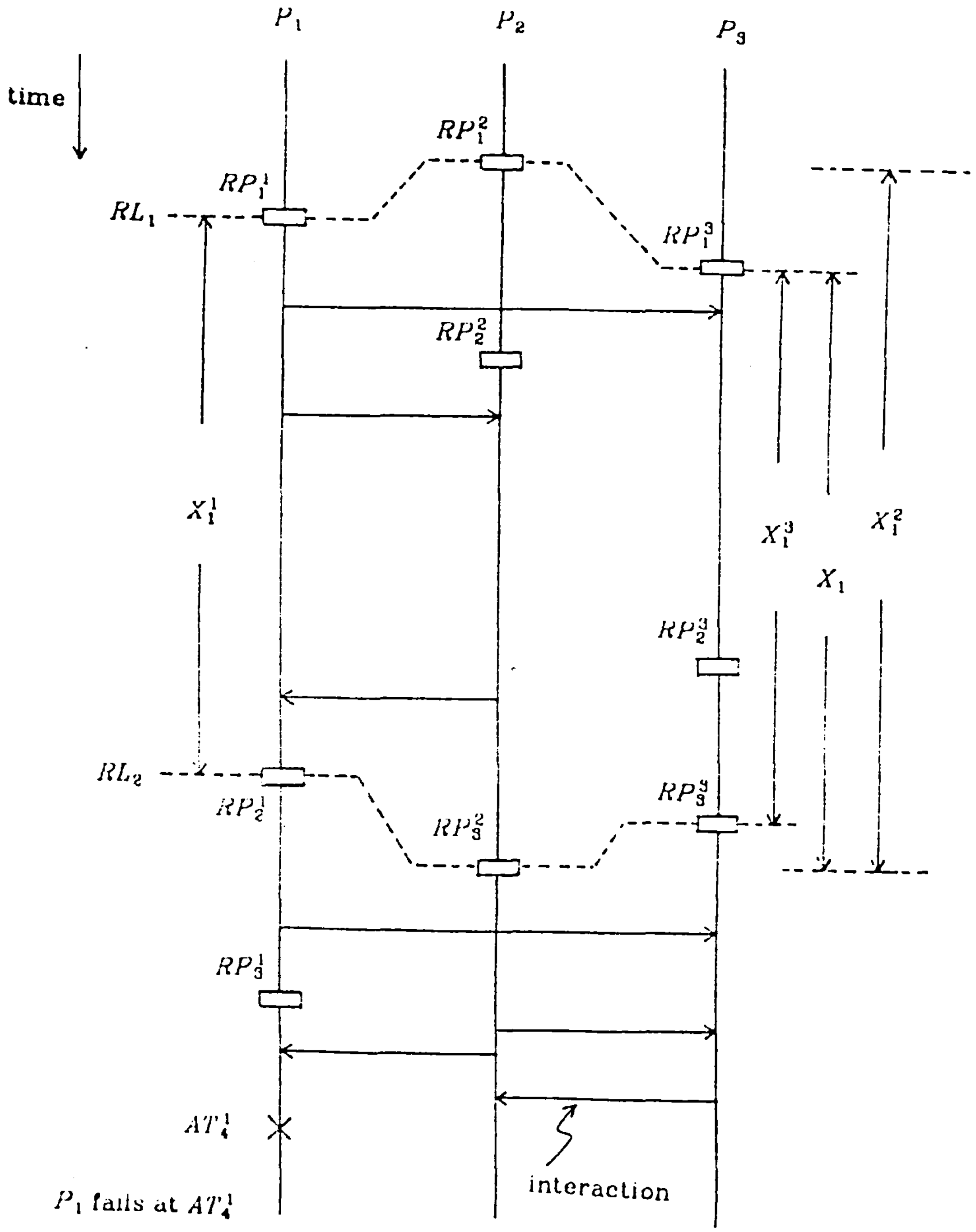


Figure 4.1 A History Diagram of Occurrence of Interactions and Recovery Points.

cooperating concurrent process to leave its acceptance test at the same moment (called *test line)*. He has also proposed a language structure in an abstract form for the conversation scheme. Other mechanizations of the conversation scheme on the basis of the same concept but with more flexibility have been devised by Kim [40]. Synchronized rollback recovery schemes for transactions using a two-phase commitment protocol or transaction ordering are also studied in [27,32,41]. Russell has proposed that information be retained for directed interactions from producers to consumers so that rollback propagation can be blocked [60-61]. Another approach is to save additional states based on the occurrence of interactions; for example, the branch recovery point [39] and the system defined checkpoint (SDCP) [36].

In this paper we propose to employ *pseudo recovery points*[3] (PRP's) to alleviate the rollback propagation problem by allowing a process to restart at a PRP in case the process is forced to roll back by others as a result of rollback propagation. Hence, we can classify these refinements into two categories, *synchronized recovery blocks* and *pseudo recovery points,* providing a contrast with the third category called *asynchronous recovery blocks*.

To implement a rollback error recovery scheme, we have to weigh trade-offs between these three categories and the characteristics of concurrent processes. A satisfactory scheme should have such features as a low (acceptable) delay in process completion due to rollbacks, the preservation of process autonomy in concurrent processing, and programmer transparency. Therefore, optimal solutions may be a combination of these

[3] We call it a pseudo recovery point(PRP) since there is no acceptance test before the saving of process state at a PRP. The states recorded at PRP's may have been contaminated and thus can not be used to recover a failed process. But PRP's can be used to prevent rollback propagations due to interactions with the faulty process as we shall see in the following.

three categories. A quantitative analysis has to precede any of such optimal solutions. For example, it is necessary to determine the mean amount of computation undone in case processes roll back, the optimal interval between two successive synchronizations, the mean size of memory space required to save states, etc. However, because the program behavior is unknown and its execution proceeds stochastically, accurate modeling is in general very difficult if not impossible.

In the following, we will develop models to quantitatively describe the characteristics of rollback recovery schemes as well as their effectiveness.

4.1. Evaluation of Asynchronous Recovery Blocks

Let us consider the history diagram in Figure 4.1 to illustrate the activities of cooperating concurrent processes P_i, $i=1,2,\ldots n$. Process P_i establishes its jth recovery point RP_j^i without synchronizing with other processes. Interprocess communications are represented by arrowed horizontal lines. Let set $A \subset \{1,\ldots,n\}$, i.e. a subset of concurrent processes. Then one may find a combination of RP_j^i for all $i \in A$, which forms a recovery line for set A, denoted as RL_r^A for the rth recovery line. For simplicity, superscripts in representing recovery lines will be omitted in the sequel as long as that does not result in ambiguity. The interval between two successive recovery lines RL_r and RL_{r+1} in process P_i is a random variable and denoted by X_r^i. Since a recovery line provides globally consistent states to all members of process set A, it is reasonable to assume that X_r^i is stochastically identical for all $i \in A$. Thus, X_r is used to represent the interval between the rth and $(r+1)$th recovery lines.

4.1.1. Modeling Assumptions

We make the following assumptions in our subsequent analyses.

A1. *Autonomous Processes:* Cooperative autonomy is regarded as the most important requirement in distributed processing. Each process should be executed according to its own program and environment, almost as if there were no process to interfere with. Thus, processes will transmit messages or establish their recovery points independently of other processes.

A2. *Perfect Local Acceptance Test:* Acceptance tests should detect all errors within the local process during the execution of recovery blocks or, at least, guarantee that the computation results have passed acceptance test are "acceptable" [59].

A3. *Probability Distribution of Interactions:* To describe the occurrences of interactions, for both tractability and simplicity, we have adopted here the concepts of constant reference rates in the multiprocessor and of exponentially distributed intervals between two successive message transmissions in the computer network. The interval for two successive interactions between P_i and P_j is assumed to be exponentially distributed with mean $1/\kappa_{ij}$ and $\kappa_{ij}=\kappa_{ji}$ for all $i,j=1,2,...,n$ and $i\neq j$.

A4. *Consistent Communications:* Let two messages m_a and m_b be sent from P_i to P_j. Consistent communications should satisfy : (i) every message sent from P_i to P_j will be received eventually by P_j, and (ii) $m_a$and m_b are received by P_j in the same order as that they are sent.

A5. *Distribution of Recovery Points:* Because of process independence and the uncertainty of execution conditions, the appearances of recovery points are random and difficult to model. To avoid complexity, establishment of recovery points in a process is assumed to be an independent Poisson process with parameter ς_i for process

P_i.

4.1.2. A Model for Asynchronous Recovery Blocks

Since individual recovery points by themselves may not be sufficient in rollback recovery due to the possibility of rollback propagations, we consider in this paper only the formation of recovery lines for asynchronous recovery blocks instead of separate individual recovery points. The requirements of a recovery line for processes P_i, for $i=1,2,...n$, can be stated as follows:

1. Each recovery line has to include one recovery point RP_j^i for every process P_i.

2. Let the moment of establishment of the jth recovery point in process P_i be $t[RP_j^i]$ and let $t_q^{ii'}$ be the moment of the qth interaction from P_i to $P_{i'}$. For every pair $(RP_j^i, RP_{j'}^{i'})$ in a recovery line, there does not exist an integer k such that $t_k^{ii'} \in [t[RP_j^i], t[RP_{j'}^{i'}]]$ if $t[RP_j^i] \leq t[RP_{j'}^{i'}]$ (otherwise, $t_k^{ii'} \in [t[RP_{j'}^{i'}], t[RP_j^i]]$). This implies that no communication from P_i to P_j (and vice versa) can be sandwiched between $t[RP_j^i]$ and $t[RP_{j'}^{i'}]$.

The basic idea underlying the model is to trace the occurrence of both recovery points and interactions. Based on the assumptions, random variable X_r can be modeled by a continuous-time Markov process starting from a recovery line (RL_r) and ending at the next recovery line (RL_{r+1}). For a set of processes, $\Omega_A=\{P_i|i\in A\}$ where $A=\{1,2,...,n\}$, two types of states are defined:

(a). End states S_r and S_{r+1}: transitions start from S_r where all processes have formed the rth recovery line, and end at S_{r+1} upon establishment of the $(r+1)$th recovery line.

(b). Intermediate states $S = (x_1, x_2, \ldots, x_n)$, where $x_i = 0$ if the previous action of P_i was an interaction, and $x_i = 1$ if it was establishment of a recovery point.

Note that both S_r and S_{r+1} are equivalent to state (1,1,...,1).

Occurrences of interactions and recovery points in a process make the system go through these states. Note that both S_r and S_{r+1} are equivalent to state (1,1,...,1). We can establish the following transition rules:

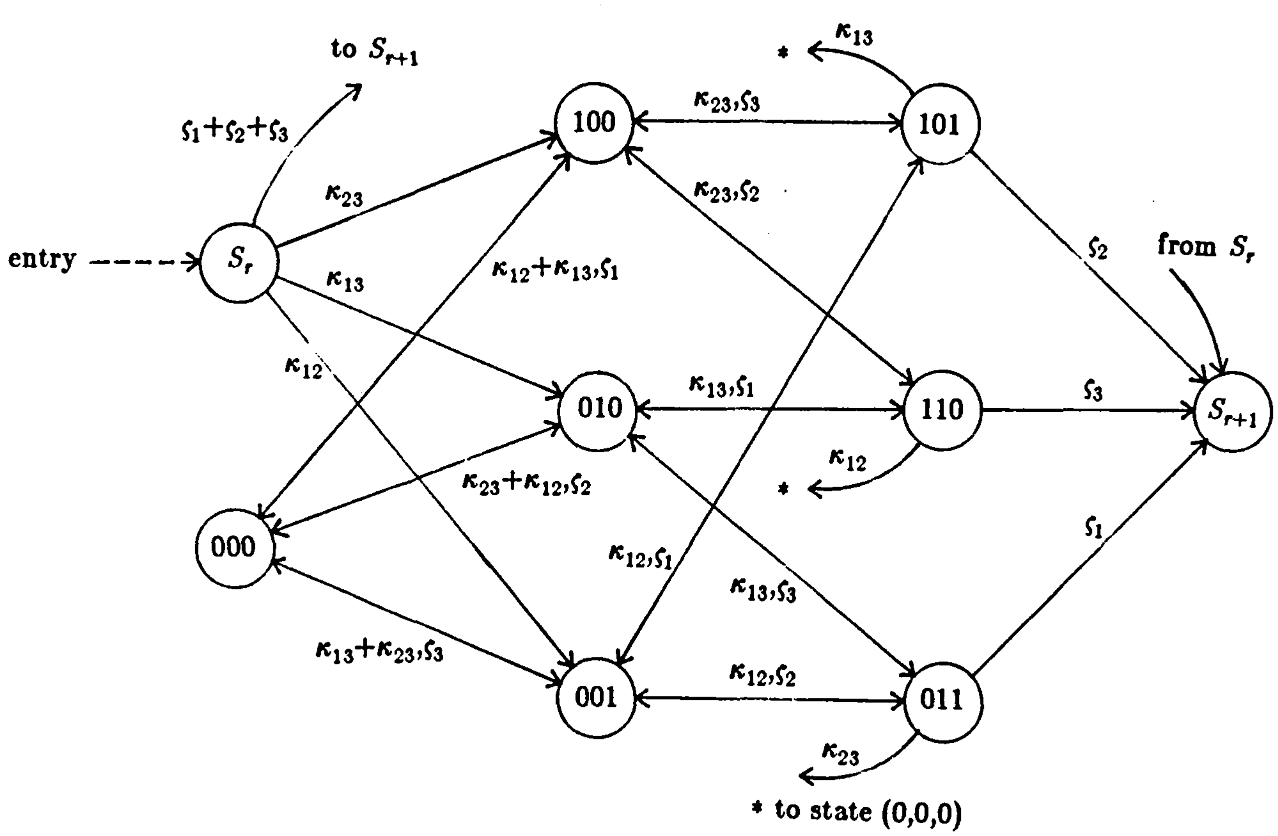
R1. The system goes to state $(x_1,..,x_{i-1},1,x_{i+1},..,x_n)$ from state $(x_1,..,x_{i-1},0,x_{i+1},..,x_n)$ with rate ς_i upon establishment of a recovery point in P_i.

R2. The system leaves state $(x_1,..,x_{i-1},1,x_{i+1},..,x_{j-1},1,x_{j+1},..,x_n)$ and enters state $(x_1,..,x_{i-1},0,x_{i+1},..x_{j-1},0,x_{j+1},..,x_n)$ with rate κ_{ij} if there is an interaction between P_i and P_j.

R3. The system arrives at state $(x_1,..,x_{i-1},0,x_{i+1},..,x_n)$ from state $(x_1,..,x_{i-1},1,x_{i+1},..,x_n)$ with transition rate $\sum_{j \in B_i} \kappa_{ij}$ where $B_i = \{j \mid x_j = 0, j \neq i \text{ and } j \in A\}$.

R4. The system can transfer directly from state S_r to state S_{r+1} with transition rate $\sum_{k=1}^{n} \varsigma_k$.

Under these transition rules a Markov model is developed for three processes P_1, P_2 and P_3, and presented in Figure 4.2. The single-arrow lines are unidirectional transitions. The double-arrow lines are bidirectional transitions in which left-hand side parameters represent leftward transition rates and right-hand side parameters rightward transition rates. The total number of states for a set of n processes is 2^n+1.



to S_{r+1}
$\varsigma_1+\varsigma_2+\varsigma_3$
entry
S_r
κ_{23}
κ_{13}
κ_{12}
100
010
000
001
$\kappa_{12}+\kappa_{13},\varsigma_1$
$\kappa_{23}+\kappa_{12},\varsigma_2$
$\kappa_{13}+\kappa_{23},\varsigma_3$
κ_{23},ς_3
κ_{23},ς_2
κ_{13},ς_1
κ_{12},ς_1
κ_{13},ς_3
κ_{12},ς_2
101
110
011
κ_{13}
κ_{12}
κ_{23}
*
* to state (0,0,0)
ς_2
ς_3
ς_1
from S_r
S_{r+1}

Figure 4.2 THe Model of Asynchronous RB's for 3 processes.

When $\varsigma_i = \varsigma_j = \varsigma$ and $\kappa_{ij} = \kappa$ for all $i, j \in A$, the model can be simplified since all intermediate states $S = (x_1, x_2, \ldots, x_n)$ containing exactly u 1's in $(x_1, x_2, \ldots, x_n)$ can be replaced by a single state $\hat{S}_u$ where $u = 0,1,2,\ldots,n-1$. A simplified model is obtained under the following transition rules and presented in Figure 4.3.

R1'. For $u = 0,1,\ldots,n-1$, the system will move to state $\hat{S}_{u+1}$ from state $\hat{S}_u$ with transition rate $(n-u)\varsigma$ when a new recovery point is formed.

R2'. For all $u \geq 2$, the system is able to leave state $\hat{S}_u$ for state $\hat{S}_{u-2}$ with rate $\dfrac{u(u-1)\kappa}{2}$.

R3'. For all $u \geq 1$, there is a transition from state $\hat{S}_u$ to state $\hat{S}_{u-1}$ with rate $u(n-u)\kappa$.

R4'. The system can transfer directly from the entry state, S_r, to the terminal state, S_{r+1}, with transition rate $n\varsigma$.

4.1.3. The Analysis of Asynchronous Recovery Blocks

With the model developed above, we can characterize the behavior of asynchronous recovery blocks in terms of the degree of interprocess communications and the distribution of recovery points. With the exponentially distributed interprocess communications and recovery points, X_r becomes stochastically identical for all r. Let X denote a random variable representing the interval between two successive recovery lines, L_i the number of states saved in process P_i during interval X. The probability distribution of X and the mean value of L_i are derived below.

(a). The distribution of X

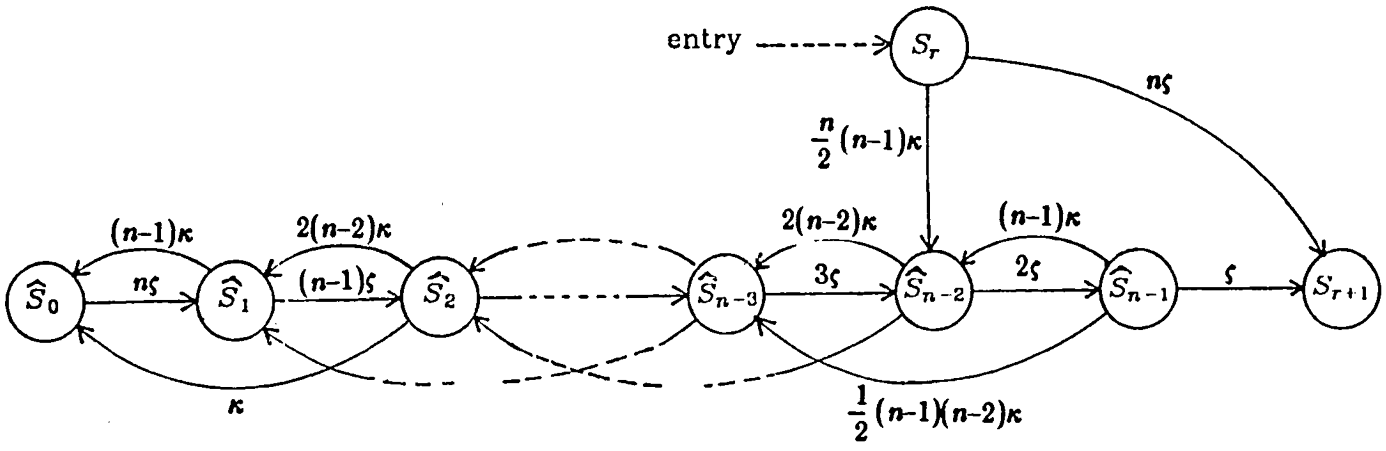


Figure 4.3 The Simplified Model of Asynchronous RB's for n Processes.

Let the state space $\Psi=\{0,1,2,...,m\}$ where $m=2^n$ be the set of states of the preceding continuous-time Markov process with the following convention for numbering states:

(a). $S_r \rightarrow$ state 0,

(b). an intermediate state $(x_1,x_2, \ldots ,x_n) \rightarrow$ state $(\sum_{i=1}^{n} x_i 2^{i-1} +1)$, and

(c). $S_{r+1} \rightarrow$ state m.

Then, the Chapman-Kolmogorov equation becomes

$$\frac{d}{dt}\pi(t) = \pi(t)\mathbf{H}_s \tag{4.1}$$

where $\mathbf{H}_s$ is the $((m+1)\times(m+1))$ transition matrix $[h_s(u,v)]$ in which the (u,v) element is the transition rate from state u to state v, and $\pi(t)$ is a vector whose kth element is the probability that the system is in state k at time t. The initial condition is $\pi(0)=[1,0,0...,0]$. The interval between two successive recovery lines, X, is equal to the time needed for transition from state 0 to state m. Therefore, the density function of X, namely $f_X(t)$, is given by

$$f_X(t)=\frac{d}{dt}\pi_m(t) \tag{4.2}$$

(b). The mean value of L_i

Since we are only concerned with the number of recovery points established by process P_i during interval X, a discrete Markov chain is used. To compute the mean value of L_i, a new Markov chain, denoted by Y_d, is constructed based on the previous model with the following two steps.

S1. Convert the previous model to a discrete model:

The new chain, Y_d, has the same states as the previous Markov process. Let $G = \sum_{i=1}^{n} \sum_{j=1, j \neq i}^{n} \kappa_{ij} + \sum_{k=1}^{n} \varsigma_k$ be the normalization factor. The transition probability from state u to state v in Y_d is equal to: for u, $v = 0,1,...,m$, $p(u,v) = \frac{h(u,v)}{G}$ if $u \neq v$, and $p(u,u) = 1 - \sum_{v=1, v \neq u}^{n} p(u,v)$

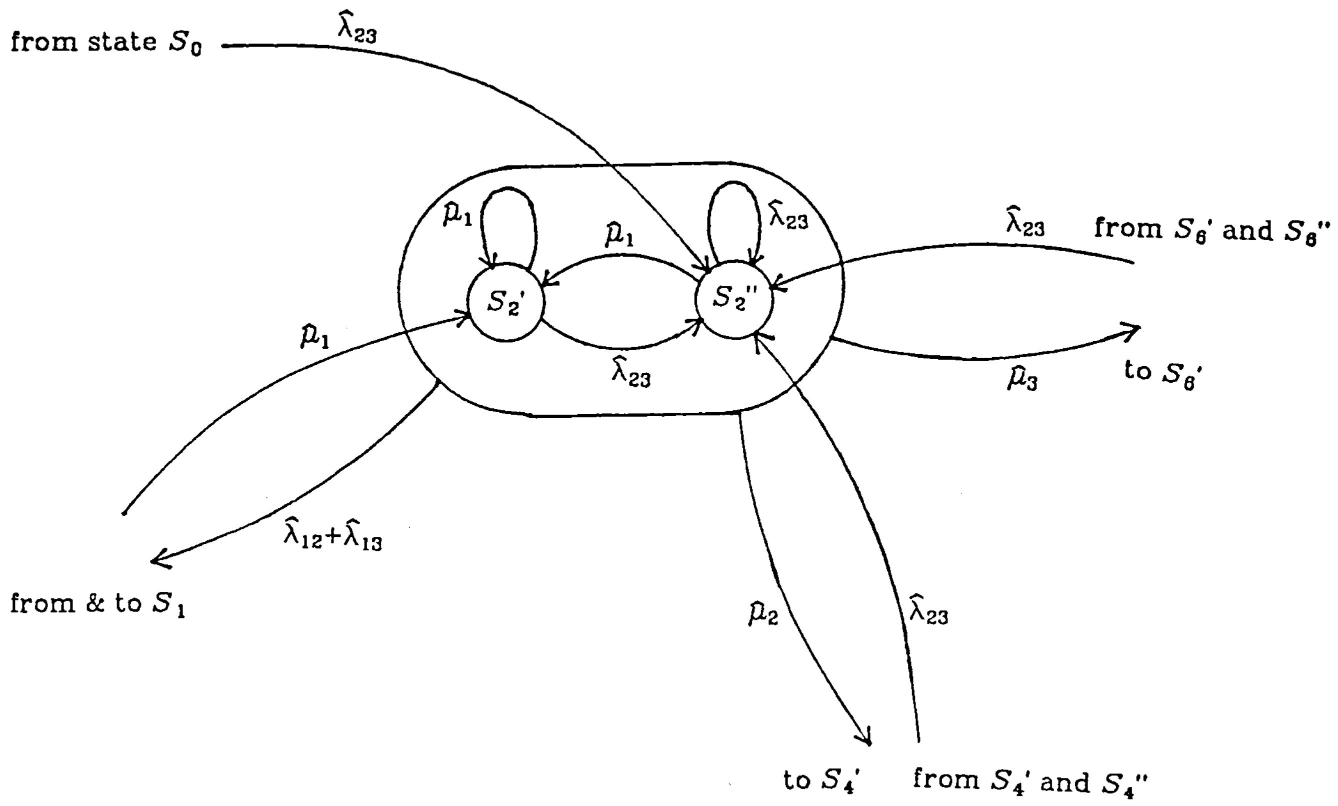
S2. Decompose states of discrete model:

Arrivals at a state $S_u = (x_1, x_2, ..., x_i, ..., x_n)$ where $x_i = 1$ can be grouped into two classes. One is formed as a result of the occurrences of RP's in P_i and the other is formed as a result of interprocess communications and establishments of RP's in processes other than P_i. Accordingly, the state $S_u = (x_1, x_2, ..., x_i, \ldots, x_n)$ with $x_i = 1$ can be split into two states S_u^1 and S_w^2 representing the two classes, respectively. Both states have the same departure processes as that of S_u. However, all arrivals at state S_u due to the occurrence of recovery points in P_i enter state S_u^1 whereas all other transitions are made to S_u^2. Hence the number of RP's associated with state S_u^1 is represented by that of arrivals at S_u^1.

Figure 4.4 shows the conversion and the split of state $S_2 = (1,0,0)$ of the Markov model for the three concurrent processes in Figure 4.2. With the new discrete model, Y_d, we can calculate the the mean number of visits to state S_w^1, denoted as $N_{S_u^1}$, and the mean value of L_i using the following relationship:

$$E(L_i) = \sum_{S_u^1 \in \Psi_{Y_d}} E(N_{S_u^1}) \qquad (4.3)$$

where Ψ_{Y_d} is the state space of Y_d.



$\hat{\lambda}_{ij}=\frac{\lambda_{ij}}{G}$, $\hat{\mu}_k=\frac{\mu_k}{G}$, and $G=\sum_{i=1}^{n}\sum_{j=1\, j\neq i}^{n}\lambda_{ij}+\sum_{k=1}^{n}\mu_k$

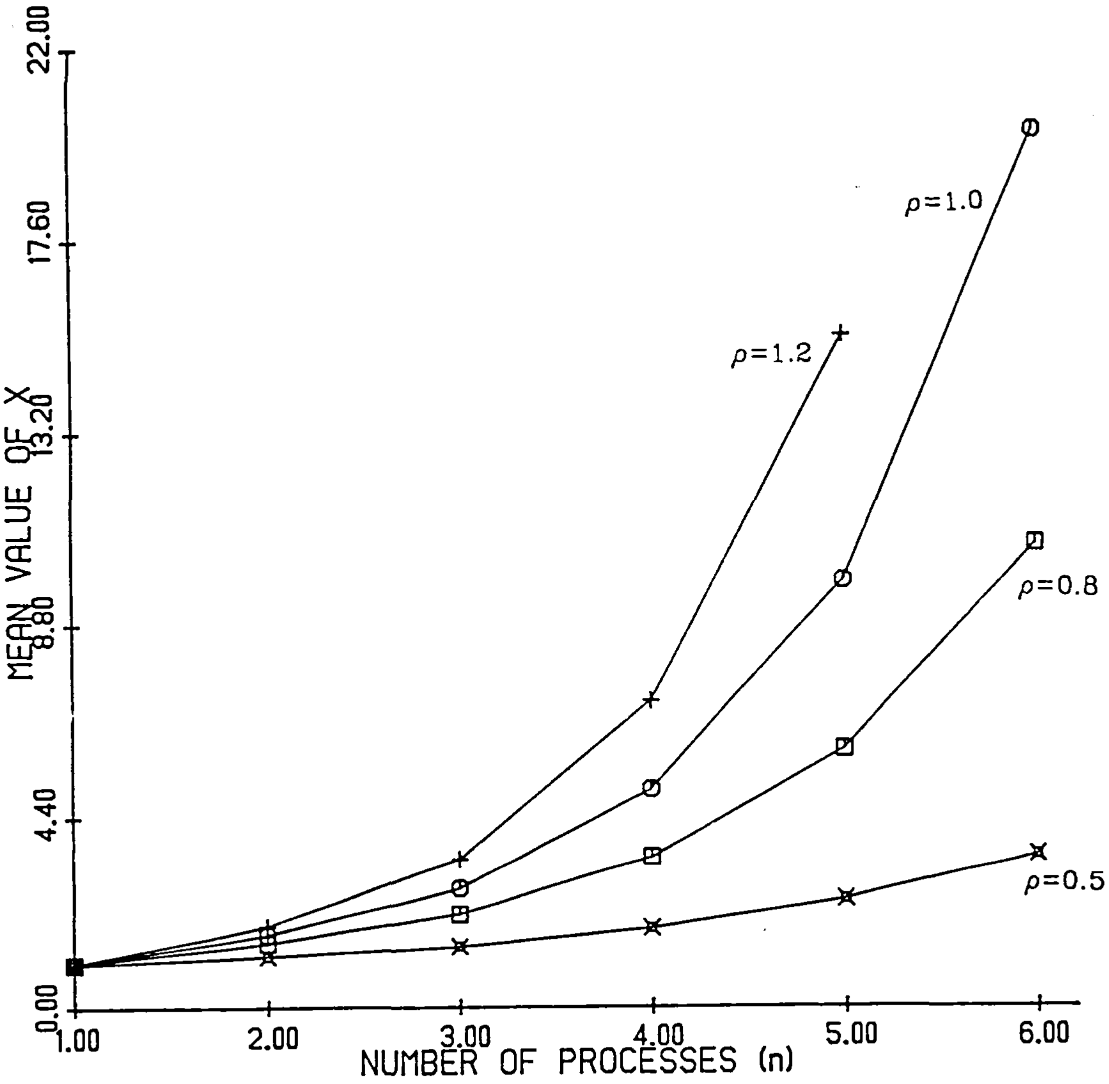
Figure 4.4 The Construction of State S_2^1 and S_2^2 of Discrete Markov Chain Y_d.

Suppose process P_i detects an error or fails the acceptance test at one of its recovery points RP_j^i, where $j=1,2,...,L_i$. The rollback of P_i may propagate to k processes in the process set, $\Omega_A = \{P_l | l \in A\}$ where $A=\{1,2,...,n\}$. Let D_j^k be the rollback distance associated with the k processes and RP_j^i for $j=1,2,..,L_i$. Then, X represents the supremum of these random variables, i.e., $D_{L_i}^n$. Let $\rho = (\sum_{i=1}^{n} \sum_{j=1,j\neq i}^{n} \kappa_{ij})/(\sum_{k=1}^{n} \varsigma_k)$ which represents the relative ratio between the density of interprocess communications and recovery point establishments. In Figure 4.5, the mean values of X are plotted as a function of n for different values of ρ. It shows that X increases drastically when there is an increase in the number of processes involved in the rollback recovery. The density function of X, $f_x(t)$, is plotted in Figure 4.6. For all the three cases in Figure 4.6, there is a sharp pulse near $t=0$, which is due to direct transitions between S_r and S_{r+1} and a longer transition time needed once the system enters intermediate states.

With a fixed value of ρ and varying values of ς's and κ's for three processes, we have performed computer simulation and the results are tabulated in Table 4.1. The minima of X and L_i occur when the distribution of recovery points among these processes is uniformly balanced (i.e., $\varsigma_1=\varsigma_2=\varsigma_3$). The distribution of interprocess communications does play an important role in determining the probability of rollback propagation but has little effect on X and L_i once the set of processes involved in rollback recovery is determined.

4.2. Synchronized Recovery Blocks

The simplest way of avoiding unbounded rollback propagation is to synchronize the establishment of recovery points during process execution. In this method, interac-



$$\rho = \left(\sum_{i=1}^{n} \sum_{j=1, j \neq i}^{n} \lambda_{ij}\right) / \left(\sum_{k=1}^{n} \mu_k\right)$$

$\lambda_{ij} = \lambda$ for all i, j and $\mu_1 = \mu_2 = \ldots. = \mu = 1.0$

Figure 4.5 Mean Value of X vs. the Number of Processes.

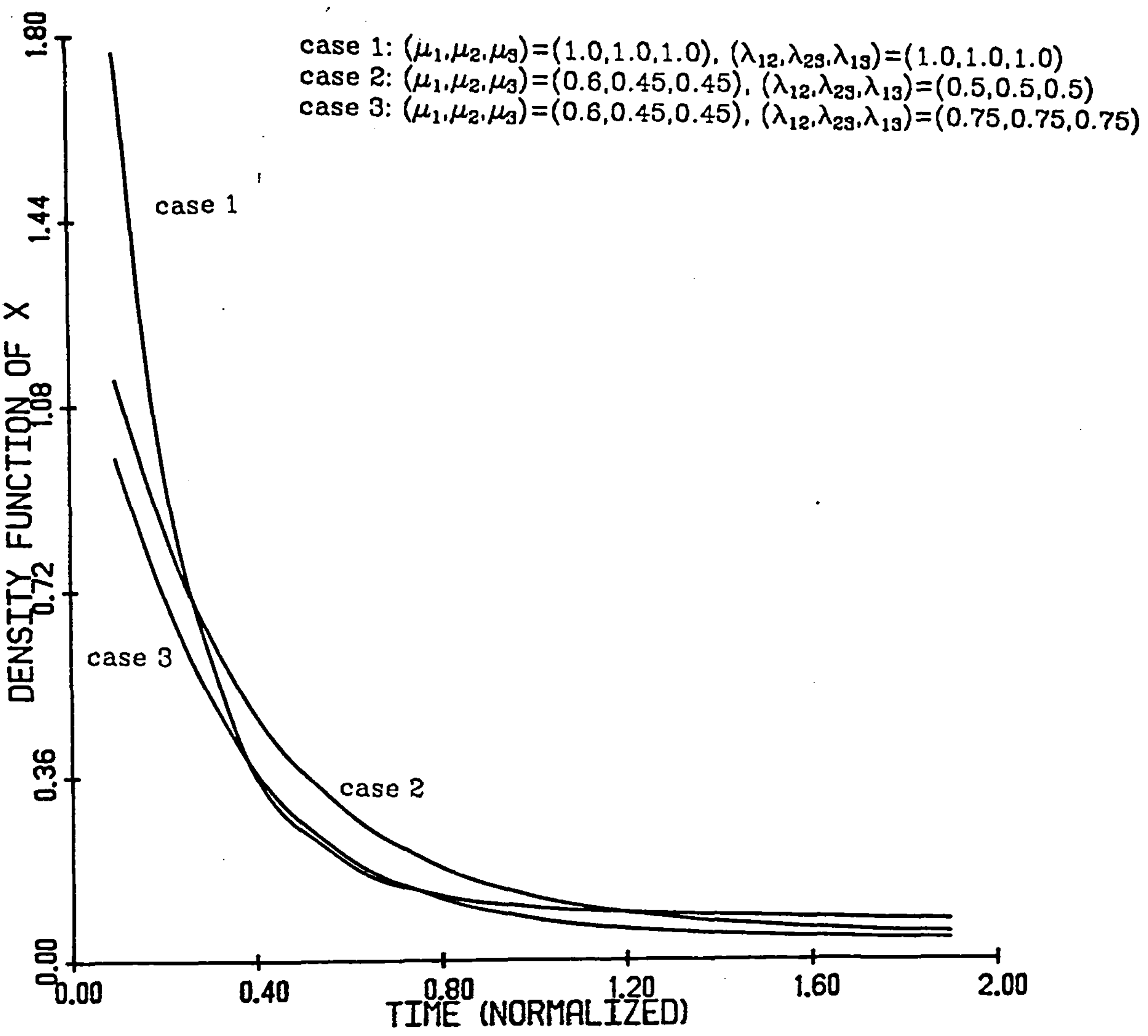


Figure 4.6 The Density Function of *X*, $f_x(t)$.

(μ_1,μ_2,μ_3) $(\lambda_{12},\lambda_{23},\lambda_{13})$	(1.0,1.0,1.0) (1.0,1.0,1.0)	(1.5,1.0,0.5) (1.0,1.0,1.0)	(1.0,1.0,1.0) (1.5,0.5,1.0)	(1.5,1.0,0.5) (1.5,0.5,1.0)	(1.5,1.0,0.5) (0.5,1.5,1.0)
$E(X)$	2.598	3.357	2.600	3.203	3.354
$E(L_1)$	2.500	4.847	2.453	4.533	4.967
$E(L_2)$	2.500	3.231	2.453	3.022	3.111
$E(L_3)$	2.500	1.616	2.453	1.511	1.656
$E(L_1+L_2+L_3)$	7.500	9.693	7.360	9.065	9.933

Table 4.1 Mean Values of X and L_i for Constant ρ.

tions are inhibited between any pair of processes during their establishment of recovery points. There are three conceivable strategies in deciding when a synchronization request is to be issued: (1) at a constant interval, denoted as T_s; (2) when the time elapsed since the previous recovery line exceeds a specified value, T_s' ; or (3) when the number of states saved after the previous recovery line is larger than a prespecified number, M_s. The implementation of the first strategy is simple since the synchronization request is issued without any knowledge of the state of execution. Nevertheless, some synchronization requests may become redundant and unnecessary if they are issued immediatedly after the formation of recovery lines. For the second and third strategies, the rollback distance and the number of saved states are prevented from becoming too large. However, for these two strategies, additional overhead will be required because each process must be aware of the occurrence of a recovery line whenever it is established. Note that the conversation scheme is a special case of the third strategy where M_s=1.

Upon the receipt of a synchronization request, every process has to prepare for establishing a recovery line and also has to wait for the commitment (for establishing a recovery line) from other processes before it executes an acceptance test. Thus, all cooperating processes perform their acceptance tests at the same instant upon receiving the commitments from all other processes. Let P_{ij}*-ready*, for j=1,2,...,n, be the flags in process P_i to indicate commitments from P_j. The steps for synchronization in each process P_i are described as follows:

1. **execute** "its own normal process" until "acceptance test";
2. set P_{ii}*-ready* := ON and then broadcast P_{ii}*-ready*;

```
3. while not (all $P_{ij}$-ready = ON) do
    receive messages;
    if a message is $P_{ji}$-ready then set $P_{ij}$-ready := ON
    else record the message
4. do "acceptance test" and record process states.
```

Establishment of recovery lines upon synchronization requests is shown in Figure 4.7. Synchronization causes the computation power to be reduced because processes have to wait for the commitments (as in step 3) from other processes. And, process autonomy, a principal characteristic of distributed computing systems, is sacrificed. Let y_i be the interval between the receiving of a synchronization request and the moment that process P_i reaches its next acceptance test (in step 1). Then, according to the assumptions in Section 2.1, y_i is an exponentially distributed random variable with parameter ς_i. Let $Z=\max\{y_1, y_2, \ldots, y_n\}$. The total loss in computation power is $CL=\sum_{i=1}^{n}(Z-y_i)$. The mean loss becomes

$$\overline{CL} = n\int_0^{\infty}(1-F_z(t))\,dt - \sum_{i=1}^{n}\frac{1}{\varsigma_i} \tag{4.4}$$

where $F_z(t)$ is the distribution function of Z, and equals $\prod_{i=1}^{n}(1-e^{-\varsigma_i t})$.

The time interval between two successive recovery lines is a function of the strategy used for issuing synchronization requests as well as characteristics of the processes involved (e.g. patterns of interprocess communications and RP establishments). Let Z^1 and Z^2 be random variables having the same distribution as $Z=\max\{y_1, y_2, \ldots, y_n\}$, then the value of this time interval becomes $T_s+Z^1-kZ^2$ where k is the largest integer

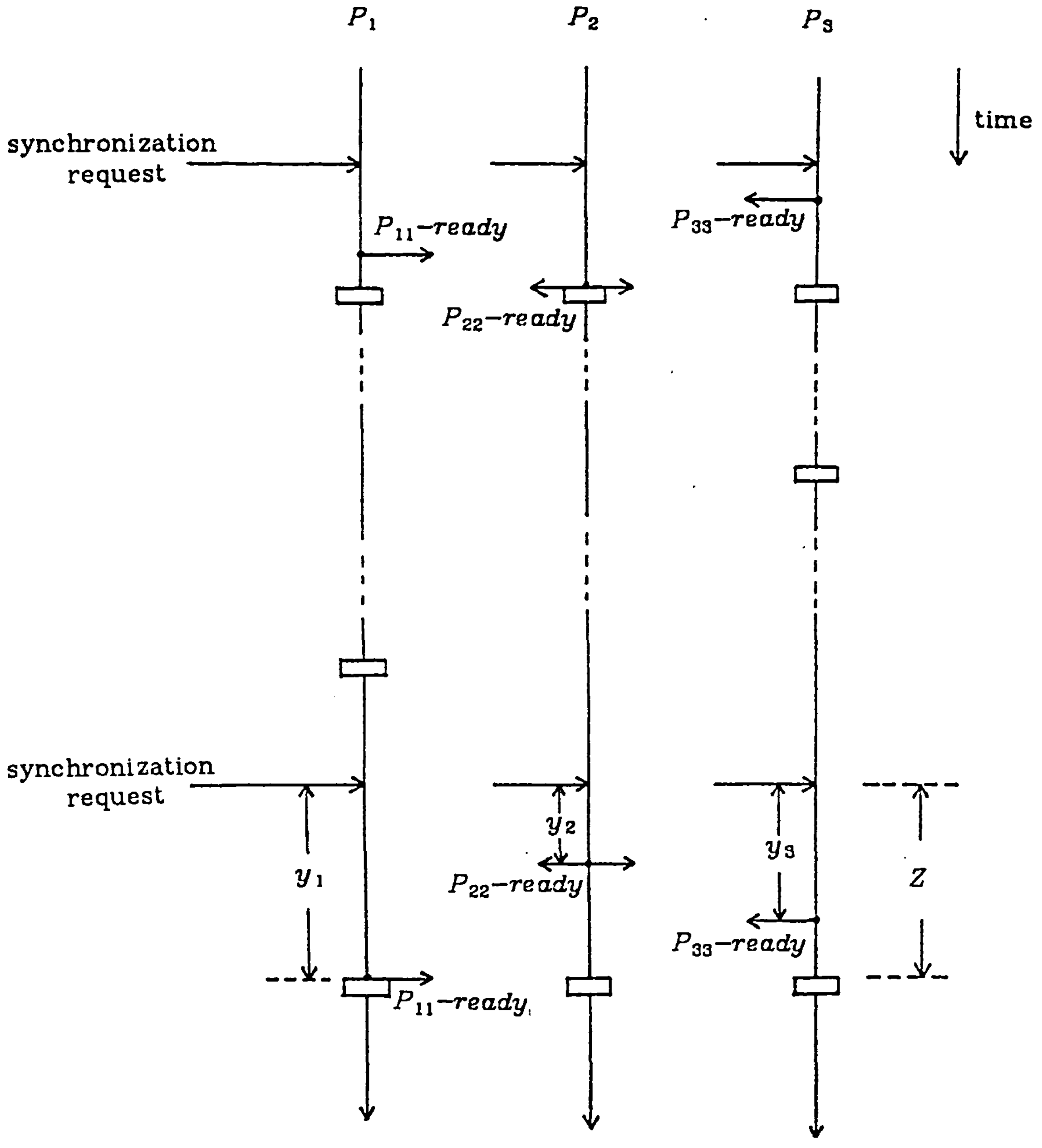


Figure 4.7 Establishment of Recovery Lines upon Synchronization Requests.

which is less than T_s/Z^2, or $T_s' + Z^1$. Observe that Z^1 and Z^2 represent the amount of time required for a process to be ready for establishing an RP after it received a synchronization request. For the third strategy, the maximum number of rollback steps is M_s. Thus the supremum of this time interval can be expressed as $\max\{z_1, z_2, \ldots, z_n\}$ where $z_i = \sum_{n=1}^{M_s} y_i$.

4.3. Implantation of Pseudo Recovery Points

In the construction of a recovery block, an acceptance test consists of a number of executable assessments provided by the programmer, followed by a state saving. Note that process states can also be recorded upon any other requests whenever they are considered useful in the rollback recovery. A ***pseudo recovery point*** (PRP) is defined as a recovery point that is established without a preceding acceptance test and is proposed here as an alternative for avoiding the domino effect in a set of cooperating concurrent processes. With a monitor as the interprocess communication means, Kim [39] and Kant and Silberschatz [36] discussed methods for implanting recovery points in a central manner. Similarly, we consider a method for implanting PRP's in the set of cooperating concurrent processes in a decentralized manner. Also, note that the use of PRP's does not require any particular interprocess communications mechanism (e.g. the implementation does not have to be based on monitors).

To make a recovery point RP_j^i in process P_i maximally useful for rollback error recovery, there should be corresponding recovery points in the other processes affected by the rollback propagation from P_i. If such recovery points do not actually exist, for a given RP_j^i in process P_i a pseudo recovery point, $PRP_j^{ii'}$, has to be inserted in process

$P_{i'}$. Further, in order to avoid the need of tracing recovery points at that particular moment, for RP_j^i a PRP is established in each of the other processes involved. An algorithm for implanting PRP's is given below.

1. When P_i establishes a recovery point RP_j^i, it broadcasts a PRP implantation request to other processes.
2. If $P_{i'}$ receives the implantation request, it records its state as $PRP_j^{ii'}$ upon the completion of the current instruction without an acceptance test. Then $P_{i'}$ broadcasts the commitment $C_{i'}$.
3. Every process executes its own normal task after it establishes RP_j^i or $PRP_j^{ii'}$. However, the messages sent to a process by $P_{i'}$ prior to $C_{i'}$ have to be retained in the state saved.

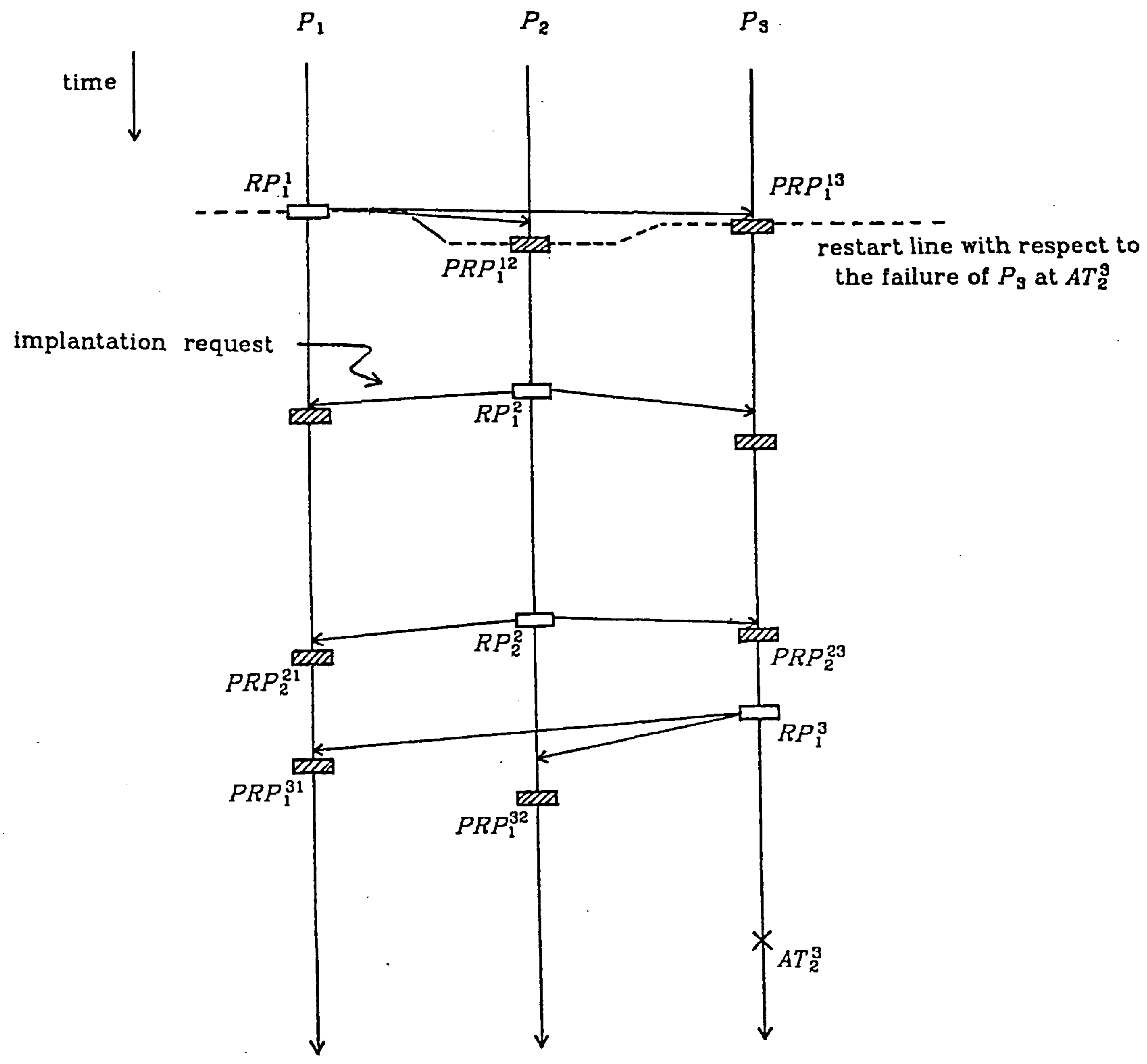
Assume that process P_i detects an error at time t_d which is prior to the establishment of RP_{j+1}^i. If this error is local to P_i then the recovery line (called a *pseudo recovery line, PRL_j^i)* formed by RP_j^i and all $PRP_j^{ii'}$'s is able to recover these processes even if the error has already propagated to other processes. However, when the error detected in P_i is due to error propagation from another process, P_l (and therefore not local to P_i), the contents of PRP_j^{il} may have already been contaminated if this error occurred prior to establishing PRP_j^{il}. The restart from the pseudo recovery line formed by both RP_j^i and all $PRP_j^{ii'}$'s may just reproduce the same error. Therefore, rollback propagation may continue until every process involved has rolled back to a pseudo recovery line, say $PRL_k^{\hat{i}}$, for which all processes but $P_{\hat{i}}$ have passed at least one of their recovery points. Since there exists an $RP_j^{i'}$ in $P_{i'}$ for all $i' \neq \hat{i}$ between $PRL_k^{\hat{i}}$ and t_d, every state belonging to $PRL_k^{\hat{i}}$ is now guaranteed to contain correct information of the corresponding process[4]. Also, note that this pseudo recovery line renders the shortest rollback distance for backward error recovery in case forced synchronization is not used. An algorithm of rollback recovery with these pseudo recovery points is given by:

1. If an error is found in process P_i, set $p := i$ where p is a rollback pointer.
2. P_p rolls back to its previous recovery point RP_j^p. All processes $P_{i'}$ affected by the rollback of P_p roll back to their respective pseudo recovery points $PRP_j^{pi'}$.
3. For every affected process $P_{i'}$, if the rollback has not passed its most recent recovery point, then set $p := i'$ and go back to step 2.

In Figure 4.8, the establishment of *PRP*'s in processes P_1, P_2, and P_3 is illustrated. When P_3 fails its acceptance test AT_2^3, all processes have to restart from the pseudo recovery line formed by (RP_1^1, PRP_1^{12}, PRP_1^{13}) if P_1 and P_2 are affected by the rollback of P_3.

In the above algorithm, we can find that every process needs to preserve a recovery point for restart in case it fails. Also (n-1) pseudo recovery points are needed for a process to form a pseudo recovery line with other processes where n is the total number of concurrent processes. It is therefore required to save n states for every *RP*, i.e. one *RP* and (n-1) *PRP*'s, and all old *RP*'s and *PRP*'s except those in the pseudo recovery lines $\{PRL_j^i \mid i = 1,...,n$, and RP_j^i is the most recent *RP* in $P_i\}$ can be purged when a new recovery point is established, thereby reducing storage requirements for saving *RP*'s and *PRP*'s. Note that rollback distance is bounded by the supremum of $\{y_1, y_2, \ldots, y_n\}$ where y_i is the interval between two successive recovery points of process P_i. The addi-

[4] If the state saved at $PRP_k^{ii'}$ was contaminated, then the error should have been detected at the subsequent recovery point, $RP_{j'}^{i'}$. Meanwhile, the state saved at RP_k^i is correct by the assumption of perfect local acceptance test.



▭ : Recovery Point (RP)
▨ : Pseudo Recovery Point (PRP)

Note: all occurrences of interactions are omitted

Figure 4.8 Establishment of Pseudo Recovery Points for Rollback Error Recovery.

tional time overhead for every recovery point is $(n-1)t_r$ where t_r is the time needed to record the process state. These overheads should be assessed against the gain of process autonomy and avoidance of unbounded rollback propagations.

5. DESIGN AND EVALUATION OF HARDWARE RECOVERY BLOCKS

In this chapter, we employ the concept of recovery blocks to construct a hardware rollback recovery mechanism for multiprocessor. 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 memory--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-saving unit implemented in hardware.

The hardware recovery block is different from the software recovery block which only saves non-local states when a checkpoint is encountered. Moreover, instead of the assertions in the acceptance test of software recovery block, the hardware resources are tested by embedded checking circuits and diagnostic routine.

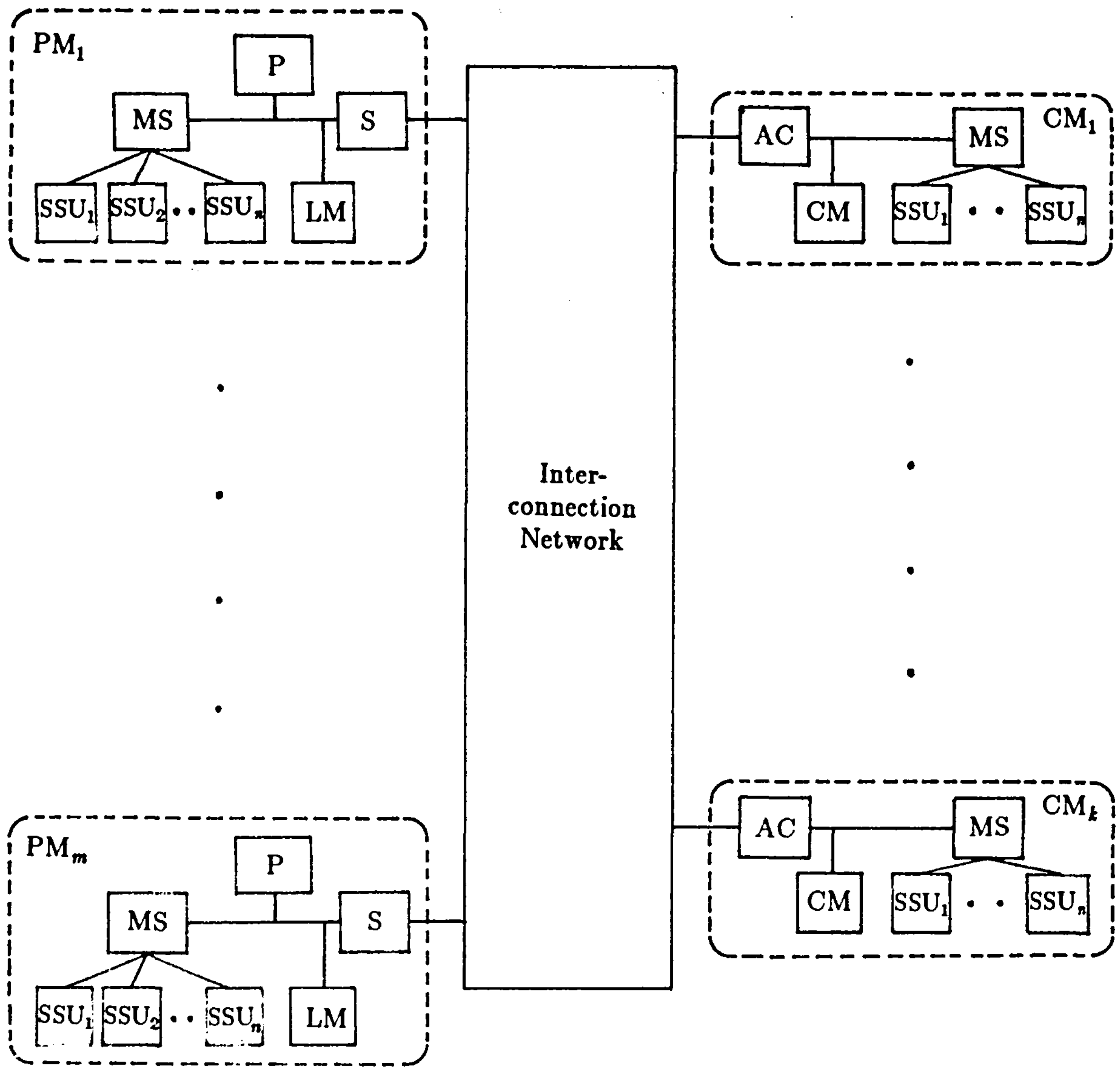
In the following, we will describe the structure of this hardware recovery block. Then, the coverage of a multi-step rollback which is the probability of having a successful rollback recovery when cooperating processes roll back multiple steps, and the performance of this method will be discussed.

5.1. Hardware Recovery Blocks for 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 [72]); in the other, the system is equipped with separate common memory modules [25] and restricts the access of local memory only to the resident processor. These two types are representatives of contemporary general-purpose multiprocessors.

5.1.1. 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 (SSUs) and a monitor switch as shown in Figure 5.1. 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 PMs) through the shared variables. Each PM saves its states (i.e. process local variables and processor status) in SSUs 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, T_{s}. This invocation signal will stimulate



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

Figure 5.1 The Organization of a Fault-Tolerant Multiprocessor Using a Rollback Recovery Mechanism.

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 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 Figure 5.2.

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 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 [38], dual-redundancy comparison, address-in-bound check, etc. These units are expected to detect malfunctions whenever the

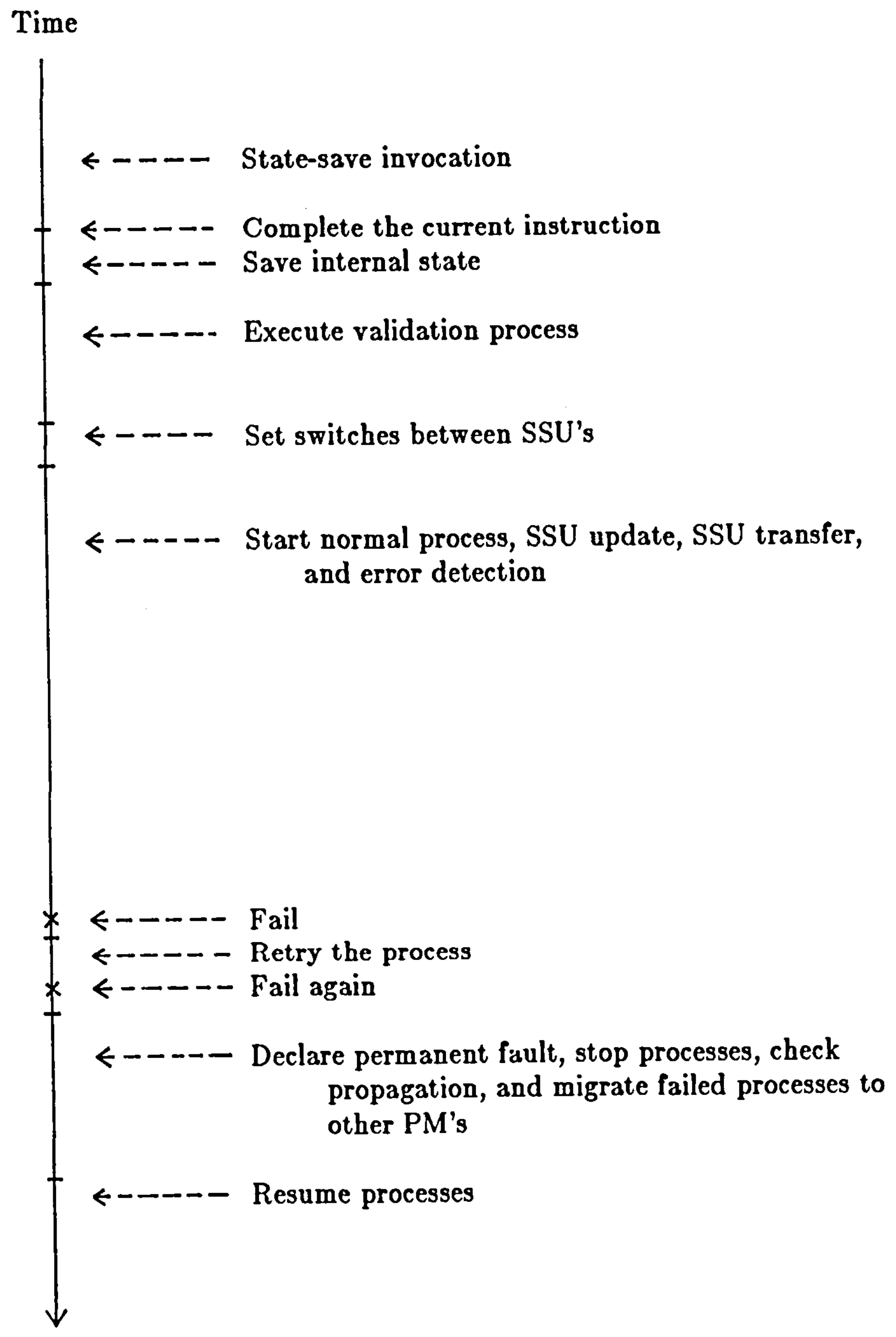

Time
State-save invocation
Complete the current instruction
Save internal state
Execute validation process
Set switches between SSU's
Start normal process, SSU update, SSU transfer, and error detection
Fail
Retry the process
Fail again
Declare permanent fault, stop processes, check propagation, and migrate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

Figure 5.2 Sequnece of a Rollbacl Recovery.

corresponding function units are used. An additional validation process which could be the execution of 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 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 Figure 5.3, 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. In the next section, we will show that only a small number of SSUs 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 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

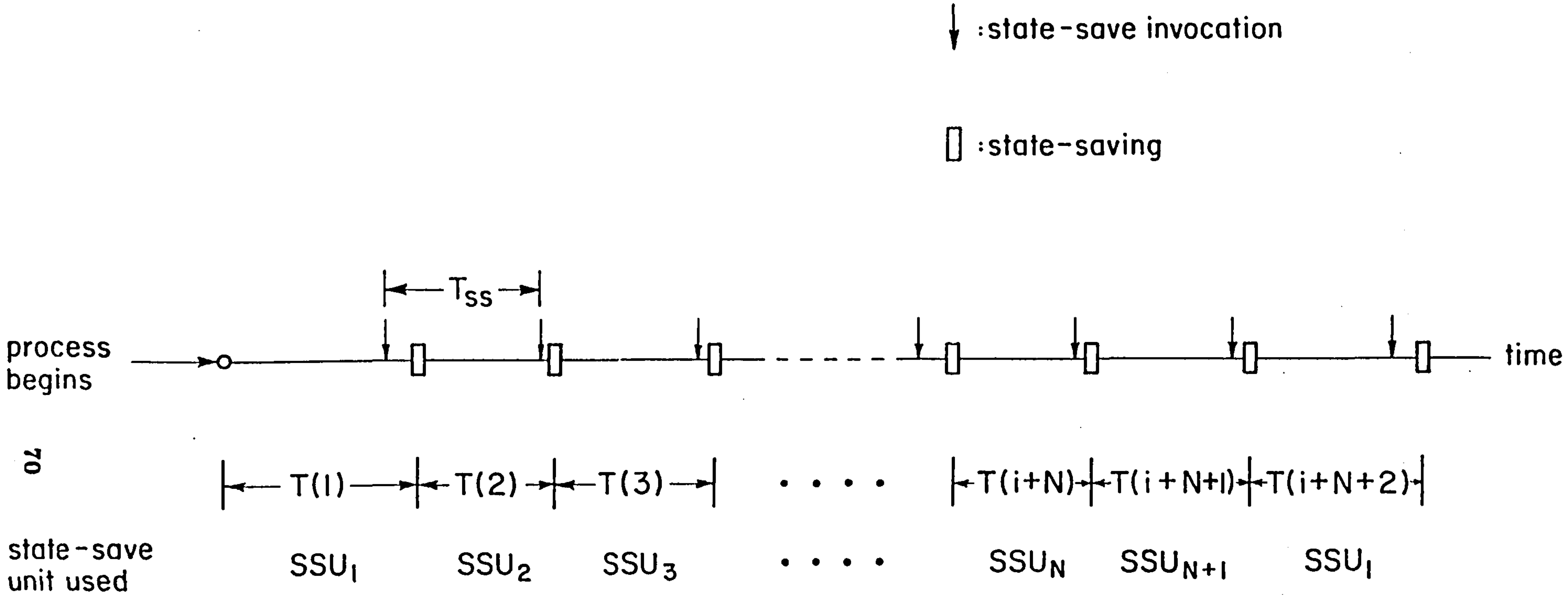


Figure 5.3 State-save Operations in One Module.

problem has been discussed in [26]. 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.

5.1.2. Rollback Recovery Operations of a Task

Suppose a task is partitioned and then allocated to M modules (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. 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 multi-step 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 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, 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.

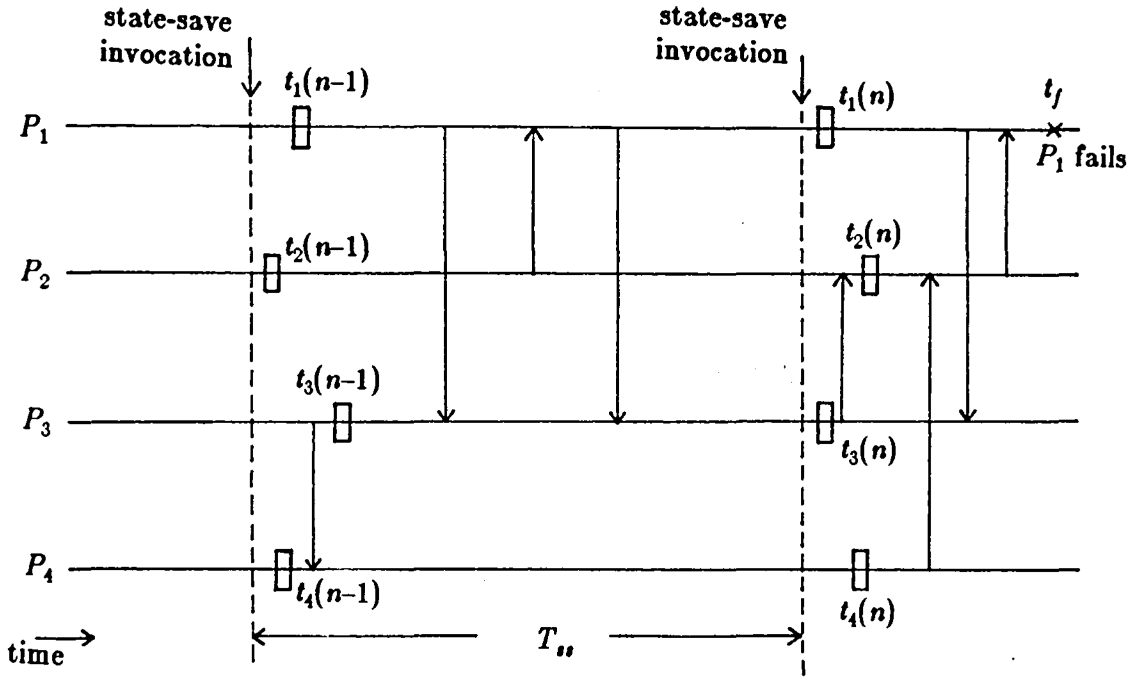
1. Stop all PMs that are executing processes of the task in question.
2. Make a decision as to rollback propagation.
3. Resume execution of the processes that are not affected by rollback propagation.
4. Find free module to replace the failed one.
5. Transfer the process or data in the failed module 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 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 PMs, which are still queued in the access queues, are cancelled.

5.2. 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 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 multi-step rollback.

5.2.1. Rollback Propagation and Multi-Step Rollback

In general rollback propagation can not be avoided if the processes interact with each other arbitrarily. For the multiprocessor organization in the previous section, a process is allocated to one PM and/or several CMs 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 n-th state-save interval of P_i be $T_i(n)$ and the beginning moment of $T_i(n)$ where P_i saved its state bt $t_i(n)$. An example is presented in Figure 5.4, 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 [80] are used, the state of



(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)

Figure 5.4 An Example of Rollback Propagation and Multi-step Rollback.

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 and P_2 to their most recently saved state still cannot provide a consistent task state. The reason that a rollback of cooperating processes can not 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 $j=1,2...,M$ 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 Figure 5.5 P_i rolls back to $t_i(k)$ because of failure or rollback propagation from another process. 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 $t_j(k\text{-}1)$ are restorable for P_i and P_j respectively, while in case (d), the states at $t_i(k\text{-}1)$ and $t_j(k)$ become restorable.

The necessary condition for recovering a task TK, where $TK=\{P_i|i=1,2,...,M\}$, 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.

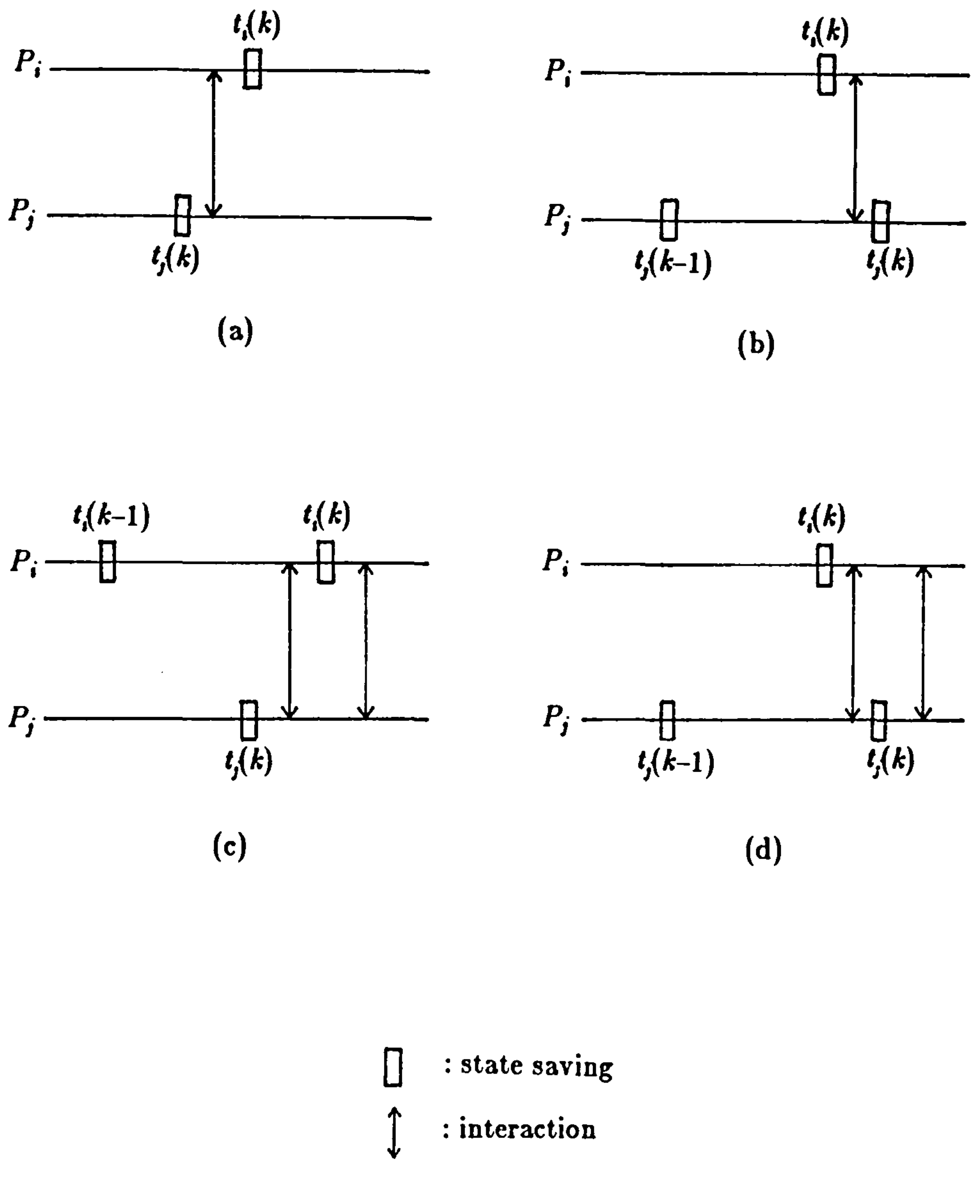


Figure 5.5 Interaction Patterns Related to Rollback Propagation.

5.2.2. 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 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 $T_i(k-n+1)$, $(n=1,2,3,\ldots,N)$, 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 $RB_i(k)$, $k=1,2,\ldots,N$, 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 $j=1,2,\ldots,M$

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

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

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 $j=1,2,\ldots,M$. 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 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 Figure 5.4, where Figure 5.4(a) describes memory references, Figure 5.4(b) is the current contents of KC and KP matrices, and Figure 5.4(c) is the result of rollback propagation.

5.2.3. The Evaluation of Multi-Step Rollback

If module i fails at time t_f during the k-th 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 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 the states recorded in SSUs are exhausted. In this section a probability model is derived to evaluate the coverage of the multi-step rollback recovery which indicates the effectiveness of the present fault-tolerant mechanism. Recall that a 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. 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. $b_{ij1}=b_{ij2}=....=b_{ijN}=b_{ij}$. 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., $b_{ij}=\text{Prob}((B_{ij}\bigcap B_{jj})\bigcup(B_{ii}\bigcap B_{ji}))$, 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 $[e_{ij}]$, $i,j=1,2,...,M$, 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 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 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 by processes can be divided into different phases each of which has a constant reference rate [7,47]. Thus, if N_t is moderately large, u_{ijk} could be assumed to be a constant during the k-th 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:

$$f_{ij} = f_{ji} = g_{ij}+g_{ji}-g_{ij}g_{ji} \qquad (5.1)$$

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:

$$\sum_{k=1}^{N_i} u_{ijk} = (T_{ef} a_{ij}) / ((T_{ss} - T_{sv}) \sum_{m=1}^{M} a_{im} e_{im}) \tag{5.2}$$

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

$$\frac{1}{e_{ij}} \geq (u_{ijk})_{\max} \geq u_{ijk} \geq 0 \tag{5.3}$$

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 $\max(f_{ij})$, occurs when $u_{ij,1} = u_{ij,2} = \ldots = u_{ij,N_i}$.

(2). The minimum value of f_{ij}, denoted as $\min(f_{ij})$ occurs when there are

(i) h intervals $\lceil h = e_{ij} T_{ef} a_{ij} / ((T_{ss} - T_{sv}) \sum_{m=1}^{M} a_{im} e_{im}) \rceil$ in which $u_{ijk} = 1/e_{ij}$,

(ii) $(N_i - h - 1)$ 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 Figure 5.6 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 [57] can be used to solve for r_{ij}:

$$r_{ij} = \bigcup_{q} (D_{ij,q}) \tag{5.4}$$

where $D_{ij,q}$ is the probability that the q-th path from node i to node j is connected and

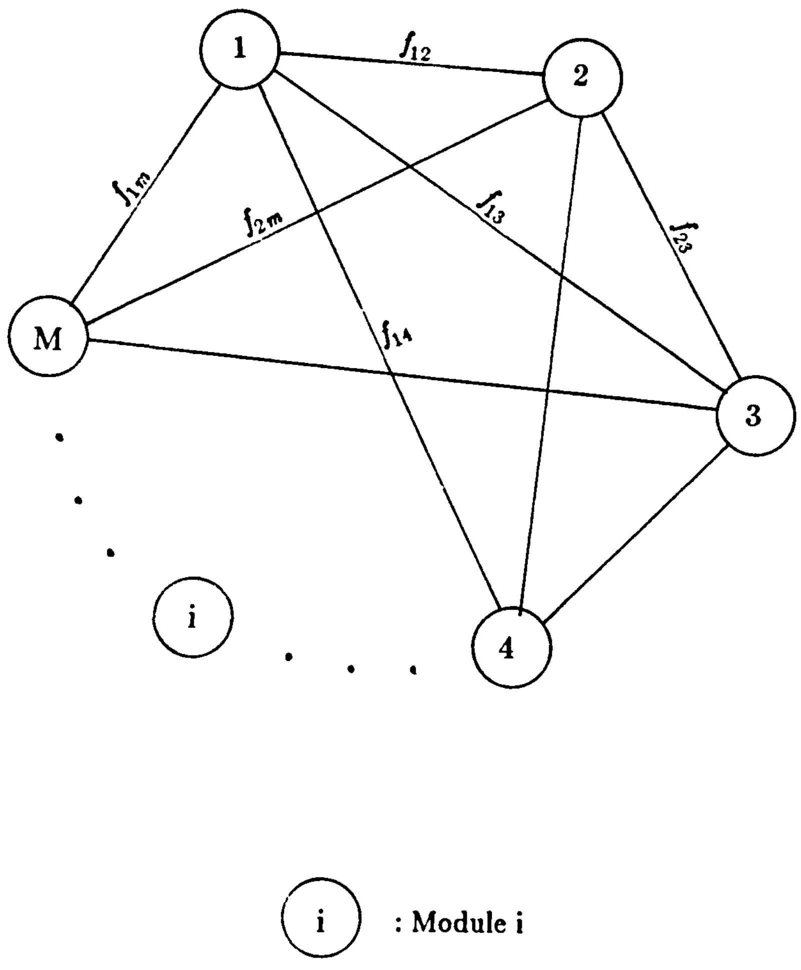


Figure 5.6 The Rollback Propagation Network.

$\bigcup$ 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

$$C(1)=(1/M)\sum_{i=1}^{M}\prod_{j=1}^{M}(1-r_{ij}(1-\sum_{k=1}^{M}b_{jk})) \tag{5.5}$$

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

$$C(h)=C(1)(1-C(h\text{-}1))+C(h\text{-}1) \tag{5.6}$$

The coverage of the multi-step 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* [72]. The numerical results are presented in Table 1 and are also plotted in Figure 5.7. These results include three cases: the best coverage computed from $\min(f_{ij})$ for different values of N_t, and the worst coverage computed from $\max(f_{ij})$. 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. On the other hand, this conclusion must be interpreted in the context of access localities; the number of SSUs 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

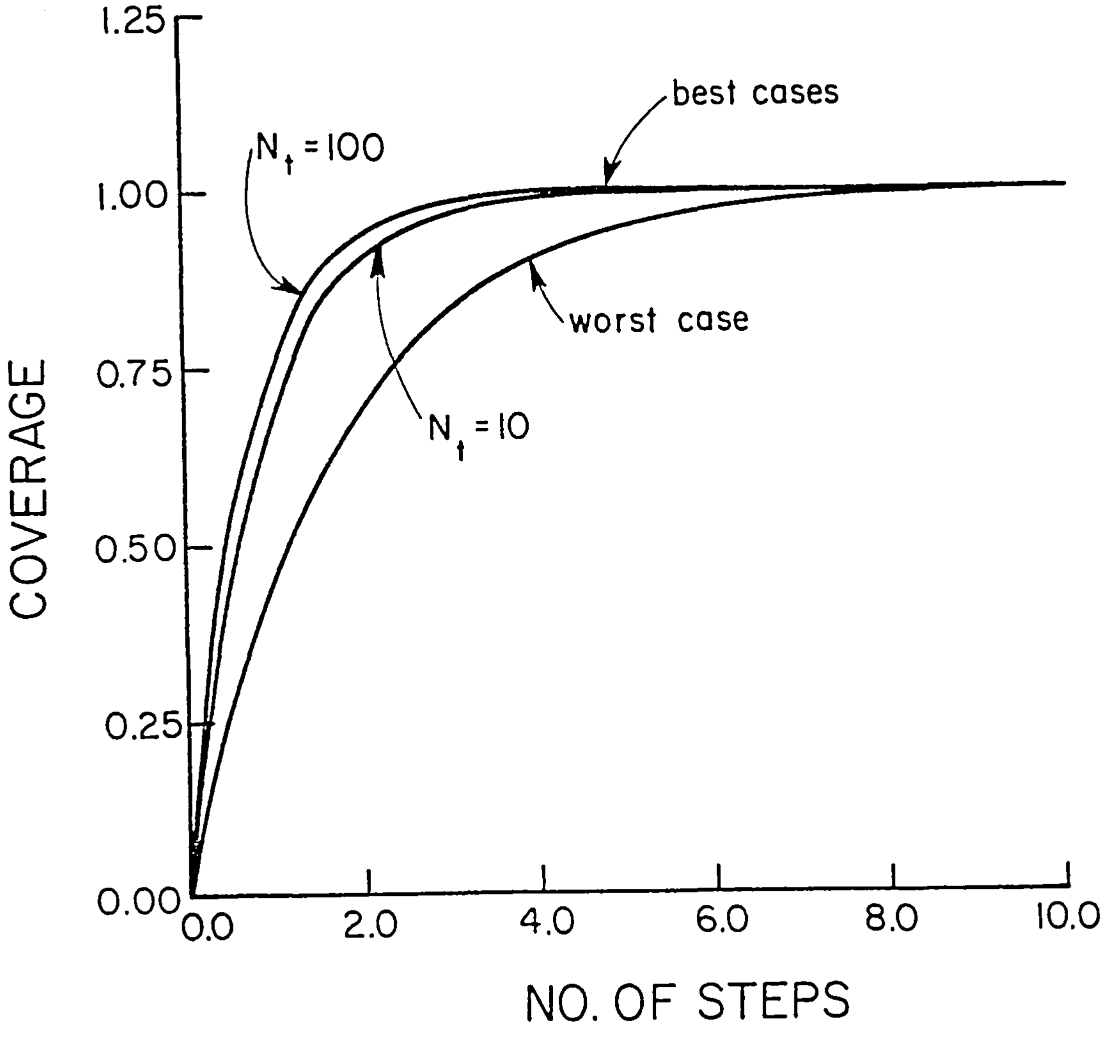


Figure 5.7 Rollback Coverage vs. No. of Rollback Steps.

be understood as an inherent problem associated with multiprocessors rather than with the present fault-tolerant mechanism (see [64] for the dependence of multiprocessor performance on access localities).

5.3. The Performance of Rollback Recovery Mechanism

Several methods for analyzing the rollback recovery system have been proposed [12-14,19-20,30-31,83]. 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 [16]. 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 multi-step rollback and the risk of restart for the rollback recovery mechanism.

5.3.1. 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.
$T^j_{roll,m}$:	The time lost due to the j-th rollback in module m which consists of the set up time for resumption, t_{sb}, 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 two consecutive rollbacks (restarts).
$C(i)$:	The accumulated coverage of rollback recovery from a single step to i steps. This value is calculated by the Equations 5.5 and 5.6 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.
$\Phi_t(s), \Phi_{real}(s)$:	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 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 be delayed 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. First, $T^j_{roll,m}$ 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

$\sum_{m=1}^{M} T^{i}_{roll,m}$. 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 $\max\{T^{i}_{roll,m}\}$ 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 [17]. 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. From the definitions of P_s, P_b, and $P_{st}(h)$, we have $P_s=1-C(N)$ when each module has $(N+1)$ 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 $P_{st}(1)=C(1)/P_b$. 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.

5.3.2. The Performance Model

The total task execution time, T_t, can be divided into several phases as shown in Figure 5.8. The last phase is always ended with the completion of task. Other phases are

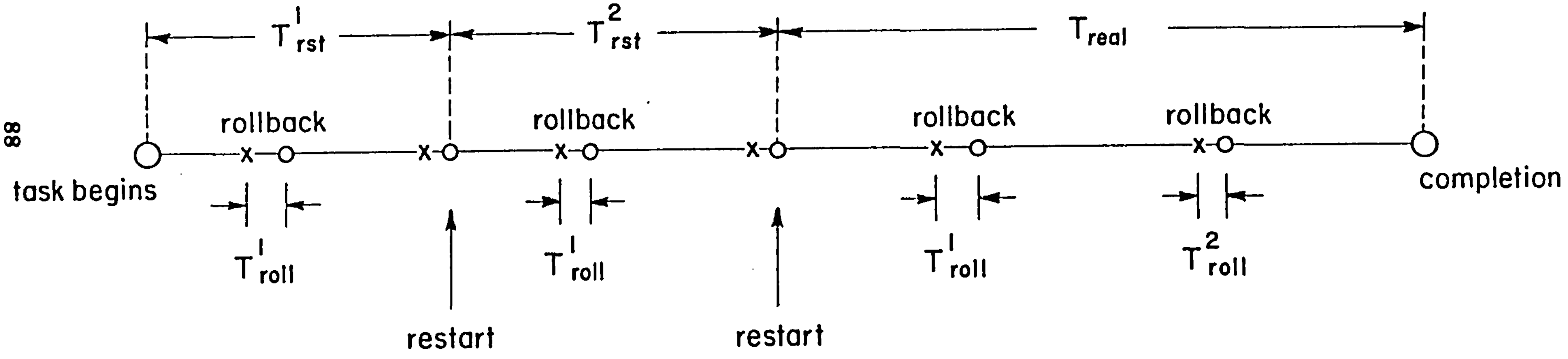


Figure 5.8 Task Execution Phases.

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 are accũmulated 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_{rst}^i for i=1,2,..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:

$$Z = \sum_{h=1}^{N} e^{-h\lambda_1 T_{ss}} P_{st}(h) \tag{5.7}$$

$$H(t,k) = \sum_{i=0}^{k} \binom{k}{i} (1-Z)^i (Z)^{k-i} G_{k-i}(t) \tag{5.8}$$

where $G_{k-i}(t)$ is the $(k-i)$-th order gamma distribution function with parameter λ_5 for $(k-i)>0$, and $G_0=1$. In Appendix B, we show that the distribution function of the accumulated effective computation after k rollbacks is $Prob(TE_k \le t) = H(t,k)$. Therefore the probability of k rollbacks during the time interval T_{real}, $P_r(k)$, is given by

$$\begin{aligned} P_r(k) &= P(TE_{k+1} > T_{ef}) - P(TE_k > T_{ef}) \\ &= H(T_{ef},k) - H(T_{ef},k+1) \end{aligned} \tag{5.9}$$

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 functions of $P_r(k)$ and $P_s(h)$, we get the characteristic function of T_{real} which is given below:

$$\Phi_{real}(s) = e^{-sT_{el}} Z_r(e^{-st_{si}} Z_s(e^{-sT_{ss}})) \tag{5.10}$$

From Figure 5.8, 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 C is given in the following:

$$\Phi_t(s) = \sum_{n=0}^{\infty} e^{-nst_{su}} \left(\frac{\lambda_s}{\lambda_s+s}\right)^n \left\{ \sum_{j=0}^{n} \binom{n}{j} (-1)^j \Phi_{real}((j+1)(\lambda_s+s)) \right\} \tag{5.11}$$

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 $\Phi_{real}(s)$ becomes $e^{-sT_{el}}$. The result obtained from the above equation is the same as that in [16]. The mean and variance of the total execution time can be obtained from $-\frac{\partial \Phi_t(s)}{\partial s} |_{s=0}$ and $\frac{\partial^2 \Phi_t(s)}{\partial s^2} |_{s=0}$. In Figure 5.9, the mean execution time for the example in the previous section 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 [15], the overhead of generating recovery blocks can be reduced with a completely self-checking mechanism. Figure 5.10 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.

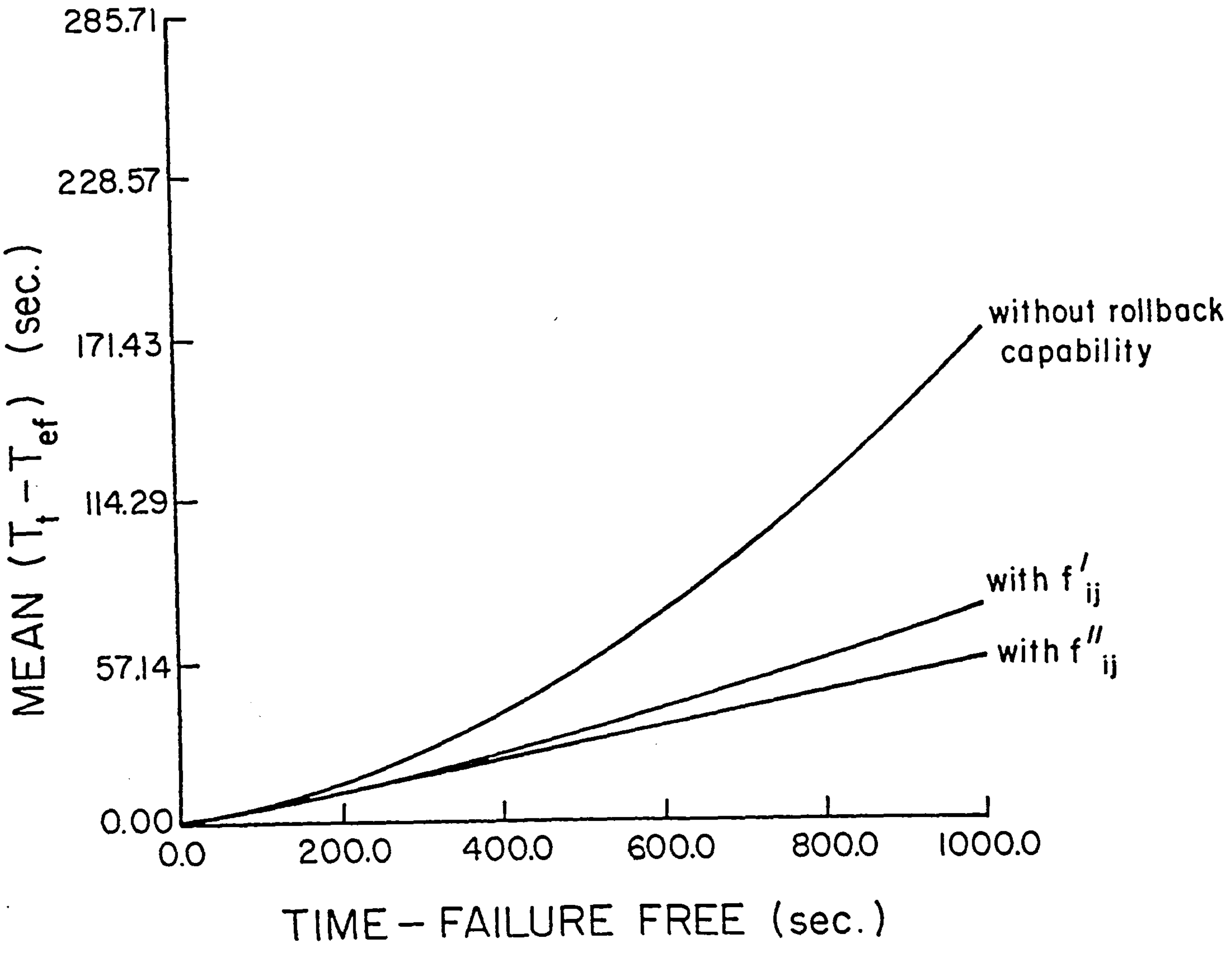


Figure 5.9 Mean Time-Overhead vs. Error-Free Execution Time.

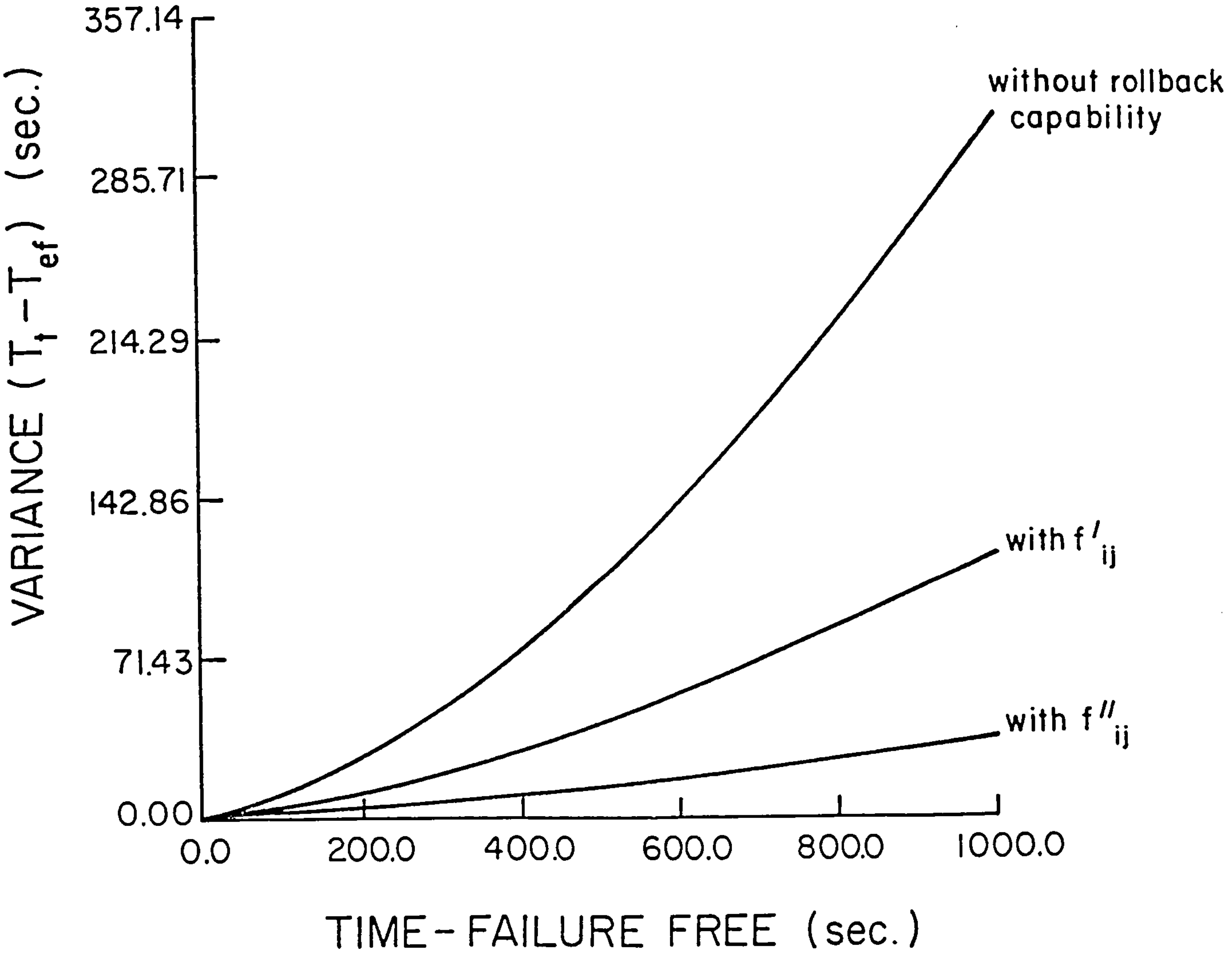


Figure 5.10 Variance Time-Overhead vs. Error-Free Execution Time.

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. Figure 5.7 indicates that 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 5.11. The Figure 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.

6. CONCLUDING REMARKS

In this report, we have presented first a general model for the error detection process and then applied it for estimating two important performance-related parameters of fault-tolerant computers. These two are not usually included in the traditional reliability models. The first parameter, the probability of having an unreliable result, indicates the degree of lack of confidence in computation results. Suspicion in the computation results is wholly due to the imperfect nature of error detection. Unfortunately, such

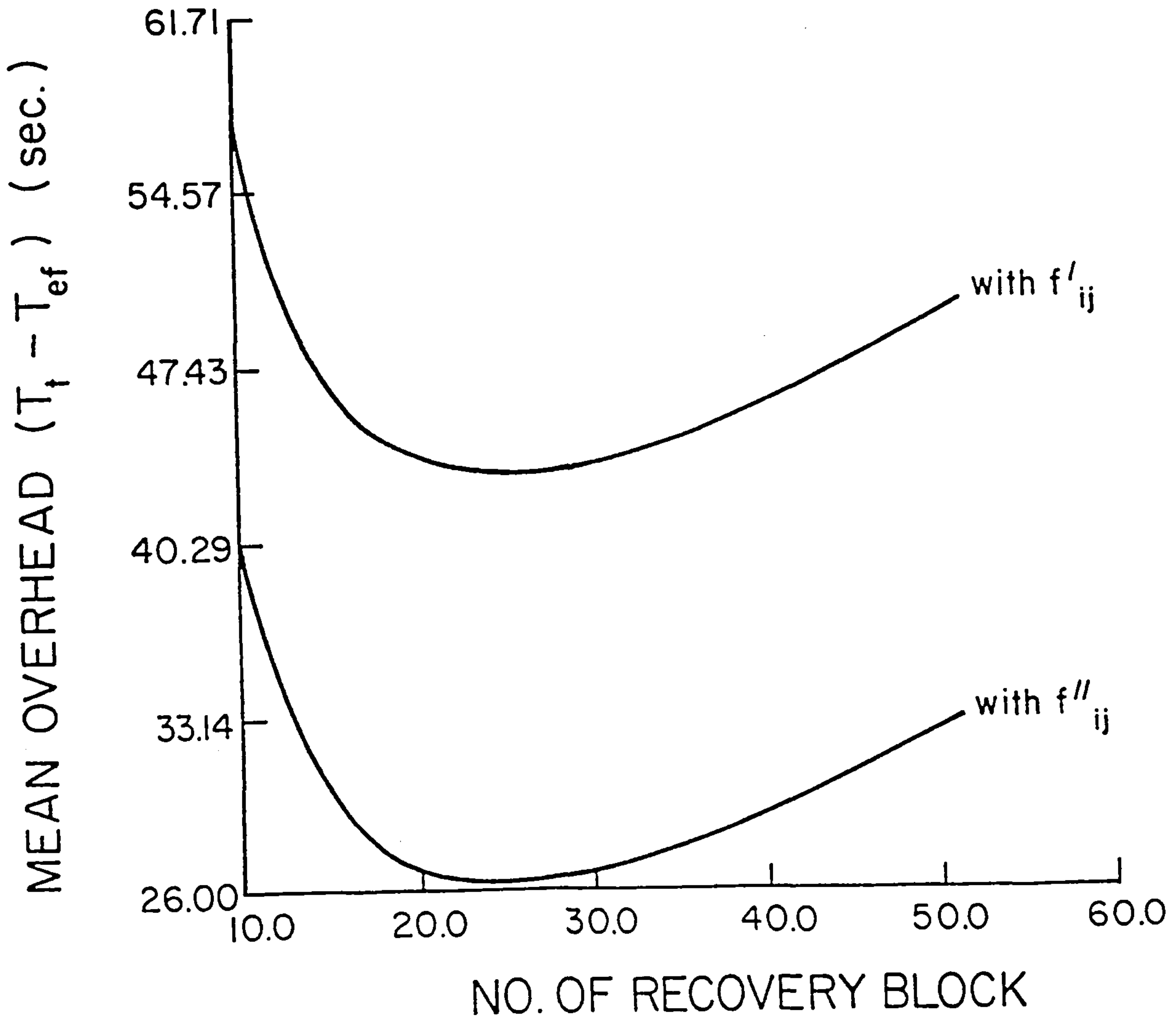


Figure 5.11 Mean Time-Overhead vs. Total Number of Recovery Blocks for a Given Task.

imperfection cannot be eliminated completely from any practical error detection mechanism. For the second parameter, we take a more detailed account of the computation loss and execution cost resulting from the occurrence of error, its detection and subsequent recovery. Since most reliable systems either include error recovery mechanisms with unknown overheads or may suffer from an erroneous output, any reliability analysis has to quantify the above overheads and uncertainty and also has to provide a good method for estimating these quantities.

Meanwhile, we have outlined a feasible design space in which a proper combination of different imperfect detection mechanisms needed to meet the specifications is indicated. Since the determination of a feasible design space of detection mechanisms must integrate the recovery methods used in the system, we also briefly presented the performance of various recovery methods. Unfortunately, we cannot determine an optimal trade-off between various detection mechanisms because of the insufficient understanding of the function level detection and the lack of relations between hardware costs and the signal level detection capability. Further research is needed along these directions, especially, experiments of program behavior under erroneous conditions and the design of function level detection mechanisms.

In the second part, the rollabck recovery strategies are examined. The software rollback recovery and hardware recovery blocks have been studied extensively. The distribution of the interval between two successive recovery lines, which is the upper bound of recovery overhead in software rollback recovery, are used to represent the performance of different strategies in software rollback recovery. For hardware recovery blocks, the distribution of task completion time has been formulated. With the combination of the model of detection mechanisms and these quantitative evaluations of recovery methods, the execution cost and the probability of failure can be obtained which will reflect the effects of fault-tolerance techniques on the system's normal operations.

After these studies in error detection and recovery, it is important to investigate the reconfiguration of system after a faulty unit is detected. Since fault-tolerance is grounded on redundancy, the management of redundancy will certainly effect the whole error handling operation and system performance. One simple example is the assignment of redundant modules for a single task in such a way that a certain number of errors can be tolerated or detected through mutual consistency, thereby blocking error propagation. Usually, system reconfiguration has two objectives; one is to enhance the computation capability, the other is meant to improve reliability. The former emphasizes the effective utilization of system resources. and the latter is to establish adequate redundancy for error detection and recovery. When the system has enough resources for both purposes, reconfiguration becomes trivial because no competition for resource exists. For an application with a long life cycle, however, this may not apply. In such case, the management of redundancy becomes essential.

Also of interest would be an analysis that allows the treatment of simultaneously extant multiple faults. Since most faults in the system are likely to be transient or intermittent, there is the possibility that the fault-latency is large. Note that the retry recovery is applied as a temporary remedy when an intermittent fault becomes benign shortly after its presence. This intermittent fault may still exist but is inactive. These would cause faults to accumulate in **F** and/or **FB**, thus making the entire system vulnerable to any environmental or other events that might activate them. The difficulty with any such model is likely to be a considerable expansion in the number of states, thus increasing the model complexity. It is likely that in any realistic analysis, some means must be sought to reduce the state-space size by approximating suitably. The approach used in CARE III [71], where states are aggregated and the state transition rates are separately determined, may be an appropriate attempt although the model is forced to be non-homogeneous. The nature of such approximations is a matter for further research.

REFERENCES

[1] Agrawal, V. D., "An Information Theoretic Approach to Digital Fault Testing", *IEEE Trans. on Computers,* Vol. C-30, No. 8, August 1981, pp. 582-587.

[2] Anderson, T. and Lee, P. A., *Fault Tolerance: Principle and Practice,* Prentice-Hall International, Inc., 1981.

[3] Andrews, D. M., "Using Executable Assertions for Testing and Fault Tolerance," *Pro. of 9-th Int'l Conf. on Fault-Tolerant Computing,* 1979, pp. 102-105.

[4] Baccelli, F. "Analysis of a Service Facility with Periodic Checkpointing", *Acta Information,* Vol. 15, 1981, pp. 67-81.

[5] Ball, H. and Hardie, F., "Effects and Detection of Intermittent Failures in Digital Systems," *AFIP Conf. Proc.,* Fall 1969, pp. 229-235.

[6] Barigazz, G. and Strigini, L., "Application-Transparent Setting of Recovery Points," *Proc. of 13-th Int'l Conf. on Fault-Tolerant Computing,* 1983, pp. 48-55.

[7] Batson, A. P., "Program Behavior at the Symbolic Level," *Computer,* Nov. 1976, pp. 21-26.

[8] Bavuso, S. J. et al., "Latent Fault Modeling and Measurement Methodology for Application to Digital Flight Control", *Advanced Flight Control Symposium,* USAF Academy, 1981.

[9] Ben Romdhane, M. and Courtois, B., "Error Confinement / Data Recovery in Distributed Ssytem," *Proc. of Reliability in Distributed Software and Data Base System,* 1982, pp. 11-18.

[10] Bossen, D. C. and Hasio, M. Y., "Model for Transient and Permanent Error-Detection and Fault-Isolation Coverage," *IBM J. Res. Develop.* Vol. 26, No. 1, Jan. 1982, pp. 67-77.

[11] Breuer, M. T., "Testing for Intermittent Faults in Digital Circuits," *IEEE Trans. on Computers,* Vol. C-22, No. 3, March 1973, pp. 241-246.

[12] Brodetskiy, G. L., "A Problem of Periodic Storing of Results," *Cybernetics,* Vol. 14, No. 3, pp. May-June, 1978, pp. 390-395.

[13] Brodetskiy, G. L.,"Effectiveness of Storage of Intermediate Results in Systems with Failures that Destroy Information," *Engineering Cybernetic,* Vol. 16, No. 6, Nov.-Dec. 1978, pp. 75-81.

[14] Brodetskiy, G. L., "Periodic Dumping of Intermediate Results in Systems with storage-Destructive Failures," *Cybernetics,* Vol. 15, No. 5, Sept.-Oct. 1979, pp. 685-689.

[15] Carter, W. C. 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.

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

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

[18] Cinlar, E., *Introduction to Stochastic Processes,* Prentice-Hall, Englewood Cliffs, N.J., 1975.

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

[20] Chandy, K. M., Browne, J. C., Dissly, C. W. and Uhrig, W. R., "Analytic Models for Rollback and Recovery Strategies in Data Base Systems," *IEEE Trans. of Software Eng.,* Vol. SE-1, no. 1, March 1975, pp. 100-110.

[21] Chandy, K. M., "A Survey of Analytic Models of Rollback and Recovery Strategies," *Computer,* Vol. 8, No. 5, May, 1975, pp. 40-47.

[22] Courtois, B.,"Some Results about the Efficiency of Simple Mechanisms for the Detection of Microcomputer Malfunction", *Proc. of the 9th Annual Int'l Symp. on Fault-Tolerant Computing,* 1979, pp. 71-74.

[23] Courtois, B., "A Methodology for On-line Testing on Microprocessors", *Proc. of the 11th Annual Int'l Symp. on Fault-Tolerant Computing,* 1981, pp. 272-274.

[24] Courtois, B., "Performance Modelling of Partially Self Checking Systems", *Proc. of 12-th Int'l Conf. on Fault-Tolerant Computing,* 1982, pp.140-146.

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

[26] Feridun, A. M. and Shin, K. G., "A Fault-Tolerant Multiprocessor System with Rollback Recovery Capabilities," *Proc. 2nd Int'l Conf. on Distributed Computing System,* April 1981, pp. 283-298.

[27] Ferran, G., "Distributed Checkpointing in a Distributed Data Management System," *Proc. of Real Time Systems Symp.,* 1981, pp. 43-49.

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

[29] Gay, F. A., "Reliablity of Partially Self-Checking Circuits," *Proc. of 7-th Int'l Conf. on Fault-Tolerant Computing,* 1977, pp. 135-142.

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

[31] Gelenbe, E., "On the Optimum Checkpoint Interval," *JACM,* Vol. 26, No. 2, April 1979, pp. 259-270.

[32] Gray, J. N., "Notes on Database Operating Systems, " *Operating Systems: A advanced course,* edited by R. Bayer, et al., Springer-Verlag, 1979, pp.393-481.

[33] Gunther, N. L. and Carter, W. C., "Remarks on the Probability of Detecting Faults", *Proc. of th 10th Annual Int'l Symp. on Fault-Tolerant Computing,* 1980, pp. 213-215.

[34] Hopkins, A. L., Smith, T. B. and Lala, J. H., "FTMP - A Highly Reliable Fault-Tolerant Multiprocessor for Aircraft," *Proceedings of the IEEE,* Vol. 66, No. 10, Oct. 1978, pp. 1221-1240.

[35] Horning, J. J., Lauer, H. C., Melliar-Smith, P. M., and Randell, B., "A Program Structure for Error Detection and Recovery," *Lecture Notes in Computer Science: Operating Systems,* Springer-Verlag, 1974, pp. 171-187.

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

[37] Kim, K. H., "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.

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

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

[40] Kim, K. H., "Approaches to Mechanizations of the Conversation Scheme Based on Monitors," *IEEE Trans. on Software Eng.,* Vol. SE-8, No.3, May 1982, pp. 189-197.

[41] Kohler, W. H., "A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems," *Computing Surveys,* Vol. 13, No. 2, June 1981, pp. 149-183.

[42] Koren, I. and Berg, M., "A Module Replacement Policy for Dynamic Redundancy Fault-Tolerant Computing Systems," *Proc. of 11-th Int'l Conf. on Fault-Tolerant Computing,* 1981, pp. 90-95.

[43] Krishna, C. M. and Shin, K. G., "Performance Measures for Multiprocessor Controllers," *Performance '83: Ninth Int'l Symp. Comp. Perf., Meas., and Eval.,* pp. 229-250.

[44] Lee, Y. H. and Shin, K. G., "Rollback Propagation Detection and Performance Evaluation of FTM^{2}P--A Fault-Tolerant Multiprocessor," *Proc. of the 9th Annual Symp. on Comp. Arch.,* 1982, pp. 171-180.

[45] Lee, Y. H. and Shin, K. G., "Design and Evaluation of a Fault-Tolerant Multiprocessor Using Hardware Recovery Blocks," To appear at *IEEE Trans. on Computer.*

[46] Liaw, C. C., Su, S. Y. H. and Malaiya, Y. K., "Test-Experiments for Detection and Location of Intermittent Faults in Sequential Circuits," *IEEE Trans. on Computers,* Vol. C-30, No. 12, Dec. 1981, pp. 989-995.

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

[48] Malaiya, Y. K. and Su, S. Y. H., "Reliability Measure of Hardware Redundancy Fault-Tolerant Digital Systems with Intermittent Faults", *IEEE Trans. on Computers,* Vol. C-30, No. 8, August 1981, pp. 600-604.

[49] Malaiya, Y. K. and Su, S. Y. H., "Analysis of an Important Class of Non-Markov System," *IEEE Trans. on Reliability,* Vol. R-31, No. 1, April 1982, pp. 64-67.

[50] Marchal, P. and Courtois, B., "On Detecting The Hardware Failures Disrupting Programs in Microprocessors", *Proc. of 12-th Int'l Conf. on Fault-Tolerant Computing,* 1982, pp. 249-256.

[51] McGough, J. G. and Swern, F. L., "Measurement of Fault Latancy in a Digital Avionic Mini Processor," *NASA Contractor Report 3462,* Oct. 1981.

[52] McGough, J. G. and Swern, F. L., "Measurement of Fault Latency in a Digital Avionic Mini Processes," *NASA Contractor Report 3651,* Jan. 1983.

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

[54] Merlin, P. M. and B. Randell, B., "State Restoration in Distributed Systems," *Proc. of 8-th Int'l Conf. on Fault-Tolerant Computing,* 1978, pp. 129-134.

[55] Ng, T. W. and Avizienis, A. A., "A Unified Reliability Model for Fault-Tolerant Computers," *IEEE Trans. on Computers,* Vol. C-29, No. 11, Nov. 1980, pp. 1002-1011.

[56] Osden, S.,"The DC-9-80 Digital Flight Guidance System's Monitoring Techniques", *Proc. of the AIAA Guidance and Control Conf.,* 1979, pp. 64-79.

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

[58] Randell, B., "System Structure for Software Fault Tolerance," *IEEE Trans. on Software Eng.,* Vol. SE-1, No. 2, June 1975, pp. 220-232.

[59] Randell, B., Lee, P. A. and Treleaven, P. C., "Reliability issues in computing system design," *Computing Surveys,* Vol. 10, No. 2, June 1978, pp. 123-165.

[60] Russell, D. L., "Process Backup in Producer-Consumer Systems," *Proc. of 6th ACM Symposium on Operating System Principles,* Nov. 1977, pp. 151-157.

[61] Russell, D. L., "State Restoration in Systems of Communicating Processes," *IEEE Trans. on Software Eng.,* Vol. SE-6, No. 2, March 1980, pp. 183-194.

[62] Shedletsky, J. J., "Random Testing: Practicality vs. Verified Effectiveness", *Proc. of the 7th Annual Int'l Symp. on Fault-Tolerant Computing,* 1977, pp. 175-179.

[63] Shedletsky, J. J., "A Rollback Interval for Networks with an Imperfect Self-Checking Property", *IEEE Trans. on Computers,* Vol. C-27, No. 6, June 1978, pp.272-274.

[64] Shin, K. G., Lee, Y. H. and Sasidhar, J., "Design of HM^{2}p--A Hierarchical Multimicroprocessor for General-Purpose Applications", *IEEE Trans. on Computers,*

Vol. C-31, No. 11, Nov. 1982, pp. 1045-1053.

[65] Shin, K. G., Krishna, C. M. and Lee, Y. H., "The Applications to the Aircraft Landing Problems of an Uniform Method for Evaluation Real-Time Controllers," *Proc. of IEEE Real-Time System Symp.* Dec. 1982, pp. 242-256.

[66] Shin, K. G. and Lee, Y. H., "Analysis of the Impact of Error Detection on Computer Performance," *Proc. of 13-th Int'l Conf. on Fault-Tolerant Computing,* June 1983, pp. 356-359.

[67] Shin, K. G. and Lee, Y. H., "Analysis of Backward Error Recovery for Concurrent Processes with Recovery Blocks," *Proc. of 1983 Int'l Conf on Parallel Processing,* Aug. 1983, pp. 362-366.

[68] Siewiorek, D. P. and Swarz, R. S., *The Theory and Practice of Reliable System Design,* Digital Press, 1982.

[69] Simoncini, L. and Friedman, A. D., "Incomplete Fault Coverage in Modular Multiprocessor Systems," *ACM Proc. of 1978 Annual Conf.,* 1978, pp. 210-215.

[70] Stiffler, J. J., "Robust Detection of Intermittent Faults," *Proc. of 10-th Int'l Conf. on Fault-Tolerant Computing,* 1980, pp. 216-218.

[71] Stiffler, J. J. and Bryant, L. A., "CARE III Phase Report - Mathematical Description," *NASA Report,* No. 3566, Nov. 1982.

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

[73] Tantawi, A. N. and Ruschitzka, M., "Performance Analysis of Checkpointing Strategies", *IBM Research Report,* RC 9999(43430), Jan. 1983.

[74] Tasar, O. and Tasar, V., "A Study of Intermittent Faults in Digit Computers," *AFIP Conf. Proc.,* 1977, pp. 807-811.

[75] Tasar, V., "Analysis of Fault-Detection Coverage of a Self-Test Software Program", *Proc. of the 8th Annual Int'l Symp. on Fault-Tolerant Computing,* 1978, pp. 65-74.

[76] Troy, R., "Dynamic Reconfiguration: An Algorithm and its Efficiency Evaluation," *Proc. of 7-th Int'l Conf. on Fault-Tolerant Computing,* 1977, pp. 44-49.

[77] Trivedi, K. S. and Geist, R. M., "A Tutorial on the CARE III Approach to Reliability Modeling", *NASA Contract Report 3488,* 1981.

[78] Tsuruoka, K., Kaneko, A. and Nishihara, Y., "Dynamic recovery schemes for distributed processes," *Proc. of Reliability in Distributed Software and Database Systems,* 1981, pp. 124-130.

[79] Wimmergren, A. L., "Verification of a Fault Tolerant Multi-Processor Architecture," *CSDL-T-782,* The Charles Stark Draper Lab., May 1982.

[80] Wood, W. G.,"A Decentralized Recovery Control Protocol," *Proc. of 11-th Int'l Conf. on Fault-Tolerant Computing,* 1981, pp. 159-164.

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

APPENDIX A

DENSITY FUNCTIONS OF TASK EXECUTION TIME

The density functions of task execution time with error occurrence due to three different types (i.e. transient, intermittent, and permanent) faults are expressed as follows:

$$f_r(t|F_1,T) = \{1 - \pi_6(T|F_1) - \pi_7(T|F_1)\,(1-\rho_1)\}\delta_T(t) + \pi_6(T|F_1)f_{rbs,1}(t,0)$$
$$+\ \pi_7(T|F_1)(1-\rho_1)[e^{-rt_r}\delta_T(t-t_r) + (1-e^{-rt_r})f_{rbs,1}(t,1)]$$

$$f_r(t|F_2,T) = \{1 - \pi_6(T|F_2) - \pi_7(T|F_2)(1-\rho_1)\}\delta_T(t) + \pi_6(T|F_2)f_{rbs,2}(t,0)$$
$$+\ \pi_7(T|F_2)(1-\rho_1)[\sum_{n=1}^{\infty}(1-\delta_2)^{n-1}\delta_2 f_{rbs,2}(t,n)]$$

$$f_r(t|F_3,T) = \{1 - \pi_6(T|F_3) - \pi_7(T|F_3)(1-\rho_1)\}\delta_T(t) + \pi_6(T|F_3)f_{rbs,2}(t,0)$$
$$+\ \pi_7(T|F_3)(1-\rho_1)f_{rbs,2}(t,1)$$

where $f_{rbs,j}(t,n)$ is the density function of the time loss in recovery from an error induced by F_j after n unsuccessful retries, which is given as follows:

$$f_{rbs,j}(t,n) = (1-p_{sv})p_{46}(t-nt_r-t_b|F_j)\frac{1}{t_{ch}}\,\{u_T(t-nt_r)-u_T(t-nt_r-t_{ch})\}$$
$$+(p_{sv}+(1-p_{sv})\{1-\int_0^{t_{ch}}\frac{p_{46}(t|F_j)}{t_{ch}}\,dt\}f^T_{start,j}(t-nt_r)$$

where $\delta_T=\delta(t-T)$, $u_T=u(t-T-t_b)$, $f^T_{start,j}=f_{start,j}(t-T)$, and $\delta(t)$ and $u(t)$ are impulse and step functions.

APPENDIX B

CALCULATION OF THE PROBABILITY OF HAVING k ROLLBACK DURING THE DURATION T_{real}

From the definition of $P_{st}(h)$ in Chapter 5, 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 j-th 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 k-th rollback, TE_k, is given by

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

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 $(X^j_r - T^j_{roll})=0$ is $\sum_{h=1}^{N} P_{st}(h)(1-e^{\lambda_b h T_{ss}})$. The density function of $(X^j_r - T^j_{roll})$ becomes

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

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

$$f_\alpha(t) = (1-Z)\delta(t) + e^{-\lambda_b t} Z$$

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

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

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

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

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

APPENDIX C

CALCULATION OF THE CHARACTERISTIC FUNCTION OF TOTAL EXECUTION TIME, $\Phi_t(s)$

From Figure 5.8, the total execution time T_t is the sum of T_{real} and T_{rst}, where $T_{rst}=\sum_{i=1}^{n} T_{rst}^{i}$ when there are n restarts. Given the conditional probability of T_t, we can write the following equation:

$$E(T_t|T_{real}) = T_{real}+E(T_{rst}|T_{real})$$

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

$$f_{rst|T_{real}}(t+t_{su}) = \frac{\lambda_s e^{-\lambda_s t}}{1-e^{-\lambda_s T_{real}}} \qquad \text{for } 0\leq t\leq T_{real}$$

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

$$P_{rs|T_{real}}(n) = (e^{-\lambda_s T_{real}})(1-e^{-\lambda_s T_{real}})^n$$

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

$$\Phi_{t|T_{real}}(s) = e^{-sT_{real}} \sum_{n=0}^{\infty} P_{rs|T_{real}}(n) (\Phi_{rst|T_{real}}(s))^n$$

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 $f_{rst|T_{real}}(t)$. By substituting $P_{rs|T_{real}}(n)$ and $\Phi_{rst|T_{real}}(s)$ into the above equation and integrating with the density function of T_{real}, the characteristic function of T_t is obtained as the Eq. (5.11) in Chapter 5.

1. Report No. NASA CR-172571	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle INTEGRATED ANALYSIS OF ERROR DETECTION AND RECOVERY		5. Report Date October 1985
		6. Performing Organization Code
7. Author(s) Kang G. Shin and Yann-Hang Lee		8. Performing Organization Report No.
		10. Work Unit No.
9. Performing Organization Name and Address Department of Electrical and Computer Engineering The University of Michigan Ann Arbor, MI 48109		11. Contract or Grant No. NAG1-296
		13. Type of Report and Period Covered Final Report: 1/1/84-12/31/84
12. Sponsoring Agency Name and Address National Aeronautics and Space Administration Washington, DC 20546		14. Sponsoring Agency Code 505-34-13-32
15. Supplementary Notes Langley Technical Monitor: Ricky W. Butler		

16. Abstract

In this report, we present an integrated modeling and analysis of error detection and recovery.

When fault latency and/or error latency exist, the system may suffer from multiple faults or error propagations which seriously deteriorate the fault-tolerant capability. We develop several detection models that enable us to analyze the effect of detection mechanisms on the subsequent error handling operations and the overall system reliability.

Following detection of the faulty unit and reconfiguration of the system, the contaminated processes or tasks have to be recovered. The strategies of error recovery employed depend on the detection mechanisms and the available redundancy. We consider several recovery methods, including, especially, the rollback recovery. The recovery overhead is evaluated as an index of the capabilities of the detection and reconfiguration mechanisms.

17. Key Words (Suggested by Author(s)) Error detection, latent errors/faults, diagnostics, unreliable results, computation loss, rollback recovery, retry, restart, recovery blocks.	18. Distribution Statement Unclassified – Unlimited Subject Category 62

19. Security Classif. (of this report) Unclassified	20. Security Classif. (of this page) Unclassified	21. No. of Pages 115	22. Price A06