EVALUATION OF AN ADAPTIVE CHECKPOINTING SCHEME FOR MULTIPROCESSOR SYSTEMS

by

Fransiskus Krisnadi Widjanarko

A Thesis Submitted to the Faculty of the

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

In Partial Fulfillment of the Requirements

For the Degree of

MASTER OF SCIENCE

WITH A MAJOR IN ELECTRICAL ENGINEERING

In the Graduate College

THE UNIVERSITY OF ARIZONA

1 9 9 5

STATEMENT BY AUTHOR

This thesis has been submitted in partial fulfillment of requirements for an advanced degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the head of the major department or the Dean of the Graduate College when in his or her judgment the proposed use of the material is in the interests of scholarship. In all other instances, however, permission must be obtained from the author.

SIGNED:________________________________

APPROVAL BY THESIS DIRECTOR

This thesis has been approved on the date shown below:

____________________________ __________________

William H. Sanders — Date

Associate Professor of

Electrical and Computer Engineering

ACKNOWLEDGMENT

This thesis work has taught me to be persistent in achieving my goal and to persevere in going through every uncomfortable situation. There are numerous people I would like to thank for their support in this endeavor. I would like to thank my advisor, Dr. William H. Sanders, for his guidance and advice, and also for giving me the opportunity to be part of the *UltraSAN* development team. I also want to thank my thesis committee, Dr. Pamela Delaney and Dr. Fredrick J. Hill, for reviewing my thesis, and to the whole PMRL crew for their friendly support.

I would like to give my special thanks to my parents and brothers for their support and encouragement since my first day in Tucson. Being far away from home for many years has made me realize how important my family is to me.

Finally , I would like to thank my friends from the *Indonesian Christian Fellowship* in Tucson, especially Mr. and Mrs. Erwin Sucipto and Ferdinand M. Pardede, for being my family during my study years at the University of Arizona. I really cherish our friendship and thank God for giving me friends like them all.

To my parents.

TABLE OF CONTENTS

ABSTRACT **8**

1. INTRODUCTION **10**

2. SYSTEM DESCRIPTION **16**

2.1. SYSTEM CONFIGURATION 16

2.2. FAULT ENVIRONMENT DESCRIPTION 19

3. THE ADAPTIVE CHECKPOINTING ALGORITHM **21**

4. SAN MODELS **32**

4.1. MULTIPROCESSOR MODEL 33

4.2. ADAPTIVE MODEL 43

4.3. FIXED MODEL 51

4.4. PERFORMANCE VARIABLES 52

5. RESULTS **56**

5.1. RESULTS OF THE MULTIPROCESSOR MODEL 57

5.2. RESULTS OF THE ADAPTIVE AND FIXED MODEL 69

6. CONCLUSIONS **77**

Appendix A. DERIVATIONS OF THE FIRST, SECOND AND THIRD MOMENT **79**

Appendix B. TRICKS IN *UltraSAN* IMPLEMENTATIONS **81**

B.1. ACCESSING SIMULATION TIME 81

B.2. INTERNAL GLOBAL VARIABLES DEFINITIONS 82

Appendix C. PROJECT DOCUMENTATIONS **83**

C.1. DOCUMENTATION OF THE MULTIPROCESSOR MODEL 83

C.2. DOCUMENTATION OF THE ADAPTIVE MODEL 89

C.3. DOCUMENTATION OF THE FIXED MODEL 107

REFERENCES **123**

ABSTRACT

Many checkpointing schemes have been proposed for multiprocessor systems. Both synchronous and asynchronous schemes have been proposed, and perform differently, depending on the fault environment and cost associated with state savings, rollback, and synchronization. The common goal of these schemes is to optimize the forward progress of the system. In this thesis, we propose a new asynchronous scheme, called *adaptive checkpointing*, for a multiprocessor system executing an application in a fault environment whose fault rate changes with time. The adaptive checkpointing scheme *adapts* its rate of checkpointing depending on the past fault arrival pattern. Basically, the scheme tries to always optimize the forward progress, by always using what it perceives as the optimum checkpoint rate as the frequency of fault arrival changes.

Finding the optimum checkpoint rate of an uncoordinated multiprocessor system with a fixed fault rate is, by itself, a hard problem and has no closed-form solution. In this thesis we construct a stochastic activity network (SAN) model to find the optimum checkpoint rate for varying values of fault rate. In addition to finding the optimum checkpoint rates, this model opens a doorway for evaluation of the adaptive checkpointing scheme, since this scheme requires the knowledge of optimum checkpoint rates. In order to evaluate the new scheme, another SAN model is built. *UltraSAN*, a stochastic activity network modeling package, is used to build and to solve both models.

Comparing the forward progress of a system with an adaptive checkpointing scheme and that of a system with the traditional fixed checkpointing scheme shows that the adaptive

checkpoint scheme performs better than the fixed scheme in all cases. The degree of superiority depends on the nature of the fault arrival process, the number of stable storages available to the algorithm, and the frequency of interactions between the processors in the system. Further study showed that the adaptive scheme is good for a system with a small number of stable storages and a high message transmission rate.

CHAPTER 1

INTRODUCTION

Many checkpointing schemes have been proposed for multiprocessor systems. A common goal in such schemes is to maximize the rate of *forward progress* that a process achieves or, in other words, maximize the fraction of time the system is in normal state and not performing checkpointing operations or re-doing computations due to rollbacks induced by faults. Maximizing the forward progress rate is equivalent to minimizing the execution time of a program. In evaluating a checkpointing scheme, it is important to determine how often the system should checkpoint in order to achieve the maximum forward progress rate. If checkpointing is done too often, it will reduce the forward progress rate due to the costs incurred in periodically saving the process state to stable storage. On the other hand, if checkpointing is not done often enough, it will take longer to recover from a faulty state. The optimal checkpointing rate is very difficult to determine, in general, and depends on many characteristics of the system, workload, and fault environment considered.

In spite of these difficulties, a significant amount of work has been done to determine the performance of various checkpointing schemes and, more specifically, the optimal checkpointing rate. Work was first done in the context of single processor systems. In this context, Young [1] first studied the optimum checkpoint interval assuming a fixed checkpointing interval and the assumption that faults do not occur during checkpointing or recovery. This work was extended by Brock [2] to model faults that occur during recovery, but not during checkpoint. Later, Chandy [3] and Chandy et al. [4] relaxed both assumptions, allowing faults to occur during both checkpointing and recovery. In parallel, Gelenbe [5] considered an exponentially distributed checkpoint interval and, in 1979, proved that the optimum checkpoint interval (in this case) is the one that is exponentially distributed.

More recent work has considered checkpointing communicating processes executing on multiprocessors. Broadly speaking, schemes for checkpointing communicating processes on multiple processor systems can be *coordinated*, where checkpoints are coordinated to insure that a fault on one processor does not cause a rollback of another processor (*rollback propagation*) to occur, or *uncoordinated*, where checkpointing is done asynchronously, but can cause rollback propagation. The general approach, among these two, that performs best depends on the application, architecture of the system, and fault environment. Practical cases do exist where each approach appears to be best, and hence both are interesting from a modeling point of view. When uncoordinated checkpointing is used, determining the optimal checkpointing rate is complicated due to the fact that data dependencies may exist between the processes, and a fault on one processor may cause a rollback of another processor. However, the uncoordinated approach does not suffer from: 1) the overhead cost of sending communication messages between processors for each checkpoint, and 2) the synchronization delay during normal operations, as in the coordinated approach.

Less work has been done to evaluate uncoordinated schemes, due to the difficulty of accounting for rollback propagation in the model. This approach of rollback scheme has been studied by Shin and Lee [24], where they concluded that the approach provides fast recovery; hence, it might be used for real-time applications. In their model, fault arrival is modeled as a simple exponential distribution, and the rate of checkpointing is not optimized. Later, Vaidya [32] reduced the overhead cost incurred in performing uncoordinated checkpoints. He proposed a recovery scheme called the *Two-Level Recovery Scheme*. In this scheme, the recovery process consists of two components. The first component, which uses volatile storages to save the checkpoints, is designed to recover from common cases of system failure. The second component, which uses stable storages to save the checkpoints, is designed to recover from uncommon cases of system failure. Because the checkpointing cost of the first component is "cheaper" than that of the second one (since accessing the volatile storages is "cheaper" than accessing the stable storages) the checkpoint rate of the first component is set higher than that of the second component. This recovery scheme, however, does not consider the optimization of the two checkpoint rates.

The work to date is limited in several respects. First, no result exists for the optimal checkpointing rate for multiple processor systems with asynchronous checkpointing, even for a Poisson fault arrival process. Second, the fault environment considered has been limited to Poisson arrival scheme with a fixed arrival rate. Results for the optimal checkpointing interval are important, since they would tell a user precisely how often checkpoint needs to be inserted, for a given fault environment, application, and system. Furthermore, more general fault environments should be considered since many recent studies have shown that the Poisson assumption is not valid in many cases. In particular, Iyer and Hsueh [26] observe several error states, each characterized by its waiting time, in their analysis of fault environments. Their examination of the mean and standard deviation of the waiting times indicate that not all waiting times are exponentials. They have also observed the occurrence of bursts of faults during which the system goes into an error-recovery cycle. Other empirical studies have shown that the fault model is more complex than the exponential model [16, 26, 27]. Specifically, the influence of a system's workload has been shown to affect fault arrival [27, 30].

This thesis addresses both of these limitations of previous models and introduces a new checkpointing scheme which adapts to a changing fault environment. Specifically, by representing the fault environment, application, checkpointing scheme, and system as a stochastic activity network (SAN), we obtain a solution for the optimal checkpointing rate for a two-processor system with uncoordinated checkpointing and Poisson fault-arrival process. This checkpointing scheme is a new scheme and is a significant contribution of the thesis.

Furthermore, we consider a more realistic (and more complicated) fault environment, where the fault rate observed by the system changes over time, depending on the state of an associated two-state Markov process, which we call a "Markov-modulated fault model." This fault model can correctly represent the occurrence of burst of faults, as observed in [26]. Since the fault rate changes over time, a checkpointing scheme with a fixed checkpoint rate, as proposed in the earlier works, will not work well in optimizing forward progress of the system. In this thesis, an adaptive checkpointing scheme is proposed to work with the fault model. The scheme *adapts* its rate of checkpointing depending on the observed fault rate, trying to determine the rate at which faults are currently occurring, and then checkpoint at the optimal rate for the current fault arrival rate.

To evaluate the new scheme and the traditional fixed checkpointing scheme, we build two more SAN models. These models enables us to evaluate and compare the performance of the adaptive checkpointing algorithm to that of the fixed checkpointing algorithm for the new fault model. The resulting forward progress shows that the adaptive checkpointing scheme is better than the fixed checkpointing scheme.

In summary, the goals of this thesis are as follow:

1. The development of a SAN model to represent a multiprocessor system with an uncoordinated checkpointing scheme and to find optimum checkpoint rates.

2. The development and evaluation of a new adaptive checkpointing scheme to work with a fault environment that can vary with time.

3. An illustration of the usefulness of the *UltraSAN* tool in modeling multiprocessor systems and fault environments.

The remainder of the thesis is organized as follows. Chapter 2 describes the multiprocessor fault-tolerant system to be evaluated. First, it describes the hardware configuration of the system, and then describes the fault environment. Chapter 3 explains the adaptive checkpointing algorithm in detail. The chapter describes in detail the four execution steps of the algorithm. The first three sections of Chapter 4 describe the three SAN models used in the thesis. We shall refer to the first, second and third SAN model as the *multiprocessor model* (SAN model to calculate the optimum checkpoint interval), the *adaptive model* (SAN model to evaluate the performance of the adaptive checkpointing scheme), and the *fixed model* (SAN model to evaluate the performance of the fixed checkpointing scheme), respectively. The last section of this chapter, Section 4.4, describes the definition of performance measures in terms of the SAN models.

Chapter 5 presents numerical results from the SAN models. The chapter is divided into two sections which present the resulting optimum checkpoint rates of the multiprocessor system, and the performance of the adaptive and fixed checkpointing scheme. Finally, Chapter 6 summarizes the results and suggests some areas of future research.

CHAPTER 2

SYSTEM DESCRIPTION

This chapter describes the multiprocessor system and the fault environment that are evaluated in this thesis. The first section of this chapter, system configuration, presents the hardware configuration of the system to be evaluated. The Markov-modulated fault model is presented in the second section.

2.1 SYSTEM CONFIGURATION

As outlined in the introduction, we consider a two-processor system with uncoordinated checkpointing. We assume that each processor has a fixed and finite amount of stable storage that can be used to hold checkpointed information. We express the size of each stable store in terms of the number of checkpoints that can be held. Note that this is an

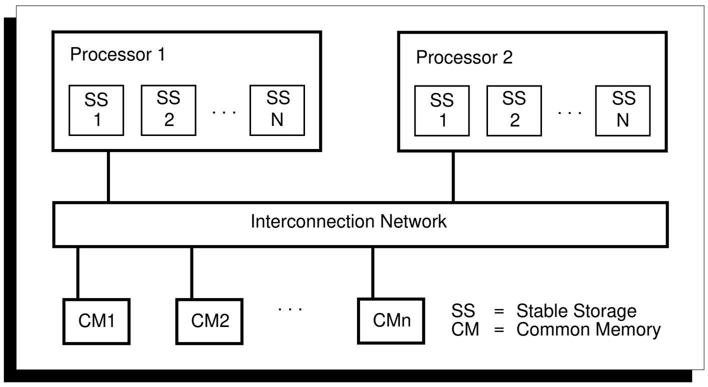


Figure 2.1: Organization of Fault-Tolerant Multiprocessor System

uncommon, but realistic assumption, since stable storage will be finite in any implementation. The two processors communicate to one another by exchanging messages via common memories. Figure 2.1 shows the organization of the system.

Given this processor configuration, we assume a single long-running application that is partitioned into two communicating processes running on the two processors. Each process is further partitioned into small tasks. Each task computes for an exponentially distributed amount of time, then sends a message to the other processor with probability P_{mesg} and checkpoints with probability $P_{chkpoint}$. We consider the cases where the checkpoint probability is fixed, and where it adapts, depending on the past fault inter-arrival times that the process has observed. Note that if a message is sent and a checkpoint is done, then the checkpoint follows the message. If a checkpoint (state saving) is done, it takes an exponentially distributed amount of time with rate λ_{state_saving}. Message transmission is assumed to be asynchronous; hence, from the point of view of the sending process, it takes a negligible amount of time. This cycle is then repeated, with the process entering another compute phase, followed by a possible message transmission and checkpoint.

Faults are assumed to be equally likely to occur in each processor, and occur according to the fault model described in the next section. When a fault occurs, the process immediately ceases execution and begins recovery. Recovery consists of rolling back to the most recent checkpoint and determining whether a message has been sent to the companion processor. If it has, then the companion processor must also roll back, since that message could have been based on corrupted data. In rolling back, the companion processor may, in turn, induce further roll backs in the original processor, if additional messages have been exchanged between them. In an iterative manner, the two processors determine how far

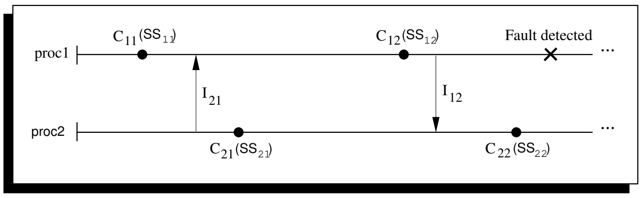


Figure 2.2: Checkpointing and interactions between two processors

they must roll back, and then resume execution from that point. Since each processor has a finite amount of stable storage, a rollback propagation may cause a processor to exhaust all of its available checkpoints (domino effect). In this case, we assume that there is a large cost associated with recovery, taking an exponentially distributed time with fixed rate $\lambda_{restart}$. This time can be thought of as the time to perform an alternative, but costly, state restoration method.

Figure 2.2 shows an example of rollback propagation. The lines represent process execution and the arrows represent the interactions between the two processors (I_{12} and I_{21}). The dots ($C_{11}, C_{12}, \ldots, C_{21}, C_{22}, \ldots$) represent checkpoint locations in the process line. Each checkpoint is associated with a stable storage (SS). Once I_{21} is executed, *proc1* records the interaction in its stable storage, in this case it is SS_{11}. When a fault is detected by *proc1*, it rolls its process back to its most recent checkpoint, which is C_{12}. The rollback algorithm looks at SS_{12} and finds out that *proc1* has interacted with *proc2*. *proc2* is, therefore, rolled back to C_{21} because of the possibility of message I_{12} being corrupted by the fault. The final checkpoints selected to recover from the error in this situation are C_{12} for *proc1* and C_{21} for *proc2*.

2.2 FAULT ENVIRONMENT DESCRIPTION

Faults in a computer system can be *permanent, intermittent* or *transient.* A permanent (hard) fault is a fault whose presence is not related to pointwise conditions of the system, either internal or external [31]. It is continuous and stable. An intermittent fault is a fault that is only occasionally present due to an unstable hardware condition. A transient (soft) fault is a fault resulting from a temporary environment (external) condition. Some examples of outside disturbances that might produce a transient fault are power jitter, ionization due to cosmic rays, electro-magnetic interference, and alpha particles from the packaging materials [18]. Of the three kinds of faults described above, transient faults have received the most attention from researchers. The reason is because the system that is inflicted by a transient fault cannot be repaired, since the hardware is physically undamaged. The other types of faults, which are due to underlying physical conditions, are potentially detectable by testing, and therefore repairable. A transient fault, on the other hand, might occur even in the absence of physical defects or unstable hardware. In this thesis we will only consider transient faults.

Two fault environment models are considered. In the first, transient faults occur as a Poisson process, with a fixed rate λ_{fixed}. This is the typical fault assumption with modeling systems such as this and is valid for certain applications. However, as argued in the introduction, empirical evidence suggests that fault rates change with time, either in bursts, or according to modes [26]. The multi-mode nature of these faults many be due to several different factors. For example, they may be due to changes in the physical environment of a computing platform, e.g., a flight control computer on a plane or spacecraft flying through areas with varying electromagnetic disturbance. Another cause may be a changing total workload on the system. Empirical results [17] suggest that transient faults occur at a higher rate when the workload is higher. To accurately model time varying fault rates, we consider a "Markov-modulated fault process," where faults occur according to a Poisson process, whose rate depends on the state of an associated two-state continuous-time Markov process. More precisely, we assume that faults occur with rate λ_1 for the first fault mode and λ_2 for the second fault mode. The rate of change between fault modes is likewise expressed by two parameters, μ_1 for the rate from fault mode 1 and 2 and μ_2 for the rate from fault mode 2 to 1.

The next chapter describes the adaptive checkpointing algorithm that is designed to work well in the above fault environment.

CHAPTER 3

THE ADAPTIVE CHECKPOINTING ALGORITHM

This chapter presents the adaptive checkpointing algorithm. As mentioned earlier, the main idea of the adaptive algorithm is to study the behavior of the fault environment and to *adapt* its checkpointing rate such that as the fault rate changes, the optimum checkpointing rate according to the *optimum checkpointing rate table* is always used. The optimum checkpoint rate table is a table which contains the optimum checkpoint rates of some specific fault rates. The construction of this table from the multiprocessor model is explained in the next chapter. Because we are assuming a two-mode fault model, the tasks of the adaptive checkpointing algorithm are, specifically, 1) to estimate the two fault rates of the two fault modes, 2) to adapt its checkpointing rate as the fault model alternates between the two fault modes.

There are four steps involved in the execution of the adaptive checkpointing algorithm in order to carry out those tasks. Below are those four steps:

1. Store the past history of fault inter-arrival time.

2. Estimate the two fault rates given the information from step 1.

3. Predict the current fault mode of the system.

4. Set the checkpointing rate of the system to the optimum value depending on the current estimated fault rate for the predicted fault mode.

Step 1 and Step 2 of the algorithm are executed every time a fault is detected. This is to make sure that the estimated fault rates are updated when a new fault inter-arrival time is observed. Step 3 and Step 4 of the algorithm are executed periodically. The frequency of executing Step 3 and Step 4 determines how often the checkpoint rates of the processors are updated. The goal is to make the processors always use the optimum checkpoint rate as the fault rate changes. Each of these steps is explained in detail in the remainder of this chapter. A complete pseudocode consisting of all four steps is given at the end of this chapter.

Step 1: Storing the past history of fault inter-arrival time. A simple way to record the history of fault arrivals is by having a long array of floating point numbers and storing the *fault inter-arrival times* in the array. The fault inter-arrival time is the time interval between two consecutive occurrences of a fault, except for the first occurrence of the fault, which is the time interval between the system start time and the first occurrence of the fault. A problem with this implementation is that the amount of memory we need in order to store the information is large. Further, the memory requirements increase as more faults are detected. A better way of recording the history of fault arrival is by encoding the information and updating the code(s) as occurrences of faults are detected.

It is important to recall that the purpose of storing this information is to be able to estimate the parameters of the two-fault mode model (i.e. the two fault rates). In this algorithm we use four variables, namely n, $\hat{m}_1$, $\hat{m}_2$, $\hat{m}_3$, to encode the past history of fault inter-arrival time. Let X be a random variable that represents the inter-arrival time of a fault. The following three equations, therefore, represent estimators of the first, second and third moment of the random variable X:

$$\hat{m}_1 = \frac{x_1 + x_2 + \cdots + x_n}{n}$$

$$\hat{m}_2 = \frac{x_1^2 + x_2^2 + \cdots + x_n^2}{n}$$

$$\hat{m}_3 = \frac{x_1^3 + x_2^3 + \cdots + x_n^3}{n},$$

where x_1, x_2, $\cdots$, x_n are the 1^{st}, 2^{nd}, $\cdots$, n^{th} fault inter-arrival time, and n is the number of fault occurrences. In an iterative way, the above equations can be represented as:

$$\hat{m}_1{'} = \frac{n}{n+1} \times \hat{m}_1 + \frac{1}{n+1} \times x \tag{3.1}$$

$$\hat{m}_2{'} = \frac{n}{n+1} \times \hat{m}_2 + \frac{1}{n+1} \times x^2 \tag{3.2}$$

$$\hat{m}_3{'} = \frac{n}{n+1} \times \hat{m}_3 + \frac{1}{n+1} \times x^3 \tag{3.3}$$

$$n^{'} = n + 1, \tag{3.4}$$

where x is the latest fault inter-arrival time, $\hat{m}_i{'}$ and $\hat{m}_i$ (i = 1, 2, 3) are the new value and the old value of the three moment estimators, and $n^{'}$ and n are the new and old values of the number of faults detected. The four variables (n, $\hat{m}_1$, $\hat{m}_2$, $\hat{m}_3$) are initially set to zero. The equations 3.1 - 3.4 are then executed every time a fault is detected. As shown in the following section, these four variables can be used to estimate the two fault rates of the fault model.

Step 2: Estimating the two fault rates. Estimating the parameters of a Markov-modulated model has been studied by several researchers. Some of the estimation methods are: *Bayesian estimators* [34, 35], *maximum likelihood estimation* [36], and *method of moments* [19]. We use the method of moments for the adaptive algorithm because it is simple and straight forward. This section explains the method of moments for estimating the two fault rates. Let the density function of X be:

$$f(x) = p\,\lambda_1\,e^{-\,\lambda_1\,x} + (1-p)\,\lambda_2\,e^{-\,\lambda_2\,x}, \tag{3.5}$$

where p is the mixture portion of the two fault modes, $\lambda_i\,e^{-\lambda_1\,x}$ is the representation of an exponentially distributed time interval between fault arrivals with rate λ_i (i = 1 and 2), and x represents the fault inter-arrival time. This equation is not the correct representation of the Markov-modulated fault model since the two fault modes are represented as a constant p instead of as a two-state continuous-time Markov process. However, it is an acceptable representation in estimating the two fault rates (λ_1 and λ_2).

Let m_1, m_2 and m_3 denote the first, second and third moment of the random variable X. We can, therefore, derive the following equations:

$$p\,\frac{1}{\lambda_1} + (1-p)\,\frac{1}{\lambda_2} = m_1 \tag{3.6}$$

$$p\,\frac{1}{{\lambda_1}^2} + (1-p)\,\frac{1}{{\lambda_2}^2} = \frac{1}{2}m_2 \tag{3.7}$$

$$p\,\frac{1}{{\lambda_1}^3} + (1-p)\,\frac{1}{{\lambda_2}^3} = \frac{1}{6}m_3 \tag{3.8}$$

The complete derivations of the three equations above are given in Appendix A. By manipulating equation 3.6 we have

$$p = \frac{m_1 - \frac{1}{\lambda_2}}{\frac{1}{\lambda_1} - \frac{1}{\lambda_2}} \tag{3.9}$$

Substituting this expression for p in 3.7 and 3.8 leads to the following two equations:

$$(m_1 - \frac{1}{\lambda_2})(\frac{1}{\lambda_1} + \frac{1}{\lambda_2}) = \frac{1}{2}m_2 - \frac{1}{{\lambda_2}^2} \tag{3.10}$$

$$(m_1 - \frac{1}{\lambda_2})(\frac{1}{{\lambda_1}^2} + \frac{1}{\lambda_1}\frac{1}{\lambda_2} + \frac{1}{{\lambda_2}^2}) = \frac{1}{6}m_3 - \frac{1}{{\lambda_2}^3} \tag{3.11}$$

Equation 3.10 may be solved for λ_i (i = 1 or 2), the solution being

$$\lambda_i = \frac{m_1 - \frac{1}{\lambda_{3-i}}}{\frac{1}{2}m_2 - m_1\frac{1}{\lambda_{3-i}}} \tag{3.12}$$

When λ_i from 3.12 is substituted in 3.11 and the result simplified, the equation for λ_j is

$$6(2{m_1}^2 - m_2)\frac{1}{{\lambda_j}^2} + 2(m_3 - 3m_1m_2)\frac{1}{\lambda_j} + 3{m_2}^2 - 2m_1m_3 = 0 \tag{3.13}$$

The two roots of this quadratic equation are $\frac{1}{\lambda_1}$ and $\frac{1}{\lambda_2}$. Below is the final form of λ_1 and λ_2 as the function of m_1, m_2 and m_3

$$\lambda_1 = \frac{24m_1m_1 - 12m_2}{-2m_3 + 6m_1m_2 + D} \tag{3.14}$$

$$\lambda_2 = \frac{24m_1m_1 - 12m_2}{-2m_3 + 6m_1m_2 - D}, \tag{3.15}$$

where D is

$$D = \sqrt{4{m_3}^2 + 72{m_2}^3 - 72m_1m_2m_3 - 108{m_1}^2{m_2}^2 + 96m_3{m_1}^3}$$

term	*meaning*
X	Random variable represents fault inter-arrival time
x	A time interval between two consecutive fault arrivals
λ_1	Rate of fault arrivals in fault mode 1
λ_2	Rate of fault arrivals in fault mode 2

Table 3.1: Definition of terms

Recall from the description of Step 1 that we record the estimator of the first, second and third moment of the random variable that represents the inter-arrival time of a fault. We can, therefore, use the equations 3.14 and 3.15 and the recorded three moment estimators ($\hat{m}_1$, $\hat{m}_2$, $\hat{m}_3$) from Step 1 to estimate λ_1 and λ_2. We refer to the estimated value of λ_1 and λ_2 as *EstimatedLambda1* and *EstimatedLambda2*, respectively. The reason why we need to use the three moments is because we have three unknown parameters to solve in equation 3.5, namely λ_1, λ_2, and p.

Step 3: Predicting the current fault mode of the system. Once λ_1 and λ_2 are calculated, the next step is to decide the current fault mode of the system. For the sake of explaining the algorithm, the conventions in Table 3 are used throughout this subsection. The task of the fault mode estimation algorithm is to decide whether this sample is generated from the first fault mode or the second fault mode. Given a value of x and the estimated values of λ_1 and λ_2 we predict the fault mode that most likely generates the inter-arrival x. Conditioning on the fault arrival mode the fault inter-arrival distributions are as follows:

$$P\{X > C_p \mid in\ fault\ mode\ 1\} = e^{-\lambda_1 \cdot C_p}$$

$$P\{X > C_p \mid in\ fault\ mode\ 2\} = e^{-\lambda_2 \cdot C_p},$$

where C_p (the cluster point) is a constant.

If we let $C_p = \frac{1/\lambda_1 + 1/\lambda_2}{2}$, then the equations become:

$$P\{X > C_p \mid in\ fault\ mode\ 1\} = e^{-\frac{1}{2} - \frac{\lambda_1}{2\lambda_2}}$$

$$P\{X > C_p \mid in\ fault\ mode\ 2\} = e^{-\frac{1}{2} - \frac{\lambda_2}{2\lambda_1}}$$

If λ_1 is much larger than λ_2, then $P\{X > C_p \mid in\ fault\ mode\ 1\}$ is much smaller compared to $P\{X > C_p \mid in\ fault\ mode\ 2\}$. For example, let $\lambda_1 = 0.05$ and $\lambda_2 = 0.001$, then

$$\begin{aligned} P\{X > C_p \mid in\ fault\ mode\ 1\} &= e^{-\frac{1}{2} - \frac{0.05}{2 \cdot 0.001}} \\ &= 8.4235 \times 10^{-12} \end{aligned}$$

$$\begin{aligned} P\{X > C_p \mid in\ fault\ mode\ 2\} &= e^{-\frac{1}{2} - \frac{0.001}{2 \cdot 0.05}} \\ &= 0.6005 \end{aligned}$$

Because the probability that $X > C_p$ given that the system is in fault mode 1 is very small, we predict that x is from the second fault mode whenever $x > C_p$. The value of the cluster point (C_p) need not necessarily be set to $\frac{1/\lambda_1 + 1/\lambda_2}{2}$. In this thesis $\frac{1/\lambda_1 + 1/\lambda_2}{2}$ is used because the fault inter-arrival times, conditioned on the fault mode, are exponentially distributed; thus, the expected values are $E[X_1] = 1/\lambda_1$ and $E[X_2] = 1/\lambda_2$, respectively.

The following rule is used to decide if the system has switched from one fault mode to the other with the assumption that $\lambda_1 > \lambda_2$. If the system is in the first fault mode and the time elapsed since the last detected fault (*time_elapsed*) is greater than the cluster point, then it is assumed that the system fault mode has changed to the second fault mode. If the system is in the second fault mode and the last inter-arrival time of fault (*last_fault_interval*) is less than the cluster point, then it is assumed that the system fault mode has changed

```
ClusterPoint = (1/EstimatedLambda1 + 1/EstimatedLambda2) / 2
if (EstimatedLambda1 > EstimatedLambda2) {
  if (EstimatedMode == 1 and time_elapsed > ClusterPoint) {
    EstimatedMode = 2
  }
  else if (EstimatedMode == 2 and last_fault_interval < ClusterPoint) {
    EstimatedMode = 1
  }
}
else {
  if (EstimatedMode == 2 and time_elapsed > ClusterPoint) {
    EstimatedMode = 1
  }
  else if (EstimatedMode == 1 and last_fault_interval < ClusterPoint) {
    EstimatedMode = 2
  }
}
```

Figure 3.1: The pseudocode of fault mode estimation rule

to the first fault mode. For the case where $\lambda_1 < \lambda_2$, the rule does not change except each "first fault mode" term becomes "second fault mode" and vice versa. Figure 3.1 shows the pseudocode of the rule. We shall refer to this rule as *fault mode estimation rule.*

Step 4: Setting the checkpointing rate to the optimum value. Once we have estimated the two fault rates (Step 2) and determined the current fault mode (Step 3), we can determine the current rate of fault arrival with the following equation:

$$\lambda = \begin{cases} EstimatedLambda1 & \text{if } EstimatedMode == 1 \\ EstimatedLambda2 & \text{otherwise,} \end{cases} \tag{3.16}$$

where λ denotes the estimated current fault rate. The next step for the algorithm is to get the optimum checkpointing rate value from the optimum checkpointing rate table according to the value of λ.

Table 3.2 shows an example of an optimum checkpointing table. As mentioned earlier, this table was constructed from the results of the multiprocessor model, which will be described in the next chapter. The algorithm compares the value of λ to the fault rate

Fault Rate	*Opt. Chkpts. Rate*
0.001	0.4
0.003	0.8
0.005	1.2
0.007	1.4
0.009	1.8
0.010	2.0
0.015	2.6
0.020	3.0
0.025	3.2
0.030	3.4
0.035	3.6

Table 3.2: An example of an optimum checkpoint rate table

values in the left column of the table starting from the first row. If the fault rate value from the table is smaller than λ, then the algorithm looks at the next fault rate value in the next row and does the same comparison. The algorithm keeps on going down the row until it hits the row in which the fault rate is greater than λ, or it hits the last row of the table. In both cases the algorithm uses the checkpointing rate value from the right column of the table associated with the last row that it hits.

In summary, Table 3.3 shows the variables used in the adaptive checkpointing algorithm and their meaning, and Figure 3.2 shows the pseudocode of the algorithm which includes the four steps explained in this chapter.

```
Initialization
    EstimatedLambda1  = 0.0          n                 = 0
    EstimatedLambda2  = 0.0          m1 = m2 = m3      = 0.0
    EstimatedMode     = 1            OptChkpointRate   = 0.0
    ClusterPoint      = 0.0

    OptTable_faultrates[41]   = {0.001, 0.002, ... , 0.040}
    OptTable_chkptsrate[41]   = {0.040, 0.060, ... , 0.350}


Adaptive_Checkpointing_Algorithm () {
    if (fault is detected) {

        last_fault_interval = the last interarrival time of fault              step 1
        m1 = (n/(n+1) * m1) + (1/(n+1) * last_fault_interval)
        m2 = (n/(N+1) * m2) + (1/(n+1) * last_fault_interval^2)
        m3 = (n/(n+1) * m3) + (1/(n+1) * last_fault_interval^3)
        n   = n + 1

        EstimatedLambda1 = estimate lambda1 by m1, m2 and m3                   step 2
        EstimatedLambda2 = estimate lambda2 by m1, m2 and m3
    }

    if (time to adapt) {

        ClusterPoint = (1/EstimatedLambda1 + 1/EstimatedLambda2) / 2           step 3
        if (EstimatedLambda1 > EstimatedLambda2) {
            if (EstimatedMode == 1 and time_elapsed > ClusterPoint) {
                EstimatedMode = 2
            }
            else if (EstimatedMode == 2 and last_fault_interval < ClusterPoint) {
                EstimatedMode = 1
            }
        }
        else {
            if (EstimatedMode == 2 and time_elapsed > ClusterPoint) {
                EstimatedMode = 1
            }
            else if (EstimatedMode == 1 and last_fault_interval < ClusterPoint) {
                EstimatedMode = 2
            }
        }


        if (EstimatedMode == 1)                                                step 4
            current_lambda = EstimatedLambda1
        else
            current_lambda = EstimatedLambda2
        for (row = 1 to 40) {
            if (current_lambda < OptTable_faultrates[row])
                exit the for loop
        }
        OptChkpointRate = OptTable_chkptsrate[row]

    }
}
```

Figure 3.2: The pseudocode of the adaptive checkpointing algorithm

Variable name	*Meaning*	*Used in Step*
EstimatedLambda1	Estimated value of λ_1	2 & 3
EstimatedLambda2	Estimated value of λ_2	2 & 3
EstimatedMode	Estimated value of the current fault mode	3
ClusterPoint	Separator value of sample from the 2 fault modes	3
n	Fault counter	1 & 2
$\hat{m_1}, \hat{m_2}, \hat{m_3}$	The first, second and third moment estimators	1 & 2
OptChkpointRate	The estimated value of the opt. checkpointing rate	4
OptTable_faultrates	The opt. checkpoint rate table; the *"Fault Rate"* column	4
OptTable_chkptsrate	The opt. checkpoint rate table; the *"Opt. Chkpts. Rate"* column	4

Table 3.3: Variables in the adaptive checkpointing algorithm

CHAPTER 4

SAN MODELS

This chapter presents the *stochastic activity network* (SAN) model of the system described in the previous chapters. Stochastic activity networks [28, 29] are stochastic extensions to Petri Nets that can be used to obtain both analytical and simulation results concerning the performance of many systems. Figure 4.1 shows an example of a SAN model. SANs consist of five types of components: *timed activities* (ovals), *instantaneous activities* (vertical lines), *places* (circles), *input gates* (triangles with their point connected to an activity), and *output gates* (triangles with their backside connected to an activity). The execution of SANs is discussed in detail in [28]. Figure 4.1 and all other model figures were generated using *UltraSAN* [14, 15], a stochastic activity network modeling package.

There are three models presented in this section. The first one, called the *multiprocessor model*, is a basic model of a two-processor system with a fixed checkpoint interval and a single-mode fault model. This model will be used to calculate the optimum checkpoint interval for different (fixed) fault arrival rates. The multiprocessor model is used to construct the *optimum checkpoint rate table*. The second model, called the *adaptive model*, is a model of a two-processor system implementing the adaptive checkpointing scheme described in the previous chapter, with the "Markov-modulated fault model" as its fault environment. The adaptive model is used to evaluate the performance of the adaptive checkpointing scheme under the two-mode fault model. The last model is called the *fixed model*. It is very similar to the adaptive model, except that it implements the fixed checkpointing scheme instead of

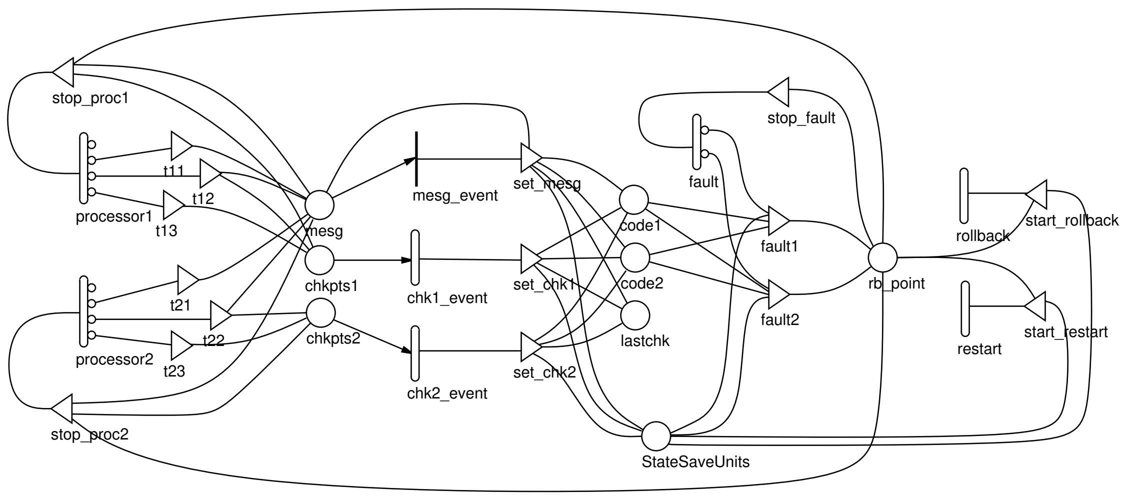


Figure 4.1: SAN-based model of the multiprocessor model

the adaptive checkpointing scheme. In the fixed model, the checkpoint model part of the adaptive model is replaced with a single global variable whose value is fixed. The global checkpoint rate value is used for both fault modes.

The following *UltraSAN* terms will be used to describe the models:

- MARK - a macro definition to represent the marking of a given place. For example, "MARK(rb_point)" represents the marking of the place *rb_point*.
- GLOBAL_S - a macro definition to represent a global variable of type short integer whose value can be either varied or fixed.
- GLOBAL_D - a macro definition to represent a global variable of type double floating point whose value can be either varied or fixed.

4.1 MULTIPROCESSOR MODEL

Figure 4.1 shows the multiprocessor model. As mentioned earlier, we consider a system with two communicating processes running on the two processors. Each process is partitioned into small tasks, each of which computes for an exponentially distributed amount of time. In the model, the two processors are represented by two activities called *processor1* and *processor2*. The time of completion of the processor activities represents the execution time of a task. We will use this time as the "basic unit" of the execution of the other processes, such as state saving process and restart process, and we will refer to the rate of the processing activities as the *processing unit* (λ_{proc}). Each activity is associated with two possible events that might occur during the task execution. The two events are 1) checkpoint insertion with probability $P_{chkpoint}$, and 2) message transmission with probability P_{mesg}. Since the two events are independent of one another, we have four possible cases that might happen during the task execution. Table 4.1 shows the four possible combinations of events and their probability of occurrence in terms of $P_{chkpoint}$ and P_{mesg}. These four events are represented by the four cases associated with the *processor1* and *processor2* activities. Notice that $P_{chkpoint}$ and P_{mesg} determine the checkpoint rate and the interaction/message rate, which can be calculated as $P_{chkpoint} \times \lambda_{proc}$ and $P_{mesg} \times \lambda_{proc}$, respectively. This is true because the two activities (*processor1* and *processor2*) are exponentially distributed with rate equal to λ_{proc}. The total message rate for the multiprocessor system becomes the number of processors (2) $\times$ the message rate of each processor. We shall use the checkpoint rate and message rate, instead of the respective probabilities, when reporting results.

In order to signal the occurrence of each event from the two processors, three places (*mesg*, *chkpts1* and *chkpts2*) are connected to the cases of each processor activity. When one of these four events happens, a token is placed in the appropriate place. Placing a

Possible events	*Probability*
A mesg. transmission w/o any checkpoint insertion	$P_{mesg} \times (1 - P_{chkpoint})$
A mesg. transmission and follow w/ a checkpoint insertion	$P_{mesg} \times P_{chkpoint}$
A checkpoint insertion w/o any mesg. transmission	$(1 - P_{mesg}) \times P_{chkpoint}$
Neither mesg. transmission nor checkpoint insertion	$(1 - P_{mesg}) \times (1 - P_{chkpoint})$

Table 4.1: Possible events on each processor

token in the place *mesg* means a message has been sent either by processor1 or processor2. Placing a token in the place *chkpts1/chkpts2* means a checkpoint is to be inserted by processor1/processor2. The places *mesg*, *chkpts1* and *chkpts2* are each connected to an activity via an arc. The activities (*mesg_event*, *chk1_event*, and *chk2_event*) represent the cost of (time associated with) executing each event. A bar representing activity *mesg_event* means the activity is an *instantaneous activity*, which means no delay is associated with the event. This activity can be easily changed to *timed activity* if we assumed some cost associated with sending a message, which is what we might want to do if we are modeling distributed systems where sending a message might take a significant time. As mentioned earlier, message transmission is assumed to be asynchronous; hence, from the point of view of the sending processor, it takes a negligible amount of time. The instantaneous activity is, therefore, used to represent this costless message transmission. The checkpoint, on the other hand, is represented by a timed activity. The rate of *chk1_event* and *chk2_event* activities (λ_{state_saving}) represents the cost of doing checkpoint/state saving. We shall use the terms checkpoint and state saving interchangeably to represent the process of checkpointing for the rest of this thesis.

As each processor interacts with the other processor and checkpoints are inserted, it is important to keep track of these events chronologically for the purpose of doing rollback recovery in case a fault is detected. The job of keeping track of all these events is handled by output gates: *set_mesg*, *set_chk1*, and *set_chk2*, and by places: *code1*, *code2* and *lastchk*. To keep track these chronological events in the SAN model, we need a way of encoding and decoding these events into an integer code whose value can be stored as a marking of a place. The encoding method is explained below; and the decoding method is explained later in this section. We use two codes to record these events.

The values of the codes are recorded as the marking of the places *code1* and *code*. Each message that comes after a checkpoint by the *processor1* is recorded in the code of the place *code1*; and each message that comes after a checkpoint by the *processor2* is recorded in the code of the place *code2*. Any messages that occur prior to the first checkpoint are not recorded because they do not have any effect on the rollback process. The marking of the place *lastchk* tells whether the last checkpoint comes from the *processor1* or *processor2*. A code, which is calculated with the formula below, is associated with each message.

$$code^{i}_{mesg} = (chkpts^{3-i} + 1) \times (SS + 2)^{chkpts^{i}},$$

where:

- i = 1 or 2.
- SS = number of stable storages in each processor.
- $code^{i}_{mesg}$ = the code of a message that comes after a checkpoint by *processor* i.
- $chkpts^{i}$ = the number of checkpoints that has been inserted by *processor* i prior to current message; and SS being its maximum number.

We shall refer to this code as the *message code*. Multiple messages that occur between two checkpoints are considered as one message. In other words, there is only one message code associated with multiple messages that occur between two checkpoints. The sum of $code^{i}_{mesg}$ for all messages that occur within the last $SS \times 2$ (number of processor) checkpoints is stored as the marking of the place *code* i, where i = 1 or 2. The place *code1* stores the sum of the message codes of the messages that come after a checkpoint of *processor1*, and *code2* stores those of the messages that come after a checkpoint of *processor2*. Any messages that occur prior to the last $SS \times 2$ checkpoints are meaningless in determining how far a process needs to be rolled back, and are, therefore, unrecorded by the *code1* and *code2* places. The complete formula for the marking of the place *code1* and *code2* is the following:

$$MARK(code\ i) = \sum_{the\ last\ SS \times 2\ chkpts.} code^{i}_{mesg} = (chkpts^{3-i} + 1) \times (SS + 2)^{chkpts^{i}},$$

where i = 1 or 2. Figure 4.2 shows an example of this message coding for a system with two stable storages. Because the markings of the place *code1* and *code2* are dependent on the occurrence of checkpoint insertion and message transmission events, their values are updated every time one of those events occurs. The gate *set_mesg* updates their values when a message transmission occurs. The gate *set_chk1*/*set_chk2* update their values when a checkpoint from *processor1*/*processor2* occurs. The number of stable storages, which is assumed to be the same for both processors, is represented by the initial marking of the place *StateSaveUnits*[1] .

The occurrences of fault in the system is represented by activity *fault*. The rate of this activity determines the fault arrival rate of the system. When a fault occurs in the system, it can come from either *processor1* or *processor2*. This possibility is represented

[1]State save unit is another term for stable storage

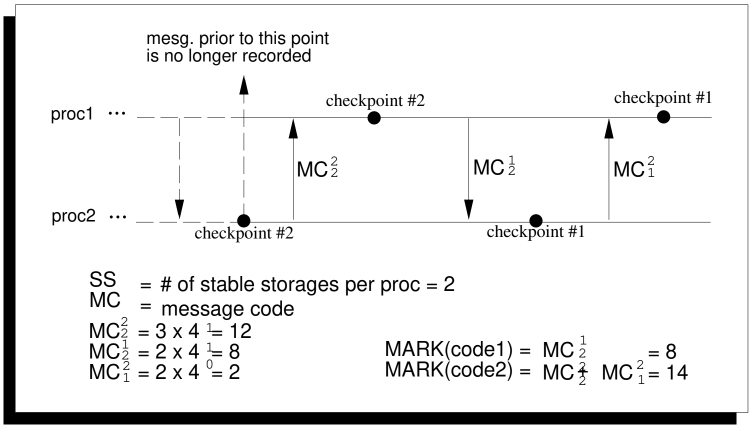


Figure 4.2: Calculations of MARK(code1) and MARK(code2) for SS = 2

by the two case probabilities associated with the activity *fault.* Since we are assuming an equal probability for each processor to generate faults, the case probabilities are set to 0.5. Upon the occurrence of a fault, activity *processor1*, *processor2* and *fault* are temporarily disabled until the recovery process is completed. The two processors are disabled because no progress is made during the recovery process. The *fault* activity is disabled because we assume no fault occurs during the recovery process. The output gate *fault1*, or *fault2* depending on which processor the fault is generated from, calculates how far behind the system has to be rolled back. Basically, *fault1/fault2* decodes the markings of places *code1* and *code2* and decides the consistent rollback point for the system to recover from the fault.

In order to explain the decoding of the marking of the place *code1* and *code2*, let us consider the previous example of interactions between the two processors in Figure 4.2. The markings of the places *code1* and *code2* are 8 and 14, respectively. We can decode these markings to get the message codes with the following formula:

$$MC_n^i = (MARK(code\ i) - MC_{n-1}^i)\ mod\ (SS+2)^n,$$

where

- MC_n^i = the message code between the checkpoint $\#n$ and the checkpoint $\#(n-1)$ of processor i, where the checkpoint #1 is the last checkpoint, checkpoint #2 is the one prior to the checkpoint #1, and so on.

- $MC_0^i = 0$.

- n = 1, 2, $\cdots$,SS.

By decoding the marking of *code1* and *code2* with the above formula, we have the following message codes:

- by decoding MARK(code1):
  - MC_1^1 = (8 - 0) mod 4^1 = 0
  - MC_2^1 = (8 - 0) mod 4^2 = 8
- by decoding MARK(code2):
  - MC_1^2 = (14 - 0) mod 4^1 = 2
  - MC_2^2 = (14 - 2) mod 4^2 = 12

These message codes are the same as the original message codes in Figure 4.2.

Once we have these message codes, the next step is to decode these message codes and determine the rollback point. Below is the formula to decode the message code:

$$CP_n^i = \frac{MC_n^i}{(SS+2)^{n-1}},$$

where CP^{i}_{n} is the checkpoint # to be used by processor i' because of the rollback propagation caused by the message whose message code is MC^{i}_{n}, and i' = 1 if i = 2, otherwise i' = 2. By decoding the above message codes we have:

- $CP^1_1 = 0 / 4^0 = 0 \Rightarrow$ processor 2 should be rolled back to the checkpoint #0 (no rollback)
- $CP^1_2 = 8 / 4^1 = 2 \Rightarrow$ processor 2 should be rolled back to the checkpoint #2
- $CP^2_1 = 2 / 4^0 = 2 \Rightarrow$ processor 1 should be rolled back to the checkpoint #2
- $CP^2_2 = 12 / 4^1 = 3 \Rightarrow$ processor 1 should be rolled back to the checkpoint #3

With the knowledge of the rollback propagation inflicted by each message we can determine how far rollback is to be executed (rollback point). Let $RB1$ be the rollback point of the *processor1* and $RB2$ be the rollback point of the *processor2*. If a fault is detected on *processor1*, then $RB1$ is initialized to 1, which means "rollback to the checkpoint #1," and $RB2$ is initialized to 0, which means "no rollback is needed." If a fault is detected on *processor2*, then $RB1$ is initialized to 0 and $RB2$ is initialized to 1. The initialization occurs like this because if we assume no rollback propagation will occur, then only the processor with the fault will be rolled back, and it is rolled back to the last checkpoint. The next step after this initialization is to check if rollback propagation occurs, and to determine the rollback point of both processors taking into acount the rollback propagation. Below are the equations to update the values of $RB1$ and $RB2$ as rollback propagation is considered:

$$RB^{'}_1 = MAX(RB_1, CP^2_{RB_2})$$

$$RB^{'}_2 = MAX(RB_2, CP^1_{RB_1}),$$

where

$$MAX(X,Y) = \begin{cases} X & \text{if } X > Y \\ Y & \text{if } X < Y \end{cases}$$

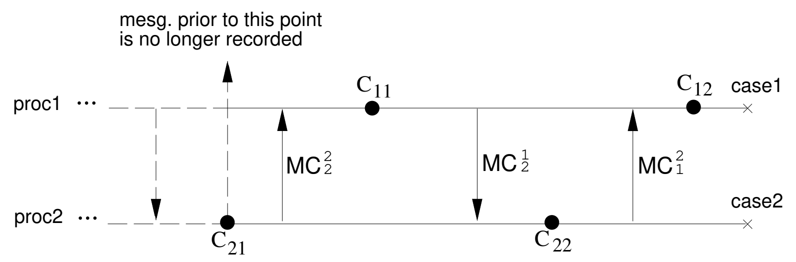
$CP^i_{RB_i}$ (i = 1 or 2) is the rollback point of *processor j* (j = 2 if i = 1; j = 1 otherwise) inflicted by *processor i* because of the interaction of the two processors. If RB_i is zero then CP^i_0 equals to zero. The two rollback points (RB_1 *and* RB_2) are iteratively updated using the above equations as long as no "consistent rollback point" has been achieved. The consistent rollback points, which are the final values of RB_1 and RB_2, are achieved if ,after executing the equations above, the new values $RB_1^{'}$ *and* $RB_2^{'}$ are the same as their old values. Once we have the consistent rollback points, the "farthest" value between RB_1 and RB_2, that is the MAX(RB_1, RB_2), is selected as the rollback point of the system. The reason of selecting the larger value is to avoid having one processor doing forward progress, while the other processor is doing recovery. Figure 4.3 shows the pseudocode of the algorithm to determine the rollback point explained above and Figure 4.4 shows the calculation of the rollback point of the previous example. There are two cases shown on the example in Figure 4.4. In the first case, where the fault occurs on the first processor, the final rollback point turns out to be the last checkpoint of processor 1. In the second case, where the fault occurs on the second processor, the system is forced to restart to recover from the fault because the final rollback point (3) turns out to be larger than the available number of checkpoints (2).

Once a rollback point is determined, an appropriate number of tokens, representing how far rollback has to be done (checkpoint #), is placed in the place *rb_point*. If the consistent rollback point turns out to be larger than the available stable storages (as in the case 2 of

```
if (fault is on processor1) {
   RB1 = 1
   RB2 = 0
}
else
   RB1 = 0
   RB2 = 1
}
while (RB points are not consistent) {
   RB1' = MAX (RB1, CP^2_RB2)
   RB2' = MAX (RB2, CP^1_RB1)
}

RBpoint = MAX (RB1, RB2)
```

Figure 4.3: The pseudocode to determine the consistent rollback point



Case1 : fault on proc1
$RB_1 = 1$
$RB_2 = 0$

$RB'_1 = max(1, CP^2_0) = max(1,0) = 1$
$RB'_2 = max(0, CP^1_1) = max(0,0) = 0$

Final RBpoint $= max(RB'_1, RB'_2) = 1$

Case2 : fault on proc 2
$RB_1 = 0$
$RB_2 = 1$

$RB'_1 = max(0, CP^2_1) = max(0,2) = 2$
$RB'_2 = max(1, CP^1_2) = max(1,2) = 2$

$RB'_1 = max(2, CP^2_2) = max(2,3) = 3$
$RB'_2 = max(2, CP^1_2) = max(2,2) = 2$

$RB'_1 = max(3, CP^2_2) = max(3,0) = 3$
$RB'_2 = max(2, CP^1_3) = max(2,2) = 2$

Final RBpoint $= max(RB'_1, RB'_2) = 3$
$=$ restart $(3 > SS)$

Figure 4.4: An example of rollback point calculations

the above example), the restart process is initiated; otherwise, reexecution of the process starting from the selected rollback point (as in the case 1 of the above example) is initiated. The restart process is represented by activity *restart* with rate $\lambda_{restart}$, while the rollback recovery process is represented by activity *rollback* with rate $\lambda_{rollback}$. Since the $\lambda_{rollback}$ represents the time to re-execute the process starting from the calculated rollback point, its value should be $\lambda_{chkpoint} \times \frac{1}{the\ rollback\ point}$. Thus, the "farther" the rollback point, the larger it takes to recover from the error. Once the recovery process, either restart or rollback, is done, activity *processor1*, *processor2* and *fault* are re-enabled, and the system goes back to execution in the normal mode.

4.2 ADAPTIVE MODEL

The SAN model of a system with the adaptive checkpointing scheme is showed in Figure 4.5. The model is similar to the model described in the previous section. The only differences are in the fault model and the checkpoint model. In this model, the fault model is different because we are considering the "Markov-modulated fault process," and the checkpoint model is different because we are considering the adaptive checkpointing algorithm. Figure 4.6 shows the fault model and the checkpoint model parts of the adaptive SAN model.

Fault model: As mentioned earlier, we are assuming a two-mode fault model where faults occur with rate λ_1 for the first fault mode and λ_2 for the second fault mode, and the rate of change between fault modes is likewise expressed by two parameters, μ_1 for the rate from fault mode 1 to 2, and μ_2 for the rate from fault mode 2 to 1. In the SAN model, the two alternating fault modes are represented by the place *faultmode*, the activity *switchmode*,

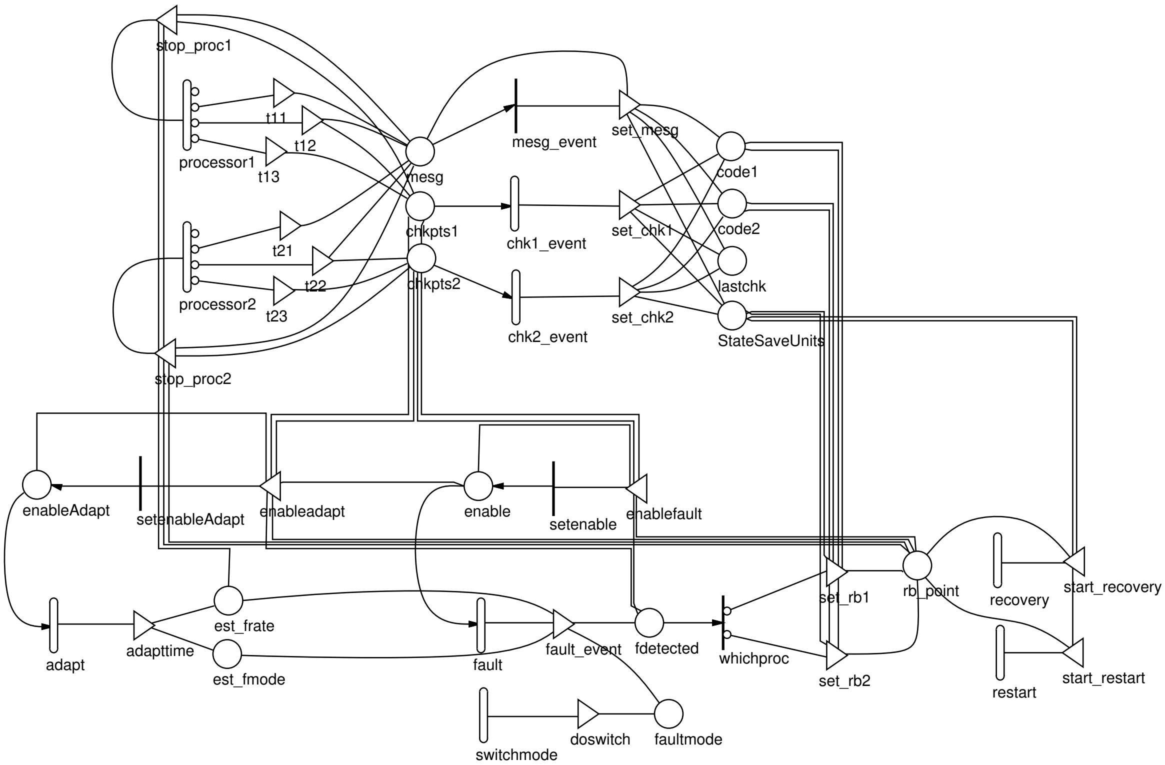


Figure 4.5: SAN representation of the multiprocessor system with adaptive checkpointing

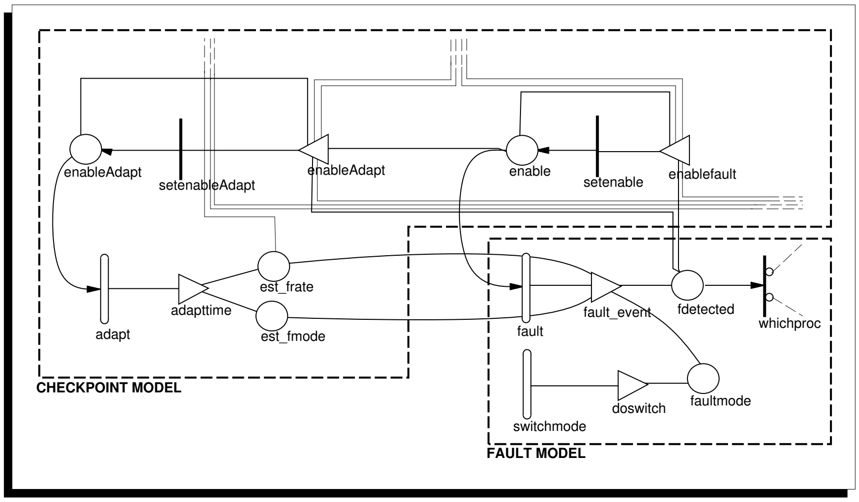


Figure 4.6: Fault model and checkpoint model parts of the adaptive model

```
if (MARK(faultmode) == 1)
  return(GLOBAL_D(MU1));
else
  return(GLOBAL_D(MU2));
```

Figure 4.7: The marking dependent rate of activity *switchmode*

and the output gate *doswitch*. The marking of the place *faultmode*, which alternates between 1 and 2, represents the current fault mode of the fault model. The initial marking of the place can be set to either 1 or 2 without affecting the performance measures since we will study the steady state behavior of the system. The activity *switchmode* represents the rate of change between the two fault modes. When the system is in the fault mode 1 (MARK(faultmode) == 1), the activity represents the rate of change from fault mode 1 to 2 (μ_1). When the system is in the fault mode 2 (MARK(faultmode) == 2), the activity represents the rate of change from fault mode 2 to 1 (μ_2). Figure 4.7 shows the rate code of the activity *switchmode*.

Let us assume the system is initially in fault mode 1. This initial state is represented by setting the initial marking of place *faultmode* to 1 (MARK(faultmode) = 1). The rate of the activity *switchmode* becomes μ_1 because of this initial marking. Once the activity *switchmode* is completed, the output gate *doswitch* changes the marking of place *faultmode* from 1 to 2, and the rate of activity *switchmode* becomes μ_2 to represent the rate of change from fault mode 2 to 1. The next time activity *switchmode* is completed the state is changed back to the initial state with the marking of *faultmode* equals to 1; and the process is repeated.

```
if (MARK(faultmode) == 1)
  return (GLOBAL_D(LAMBDA1));
else
  return(GLOBAL_D(LAMBDA2));
```

Figure 4.8: The marking dependent rate of activity *fault*

The activity *fault* represents the rate of fault arrival. In order to associate a different rate of fault arrival for each fault modes, the rate of the activity *fault* is made dependent on the marking of the place *faultmode*. When the marking of *faultmode* is 1, the rate of activity *fault* equals λ_1 which is the rate of fault arrival when the system is in fault mode 1. When the marking of *faultmode* is 2, the rate of activity *fault* equals λ_2 which is the rate of fault arrival when the system is in fault mode 2. Figure 4.8 shows the rate code of the activity *fault*.

Checkpoint model: Recall from the previous chapter that the adaptive checkpointing algorithm consists of the four steps below:

- Step1: Store the past history of fault inter-arrival time.
- Step2: Estimate the two fault rates.
- Step3: Determine the fault mode the system is currently in.
- Step4: Set the checkpointing rate of the system to the optimum value depending on the current estimated fault rate.

The first two steps (Step 1 and 2) are implemented inside the output gate *fault_event* because, as mentioned earlier, they are executed every time a fault is detected. The last

```
double EstimatedLambda1 = 0.0;
double EstimatedLambda2 = 0.0;
double EstimatedP = 0.0;
double m1 = 0.0;
double m2 = 0.0;
double m3 = 0.0;
double n = 0.0;

int EstimatedMode = 0;
int AdaptNow = 0;

float aveCheckpoint = 0.0;

/************************************************************
 * The optimum checkpoint rate table for SS = 2
 ************************************************************/
float OptTable_faultrates [] = {0.005, 0.010, 0.015, 0.020, 0.025,
                                0.030, 0.035, 0.040, 0.045, 0.050};
float OptTable_chkptsrate [] = {0.09, 0.12, 0.14, 0.17, 0.20, 0.24,
                                 0.28, 0.30, 0.32, 0.34};

/************************************************************
 * Variables to hold simulation times
 ************************************************************/
timeValue currentData = 0.0;
timeValue enableTime = 0.0;
timeValue enableAdaptTime = 0.0;
timeValue elapsed_time = 0.0;
timeValue last_fault_interval = 0.0;
```

Figure 4.9: The file *myglobal.h* which contains global variable declarations

two steps (Step 3 and 4) are implemented inside the output gate *adapttime* because they are executed periodically according to the rate of the activity *adapt*, that is λ_{adapt}.

All variables used in the adaptive algorithm as shown in Table 3.3 are declared globally and initialized in a header file called *myglobal.h*. Figure 4.9 shows this header file. Once a variable is declared in this header file, it becomes accessible from any gate and activity in the model. A detailed explanation of how this global variables declaration works is given in Appendix B.

Step 1: Storing the past history of fault inter-arrival time. As described in the previous chapter, the past history of fault inter-arrival time is recorded in four variables, namely n, m_1, m_2, and m_3. These four variables are declared and initialized to zeros in myglobal.h header file. In order to calculate each inter-arrival time of a fault, the SAN model needs to record two simulation time-stamps. These two time-stamps will be the starting and ending time of the fault inter-arrival time. The time at which activity *fault* is enabled is the first time-stamp, and the time of the occurrence of the last fault is the second time-stamp. The time interval between the two time-stamps is regarded as the interval of time between two consecutive fault arrivals. It is important to note that the two time-stamps are simulation time. They are not real time. This is the reason why we use these two time-stamps to calculate each fault inter-arrival time. During the simulation, after every occurrence of a fault, the recovery process, either the rollback recovery or the restart recovery, is executed. If we use the occurrence of the previous fault as the first time-stamp, we are including the recovery time in the calculation of the inter-arrival time of a fault. This is certainly not correct. In order to exclude the recovery time, the model regards the simulation time at which the activity *fault* is enabled as the first time-stamp.

Figure 4.10 shows the two time-stamps on the simulation time line. The line represents the simulation time line, and each dot represents the point in the simulation when a specific event happens. The events are described in the boxes below the dots. The thick line represents the correct inter-arrival time of a fault in the simulation process. The task of recording the first time-stamp is given to the input gate *enablefault*. The second time-stamp is recorded by the output gate *fault_event*. The *UltraSAN* trick to access these simulation times is described in Appendix B. The time interval between the last two fault arrivals is stored in the variable *last_fault_interval* which is declared in the header file myglobal.h.

Once we have the time interval between the last two fault arrivals, the first, second and third moments (m_1, m_2, and m_3) and the fault counter (n) are updated using the equations 3.1 - 3.4. This step is executed every time a fault is detected.

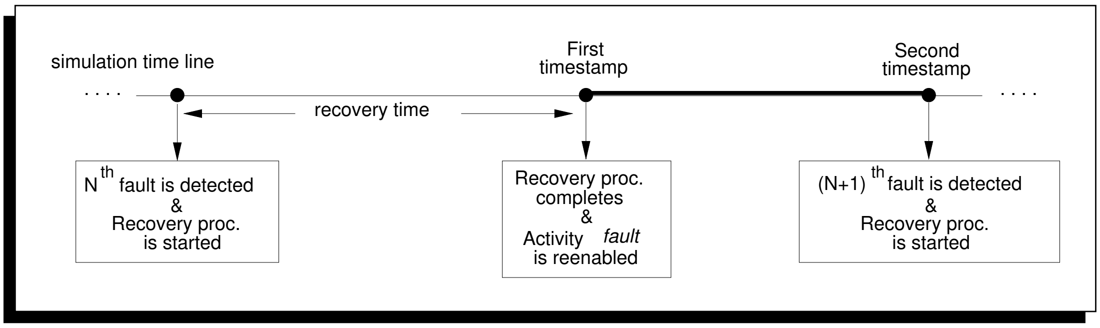


Figure 4.10: The simulation time line between two consecutive fault arrivals

Step 2: Estimating the two fault rates. Equations 3.14 and 3.15 show how to estimate the two fault rates (λ_1 & λ_2) from the three moments (m_1, m_2, m_3). The two equations are implemented inside the *fault_event* gate and are executed every time a new fault is detected. The calculations are executed after the three moments are updated to take into account the last detected fault (Step 1). The estimated values of λ_1 and λ_2 are then stored in the variables *EstimatedLambda1* and *EstimatedLambda2*, which are declared in myglobal.h header file. These global variables (*EstimatedLambda1* and *EstimatedLambda2*) will be used later in the execution of Step 3 and 4 of the algorithm.

Step 3: Determining the fault mode the system is currently in. Unlike step 1 and 2, which are executed every time a fault is detected, step 3 and 4 of the adaptive algorithm are executed periodically and are independent of the arrival of the fault event. The rate of the activity *adapt* (λ_{adapt}) determines the frequency of executing these two steps. In this model this rate is set equal to the processing rate ($\lambda_{adapt} = \lambda_{proc}$). Each time the activity *adapt* is completed, the function inside the output gate *adapttime*, which consists of the fault mode estimation rule (Step 3) and the optimization of checkpoint rate (Step 4), is executed. The function inside the gate *adapttime* is given in the Appendix C.

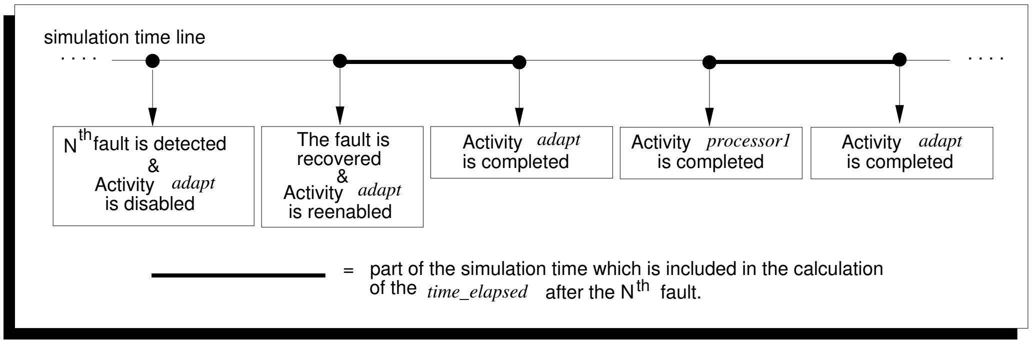


Figure 4.11: The simulation time line and parts of the time_elapsed

One variable that we need to execute the fault mode estimation rule is the *time_elapsed*, that is the time that has elapsed since the last occurrence of fault. To calculate the *time_elpased* from the simulation time line, we need to sum up the time intervals between the time at which the activity *adapt* is enabled and the time at which the activity *adapt* is completed since the last occurrence of a fault. Figure 4.11 shows the simulation time line and some fractions of it that can be considered as the *time_elapsed*. The task of recording the time at which the activity *adapt* is enabled is given to the gate *enableAdapt*, and the task of recording the time at which the activity *adapt* is completed is given to the gate *adapttime*. Once the current fault mode is estimated, the associated value of the mode (1 for fault mode 1 and 2 for fault mode 2) is placed as the marking of the place *est_fmode*.

Step 4: Setting the checkpointing rate of the system to its optimum value. Once we have estimated the two fault rates (Step 2) and determined the current fault mode (Step 3), we can determine the current fault rate (λ) by using equation (3.16). The next step of the algorithm is to get the optimum checkpoint rate value from the optimum checkpoint rate table according to the value of λ. The optimum checkpointing rate table is declared as two arrays of floating points in the myglobal.h header file. The first array, *OptTable_faultrates[]*, contains a varying value of fault rates. The second array, *OptTable_chkptsrate[]*, is associated with the first array in the following way: its element in each row is the optimum checkpointing rate value of the system whose current fault rate is as shown in the first array in the same row. These two arrays are filled with the fault rates and the optimum checkpoint rates, which are the results of the first model described in the previous section of this chapter (multiprocessor model).

By comparing the fault rates in the optimum checkpoint rate table with the current fault rate (λ) as described earlier, we can determine the specific row of the table which contains the optimum checkpoint rate of the current value of λ. The index value of this row is then recorded as the marking of the place *est_frate*. The processor activity then looks at this place to get the index value and set the optimum checkpoint rate as its checkpoint rate.

4.3 FIXED MODEL

This model is very similar to the adaptive model described in the previous section. The only difference in this model is in its checkpointing scheme. Instead of using the adaptive checkpointing scheme this model determines its checkpoint rate by using the value of a global variable called the "GLOBAL_D(chkpoint)." The value of this global variable is fixed to a single checkpoint rate. With this model the performance of the system with this fixed checkpointing scheme is evaluated. By varying the value of "GLOBAL_D(chkpoint)" we can evaluate the performance of the system for different values of checkpoint rate. In the chapter on results we compare the performance of the system with the fixed checkpointing scheme with the performance of the system with the adaptive checkpointing scheme. Figure 4.12 shows the SAN model of the fixed model.

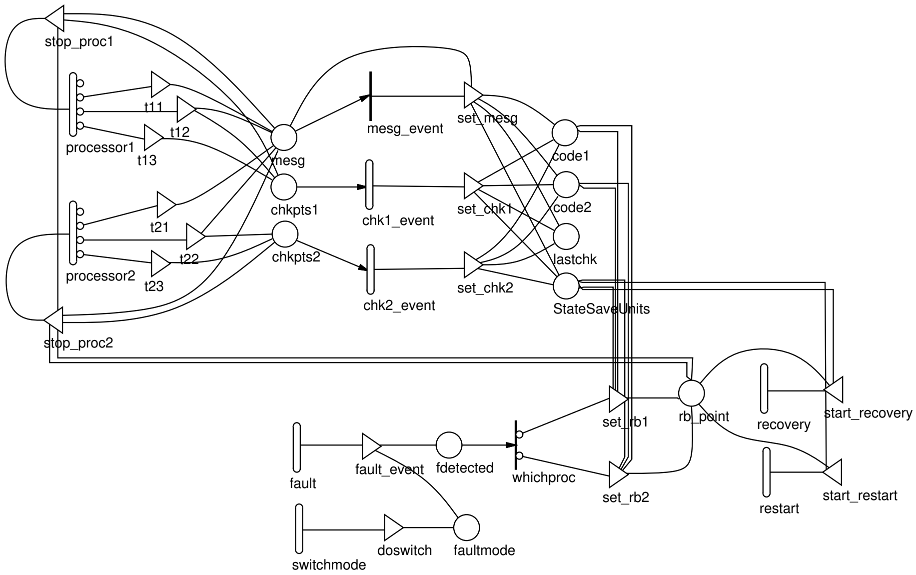


Figure 4.12: SAN-based model of the fixed model

Perf. Vars.	*Meaning: fraction of time*	*UltraSAN syntax*
forward prog.	processor1 in normal state	MARK(rb_point)==0 && MARK(chkpts1)==0
state saving	processor1 doing chkpts. proc.	MARK(chkpts1) > 0
rollback	sys. doing rollback recovery	MARK(rb_point) > 0 && MARK(rb_point) < MARK(StateSaveUnits)+1
restart	sys. doing restart recovery	MARK(rb_point) == MARK(StateSaveUnits)+1

Table 4.2: Performance variables of the SAN models

4.4 PERFORMANCE VARIABLES

In order to evaluate how well the system performs, we need measures of system performance. The measures that will be used in the models are shown in Table 4.4. The table lists the name of each performance variable, its meaning, and the *UltraSAN* syntax for each variable. Each variable represents the fraction of time the system is doing a specific process. The *UltraSAN* representation for each variable is shown on the right most column.

Forward progress, which represents the fraction of time spent by the system performing useful work, is the variable used to find the optimum checkpoint rate for different fault rates. A checkpoint rate is said to be optimum if the resulting forward progress is maximum. State saving is the overhead cost associated with doing checkpoint insertion. This cost, obviously, increases as checkpoint rate increases. Rollback and restart represent recovery costs. These costs should decrease as checkpoint rate increases.

One performance variable mentioned earlier that is not listed in Table 4.4 is the coverage of rollback recovery, which is the probability of successful recovery without executing the restart process. The coverage of rollback variable can be calculated by first calculating the probability of restart when a recovery is attempted, which is derived as follow:

$$\begin{aligned}
P[restart] &= \frac{the\ \#\ of\ restarts\ attempted}{the\ \#\ of\ fault\ occurrences} \\
&= \frac{the\ time\ spent\ in\ doing\ restart/the\ expected\ time\ of\ each\ restart}{the\ time\ spent\ in\ normal\ state/the\ expected\ time\ of\ fault\ interarrival} \\
&= \frac{restart/\frac{1}{\lambda_{restart}}}{(1 - rollback - restart)/\frac{1}{\lambda}} \\
&= \frac{restart \times \lambda_{restart}}{(1 - rollback - restart) \times \lambda},
\end{aligned}$$

where restart and rollback are the performance variables, and $\lambda_{restart}$ and λ are the rate of activity *restart* and *fault* in the SAN model. The expected time of each restart process and the expected time between two occurrences of a fault are $\frac{1}{\lambda_{restart}}$ and $\frac{1}{\lambda}$, respectively, because both processes are exponentially distributed with their corresponding rates $\lambda_{restart}$

and λ. The coverage of rollback recovery, which is the probability of no restart during recovery, can be calculated as 1.0 - P[restart].

In the second model, the fault model has two modes, each of which has its own fault rate. To calculate the probability of restart of the second model, the following equation is used:

$$P[restart] = \frac{restart \times \lambda_{restart}}{(1 - rollback - restart) \times ((\lambda_1 \times mode1) + (\lambda_2 \times mode2))},$$

where λ_1 and λ_2 are the fault rate of the first and second fault mode, and $mode1$ and $mode2$ are performance variables that represent the fraction of time the system is in the first and second fault mode, respectively. The derivation of this equation is very similar to the previous equation, except the inter-arrival fault rate (λ) becomes the combination of the two inter-arrival fault rates from the two fault modes, which is $\lambda_1 \times mode1 + \lambda_2 \times mode2$.

UltraSAN offers two methods to solve the performance variables. The first solution is called *analytical solution*. This solution requires *UltraSAN* to generate a reduced base model (Markov reward model)[29] which represents the behavior of the SAN model and supports the specified performance variables. Because of this requirement an analytical solution requires a lot of memory space. The advantage of this solution, however, is that it gives an exact answer for each performance variable defined. Another solution is called the *simulation solution*. This solution does not require a lot of memory. However, it gives only an estimation of the performance variables instead of exact answers. Each result from the simulation solution is, therefore, associated with a confidence interval. The confidence interval of all simulation runs in this thesis is set to 95%, which means for all simulation runs we can say with 95% confidence that the answers are correct within the specified range. Solution by simulation also tends to take a longer time compared to that for the analytical solution. In this work we use both the analytical and simulation solutions. The analytical solution is used to solve the performance variables of the multiprocessor model, except when the number of stable storages is greater than two, in which case the simulation solution is used. The simulation solution is also used to solve the performance variables of the adaptive model.

CHAPTER 5

RESULTS

This chapter presents the results that were obtained from the SAN models described in the previous chapter. Theoretically, the models can be used to analyze the performance of a two-processor system with any number of stable storages, but the state space and execution time grow very rapidly as we increase the number of stable storages. The analysis is therefore limited by the memory constraint of the computer used to do the evaluation. As mentioned earlier, for the analytical solution of the multiprocessor model the number of stable storages is limited to two. Solution by simulation is used for the model with stable storages larger than two since the state space size increases rapidly when larger stable storages are considered.

This chapter is divided into two sections. The first section presents the results of the multiprocessor model, which gives the optimum checkpointing rates of the system for fixed fault arrival rates. The section also discusses some system variables that affect the value of the optimum checkpoint rate. The results of varying these variables are shown. The resulting optimum checkpoint rates of this model are then used in the second model to initialize the optimum checkpoint rate table, which is used in the execution of the adaptive checkpointing algorithm.

The second section presents the forward progress and coverage of multistep rollback measures of the system with fixed and adaptive checkpointing schemes. Comparison studies between the two schemes are given. The results in the second section are generated by the *UltraSAN* simulation solution, called *steady state simulator*.

5.1 RESULTS OF THE MULTIPROCESSOR MODEL

This section presents the analytical and simulation results of the performance of the two-processor system. The parameters for all sets of runs in this section, unless otherwise stated, are as follows:

- Stable storage = 2.
- Processing rate (λ_{proc}) = 10.
- Fault rate (λ) = 0.001 - 0.030.
- Checkpoint rate ($\lambda_{chkpoint}$) = 0.2 - 2.0.
- Message transmission rate (λ_{mesg}) = 1.0 for each processor.
- Restart rate ($\lambda_{restart}$) = $1/100 \times \lambda_{proc}$ (i.e. the cost of restart is 100 $\times$ processing unit).
- State saving rate = $10 \times \lambda_{proc}$ (i.e. the cost of checkpointing is $1/10 \times$ processing unit).

The processing rate of each processor is the same (λ_{proc}). As mentioned earlier, this rate (the processing unit) is the "basic unit" of the other rates.

As mentioned earlier, the optimum checkpoint rate exists because there are trade offs concerning the checkpoint rates. The recovery cost decreases as checkpoint rate is increased,

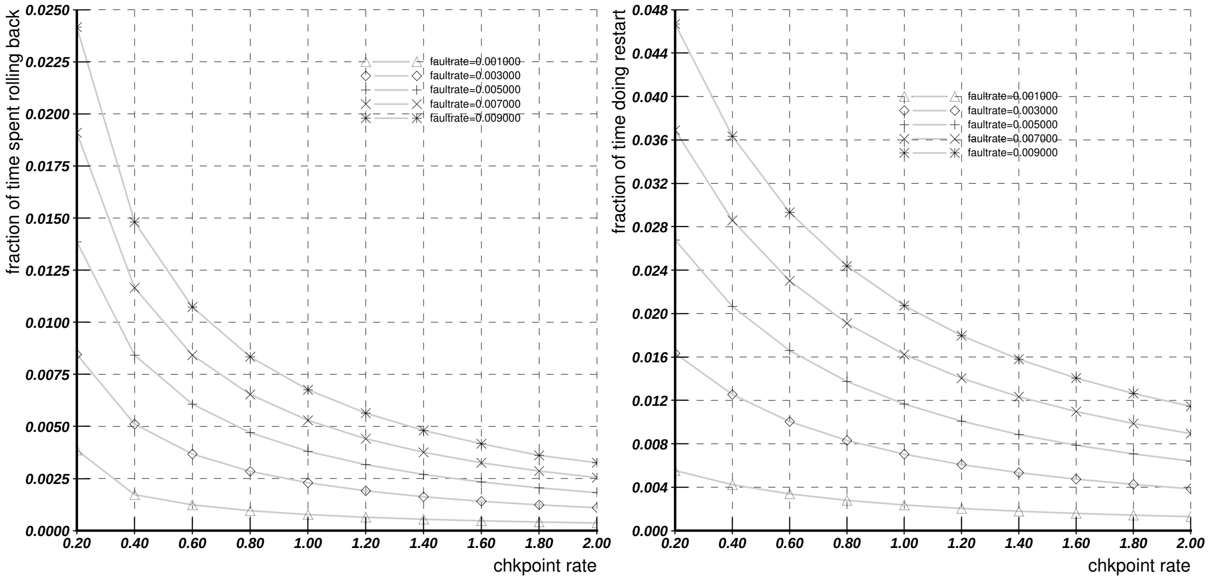


Figure 5.1: Graph of rollback costs (left) and restart costs (right)

(Stable storage = 2, $\lambda_{mesg} = 1.0$, $\lambda_{restart} = 0.1$)

while state saving cost increases as checkpoint rate is increased. Figure 5.1 shows the graphs of the recovery costs as checkpoint rate is varied. The recovery costs consist of the rollback cost and the restart cost. Each of these costs is shown as a separate graph in Figure 5.1. Figure 5.2 shows the state saving cost as checkpoint rate is varied. As explained earlier, these costs represent the fraction of time the system is doing the associated process. For example, restart cost represents the fraction of time the system is doing restart process.

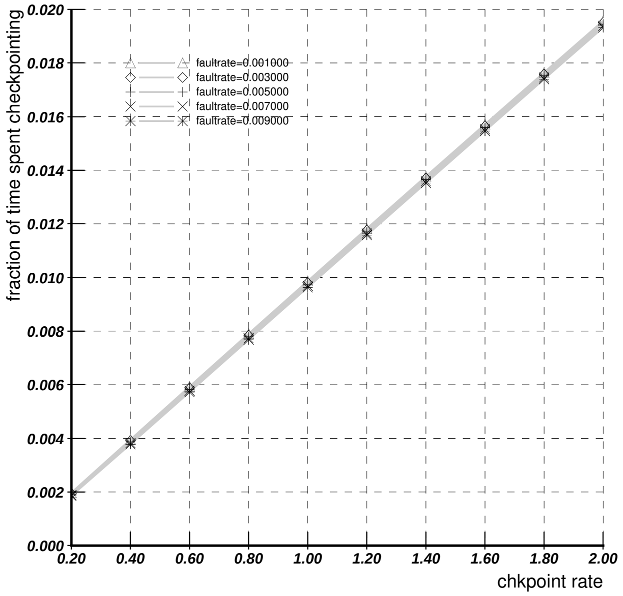


Figure 5.2: Graph of state saving costs vs. checkpoint rate
(Stable storage = 2, $\lambda_{mesg} = 1.0$, $\lambda_{restart} = 0.1$)

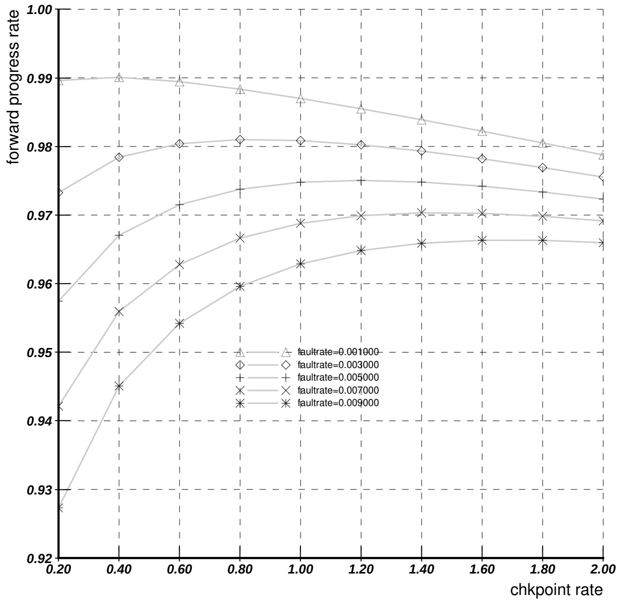


Figure 5.3: Graph of forward progress vs. checkpoint rate
(Stable storage = 2, $\lambda_{mesg} = 1.0$, $\lambda_{restart} = 0.1$)

The trade offs on varying the checkpoint rate described above imply the existence of an optimum checkpoint rate, which is the checkpoint rate at which the system achieves the most rapid forward progress. The graph of the forward progress as checkpoint rate is varied is shown in Figure 5.3. The graph shows the curves of forward progress for different fault rates. The optimum checkpoint rate for different fault rates is represented by the top point of each curve. To the left of the highest point the curve goes down as it suffers more and more restart cost. To the right of the highest point the curve goes down as it suffers more and more state saving cost. Since the restart cost is much larger than the state saving cost, the slope to the left of the highest point is steeper compared to the slope to its right. Also notice that curves corresponding to the lower fault rates have steeper slopes. This is because as the fault rate is reduced, the cost of state saving is increased as more checkpoints are wasted.

Another interesting point from the forward progress graph above is the fact that the optimum checkpoint rate, especially for larger fault rates, is close to the message rate (optimum checkpoint rate of $\lambda = 0.007$ is 1.5; the message transmission rate is $\lambda_{mesg} \times 2$ (# of processor) = 2.0). This is rather suspicious, because it means a checkpoint should be inserted about every time a message is sent. Further study showed that this behavior is caused by the low coverage of rollback recovery (i.e. high probability of restart) of the system. The coverage of rollback recovery of the above set of runs is 0.5373. The probability of restart should be kept low (less than 0.05 [24]). There are several reasons that might cause a low coverage. The first reason is the small number of stable storages in the model. In real systems the number of stable storages is very large, therefore the probability of restart caused by the checkpoint being exhausted is very small. The second reason is that there is no garbage collection implemented in our model. Wang et al. [33] proved that the number of *useful checkpoint* is limited given that the garbage collection to discard *garbage checkpoints* is implemented. Our model does not consider garbage collection but has a limited number of stable storages.

To further study the relationship of the coverage of rollback recovery to the value of the optimum checkpoint rate, we shall vary the system parameters that might affect its coverage measure and analyze the resulting optimum checkpoint rates. Two system parameters that have a direct effect on the measure of the coverage are: 1) the number of stable storages in each processor, and 2) the frequency of message transmission (λ_{mesg}). The number of stable storages obviously determines the probability of restart during recovery. The more the stable storage the processor has, the more checkpoints it can hold, so it is less likely that checkpoints are exhausted during the recovery process. The rate of message transmission affects the coverage since each message transmission adds the possibility of rollback propagation. When the rate of message transmission is high, the probability of rollback propagation during the recovery process is increased, leading to an increase in the probability of restart.

The forward progress graphs resulting from varying the two parameters are presented below. We first study the system with varying number of stable storages, and then study the system with varying message transmission rate. In both cases the resulting optimum checkpoint rates are analyzed. After these studies, we also look at the effects of varying restart cost and state saving cost on the optimum checkpoint rates.

Varying the number of stable storages: Figure 5.4 shows the graph of the coverage measure as the number of stable storages is varied. The calculation of the coverage

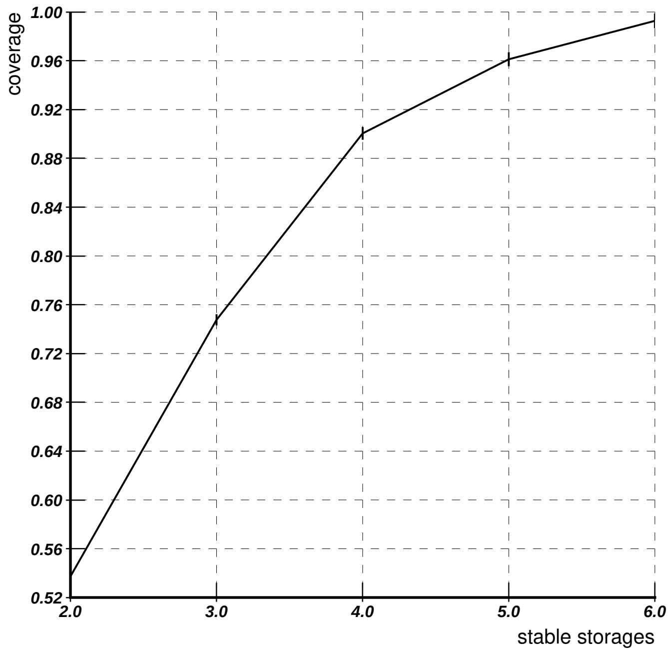


Figure 5.4: Graph of coverage vs. stable storages

$(\lambda_{mesg} = 1.0,\ \lambda_{restart} = 0.1)$

of rollback recovery is as explained in the previous chapter. This graph is the result of the simulation solution except for the stable storage equals to two, which is calculated analytically. The bar at each point, as discussed in the previous chapter, represents the 95% confidence interval. The largest number of stable storages in this run is 6 because the simulation time becomes very long as we increase the number of the stable storages to 7. The graph shows that the coverage of rollback recovery increases very rapidly as the number of stable storages increases.

To analyze the resulting optimum checkpoint rate as the number of stable storages is varied, we compare the resulting forward progress graphs of system with 2 and 6 stable

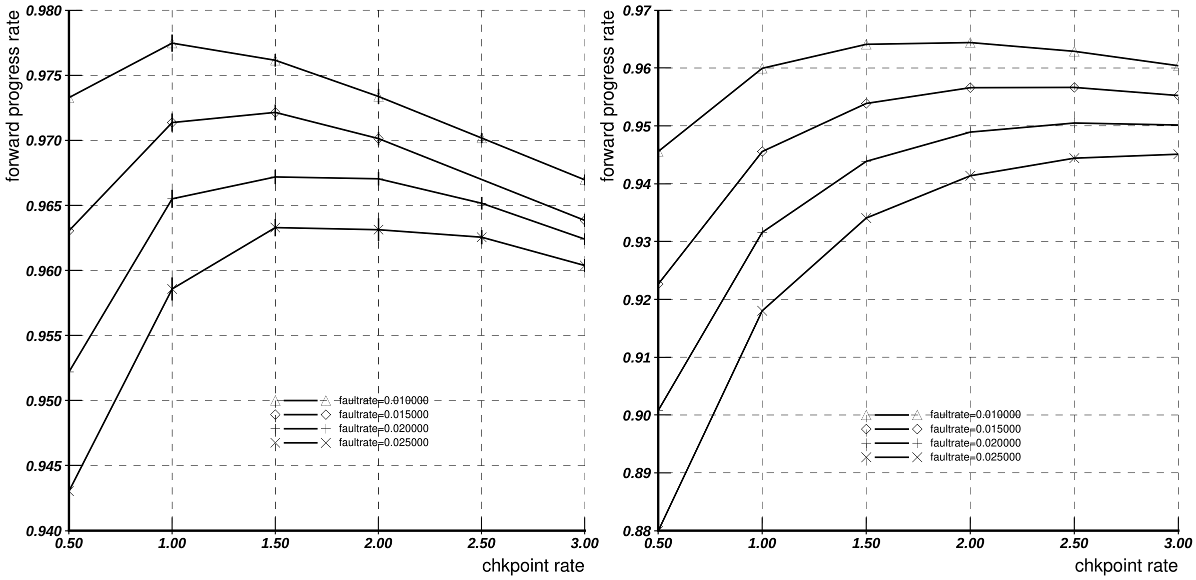


Figure 5.5: Graph of forward progress of system

with 6 (left) and 2 (right) stable storages ($\lambda_{mesg} = 1.0$, $\lambda_{restart} = 0.1$)

storages. Figure 5.5 shows these two graphs. The graph on the left is the forward progress graph of the system with 6 stable storages, and the one on the right is that of the system with 2 stable storages. The forward progress graph of the system with 2 stable storages is different than the one shown in Figure 5.3 because the fault rates are different. The graph in Figure 5.3 varies the fault rates from 0.001 to 0.009, while the right graph in Figure 5.5 varies the fault rates from 0.010 to 0.025.

By comparing the two graphs in Figure 5.5 we observe the following:

1. Forward progress increases as more stable storage is added.

2. The optimal checkpoint rate decreases as stable storages increases.

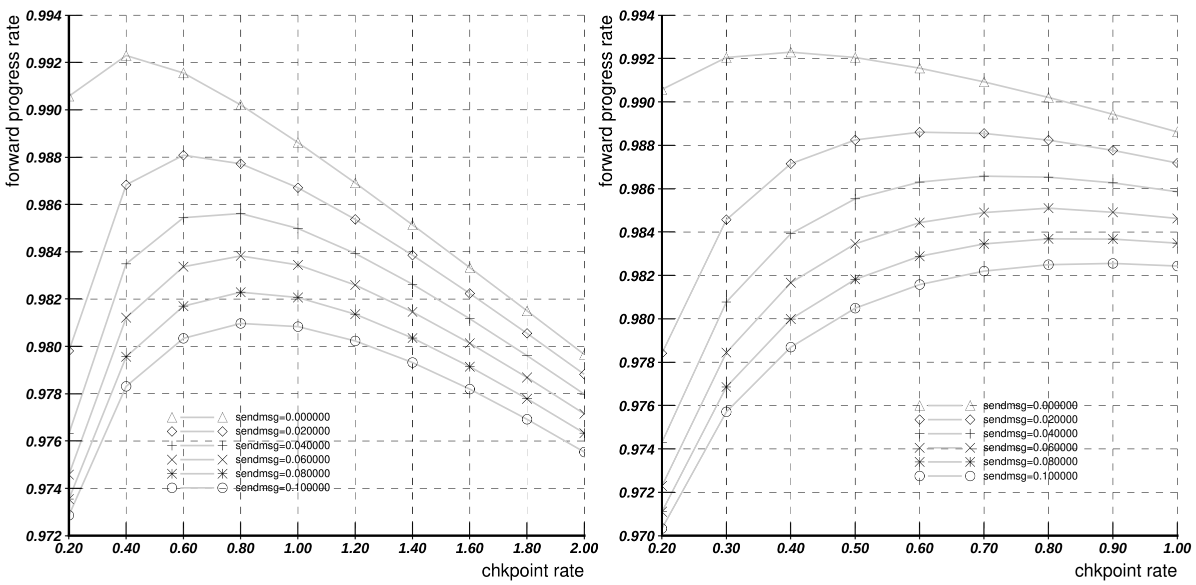


Figure 5.6: Graphs of forward progress for varying message rate

SS = 2 (left) and SS = 3 (right) ($\lambda = 0.003$, $\lambda_{restart} = 0.1$)

The first observation is obvious and is clearly shown by the graph. For example, for fault rate equals 0.010, the maximum forward progress of the system with 6 stable storages is 0.9775 ± 0.0007 , and the maximum forward progress of the system with 2 stable storages is 0.9644. The second observation shows that the change in the number of stable storages does affect the optimum checkpoint rate as discussed earlier. The optimum checkpoint rates of the system with 6 stable storages are lower than those of the system with 2 stable storages. For example, for fault rate equals 0.020, the optimum checkpoint rate of the system with 6 stable storages, as shown by the graph, is between 1.50 and 2.00, and that of the system with 2 stable storages is 2.50.

Varying the message transmission rate: Figure 5.6 shows the forward progress graphs for varying message rate. The graph on the left is for the system with 2 stable storages and the one on the right is for the system with 3 stable storages. In both graphs the message transmission rate is varied from 0.00 to 0.10. This range is not the probability value of doing message transmission (P_{mesg}), it is the rate of message transmission, which is calculated as $P_{mesg} \times$ the processing unit (λ_{proc}). A message rate of 0.00 means "no message transmission has ever occurred." Both graphs show that the optimum checkpoint rates are increased as the message transmission rate is increased. This makes sense because by increasing the message transmission rate, the probability of restart is increased, so the forward progress is reduced. To keep the forward progress maximized the optimum checkpoint rate should, therefore, be increased. In the left graph, where stable storage is set to 2, however, the changes in the optimum checkpoint rate are unnoticeable for the curves whose message rates are above 0.05. This is because the probability of restart is getting so high that increasing the checkpoint rate does not make the forward progress any better. By increasing the stable storage to three, as in the right graph, the probability of restart is reduced and the optimum checkpoint rate keeps increases for all curves as message rate increases.

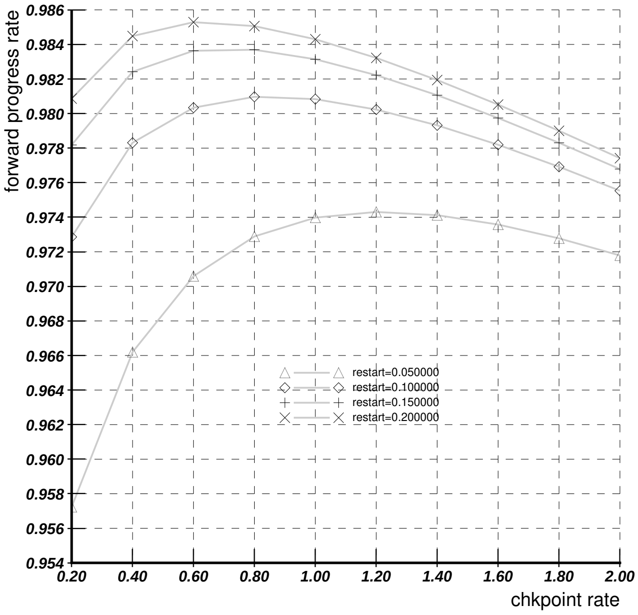


Figure 5.7: Opt. checkpoint graph for varying restart costs ($\lambda = 0.003$,$\lambda_{mesg} = 1.0$)

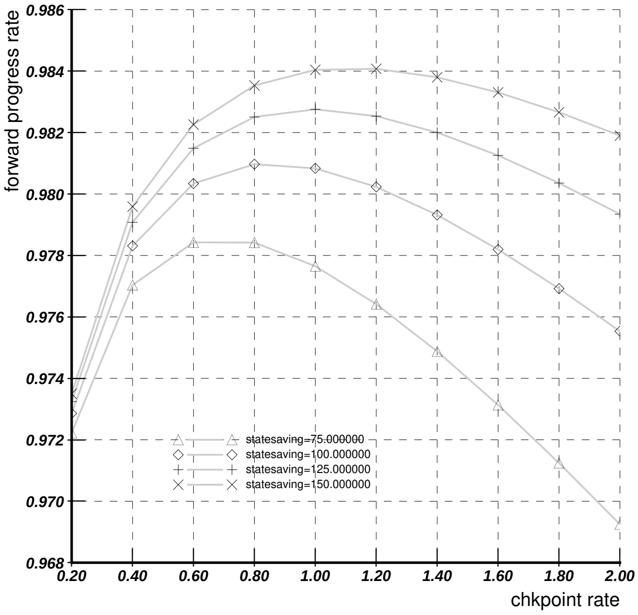


Figure 5.8: Opt. checkpoint graph for varying state saving costs ($\lambda = 0.003$,$\lambda_{mesg} = 1.0$)

Varying the restart cost: Even though the restart cost does not have any effect on the probability of restart, it is interesting to study its effect on the optimum checkpoint rates. Figure 5.7 shows the forward progress graph as the restart cost is varied. In this set of runs the fault rate (λ) is set to 0.003 and the restart rate is varied from 0.05 to 0.20, which means varying restart costs from 200 to 50 times the processing unit ($\lambda_{proc} = 10$). As the restart cost is increased, the resulting forward progress is obviously decreased. Lower forward progress is caused not only by higher restart cost but also by the additional state saving required to maintain the optimum performance. As the cost of restart is increased, the system tries harder to avoid restart by adding more checkpoints, and thus the optimum checkpoint rate is increased.

Varying the state saving cost: Another variable affects the optimum checkpoint rate is the cost of state saving. This is shown by Figure 5.8, on which state saving cost is varied from $\frac{1}{150}$ to $\frac{1}{75}\times$ the processing unit time. As state saving cost is increased, both the optimum checkpoint rate and the forward progress are decreased. The optimum checkpoint rate is decreased because more overhead cost is obviously involved for each checkpoint insertion. The forward progress is decreased mainly because the system suffers more restart cost as checkpoint rate is decreased.

As the summary of this section, below are the conclusions of the studies we have done in this section:

1. The optimum checkpoint rates of the multiprocessor system for fixed fault rates are calculated. These rates are obtained by varying the system checkpoint rate and analyzing the resulting forward progress graphs. An example of the optimum checkpoint

Fault Rate	*Opt. Chkpts. Rate*
0.001	0.4
0.003	0.8
0.005	1.2
0.007	1.4
0.009	1.8

Table 5.1: The optimum checkpoint rate table of system with SS=2

rate table constructed from the resulting forward progress graph, in this case the forward progress graph in Figure 5.3, is shown in Table 5.1.

2. The optimum checkpoint rates are sensitive to the following system parameters:

   - The number of stable storages in each processor: the optimum checkpoint rate decreases as the number of stable storages increases.
   - The message transmission rate: the optimum checkpoint rate decreases as the message transmission rate decreases.
   - The restart cost: the optimum checkpoint rate decreases as the restart cost decreases.
   - The state saving cost: the optimum checkpoint rate decreases as the state saving cost increases.

3. The optimum checkpoint rates are close to the message transmission rate because the coverage of rollback recovery of the system is very low. The optimum checkpoint rate decreases (i.e. moves away from the message transmission rate) as the coverage of rollback recovery, which is sensitive to the number of stable storages and the message transmission rate, is increased.

5.2 RESULTS OF THE ADAPTIVE AND FIXED MODEL

This section presents the comparisons between the adaptive and the fixed checkpointing scheme. The first part of this section highlights the superiority of the adaptive checkpointing scheme over the fixed checkpointing scheme. Later in the section some comparisons between both schemes for different system parameter settings are discussed. Since the performance (forward progress) of the fixed checkpointing scheme changes as the checkpoint rate is varied, the comparisons throughout this section use the peak performance of the fixed checkpointing scheme.

Unless otherwise stated, below are the parameter settings of the adaptive and fixed model used in the sets of runs in this section:

- Stable storages = 2.
- Processing rate (λ_{proc}) = 10.
- Fault rates: $\lambda_1 = 0.003$ & $\lambda_2 = 0.030$.
- Checkpoint rate of the fixed scheme ($\lambda_{chkpoint}$) = 0.2 - 2.5.
- Message transmission rate (λ_{mesg}) = 0.5 for each processor.
- Restart rate ($\lambda_{restart}$) = $1/100 \times \lambda_{proc}$ (i.e. the cost of restart is $100 \times$ processing unit).
- State saving rate = $10 \times \lambda_{proc}$ (i.e. the cost of checkpointing is $1/10 \times$ processing unit).

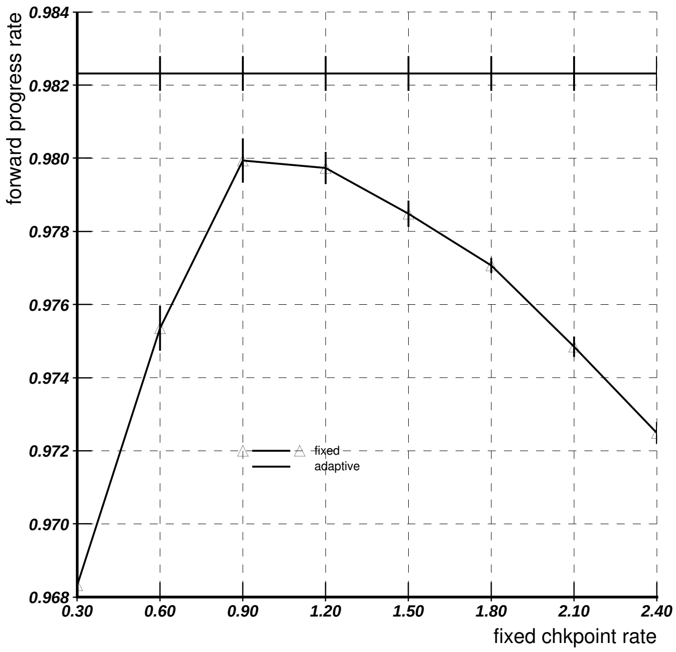


Figure 5.9: Forward progress of Fixed and Adaptive scheme

(λ_{mesg} = 0.5, λ_1 = 0.003, λ_2 = 0.03, SS = 2)

As explained earlier, the processing unit represents the rate of processing a task. The expected time of processing a single task is, therefore, represented by $\frac{1}{the\ processing\ unit} = \frac{1}{\lambda_{proc}} = 0.1$. If we assume an "hour" to be our unit time, then the expected time to process a task is 0.1 hours (6 minutes, which is a typical time to process a task [27]). By using the same derivation, we have the expected inter-arrival time of fault from the first fault mode equals $\frac{1}{\lambda_1} = \frac{1}{0.003} = 333$ hours. This expected inter-arrival time of fault (333 hours) is taken from an empirical study done by Siewiorek and Swarz [18]. In their study, the distribution of transient faults is modeled as exponential distribution with rate ranging from 0.0018 (mean = 556 hours) to 0.0065 (mean = 154 hours), and they measured the average inter-arrival time to be about 350 hours. The data was collected for about 15000 hours (1.7 years). The second rate of fault inter-arrival in our model is set 10 times faster than the first rate. Meyer and Wei [30] derived a formula for $MTTF$ (Mean Time To Failure) from system workload and showed that a system fault rate can easily get 10 times faster as workload changes.

The forward progress of the adaptive and fixed model: Figure 5.9 shows the performance (forward progress) of the system with the adaptive and fixed checkpointing scheme. The curve shows the performance of fixed checkpointing schemes for different checkpoint rates. It is clear from the graph that the adaptive checkpointing scheme outperforms the best performance of the fixed checkpointing scheme. The forward progress of the system with the adaptive scheme is 0.9823 ± 0.0003, and the highest forward progress of the system with the fixed scheme is 0.9799 ± 0.0004.

Another way to compare the performance between the two checkpointing schemes is by comparing the number of stable storages needed for both systems to have equal forward progress. Figure 5.10 shows the forward progress curves of the system with the fixed checkpointing scheme with the number of stable storages equals to 2 and 3. The peak forward progress of the curve with 3 stable storages is close to the forward progress of the system with the adaptive checkpointing and 2 stable storages. This graph tells us that for the system with the fixed checkpointing scheme to achieve equal forward progress as the system with the adaptive checkpointing scheme, it needs to increase its number of stable storages. This shows a significant advantage of an adaptive checkpointing scheme since adding a unit of stable storage means adding more hardware cost to the system.

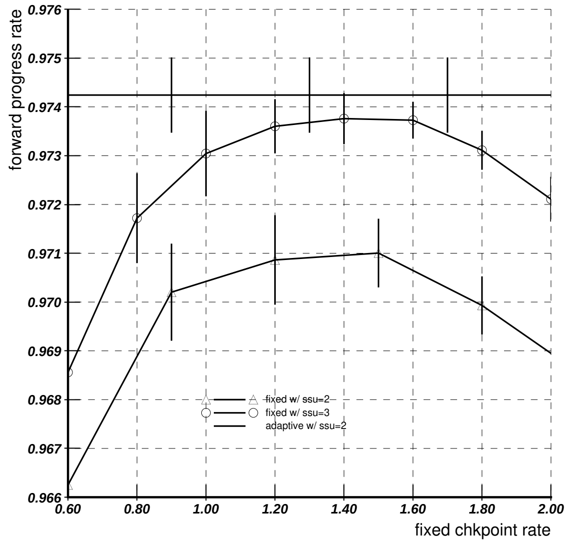


Figure 5.10: Forward progress of Fixed and Adaptive scheme

($\lambda_{mesg} = 0.5$, $\lambda_1 = 0.003$, $\lambda_2 = 0.03$)

Varying the message transmission rate: Figure 5.11 shows the graphs of the system with the two checkpointing schemes with 0.5 message transmission rate (left graph) and with 1.5 message transmission rate (right graph). The left graph is the same as the one in Figure 5.9. The right graph shows similar graph for system with different message transmission rate. In this graph, the forward progress of the system with the adaptive scheme is 0.9742 ± 0.0005, and the highest forward progress of the system with the fixed scheme is 0.9710 ± 0.0004. In both graphs the system with the adaptive checkpointing scheme performs better than the system with the fixed checkpointing scheme. An interesting difference between the two figures, however, is the distance between the highest forward progress of the fixed scheme and the forward progress of the adaptive scheme. When message transmission rate equals 1.5, the distance is 0.0032 ± 0.0009; when message transmission rate

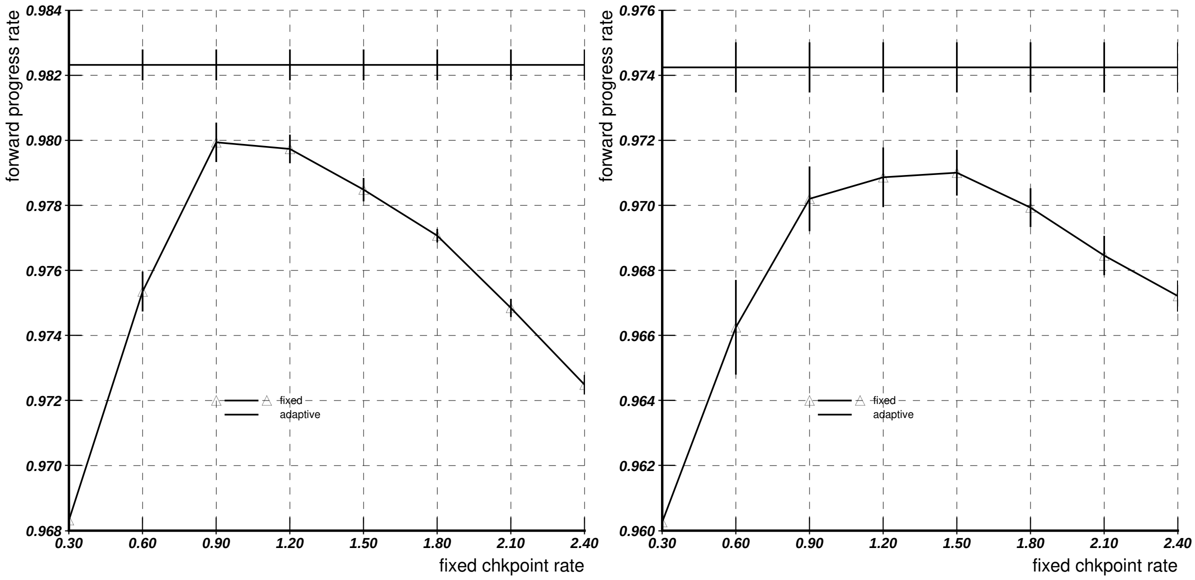


Figure 5.11: Forward progress graphs with different message rate

(Left: $\lambda_{mesg} = 0.5$, Right: $\lambda_{mesg} = 1.5$) ($\lambda_1 = 0.003$, $\lambda_2 = 0.03$, SS = 2)

equals 0.5, the distance is 0.0020 ± 0.0007. This is because as message transmission rate decreases, the probability of rollback propagation decreases, and thus adaptive checkpointing scheme becomes less significant. If message transmission rate is very low (close to 0.0), the optimal performance of fixed scheme will be very close to the performance of adaptive scheme. We can, therefore, conclude that the performance of adaptive checkpointing scheme is much better compared to the fixed checkpointing scheme in systems with high message transmission rate.

Varying the number of stable storages: Figure 5.12 shows the graphs of the system with 2 and 6 stable storages. In both graphs the message transmission rate is 1.5. In the left graph (SS = 2), the forward progress of the system with the adaptive scheme is

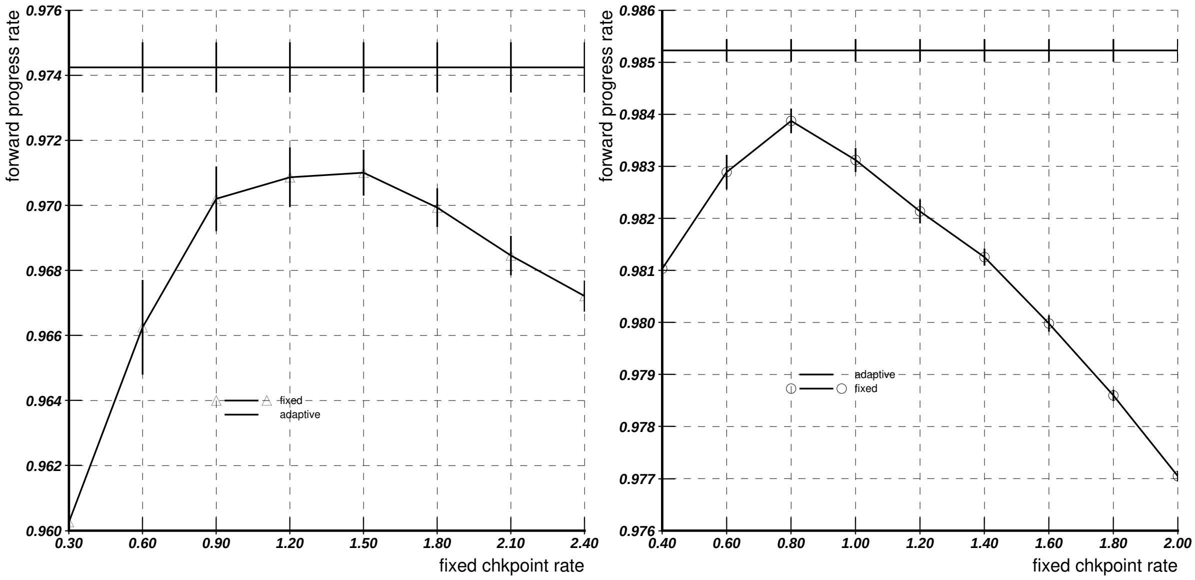


Figure 5.12: Forward progress graphs (Left: SS = 2, Right: SS = 6)

(λ_{mesg} = 0.5, λ_1 = 0.003, λ_2 = 0.03)

0.9742 $\pm$ 0.0005, and the highest forward progress of the system with the fixed scheme is 0.9710 $\pm$ 0.0004. The difference between the two forward progress is 0.0032 $\pm$ 0.0009. In the right graph (SS = 6), the forward progress of the system with the adaptive scheme is 0.9852 $\pm$ 0.0002, and the highest forward progress of the system with the fixed scheme is 0.9839 $\pm$ 0.0002. The difference between the two forward progress is 0.0013 $\pm$ 0.0004. By comparing the performance difference between the two methods when SS = 2 and when SS = 6, we notice that the level of superiority of the adaptive scheme is lower when the number of stable storages is high, and it is higher when the number of stable storages is low. As the number of stable storages is increased, the probability of restart is decreased, and thus the significance of having adaptive checkpointing is decreased. This is also showed

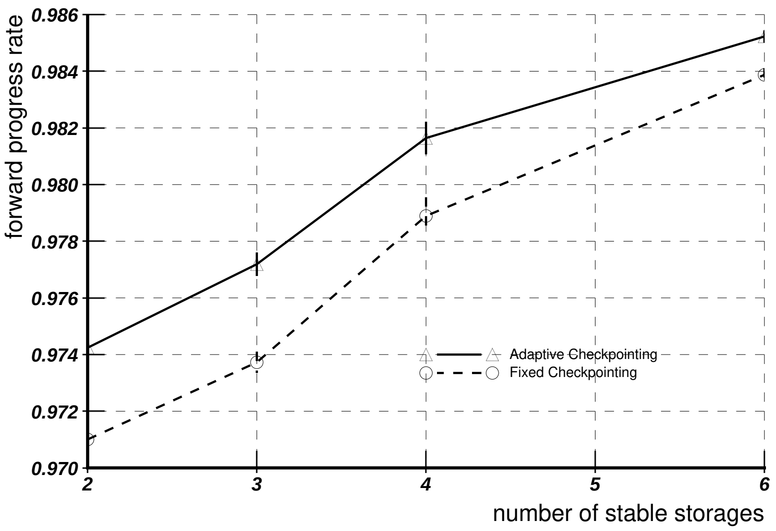


Figure 5.13: Forward progress of fixed & adaptive alg. with varying stable storages

$(\lambda_{mesg} = 0.5,\ \lambda_1 = 0.003,\ \lambda_2 = 0.03)$

in Figure 5.13, which shows the forward progress of both the fixed and the adaptive checkpointing algorithm when the number of stable storages is varied. If the number of stable storages in each module is unlimited, the probability of restart is zero; the performance of the fixed scheme would be very close to that of the adaptive scheme. From these experiments we learn that the adaptive checkpointing scheme performs much better than the fixed checkpointing scheme for the systems with small number of stable storages.

To summarize this section, here are the conclusions we can draw on the performance of the adaptive checkpointing scheme:

1. In all cases the performance of the adaptive checkpointing scheme of the systems with "Markov-modulated fault process" is better than that of the fixed checkpointing scheme.

2. The level of superiority of the adaptive scheme depends on the following factors:

   - The message transmission rate: the adaptive scheme performs much better under a high message transmission rate.
   - The number of stable storages: the adaptive scheme performs much better under a low number of stable storages.

CHAPTER 6

CONCLUSIONS

In this thesis the SAN models to determine the optimum checkpoint rate of a two-processor system and to evaluate the performance of a system with adaptive checkpointing algorithm have been presented. By defining the model at the SAN level, rather than the state level, we have been able to build a much more complex model than has previously been done. As far as we know, no one has constructed a similar model to evaluate the optimum checkpoint rate for an uncoordinated multiprocessor system.

Although the model incorporates only two processors, it enables us to study the trade off between many system parameter values. These optimum rate values turn out to be sensitive to some system parameters, namely state saving cost, restart cost, message transmission rate of each processor and the number of stable storages in each processor.

Another advantage of this multiprocessor model is that it permits the evaluation of the adaptive checkpointing scheme proposed in the third chapter. The adaptive scheme, which adapts its checkpoint rate according to the current fault rate, requires a knowledge of the optimum checkpoint rate for different fault arrival rates. This knowledge is obtainable from the multiprocessor model. The comparisons of systems with adaptive and fixed checkpointing scheme shows the significance of the adaptive algorithm; especially for a system with a small number of stable storages and high message transmission rate.

Future research on this area should attempt to simplify the model in order to reduce the state space without eliminating its capabilities. With this state space reduction the model should be able to evaluate larger systems (i.e. more interacting processors and more state space units).

Appendix A

DERIVATIONS OF THE FIRST, SECOND AND THIRD MOMENT

This appendix presents the derivations of the first, second and third moment (m_1, m_2, and m_3) of a random variable X whose density is shown below:

$$f(x) = p\,\lambda_1\,e^{-\lambda_1\,x} + (1-p)\,\lambda_2\,e^{-\lambda_2\,x}$$

Derivation of the first moment (m_1)

$$\begin{aligned}
m_1 &= E[X] \\
&= \int_0^\infty x \cdot f(x)\delta x \\
&= \int_0^\infty x \cdot p\,\lambda_1\,e^{-\lambda_1\,x} + x \cdot (1-p)\,\lambda_2\,e^{-\lambda_2\,x}\delta x \\
&= p \cdot \int_0^\infty e^{-\lambda_1\,x}\delta x + (1-p) \cdot \int_0^\infty e^{-\lambda_2\,x}\delta x \\
&= p \cdot (-\tfrac{1}{\lambda_1}e^{-\lambda_1\,x}\,|_0^\infty) + (1-p) \cdot (-\tfrac{1}{\lambda_2}e^{-\lambda_2\,x}\,|_0^\infty) \\
&= p \cdot (-\lim_{x\to\infty}\tfrac{1}{\lambda_1}e^{-\lambda_1\,x} + \tfrac{1}{\lambda_1}) + (1-p) \cdot (-\lim_{x\to\infty}\tfrac{1}{\lambda_2}e^{-\lambda_2\,x} + \tfrac{1}{\lambda_2}) \\
&= p \cdot \tfrac{1}{\lambda_1} + (1-p) \cdot \tfrac{1}{\lambda_2}
\end{aligned}$$

Derivation of the second moment (m_2)

$$\begin{aligned}
m_2 &= E[X^2] \\
&= \int_0^\infty x^2 \cdot f(x)\delta x \\
&= \int_0^\infty x^2 \cdot p\lambda_1 e^{-\lambda_1\,x} + x^2 \cdot (1-p)\lambda_2 e^{-\lambda_2\,x}\delta x \\
&= 2p \cdot \int_0^\infty x \cdot e^{-\lambda_1\,x}\delta x + 2(1-p) \cdot \int_0^\infty x \cdot e^{-\lambda_2\,x}\delta x
\end{aligned}$$

$$= 2p\frac{1}{\lambda_1} \cdot \int_0^\infty e^{-\lambda_1 x}\delta x + 2(1-p)\frac{1}{\lambda_2} \cdot \int_0^\infty e^{-\lambda_2 x}\delta x$$

$$= 2p\frac{1}{\lambda_1} \cdot (-\frac{1}{\lambda_1}e^{-\lambda_1 x} \mid_0^\infty) + 2(1-p)\frac{1}{\lambda_2} \cdot (-\frac{1}{\lambda_2}e^{-\lambda_2 x} \mid_0^\infty)$$

$$= 2p\frac{1}{\lambda_1} \cdot (-\lim_{x\to\infty} \frac{1}{\lambda_1}e^{-\lambda_1 x} + \frac{1}{\lambda_1}) + 2(1-p)\frac{1}{\lambda_2} \cdot (-\lim_{x\to\infty} \frac{1}{\lambda_2}e^{-\lambda_2 x} + \frac{1}{\lambda_2})$$

$$= 2p \cdot \frac{1}{\lambda_1^2} + 2(1-p) \cdot \frac{1}{\lambda_2^2}$$

$$= 2\ (p \cdot \frac{1}{\lambda_1^2} + (1-p) \cdot \frac{1}{\lambda_2^2})$$

Derivation of the third moment (m_3)

$$\begin{aligned}
m_3 &= E[X^3] \\
&= \int_0^\infty x^3 \cdot f(x)\delta x \\
&= \int_0^\infty x^3 \cdot p\lambda_1 e^{-\lambda_1 x} + x^3 \cdot (1-p)\lambda_2 e^{-\lambda_2 x}\delta x \\
&= 3p \cdot \int_0^\infty x^2 \cdot e^{-\lambda_1 x}\delta x + 3(1-p) \cdot \int_0^\infty x^2 \cdot e^{-\lambda_2 x}\delta x \\
&= 6p\frac{1}{\lambda_1} \cdot \int_0^\infty x e^{-\lambda_1 x}\delta x + 6(1-p)\frac{1}{\lambda_2} \cdot \int_0^\infty x e^{-\lambda_2 x}\delta x \\
&= 6p\frac{1}{\lambda_1^2} \cdot \int_0^\infty e^{-\lambda_1 x}\delta x + 6(1-p)\frac{1}{\lambda_2^2} \cdot \int_0^\infty e^{-\lambda_2 x}\delta x \\
&= 6p\frac{1}{\lambda_1^2} \cdot (-\frac{1}{\lambda_1}e^{-\lambda_1 x} \mid_0^\infty) + 6(1-p)\frac{1}{\lambda_2^2} \cdot (-\frac{1}{\lambda_2}e^{-\lambda_2 x} \mid_0^\infty) \\
&= 6p\frac{1}{\lambda_1^2} \cdot (-\lim_{x\to\infty} \frac{1}{\lambda_1}e^{-\lambda_1 x} + \frac{1}{\lambda_1}) + 6(1-p)\frac{1}{\lambda_2^2} \cdot (-\lim_{x\to\infty} \frac{1}{\lambda_2}e^{-\lambda_2 x} + \frac{1}{\lambda_2}) \\
&= 6p \cdot \frac{1}{\lambda_1^3} + 6(1-p) \cdot \frac{1}{\lambda_2^3} \\
&= 6\ (p \cdot \frac{1}{\lambda_1^3} + (1-p) \cdot \frac{1}{\lambda_2^3})
\end{aligned}$$

Appendix B

TRICKS IN *UltraSAN* IMPLEMENTATIONS

This appendix describes some "tricks" in implementating the model using *UltraSAN*. These tricks are needed to access some internal global variables in the model definitions. Two tricks described in this appendix are: 1. to access the simulation time, and 2. to declare some internal global definitions.

B.1 ACCESSING SIMULATION TIME

As mentioned earlier in the SAN model chapter, it is necessary to access the simulation time to know the exact timing of fault arrivals. This timing is needed to execute the fault rate estimation algorithm.

The global variable within the *UltraSAN* package that contains this information is called *currentTime*. Each time an activity completes this global variable is updated to current simulation time. When the output gate of the associated activity accesses this variable, it still contains the exact time of activity completion. This variable can be accessed by placing *"extern timeValue currentTime"* syntax on the top of gate definition. An example of the use of currentTime global variable is in gate *fault_event*.

B.2 INTERNAL GLOBAL VARIABLES DEFINITIONS

In order to implement the adaptive algorithm in the SAN model, we have to define some global variables that can be accessed by almost all gates in the model. These global variables are different than the *global variables* defined in study editor. We shall refer to these global variables as *internal global variables* to distinguish them from the other ones.

A header file containing all definitions of internal global variables was created, and was placed in *project_name/int* directory. We called this file *myglobal.h*. Since we are using steady state simulation for the solution method, the header file (myglobal.h) is included inside *SSim.c* file, which is located in the same project_name/int directory. Because this SSim.c file is recreated every time we do a *save* in composed model editor, the SSim.c file has to be reedited to include the header file everytime composed model editor is done with the save command. With this header file included in SSim.c, we can use these internal global variables in each gate, both input gate and output gate. For example in *fault_event* output gate, we can declare *"extern double LAMBDA1"* and use global variable LAMBDA1 inside the gate.

Appendix C

PROJECT DOCUMENTATIONS

This appendix contains the documentations of the SAN models. These documentations are generated by the *UltraSAN* software package. Every definitions of each model, starting from the SAN model itself, the performance variable definitions, and the parameter settings, are contained in the documentation.

The first section is the documentation of the multiprocessor model, the second section is the documentation of the adaptive model, and the last section is the documentation of the fixed model.

C.1 DOCUMENTATION OF THE MULTIPROCESSOR MODEL

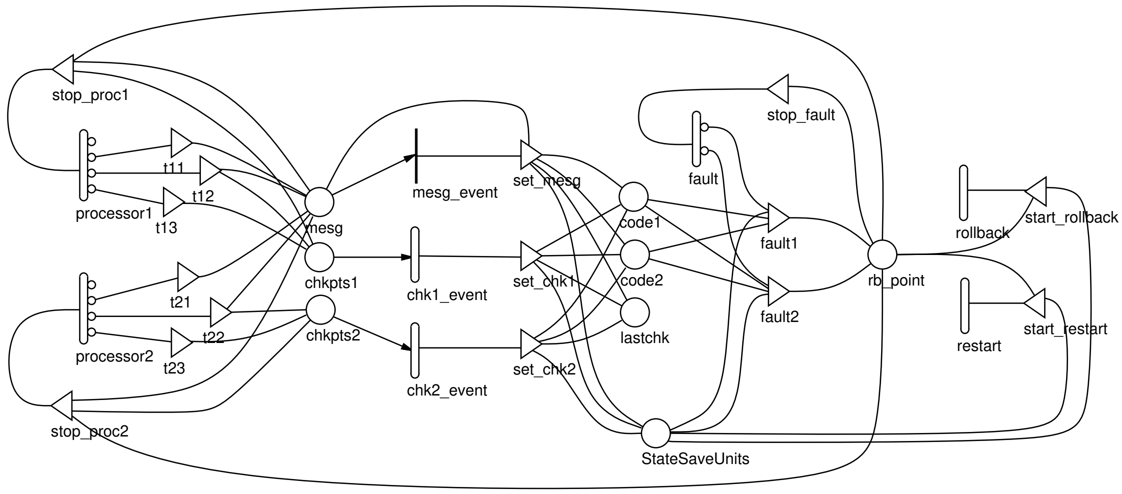


Figure C.1: Composed Model: *Multiprocessor Model*

Table C.1: Reward Variable Definitions for Project *Multiprocessor Model*

Variable	**Definition**
forward progress	
	Rate rewards Subnet = *processor* Predicate: *MARK(rb_point) == 0 && MARK(chkpts1) == 0* Function: *1*
	Impulse rewards none
	Simulator statistics Estimate mean only Confidence Level = *0.95* Relative Confidence Interval = *0.10* Initial Transient = *1000* Batch Size = *10000* Variable type = *Instant of Time* Start of Interval = *10.0* Length of Interval = *100.0*
state savings	
	Rate rewards Subnet = *processor* Predicate: *MARK(chkpts1)* Function: *1*
	Impulse rewards none
	Simulator statistics Estimate mean only Confidence Level = *0.95* Relative Confidence Interval = *0.10* Initial Transient = *1000* Batch Size = *10000* Variable type = *Instant of Time* Start of Interval = *10.0* Length of Interval = *100.0*

Table C.2: Reward Variable Definitions for Project *Multiprocessor Model*

Variable	Definition
error recovering	
	Rate rewards Subnet = *processor* Predicate: *MARK(rb_point)* Function: *1*
	Impulse rewards none
	Simulator statistics Estimate mean only Confidence Level = *0.95* Relative Confidence Interval = *0.10* Initial Transient = *1000* Batch Size = *10000* Variable type = *Instant of Time* Start of Interval = *10.0* Length of Interval = *100.0*
restart	
	Rate rewards Subnet = *processor* Predicate: *MARK(rb_point) == MARK(StateSaveUnits)+1* Function: *1*
	Impulse rewards none
	Simulator statistics Estimate mean only Confidence Level = *0.95* Relative Confidence Interval = *0.10* Initial Transient = *1000* Batch Size = *10000* Variable type = *Instant of Time* Start of Interval = *10.0* Length of Interval = *100.0*

Table C.3: Reward Variable Definitions for Project *Multiprocessor Model*

Variable	Definition
rollback	
	Rate rewards Subnet = *processor* Predicate: $MARK(rb\_point)$ && $(MARK(rb\_point) < MARK(StateSaveUnits)+1)$ Function: *1*
	Impulse rewards none
	Simulator statistics Estimate mean only Confidence Level = *0.95* Relative Confidence Interval = *0.10* Initial Transient = *1000* Batch Size = *10000* Variable type = *Instant of Time* Start of Interval = *10.0* Length of Interval = *100.0*

Table C.4: Study Editor Range Definitions for Project *Multiprocessor Model*

Study	Variable	Type	Range	Increment	Add./Mult.
vary_chkcost					
	chkpoint	double	[0.02, 0.2]	0.02	Add.
	faultrate	double	0.003	–	–
	restart	double	0.1	–	–
	sendmsg	double	0.1	–	–
	ssu	short	2	–	–
	statesaving	double	[75, 150]	25	Add.
vary_chkpointing					
	chkpoint	double	[0.05, 0.3]	0.05	Add.
	faultrate	double	[0.01, 0.03]	0.005	Add.
	restart	double	0.1	–	–
	sendmsg	double	0.2	–	–
	ssu	short	3	–	–
	statesaving	double	100	–	–
vary_msgrate					
	chkpoint	double	[0.01, 0.1]	0.02	Add.
	faultrate	double	0.003	–	–
	restart	double	0.1	–	–
	sendmsg	double	[0, 0.1]	0.02	Add.
	ssu	short	3	–	–
	statesaving	double	100	–	–
vary_restart					
	chkpoint	double	[0.02, 0.2]	0.02	Add.
	faultrate	double	0.003	–	–
	restart	double	[0.05, 0.2]	0.05	Add.
	sendmsg	double	0.1	–	–
	ssu	short	2	–	–
	statesaving	double	100	–	–
vary_ssu					
	chkpoint	double	[0.1, 0.5]	0.1	Add.
	faultrate	double	[0.003, 0.03]	10	Mult.
	restart	double	0.1	–	–
	sendmsg	double	0.1	–	–
	ssu	short	[16, 28]	4	Add.
	statesaving	double	100	–	–

Table C.5: Non-zero or Variable Initial Markings for SAN Model *processor*

Place	**Marking**
StateSaveUnits	*GLOBAL_S(ssu)*
lastchk	*1*

Table C.6: Activity Time Distributions for SAN Model *processor*

Activity	**Distribution**	**Parameter values**
chk1_event	exponential	
	rate	*GLOBAL_D(statesaving)*
chk2_event	exponential	
	rate	*GLOBAL_D(statesaving)*
fault	exponential	
	rate	*GLOBAL_D(faultrate)*
mesg_event	instantaneous	
processor1	exponential	
	rate	*10*
processor2	exponential	
	rate	*10*
restart	exponential	
	rate	*GLOBAL_D(restart)*
rollback	exponential	
	rate	*10* $*$ *GLOBAL_D(chkpoint)*

Table C.7: Activity Case Probabilities for SAN Model *processor*

Activity	**Case**	**Probability**
fault	1	*0.5*
	2	*0.5*
processor1	1	$(1.0 - GLOBAL\_D(chkpoint)) * (1.0 - GLOBAL\_D(sendmsg))$
	2	$(1.0 - GLOBAL\_D(chkpoint)) * GLOBAL\_D(sendmsg)$
	3	$GLOBAL\_D(chkpoint) * GLOBAL\_D(sendmsg)$
	4	$GLOBAL\_D(chkpoint) * (1.0 - GLOBAL\_D(sendmsg))$
processor2	1	$(1.0 - GLOBAL\_D(chkpoint)) * (1.0 - GLOBAL\_D(sendmsg))$
	2	$(1.0 - GLOBAL\_D(chkpoint)) * GLOBAL\_D(sendmsg)$
	3	$GLOBAL\_D(chkpoint) * GLOBAL\_D(sendmsg)$
	4	$GLOBAL\_D(chkpoint) * (1.0 - GLOBAL\_D(sendmsg))$

C.2 DOCUMENTATION OF THE ADAPTIVE MODEL

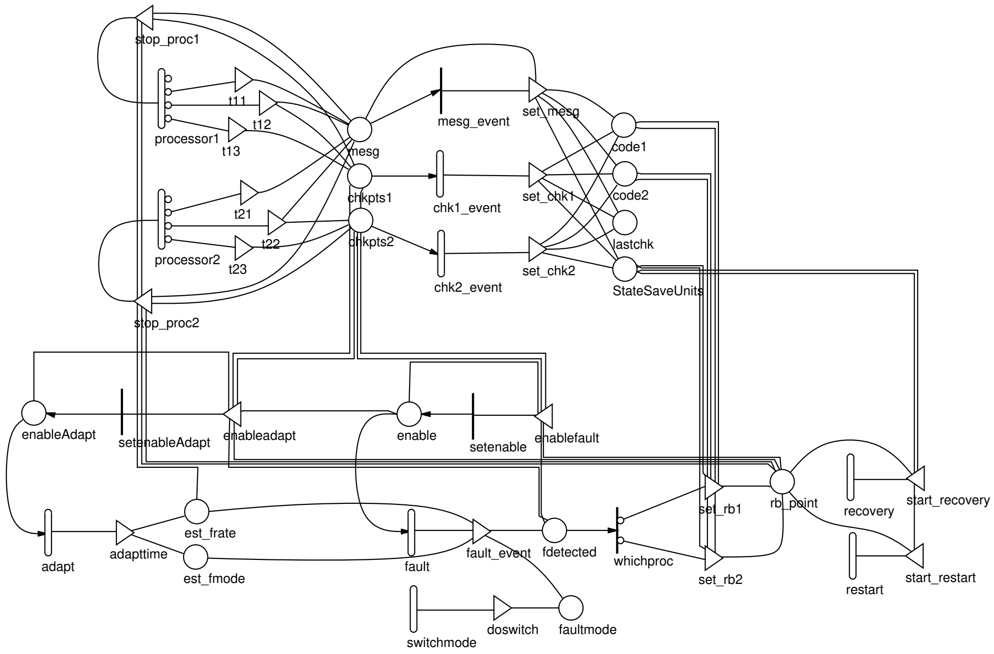


Figure C.2: Composed Model: *Adaptive Model*

Table C.8: Input Gate Definitions for SAN Model *processor*

Gate	**Definition**
start_restart	Predicate $MARK(rb\_point) == MARK(StateSaveUnits) + 1$
	Function $MARK(rb\_point) = 0;$
start_rollback	Predicate $MARK(rb\_point) > 0 \&\& MARK(rb\_point) < (MARK(StateSaveUnits)+1)$
	Function $MARK(rb\_point) - -;$
stop_fault	Predicate $MARK(rb\_point) == 0$
	Function *identity*
stop_proc1	Predicate $MARK(mesg) != 1 \&\& MARK(chkpts1) == 0 \&\& MARK(rb\_point) == 0$
	Function *identity*
stop_proc2	Predicate $MARK(mesg) != 2 \&\& MARK(chkpts2) == 0 \&\& MARK(rb\_point) == 0$
	Function *identity*

Table C.9: Output Gate Definitions for SAN Model *processor*

<table>
<tr><th>Gate</th><th>Definition</th></tr>
<tr><td>fault1</td><td><pre>
#define SSU 3
#define MAX(x,y) (((x) > (y)) ? (x) : (y))

int RB1[SSU+2], RB2[SSU+2];
int oldRB1, newRB1;
int oldRB2, newRB2;
int n;
int code1, code2;

/* decode markings */
/* −−−−−−−−−−−−−− */
code1 = MARK(code1);
RB1[0] = −1;
for (n=1; n<=SSU+1; n++) {
   RB1[n] = code1 % (int) pow(1.0*(SSU+2), 1.0*(n+1));
   code1 − = RB1[n];
   RB1[n] = RB1[n] / pow(1.0*(SSU+2), 1.0*n);
}
code2 = MARK(code2);
RB2[0] = −1;
for (n=1; n<=SSU+1; n++) {
   RB2[n] = code2 % (int) pow(1.0*(SSU+2), 1.0*(n+1));
   code2 − = RB2[n];
   RB2[n] = RB2[n] / pow(1.0*(SSU+2), 1.0*n);
}

oldRB1 = newRB1 = 1;
newRB2 = (RB2[1] > 0) ? RB2[1] : 0;
oldRB2 = −1;
while ((oldRB1 != newRB1) || (oldRB2 != newRB2)) {
   oldRB1 = newRB1;
   oldRB2 = newRB2;
   newRB2 = MAX(oldRB2, RB1[newRB1]);
   newRB1 = MAX(oldRB1, RB2[newRB2]);
}
MARK(rb_point) = newRB1;
</pre></td></tr>
</table>

Table C.10: Output Gate Definitions for SAN Model *processor*

Gate	Definition

fault2	`#define SSU 3`
	`#define MAX(x,y) (((x) > (y)) ? (x) : (y))`
	
	`int RB1[SSU+2], RB2[SSU+2];`
	`int oldRB1, newRB1;`
	`int oldRB2, newRB2;`
	`int n;`
	`int code1, code2;`
	
	`/* decode markings */`
	`/* ------------- */`
	`code1 = MARK(code1);`
	`RB1[0] = -1;`
	`for (n=1; n<=SSU+1; n++) {`
	`   RB1[n] = code1 % (int) pow(1.0*(SSU+2), 1.0*(n+1));`
	`   code1 -= RB1[n];`
	`   RB1[n] = RB1[n] / pow(1.0*(SSU+2), 1.0*n);`
	`}`
	`code2 = MARK(code2);`
	`RB2[0] = -1;`
	`for (n=1; n<=SSU+1; n++) {`
	`   RB2[n] = code2 % (int) pow(1.0*(SSU+2), 1.0*(n+1));`
	`   code2 -= RB2[n];`
	`   RB2[n] = RB2[n] / pow(1.0*(SSU+2), 1.0*n);`
	`}`
	
	`oldRB1 = newRB1 = 1;`
	`newRB2 = (RB2[1] > 0) ? RB2[1] : 0;`
	`oldRB2 = -1;`
	`while ((oldRB1 != newRB1) || (oldRB2 != newRB2)) {`
	`   oldRB1 = newRB1;`
	`   oldRB2 = newRB2;`
	`   newRB2 = MAX(oldRB2, RB1[newRB1]);`
	`   newRB1 = MAX(oldRB1, RB2[newRB2]);`
	`}`
	`MARK(rb_point) = newRB2;`

Table C.11: Output Gate Definitions for SAN Model *processor*

Gate	Definition

set_chk1	int SSU = MARK(StateSaveUnits);
	int n;
	int code;
	
	MARK(code1) *= (SSU+2);
	MARK(code1) = MARK(code1) % (int) pow(1.0*(SSU+2), 1.0*(SSU+2));
	
	code = MARK(code2);
	for (n=1; n<=SSU+1; n++) {
	if ((code % (int) pow(1.0*(SSU+2), 1.0*(n+1)) >= (int) pow(1.0*(SSU+2), 1.0*n)) && (code % (int) pow(1.0*(SSU+2), 1.0*(n+1)) < (SSU+1) * (int) pow(1.0*(SSU+2), 1.0*n)))
	code += (int) pow(1.0*(SSU+2), 1.0*n);
	}
	MARK(code2) = code;
	MARK(lastchk) = 1;
	/*
	printf("after chk1: code1=%d, code2=%d\n",MARK(code1),MARK(code2));
	*/

set_chk2	int SSU = MARK(StateSaveUnits);
	int n;
	int code;
	
	MARK(code2) *= (SSU+2);
	MARK(code2) = MARK(code2) % (int) pow(1.0*(SSU+2), 1.0*(SSU+2));
	
	code = MARK(code1);
	for (n=1; n<=SSU+1; n++) {
	if ((code % (int) pow(1.0*(SSU+2), 1.0*(n+1)) >= (int) pow(1.0*(SSU+2), 1.0*n)) && (code % (int) pow(1.0*(SSU+2), 1.0*(n+1)) < (SSU+1) * (int) pow(1.0*(SSU+2
	), 1.0*n)))
	code += (int) pow(1.0*(SSU+2), 1.0*n);
	}
	MARK(code1) = code;
	MARK(lastchk) = 2;
	/*
	printf("after chk2: code1=%d, code2=%d\n",MARK(code1),MARK(code2));
	*/

set_mesg	int SSU = MARK(StateSaveUnits);
	
	if (MARK(lastchk) == 1) {
	if (MARK(code1) % (int) pow(1.0*(SSU+2), 2.0) == 0)
	MARK(code1) += SSU+2;
	}
	else {

Table C.12: Output Gate Definitions for SAN Model *processor*

Gate	Definition
	if (MARK(code2) % (int) pow(1.0(SSU+2), 2.0) == 0)* *MARK(code2) + = SSU+2;* } /* *printf("after mesg: code1=%d, code2=%d\n",MARK(code1),MARK(code2));* */
t11	*MARK(mesg) = 1;*
t12	*MARK(mesg) = 1;* *MARK(chkpts1) = 1;*
t13	*MARK(chkpts1) = 1;*
t21	*MARK(mesg) = 1;*
t22	*MARK(mesg) = 1;* *MARK(chkpts2) = 1;*
t23	*MARK(chkpts2) = 1;*

Table C.13: Reward Variable Definitions for Project *Adaptive Model*

Variable	Definition
forward progress	
	Rate rewards Subnet = *processors* Predicate: *MARK(chkpts1) == 0 && MARK(rb_point) == 0* Function: *1*
	Impulse rewards none
	Simulator statistics Estimate mean only Confidence Level = *0.95* Relative Confidence Interval = *0.10* Initial Transient = *400000* Batch Size = *400000* Variable type = *Instant of Time* Start of Interval = *10.0* Length of Interval = *100.0*
state savings	
	Rate rewards Subnet = *processors* Predicate: *MARK(chkpts1)* Function: *1*
	Impulse rewards none
	Simulator statistics Estimate mean only Confidence Level = *0.95* Relative Confidence Interval = *0.10* Initial Transient = *400000* Batch Size = *400000* Variable type = *Instant of Time* Start of Interval = *10.0* Length of Interval = *100.0*

Table C.14: Reward Variable Definitions for Project *Adaptive Model*

Variable	Definition
restart	
	Rate rewards Subnet = *processors* Predicate: *MARK(rb_point) == MARK(StateSaveUnits) + 1* Function: *1*
	Impulse rewards none
	Simulator statistics Estimate mean only Confidence Level = *0.95* Relative Confidence Interval = *0.10* Initial Transient = *400000* Batch Size = *400000* Variable type = *Instant of Time* Start of Interval = *10.0* Length of Interval = *100.0*
prob in fault mode 1	
	Rate rewards Subnet = *processors* Predicate: *MARK(faultmode) == 0* Function: *1*
	Impulse rewards none
	Simulator statistics Estimate mean only Confidence Level = *0.95* Relative Confidence Interval = *0.10* Initial Transient = *400000* Batch Size = *400000* Variable type = *Instant of Time* Start of Interval = *10.0* Length of Interval = *100.0*

Table C.15: Reward Variable Definitions for Project *Adaptive Model*

Variable	**Definition**
prob incorrect fault−mode estimation	
	Rate rewards Subnet = *processors* Predicate: *MARK(faultmode) != MARK(est_fmode)* Function: *1*
	Impulse rewards none
	Simulator statistics Estimate mean only Confidence Level = *0.95* Relative Confidence Interval = *0.10* Initial Transient = *400000* Batch Size = *400000* Variable type = *Instant of Time* Start of Interval = *10.0* Length of Interval = *100.0*
prob rollback	
	Rate rewards Subnet = *processors* Predicate: *MARK(rb_point) && (MARK(rb_point) < MARK(StateSaveUnits)+1)* Function: *1*
	Impulse rewards none
	Simulator statistics Estimate mean only Confidence Level = *0.95* Relative Confidence Interval = *0.10* Initial Transient = *400000* Batch Size = *400000* Variable type = *Instant of Time* Start of Interval = *10.0* Length of Interval = *100.0*

Table C.16: Study Editor Set Definitions for Project *Adaptive Model*

Study	**Experiment**	**Variable**	**Type**	**Value**
single				
	exp1			
		myLAMBDA1	double	0.03
		myLAMBDA2	double	0.003
		myMU1	double	0.0012
		myMU2	double	0.00012
		restart	double	0.1
		sendmsg	double	0.15
		ssu	short	4
		statesaving	double	100

Table C.17: Non-zero or Variable Initial Markings for SAN Model *processors*

Place	**Marking**
StateSaveUnits	*GLOBAL_S(ssu)*
enable	*1*
enableAdapt	*1*
lastchk	*1*

Table C.18: Activity Time Distributions for SAN Model *processors*

Activity	**Distribution**	**Parameter values**
adapt	exponential	
	rate	*0.1*
chk1_event	exponential	
	rate	*GLOBAL_D(statesaving)*
chk2_event	exponential	
	rate	*GLOBAL_D(statesaving)*
fault	exponential	
	rate	*if (MARK(faultmode) ==0)* *return (GLOBAL_D(myLAMBDA1));* *else* *return (GLOBAL_D(myLAMBDA2));*
mesg_event	instantaneous	
processor1	exponential	
	rate	*10*
processor2	exponential	
	rate	*10*
recovery	exponential	
	rate	*extern float aveCheckpoint;* *return (10 * aveCheckpoint);*
restart	exponential	
	rate	*GLOBAL_D(restart)*
setenable	instantaneous	
setenableAdapt	instantaneous	
switchmode	exponential	
	rate	*if (MARK(faultmode) ==0)* *return (GLOBAL_D(myMU1));* *else* *return (GLOBAL_D(myMU2));*
whichproc	instantaneous	

Table C.19: Activity Case Probabilities for SAN Model *processors*

Activity	**Case**	**Probability**
processor1	1	*extern float chkptsTable[];* *return ((1.0 − chkptsTable[MARK(est_frate)]) ∗ (1.0 − GLOBAL_D(sendmsg)));*
	2	*extern float chkptsTable[];* *return ((1.0 − chkptsTable[MARK(est_frate)]) ∗ GLOBAL_D(sendmsg));*
	3	*extern float chkptsTable[];* *return (chkptsTable[MARK(est_frate)] ∗ GLOBAL_D(sendmsg));*
	4	*extern float chkptsTable[];* *return ((chkptsTable[MARK(est_frate)]) ∗ (1.0 − GLOBAL_D(sendmsg)));*
processor2	1	*extern float chkptsTable[];* *return ((1.0 − chkptsTable[MARK(est_frate)]) ∗ (1.0 − GLOBAL_D(sendmsg)));*
	2	*extern float chkptsTable[];* *return ((1.0 − chkptsTable[MARK(est_frate)]) ∗ GLOBAL_D(sendmsg));*
	3	*extern float chkptsTable[];* *return (chkptsTable[MARK(est_frate)] ∗ GLOBAL_D(sendmsg));*
	4	*extern float chkptsTable[];* *return (chkptsTable[MARK(est_frate)] ∗ (1.0 − GLOBAL_D(sendmsg)));*
whichproc	1	*0.5*
	2	*0.5*

Table C.20: Input Gate Definitions for SAN Model *processors*

Gate	**Definition**
enableadapt	Predicate MARK(rb_point) ==0 && MARK(enableAdapt) ==0 && MARK(fdetected) ==0 && MARK(enable) >0 && MARK(chkpts1) ==0 && MARK(chkpts2) ==0
	Function extern timeValue enableAdaptTime; extern timeValue currentTime; enableAdaptTime = currentTime;
enablefault	Predicate MARK(rb_point) ==0 && MARK(enable) ==0 && MARK(fdetected) ==0 && MARK(chkpts1) ==0 && MARK(chkpts2) ==0
	Function extern timeValue enableTime; extern timeValue currentTime; enableTime = currentTime;
start_recovery	Predicate MARK(rb_point) > 0 && MARK(rb_point) < (MARK(StateSaveUnits)+1)
	Function MARK(rb_point) − −;
start_restart	Predicate MARK(rb_point) == MARK(StateSaveUnits) + 1
	Function MARK(rb_point) = 0;
stop_proc1	Predicate MARK(mesg) != 1 && MARK(chkpts1) == 0 && MARK(rb_point) == 0

	Function
	extern float chkptsTable[];
	extern float aveCheckpoint;
	static float avecounter = 0.0;
	
	aveCheckpoint = (avecounter/(avecounter+1.0))∗ aveCheckpoint + (1.0/(avecounter+1.0)) ∗ chkptsTable[MARK(est_frate)];
	avecounter++;

stop_proc2	Predicate MARK(mesg) != 2 && MARK(chkpts2) == 0 && MARK(rb_point) == 0

Table C.21: Input Gate Definitions for SAN Model *processors*

Gate	Definition
	<u>Function</u> *identity*

Table C.22: Output Gate Definitions for SAN Model *processors*

Gate	Definition

adapttime	`#define N 1500`
	`#define FAULT_MODE 2`
	`#define MAXTABLE 10 /* chkptsTable size */`
	
	`/* GLOBAL VARIABLES */`
	`extern double LAMBDA1; /* estimated value of LAMBDA1 */`
	`extern double LAMBDA2; /* estimated value of LAMBDA2 */`
	`extern double PROB; /* estimated value of MIXTURE */`
	`extern float MU1, MU2; /* estimated value of MU1 & MU2 */`
	`extern float AveDistMu1;`
	`extern float AveDistMu2;`
	`extern int Counter1, Counter2;`
	`extern float Total1, Total2;`
	`extern int CurrentMode;`
	`extern int AdaptNow;`
	`extern float chkptsTable[MAXTABLE];`
	`extern float frate;`
	`extern timeValue currentData;`
	`extern timeValue currentTime;`
	`extern timeValue enableAdaptTime;`
	`extern timeValue elapseTime;`
	`extern timeValue Data[N];`
	
	`/* LOCAL VARIABLES */`
	`static int FirstTime = 1;`
	`float clusterpoint;`
	`float estrate;`
	`int i;`
	
	`/* FIRST TIME ADAPTIVE SCHEME IS EXEC.; FIND CURRENT FAULT-MODE */`
	`if (AdaptNow && FirstTime) {`
	`FirstTime = 0;`
	`Counter2 = 0;`
	`}`
	
	`if (AdaptNow) {`
	`clusterpoint = (LAMBDA1 + LAMBDA2) / 2.0;`
	`elapseTime += currentTime - enableAdaptTime;`

Table C.23: Output Gate Definitions for SAN Model *processors*

Gate	Definition

	/* CHECK IF IT'S TIME TO SWICTH FAULT−MODE */
	if (CurrentMode==0 && elapseTime>clusterpoint) {
	CurrentMode = (CurrentMode + 1) % FAULT_MODE;
	}
	else if (CurrentMode==1 && Counter2>0 && elapseTime<clusterpoint)
	{
	CurrentMode = (CurrentMode + 1) % FAULT_MODE;
	Counter2 = 0;
	}
	}
	
	/* OBTAIN OPTIMUM CHKPTS INTERVAL FROM chkptsTable */
	if (CurrentMode == 0)
	estrate = 1.0/LAMBDA1;
	else
	estrate = 1.0/LAMBDA2;
	for (i=0; i<MAXTABLE; i++) {
	if (estrate <= i * frate) break;
	}
	if (i==MAXTABLE) i−−;
	
	MARK(est_frate) = i;
	MARK(est_fmode) = CurrentMode;

doswitch	#define FAULT_MODE 2 MARK(faultmode) = (MARK(faultmode) + 1) % FAULT_MODE;
fault_event	#define DBL_MAX 1e+20 #define N 1500 /* number of data collected */ #define MODE 2 /* fault mode in system */ #define STEP 100 /* step to estimate MUs */ #define SMALLPROB 0.01 /* GLOBAL VARIABLES */ extern double LAMBDA1; extern double LAMBDA2; extern double PROB; extern float MU1; extern float MU2; extern float AveDistMu1; extern float AveDistMu2; extern timeValue Data[N]; /* array to store fault intervals */

Table C.24: Output Gate Definitions for SAN Model *processors*

<table>
<tr><th>Gate</th><th>Definition</th></tr>
<tr><td></td><td><pre>
extern timeValue currentData; /* the last fault intervals   */
extern timeValue currentTime; /* current simulation time   */
extern timeValue enableTime; /* time at which 'fault' enabled */
extern timeValue elapseTime;
extern timeValue totalcurrentmode; /* total time in current mode */
extern int CurrentMode;
extern int AdaptNow;
extern int Counter1, Counter2;
extern float Total1, Total2;
extern double totalMoment1;
extern double totalMoment2;
extern double totalMoment3;
extern double momentCounter;

/* LOCAL VARIABLES */
static int currentindex   = 0;
static float MuIteration   = 0.0;
static float estimateCounter = 0.0;
double curLAMBDA1, curLAMBDA2, curPROB;
double m1,m2,m3;
double temp, newmu;
double clusterpoint, p;
int   clusteredData[N];
int   i, j, max;
int   start=0, end=0;
int   startMU=0, distanceMU=0;

MARK(fdetected) = 1;
currentData = currentTime − enableTime;
totalcurrentmode + = currentData;
if (totalMoment3 < DBL_MAX−(currentData*currentData*currentData))
{
   totalMoment1 + = currentData;
   totalMoment2 + = currentData * currentData;
   totalMoment3 + = currentData * currentData * currentData;
   momentCounter++;
}

/* STORE EVERY N FAULT−INTERVALS IN ARRAY Data */
if (currentindex == N) {
</pre></td></tr>
</table>

Table C.25: Output Gate Definitions for SAN Model *processors*

Gate	Definition

	`` currentindex = 0; }
	Data[currentindex] = currentData;
	currentindex++;
	
	if (currentindex == N) {
	  AdaptNow = 1;
	
	  /* ESTIMATE FAULT-RATES FROM THE MOMENTS */
	  m1 = totalMoment1/momentCounter;
	  m2 = totalMoment2/momentCounter;
	  m3 = totalMoment3/momentCounter;
	  temp = 4*m3*m3 + 72*pow(m2,3.0) - 72*m1*m2*m3 -
	         108*m1*m1*m2*m2 + 96*m3*pow(m1,3.0);
	  LAMBDA1 = ((-2*m3 + 6*m1*m2) + sqrt(temp)) / (24*m1*m1 - 12*m2);
	  LAMBDA2 = ((-2*m3 + 6*m1*m2) - sqrt(temp)) / (24*m1*m1 - 12*m2);
	  PROB = (m1 - curLAMBDA2) / (curLAMBDA1 - curLAMBDA2);
	}
	
	/* COUNTER FOR ESTIMATION OF TIME TO SWITCH FAULT-MODE */
	if (CurrentMode==0) {
	  elapseTime = 0.0;
	  Total1 += currentData;
	}
	else {
	  Counter2++;
	  clusterpoint = (LAMBDA1 + LAMBDA2) / 2.0;
	  if (elapseTime > clusterpoint) {
	    Counter2 = 0;
	    elapseTime = 0.0;
	  }
	  Total2 += currentData;
	} ```

set_chk1	``` int SSU = MARK(StateSaveUnits); int n; int code; MARK(code1) *= (SSU+2); MARK(code1) = MARK(code1) % (int) pow(1.0*(SSU+2), 1.0*(SSU+2)); ```

C.3 DOCUMENTATION OF THE FIXED MODEL

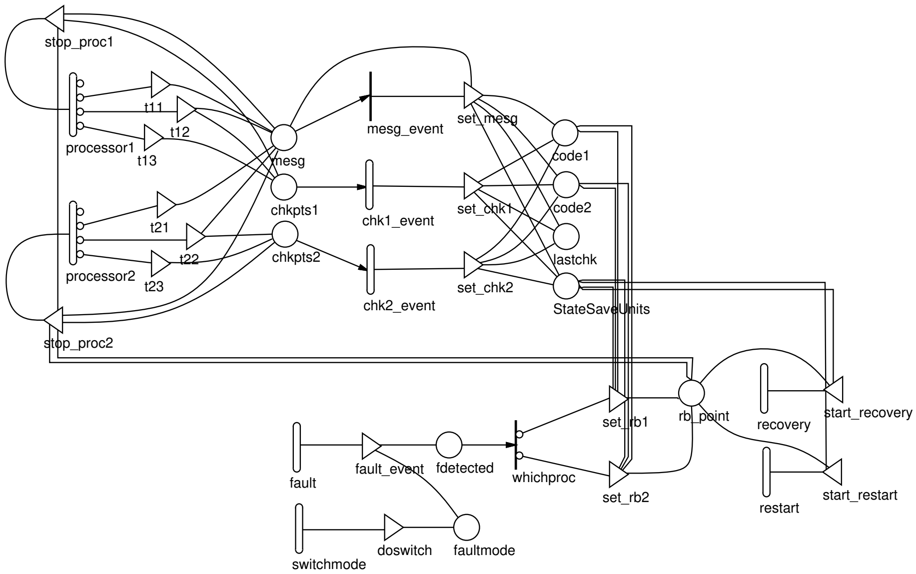


Figure C.3: Composed Model: *Multiprocessor Model*

Table C.26: Output Gate Definitions for SAN Model *processors*

Gate	Definition

	code = MARK(code2);
	for (n=1; n<=SSU+1; n++) {
	if ((code % (int) pow(1.0*(SSU+2), 1.0*(n+1)) >= (int) pow(1.0*(SSU+2), 1.0*n)) && (code % (int) pow(1.0*(SSU+2), 1.0*(n+1)) < (SSU+1) * (int) pow(1.0*(SSU+2), 1.0*n)))
	code += (int) pow(1.0*(SSU+2), 1.0*n);
	}
	MARK(code2) = code;
	MARK(lastchk) = 1;

set_chk2	int SSU = MARK(StateSaveUnits);
	int n;
	int code;
	
	MARK(code2) *= (SSU+2);
	MARK(code2) = MARK(code2) % (int) pow(1.0*(SSU+2), 1.0*(SSU+2));
	
	code = MARK(code1);
	for (n=1; n<=SSU+1; n++) {
	if ((code % (int) pow(1.0*(SSU+2), 1.0*(n+1)) >= (int) pow(1.0*(SSU+2), 1.0*n)) && (code % (int) pow(1.0*(SSU+2), 1.0*(n+1)) < (SSU+1) * (int) pow(1.0*(SSU+2
	), 1.0*n)))
	code += (int) pow(1.0*(SSU+2), 1.0*n);
	}
	MARK(code1) = code;
	MARK(lastchk) = 2;

set_mesg	int SSU = MARK(StateSaveUnits);
	
	if (MARK(lastchk) == 1) {
	if (MARK(code1) % (int) pow(1.0*(SSU+2), 2.0) == 0)
	MARK(code1) += SSU+2;
	}
	else {
	if (MARK(code2) % (int) pow(1.0*(SSU+2), 2.0) == 0)
	MARK(code2) += SSU+2;
	}

set_rb1	#define MAXSSU 50 #define MAX(x,y) (((x) > (y)) ? (x) : (y)) int RB1[MAXSSU+2], RB2[MAXSSU+2]; int oldRB1, newRB1; int oldRB2, newRB2; int n; int code1, code2; int SSU = MARK(StateSaveUnits);

Table C.27: Output Gate Definitions for SAN Model *processors*

Gate	Definition

	`/* DECODE MARKINGS */`
	`code1 = MARK(code1);`
	`RB1[0] = -1;`
	`for (n=1; n<=SSU+1; n++) {`
	`   RB1[n] = code1 % (int) pow(1.0*(SSU+2), 1.0*(n+1));`
	`   code1 -= RB1[n];`
	`   RB1[n] = RB1[n] / pow(1.0*(SSU+2), 1.0*n);`
	`}`
	`code2 = MARK(code2);`
	`RB2[0] = -1;`
	`for (n=1; n<=SSU+1; n++) {`
	`   RB2[n] = code2 % (int) pow(1.0*(SSU+2), 1.0*(n+1));`
	`   code2 -= RB2[n];`
	`   RB2[n] = RB2[n] / pow(1.0*(SSU+2), 1.0*n);`
	`}`
	
	`/* SET ROLLBACK POINT */`
	`oldRB1 = newRB1 = 1;`
	`newRB2 = (RB2[1] > 0) ? RB2[1] : 0;`
	`oldRB2 = -1;`
	`while ((oldRB1 != newRB1) || (oldRB2 != newRB2)) {`
	`   oldRB1 = newRB1;`
	`   oldRB2 = newRB2;`
	`   newRB2 = MAX(oldRB2, RB1[newRB1]);`
	`   newRB1 = MAX(oldRB1, RB2[newRB2]);`
	`}`
	`MARK(rb_point) = newRB1;`

set_rb2	`#define MAXSSU 50`
	`#define MAX(x,y) (((x) > (y)) ? (x) : (y))`
	
	`int RB1[MAXSSU+2], RB2[MAXSSU+2];`
	`int oldRB1, newRB1;`
	`int oldRB2, newRB2;`
	`int n;`
	`int code1, code2;`
	`int SSU = MARK(StateSaveUnits);`
	
	`/* DECODE MARKINGS */`
	`code1 = MARK(code1);`
	`RB1[0] = -1;`
	`for (n=1; n<=SSU+1; n++) {`
	`   RB1[n] = code1 % (int) pow(1.0*(SSU+2), 1.0*(n+1));`

Table C.28: Output Gate Definitions for SAN Model *processors*

<table>
<tr><th>Gate</th><th>Definition</th></tr>
<tr><td></td><td>code1 − = RB1[n];
RB1[n] = RB1[n] / pow(1.0*(SSU+2), 1.0*n);
}
code2 = MARK(code2);
RB2[0] = −1;
for (n=1; n<=SSU+1; n++) {
RB2[n] = code2 % (int) pow(1.0*(SSU+2), 1.0*(n+1));
code2 − = RB2[n];
RB2[n] = RB2[n] / pow(1.0*(SSU+2), 1.0*n);
}

/* SET ROLLBACK POINT */
oldRB1 = newRB1 = 1;
newRB2 = (RB2[1] > 0) ? RB2[1] : 0;
oldRB2 = −1;
while ((oldRB1 != newRB1) || (oldRB2 != newRB2)) {
oldRB1 = newRB1;
oldRB2 = newRB2;
newRB2 = MAX(oldRB2, RB1[newRB1]);
newRB1 = MAX(oldRB1, RB2[newRB2]);
}
MARK(rb_point) = newRB2;</td></tr>
<tr><td>t11</td><td>MARK(mesg) = 1;</td></tr>
<tr><td>t12</td><td>MARK(mesg) = 1;
MARK(chkpts1) = 1;</td></tr>
<tr><td>t13</td><td>MARK(chkpts1) = 1;</td></tr>
<tr><td>t21</td><td>MARK(mesg) = 1;</td></tr>
<tr><td>t22</td><td>MARK(mesg) = 1;
MARK(chkpts2) = 1;</td></tr>
<tr><td>t23</td><td>MARK(chkpts2) = 1;</td></tr>
</table>

Table C.29: Reward Variable Definitions for Project *Fixed Model*

Variable	Definition
forward progress	
	Rate rewards Subnet = *processors* Predicate: *MARK(chkpts1) == 0 && MARK(rb_point) == 0* Function: *1*
	Impulse rewards none
	Simulator statistics Estimate mean only Confidence Level = *0.95* Relative Confidence Interval = *0.10* Initial Transient = *400000* Batch Size = *400000* Variable type = *Instant of Time* Start of Interval = *10.0* Length of Interval = *100.0*
state savings	
	Rate rewards Subnet = *processors* Predicate: *MARK(chkpts1)* Function: *1*
	Impulse rewards none
	Simulator statistics Estimate mean only Confidence Level = *0.95* Relative Confidence Interval = *0.10* Initial Transient = *400000* Batch Size = *400000* Variable type = *Instant of Time* Start of Interval = *10.0* Length of Interval = *100.0*

Table C.30: Reward Variable Definitions for Project *Fixed Model*

Variable	Definition
restart	
	Rate rewards Subnet = *processors* Predicate: *MARK(rb_point) == MARK(StateSaveUnits) + 1* Function: *1*
	Impulse rewards none
	Simulator statistics Estimate mean only Confidence Level = *0.95* Relative Confidence Interval = *0.10* Initial Transient = *400000* Batch Size = *400000* Variable type = *Instant of Time* Start of Interval = *10.0* Length of Interval = *100.0*
prob in fault mode 1	
	Rate rewards Subnet = *processors* Predicate: *MARK(faultmode) == 0* Function: *1*
	Impulse rewards none
	Simulator statistics Estimate mean only Confidence Level = *0.95* Relative Confidence Interval = *0.10* Initial Transient = *400000* Batch Size = *400000* Variable type = *Instant of Time* Start of Interval = *10.0* Length of Interval = *100.0*

Table C.31: Reward Variable Definitions for Project *Fixed Model*

Variable	Definition
prob incorrect fault−mode estimation	
	Rate rewards Subnet = *processors* Predicate: *MARK(faultmode) != MARK(est_fmode)* Function: *1*
	Impulse rewards none
	Simulator statistics Estimate mean only Confidence Level = *0.95* Relative Confidence Interval = *0.10* Initial Transient = *400000* Batch Size = *400000* Variable type = *Instant of Time* Start of Interval = *10.0* Length of Interval = *100.0*
prob rollback	
	Rate rewards Subnet = *processors* Predicate: *MARK(rb_point) && (MARK(rb_point) < MARK(StateSaveUnits)+1)* Function: *1*
	Impulse rewards none
	Simulator statistics Estimate mean only Confidence Level = *0.95* Relative Confidence Interval = *0.10* Initial Transient = *400000* Batch Size = *400000* Variable type = *Instant of Time* Start of Interval = *10.0* Length of Interval = *100.0*

Table C.32: Study Editor Set Definitions for Project *Fixed Model*

Study	**Experiment**	**Variable**	**Type**	**Value**
single				
	exp1			
		myLAMBDA1	double	0.03
		myLAMBDA2	double	0.003
		myMU1	double	0.0012
		myMU2	double	0.00012
		restart	double	0.1
		sendmsg	double	0.15
		ssu	short	4
		statesaving	double	100

Table C.33: Non-zero or Variable Initial Markings for SAN Model *processors*

Place	**Marking**
StateSaveUnits	*GLOBAL_S(ssu)*
lastchk	*1*

Table C.34: Activity Time Distributions for SAN Model *processors*

Activity	**Distribution**	**Parameter values**
chk1_event	exponential	
	rate	*GLOBAL_D(statesaving)*
chk2_event	exponential	
	rate	*GLOBAL_D(statesaving)*
fault	exponential	
	rate	*if (MARK(faultmode) ==0)* *return (GLOBAL_D(myLAMBDA1));* *else* *return (GLOBAL_D(myLAMBDA2));*
mesg_event	instantaneous	
processor1	exponential	
	rate	*10*
processor2	exponential	
	rate	*10*
recovery	exponential	
	rate	*extern float aveCheckpoint;* *return (10 * aveCheckpoint);*
restart	exponential	
	rate	*GLOBAL_D(restart)*
switchmode	exponential	
	rate	*if (MARK(faultmode) ==0)* *return (GLOBAL_D(myMU1));* *else* *return (GLOBAL_D(myMU2));*
whichproc	instantaneous	

Table C.35: Activity Case Probabilities for SAN Model *processors*

Activity	Case	Probability
processor1	1	$(1.0 - GLOBAL\_D(chkpoint)) * (1.0 - GLOBAL\_D(sendmsg))$
	2	$(1.0 - GLOBAL\_D(chkpoint)) * GLOBAL\_D(sendmsg)$
	3	$GLOBAL\_D(chkpoint) * GLOBAL\_D(sendmsg)$
	4	$GLOBAL\_D(chkpoint) * (1.0 - GLOBAL\_D(sendmsg))$
processor2	1	$(1.0 - GLOBAL\_D(chkpoint)) * (1.0 - GLOBAL\_D(sendmsg))$
	2	$(1.0 - GLOBAL\_D(chkpoint)) * GLOBAL\_D(sendmsg)$
	3	$GLOBAL\_D(chkpoint) * GLOBAL\_D(sendmsg)$
	4	$GLOBAL\_D(chkpoint) * (1.0 - GLOBAL\_D(sendmsg))$
whichproc	1	*0.5*
	2	*0.5*

Table C.36: Input Gate Definitions for SAN Model *processors*

Gate	Definition
start_recovery	Predicate $MARK(rb\_point) > 0$ && $MARK(rb\_point) < (MARK(StateSaveUnits)+1)$
	Function $MARK(rb\_point) - -;$
start_restart	Predicate $MARK(rb\_point) == MARK(StateSaveUnits) + 1$
	Function $MARK(rb\_point) = 0;$
stop_proc1	Predicate $MARK(mesg)$!= 1 && $MARK(chkpts1) == 0$ && $MARK(rb\_point) == 0$
	Function *identity*
stop_proc2	Predicate $MARK(mesg)$!= 2 && $MARK(chkpts2) == 0$ && $MARK(rb\_point) == 0$
	Function *identity*

Table C.37: Output Gate Definitions for SAN Model *processors*

Gate	Definition
doswitch	#define FAULT_MODE 2 MARK(faultmode) = (MARK(faultmode) + 1) % FAULT_MODE;

fault_event	#define DBL_MAX 1e+20
	#define N 1500 /* number of data collected */
	#define MODE 2 /* fault mode in system */
	#define STEP 100 /* step to estimate MUs */
	#define SMALLPROB 0.01
	
	/* GLOBAL VARIABLES */
	extern double LAMBDA1;
	extern double LAMBDA2;
	extern double PROB;
	extern float MU1;
	extern float MU2;
	extern float AveDistMu1;
	extern float AveDistMu2;
	extern timeValue Data[N]; /* array to store fault intervals */
	extern timeValue currentData; /* the last fault intervals */
	extern timeValue currentTime; /* current simulation time */
	extern timeValue enableTime; /* time at which 'fault' enabled */
	extern timeValue elapseTime;
	extern timeValue totalcurrentmode; /* total time in current mode */
	extern int CurrentMode;
	extern int AdaptNow;
	extern int Counter1, Counter2;
	extern float Total1, Total2;
	extern double totalMoment1;
	extern double totalMoment2;
	extern double totalMoment3;
	extern double momentCounter;
	
	/* LOCAL VARIABLES */
	static int currentindex = 0;
	static float MuIteration = 0.0;
	static float estimateCounter = 0.0;
	double curLAMBDA1, curLAMBDA2, curPROB;
	double m1,m2,m3;
	double temp, newmu;
	double clusterpoint, p;

Table C.38: Output Gate Definitions for SAN Model *processors*

Gate	Definition

	int clusteredData[N];
	int i, j, max;
	int start=0, end=0;
	int startMU=0, distanceMU=0;
	
	MARK(fdetected) = 1;
	currentData = currentTime − enableTime;
	totalcurrentmode + = currentData;
	if (totalMoment3 < DBL_MAX−(currentData∗currentData∗currentData))
	{
	totalMoment1 + = currentData;
	totalMoment2 + = currentData ∗ currentData;
	totalMoment3 + = currentData ∗ currentData ∗ currentData;
	momentCounter++;
	}
	
	/∗ STORE EVERY N FAULT−INTERVALS IN ARRAY Data ∗/
	if (currentindex == N) {
	currentindex = 0;
	}
	Data[currentindex] = currentData;
	currentindex++;
	
	if (currentindex == N) {
	AdaptNow = 1;
	
	/∗ ESTIMATE FAULT−RATES FROM THE MOMENTS ∗/
	m1 = totalMoment1/momentCounter;
	m2 = totalMoment2/momentCounter;
	m3 = totalMoment3/momentCounter;
	temp = 4∗m3∗m3 + 72∗pow(m2,3.0) − 72∗m1∗m2∗m3 − 108∗m1∗m1∗m2∗m2 + 96∗m3∗pow(m1,3.0);
	LAMBDA1 = ((−2∗m3 + 6∗m1∗m2) + sqrt(temp)) / (24∗m1∗m1 − 12∗m2);
	LAMBDA2 = ((−2∗m3 + 6∗m1∗m2) − sqrt(temp)) / (24∗m1∗m1 − 12∗m2);
	PROB = (m1 − curLAMBDA2) / (curLAMBDA1 − curLAMBDA2);
	}
	
	/∗ COUNTER FOR ESTIMATION OF TIME TO SWITCH FAULT−MODE ∗/
	if (CurrentMode==0) {

Table C.39: Output Gate Definitions for SAN Model *processors*

Gate	Definition

	elapseTime = 0.0;
	Total1 += currentData;
	}
	else {
	Counter2++;
	clusterpoint = (LAMBDA1 + LAMBDA2) / 2.0;
	if (elapseTime > clusterpoint) {
	Counter2 = 0;
	elapseTime = 0.0;
	}
	Total2 += currentData;
	}

set_chk1	int SSU = MARK(StateSaveUnits);
	int n;
	int code;
	
	MARK(code1) *= (SSU+2);
	MARK(code1) = MARK(code1) % (int) pow(1.0*(SSU+2), 1.0*(SSU+2));
	
	code = MARK(code2);
	for (n=1; n<=SSU+1; n++) {
	if ((code % (int) pow(1.0*(SSU+2), 1.0*(n+1)) >= (int) pow(1.0*(SSU+2), 1.0*n)) && (code % (int) pow(1.0*(SSU+2), 1.0*(n+1)) < (SSU+1) * (int) pow(1.0*(SSU+2), 1.0*n)))
	code += (int) pow(1.0*(SSU+2), 1.0*n);
	}
	MARK(code2) = code;
	MARK(lastchk) = 1;

set_chk2	int SSU = MARK(StateSaveUnits);
	int n;
	int code;
	
	MARK(code2) *= (SSU+2);
	MARK(code2) = MARK(code2) % (int) pow(1.0*(SSU+2), 1.0*(SSU+2));
	
	code = MARK(code1);
	for (n=1; n<=SSU+1; n++) {
	if ((code % (int) pow(1.0*(SSU+2), 1.0*(n+1)) >= (int) pow(1.0*(SSU+2), 1.0*n)) && (code % (int) pow(1.0*(SSU+2), 1.0*(n+1)) < (SSU+1) * (int) pow(1.0*(SSU+2
	), 1.0*n)))
	code += (int) pow(1.0*(SSU+2), 1.0*n);
	}
	MARK(code1) = code;
	MARK(lastchk) = 2;

Table C.40: Output Gate Definitions for SAN Model *processors*

<table>
<tr><th>Gate</th><th>Definition</th></tr>
<tr><td>set_mesg</td><td>int SSU = MARK(StateSaveUnits);

if (MARK(lastchk) == 1) {
   if (MARK(code1) % (int) pow(1.0*(SSU+2), 2.0) == 0)
      MARK(code1) += SSU+2;
}
else {
   if (MARK(code2) % (int) pow(1.0*(SSU+2), 2.0) == 0)
      MARK(code2) += SSU+2;
}</td></tr>
<tr><td>set_rb1</td><td>#define MAXSSU 50
#define MAX(x,y) (((x) > (y)) ? (x) : (y))

int RB1[MAXSSU+2], RB2[MAXSSU+2];
int oldRB1, newRB1;
int oldRB2, newRB2;
int n;
int code1, code2;
int SSU = MARK(StateSaveUnits);

/* DECODE MARKINGS */
code1 = MARK(code1);
RB1[0] = −1;
for (n=1; n<=SSU+1; n++) {
   RB1[n] = code1 % (int) pow(1.0*(SSU+2), 1.0*(n+1));
   code1 −= RB1[n];
   RB1[n] = RB1[n] / pow(1.0*(SSU+2), 1.0*n);
}
code2 = MARK(code2);
RB2[0] = −1;
for (n=1; n<=SSU+1; n++) {
   RB2[n] = code2 % (int) pow(1.0*(SSU+2), 1.0*(n+1));
   code2 −= RB2[n];
   RB2[n] = RB2[n] / pow(1.0*(SSU+2), 1.0*n);
}

/* SET ROLLBACK POINT */
oldRB1 = newRB1 = 1;
newRB2 = (RB2[1] > 0) ? RB2[1] : 0;
oldRB2 = −1;
while ((oldRB1 != newRB1) || (oldRB2 != newRB2)) {
   oldRB1 = newRB1;</td></tr>
</table>

Table C.41: Output Gate Definitions for SAN Model *processors*

<table>
<tr><th>Gate</th><th>Definition</th></tr>
<tr><td></td><td><pre>
   oldRB2 = newRB2;
   newRB2 = MAX(oldRB2, RB1[newRB1]);
   newRB1 = MAX(oldRB1, RB2[newRB2]);
}
MARK(rb_point) = newRB1;
</pre></td></tr>
<tr><td>set_rb2</td><td><pre>
#define MAXSSU 50
#define MAX(x,y) (((x) > (y)) ? (x) : (y))

int RB1[MAXSSU+2], RB2[MAXSSU+2];
int oldRB1, newRB1;
int oldRB2, newRB2;
int n;
int code1, code2;
int SSU = MARK(StateSaveUnits);

/* DECODE MARKINGS */
code1 = MARK(code1);
RB1[0] = -1;
for (n=1; n<=SSU+1; n++) {
   RB1[n] = code1 % (int) pow(1.0*(SSU+2), 1.0*(n+1));
   code1 -= RB1[n];
   RB1[n] = RB1[n] / pow(1.0*(SSU+2), 1.0*n);
}
code2 = MARK(code2);
RB2[0] = -1;
for (n=1; n<=SSU+1; n++) {
   RB2[n] = code2 % (int) pow(1.0*(SSU+2), 1.0*(n+1));
   code2 -= RB2[n];
   RB2[n] = RB2[n] / pow(1.0*(SSU+2), 1.0*n);
}

/* SET ROLLBACK POINT */
oldRB1 = newRB1 = 1;
newRB2 = (RB2[1] > 0) ? RB2[1] : 0;
oldRB2 = -1;
while ((oldRB1 != newRB1) || (oldRB2 != newRB2)) {
   oldRB1 = newRB1;
   oldRB2 = newRB2;
   newRB2 = MAX(oldRB2, RB1[newRB1]);
   newRB1 = MAX(oldRB1, RB2[newRB2]);
}
MARK(rb_point) = newRB2;
</pre></td></tr>
</table>

Table C.42: Output Gate Definitions for SAN Model *processors*

Gate	**Definition**
t11	*MARK(mesg) = 1;*
t12	*MARK(mesg) = 1;* *MARK(chkpts1) = 1;*
t13	*MARK(chkpts1) = 1;*
t21	*MARK(mesg) = 1;*
t22	*MARK(mesg) = 1;* *MARK(chkpts2) = 1;*
t23	*MARK(chkpts2) = 1;*

REFERENCES

[1] J.W. Young, "A first order approximation to the optimum checkpoint interval", *Communications of ACM*, 17, Sept 1974, pp. 530-531.

[2] A. Brock, "An analysis of checkpointing", *Proceedings of the 6th Inter. Conf. on Production Research*, Aug 24-29, 1981, pp. 707-711.

[3] K.M. Chandy, "A survey of analytic models of rollback and recovery strategies", *Computer*, vol. 8, May 1975, pp. 40-47.

[4] K.M. Chandy, J.C. Browne, C.W. Dissly, and W.R. Uhrig, "Analytic models for rollback and recovery strategies in database systems", *IEEE Transactions on Software Engineering*, vol 1, Mar 1975, pp. 100-110.

[5] E. Gelenbe, "On the optimum checkpoint interval", *Journal of ACM*, vol. 26, Apr. 1979, pp. 259-270.

[6] Asser N. Tantawi and Manfred Ruschitzka, "Performance analysis of checkpointing strategies", *ACM Transactions on Computer Systems*, vol. 2, no. 2, May 1984.

[7] D.B. Hunt and P.N. Marinos, "A General-Purpose Cache-Aided Rollback Error Recovery Technique", *Proc. 17th Symp. Fault-Tolerant Computing, IEEE*, no. 778, 1987, pp. 170-175.

[8] R.E Ahmed, R.C. Frazier, and P.N. Marinos, "Cache-Aided Rollback Error Recovery algorithm for shared memory multiprocessor systems", *Proc. 20th Symp. Fault-Tolerant Computing, IEEE*, no. 2051, 1990, pp. 82-88.

[9] K.L. Wu, W.K. Fuchs and J.H. Patel, "Error recovery in shared-memory multiprocessors using private caches", *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, no. 2, Apr. 1990, pp. 231-240.

[10] Parameswaran Ramanathan and Kang G. Shin, "Use of common time base for checkpointing and rollback recovery in a distributed system", *IEEE Transactions on Software Engineering*, vol. 19, no. 6, Jun. 1993, pp. 571-583.

[11] J.L.W. Kessels, "Two designs of a fault-tolerant clocking system", *IEEE Transactions on Computing*, vol. C-33, no. 10, Oct. 1984, pp. 912-919.

[12] C.M. Krishna, K.G. Shin, and R.W. Butler, "Ensuring fault tolerance of phase-locked clocks", *IEEE Transactions on Computing*, vol. C-34, no. 8, Aug. 1985, pp. 752-756.

[13] K.G. Shin and P. Ramanathan, "Clock synchronization of a large multiprocessor system in the presence of malicious faults", *IEEE Transactions on Computing*, vol. C-36, no. 1, Jan. 1987, pp. 2-12.

[14] J.A. Couvillion, R.Freire, R. Johnson, W.D. Obal II, M.A. Qureshi, M. Rai, W.H. Sanders, and J.E. Tvedt, "Performability modeling with UltraSAN", *IEEE Software*, pp. 69-80, 1991.

[15] W.H. Sanders, W.D. Obal II, M.A. Qureshi, and F.K. Widjanarko, "UltraSAN Version 3: Architecture, Features, and Implementation", *Proceedings of the AIAA Computing in Aerospace 10 Conference, San Antonio*, pp. 327-338, March 28-30, 1995.

[16] S.R. McConnel, D.P. Siewiorek, and M.M. Tsao, "The measurement and analysis of transient errors in digital computing system", *Proc. 9th Annual Intr. Conf. on Fault-Tolerant Comput. (FTCS-9)*, Jun. 1979, pp. 67-70.

[17] S.E. Butner and R.K. Iyer, "A statistical study of reliability and system load at SLAC", *Central for Reliability Computing, Standford Univ.*, Tech. Rep., Jan 1980.

[18] Daniel P. Siewiorek and Robert S. Swarz, "Reliable Computer Systems: Design and Evaluation", Digital Press, second edition, 1992.

[19] Paul R. Rider, "The method of moments applied to a mixture of two exponential distributions", *Annals of Mathematical Statistics*, vol. 32, 1961, pp. 143-147.

[20] V.S.S. Nair and J.A. Abraham, "Hierarchical design and analysis of fault-tolerant multiprocessor systems using concurrent error detection", *IEEE 20th Inter. Symp. on Fault-tolerant Computing*, Jun. 1990, pp. 130-136.

[21] Joseph L.A Hughes, "Error detection and correction techniques for dataflow systems", , pp. 318-321.

[22] M. Ball and F. Hardie, "Effects and detection of intermittent failures in digital systems", *AFIPS Conf. Proceedings*, vol. 35, 1969, pp. 329-335.

[23] Omur Tasar and Vehbi Tasar, "A study of intermittent faults in digital computer", *AFIPS Conf. Proceedings*, vol. 46, 1977, pp. 807-811.

[24] Yann Hang Lee and Kang G. Shin, "Design and Evaluation of a Fault-Tolerant multiprocessor using hardware recovery blocks", *IEEE Transactions on Computers*, vol. C-33, no. 2, Feb. 1984, pp. 113-124.

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

[26] Ravishankar K. Iyer and Mei Chen Hsueh, "Analysis of field data on computer failures", *Journal of Comput. Sci. and Technol.*, vol. 5, no. 2, 1990, pp. 99-108.

[27] M.C. Hsueh, R.K. Iyer and K.S. Trivedi, "Performability modeling based on real data: A case study", *IEEE transactions on computers*, vol. 37, no. 4, April 1988, pp. 478-484.

[28] J.F. Meyer, A. Movaghar, and W.H. Sanders, "Stochastic activity networks: Structure, behavior, and application", *Proceedings of the International Conference on Timed Petri Nets*, July 1985, pp. 106-115.

[29] W.H. Sanders and J.F. Meyer, "Reduced base model construction methods for stochastic activity networks", *IEEE Journal on Selected Areas in Communications*, vol. 9, January 1991, pp. 25-36.

[30] J.F. Meyer and L. Wei, "Analysis of workload influence on dependability", *Proc. of the 18th Intern. Symp. on Fault Tolerant Computing System*, 1988, pp. 84-89.

[31] J.C. Laprie, "Dependability: Basic Concepts and Terminology", *WG 10.4 - Dependability Computing and Fault Tolerance*, August 1994.

[32] Nitin H. Vaidya, "Another Two-Level Failure Recovery Scheme: Performance Impact of Checkpoint Placement and Checkpoint Latency", *Texas A&M University, College Station, Department of Computer Science, Technical Report 94-068*, December 1994.

[33] Yi-Min Wang, Pi-Yu Chung and W. Kent Fuchs, "Tight Upper Bound on Useful Distributed System Checkpoints", *University of Illinois at Urbana-Champaign*, 1994.

[34] Alex S. Papadopoulos and Ram C. Tiwari, "Bayesian Approach to Life Testing and Reliability Estimation Under Competing Exponential Failure Distributions", *Microelectron. Reliab.*, Vol. 29, No. 6, pp. 1039-1050, 1989.

[35] Rajesh Singh, S. K. Upadhyay, and Umesh Singh, "Bayesian Estimators of Exponential Parameters Utilizing A Guessed Estimate of Location", *Microelectron. Reliab.*, Vol. 33, No. 4, pp. 521-527, 1993.

[36] Dallas R. Wingo, "Maximum Likelihood Estimation of Exponential Distribution Parameters Using Interval Data", *Microelectron. Reliab.*, Vol. 33, No. 1, pp. 57-62, 1993.