Febuary 1994

UILU-ENG-94-2204
CRHC-94-03

Center for Reliable and High-Performance Computing

DESIGN FOR DEPENDABILITY: A SIMULATION-BASED APPROACH

Kumar K. Goswami

Coordinated Science Laboratory
College of Engineering
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) UILU-ENG-94-2204 CRHC-94-03	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Coordinated Science Lab University of Illinois 6b. OFFICE SYMBOL (If applicable) N/A	7a. NAME OF MONITORING ORGANIZATION Office of Naval Research, National Aeronautics Space Administration, Tandem, and CSC
6c. ADDRESS (City, State, and ZIP Code) 1308 W. Main St. Urbana, IL 61801	7b. ADDRESS (City, State, and ZIP Code) Arlington, VA; 22217 Moffet Field, CA 95043 Cupertino, CA 95014 and Falls Church VA 222[illegible]C
8a. NAME OF FUNDING / SPONSORING ORGANIZATION 7a 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER N00014-91-J-1116, NASA NAG 1-613, Tandem, GSA CSC 468969
8c. ADDRESS (City, State, and ZIP Code) 7b	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)

Design for Dependability: A Simulation-Based Approach

12. PERSONAL AUTHOR(S) GOSWAMI, Kumar K.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical	FROM _____ TO _____	94-02-02	148

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	dependability, simulation, acceleration, algorithms, hybrid

19. ABSTRACT

This research addresses issues in simulation-based system level dependability analysis of fault-tolerant computer systems. The issues and difficulties of providing a general simulation-based approach for system level analysis are discussed and a methodology that address and tackle these issues is presented. The proposed methodology is designed to permit the study of a wide variety of architectures under various fault conditions. It permits detailed functional modeling of architectural features such as sparing policies, repair schemes, routing algorithms as well as other fault-tolerant mechanisms, and it allows the execution of actual application software. One key benefit of this approach is that the behavior of a system under faults does not have to be pre-defined as it is normally done. Instead, a system can be simulated in detail and injected with faults to determine its failure modes.

The thesis describes how object-oriented design is used to incorporate this methodology into a general purpose design and fault injection package called DEPEND. A software model is presented that uses abstractions of application programs to study the behavior and effect of software on hardware faults in the early design stage when actual code is not available. Finally, an acceleration technique that combines hierarchical simulation, time acceleration algorithms and hybrid simulation to reduce simulation time is introduced.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code) / 22c. OFFICE SYMBOL

An extensive simulation based study of the Tandem Integrity S2 is conducted with DEPEND to illustrate its capabilities. The system is evaluated to determine how it handles near-coincident errors caused by correlated and latent faults. Issues such as memory scrubbing, re-integration policies and workload dependent repair times which affect how the system handles near-coincident errors are also evaluated. Application specific analysis of the existing and a newly proposed scrubbing scheme is performed. Unlike other simulation-based dependability studies, measurements from injection experiments conducted on an actual Tandem Integrity S2 computer are used to validate the software model and the simulation model of the Integrity S2. Results from a detailed simulation model that does not use acceleration are used to validate the acceleration technique.

DESIGN FOR DEPENDABILITY: A SIMULATION-BASED APPROACH

BY

KUMAR K. GOSWAMI

B.S.C.S., Embry-Riddle Aeronautical University, 1982
M.S., University of Illinois at Urbana-Champaign, 1988

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1993

Urbana, Illinois

Abstract

This research addresses issues in simulation-based system level dependability analysis of fault-tolerant computer systems. The issues and difficulties of providing a general simulation-based approach for system level analysis are discussed and a methodology that address and tackle these issues is presented. The proposed methodology is designed to permit the study of a wide variety of architectures under various fault conditions. It permits detailed functional modeling of architectural features such as sparing policies, repair schemes, routing algorithms as well as other fault-tolerant mechanisms, and it allows the execution of actual application software. One key benefit of this approach is that the behavior of a system under faults does not have to be pre-defined as it is normally done. Instead, a system can be simulated in detail and injected with faults to determine its failure modes.

The thesis describes how object-oriented design is used to incorporate this methodology into a general purpose design and fault injection package called DEPEND. A software model is presented that uses abstractions of application programs to study the behavior and effect of software on hardware faults in the early design stage when actual code is not available. Finally, an acceleration technique that combines hierarchical simulation, time acceleration algorithms and hybrid simulation to reduce simulation time is introduced.

An extensive simulation based study of the Tandem Integrity S2 is conducted with DEPEND to illustrate its capabilities. The system is evaluated to determine how it handles near-coincident errors caused by correlated and latent faults. Issues such as memory scrubbing, re-integration policies and workload dependent repair times which affect how the system handles near-coincident errors are also evaluated. Application specific analysis of the existing and a newly proposed scrubbing scheme is performed. Unlike other simulation-based dependability studies, measurements from injection experiments conducted on an actual Tandem Integrity S2 computer are used to validate the software model and the simulation model of the Integrity S2. Results from a detailed simulation model that does not use acceleration are used to validate the acceleration technique.

Results from the study show that accurate application specific analysis using the software model can produce coverage values that are over 100% different from those obtained with simple

estimations of program behavior. A simulation model using the acceleration technique is shown to produce results that are within the 99% confidence interval of a simulation model that does not use acceleration, but it does so in 7 minutes as opposed to 36 hours. Accurate modeling of the "staggered failure" effect caused by near-coincident errors observed on actual systems show that the impact of correlation (for the Integrity S2) is dependent on the error latency distribution. Simple analytical models that fail to represent this relationship produce mean time between failures (MTBF) that are orders of magnitude lower. Different latency distributions with the same mean are shown to produce statistically significant changes in system MTBF when correlation is considered. Inter-component dependencies, which are a function of the architecture and error latency, are shown to substantially reduce system MTBF. Scrubbing is shown to be ineffective unless the latency times exceed half the cycle time of the scrubber.

Acknowledgements

I am very grateful to my advisor, Professor Ravi Iyer for his continual support and encouragement. His patience, insight and assistance made this thesis possible. I would also like to thank Professors W. Kent Fuchs, Laxmikant Kale, Jane Liu, David Padua and Janak Patel for serving on my dissertation committee.

I would like to thank Luke Young for allowing me to freely use his hybrid injection facility and answer my million and one questions. Special thanks go to Tim Tsai for valuable discussions and for the time he took to run many fault injection experiments for me. His effort made it possible for me to complete the thesis on time. Discussions, debates and collaboration with Jim Barnette, Axel Hein, Dan Olson and Darren Sawyer gave me the second wind that I needed to complete this work. I truly appreciate all their effort and encouragement. Thanks are due to In Hwan Lee, Dong Tang and Gwan Choi, who were always there to answer questions, review my papers and offer suggestions.

I was fortunate to have a very supportive and special bunch of friends at CRHC who made my stay in Urbana fun and enjoyable: Jeff Baxter, Paul Chen, Bob Dimpsey, John Fu, John Holm, Bob Janssens, Antoine Mourad, Mike Peercy, Paul Ryan, Jonathan Simonson, and Nancy Warter. I will miss their company. I would like to thank Krishna Subramanian for being a dear friend during the final stretch. I am very grateful to a very special friend, Rahul, for his loving warmth and affection (and for those lovely B.D. Baggies). Finally, I am especially indebted to my parents, my two sisters and Karyn whose unwavering support has been my source of strength and inspiration over these past years.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The increasing demand and growing complexity of dependable systems has spurned the need for automated design tools that can reduce design time while ensuring that dependability (e.g. availability, reliability) specifications are met. The focus has been on developing electrical and gate level fault analysis tools. Though these tools are ideal for analyzing LSI and VLSI components they are not suited to analyze a complete system. Dependability is a *system* issue and as such isolated analysis of the individual components is insufficient. Ideally, system level dependability analysis should consider the hardware, the system software, the application and the interaction and inter-dependencies of all of these components under realistic fault scenarios. This is an extremely complex task and the difficulties that arise depend on the approach taken. Currently, two approaches are primarily used for system level dependability analysis. Analytical tools, and in particular tools that represent systems with continuous time Markov chains (CTMC), are used for very high-level trade-off analysis. These tools are best suited for the early design stages when little information is known about the target machine. However, the accuracy of their results is limited by the level of detail to which they can represent a system. The second approach is fault injection of proto-type systems. Though this approach provides the most believable results, it has three major drawbacks. First, it does not provide feedback during the design stages. Second, because the system is fixed, it cannot be used to study alternate configurations. Third, it cannot provide many dependability measures such as availability and mean time between failure (MTBF) because it would require measuring the system for years.

This thesis explores a third approach: functional simulation-based system level dependability analysis. Unlike analytical models, functional simulation can accurately model the functional behavior of the system components, the inter-component dependencies, workload patterns and reconfiguration schemes. This makes the approach more useful during the intermediate stages of system development when more information is available about these system characteristics. Furthermore, if the functional simulation tool can also inject faults, it can be used to perform injection experiments and provide early design feedback. There are a few simulation-based tools that are specifically geared for system level dependability analysis. However, they either handle very specific architectures and fault models or only use probabilistic modeling and hence are essentially extensions of analytical tools. This thesis focusses on the development of an integrated fault injection and functional simulation environment for system level dependability analysis that can be used to evaluate a wide variety of systems. The thesis identifies the major issues that impede the development of such a tool, presents solutions to tackle these issues and uses them to perform a comprehensive study of an actual fault-tolerant machine.

1.1 Related Work

In [19], the authors describe and compare several tools that solve CTMC models. There is increasing work in software and hardware fault injection of proto-type systems [38, 64, 75, 18, 36]. However, there is a lack of simulation tools for system level dependability analysis. Most simulation tools are designed to facilitate performance analysis (CSIM [60], ASPOL [43], SES Workbench [61], RESQ [58]). These tools do not provide the mechanisms needed to interrupt the system and inject faults. VHDL [30] is a powerful hardware specification language but it does not contain built-in facilities to support dependability analysis. NEST [17] is a functional simulation and proto-typing tool used explicitly to analyze distributed networks and system protocols. It is very specialized and has a limited set of facilities to fail links and nodes. Another functional simulation tool called REACT [12] is specifically designed for analyzing alternative TMR architectures. Metasan [56] and the Rainbow Net [35] are Petri-net tools. An extended Petri-net structure is used to input a model and solve it via simulation. Though the Petri-net tools have greater applicability than analytical tools, they rely solely on probabilistic modeling to describe a system. This is a major limitation because, as we show in detail in

Chapter 3, it requires that the user pre-define the fault behavior of a system. Furthermore, these tools provide only a limited set of fault models and provide no mechanism to reduce simulation time explosion.

1.2 Contributions

The focus of this research is the development and use of simulation techniques for system level dependability analysis. Ad-hoc techniques are commonly used to conduct detailed system level dependability analysis using functional simulation. Each system design effort is accompanied by the fresh development of simulation models specific to the architecture being built and the fault models being considered. There has been no prior attempt to characterize and provide a more structured approach to functional simulation-based dependability analysis at the system level. The goals of this thesis are to provide a structured approach and to develop methods that allow a comprehensive study of a system including the hardware and software components. Towards these goals, several issues that impede the development of a general-purpose, integrated fault injection and simulation environment for system level dependability analysis are identified. Broadly, they are:

- Modeling a large variety of components.
- Coping with the large fault model domain.
- Reducing model development time and model complexity.
- Incorporating software behavior under faults in a simulation-based study.
- Reducing simulation time explosion.

Methods to tackle these issues are discussed and presented. A synergy of these methods is used to analyze an actual system under realistic fault conditions for specific applications. The methods are validated by comparing the simulation results with those obtained from fault injection experiments on an actual machine. Finally, the methods are incorporated into a simulation tool called DEPEND.

1.3 Overview

Chapter 2 discusses the major issues in developing a general-purpose system level functional simulation tool. Chapter 3 presents the approach used to tackle the first three issues mentioned earlier. It also presents the DEPEND integrated simulation and fault injection environment that incorporates these approaches. The chapter presents the key features of the tool and the fault models it provides. A table comparing DEPEND's features with existing simulation-based system level dependability analysis tools is also included.

Having described the essential features, Chapter 4 illustrates the uses and benefits of the tool. One can ask, given the large number of analytical and Petri-net based simulation tools, what is the need for functional simulation tools for system level dependability analysis? What additional information and capabilities can they provide over analytical tools and fault-injection environments? The goal of Chapter 4 is to answer these questions with a few examples.

DEPEND is used to analyze a triple-modular redundant (TMR) system and in particular the Tandem Integrity S2 fault-tolerant system under realistic fault scenarios. It is well established that this system is very effective against single faults [34, 75]. An important question is how such systems cope with near-coincident errors caused by correlated and latent faults. Architectural issues that have a bearing on how the system handles near-coincident faults include memory scrubbing, re-integration policies and workload dependent repair times. Since the reliability of a system is also affected by the application and because most fault-tolerant systems are geared to perform specific functions, it is also imperative to study such systems within the context of an application. To study these issues, DEPEND is used to simulate the Integrity S2 system and evaluate the combined effect of all these factors. Furthermore, the simulation of the Integrity S2 system is validated by comparing the results of the simulations with measurements obtained from fault injection experiments conducted on a production Integrity S2 machine. Chapter 5 describes the salient features of the Tandem Integrity S2 system. It also presents the basic simulation model developed with DEPEND.

The focus of Chapter 6 is on application specific dependability analysis. It presents the simulation-based model developed to analyze software behavior under hardware faults and to study the impact of latent errors on system dependability. A probabilistic graph model is proposed that represents program behavior at an abstract level and can obtain key application

specific parameters that affect system dependability. These parameters are used to perform application specific analysis of functional detection schemes. It is shown that if alternate, simple techniques are used to model application program behavior the system dependability measures obtained can be grossly inaccurate.

Chapter 7 presents the acceleration technique that reduces simulation time explosion. The technique uses a combination of hierarchical simulation, time acceleration, and hybrid simulation. This technique makes it possible to conduct a detailed simulation study of the TMR system. Unlike other simulation acceleration techniques, this technique is general and not limited to any particular type of simulation model.

Chapter 8 describes the simulation experiments in detail and presents results of the study. The impact of correlated errors, latent errors and various repair and memory scrubbing schemes are quantized and dependability bottlenecks are identified. Results obtained with simplistic fault models are compared with those obtained with DEPEND's fault models to illustrate the need for accurate modeling of the fault occurrence process. Finally, Chapter 9 summarizes the main points of the thesis and the findings of the TMR study. It also contains an assessment of functional simulation-based dependability analysis and suggests future extensions of this research.

Chapter 2

Development Issues

Functional simulation (also referred to as behavioral simulation) allows more detailed and accurate modeling of computer systems. But this accuracy comes at a cost. Accurate modeling requires greater information about the system components, more knowledge about their specific fault models, and more time in developing a simulation. There are at least five issues that impede the development of general-purpose, functional simulation tools for system level dependability analysis. The first is a lack of well established system level fault models. This is partly due to the second issue which is a large and varied component domain. At the gate level, the basic components are gates with single functions and well defined interconnections. At this level, it is possible to establish a fault model such as the single stuck-at fault model that can consistently be applied to all gates to model their fault behavior. At the system level, the basic components include CPUs, communication channels, disks, software systems and memory. The components have complex inputs, perform multiple functions, have varied physical attributes (e.g. hardware and software) and complex interconnections. In addition to the diversity of the components that comprise a system, two similar components (such as two CPUs) can have different functions and behavior. This makes it difficult to establish a single fault model that can be consistently applied to all components. Limiting the types of components or fault models represented is one solution but it restricts the tool's applicability (e.g. NEST).

The third issue, which is especially significant when simulating large complex systems, is the effort and time required to develop a functional simulation model. For fault injection studies and dependability analysis, there are two factors that contribute to this. One is the time and

effort needed to describe the detailed functionality of the system components. The other is the time and effort required to inject faults, initiate repairs, abort, reschedule and synchronize events and maintain a whole host of fault statistics. As the number of components in the system becomes large, a well formulated, structured and automated approach is needed to contend with the complexity.

The fourth issue is the impact of the software on system dependability. Dependability studies have tended to focus on a system's hardware components. But as the hardware becomes more reliable, the software component is becoming a more dominant factor [27]. The behavior and effect of the software on hardware faults is a major concern. The process of software execution determines parameters such as detection latency times, probability of propagation and propagation times. These parameters have a bearing on the effectiveness of functional detection and repair schemes, which in turn have a huge impact on the dependability of a system. Since the process of software execution is determined by an application program, it is imperative to perform application specific analysis of functional detection schemes and fault-tolerant mechanisms in the early design stages. Hence, methods are needed that allow the designer to incorporate the application into the overall dependability study.

The fifth issue, an extremely important issue and one that is common to both functional and probabilistic simulation, is simulation time explosion. This occurs when the system modeled has extremely small failure probabilities requiring large simulation runs to obtain statistically significant results. This is especially a problem with functional simulation because its primary benefit is detailed modeling which further contributes to simulation time explosion. Unless effective techniques are found to reduce simulation execution time, functional simulation-based dependability analysis will not be generally applicable. Two approaches to reduce simulation time that have been extensively studied in the literature are importance sampling and parallel simulation. Importance sampling [50, 41, 72] is a statistical technique that skews the probability of failure, thereby reducing the number of trials needed to collect failure data, and then "unskews" the results obtained. Though this heuristic has been shown to be very effective in selected cases, it has several drawbacks. It is not generally applicable, it is meant for use with probabilistic simulation, and it cannot be used to obtain many steady-state dependability measures (e.g. mean time between failure) for general failure distributions. Hence, if this is the

sole acceleration technique used, it will preclude the use of behavioral models and the generality of the tool.

Parallel simulation breaks up the simulation model into several processes and executes them on different machines. Many heuristics [2, 33, 32, 49, 23] have been developed to synchronize the simulation so that events generated on the separate processors are executed in chronological order. Unfortunately, the effectiveness of these schemes are application dependent and work well only with applications that require little synchronization. The focus of past research has been on synchronization heuristics, even though dividing a simulation into separate logical processes and assigning them to processors is not a trivial problem and has a significant bearing on the speed-up achieved. Even when this approach is successful for certain applications, the speed-up achieved is not linear and is limited to the number of processors available to the user.

To summarize, a general-purpose simulation-based system level dependability analysis tool must effectively handle the large component and fault model domain, provide an environment that facilitates the development of functional simulation models, incorporate the impact of software to allow application specific analysis and furnish generally applicable methods to accelerate the simulation.

Chapter 3

Simulation Environment

This chapter presents the approach used to solve the first three issues mentioned in the previous chapter and it describes the key features of the integrated fault injection and simulation tool, DEPEND, that has been developed.

3.1 Approach

The object-oriented design paradigm is used to tackle the first three issues discussed earlier. Object-oriented design is the construction of a software system as a structured collection of abstract data type implementations [47]. A class is an implementation of an abstract data type and an object is a instantiation of a class. A class contains private internal data structures and methods, which are analogous to functions and procedures, that manipulate its internal data. The word *collection* reflects that each class is an independent, useful component of the system. The key, however, is the word *structured* which reflects an intelligent division of the system into classes and the existence of important relationships among them.

Through an intelligent, structured, development of classes, the problems of large component and fault domain are tackled. A combination of two criteria: modular decomposition and modular composability [46], is used to break up the simulation process into individual classes. Modular decomposition consists of breaking down a problem into small elements whereas modular composition favors production of elements that can be freely combined with each other to provide new functionality. Two relations, *clientship* and *inheritance* [6] are used to combine the elements (or classes). In a clientship, one class is a client of another that provides the service.

For example, a class that implements a stack could rely on a singly linked list for its implementation and thus be a client of the class that implements the linked list. Inheritance, a central mechanism of object oriented programming, allows a new class to be derived from an existing class. The new (or derived) class inherits the features of and is an extension or a specialization of the existing class, the ancestor. An example will help to clarify how modular decomposition and composition are used to tackle the large fault model domain. The fault injection process can be broken down into two processes. One process determines when to inject a fault and interrupt the system and another process responds to the fault injected. The first process is common to all fault injection methods. It encapsulates the various mechanisms used to determine the arrival time of a fault and interrupt the system. The second process is the fault model and is specific to the component and the type of fault being injected. If we encapsulate the first process into a class called the *fault injector class* and combine it with one of a multiple *fault model* classes, it is possible to create an environment that can inject many types of faults. The *fault injector* class is ignorant of the types of faults injected and the components injected. The *fault model* class encapsulates this component and fault specific information. As a result, this approach can be used to inject not only electronic components but mechanical components. This same structured, modular approach is used to cope with the large component domain; common aspects of similar components are encapsulated in a class which then combines with other classes to provide more specific functionality. Furthermore, because users can specify the classes that model component specific behaviors, the tool is not limited to any pre-defined set of fault models or component types.

A library of pre-developed classes is used to reduce the time and effort needed to develop simulation models. The classes in the library provide the skeletal foundation necessary to rapidly model an architecture and conduct simulated fault injection experiments. The classes form a hierarchy with fundamental classes at the top and derived classes, which inherit these fundamental classes, at the bottom. The complexity and specificity of the classes increase as the bottom of the hierarchy is reached. The fundamental classes provide functionality that is general and widely used. Such functions include the fault injection mechanism, fault tracing where a list of all faults injected is recorded and reported, and maintaining fault statistics (e.g. MTBF, MTBR, Availability, Coverage). Other fundamental classes include events, mailboxes, queues, stacks and servers. These are basic constructs needed for functional simulation. At the

bottom of the hierarchy are classes that model components found in fault-tolerant computer systems: voters, self-checking components, fault tolerant servers, communication links and triple-modular redundant systems. These classes are developed incrementally by combining (clientship and inheritance) several fundamental classes and adding new methods that are specific to the class.

Object oriented design facilitates reusability [47, 5] and the classes in the library were specifically designed for reuse. Model development time is reduced by the library because it provides a kernel of classes that can be used as is, or readily inherited to create more specific classes thus eliminating the need to develop them from scratch. The complexity of injecting faults, initiating repairs, maintaining statistics etc. is reduced because each derived class inherits fundamental classes that provide these services. As a result, each class automatically injects its own faults, initiates its own repairs and maintains all statistics. At the beginning of a simulation, the user provides the fault and repair distributions and at the end, the user polls the classes to obtain the fault statistics. This automation makes it possible to model large systems more easily.

3.2 Architecture

DEPEND is an integrated design and fault injection environment. It provides facilities to rapidly model fault-tolerant architectures and conduct extensive fault injection studies. Figure 3.1 shows the DEPEND architecture. DEPEND can be executed on various hardware platforms. The hardware specific code is contained in the simulation engine which provides a pseudo-parallel run time environment. The objects provide the skeletal framework with which a user can rapidly build a simulation model for fault injection and dependability analysis.

DEPEND is a functional, discrete-event, process-based [37, 60] simulation tool. The system behavior is described by a collection of asynchronous processes that interact with one another. These processes (also called coroutines) are independent threads of execution that can be suspended and resumed at user specified points in the code [53]. The simulation engine implements processes entirely at the user-level without operating system services. As a result, the processes are light weight and do not execute simultaneously. The simulation engine provides a pseudo-parallel environment. It does this by sequentially executing all processes scheduled to run at

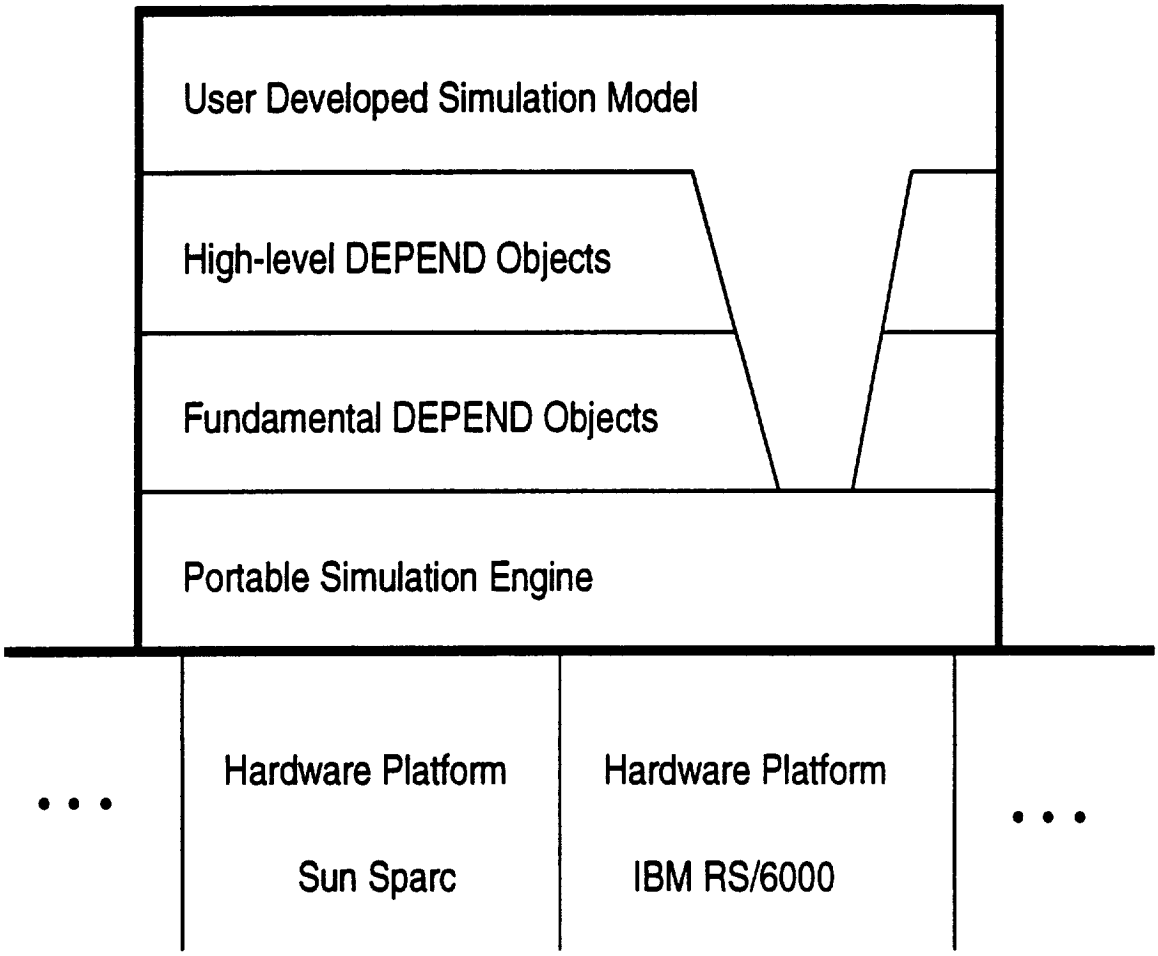


Figure 3.1: The DEPEND architecture.

the same time and then incrementing the simulation clock. The current version of DEPEND uses CSIM to support the process-based paradigm and the pseudo-parallel environment. CSIM uses a single function call stack for all processes. When an active process is suspended, its runtime stack frames and registers are copied into its static process control block. When a process is resumed, the saved stack is copied back onto the runtime stack and the saved registers are restored [15]. The original CSIM has been modified to support fault-injection and the ability to abort, reschedule and kill processes.

An alternative to process-based simulation is event-driven simulation where a system is described by a set of events and their associated actions. Even though process-based simulation has a large context-switching overhead not present in event-driven simulation, it was selected for several reasons. It is an effective and intuitive way to model system behavior, repair schemes, and system software in detail. It facilitates modeling of inter-component dependencies, especially when the system is large and the dependencies are complex, and it allows actual programs to be executed within the simulation environment with minimal retrofitting.

3.2.1 User Environment

The steps required to develop and execute a model are shown in Figure 3.2. The user writes a control program in C++ using the objects in the DEPEND library. The program is then compiled and linked with the DEPEND objects. The model is executed in the pseudo-parallel run time environment. Here, the assortment of objects including the fault injectors, CPUs and communication links execute simultaneously to simulate the functional behavior of the architecture. Faults are injected and repairs are initiated, according to the user's specifications and a report containing the essential statistics of the simulation is generated.

3.2.2 The Object Library

The object library contains fundamental and complex classes. A fundamental class is not a derived class. A complex class may be a derived class, it may consist of a combination of several classes or both. The classes were designed with four criteria:

- Simulate the general behavior of a computer system component (Decomposability & composability).

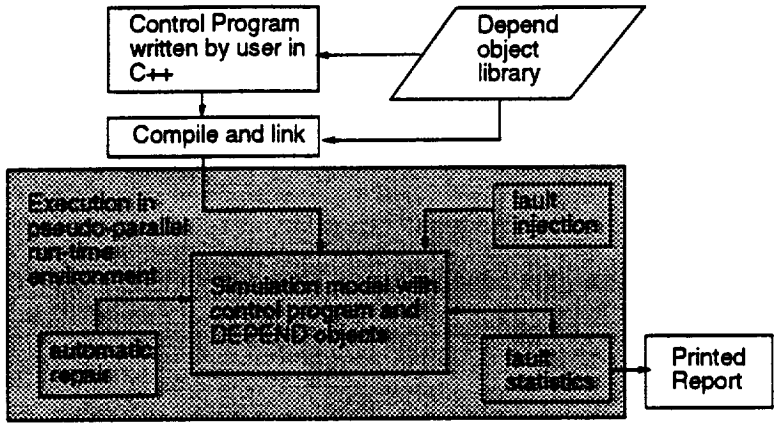


Figure 3.2: Steps in developing and simulating a model with DEPEND.

Name	Description
Event	Basic synchronization mechanism. Processes wait at and set events.
Active_elem	Simulates a server with various queueing disciplines (FCFS, round-robin, etc.). Processes queue at server to model resource contention.
Injector	Automatically injects faults based on statistical distributions and trace files.
Fault Reporter	Compiles fault statistics. Displays MTBF, MTBR, availability and coverage. Generates output of every fault injected and repair attempted.

Table 3.1: A few fundamental classes.

- Allow users to specify key parameters (Parameterization).
- Provide default functions to minimize design time (e.g. fault models, sparing policies).
- Permit easy reuse.

The fundamental classes provide basic functions like injecting faults and compiling statistics. Some key fundamental classes are described in Table 3.1. Complex classes created from these fundamental classes simulate components found in fault-tolerant architectures. Such components include CPUs, self-checking processors, N-modular redundant processors, communication links, voters and memory. This kernel of classes can be combined and replicated to simulate a wide range of fault-tolerant architectures.

Name	Description
Voter	Simulates a basic voter with timeout. Default voting scheme: byte by byte comparison. Allows user defined voting algorithms.
Server	Inherits Active_elem. Simulates server with spares. Three sparing policies: no spare, graceful degradation, stand-by sparing. Automatic repair and reconfiguration. Automatic injection of faults.
Link	Simulates communication channels. Several fault types: link dead, packet corruption, packet loss and user defined faults. Automatic retry.
NMR	Simulates dual self-checking, triple-modular redundant and N-modular redundant components.
Fault Manager	Simulates software fault management schemes. Logs faults and shuts off components which exceed their fault threshold.

Table 3.2: Some key classes that model the architectural components of a fault-tolerant system.

Table 3.2 lists a few complex classes in the library. A detailed description of all classes can be found in [22]. The next subsection describes the basic fault models provided by DEPEND and the following subsections present three essential classes in the library: the fault injector, the fault-tolerant server, and the fault-tolerant communication link. This is followed by an example simulation program.

3.2.3 Fault Models

Functional fault models are used to simulate the system level manifestation of gate-level faults such as stuck-at faults. Functional fault models are used because they are best suited for system level fault injection where the focus is on the behavior of a component due to a fault. Furthermore, at this high level of integration, gate-level implementation faults become prohibitive [14]. Functional testing approaches and fault models for microprocessors have been presented in [69, 65, 52]. DEPEND provides four types of faults which can be used to simulate specific fault behavior:

- Status faults.
- Process faults.

- Server faults.
- Data faults.

The simplest fault is the *status fault.* It is called a status fault because a flag is set to indicate that the status of the component is faulty. Methods of the injected component's class can be queried to determine its state and, based on the status, specific fault actions can be invoked.

A *process fault* either aborts or kills a process. Aborting a process entails dequeuing it from an event or server queue, setting the "aborted" flag in its program control block (PCB) and returning control to the process. The process can then poll its "aborted" flag and if set take special action. Killing a process entails dequeuing the process, releasing all resources it has acquired, and destroying its PCB.

A *server fault* can be injected into any class that is derived from the *Active_elem* class (see Table 3.1). A server fault causes the server to abort all processes queued for service and update its fault and performance statistics. The aborted processes can then take remedial action such as rollback or restart to simulate the effect of the fault.

When actual programs are executed with DEPEND, the *data fault* can be used to corrupt data elements used by the program. Given a mask, a size and a pointer to the data region, a byte is selected randomly from the region and XORed with the mask.

Note that these basic fault models are general and only interrupts the simulation (e.g. stops a server and dequeues all requests). All specific fault models, including the default fault models provided with DEPEND classes, are built from these core fault models. For instance, the *data fault* model is used by *FT_link* (see below) to corrupt the contents of message packets routed by the link. The basic fault models or the default fault models provided with a class can be specialized by the user to create more specific fault models. These fault models can be derived from low-level simulations or from measurements of existing systems. For instance, Choi [10] simulates a processor at the gate-level and executes a workload while injecting transient faults. The results of the faults on the behavior of the workload and the failure mode are stored in a fault dictionary. The fault dictionary can then serve as the basis for the processor's system-level fault model. Each time a process is aborted due to a transient fault in a server, one of the failure modes can be selected randomly and used to model the process's failure mode.

3.2.4 Fault injector

The *fault injector* is a fundamental object of DEPEND. It encapsulates the mechanism for injecting faults. To use the injector, a user specifies the number of components, the time to fault distribution for each component, and the fault subroutine which specifies the fault model. The distributions supported are constant time (mostly used for debugging), exponential, hyperexponential and Weibull. The object also allows user specified distributions. When initialized, the injector samples from a random number generator to determine the earliest time to fault, sleeps until that time and calls the fault subroutine.

Initially, the injector used conditional failure distributions to determine the time to next fault, under the assumption that a set of components have independent, identically distributed failure distributions. For instance, for a system with two components using a gracefully degrading sparing policy, the time to failure of the second component, X, given that the first component failed at time t is given by $Pr[X < x|X > t]$. For a Weibull time to fault distribution with rate parameter λ and shape parameter α, the conditional distribution is:

$$Pr[X < x|X > t] = 1 - exp^{\lambda[(x+t)^{\alpha} - t^{\alpha}]} \tag{3.1}$$

This approach has several drawbacks. For non-exponential distributions, the conditional distributions become complex and cumbersome as repaired components are re-integrated into the system or cold spares are activated. The conditional distributions depend on the sparing policy. As a result, additional information regarding sparing policies has to be specified. The routines used to generate random samples are more compute bound. Generating a random sample using the inverse transform method for equation 3.1 requires and additional subtraction and an extra call to the math library's **power()** function than for a Weibull distribution. For large simulation runs, where thousands of faults are injected, the time spent on these additional calls becomes significant.

The current version of the injector uses a table-based approach. An entry is kept for each component, specifying its condition (OK, Failed), injection status (Injection off, Injection on), time to fault distribution, and time to next fault. The algorithm used to determine which component to inject is:

```
Initialize (performed one time)
    do for all components
```

```
        if (component is OK & On)
            compute and store time to fault
        else
            time to fault is ∞
        end if
    end do

Main body
    do forever
        find minimum_time_to_fault among components
        sleep (minimum_time_to_fault - current_time)
        if (sleep not aborted)
            call fault subroutine
            set time to fault of this component to ∞
        end if
    end do
```

Any time a component is repaired or turned on, its time to fault is computed and entered in the table. The injector is then awakened so that it takes the new component's fault time into account.

The table-based approach is versatile. The time to fault distributions of the components do not have to be identical and any user specified distribution can be supported. For example, the full "bathtub" reliability curve can be model with DEPEND. The table-based approach automatically takes the age of each component into account without using conditional probability distributions and averts the problem with modeling local and global times found in most analytical tools [71].

DEPEND provides a workload dependent injection facility to model the workload/failure dependency observed in [7, 31]. It can be used to test systems under stress conditions. To implement a workload dependent injection strategy, a statistical clustering algorithm is first used to identify high-density regions of the workload. These regions (or states) are used to specify a state transition diagram that characterizes the workload [29]. Associated with each state is a visit counter which counts the number of visits to that state and a fault rate, λ, which the system experiences in that state. The user provides a workload function which the injector polls periodically to identify the workload state and to update its visit counter. For

example, the workload function may be the utilization of a processor or it may be any other function that provides a measure between 0 (low workload) and 1 (high workload). Based on an *injection_interval* specified by the user, the information from the state transition diagram is used to estimate a weighted average failure arrival rate (Wgt_lambda) as follows:

$$Wgt\_lambda = \sum_{i=1}^{N} visit\_ratio_i \times \lambda_i$$

where:

N is the number of states

$$visit\_ratio_i = \frac{counter\ for\ state_j}{total\ visits\ to\ all\ states}$$

Once Wgt_lambda is determined, it is used to compute the probability of a failure injection ($P\_inject(t)$) over the last interval t (= *injection_interval*) as follows:

$$P\_inject(t) = 1 - e^{-Wgt\_lambda \times t}$$

The *fault injector* illustrates what we mean by "providing a simulation framework". The injector provides the basic algorithms and mechanisms needed to inject faults allowing the user to concentrate on the application specific aspects, the fault model and the simulation of a fault. Furthermore, the modular, object-oriented approach allows the user to easily experiment with different time to fault distributions, fault models and workload functions. Other DEPEND objects are similarly designed to provide the functionality that are commonly needed without restricting the applicability of the object.

3.2.5 Fault-tolerant server

The *server* class, which is typically used to model CPUs and other processors, is an example of a complex class (Figure 3.3) that is built from several elementary classes. It uses a *fault injector* class to inject faults, the *active_elem* class to model the behavior of a server, and other classes to compile and output fault statistics. It inherits methods from *active_elem* to simulate the acquisition, the use and the release of a server and to provide several service disciplines including, first come first serve (FCFS), round-robin and pre-emptive round-robin.

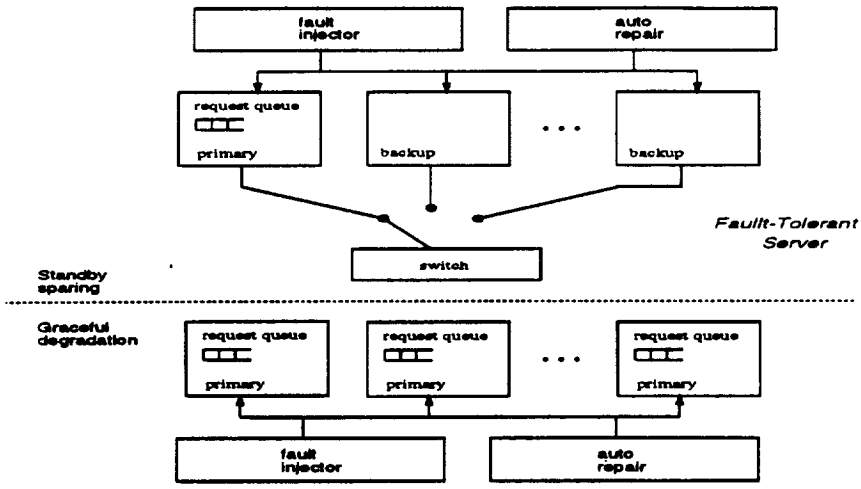


Figure 3.3: The Fault-Tolerant Server Object.

The server offers three sparing modes: the no spare mode, the *graceful degradation* mode, and the *standby cold sparing* mode. In the graceful degradation mode, all the spares operate on incoming requests. The entire server fails when all the spares become faulty. In the standby sparing mode, only the primary server operates on requests. When it fails, a reconfiguration takes place and a healthy spare becomes the primary server. The entire server continues to function as long as there is at least one healthy server. The number of spares and the type of sparing policy is user selected.

The server provides three default fault types: permanent faults, transient faults and user defined faults. Transient faults last for a specified period of time, after which the server returns to a healthy state where it can be used again. When a transient or permanent fault is injected in a server, the *server fault* model is used to dequeue pending requests an place the server in a hung state. Other fault models can be simulated by specifying user defined faults. To simulate user defined faults, the server invokes a pre-specified fault method, written by the user, that can simulate any fault action a user requires.

The user can customize the behavior of a server by specifying, among other things, the reconfiguration time, repair coverage and the fault arrival rate. The user can also override the default repair and fault behavior of the server by specifying user written methods to be invoked when a fault or repair event occurs.

3.2.6 Fault-tolerant communication link

Another complex class is the *link* shown in figure 3.4. It is designed to simulate various types of communication links. It consists of a redundant set of communication links with redundant connections from the links to the ports. The *link* is built from several classes. Instances of the *server* class are used to model the ports and the links. An instance of the *injector* class is used to inject faults. The rest of the *link* class consists of additional software to model the behavior of a communication medium. The link transfers messages between specified ports. Once a message reaches its destination port, it is queued until a process dequeues the message by invoking the receive() method and providing the port number. To initialize a *link*, the user specifies: the number of redundant links and redundant connections to the links, the number of ports, the time required to send data via the links and the types of faults to be injected. The class offers automatic retry. Messages sent back and forth contain checksums. If a checksum error is detected by a receiving port, a negative acknowledgment is sent triggering a retransmission of that message. The number of retransmissions is user specified. Several default fault models,

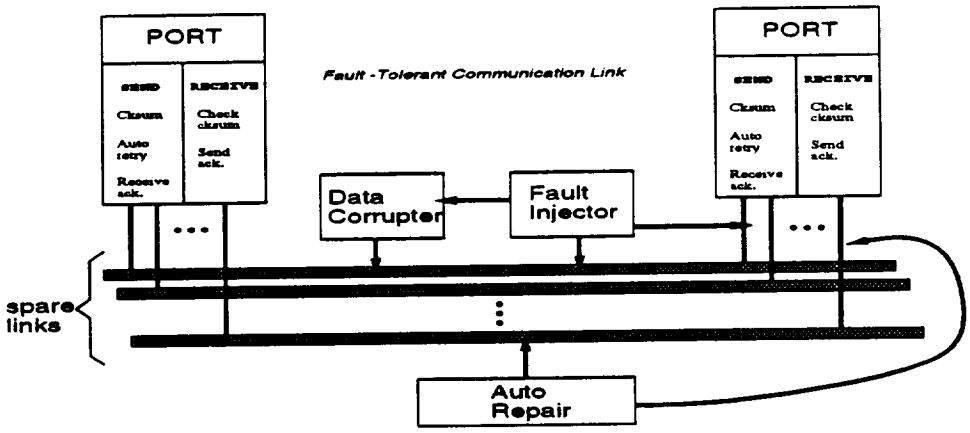


Figure 3.4: The Fault-Tolerant Communication Link.

including faulty link, faulty port, lost message and corrupted message are offered. The *server fault* model is used to simulate a faulty link and faulty ports. A lost message is modeled by failing to send the message to its destination. Messages are corrupted using the *data fault* model. If a link or port is faulty or if a message is lost, the message does not reach its destination. The sender times out waiting for an acknowledgment and then retransmits the message. Like the *server* object, these default fault models can be overridden by supplying user written fault

models. All the redundancy and fault-tolerant features described are switch selectable so the user has a range of options and can select from a simple link with no fault-tolerance to a link with all the fault-tolerance capabilities described.

3.2.7 Programming environment

The C++ object-oriented language is used to specify a DEPEND simulation model. The C++ language was chosen as the user interface to DEPEND for several reasons. First, it means not having to learn an esoteric simulation language; only knowledge of C++ is necessary. Furthermore, the entire C++ programming environment is available to the user. DEPEND enhances C++ by offering simulation facilities not available through regular C++ constructs. The user can use actual C++ or C programs as a part of the simulation model. This makes it possible to test proto-type software algorithms within DEPEND. C++'s strong type checking makes it easier to write large simulations efficiently with fewer bugs than other languages. Finally, C++ produces efficient code, and the C++/UNIX environment is widely available.

Figures 3.5 and 3.6 show a simple, example program that illustrates the programming environment and some of DEPEND's features [1]. The program models two processes that communicate via a fault-tolerant communication link. The link is declared in line 2 to consist of 2 links, using the graceful degradation spareing mode. There are two ports connected to the link and each port has an additional spare port. The **create()** statement in routines **sim()** and **receiver()** distinguish them from ordinary C++ subroutines. **sim()** and **receiver()** are light-weight co-routines that execute in a pseudo parallel environment. Each time the **create()** statement is executed an instance of the co-routine is created.

The main co-routine initializes the link. Lines 8 through 13 specify the cost of sending a message, the time to send acknowledgments and that automatic retry is desired. Lines 14-17 specify the percent of messages lost and corrupted and they also specify the range of bytes (from the beginning of the message) that can be corrupted and the mask to be used. If these two types of faults are not sufficient, the user can opt to specify his or her own fault routine by removing lines 16 and 17 and including line 18 (" //"is a C++ comment delimiter). Lines 19-22 specify the failure rates and distributions of the ports and the links. By default, permanent faults are

[1] The program is written in "pseudo" C++ for simplicity and ease of understanding.

```
01) #define NUM 2
        // link with one spare and two ports and 1 spare port for each port
02) FT_link ln ("link", FT_GRACEFUL_SPARE, 1, 2, 1);
03) Event done("event");
04) int cnt;
05) void main(int argc, char* argv[])                    // main control program
06) {     create("main");
07)       void recv(int);

08)       ln.msg_xferr_time(10.0, 0.01);                 // 10 units startup, 0.01 per/byte cost
09)       ln.set_checksum();                             // checksum the messages
10)       ln.set_auto_retry();                           // retransmit if there is an error
11)       ln.set_num_try(3);                             // retry 3 times before giving up
12)       ln.set_timeout(500.0);                         // wait 500 units for ack
13)       ln.set_reply_xfer(10.0);                       // 10 units to send reply ack

14)       ln.msg_loss(0.01);                             // 1% msgs lost
15)       ln.msg_corrupt(0.05);                          // 5% msgs corrupted
16)       ln.set_corrupt_range(0, 6)                     // corrupt any of 1st 7 bytes of msg
17)       ln.set_mask('2');                              // XOR byte with mask = '2'
18)       // ln.set_msg_fault_func(&my_fault_func);      // optional, user selected fault

19)       ln.port_exp_inject(0.000001);                  // failure rate of port
20)       ln.port_switch_time(1000.0);                   // time to switch to a spare port
21)       ln.port_switch_coverage(0.999);                // prob. switch is successful
22)       ln.set_weib_inject(x,y);                       // Weibull failure rate for links

23)       ln.detailed_record_on();                       // detailed record of all injections
24)       ln.finject_start();                            // start the injector

25)       for (i=1 to NUM)                               // start processes that send and
26)         receiver(i);                                 // receive msgs at these ports
27)       done.wait();                                   // wait for simulation to end
28)       report();                                      // produce fault report
29)       ln.fault_report_full();
    }
```

Figure 3.5: A simple example program using DEPEND – the main co-routine.

```
30) void receiver(int port_id)
31) {     create("recv");          // this is a co-routine - not a subroutine
32)       int stat, finished = 0;
33)       while (!finished && ln.cond_ok()) {
34)             .... do computations ...
35)             .... put results into msg ...
                      // send( to, from, msg, size )
36)             ln.send((port_id+1)%NUM, port_id, &msg, sizeof(msg));
37)             int reply = ln.receive(id, msg[id], stat);
38)             if (stat) {
39)                   ... received message without error ...
40)                   ... do more computation ...
41)             } else {
42)                   ... message as error - take remedial action ...
43)             }
              }
44)       cnt--;
45)       if (cnt == 0)
46)             done.set();
    }
            /* optional fault routine to do specific fault injections
47) void my_fault_func(int &msg)
48) { .... code to corrupt fields in the message ... } */
```

Figure 3.6: A simple example program using DEPEND - the co-routine.

injected into the ports and links because a fault type has not been specified. If transient faults or user defined faults are injected, repair times, coverage and user defined fault subroutines for link and port faults can be specified.

The main co-routine creates two of the **receiver()** co-routine (line 26) and then goes to sleep waiting for the event **done** to be set before printing the performance (e.g., throughput, response time etc.) and fault (e.g., MTTF, availability, fault and repair times) reports.

The general structure of the **receiver()** co-routines is shown in lines 30 through 46. Each co-routine performs some computation, puts results in a message and then sends it to the tandem co-routine. Each co-routine then waits for a return message, performs additional computations and repeats the entire process until all computations are complete or until the communication link fails (i.e., both links are failed or 1 port has failed). The last co-routine to complete wakes up the main co-routine by setting the **done** event.

The detail of the simulation model depends on the user. For the computations (line 34, 35, etc.), the user may choose to simply forward the simulation clock and the data in the messages may be fake, meaningless values, or the user may execute actual code and send real data back and forth, making it possible to test actual programs such as algorithm-based fault-tolerant

programs, communication protocols and the like. Such an approach was used to determine the fault sensitivity and failure behavior of two load balancing heuristics operating in a distributed system [20].

The simple example in Figures 3.5 and 3.6 illustrates several basic features of DEPEND. DEPEND provides the framework and the general behavior of the fault-tolerant communication link. The specific behavior is controlled by the user by providing key parameters, specifying options and by furnishing fault subroutines. The user is relieved from simulating the operations of the simulation medium, the ports, the checksumming logic, the automatic retry logic and the injection of faults to the links, ports and messages. Several methods are available to interact with the class and react to important events such as link and port failures. The example shows two methods: a *polling function* that returns the status of the link (**cond_ok()**, line 33) and a *status variable* returned by the class (**stat**, line 37). In addition to various other polling functions such as functions to determine the status of the ports, the number of messages that have been corrupted and the overhead of sending acknowledgments, there are *signals* that are set any time failures and repairs occur. Co-routines can sleep until these signals are set and then wakeup and perform specific tasks.

The example also demonstrates that the impact of different number of spares, different fault arrival rates and distributions and repair coverage can be easily modeled by simply changing a few parameters. A larger, common-bus system consisting of several ports and **receiver()** routines operating on these ports can be quickly modeled by increasing the number of ports (line 2) and appropriately increasing **NUM** (line 1). Distributed algorithms can be tested for correctness and to determine their reaction to faults in the communication medium. Other architectures, such as the hypercube, can also be modeled by creating several instances of the link class, one for each link in the cube. By modifying the link declaration, a comparative performability study can be conducted to determine whether the *cold spareing* or the *graceful degradation* spareing mode is best suited for a particular application. .

3.3 Comparison

Table 3.3 compares the features of several simulation-based system level dependability analysis tools. Rainbow Net and Metasan are purely probabilistic tools. NEST and REACT allow both

Tool	Simulation		Execute Code	Software Model	Process Faults	Data Faults	Simulation Acceleration
	Functional	Probabilistic					
NEST	√	√	√				
RAINBOW NET		√					
METASAN		√					
REACT	√	√					
DEPEND	√	√	√	√	√	√	√

Table 3.3: Comparison of the features of several simulation tools.

functional and probabilistic simulation, and NEST can execute actual routing algorithms. However, neither tool provides facilities to inject process or data faults. Only DEPEND provides an environment to represetn application software at an abstract level and use it to perform application specific system analysis in the early design phase when actual code does not exist. Furthermore none of the tools provide acceleration techniques to reduce simulation time explosion.

Chapter 4

Tool Capabilities

This goal of this chapter is to answer the following questions with two examples. Given the large number of analytical tools and petri-net based simulation tools, what is the need for functional (behavioral) simulation tools for system level dependability analysis? What additional information and capabilities can they provide over analytical tools and fault-injection environments?

4.1 Behavioral Modeling

Analytical and Petri-net tools only use probabilistic modeling to represent system behavior. In essence, these tools pre-define a system's fault behavior with probabilities and distributions. Because DEPEND also allows behavioral modeling, the fault behavior of a system does not have to be pre-defined. Rather, with DEPEND, the fault behavior can be determined from the simulated system. This section presents a study conducted with DEPEND to illustrate this claim.

A distributed system using a centralized, prediction-based load balancing heuristic is evaluated under various types of faults. The heuristic has been shown to be very effective in reducing the mean response times of the processes under normal operating conditions [26, 20]. The question is how will it perform if the system is subjected to faults? Will the load balancing heuristic make the system less robust? Here the measure of robustness is the mean response time seen by the processes. To answer these questions, DEPEND is used to simulate a distributed system.

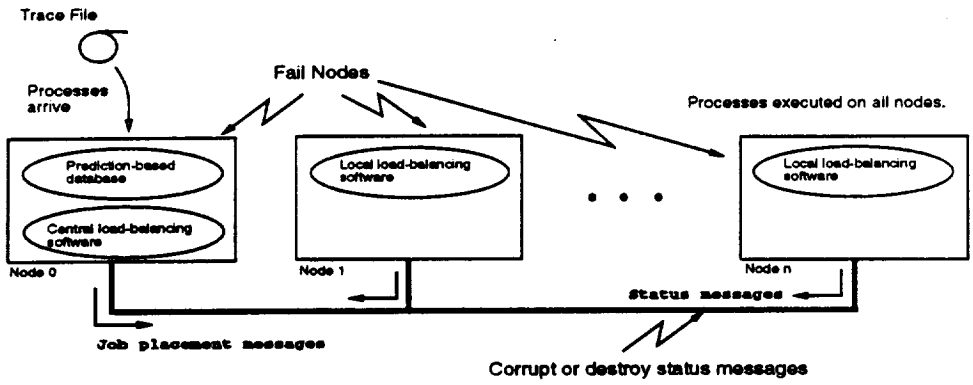


Figure 4.1: Distributed system executing the load balancing heuristic.

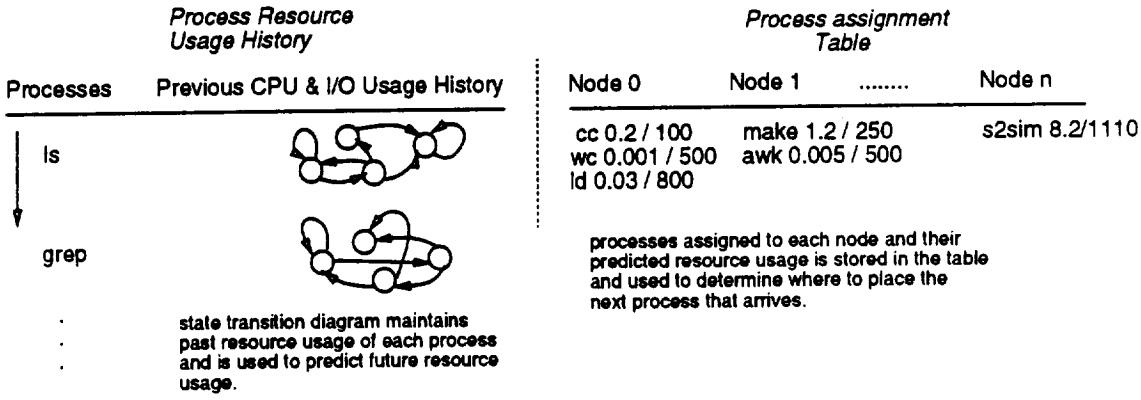
Because actual code can be executed within DEPEND, the complete load balancing heuristic, written in C++, is executed upon the simulated system.

The distributed system was simulated using one *link* class to represent the communication channel and several *servers* to represent the nodes in the system (Figure 4.1). Processes arrive at node 0 where the load balancing heuristic assigns it to a node estimated to provide the lowest response time. The assignment decision is based on the resources the process is *predicted* to use and the current assignment of processes to the nodes. This information is maintained in a database in node 0 (Figure 4.2a). When a process completes, its actual resources used is sent, via a status message, to node 0 where it is used to update the process's resource usage history and delete its entry from the process assignment table (Figure 4.2b). Each process is represented by its CPU and I/O requirements. A trace file obtained from measuring the CPU and I/O used by processes executing on a VAX 11/780 (running 4.3Bsd UNIX) over a period of one week is the source of the processes. Details about the heuristic can be found in [20].

The load balancing heuristic is started and processes are fed into the system. Meanwhile, the default fault models in the *link* and *server* classes are used to corrupt fields in the status messages, destroy status messages and fail node 0 and erase its database. These faults corrupt or destroy the database and ultimately impair the assignment decisions made by the load balancing heuristic. The mean response times of the processes are measured to determine the impact of the faults.

If a purely probabilistic modeling tool were used for this study, the user would have to pre-specify:

a. Scheduling Node's Database



b. Status Message

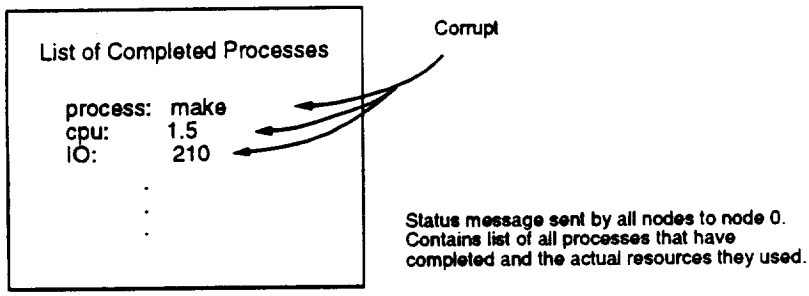


Figure 4.2: Database maintained by Node 0 and the status messages it receives.

- the probability that a fault will corrupt the database.
- how each fault will corrupt the database.
- which portions of the database will be corrupted.
- quantify the extent of corruption.
- quantify how each corruption will impair the placement decision made by the load-balancing software

Needless to say, these factors are extremely difficult to obtain without a thorough *prior* fault injection study and they pre-define the fault behavior of the system. Because DEPEND executes the actual software, these parameters are the *results of* (and not inputs to) the fault-injection experiment. Only the fault arrival rates and the types of faults injected need to be specified. Thus, DEPEND can identify the failure mechanisms, obtain failure probabilities, and quantize the effect of faults. It can be used to select the key features that must be modeled and help to determine and specify both the structure of, and the parameters to analytical models.

Only a summary of the results of the fault injection experiments are presented in this section. Detailed results can be found in [21]. The fault injection experiments show that unless errors are injected into the node 0 in rapid, sustained bursts the impact on response time is minimal. This finding indicates that the database is regenerated quickly after each fault and there is no need to save a backup copy. The impact on response time when status messages are corrupted is shown in Figure 4.3. Interestingly, as the percent of messages corrupted increases beyond a certain point (about 10%), the increase in response time reaches a plateau. Once more than 10% of the messages are injected, the database becomes so corrupted that the assignments are made randomly and further corruption has little impact. This behavior shows that the heuristic is robust and there is no need for additional, expensive fault-tolerant mechanisms.

A single distinguishing advantage of behavioral modeling over probabilistic modeling is accentuated in the results obtained when status messages are destroyed. Figure 4.4 (the curve labeled 'before') shows that the degradation in response time continues to increase without reaching a plateau. Upon close examination, the increasing poor performance was found to be caused by an implementation detail of the load balancing heuristic. Once the heuristic was modified, the erratic increase in the response time ceased. Thus by using behavioral modeling, it

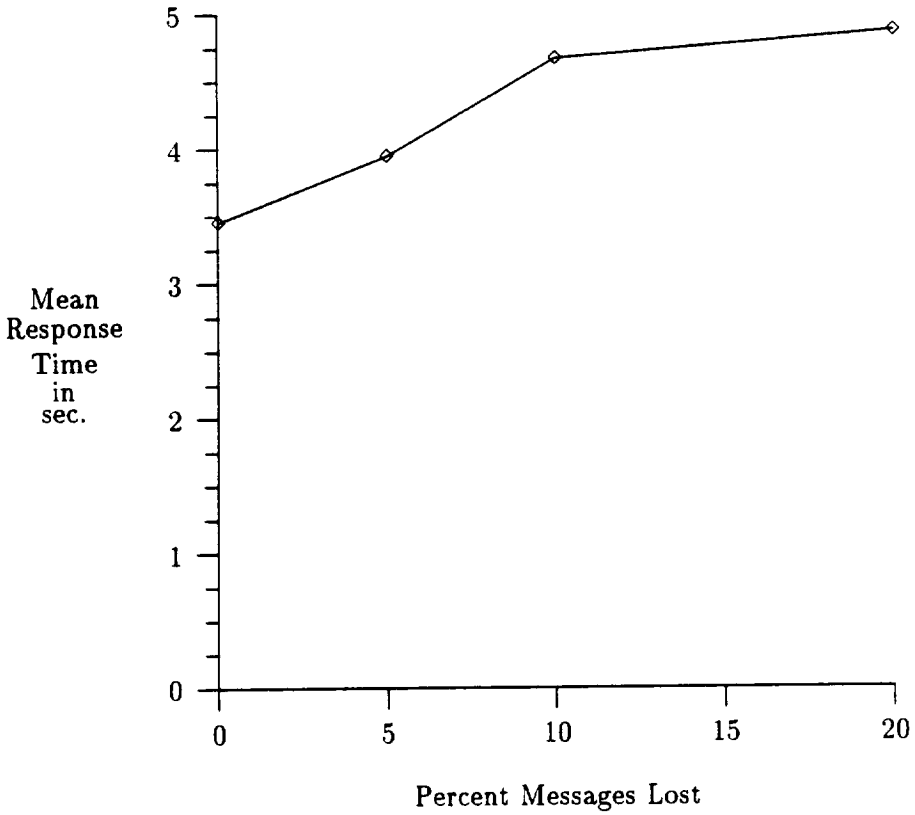


Figure 4.3: Impact of corrupted status update messages.

was possible to not only quantize the impact of the various faults, determine whether additional fault-tolerance mechanisms are needed if the system is operated in an adverse environment. it was also possible to verify the heuristic and identify a design feature that makes it susceptible to a specific fault type. Not only are such studies beyond the range of analytical tools. to the authors' knowledge, this experiment would be difficult to conduct with software fault injection tools like FERRARI [36] and FIAT [18] because they cannot evaluate software running on distributed systems.

4.2 Modeling Real Fault Scenarios

DEPEND uses a combination of behavioral and probabilistic modeling to simulate many realistic fault scenarios. In this section, a simple example is used to illustrate how DEPEND models latent errors that can substantially degrade system reliability. Latent errors can remain undetected in a system for long periods of time and are a potential hazard [8]. A measurement

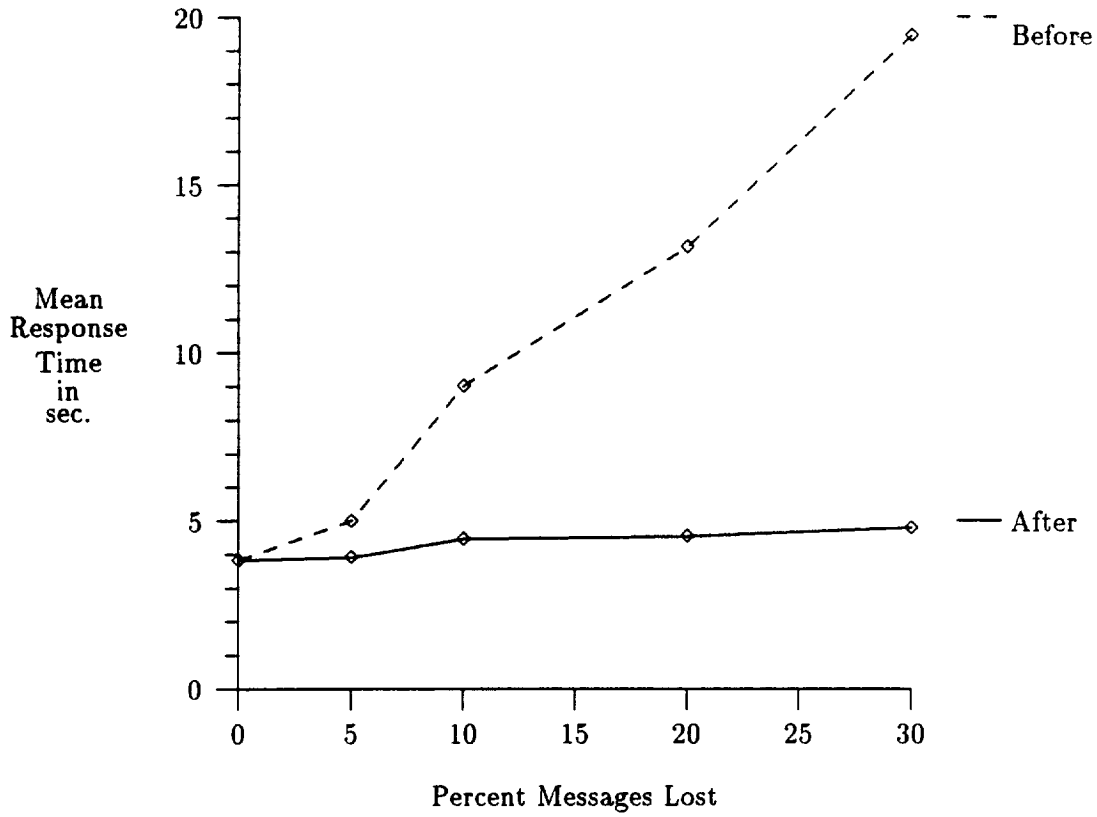


Figure 4.4: Impact of destroyed status update messages – before and after design change.

study of a VAX 11/780 [9] has shown the mean latency of an error can be in the order of minutes ($\mu = 44min.$, $\sigma = 29min.$) during peak hours and to several hours ($\mu = 8hrs.$, $\sigma = 4hrs.$) during off-times.

Modeling latent errors is difficult with Markov models for several reasons. First, the state space of the Markov model can be large for even small systems if each latent error and its location within a component is represented. Second, simplifying assumptions such as independent repair processes must be eliminated in order to accurately evaluate the impact of latent errors. For example, in a self-repairing system like the Tandem Integrity S2 (detailed description can be found in Chapter 5), the healthy processors reconfigure or repair a failed processor. If latent errors are detected in the healthy processors during a repair, the system fails. Modeling this *inter-component dependence* typically requires that the entire CTMC, its failure and repair process, be evaluated together thus potentially leading to large, stiff models. In [16], the authors present a novel decomposition technique to avoid such large, stiff models. With their approach, the repair coverage is evaluated in isolation and then 'adjusted' to account for the

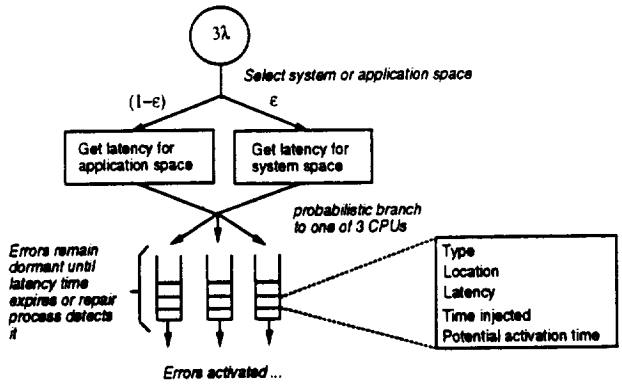


Figure 4.5: The error injection process that models error latency.

probability of a second, independent error in another component. The example below extends this analysis to also evaluate the impact of near-coincident errors due to latent errors in a repairing CPU. To do this, the example models the inherent dependencies that exist among the system components.

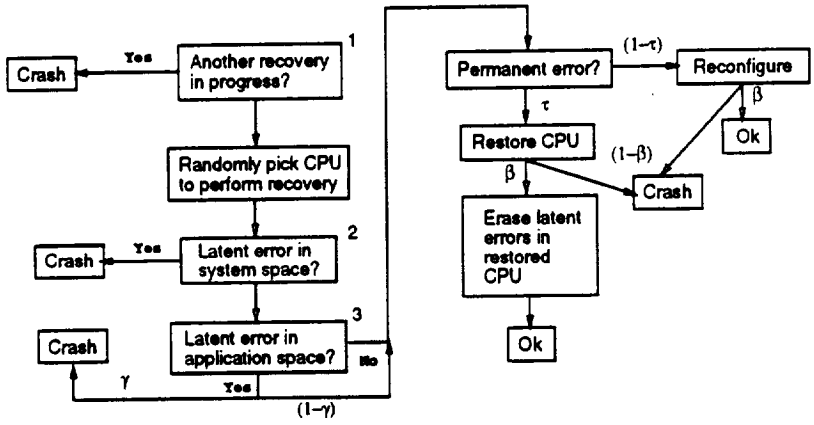


Figure 4.6: The repair process that considers the state of the other CPUs in the system.

Figure 4.5 illustrates how latent errors are injected into a system with three processors. DEPEND uses a chronologically sorted queue to maintain the latent errors injected into the system. Among the information associated with each latent error include the time at which the error is injected, its location (the component and memory address), and its latency period. Typically, the errors are detected when their latency period expires. However, the errors can be detected earlier by the repair process. Fault injection experiments of the Tandem Integrity S2 have shown that latent errors in the repairing processor are detected with high probability

Description	Value assigned
Error Arrival Rate, λ_1	0.01388
Repair Rate, μ	30.0
Percent Transient error, τ	0.99
Repair coverage, β	0.98
Latent error in system space, ϵ	0.05
Latency for system space, exponential with mean	15.0
Latency for application space, exponential with means	1, 2, 4, 6 & 8
Prob. Failure due to latent errors in repairing CPU, γ	0.1

Table 4.1: The parameters used. All times are in hours.

during repair and reconfiguration because much of the system is exercised during a repair. Figure 4.6 is a flowchart of the repair process that models this phenomenon. It is invoked each time a latent error is activated. Note that the repair process models near-coincident errors caused by a second, independent error in another processor (box 1). The dynamic determination of the repair coverage which depends on whether there are latent errors in the repairing processor is shown in boxes 2 and 3. To reiterate, by storing latent errors in a queue, DEPEND can dynamically model the *actual* activation time of latent errors which is *dependent* on: 1) the component the error is located in, 2) the location within a component, 3) the failure rate of the components in the system, and 4) the system's repair scheme.

The example is evaluated under three different conditions. First, permanent and transient errors with no latency are injected (M1). Second, near-coincident errors due to a second, independent error in another processor is modeled but error latency is not considered (M2). The decomposition technique in [16] model this second condition. Third, the simulation model described above is evaluated. It includes the conditions in M2, and it also considers near-coincident errors caused by latent errors (M3). The specific parameters of the models are listed in Table 4.1. Figure 4.7 shows a 33% decrease in MTTF when latent errors and their impact on system reliability is taken into account. The result emphasizes the importance of modeling real phenomenon such as latency, and it demonstrates that by so doing, more precise evaluation of fault tolerant mechanisms and repair schemes are possible. Later, in Chapter 7, the "staggered machine failure" phenomenon found to be caused by correlated errors [73] is mimicked by injecting correlated errors with different latency times. This more faithful model

is found to produce MTTF figures that are an order of magnitude larger than those produced with traditional models that assume correlated errors are detected simultaneously.

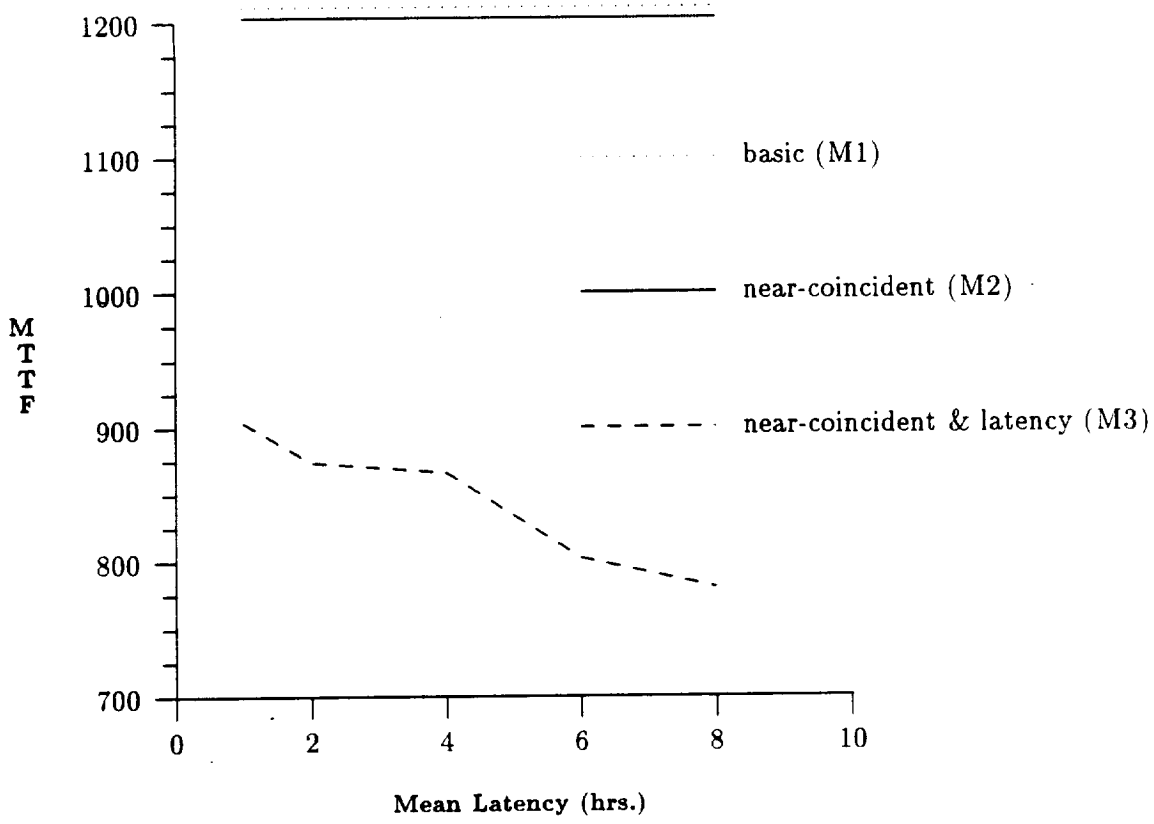


Figure 4.7: Comparison of system MTTF for the three models.

4.3 Discussion

This Chapter uses two examples to highlight the benefits of using DEPEND during the intermediate stages of system design. With analytical tools and petri-net tools a system's fault behavior must be pre-defined. DEPEND's integrated fault injection and functional simulation environment allows detailed modeling of system behavior and fault injections that help *determine* the system's fault behavior. The first example demonstrates how software behavior under hardware faults can be studied and how DEPEND can also be used to verify the software and identify design features that make it susceptible to particular fault types. The second example illustrates how accurate modeling of realistic fault scenarios and the ability to consider the inter-component dependencies provide significantly different MTBF measures than those ob-

tained when these issues are not considered. The ability to perform these sorts of studies in the early design stages is crucial to ensuring that a system meets its dependability specifications.

Chapter 5

A Case Study

This chapter describes the Tandem Integrity S2 fault-tolerant system. It also presents the simulation model of an S2 like system developed with DEPEND. The primary reason for selecting the Tandem Integrity S2 as the target system is that we have a hybrid injection and measurement environment [75] for an actual Tandem Integrity S2 machine. With this hybrid environment, injection experiments on the actual machine are conducted and used to validate the simulation methods and the models developed. A brief description of the hybrid environment and the setup used to conduct the validation experiments can be found in chapter 6.

5.1 The Tandem Integrity S2 Architecture

The Tandem Integrity S2 fault-tolerant system [34] is shown in figure 5.1. It is a triple modular redundant system (TMR). Each CPU is a MIPS R3000 RISC processor with an on-chip virtual memory mechanism and a separate clock. The processors execute the same instruction stream simultaneously. The processors are synchronized and their requests are checked by the voter when global memory is accessed, I/O is performed, or 2047 cycles have elapsed. If there is a discrepancy during voting, the processor in disagreement is shut down. The faulty processor performs a power-on self-test (POST), and if successful, the system is halted and the contents of the good processors are copied to it. The POST takes approximately 70 seconds and the re-integration takes approximately 2.0 seconds for a system with 8 Mbytes of local memory.

The local memories in the Integrity S2 do not have parity or ECC circuitry. The Integrity S2 relies on memory scrubbing to correct transient memory errors. The TMRC contains the

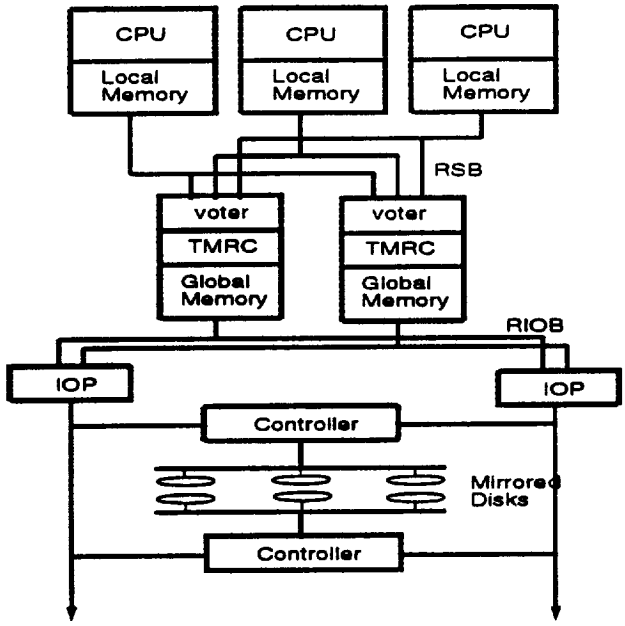


Figure 5.1: The Tandem Integrity S2 processing subsystem.

Percent of Time CPU is Idle	Re-integration Time
99%	30 sec.
59%	2 min. 25 sec.
37%	3 min. 46 sec.
27%	4 min. and 5 sec.
16%	4 min. and 40 sec.
0%	5 min. and 29 sec.

Table 5.1: Measured global memory re-integration times with varying machine idle percentages

voter and up to 128MB of global memory. The primary function of the TMRC is to vote upon the transactions sent by the CPUs. The global memories are protected by parity. When a parity error is detected by the TMRC, the backup memory takes over. A global memory is re-integrated in the background, interleaved with ordinary processing. The re-integration time is load dependent. Table 5.1 contains the measured re-integration times from a system with 32MB of global memory. Global memory re-integration has lower priority than CPU re-integration and is aborted and restarted in case of a CPU re-integration.

5.2 Simulation Model

To develop an accurate simulation model of a Tandem Integrity S2 like system, several key characteristics of the Tandem Integrity S2 system are simulated. These include:

1. the loose synchronization policy of the Integrity system. The fact that the processors idle at the voters to synchronize and the exact time needed by the voting operation are accounted for in the model.

2. the CPU (with its local memory) and the global memory structure that is unique to the Integrity system.

3. the functional behavior of the error-detection mechanisms of the CPU and global memory structure.

4. the CPU off-line POST and the on-line re-integration process and the global memory background re-integration process that are unique to the Tandem Integrity.

5. the behavior of the Tandem Integrity when a CPU and global memory re-integration occurs simultaneously (prioritized re-integration).

These details of the system architecture and how it reacts to faults were determined by studying its layouts, descriptions and manuals, discussing the matter with its designers and conducting several fault injection studies (in addition to the validation experiments mentioned below). Simultaneous injections into various components of the system helped to uncover interesting characteristics of the system that were subsequently incorporated into the Integrity S2 simulation model.

The simulation model of the Tandem Integrity S2, developed with DEPEND, is shown in figure 5.2. The blocks on the right are the DEPEND classes used in the simulation model. The block on the left summarizes the program written to create the simulation model and control the operations of each of the components.

The **NMR class** in the DEPEND library is the primary object used in the simulation. The NMR class simulates dual self-checking, triple-modular redundant and N-modular redundant systems. The servers idle until they receive a task to process. They then execute for a specified time period and feed their results to the voter. The voter waits for all the servers and then

executes a voting algorithm. A timeout condition is used to prevent hanging in cases where a server fails to report to the voter. The NMR class provides two voting algorithms: *bit stream* voting and *error based* voting. The *bit stream* voting scheme performs a bit by bit comparison of the data deposited by the servers. The *error based* algorithm flags a server's result as being faulty if an error has been injected into the server. This option was used in this simulation

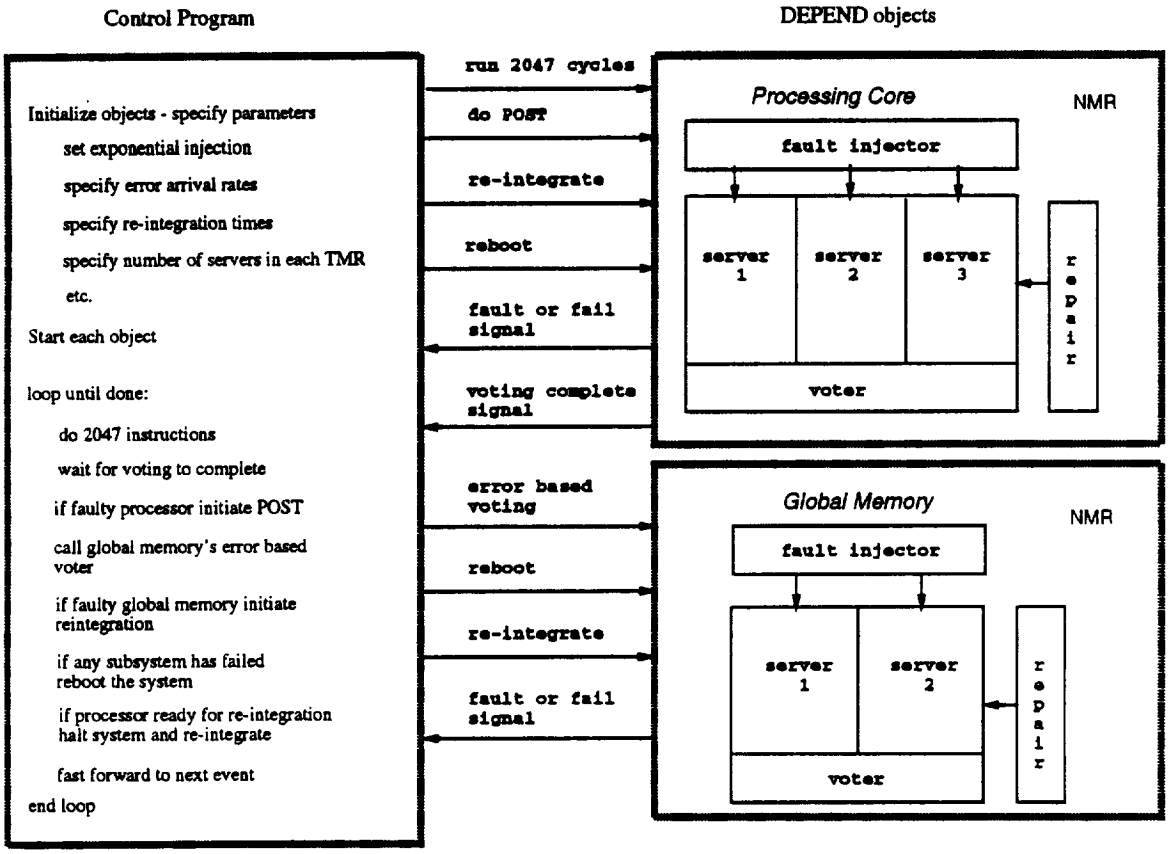


Figure 5.2: Simulation model of the Integrity S2 system developed with DEPEND.

because the processors were not given real data to process. The NMR shuts down the servers with faulty results. Automatic repair schemes are not provided, but functions can be called to repair the individual servers. This feature is used to simulate the automatic re-integration feature of the Tandem Integrity S2. The NMR's fault injector injects latent and correlated errors.

The **processing core** of the Integrity S2 is simulated with a NMR class containing 3 servers. Each server simulates a processor board containing a CPU and a local memory. The NMR's injector is used to inject errors into both the processor and the local memory.

The two **global memory** boards are simulated with a NMR with 2 servers. Every 2047 cycles, when the processors synchronize, the global memory's *error based* voting function is explicitly invoked by the control program to check each server and shut down any that has an active error. This simulates the actual operation of the Integrity S2 system because parity errors in the global memory are detected when the processors synchronize at the voter to access global memory.

The **control program** in figure 5.2 is the only part that is written by the user. It declares instances of the two DEPEND objects and initializes and customizes them. Initialization consists of specifying the error arrival distribution, the error arrival rate, the error latency distribution and so on. Then each object is "started" causing them to automatically perform tasks based on the parameters specified. All actions are automatically logged and the user can call functions to obtain a detailed report of every fault or repair. In addition, statistics such as availability, MTBF, the number of faults injected, the mean time between repair and the repair coverage are available.

The control program simulates the execution of 2047 instructions and the voting process. If any detectable errors are found in any of the components during voting, the components are shutdown. The status of the system is checked to determine whether it has failed (i.e. two CPUs or both global memory boards have failed). If the system has failed, it is rebooted and the simulation restarts. If the system has not failed, a background re-integration process is started for any component that was shutdown earlier by the voter. Finally, the control program checks to see if re-integrations initiated in an earlier cycle has completed. If so, they are handled based on the component type (CPU or global memory) as described in the previous section. Though not shown in figure 5.2, the memory scrubbing process is also simulated. Although the simulation can produce many results, the one we are most concerned with is the mean time between system failures (MTBF).

To summarize, this chapter describes the Tandem Integrity S2 architecture and presents the overall simulation model developed with DEPEND. The next two chapters describe specific aspects of the simulation model such as the approach used to model software behavior under hardware faults and the acceleration techniques used to execute the simulation and produce statistically valid results.

Chapter 6

Software Behavior Under Hardware Faults

The impact of software on system dependability is a primary concern because software is a major component of a system. One aspect is software reliability, which is concerned with design errors in the software. Another aspect, and the focus of this chapter is the behavior and the effect of software on hardware faults. In the early design stages of a system, a designer has a general idea of the structure and the underlying algorithm of the application software, or has access to the software that will be ported to the new system. An effective design strategy at this early stage is to evaluate the system while executing an abstract representation of the application software. This can provide key application dependent parameters that impact system dependability. It permits meaningful application specific evaluation and trade-off analysis of function and system level detection and recovery mechanisms.

In this chapter, we present a simulation-based software model that uses a *probabilistic control flow graph* to represent a program at an abstract level. The model is executed on a simulated system to obtain application specific error latency distributions and evaluate two functional error detection mechanisms. Error latency distributions obtained from the model are validated by comparing them with those obtained from running the actual programs on the Tandem Integrity S2 system. The impact of program control flow on error latency times is studied, a model of the detection process of a RISC-based system is developed and the need for application specific analysis of functional detection schemes is demonstrated.

Many studies have evaluated the behavior of software under hardware faults [1, 11, 14, 36, 44, 48, 18, 75]. Actual programs are executed on simulated or real processors in which errors are injected. These studies require an actual system or a detailed model of the hardware architecture. Tools like FIAT and FERRARI verify and study software systems by injecting software-implemented faults into actual programs. Laprie [39] uses a Markov model to evaluate the reliability of software systems during their operational life. The focus of the work is on software design errors. In [64], a stochastic model of error propagation is developed with a digraph that represents the interconnections between the hardware and software models. The propagation times are based on random variables with general distributions. In [51], a control flow graph automatically generated from syntactic analysis of a program, is used to estimate execution times. DEPEND [24] and REACT [12] provide a simulation-based fault injection environment to analyze the dependability of hardware architectures. As such, there are no mechanisms to study, at the early design phase, software behavior under hardware faults.

The software model presented runs within the DEPEND environment and extends DEPEND's ability to study software issues in the early design stage [25]. The high-level abstraction greatly reduces simulation time explosion and permits hundreds of error injections within seconds. Large observation periods are possible and allow collection of detection latency times that are in the order of minutes. Such large detection latencies are typical of real programs. Collecting them allows evaluation of detection schemes under realistic conditions. The contribution of this work is a method to:

1. Create a high-level abstract representation of application software that can be used to obtain application specific parameters that affect the dependability of a system.

2. Evaluate function and system level detection and recovery schemes within the context of application programs.

3. Study the impact of program control flow on one parameter, the error latency distribution.

4. Develop a model of the detection process of a RISC-based processor.

5. Provide feedback early in the design phase.

6.1 Model Overview

The software model environment is shown in Figure 6.1. The model consists of programs that are represented by probabilistic control flow graphs, G, and a virtual memory system, M, within which the programs execute. Contention for resources is modeled using the first-come-first-served or round-robin scheduling discipline. Errors are injected into the physical memory space.

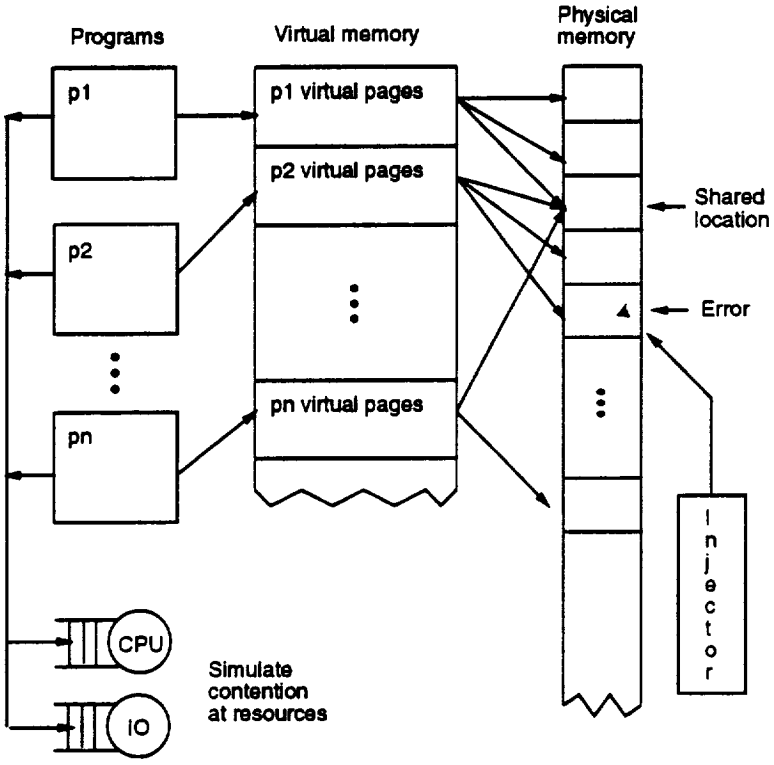


Figure 6.1: The complete software model execution environment.

The distinguishing feature of the software model is the virtual memory system M and the link between G and M. The virtual memory provides two functions. First, it permits simultaneous execution of several programs allowing evaluation of their combined impact. Second, it allows these programs to share pages in memory. Interprogram communication, and hence propagation of errors among programs, is modeled with shared pages. In fact, the shared memory primitive can be used to model various communication paradigms. In [42], the authors give a convincing argument for the claim that "variable sharing" is a close reflection of com-

puter hardware and it is a primitive upon which all other realizable distributed models can be described.

The error occurrence process is modeled by injecting errors into the physical memory and allowing the simulated execution of G, to discover the error. Manifestation of transient hardware faults are modeled as memory errors. A similar approach was taken in [44], in which the authors model all control flow errors as memory errors. The model is realistic for faults in the memory subsystem, however, it can also be applied to several types of faults in the processor. For instance, a fault in the ALU can propagate, eventually manifesting as incorrect data written to memory. Similarly, errors in the bus can cause incorrect data to be written to memory or written to the wrong location. Errors in memory mapped registers and I/O devices can also be represented by memory errors.

The user provides a control flow graph of an algorithm, the location of the memory accessed by each node and the total virtual memory used by the graph. The memory access pattern is derived from the graph. The model is hierarchical allowing designers to refine their abstract program models as more information is available. The software model simulates the execution of the graph on the underlying hardware. Outputs of the model include detection latency times, error propagation times, the probability of error propagation, and detection coverages. The software model is intended for use in the early design stages when an actual system does not exist and details of the applications are not known. It is not meant to replace fault injection studies of existing systems.

6.2 Model Description

The software model SW is defined as

$$SW = (M, (G, S, Ex)^+, I)^1$$

where:

- M is the virtual memory system.
- G is a *probabilistic control flow graph* (PCFG).

[1] The unary operator '+' means "one or more instances of."

- S is the subspace in M, allocated to G.
- Ex is the execution environment given by:

$$Ex = \begin{cases} Ex_{no\_error} & \text{Simulates execution of } G \text{ in a multiprogramming environment.} \\ Ex_{text\_error} & \text{Executes } G \text{ and simulates detection of errors in the text space of } G. \\ Ex_{text\_data} & \text{Executes } G \text{ and simulates detection of errors in the data space of } G. \\ Ex_{propagate} & \text{Executes } G \text{ and simulates detection and propagation of errors.} \end{cases}$$

- I is an injector which injects errors into M.

6.2.1 The Memory System

A paged virtual memory system is simulated and is defined as $M = (V_m, P_m)$, where, V_m is the virtual memory and P_m is the physical memory. A virtual memory system allows more than one G to execute simultaneously and share pages. The functions that operate on M are:

- **access_memory(PID,** v_{beg}, v_{end}) – Maps the virtual block $< v_{beg}, v_{end} >$ of a PCFG indicated by **PID** to a physical block $< p_{beg}, p_{end} >$ in P_m. The address of the first error encountered in the range $< p_{beg}, p_{end} >$ is returned. If there are no errors, -1 is returned.
- **inject_error(PID,** v_a) – The virtual address v_a is mapped to a physical address, p_a, in P_m. A bit is flipped and location p_a is marked as corrupted.
- **inject_error(**p_a) – A bit in the word addressed by p_a is flipped and p_a is marked as corrupted.
- **erase_error(PID,** v_a) – The virtual address v_a is mapped to a physical address, p_a, in P_m. The word is corrected and p_a is marked as uncorrupted.
- **erase_error(**p_a) – The word addressed by p_a is corrected and marked as uncorrupted.

Function **access_memory** is used by Ex to detect errors in the memory accessed by a PCFG. Functions **inject_error** and **erase_error** are used by the injector to introduce and correct errors in the physical memory, P_m.

6.2.2 The Probabilistic Control Flow Graph

G is a probabilistic control flow graph, $G = (N, E)$, where N is the set of nodes and E is the set of edges.

Definition 3.1: The node set $N = \{n_1, n_2, ..., n_k\}$ where n_i is an abstract definition of a program segment consisting of:

- Required:

name	name or identification of the node.
text space	location of node's text in virtual memory, denoted by $< \mathbf{tx}_{beg}, \mathbf{tx}_{end} >$, the address of the first and last words.
exec time	number of cycles to execute the node.
ρ_t	probability of detecting an error in the text space on the first encounter.
function	function performed by the block (**Read, Write, Read/Write, IO, Err_correct, Err_detect, Checkpoint**)

- Optional:

data space	location of data blocks processed by the node denoted by $(< \mathbf{dt}_{beg_1}, \mathbf{dt}_{end1} >, \ldots, < \mathbf{dt}_{beg_k}, \mathbf{dt}_{endk} >)$ where $< \mathbf{dt}_{beg_i}, \mathbf{dt}_{end_i} >$ is the address of the first and last words of data block i. Data can be located in shared memory space.
ρ_d	probability of detecting an error in data space.
ρ_{prop}	probability of propagating an error in data space.

The optional parameters are needed only if the Ex_{text_data} or the $Ex_{propagate}$ execution environment is used.

Definition 3.2: The edge set $E = \{e_1, e_2, \ldots, e_k\}$ where e_i is a directed edge defined as $e_i = ((n_i, n_j), \alpha_i, m_i)$:

- (n_i, n_j) is an ordered pair of any two nodes in N with n_i as the tail and n_j as the head of the directed edge,

- α_i is the probability of traversing the edge,
- m_i is a count of the number of times the edge is to be traversed.

Definition 3.3: A simple edge e_i is defined such that n_j is not an ancestor of n_i, $n_j \neq n_i$, $\alpha_i > 0$, and $m_i = 0$.

Definition 3.4: A loopback edge e_i is defined such that n_j is an ancestor of n_i or $n_j = n_i$, $\alpha_i = 0$, and $m_i > 0$.

Definition 3.5: The set $\xi_p^* \subseteq E$ is the set of directed edges from node n_p such that

$$\sum_{i=1} \alpha_i = 1, \quad \forall_i : e_i \in \xi_p^*$$

6.2.3 The Memory Subspace

S is the subspace of the total text, data and shared space in virtual memory allocated to G and is given by:

$$S = (< S_b(text), S_e(text) >, < S_b(data), S_e(data) >, < S_b(shared), S_e(shared) >)$$

where,

- $S_b(x)$ = address of the first word of space x in virtual memory.
- $S_e(x)$ = address of the last word of space x in virtual memory.

6.2.4 The Execution Environment

A discrete-event simulator [60, 43, 24] is used to execute G in a simulated multiprogramming environment. Ex can be one of the following paradigms.

6.2.4.1 Execution without error detection

Ex_{no_error} models the probabilistic execution of G and is the basis upon which other execution paradigms are built.

Input: cycle_time, IOserver_ID, CPUserver_ID

Function:

```
        Assign PID to PCFG
        Allocate text, data, and shared memory    space in M
        Traverse the PCFG
           I = top most node in PCFG
           while (I <> terminal node)
              texterror_addr = memory_access(PID, text space_I)
              if (data space_I specified)
                 dataerror_addr = memory_access(PID, data space_I)
S1:           server_time = exec time_I × cycle_time
              if (function_I = IO)
                 reserve(IO_server_ID)
                 hold(server_time)
                 release(IO_server_ID)
              else
                 reserve(CPU_server_ID)
                 hold(server_time)
                 release(CPU_server_ID)
S2:           I = get_next_node()
           end while
```

where,

`reserve()` & `release()` — simulate the queueing activity at a server.

`hold()` — increments the system clock.

```
get_next_node()
   if (ξ*_I = {e_l, e_s}, where e_l is a loopback edge and
      e_s is a simple edge) then
      if (m_l > 0) then
         decrement m_l
         node_addr = head(e_l)
      else
         node_addr = head(e_s)
```

else if ($\xi_I^* = \{e_1, e_2, ..., e_j\}$, and ξ_I^* contains no loopback edges) then

select $e_i \in \xi_I^*$ with probability α_i

node_addr = head(e_i)

return(node_addr)

6.2.4.2 Execution with detection of text errors

Ex_{text_error} is an extension of Ex_{no_error} which models the probabilistic detection of errors present in the text space. It requires three additional inputs:

Additional input: detect_function, record_error, ν

where,

`detect_function`	models the probabilistic detection process (see below).
`record_error`	records pertinent statistics, such as the time of detection. Execution can halt or continue after detection as specified by the user (default: continue).
ν	parameter used by the detection function, $0 \leq \nu \leq 1$.

Statements **S1** and **S2** of Ex_{no_error} are replaced with the following:

S1:

```
error_detected = detect_function(
                 texterror_addr, ν)
if (error_detected)
   server_time = (texterror_addr - tx_beg_I) / (tx_end_I - tx_beg_I + 1)
                 × exec time_I × cycle_time
else
   server_time = exec time_I × cycle_time
```

S2:

```
if(error_detected)
   record_error()
   use server for the remaining time
I = get_next_node()
```

In S1 above, the time to detection (server_time) is computed using a linear relationship based on the location of the error within the text space. The detection function returns a **FALSE** if the `text_error_addr` is -1. Otherwise it models error detection as follows:

```
detect_function:
    increment err_pass
    γ = U[0,1) -- a Uniform distribution
    if(ρ_t · ν^(err_pass-1) < γ AND err_pass < LIMIT)
        return TRUE
    else
        return FALSE
```

where,

`err_pass`	is the number of times the specific error is encountered.
`LIMIT`	is the maximum number of encounters within which the error must be detected.

The detection function decreases the probability of detection exponentially for $\nu < 1$, as the number of encounters increases. The total probability of detecting an error, D, within K encounters is:

$$D = \sum_{\kappa=1}^{K} D_\kappa \tag{6.1}$$

where,

$$D_\kappa = \begin{cases} \rho_t \nu^{\kappa-1}(1 - D_{\kappa-1}), & \kappa > 1 \\ \rho_t, & \kappa = 1 \end{cases}$$

The software model is designed to simulate and evaluate functional detection schemes. The detection function presented here models the low level detection mechanism found in most processors. Such mechanisms include mathematical exceptions (e.g. division by zero), bus errors caused by invalid address or access types, and address error caused when an unaligned word is fetched or stored. Later in this chapter, we show that the detection function accurately captures the behavior of the low level error detection process for the programs modeled.

6.2.4.3 Execution with detection of text and data errors

Ex_{text_data} is an extension of Ex_{text_error}. Identical service time calculations and detection functions are used to simulate the detection of errors in the **data space** of a node. If the **function** of the node is **Write**, then errors encountered in the **data space** are overwritten with probability ρ_d.

6.2.4.4 Execution with error propagation

$Ex_{propagate}$ is an extension of Ex_{text_data} which simulates the propagation of errors. If an error is not detected, then with probability ρ_{prop}, it is propagated based on the function of the node at which the error has occurred.

An error in the **text space** is assumed to cause a **Read** node to read the wrong data and a **Write** node to write data to the wrong location. A **Read** node with an error causes m (an input parameter) errors to be injected into the **data space**. This models the phenomenon that a wrong word was read and used for computation thus propagating the error. For a **Write** node, $2m$ errors are injected. One error is injected because the wrong location is overwritten, and another because the correct location is not updated[2].

An error in the **data space** of a node causes m errors to be injected into the **data space** if the node's function is a **Read**. If the function of the node is **Write**, the error in the **data space** is corrected.

If a node's **data space** has more than one data block, the block injected is selected randomly. The parameter m is typically 1. Since the **data space** of one node can overlap with that of another, errors can propagate between nodes and within the memory space of a PCFG. If a PCFG shares pages with other PCFGs, errors can propagate between them.

6.2.5 The Injector

The injector injects errors into the physical memory, P_m, using the functions described in Section 3.2.1. The injector can be specified to inject one or many errors. If many errors are to be injected, the user specifies one of the three error arrival distributions: constant, exponential

[2] Propagation for a **Read/Write** node combines the effect modeled for **Read** and **Write**. To keep the paper concise, a detailed description is not provided.

or Weibull. The range of locations to be injected must also be specified. The words injected

```
main()
    for i = 1 to 1000
        list[i] = 1000 - i
    end for

    input(k)

    if (k = ''sort'')
        call sort()
    else
        call print()
end main

sort()
    for i = 1 to 1000
        for j = i to 1000
            find min entry
        end for
        exchange list[min] with list[i]
    end for
end sort

print()
    for i = 1 to 1000
        output(list[i])
    end for
end print
```

a. Original program

```
bind { pt 0.8; pd 0.5; sz 1000; sz_half 500 }
memory{ data 0 1000; text 1001 1500 }
node main {
   node init    WRITE 19 * sz pt 1001 1018 data pd 0 1000
   node input   READ  10      pt 1019 1029
   node sort
   node print
   flow{
                init input
                input sort BRANCH 0.5
                input print BRANCH 0.5
   }
}
sort{
   node fori    READ  4       pt 1030 1034
   node forj    READ  4       pt 1035 1038
   node min     READ  10      pt 1039 1048 data pd 0 1000
   node endj    READ  2       pt 1048 1050
   node xchng   WRITE 12      pt 1051 1062 data pd 0 1000
   node endi    READ  2       pt 1063 1064
   flow{
                fori forj
                forj min
                min endj
                endj min LOOP sz_half xchng
                xchng forj LOOP sz endi
   }
}
print{ ... }
```

b. PCFG definition of program

Figure 6.2: Example program and corresponding PCFG.

are selected randomly from this range.

6.2.6 Specification Language

A simple program and the language used to specify a PCFG of the program are illustrated in Figure 6.2. The "`memory{}`" construct specifies the virtual space required by the text, data and shared segments. The "`bind{}`" construct binds constants to identifiers. The structure of the program is defined with the "`node{}`" construct. A node can be a simple node containing items listed in definition 3.1. It can also be a "meta" node consisting of one or more simple and meta nodes and one "`flow{}`" construct that defines the control flow within that meta node. In the figure, node `init`'s function is **Write**. Its **exec time** is 19×1000 cycles, its **text space** is $<1001, 1018>$ and it has one data block $<0, 1000>$.

The level of abstraction used will depend on the nature of the experiment and the information available about the program. As more information is known, it can be added by converting simple nodes to meta nodes. In this example, each node represents a set of program statements [3]. Some nodes are more "abstract" than others. For instance, node `init` represents the entire "for loop" used to initialize the array, whereas the loops in `sort()` are modeled in more detail. For this level of abstraction, a node's **text space** can be determined based on the number of instructions it represents. For a RISC processor, where the instructions are of fixed length, the size can be estimated based on the statement to instruction ratio. The number of cycles per node is 1 to 1.4 times the number of instructions. Once the PCFG is specified, it is compiled to create a binary file that is executed by *Ex*.

6.3 Model Application

One application of the software model is illustrated. It is used to obtain error latency distributions of two different programs and evaluate two memory scrubbing schemes. The error latency times obtained are validated with measured latencies from an actual system. The experiments also validate the detection function introduced in the previous section.

6.3.1 Application Programs

Two programs are represented with the software model. The first is a Gaussian elimination program with partial pivoting and back substitution. This program was chosen because it is representative of a large class of numerical and statistical applications. The Gaussian elimination program (**Gauss** executes indefinitely. Each iteration takes 35.6 seconds and consists of filling a 300-by-300 matrix with randomly generated numbers, solving the matrix and printing the result. The program contains 280 lines of C code and was modeled with a PCFG consisting of 23 nodes. The second program is an insertion sort (**Sort**). Unlike **Gauss**, it manipulates integers and does not use the floating point co-processor. **Sort** repeatedly sorts a randomly generated array of 7000 numbers which requires 30 seconds for each iteration. The program contains 200 lines of code and is represented with a PCFG with 11 nodes. Both PCFGs are

[3] Typically, the level of abstraction is much higher than shown here, but this serves well as a simple illustration.

executed with the Ex_{text_error} paradigm. Several instances of the **Gauss** PCFG are executed simultaneously to demonstrate the software model's ability to execute more than one PCFG.

6.3.2 Experiment Setup

The software model was initialized with P_m = 8MB of memory. One to 16 instances of a PCFG are executed simultaneously. A procedure that sweeps through P_m every hour was added to simulate the scrubber. Each experiment run consists of the following steps:

1. Start the PCFG(s).
2. Randomly select a PCFG and a word within the PCFG's virtual memory space to be injected.
3. Inject an error – flip a bit of the selected word.
4. Execute the PCFG(s) until the error is detected or the observation period has expired.
5. Record the detection time and goto step 2.

The error is assumed to be caused by a transient fault and is repaired by rewritting the corrupted word. There are two possible outcomes of an injection: 1) a PCFG detects the error, or 2) the scrubber detects the error. Several terms used throughout this section are now defined.

Definition 5.1: An *active error*, e_a, is an error that is detected by the program (i.e. detected by the process of program execution) assuming there is no scrubber.

Definition 5.2: The *program detection latency* is the time from injection to the time the error is detected by the program.

Definition 5.3: The *scrubber detection latency* is the time from injection to the time the error is corrected by the scrubber.

6.3.3 Validation Environment

A Tandem Integrity S2, instrumented with a fault injection and a monitoring device, was used to verify the detection times obtained from the software model. The original C version of the application programs were executed while injecting errors into their text space and measuring the detection times. Figure 6.3 shows the hybrid monitoring environment used to conduct the

experiments. A detailed description of the environment can be found in [75, 76]. The hybrid environment takes advantage of the Integrity S2's ability to re-integrate a failed component of a subsystem on-the-fly. The environment is automated and can execute for days repeatedly injecting errors and collecting measurements. A Tektronix DAS 9200 digital logic analyzer is used to monitor the bus activity on the CPU that is injected with errors. A *finite state machine* is used to specify the data that is to be collected, such as the times when an injection occurs, the corrupted location is read or written, an exception is raised, and when POST is initiated.

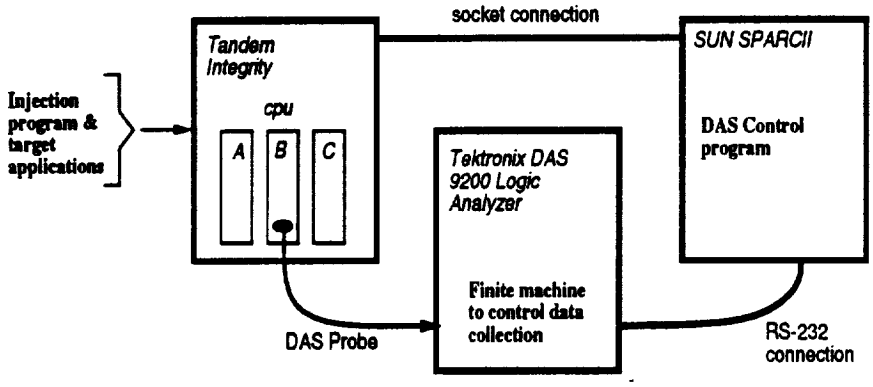


Figure 6.3: An injection environment using hybrid monitoring.

A *DAS control program* running on a Sun SparcII workstation accepts *start*, *stop* and *data upload* commands from the *injection program*, translates them, and relays them to the DAS. The *injection program* runs on the Tandem Integrity S2 machine and injects errors into the text region (the region containing the machine instructions) of a target application. Injecting an error consists of randomly selecting a word, and randomly corrupting one bit of the word residing in the memory of CPUB. If the word resides in the cache, it is deleted to ensure that the corrupted version of the word is used.

Figure 6.4 shows the program used to inject errors and collect data. Each error injected has one of three possible outcomes:

1. The error is detected by the MIPS R3000 (e.g. bus error, address error, arithmetic exception) on CPUB. In this case, CPUB sends an interrupt to the voter. The voter, not receiving interrupts from the other CPUs, shuts down CPUB.

2. The error is detected by the voter. This can happen in two ways. First, all CPUs access global memory, perform I/O or are forced to synchronize at the voter. At this point, the

```
1) Start one or more target application program(s).
2) Start the DAS controller and request it to start the DAS.
3) Randomly select
        the target application to inject
        the address of the word to be corrupted
        the mask to use.
4) Inform the DAS of the address of the word corrupted.
5) Inject the error into the word (flip a sigle bit).
6) Wait a specified time or until CPUB is shutdown.
        meanwhile, the DAS records reads,writes,exceptions, POST etc.
7) If CPUB was not shut down
        shut down CPUB.
8) Initiate the POST and re-integration process.
        this cleans out any effect of the injection.
9) Request the DAS controller to upload the data collected by the DAS.
10) Goto step 3.
```

Figure 6.4: Program used to inject errors and obtain measurements.

voter detects a discrepancy in the command or the data submited by CPUB. Second, the injection causes only CPUB to erroneously access global memory or perform I/O. The voter detects the discrepancy and shuts down CPUB.

3. The error is not detected by either detection scheme and is eventually corrected by the memory scrubber.

The behavior of the detection process and the measurements obtained should not be construed to be purely characteristic of the Tandem Integrity S2 because it uses the error detection mechanism of the MIPS R3000 – a commercially available RISC processor. However, because of the additional detection circuitry of the Integrity S2, the error detection coverage will be larger and the error latency times will be smaller.

6.3.4 Program Detection Latency

Experiments are initially conducted with the **Gauss** PCFG to determine if the high level abstraction can provide program detection latency times similar to those observed by the real program executing on the Integrity S2. Three different detection models, obtained by changing the parameters of the detection function (see Equation 6.1), are used. The specific parameters and the names of the detection models are listed in Table 6.1. Parameter ρ_t was selected so

Name	ρ_t	ν	LIMIT
SIM1	0.67	1.0	1
SIM2	0.425	1.0	2
EXP	0.589	1/3	2

Table 6.1: Parameters for the three low level error detection models.

that D (Equation 6.1) is 0.67 [4], and it is assumed to be the same for all nodes in the PCFG. SIM1 detects errors on the first encounter only. SIM2 detects errors on second encounters but it uses the same value of ρ_t on each encounter because ν is one. EXP reduces ρ_t exponentially with each encounter. There are two components to program detection latency: the *access time* and the *detection time*. The *access time* is the time from injection to the time the corrupted location is first accessed by the program. The *detection time* is the time from the first access of the corrupted location to the time the error is detected. In the model, the time to traverse the PCFG until the corrupted location is accessed is the access times. The detection model is used to capture the detection time and the probability of detection. Since the access times are orders of magnitude greater than the detection times, errors that are detected immediately after being accessed are assumed to be detected in zero time. Detection models SIM2 and EXP are designed to capture the cases where the detection times are large.

One thousand errors are injected into an executing PCFG to obtain program detection latency times. Figure 6.5 shows the cumulative distribution functions (CDF) of the measured and simulated program detection latency. The figure also contains the CDFs obtained from using two distributions that estimate the spatial distribution of the memory accesses [47]. One is the uniform distribution where all addresses are accessed with equal probability. The other is a ramp distribution where the probability of accessing an address increases as the order of the address increases. That is, high-order addresses are more likely to be accessed than low-order addresses. The graph shows that the uniform and ramp functions, which do not model the control flow of the program, fail to capture the spatial locality exhibited by the program. In the measured CDF over half the detection times are less than a second whereas the 50th percentile

[4] $D = 0.67$ was determined from measurements, however, later it is shown that D (which is a function of ρ_t) is not the dominant factor in determining latency times.

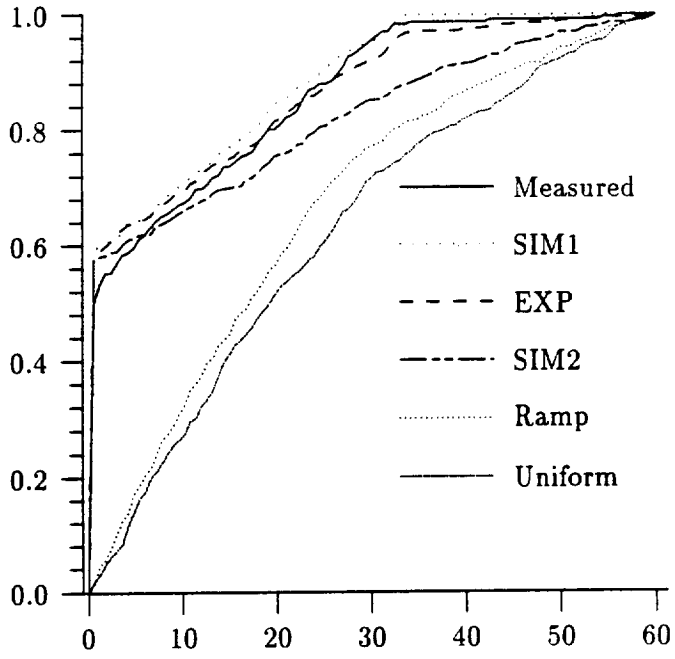


Figure 6.5: Cumulative detection latency distribution functions.

for Ramp and Uniform is approximately 18 seconds. The three functions which model the program's control flow captures the program detection latency distribution remarkably well.

Figure 6.6b is a histogram of the measured error latency times for **Gauss**. The spike near the origin results because of the structure of the program and because most errors, when encountered, are detected within milliseconds. Most of the time in **Gauss** is spent in loops where a set of instructions are executed repeatedly. Errors injected in the body of a loop containing the program counter (PC) will normally have small access times. These errors account for more that 50% of all the errors. Errors injected outside a loop containing the PC will have access times that are at least as large as the time needed for one iteration of the loop (Figure 6.7). The actual access time will depend on the number of times the loop will be executed. These errors account for much of the tail in Figure 6.6b.

A close scrutiny of the measured data shows that many errors are not detected immediately after they are accessed. Figure 6.8 is a probability distribution function of the detection time for **Gauss**. Though a large percent are detected within a few microseconds, a good segment (10%) have detection times that are greater than 100 milliseconds. Furthermore, Figure 6.9 shows the frequency count of the number of multiple read accesses of the corrupted instructions

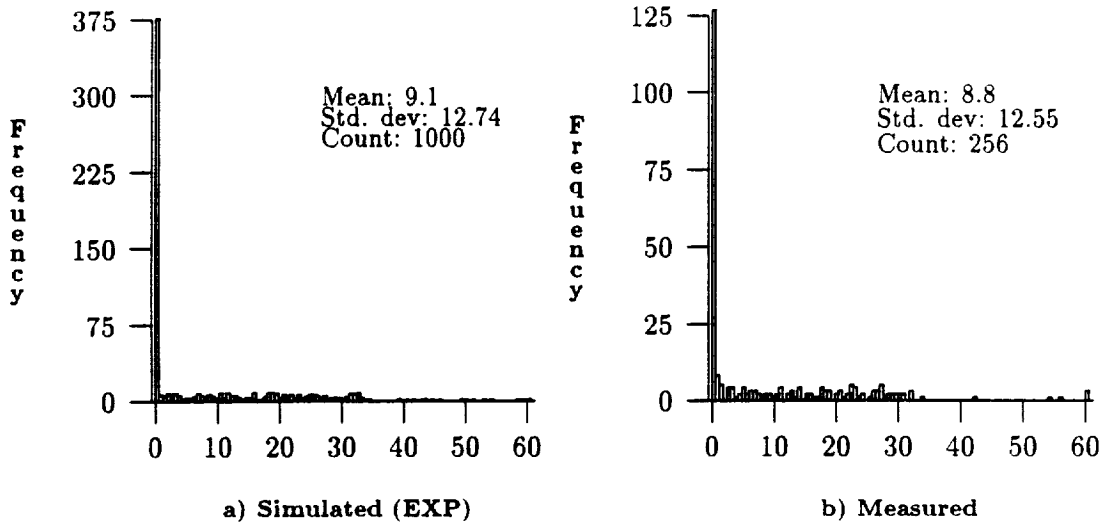


Figure 6.6: Histogram of the program detection latency in seconds, for 1 program.

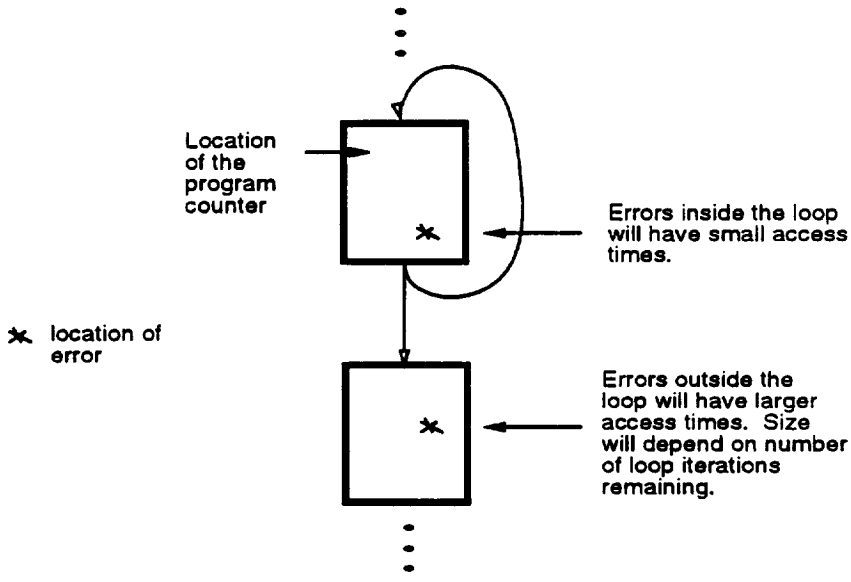


Figure 6.7: Access time depends on location of the PC and the error.

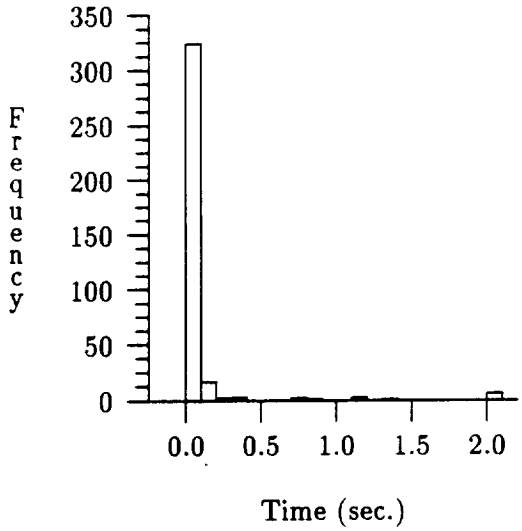


Figure 6.8: Histogram of the measured detection times for **Gauss**.

prior to detection. The cause for this behavior is probably due to data dependencies. For instance, if the instruction **bgt r5, 0x4000** is corrupted and becomes **bgt r7, 0x4000**, the error may not be detected if the contents of register r7 is greater than zero. On a later access, the value of r7 may be less than zero, thus potentially causing an error. The other reason is that the error propagates and is eventually detected while executing some other instruction. The software model does not directly model this later phenomenon. However, it models the multiple read phenomenon with error detection models which detect errors on a second, third or more access. The time to re-access the corrupted location is determined by traversing the PCFG and is the detection time. Note that the probability of detection tends to reduce exponentially with increasing number of read accesses (Figure 6.9). Of the detection models, only EXP mimics this behavior. SIM1 detects errors only on the first access. SIM2 overestimates the number of errors detected because it does not reduce the probability of detection with increasing accesses. The value of $\nu = 0.33$ for EXP was obtained experimentally. Repeated trials with various values were tried and a value of 0.33 was found to produce the most accurate representation of the measured behavior. Comparison of Figure 6.6a and 6.6b, shows that EXP captures the error latency distribution very well. It even captures the tail of the latency distribution. Results of a statistical verification of the detection models are shown in Table 6.2. The table lists the mean and standard deviation of the error latency distribution and the sum of the square of the

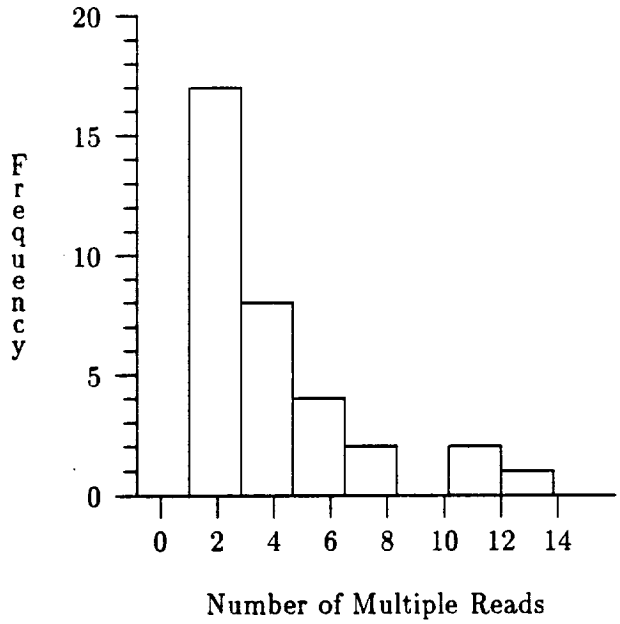


Figure 6.9: Histogram of multiple reads of corrupted locations.

residuals, R,

$$R = \sum_i (y_i - \hat{y}_i)^2$$

where, y_i is the ith entry from the measured CDF

$\hat{y}_i$ is the ith entry from the simulated CDF

for all detection models and for the Ramp and Uniform methods. Of all the models, EXP produces R values that are 3 to 350 times smaller.

Name	Mean	Std.	R	Normalized R w.r.t. EXP
SIM1	6.9	10.2	0.066	3.00
SIM2	11.7	17.36	0.345	15.68
EXP	9.1	12.74	0.022	1.0
MEASURED	8.8	12.55	-	-
Uniform	37.64	34.68	7.58	344.55
Ramp	37.13	39.73	5.68	258.18

Table 6.2: Statistics of the measured and simulated program detection latency for the Gaussian elimination program.

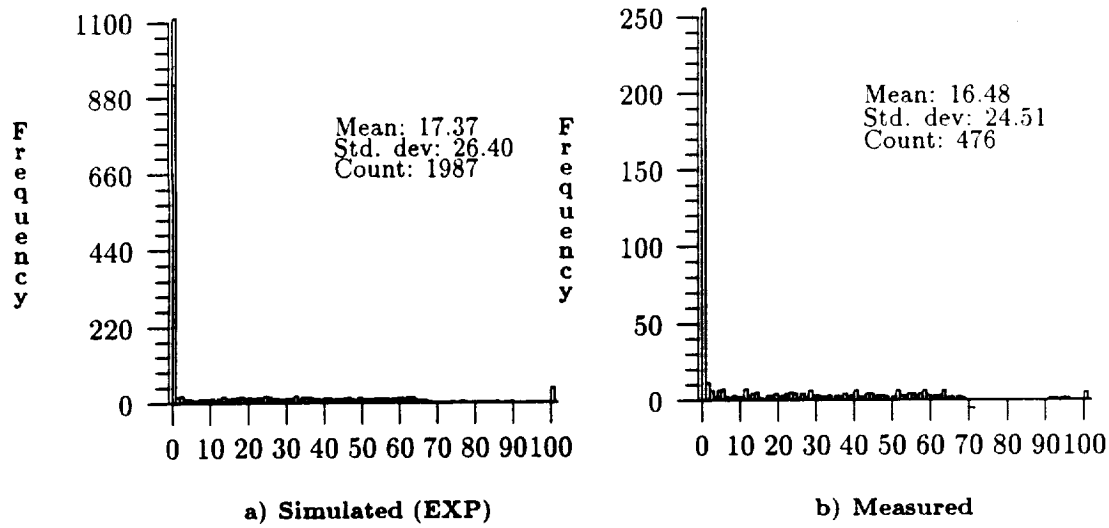


Figure 6.10: Histogram of the program detection latency of two programs (sec.).

The software model was used to execute two PCFGs to obtain their combined detection latency times. Figure 6.10 contains the measured and simulated latency times of two copies of the Gaussian elimination program executing simultaneously. The two means are similar and demonstrates that the model can predict the change in latency times due to the increased contention at the CPU.

6.3.4.1 Applicability of the Model

The previous subsection demonstrates that the software model can be used to determine the program detection latency time for the Gaussian elimination program. The question is can this same model be applied to different programs? One issue is the parameter ρ_t. It varies with the system and is very dependent on the structure of the program. For example, programs with many branches that are seldom traversed will tend to have a smaller value for ρ_t. Though the PCFG of a program can be used to estimate ρ_t, it is useful to determine the sensitivity of the results of the model with respect to ρ_t. If the model is highly sensitive to the estimated ρ_t, it will not be generally useful. Another issue is the detection model. It is not clear that the same detection model will hold for all program types.

To determine the sensitivity of the results to ρ_t, the Gaussian elimination PCFG was executed with various values of ρ_t. Table 6.3 contains the mean and standard deviation of the program's detection latency and the R-values for varying values of ρ_t. The EXP detection

ρ_t	Mean	Std. Dev.	R
0.2	12.42	21.7	0.071
0.4	11.40	19.6	0.047
0.6	10.02	16.79	0.021
0.8	8.83	14.14	0.014
1.0	7.41	10.4	0.018

Table 6.3: Sensitivity of the mean detection latency time to varying values of ρ_t (LIMIT = 5).

Name	ρ_t	ν	LIMIT
SIM1	0.60	1.0	1
SIM2	0.26	1.0	3
EXP	0.42	1/3	3

Table 6.4: Parameters for the low level error detection models.

model with LIMIT set to 5 was used. Very large values for LIMIT tend to skew the mean and produce the worst case results. The table shows that the relationship between ρ_t and the mean latency is linear. The R-value also displays an inverse linear relationship and indicates that the distribution generated by the model fits the measured distribution well. For all values of ρ_t except 0.2, the R-value is less than that produced with SIM1 and SIM2 (Table 6.2).

To determine whether the detection model developed for **Gauss** can be applied to other programs, it is used to model an insertion sort program. The sort program was selected because it operates on integers, does not use the floating point co-processor, and hence is quite different from **Gauss**. An arbitrary value of $D = 0.6$ is used for all the nodes in the sort's PCFG. For comparison purposes, detection models SIM1 and SIM2 are also used. The parameters for all three detection models are listed in Table 6.4. Table 6.5 lists the mean and standard deviation of the error latency distribution and the R values for each detection model, and Figure 6.11 contains the measured and simulated cumulative distribution functions. Again, EXP produced relatively more accurate results both with respect to the mean and the overall latency distribution. But more importantly, the Table and the Figure indicate that the model can be generally applied to other programs. Figure 6.12, which shows the measured frequency of multiple reads before detection, explains why EXP works well with sort. **Sort** displays a behavior similar to that seen with **Gauss**. Based on only two programs, one cannot state as

Name	Mean	Std.	R	Normalized R w.r.t. EXP
SIM1	9.6	11.72	0.073	1.20
SIM2	11.1	17.12	0.128	2.13
EXP	10.42	14.05	0.061	1.0
MEASURED	10.67	11.93	-	-

Table 6.5: Statistics of the measured and simulated program detection latency for **Sort**.

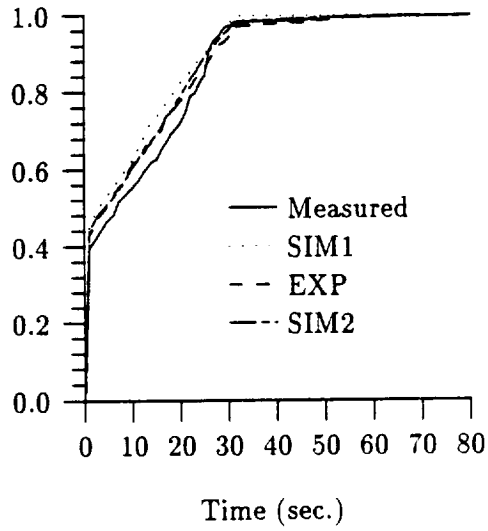


Figure 6.11: Measured and simulated cumulative detection latency distribution functions.

fact that all programs will display such behavior thus making the EXP detection model widely applicable. However, given that the two programs are very different in structure and in the types of data they process, one can conjecture that for RISC machines, similar behavior will be seen across a large number of programs. We say this because with RISC's limited instruction set, the instruction mix for different programs are remarkably similar. In [76], the author lists the frequency of occurrence of each instruction of the Gaussian elimination program and a program that finds anagrams in a large string of characters. The instruction mix for these very different programs are quite similar.

The next subsection will use the Gaussian elimination PCFG to analyze two scrubbing mechanisms.

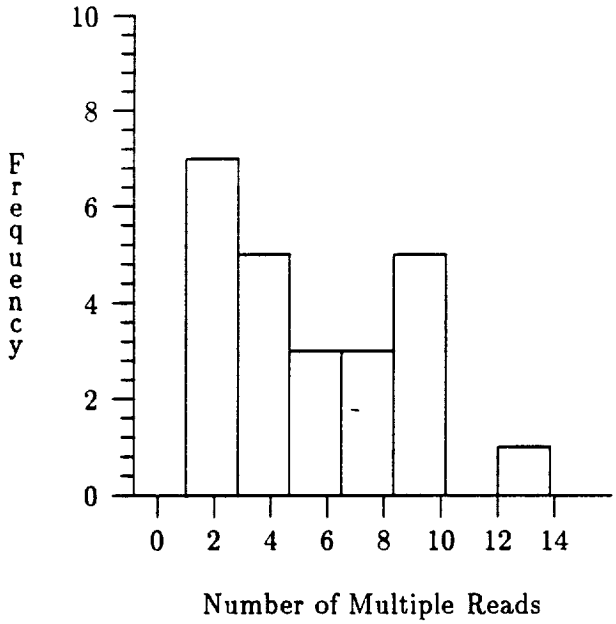


Figure 6.12: Frequency of multiple accesses of corrupted location (Measured).

6.3.5 Coverage of the Memory Scrubber

This section defines and computes the *active coverage* of the scrubber. The scrubber coverage, SC, is:

$$SC = Pr[e_a]Pr[S_l < P_l] + (1 - Pr[e_a])$$

The scrubber's *active coverage* is $Pr[S_l < P_l]$, and is the probability that the scrubber will detect an active error before the program. S_l is a random variable denoting the detection latency of the scrubber. P_l is a random variable denoting the detection latency of the program.

Emphasis is placed on the active coverage because active errors not caught by the scrubber cause a processor board to be shutdown followed by a lengthy POST and re-integration period. This time is a "window of vulnerability" within which a second error (whether active or not) will cause the entire system to fail. Increasing the active coverage of the scrubber will reduce the probability of processor board failures and increase the reliability and availability of the system.

The latency distribution of the scrubber, $f_{S_l}(s)$, can be shown to be uniformly distributed, $f_{S_l}(s) = 1/T, 0 \leq s \leq T$, where T is the time required to complete one sweep through the memory. If $f_{P_l}(p)$ is the distribution of the program's detection latency, then the joint distribution

No. Programs	$Pr[S_l < P_l]$	
	Simulation	$E[P_l]/T$
1	0.0025	0.00246
2	0.005	0.0048
4	0.009	0.0087
8	0.016	0.0165
16	0.032	0.0325

Table 6.6: The *active coverage* of the scrubber.

of the two detection latencies is $f_{SP}(s,p) = f_{P_l}(p)/T$, by stochastic independence and:

$$\begin{aligned} Pr[S_l < P_l] &= \int_{p=0}^{T}\int_{s=0}^{p} f_{SP}(s,p)\ dsdp + \int_{p=T}^{\infty}\int_{s=0}^{T} f_{SP}(s,p)\ dsdp \\ &= 1/T \int_{p=0}^{T} p \cdot f_{P_l}(p)\ dp\ +\ (1 - F_{P_l}(T)) \end{aligned} \tag{6.2}$$

For T much larger than the mean program detection latency time, the approximate active coverage is:

$$Pr[S_l < P_l] \ \cong\ E[P_l]/T \tag{6.3}$$

One hundred thousand errors were injected to determine the active coverage of the scrubber for $T = 3600$ seconds. Table 6.6 contains the results obtained with simulation and with Equation 6.3. Equation 6.3 provides accurate active coverage values for large T. Hypothesis testing was used to show that the means in column 2 and 3 are statistically identical. By the Central Limit Theorem, a normal distribution was assumed. Table 6.6 shows that the active coverage is extremely small. It improves with increasing number of programs because the combined program detection latency times increase.

The coverage of the scrubber can be improved by using two scrubbers: one to scrub only unused memory while another scrubs the allocated pages in memory. This heuristic can be implemented easily because the memory system keeps track of allocated and unallocated pages. The overhead of the original scrubbing scheme is:

$$OVHD_{old} = Nk/T$$

where, N is the number of 32-byte blocks in the memory and k is the overhead for scrubbing one block. The overhead per hour, for $T = 3600$ seconds is approximately 1.5 seconds. The

Active Coverage of New Scrubber			
$OVHD_{new}/OVHD_{old}$	1 Prog.	4 Prog.	16 Prog.
1	0.33	0.32	0.31
2	0.41	0.40	0.38
4	0.45	0.43	0.41
8	0.51	0.45	0.44
16	0.61	0.49	0.46

Table 6.7: Active coverage of the new scrubber.

overhead of the new scrubbing scheme is:

$$OVHD_{new} = k(N_u/T_u + N_a/T_a) \tag{6.4}$$

where, N_u is the number of unallocated 32-byte blocks, N_a is the number of allocated blocks, and $N = N_u + N_a$. T_u and T_a are the scrubber sweep times for the unallocated and allocated pages, respectively. Table 6.7 contains the active coverage of the new scrubber for various overheads, with $T_u = 4T$. T_a is determined from Equation 6.4 for a specified overhead. The executable image of one Gaussian elimination program is 32KB (N_a = 1K blocks). The new scrubber improves the active coverage by one or two orders of magnitude without increasing the overhead. However, since T_u is four times slower than T, the impact of the two scrubbers on system dependability should be evaluated before one is selected.

Since the dominant factor of the new scrubber is T_a, Equation 6.2 can provide an approximate active coverage if $f_{P_l}(p)$ is known. A statistical package was used to fit the measured program detection latency shown in Figure 6.6b with a 2-phase hyperexponential distribution, HYPER($\alpha_1, \lambda_1, \alpha_2, \lambda_2$). The fitted curve is HYPER(0.5, 0.045, 0.5, 5.435) with an r^2 value of 0.99. Using the 2-phase hyperexponential and substituting T_a for T in equation 6.2, the active coverage is:

$$Pr[S_l < P_l] = \alpha_1(\frac{1}{T_a\lambda_1} - \frac{e^{-\lambda_1 T_a}}{T\lambda_1}) + \alpha_2(\frac{1}{T\lambda_2} - \frac{e^{-\lambda_2 T}}{T\lambda_2}) \tag{6.5}$$

Table 6.8 contains the T_a values and the coverage estimated with the above equation. The results match those shown in columnn 2 of Table 6.7 well and verify Equation 6.5.

To show the impact of using simple estimations of application behavior, we compute the active coverage using the ramp function. We also compute the active coverage assuming $f_{P_l}(p)$ is exponentially distributed with a mean of 8.8 seconds (the measured mean for 1 program).

$OVHD_{new}$ / $OVHD_{old}$	T_a (sec.)	Estimated Active Coverage
1	19.62	0.34
2	8.41	0.43
4	3.93	0.48
8	1.90	0.53
16	0.935	0.59

Table 6.8: The estimated active coverage of the new scrubber for one program using a fitted $f_{P_i}(p)$.

Active Coverage of New Scrubber for 1 Program				
$OVHD_{new}$ / $OVHD_{old}$	Ramp		Exponential	
	Cvrg.	% Error	Cvrg.	% Error
1	0.70	112.1	0.41	24.2
2	0.85	107.3	0.65	58.5
4	0.93	106.6	0.81	80.0
8	0.97	90.2	0.90	76.5
16	0.99	62.3	0.95	55.7

Table 6.9: Active coverage of the new scrubber using ramp and an exponential detection latency distribution.

Table 6.9 contains the coverages obtained. The error column compares the difference in coverage with those obtained using the software model (column 2 of table 6.7). The error is due to an overestimation of the number of large detection latency times. Figure 6.5 shows that over half of the measured detection times are less than a second, whereas the 50th percentile for ramp is approximately 18 seconds. The coverages are very misleading. Furthermore, they can produce extremely optimistic dependability figures causing the designers to falsely assume that dependability specifications are being met. These results emphasize the importance of application dependent evaluation - especially when studying application specific systems.

6.4 Summary

In this chapter, we introduced a software model that provides a framework to evaluate the behavior and effect of software on hardware faults. The model represents application programs by decomposing them into *graph models* consisting of a set of nodes, a set of edges that prob-

abilistically determine the flow from node to node, and a mapping of the nodes to memory. The software model simulates the execution of the programs while errors are injected into their memory space. The result provides application dependent parameters such as detection and propagation times. The model is especially useful in the early design stages because it allows designers to make application dependent evaluation of function and system level error detection and recovery schemes. One use of the model was illustrated with a case study. The model was used to obtain error detection latency times of the **Gauss** and **sort** programs running on a Tandem Integrity S2 system and evaluate the coverage of two memory scrubbing schemes. The applicability of the model for different programs was evaluated by studying its sensitivity to the detection parameter ρ_t. The EXP detection model was shown to be applicable to both **Gauss** and **Sort** which use different instruction sets. We feel that this model is generally applicable to most compute bound programs running on RISC processors. Error detection latency times obtained with the model were validated with measurements from an actual Integrity S2 system. Formulae which were derived to estimate application dependent *active coverage* values of the scrubbing schemes were verified with the software model. The application dependent coverage values obtained with the model were compared with those obtained via traditional schemes that assume uniform or ramp memory access patterns. For our program, some coverage values obtained using the traditional approach were found to be 100% larger than those obtained with the software model. This result emphasizes the need for application dependent evaluation – especially when evaluating the dependability of application specific systems.

Chapter 7

Simulation Acceleration

The previous chapter evaluates the impact of application software running on a single processor. The question is how can the study be extended to perform application specific dependability analysis of the entire system? A naive approach that models the execution of each PCFG over a simulated period of several years would require astronomically large execution times. Clearly, acceleration techniques are necessary to speedup the simulation. This is especially the case for functional simulation tools such as DEPEND because one of its advantages is the ability to simulate system functionality with detailed algorithms. How you evaluate detailed simulations of systems for extended periods of time is the focus of this chapter.

This chapter presents acceleration techniques used to reduce simulation execution time while still allowing detailed analysis of a system. A combination of hierarchical, time acceleration, and hybrid simulation is used. The overall approach is discussed and then illustrated in stages. The results of each stage are then validated, either with measurements from the Integrity S2 or with simulations that do not use the acceleration technique. The automated environment that allows the user to employ this acceleration facility is described in Appendix A.

7.1 Acceleration Approach

The acceleration technique uses a combination of three schemes to provide simulation speedup. It uses hierarchical simulation, a time acceleration algorithm, and hybrid simulation. Because the terms hierarchical simulation and hybrid simulation have been widely used to mean different things, we now define how the terms are used in this thesis. By *hierarchical simulation*, we mean

that the same simulation approach is used to model a system at different levels of abstraction. Results from a detailed model are used by a more abstract model thus tieing the two together. With *hybrid simulation* the level of abstraction remains the same but a combination of two or more techniques are used to solve the simulation model. In DEPEND, a functional simulation is either combined with a Monte Carlo simulation [40, 28] or with a Markov or Semi-Markov model [70]. Existing Markov analysis tools are used to solve the Markov model. While many define Monte Carlo simulation to be *any* kind of simulation that uses random numbers, we differentiate between functional simulation and Monte Carlo simulation. In the latter approach, the passage of time does not play a substantive role [40].

The general framework of the acceleration technique is illustrated in Figure 7.1. Portions of the system are first analyzed with detailed functional simulation models. Key characteristics of the models are extracted and stored as statistical models. A statistical model can be a point statistic such as a mean or a median, an entire distribution such as the program detection latency distribution, or it can be a table. A fault dictionary is an example of a table. A fault dictionary may consist of a list of possible gate-level faults and for each fault contain all possible failure modes and their probabilities. The statistical models are used by more abstract functional simulation models to represent the behavior of the detailed model. As Figure 7.1 shows, the hierarchical approach can be used repeatedly to ascend to higher levels of abstraction. Simulation speedup is achieved because the events in the detailed model are not simulated but are replaced with its statistical model. This allows the abstract model to simulate more of the system while maintaining reasonable execution times.

The time acceleration technique is intimately entwined with the hierarchical simulation approach. It is based on the notion that a statistical model provides information about a future event. For instance, if the statistical model is a *program detection latency* distribution, then the time at which an error will be detected, $t + x$, can be determined by sampling from the distribution at time, t, when the error is injected. Using this information, the time acceleration technique can leap to a point just before time $t+x$ and begin to simulate the detailed behavior of a system. Once the error is detected and its effect has subsided, the time acceleration mechanism can leap forward to the next error detection (Figure 7.2). This approach can significantly speedup simulation time, especially if the events representing the detailed behavior occur in nanoseconds while error latency times are in seconds.

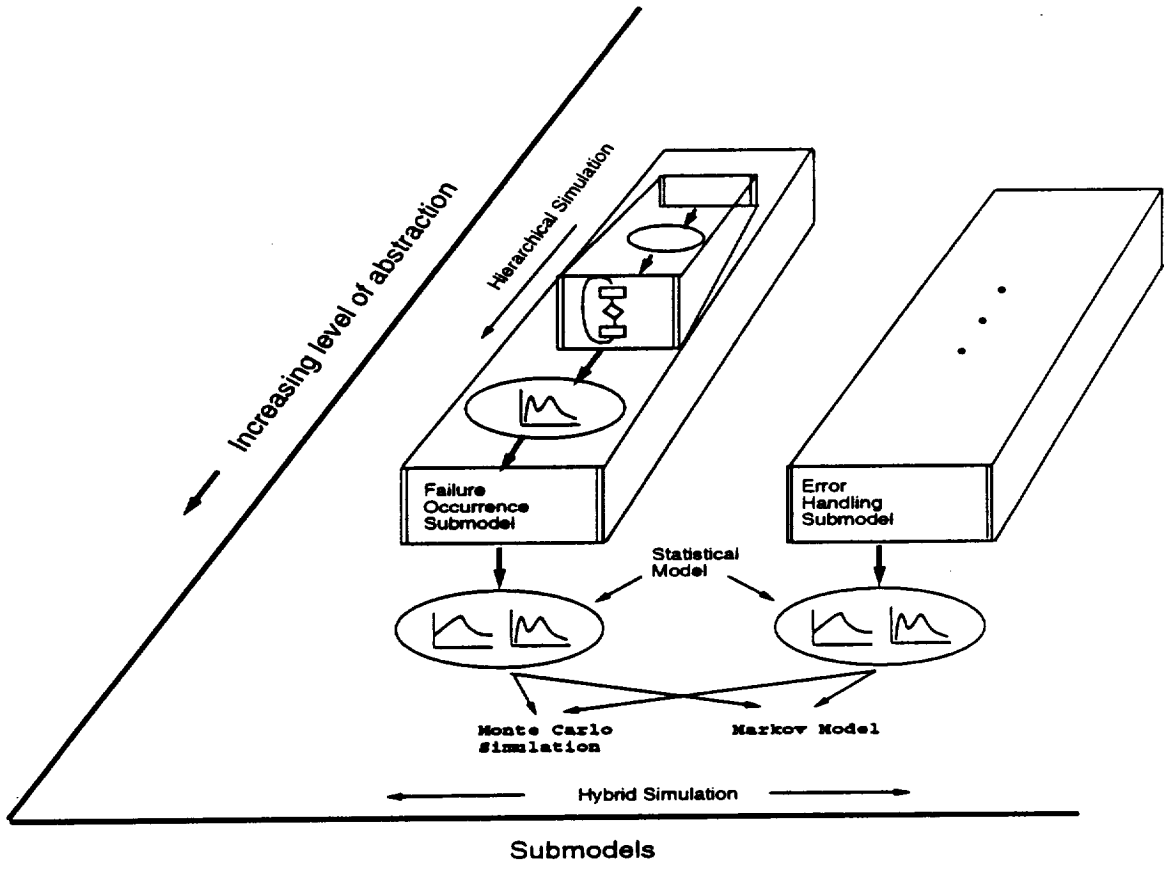


Figure 7.1: The framework of the acceleration technique.

The foundation of the hybrid simulation approach is based on the notion of variable aggregation and decomposability [13]. With this technique, a large complex model is broken down into simpler submodels. The submodels are analyzed individually and their results are combined to derive the solution of the entire system. So long as the interactions among the subsystems are weak, this approach provides valid results. Figure 7.3 illustrates how this approach is readily adapted for dependability evaluation. A model of a system is broken down into two submodels: the *failure occurrence submodel* and the *repair submodel.* Each submodel is modeled with a functional simulation – which can itself use hierarchical simulation and time acceleration. The submodel is executed to extract statistical models (e.g. a failure distribution) that represent its behavior. The statistical models are then used to drive a Continuous Time Markov Chain (CTMC) or a simple Monte Carlo simulation of the failure and repair process of the entire system. The approach is not limited to any number or type of submodels. As Figure 7.1 shows,

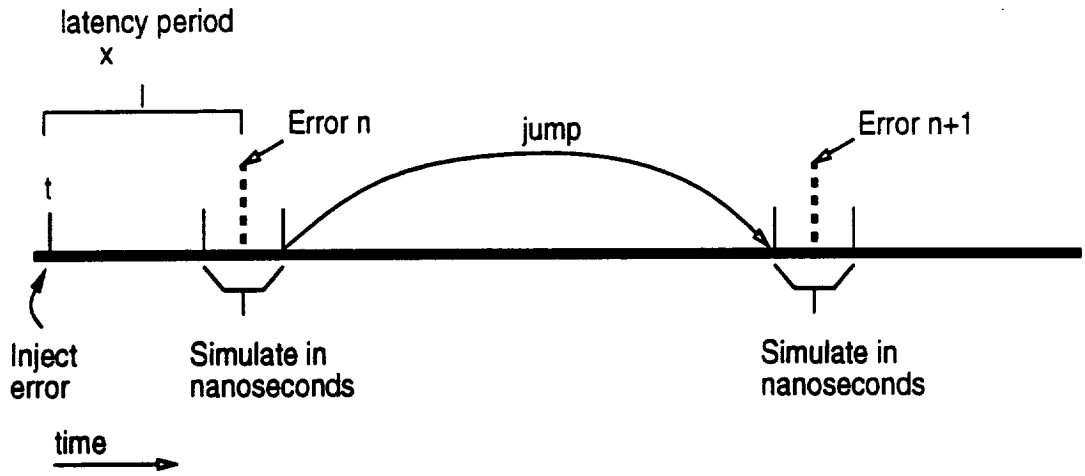


Figure 7.2: Time acceleration: "Error" driven simulation.

all three approaches can be combined to cater to the specific needs of the user. The automation of the acceleration technique is discussed in Appendix A.

There has been extensive investigation into importance sampling to reduce simulation time [41, 50, 62, 72]. Importance sampling is a statistical approach that increases the probability of failure occurrences to reduce the time required for the simulation to converge. The results produced by the simulation are then "unskewed" by multiplying by a factor that takes into account the increased failure probability. These heuristics are easiest to apply to Monte Carlo simulations as opposed to functional simulations, and their estimation of stationary measures (e.g. MTTF and availability) are mostly restricted to regenerative models. Hybrid techniques have been used in the past. Schwetman [59] applies this technique to analyze a multiprogramming computing center. A simulation models the arrival and activation of jobs and calculates the percentage of time spent at each multiprogramming level. A central-server queueing network is used to model the processing subsystem for each of the mulitprogramming levels. The hybrid model is 18 to 200 times faster than a pure simulation model because it does not simulate the system processor in which events occur in the order of milliseconds. In other words, speed up is achieved by replacing the time consuming segment of the model with an approximate analytical model. The most prominent example of the use of hybrid techniques in reliability analysis is the HARP tool [4]. HARP decomposes a model into a fault occurrence/repair model (FORM) and one or more fault/error-handling models (FEHM). The FEHMs are simulated with an extended stochastic Petri Net (ESPN) to obtain instantaneous coverage probabilities.

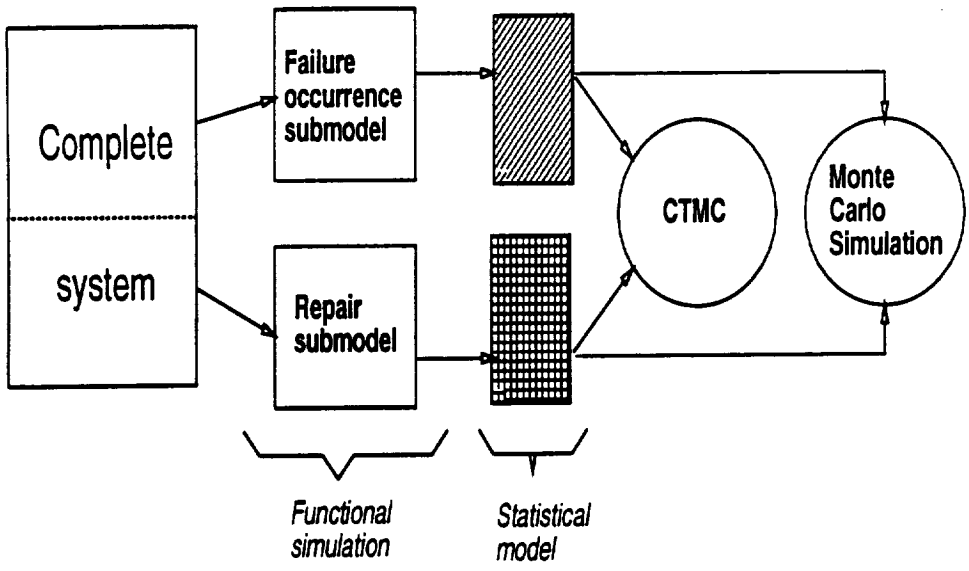


Figure 7.3: Hybrid simulation for dependability evaluation.

These probabilities are then automatically incorporated into the FORM model, represented by a CTMC, and solved to obtain system reliability measures.

Our acceleration technique differs from the others in that it does not rely solely on analytical models to achieve speedup. Schwetman replaces the time consuming portion of the simulation with an analytical approximation. HARP imposes a Markov or semi-Markov fault occurrence process which can only use instantaneous coverage probabilities. These techniques are not widely applicable. Our acceleration technique is more widely applicable and concurs with the design philosophy of DEPEND. The acceleration approach can be used with functional simulation. It is not limited to statistical models consisting of single point statistics but is designed to collect distributions and tables. Later in the chapter, we show that this can be crucial to the results obtained. Since the approach breaks down an entire functional simulation model into smaller submodels, importance sampling techniques that may be impossible to apply to the entire model may be readily applied to the submodels. Finally, this is the first automated, general-purpose acceleration technique provided with a simulation tool.

In the rest of the chapter, the details of the acceleration approach is illustrated in stages. The results at each stage are verified with measurements.

7.2 Stage 1: Hierarchical Simulation and Time Acceleration

To perform application specific analysis of the Tandem Integrity S2 system, the software simulation model described in chapter 6 and the hardware simulation model described in chapter 5 are combined using hierarchical simulation and time acceleration. The software model is used to provide the application specific error latency distribution. The hardware model provides an error occurrence model that takes advantage of the empirical distribution to speedup simulation. Together, the approach is used to study the effectiveness of the scrubbing scheme and the availability and MTBF of the Tandem Integrity S2 system running a specific application.

The complete error occurrence process modeled by the hardware simulation, for just two CPUs, is illustrated in Figure 7.4. A similar process is used to inject errors into the global memory. As shown in the figure, the error arrival times are exponentially distributed with a mean of λ hours. The error occurrence process can model correlated errors that affect more than one component, active errors that are detected within 2047 cycles and latent errors that remain undetected in the system for periods exceeding 2047 cycles. For each error injected, probabilistic branches are used to determine whether it is a correlated (affecting more than 1 component) or a single error, and whether it is *active* or *latent*. The probabilities are specified at run time. The hardware simulation model uses the approach described in chapter 4 to represent latent errors. It uses a chronologically sorted queue to maintain the latent errors injected into the system. Among the information associated with each latent error include the time at which the error is injected, its location (the component and memory address), and its *latency period*. Typically, the errors are detected when their latency period expires. However, the errors can also be corrected by the scrubber or detected during a repair. Different scrubbing algorithms, the location in which the errors are injected, and component failure frequency changes the activation times of a latent error. Storing the latent errors in a queue allows dynamic determination of whether a latent error is detected or corrected and the time of detection or correction.

The *latency period* of each error is determined from an empirical program detection latency distribution. This distribution is obtained from the software model. The steps in the hierarchical simulation approach are as follows:

- Execute software model with one or more PCFGs.

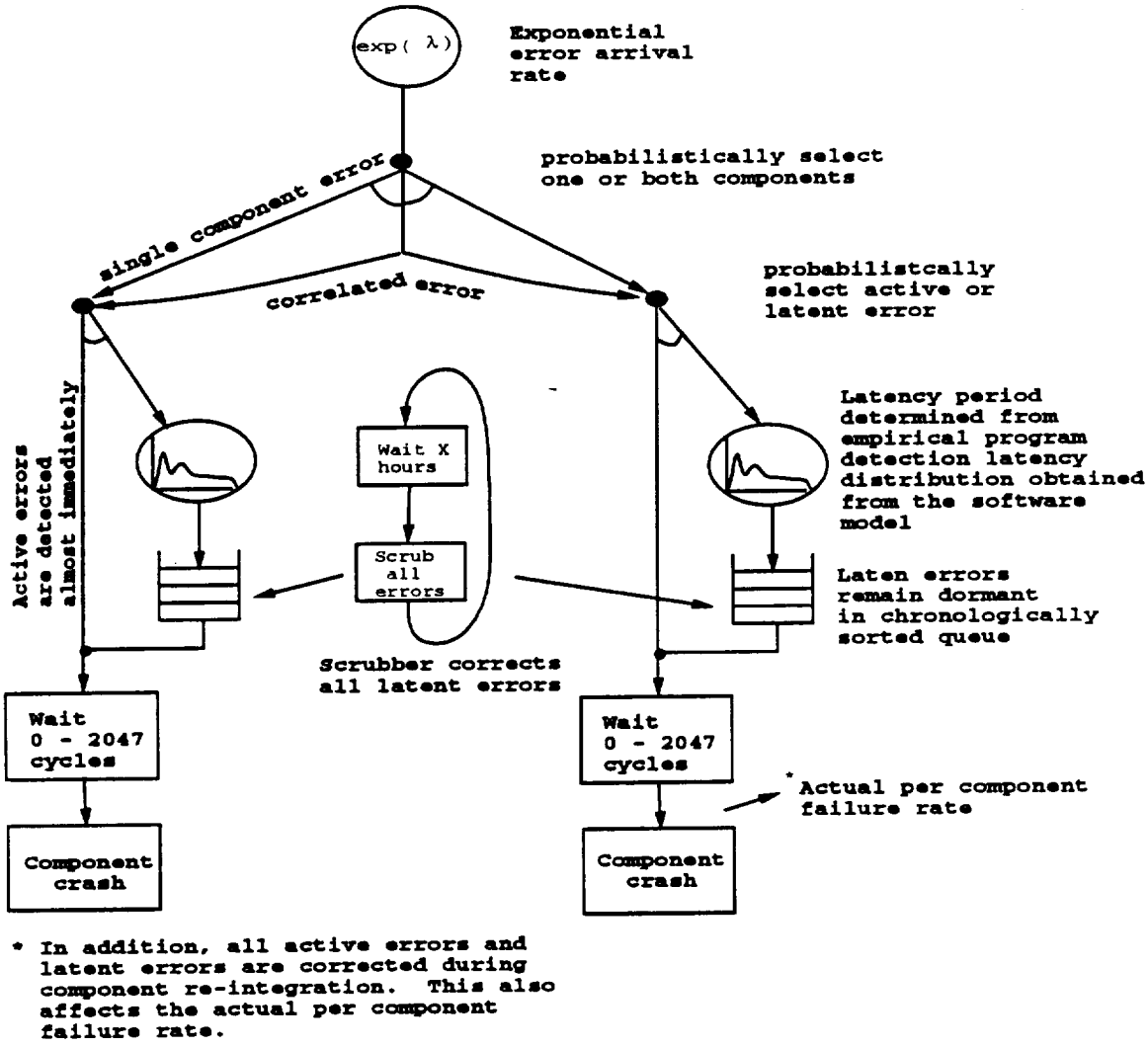


Figure 7.4: The error occurrence process for 2 CPUs.

- Store the *program detection latency* distribution. This is the statistical model that represents the error detection process of the programs.
- Execute the hardware simulation model with the error occurrence process described above. Sample from the empirical program detection latency distribution to determine the *latency period* of each error.

Speedup is achieved because the time consuming low level simulation of the detection process of each error is replaced with a distribution. Further speedup is achieved using time acceleration. Time acceleration takes advantage of the fact that the detection time of each latent error is known *apriori*. The time acceleration algorithm works as follows:

1. Poll the processors and global memory to determine time of earliest event among them. There are three possible events:
   - An error's latency period expires.
   - Scrubber corrects an error.
   - Re-integration begins.
2. Leap forward in time to earliest event.
3. Simulate system activity at nanosecond granularity until event occurs.
4. Goto 1.

The DEPEND objects used to simulate the processor and global memory contain methods that provide the time and type of the earliest event to facilitate the time acceleration algorithm.

7.2.1 Validation of the First Stage

To validate the combined functional simulation model developed using hierarchical simulation and time acceleration, the results from the simulation are compared with those from a fault injection experiment on the Integrity S2. The hybrid monitoring environment described in chapter 6 is used to conduct the fault injection experiments. Two **Gauss** programs are used as the target applications. Figure 7.5 shows the injection program that was executed to obtain the measured mean time between failures (MTBF) for the Integrity S2. The injection experiment was conducted for 28 hours during which 414 errors were injected. The mean time between failure (MTBF) and the number of undetected, latent errors present in CPUB just prior to shutdown [1] was collected. The experiment was then repeated with the functional simulation model just described. The simulation injection scenario is identical to that outlined in Figure 7.5. As in the hybrid monitoring system, the simulation model injected errors only into CPUB. The model was executed for a simulated time period of 500,000 seconds (5.78 days) with an exponential error arrival rate with a mean of 3 minutes. Figure 7.6 shows the measured and simulated CPU shutdown distributions and their means, medians, standard deviations and the sample counts. The means, medians and the standard deviations are statistically identical.

[1] This is simply a count of the number of errors injected prior to a CPU shutdown. So if four errors are injected before a shutdown, it is assumed that there are three undetected errors in the CPU prior to a shutdown.

1) Start the two Gaussian elimination workload programs.
2) Start the DAS controller and request it to start the DAS.
3) Randomly select
    the program to inject
    the address of the word to be corrupted
    the mask to use.
4) Inform the DAS of the address of the word corrupted.
5) Inject the error into the word (flip a single bit).
6) Determine time of next error, t $(exp(\lambda = 3minutes))$.
7) Wait for CPU shutdown or until t – whichever comes first.
8) If (CPU is shutdown before t elapses)
    re-integrate the CPU
    sleep until t elapses
9) Goto step 3.

Figure 7.5: Injection program used to measure the MTBF.

Comparing the distributions, the general shape of both distributions is similar in spite of the fact that the measured distribution has 5 times fewer samples. A closer look at the two distributions reveal that many of the peaks in the measured distribution can also be found in the distribution obtained from simulation. For example, the simulated distribution captures the peaks which occur between 640 and 960 seconds and between 960 and 1280 seconds. Figure 7.7 contains the measured and simulated distributions of the number of undetected, latent errors in the CPU at the time of a shutdown. The distribution obtained from the simulation model tracks the one obtained from measurement very well. We feel this close correspondence between the measured and simulated results validates the high fidelity of the simulation model using hierarchical and time acceleration techniques.

7.3 Stage 2: Hybrid Simulation

This section describes the hybrid approach that combines analytical and simulation techniques to reduce the execution time of the functional simulation model described in the previous section. In spite of the use of hierarchical simulation and time acceleration, the simulation execution time can be in the order of days. Certain parameter settings (e.g. frequent scrubbing coupled with large detection latency times) erases most errors injected thus requiring the simulation of a very large number of errors for each system failure. To make matters worse, the

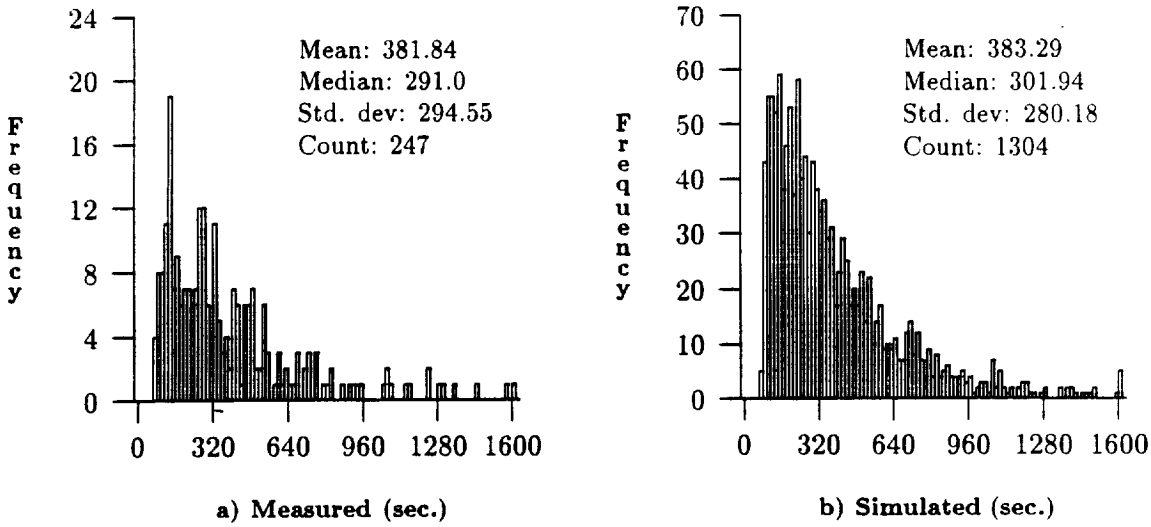


Figure 7.6: The Time to CPU shutdown distribution (in seconds).

system must be run for extended periods of time to collect enough system failure events to obtain statistically valid results. If the MTBF of a system is 10 years, it may need to be simulated for 300 to 500 years to obtain valid results. Features of the model such as the deterministic scrubbing scheme, modeling the location of each error and the inter-dependence between the system components (see chapter 4), and the non-exponential error latency distributions make this model very difficult, if not impossible, to model accurately with analytical methods. Importance sampling techniques cannot be used because they cannot accurately calculate the steady state MTBF (Mean Time Between Failures) for non-regenerative processes [50].

The hybrid approach breaks up the simulation model into the failure occurrence and repair submodels. To simplify the example, we will forego developing the repair submodel by assuming that repair coverage is 100% and repair times have an exponential distribution. The failure submodel is the error occurrence portion of the functional simulation model described in the previous section (see Figure 7.4). The functional simulation model is executed for a short time to collect component inter-failure times – the time period between successive failures. This is feasible because the component failure rate is significantly faster than the system failure rate, and so a large number of samples can be collected in a short period of time. The measured failure distributions and the repair distributions are then used with CTMC and Monte Carlo simulations which model the failure/repair process of the entire system. The detailed steps of the approach are illustrated in Figure 7.8.

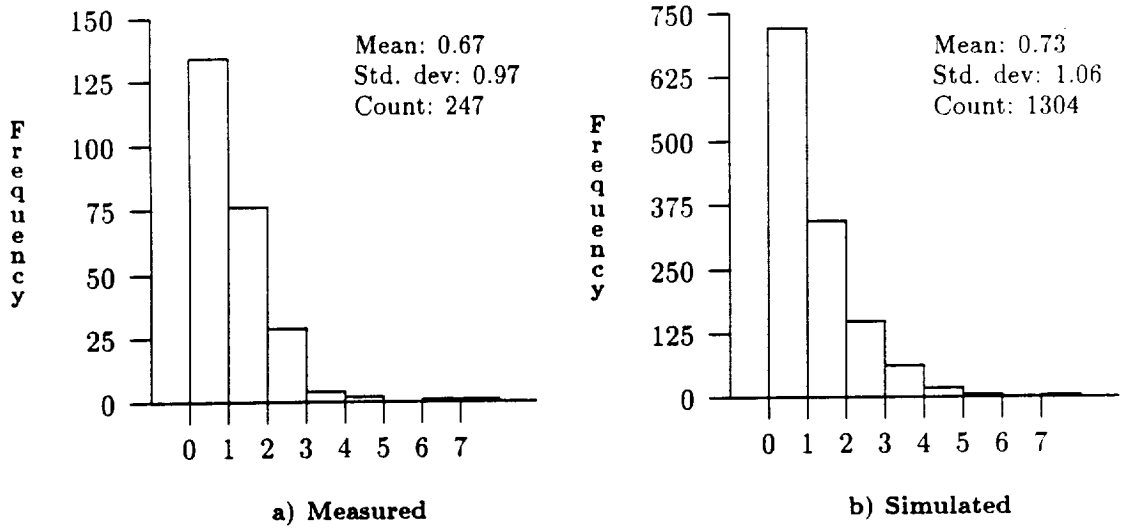


Figure 7.7: Distribution of the number of latent errors prior to a CPU shutdown.

7.3.1 Markov modeling

In order to use a CTMC, the measured failure distributions are first fitted with an exponential polynomial. A statistical analysis package, SAS [57], is used for curve fitting. The chi-square and Kolmogorov-Smirnov tests [70] are used to verify the goodness of the fit at the 0.01 significance level. The CTMC models used for the failure/repair process will depend on the fitted polynomials. If they are simple exponentials, the result is the CTMC model shown in Figure 7.9.

In the model, *0c0g* is the initial state where zero processors and zero global memory boards have failed. In state *1c1g*, 1 processor and 1 global memory board has failed etc. State F is the absorbing system failure state. The rates λ_χ and λ_γ are the empirical failure rates for the processors and the global memory, respectively. Exponential repair times with means ($1/\mu_\chi$ and $1/\mu_\gamma$) equal to the constant repair times of the system are used in the model. This CTMC can be solved with any existing analytical tool to obtain the system MTBF. The models in this paper are solved using SHARPE [54].

7.3.2 Monte Carlo simulation

With Monte Carlo simulation [28], the empirical failure distributions can be used directly without any fitting. Monte Carlo simulation is useful when the empirical distributions cannot

C-2

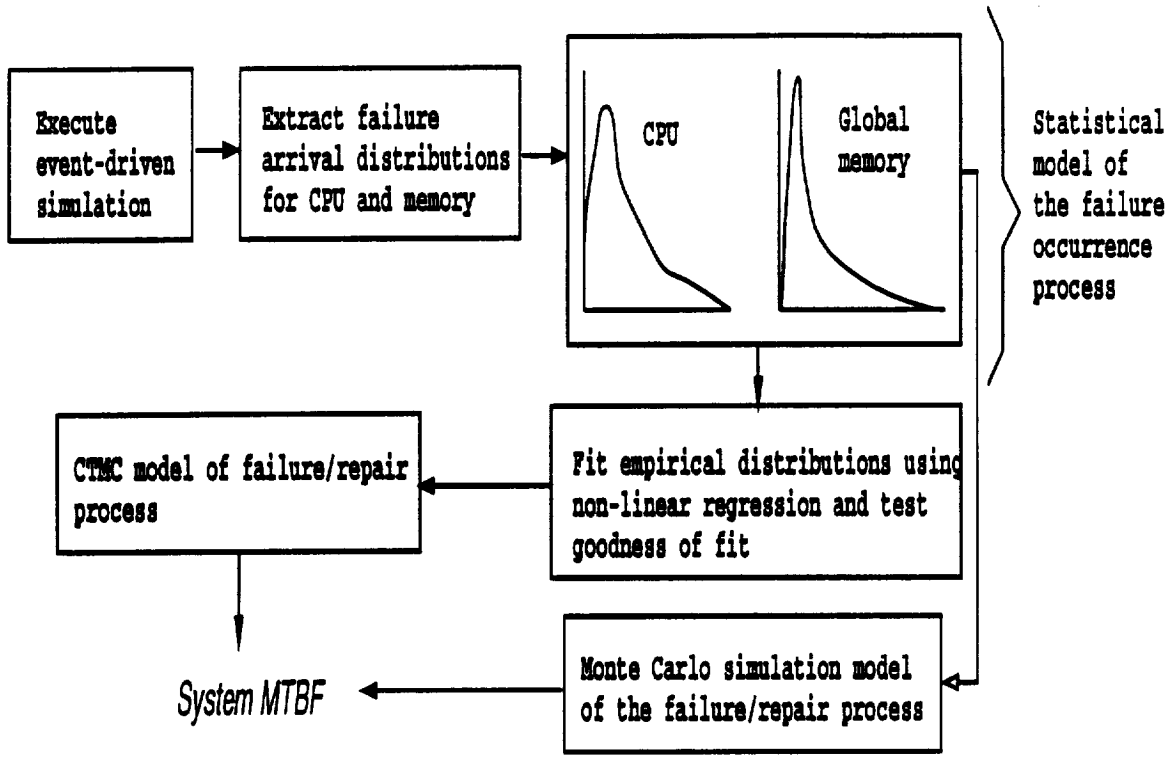


Figure 7.8: The hybrid approach.

be fitted to an analytical distribution, when the state space of the CTMC model is too large to be solved with current numerical methods and when the distributions are not exponential. Monte Carlo simulation was used to simulate the Markov model in Figure 7.9. The failure times of the components are sampled from the empirical failure distributions using an inverse-transform method [40]. The constant repair times used in the functional simulation are used with the Monte Carlo simulation also. To further reduce the execution times of the Monte Carlo simulations, a variance reduction technique called *antithetic variates* was employed [28, 40].

The success of the hybrid approach for this application depends on, 1) the degree of interaction between the failure process and the repair process, and 2) on whether the failure distributions adequately represent the failure process. The repair process has minimal impact on the failure process. Therefore, the primary issue is whether the distributions collected adequately represent the actual failure process.

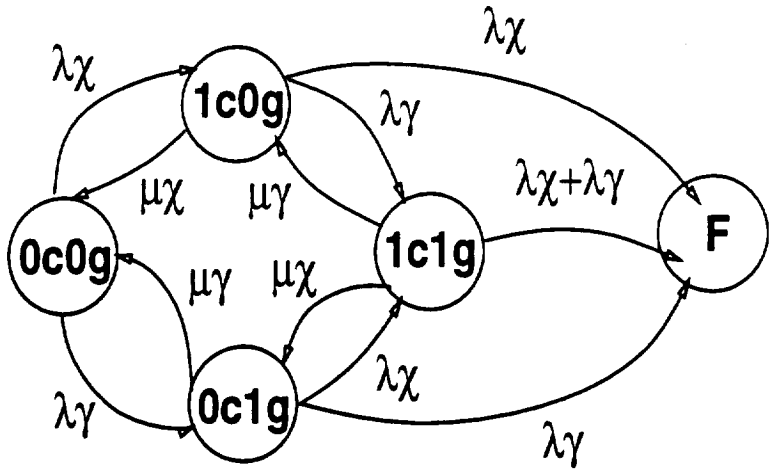


Figure 7.9: Markov model of system with exponential failure & repair distributions.

7.3.3 Validation of the Hybrid Approach

Three experiments are conducted using the hybrid approaches to verify their efficiency and validate their accuracy. The results obtained using the hybrid techniques are compared with results obtained from executing the functional simulation model described in the previous section. Parameters are chosen so that the execution times of the functional simulation times are reasonable. To further speedup the functional simulation model, the empirical latency distribution is replaced with a normal distribution. This was done because sampling from an empirical distribution is relatively expensive requiring a binary search of the entire distribution file for each sample produced.

In the first experiment, a normally distributed error latency with a mean of 44 minutes and a standard deviation of 29 minutes ($N(44, 29)min.$) was used. Two percent of the errors injected were correlated, and memory scrubbing was not activated. The second experiment uses the same parameters, except hourly scrubbing is activated. The third experiment evaluates the functional model with two different latency distributions. The first is $N(44, 29)$ minutes and the second is $N(36, 18)$ hours. Correlated errors are not injected and scrubbing is turned off. All three experiments assume two minutes are required for global memory reintegration, 2 seconds for CPU reintegration and 70 seconds for POST.

The functional simulation program was retrofitted to extract the inter-failure times of the processors and the global memories. Figure 7.10 shows how CPU inter-failure times are col-

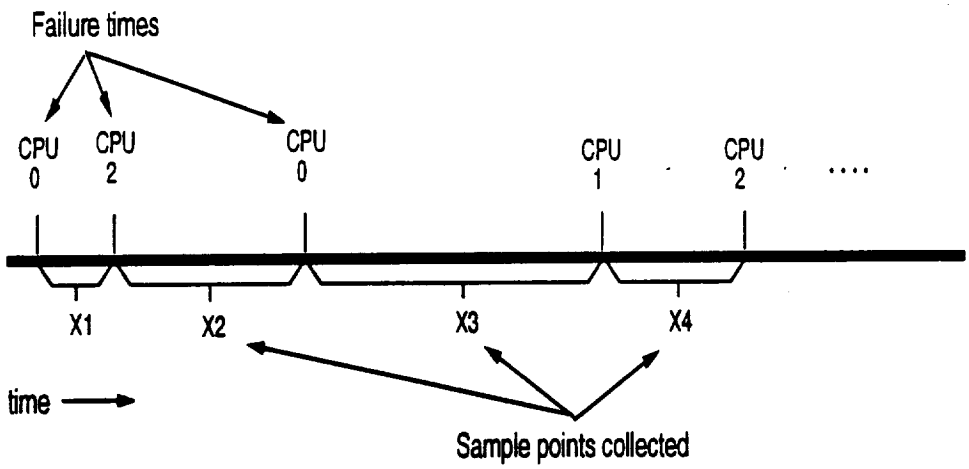


Figure 7.10: Inter-failure times extracted from an execution of the functional simulation.

lected. Inter-failure times for the global memory are collected in an identical fashion. Note that the failure distributions are collected without distinguishing between the specific components. This approach was used for two reasons. First, it captures the occurrence of near-coincident failures between components. Second, it significantly reduces the state space of the Markov model in Figure 7.14 because there is no need to distinguish between the individual CPUs and memory units.

For the first experiment, the functional simulation was executed once for a simulated period of 300 years. The program took 40 minutes to complete on a Sun SparcI workstation. Fifty thousand global memory, and 42891 processor inter-failure times were collected using the method illustrated in figure 7.10. Histograms with a range from 0 to 18000 minutes (0 to 12.5 days) were used to group the samples. For the processors, 369 out of 42891 points (0.86%) fell outside this range and were discarded as outliers. For the global memory, 24 out of 50000 (0.06%) were outliers. These points were discarded to simplify the histogramming and curve fitting process. Removing these points should have little or no bearing on the system MTBF, because the impact of a few, large inter-failure times is negligible.

For the second experiment, the event-driven simulation was executed for 700 years and took approximately 100 minutes to complete. Approximately twenty thousand (20606) global memory and 17,585 CPU inter-error arrival times were collected. The sample data was grouped using histograms with a range from 0 to 575 hours (0 to 24.3 days). Between one and two percent

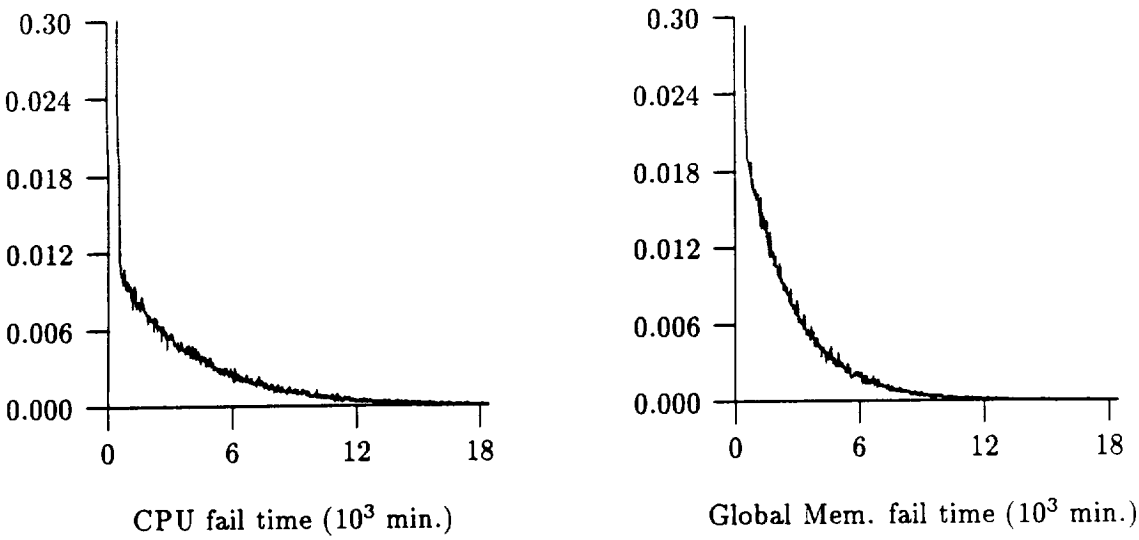


Figure 7.11: Failure density functions for experiment 1.

of the sample points fell outside the 575 hour range. For the third experiment one simulation run was executed for a period of 300 years for each error latency distribution.

The empirical failure density functions $f(x)$ for the processors and memory, for the first experiment, are shown in Figure 7.11. The spikes near the Y-axis are caused by near-coincident errors. The failure density functions for the second experiment are almost identical and are not shown. The only difference is that the spike near the Y-axis is not as pronounced. Hence, in spite of frequent, hourly scrubbing, latent, correlated errors substantially reduce inter-failure times thereby increasing the number of system failures. The empirical failure probability density functions $f(x)$ for the two latency distributions used in the third experiment are shown in Figure 7.12. The functions with the larger latency are particularly interesting. The unique shapes arise because a larger number of the latent errors are corrected when the system crashes or the board in which they reside is re-integrated. Note that the spike characteristic of correlated errors is not visible.

A two-phase hyperexponential distribution HYPER (α_1, λ_1, α_2, λ_2) [70] was found to fit the measured failure distributions collected from the first two experiments. The probability density function for a two-phase hyperexponential random variable, denoted by

$$f(x) = \alpha_1 \lambda_1 e^{-\lambda_1 t} + \alpha_2 \lambda_2 e^{-\lambda_2 t} \tag{7.1}$$

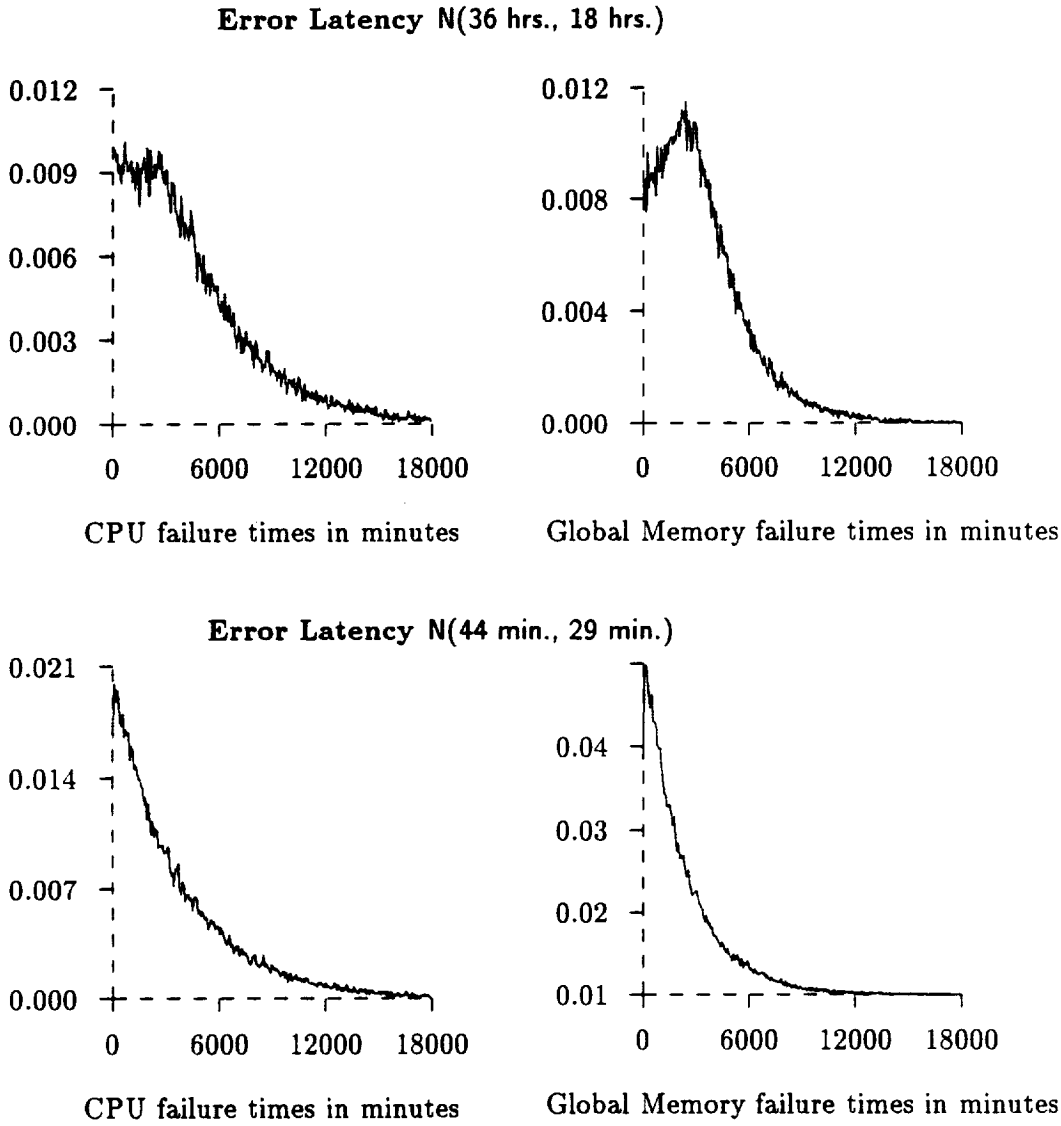


Figure 7.12: Failure density functions for the two error latencies (Experiment 3).

Number of Phases	HYPO($\lambda_1, ..., \lambda_r$)	
	CPU	Memory
2	0.000128, 0.00273	0.00299, 0.00301
3	0.000735, 0.000129 0.0006695	0.0005367, 0.000535 0.000611
4	0.000998, 0.0001181 0.001010, 0.001546	0.000481, 0.000539 0.001326, 0.001342

Table 7.1: Empirical pdfs fitted with hypoexponential pdfs.

For the first experiment, the fitted hyperexponential failure distributions of the global memories and the processors are HYPER(0.987, 0.00043, 0.013, 0.023) and HYPER(0.972, 0.000268, 0.028, 0.027), respectively. The fit was tested using both the Chi-square test and the Kolmogorov-Smirnov test at the 0.01 significance level. Graphs of the fitted curves are also shown in Figure 7.11. They are not visible, because they are almost perfectly superimposed on top of the measured curves. The fitted failure distributions for the second experiment are HYPER(0.9936, 0.00013, 0.0064, 0.09) and HYPER(0.9849, 0.00017, 0.0151, 0.095), for the global memories and the processors. These also passed the Chi-square and the Kolmogorov-Smirnov test at the 0.01 significance level.

In the third experiment, both a 2-stage hypoexponential and a simple exponential distribution was used for the case with the small error latency distribution. Both resulted in good fits according to the Kolmogorov-Smirnov test and so the simpler exponential distribution is used to represent the failure distribution. The specific fitted distributions are EXP(0.000260) and EXP(0.000414) for the CPU and the global memory failure distributions, respectively. The failure distributions for the case with the large error latencies could not be fitted with the exponential distribution. The nature of the Markov model for hypoexponential failure distributions, required that the single processor and global memory failure distributions be collected. The functional simulation model was re-executed to collect the failure distributions of a single processor and a single global memory board. To collect enough sample points, the simulation was executed for 1200 years. The resulting failure distributions were fitted with 2-, 3- and 4-phase hypoexponential distributions [70]. The pdf for an r-phase hypoexponential random variable,

denoted by HYPO(λ_1, λ_2, ..., λ_r) is:

$$f(x) = \sum_{i=1}^{r} a_i \lambda_i e^{-\lambda_i x} \tag{7.2}$$

where

$$a_i = \prod_{j=1, j \neq i}^{r} \frac{\lambda_j}{\lambda_j - \lambda_i} \tag{7.3}$$

The specific fitted hypoexponential distributions for the single memory and the single processor failure distribution are shown in table 7.1. Only the 4-phased hypoexponential distributions

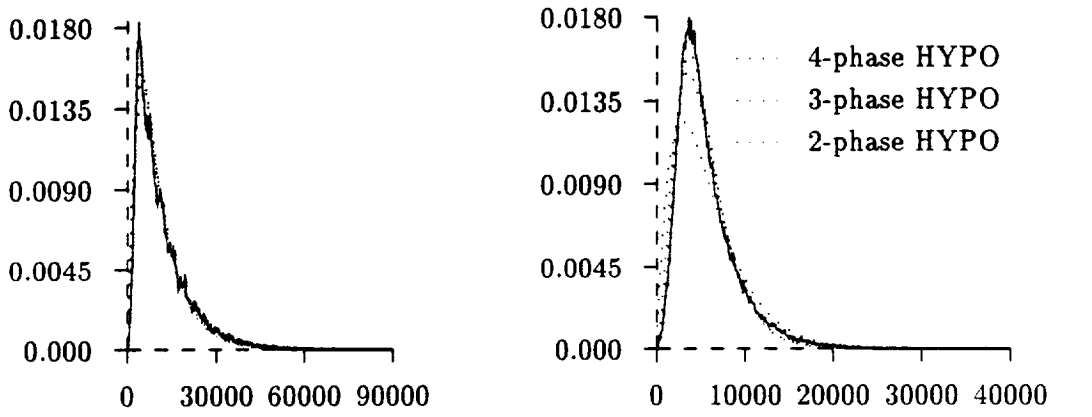


Figure 7.13: Empirical and fitted failure pdfs for a single processor & memory.

produced a good fit, according to the Kolmogorov-Smirnov test. The empirical failure distributions and the fitted distributions are shown in figure 7.13.

Once the empirical failure distributions have been collected and fitted they are used with Monte Carlo and Markov models to analyze the entire system.

7.3.3.1 Results from Experiments 1 and 2

The hyperexponential failure distributions for experiments 1 and 2 make the failure process semi-Markov. Using a conversion technique [70], it can be converted to the Markov model shown in Figure 7.14. The Monte Carlo simulation, however, can still simulate the model in Figure 7.9 using hyperexponential sojourn times.

Table 7.2 shows the times taken and the results produced by the original functional simulation and the hybrid approaches. For experiment 1, the functional simulation program, simulated a period of 80 years for each run. For experiment 2, it simulated 120 years in each run. All

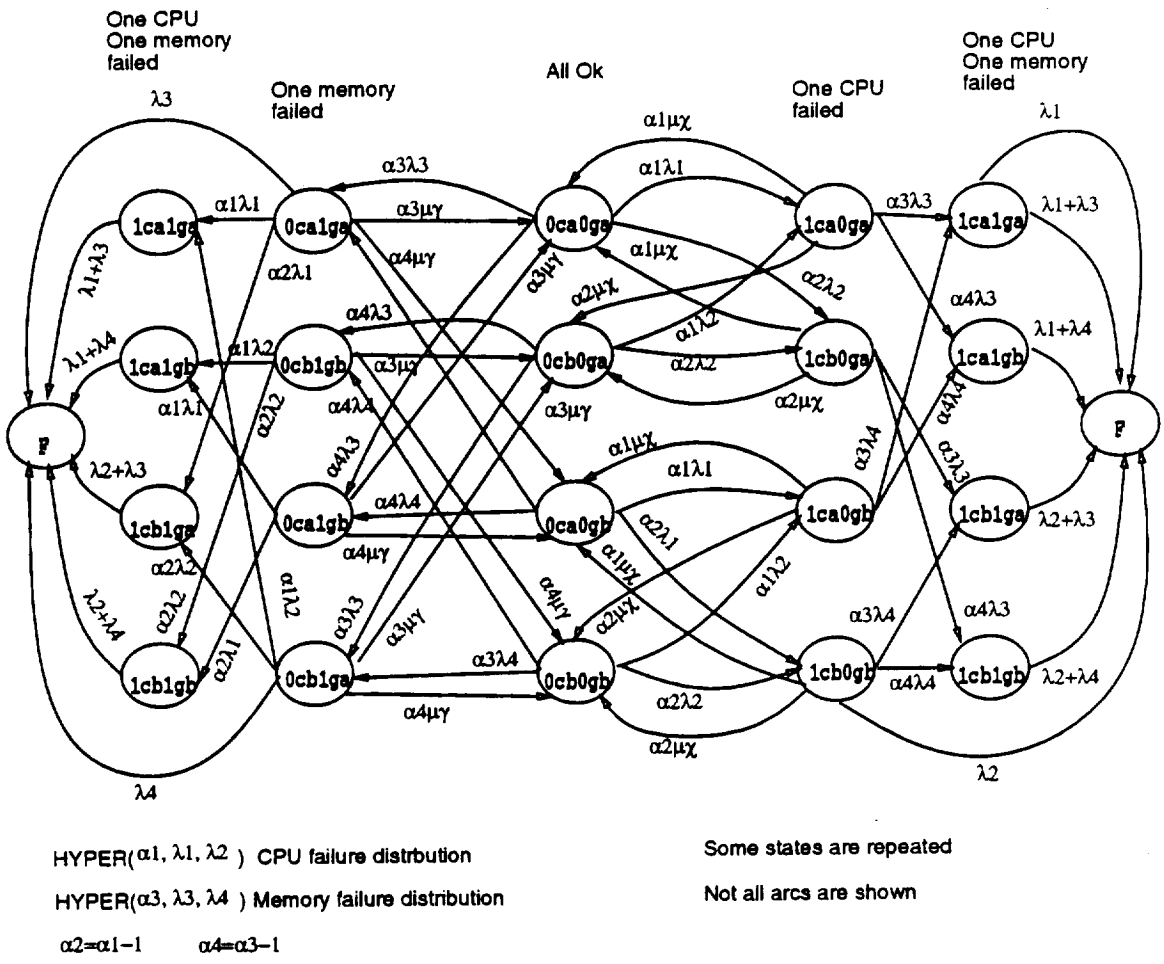


Figure 7.14: CTMC that models the system with hyperexponential failure distributions.

the results produced by the hybrid models fall within the 99% confidence interval of the results produced with the original functional simulation. The largest deviation between the hybrid models and the functional simulation model is 6.04%. The data in Table 7.2 demonstrates a substantial reduction in execution time in favor of the hybrid models. Even taking into account the time required to generate the empirical distribution, ther is more than a 36 time speedup for experiment 2. The actual speedup will vary depending on the parameters of the model and the can be much larger.

Exp.	Type	99% Conf. Int. MTBF yrs.			Time	No. Runs
1	Sim.	1.9	2.03	2.18	6.75 hrs.	30
	Hybd. Mkv	-	2.12	-	8 sec.	-
	Hybd. MC	2.09	2.11	2.13	50 sec.	500
2	Sim.	4.39	4.8	5.33	36 hrs.	30
	Hybd. Mkv	-	4.51	-	8 sec.	-
	Hybd. MC	4.957	4.96	4.975	6.95 min.	2000

Table 7.2: System MTBF obtained with the pure simulation and hybrid approaches.

7.3.3.2 Results from Experiment 3

For the case with the small error latency distribution ($N(44, 29)$ minutes), the Markov model shown in Figure 7.9 is used to determine the system reliability. The results obtained with the hybrid techniques and the functional simulation are compared in Table 7.3.

Method	Error Latency	Failure Distribution Type	99% Confidence Interval (MTBF in years)		
Simulation	N(44 min., 29 min.)	-	7.48	8.28	9.27
	N(36 hrs., 18 hrs.)	-	14.38	15.68	17.24
Markov	N(44 min., 29 min.)	Exponential	-	8.75	-
	N(36 hrs., 18 hrs.)	2-phase HYPO	-	13.93	-

Table 7.3: System MTBF for the two error latencies (Experiment 3).

The general stochastic petri net (GSPN) description of the system that models a 2-phase hypoexponential failure distribution is shown in Figure 7.15. Since by definition a 2-phase hypoexponential distribution is the sum of 2 independent exponential distributions, the hypoexponential failure distribution is modeled by adding an extra *failing* state between the *OK* and *failed* states. The GSPN description can be easily extended to model 3- and 4-phase hypoexponential failure distributions by increasing the number of *failing* states to represent the additional phases. A Markov model equivalent to the GSPN model in Figure 7.15 has 54 states if the failure states are not coalesced. For the TMR system with 3-phase hypoexponential failure distributions, the equivalent Markov model contains nearly 200 states and for the system with 4-phase hypoexponential distributions, the Markov model contains over 500 states. The Sharpe tool was used to solve all three GSPN models. However, due to the rapid growth in the

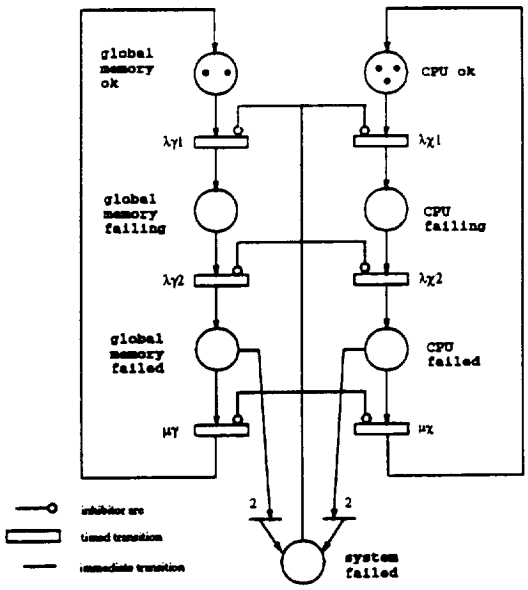


Figure 7.15: GSPN description with 2-phase HYPO failure distribution.

number of states of the equivalent Markov models, Sharpe was unable to solve the models for the 3- and 4-phase hypoexponential failure distributions. The result obtained with the 2-phase hypoexponential failure distributions is shown in Table 7.3. The result does not fall within the 99% confidence interval because the fitted 2-phase hypoexponential distributions fail to capture the spike and they exaggerate the probability of having small inter-failure times.

The Monte Carlo program simulates the Markov model in Figure 7.9. Because individual processor and memory failure distributions are not needed, the empirical distributions shown in Figure 7.12 are input to the program. The MTBF figure generated by the Monte Carlo program is 14.8 years and falls within the 99% confidence interval produced by the original simulation.

In the past, researchers have used point statistics rather than distributions for their statistical models. For detailed functional simulation models of this nature and of the type described in chapter 4, single point statistics may not be representative of the behavior of teh detailed model it replaces. For experiment 1, if the mean inter-failure times are used (implying an exponential distribution) the MTBF is 4.21 years. This is more than double the actual MTBF. This demonstrates the need to consider an entire distribution because it demonstrates that various distributions with identical means can produce significantly different results.

Once the measured failure distributions have been collected, the hybrid models can be used to evaluate alternative configurations without reverting back to the original functional

Method	Repair Min.	99% Conf. Int. MTBF yrs.			Time	No. Runs
Sim.	5.0	0.99	1.04	1.09	6.75 hrs.	30
	10.0	0.59	0.61	0.62	6.75 hrs.	30
Monte	5.0	1.09	1.1	1.11	30 sec.	5000
Carlo	10.0	0.616	0.62	0.625	18 sec.	5000
Markov	5.0	-	1.06	-	8 sec.	-
Model	10.0	-	0.60	-	8 sec.	-

Table 7.4: System MTBF for various memory re-integration times.

simulation. The impact of architectural changes that do not alter the failure distribution, such as different re-integration times, can be evaluated rapidly. Table 7.4 contains results where the global memory re-integration times of 5 minutes and 10 minutes are used. Results obtained directly with the functional simulation are also shown to verify that the hybrid approaches produce valid results.

7.4 Discussion

This chapter presented the acceleration approach provided by DEPEND. The approach, consisting of an unique combination of hierarchical simulation, time acceleration and hybrid simulation, was illustrated in stages. The system evaluated was the Tandem Integrity S2 system. Results of each stage were validated with either measurements from an Integrity S2 or from a functional simulation that does not use the acceleration technique. The results demonstrate that the acceleration technique can produce accurate results at a fraction of the cost. How the approach is automated within DEPEND is described in Appendix A.

The acceleration approach replaces processes, which are difficult to model analytically and expensive to simulate, with statistical models. For the Integrity S2 application, the statistical models used were failure distributions. However, for different applications, other statistical models may be used. The acceleration approach is particularly attractive if a system is analyzed extensively. Then, the overhead of extracting and fitting distributions can be amortized over the time needed to run various experiments and study alternative configurations. Using hierarchical and hybrid simulation, it is possible to represent a complex, functional simulation (Figures 5.2 & 7.4) with a much simpler, abstract model (Figure 7.9). It is easier to apply variance reduction

techniques to this model than the original simulation program. Finally, for extremely large, complex systems, where the complexity of the submodels are significant, this approach can be used recursively on the submodels as well. This chapter illustrated how the approach can be used to include the impact of software under hardware faults when analyzing the entire system. This same approach may be a viable way of including the impact of chip-level design when evaluating system reliability measures.

Chapter 8

Analysis of the TMR-based System

The major issues in simulation-based dependability analysis have been discussed and our solutions for them have been presented in the previous chapters. This chapter uses the DEPEND tool, the simulation models described in chapters 5 and 6 along with the acceleration techniques presented in chapter 7 to analyze a Tandem Integrity S2-like machine. The purpose of the study is to investigate issues which include: correlated errors, accurate modeling of correlated errors, latent errors, inter-component dependencies during automatic repair, memory scrubbing heuristics, application specific analysis of scrubbing schemes, impact of repair times, impact of different configurations, and isolation of dependability bottlenecks. The study is conducted in phases to isolate and analyze the impact of factors such as correlation and error latency. In addition, studies that evaluate the combined impact of several of the factors are also conducted. The goal of the study is to determine what error conditions are especially detrimental and what architectural features are especially susceptible to the types of errors injected. This comprehensive study illustrates many of the capabilities of DEPEND in a realistic setting.

8.1 Assumptions and Parameters Used in the Simulations

For the most part, the studies use a standard set of failure rates, latency distributions and architectural configurations. This section describes the parameters and the assumptions made. When other parameters are used, it will be explicitly stated.

The error arrival means are based on findings from a measurement-based analysis of real error data collected from a DEC VAXcluster multicomputer system[68]. Tang found that the

mean time between CPU errors ($1/\lambda_{CPU}$) in the system was 265.8 hours with a standard deviation of 497.6 hours. The mean time between memory errors ($1/\lambda_{Memory}$) was 27.0 hours with a standard deviation of 150.4 hours. The combined error arrival rate is approximately 1 every 24 hours. Of this combined rate, approximately 62% of the errors are injected into the global memory and 38% are injected into the processor board containing the CPUs and the local memories. These numbers are based on the size of the memories (8Mbytes of local memory per board and 32Mbytes of global memory per board) and the contribution of the CPU error arrival rate to the combined error arrival rate. Of course, one cannot assert that the error rate of the Tandem Integrity S2 is similar to that of the VAXcluster. Since the studies are not concerned with absolute reliability figures but rather with the trends and changes to system reliability due to various conditions, this does not pose a problem. An exponential distributed error arrival rate is assumed.

Several different error latency distributions are used in the studies. The latency distributions obtained in chapter 6 are used when studying different memory scrubbing schemes and to perform application specific analysis of the system. Otherwise, exponentially distributed error latency distributions with various means are used to analyze trends in system reliability as the latency gradually increases or decreases. Normally distributed latency distributions are also used. In some studies, very large latencies are used and are based on findings from a measurement study of the VAX 11/780 by Chillarege [9]. Chillarege found that error latency is workload dependent. Errors injected at midnight, when the workload was low, had a mean latency of 8 hours with a standard deviation of approximately 4 hours. Errors injected at noon, when the machine was used heavily, had a mean latency of approximately 44 minutes and a standard deviation of 29 minutes. These measured error latencies are approximated by normal distributions with the means and standard deviations just mentioned.

All errors, including undetected latent errors, residing in a processor or in a global memory are assumed to be corrected when the component undergoes a re-integration, the system is rebooted or when scrubbing takes place – regardless of the number of latent errors residing in the system. Therefore, not all the errors injected are detected and the actual error arrival distribution of *detected* errors depends on the scrubbing rate, the component re-integration rate, the error latency and the injection rate. Because error latencies and global memory re-integration times are workload dependent, various system workloads are implicitly modeled

by varying these parameters. Finally, since we are primarily concerned with transient errors, permanent errors are not injected and the MTBFs presented do not reflect their impact on system reliability. The voters are assumed to be error free. For most studies, the repair coverage, the probability that a failed component will be successfully re-integrated into the system after an error is detected, is assumed to be 1 so long as a second error is not detected before the re-integration is completed.

Except where explicitly stated, the simulations are executed with a POST time of 60 seconds, a global memory re-integration time of 2 minutes and with memory scrubbing turned off. The mean time between failures (MTBF) is calculated by dividing the simulation period by the average number of system failures. The confidence level for the results shown is 95% for an interval with a relative half-width of 5%. The MTBF figures presented in this paper should not be construed to reflect the MTBF figures of an actual Tandem Integrity S2 system because the error arrival rate which has a direct bearing on this measure, was *not* obtained from measurements of the Tandem Integrity S2.

8.2 Impact of Latent Errors

In this section, several experiments are conducted to determine the impact of latent errors on the MTBF of the system. Although there have been many studies to measure fault and error latency [9, 45, 63, 66, 74] and most researchers conjecture that latency degrades system reliability, there are very few studies that have actually tried to quantify and determine the impact of latency on the system. In [66], the authors study the effect of latency (which they define to be the sum of fault and error latency) on a TMR flight control system. They derive the probability of system failure as the probability of a second failure that produces the same erroneous output as an earlier, latent error. With both errors triggering at the same time, the voter is fooled and the error escapes and affects the flight control system. A gate-level simulation of one of the processors was injected with faults to determine parameters used in the derivation. Their results show that unless the latency is very large, the effect of latency on system reliability is very small.

In this study, we analyze the impact of the accumulation of latent errors in the system. If there are a large number of latent errors in the processor boards, is the probability of a

System MTBF in years		
Mean (min.)	Exponential	Normal
2	6.9 ±0.33	6.4 ±0.32
4	6.56 ±0.32	6.83 ±0.34
8	7.1 ±0.35	7.0 ±0.35
16	7.04 ±0.34	6.77 ±0.34
32	7.08 ±0.34	6.87 ±0.35
64	6.64 ±0.33	7.05 ±0.36

Table 8.1: System MTBF for two latency distributions with various means.

near-coincident error increased? The study also considers the inter-dependencies due to latent errors. In the Integrity S2, a latent error in a *healthy* CPU that is detected during re-integration will cause the system to fail. Latent errors can also propagate creating more errors, however, this effect is not considered in this study.

In the first experiment, the effect of an accumulation of latent errors is investigated. Errors are injected into the system at a rate of one error a day. Exponential and normal latency distributions with various means are tried to determine if there is a trend in system MTBF. For the normal distributions, the standard deviation is always half the mean. Table 8.1 presents the results of the simulation. The MTBF for the case in which the latency is zero is 6.76 (±0.32) years. The table indicates that the MTBFs for all the latency distributions produce results similar to the case when there is no error latency. This is due to two factors. First, because the error latency is much smaller than the error arrival rate, there is not a substantial accumulation of errors. Second, the detection of any error in a processor and the ensuing re-integration erases all other latent errors in the processor thus further reducing the accumulation of latent errors. The table also shows that the specific distribution of the latency times, whether normal or exponential, does not affect system MTBF.

To increase the accumulation of latent errors, two *system*[1] error arrival rates are tried: one error every 30 minutes and one error every 5 minutes. The effect on system MTBF, for exponential latency distribution times, is shown in Table 8.2. Detailed analysis of the simulation output files shows that the number of accumulated error in a processor or global memory increases, however, it still fails to reduce system MTBF. On the contrary, the system

[1] This is the number of errors injected into the system and not into each component.

Mean Latency in minutes	System MTBF in years	
	$\lambda = 1/30min.$	$\lambda = 1/5min.$
0	0.0031 ±0.00016	0.0001 ±0.00001
2	0.0034 ±0.00018	0.0001 ±0.00001
4	0.0033 ±0.00017	0.0002 ±0.00001
8	0.0037 ±0.00018	0.0002 ±0.00001
16	0.0041 ±0.00021	0.0003 ±0.00002
32	0.0052 ±0.00026	0.0004 ±0.00002
64	0.0068 ±0.00036	0.0007 ±0.00004

Table 8.2: System MTBF for two fast error arrival rates.

MTBF increases with larger latency times indicating that the accumulation of errors alone is not detrimental to system reliability. The increase in the MTBF occurs because the latency times are of the same order of magnitude as the error arrival rate and hence they increase the mean time between error detections noticeably.

These results indicate that the accumulation of latent errors, in and of itself, does not cause near coincident errors. However, the study was made under two assumptions. First, re-integration coverage is 100% and second, re-integration coverage does not depend on the state of the other components in the system. That is, there is no *intercomponent dependence* (see chapter 4). In reality, in the Integrity S2, the remaining healthy processors in the system control the re-integration of the faulty processor. During re-integration, the entire content of the two healthy processors is copied to the faulty one. If during that time, there is a discrepancy in the contents of the two healthy processors, the system fails. The same holds for the global memory boards. To model this scenario, the simulation model fails during re-integration if any healthy component contains a latent error. Table 8.3 contains the results for various different exponentially distributed latency times. The error arrival rate is one error a day. The results very clearly indicate the adverse impact of latent errors on system MTBF. Even with a mean latency time of just 2 minutes, and a system error arrival rate of 1 per day (which means, an error is injected into a processor once every 8 days, and into a global memory board once every 3.2 days) system MTBF is slashed by more than 50 percent. As the latency times increase, the MTBF is further degraded.

To summarize, the accumulation of latent errors does not cause near coincident errors. Rather, it is a combination of error latency times, system activity and repair policy that cause

Mean Latency in minutes	System MTBF in years
0	6.7 ±0.32
2	3.14 ±0.16
4	2.09 ±0.11
8	1.3 ±0.07
16	0.75 ±0.04
32	0.40 ±0.02
64	0.32 ±0.01

Table 8.3: System MTBF with inter-component dependence.

near coincident errors that adversely impair system reliability. A system like the Integrity S2 is especially vulnerable to latent errors because its on-line re-integration scheme requires that the entire contents of the healthy boards be copied to the newly re-integrated board. Thus any latent error, in any segment of either healthy board, is guaranteed to cause system failure. Other systems with different repair policies may not be as sensitive to latent errors. To properly evaluate and quantify the impact of latent errors, it is imperative that system activity such as repair policies be incorporated into the model.

8.3 Impact of Correlated Errors

TMR systems have been shown to be extremely effective against single, independent errors. In this experiment, correlated errors are injected to determine their impact on system reliability. Inter-component dependence is not modeled in the first two experiments. In the first experiment, errors with zero latency are injected, and of these, 2% are correlated errors (see Figure 7.4). The modeling is similar to the 'partial coverage' technique commonly used with analytical models. The system MTBF was found to be 0.129 (± 0.006) years. Comparing this to an MTBF of 6.76 years (determined when latency is zero and there is no correlation) we see that even a very small fraction of correlated errors causes an order magnitude reduction in the MTBF. However, measurement studies [67, 73] show that correlated failures do not occur simultaneously as modeled here.

Correlated errors with latency are used to mimic the phenomenon of "staggered machine failures" found to be caused by correlated errors [73]. Table 8.4 shows the system MTBF for two

System MTBF in years		
Mean (min.)	Exponential, $C_X = 1$	Normal $C_X = 0.5$
2	0.216 ±0.006	0.158 ±0.008
4	0.334 ±0.011	0.245 ±0.012
8	0.560 ±0.016	0.453 ±0.023
16	0.960 ±0.048	0.825 ±0.041
32	1.67 ±0.083	1.46 ±0.073
64	2.71 ±0.129	2.46 ±0.123
120	3.97 ±0.201	3.65 ±0.182
240	5.36 ±0.266	5.05 ±0.252
480	6.59 ±0.333	6.61 ±0.33
2160	12.06 ±0.6	13.2 ±0.683

Table 8.4: System MTBF for two latency distributions with various means.

latency distributions, exponential and normal, with various means. The standard deviation of the normal distributions were set to 1/2 of the mean. The most important result shown in the table is that when error latency is considered, the degradation in the MTBF due to correlation is not as significant and may not be orders of magnitude less. For instance, when the mean of the latency distribution is 32 minutes, the MTBF is nearly 13 times larger than the case when error latency is not considered. For really large latency times exceeding 16 minutes, there is no longer an order degradation reduction in the MTBF, and for very large latency times, there is an increase in the MTBF. The table also shows an inverse relationship between the degradation in the MTBF and the mean of the latency distribution.

The results in Table 8.4 also show that the normal latency distribution produces smaller MTBF figures. The difference in the MTBF between the two types of distributions is statistically significant for means upto 32 minutes, and in cases are 25% smaller. To determine whether this phenomenon is a characteristic of the normal distribution or due to the fact that the coefficient of variation ($C_x = \sigma/\mu$) is smaller than that of the exponential distribution, the experiment was repeated with different C_x values. Table 8.5 contains the results. Recall that the coefficient of variation for the exponential distribution is always 1. The results in Table 8.5 show that increasing C_x increases the system MTBF. It also shows that even when the C_x of the normal distribution is the same as that of the exponential distribution, its MTBF figures are, for the most part, statistically different, and in this case larger. The reason is that the

System MTBF in years			
Mean (min.)	$C_X = 0.5$	$C_X = 1.0$	$C_X = 2.0$
2	0.158 ±0.008	0.211 ±0.01	0.328 ±0.016
4	0.245 ±0.012	0.368 ±0.018	0.5832 ±0.029
8	0.453 ±0.023	0.649 ±0.032	1.0691 ±0.0538
16	0.825 ±0.041	1.26 ±0.063	1.8015 ±0.09
32	1.46 ±0.073	2.09 ±0.104	2.8108 ±0.139
64	2.46 ±0.123	3.38 ±0.169	4.0841 ±0.203

Table 8.5: System MTBF for the normal latency distributions with varying C_x values.

System MTBF in years		
Mean (min.)	Exponential, $C_X = 1$	Normal $C_X = 1.0$
2	0.126 ±0.006	0.128 ±0.006
4	0.124 ±0.006	0.124 ±0.006
8	0.115 ±0.006	0.114 ±0.006
16	0.11 ±0.005	0.106 ±0.005
32	0.099 ±0.005	0.094 ±0.005
64	0.081 ±0.004	0.073 ±0.004

Table 8.6: System MTBF for the exponential and normal latency distributions with inter-component dependence.

exponential distribution has a larger density of latency times below the mean, thus there is an increased probability that two correlated errors will be detected near coincidentally.

The results of both tables seem to indicate that the larger the latency the better. In fact, in Table 8.4 the system MTBF is nearly doubled (from the case with zero latency) for latency times with a mean of 36 hours. However, recall these experiments do not consider inter-component dependence. Table 8.6 shows the MTBF figures obtained when inter-component dependence is considered. Now it becomes clear that *for this architecture*, the smaller the latency the better the system MTBF.

To summarize, the experiments conducted reveal that simple analytical models that fail to consider error latency exaggerate the impact of correlated errors. If latency is considered, the impact of correlated errors is noticeably less, and it becomes negligible if the latency is very large. In the previous section where correlated errors were not injected, there was no statistically significant difference in the MTBF results produced with the normal and exponen-

tial distributions (see Table 8.1). In this section, for means less than 64 minutes, the latency distribution produce statistically different MTBF figures. Between the exponential and normal distributions, if they both have the same C_X value, the exponential distribution produces smaller MTBF figures. Furthermore, the larger the C_X, the larger the MTBF. What this means to the designer is that if the latency times are very small, correlated errors are more likely to have an adverse impact. If the latency times are very large, correlation is not as important an issue. Typically, however, latency distributions are mixed with a large number of errors with small latency times and a few with very large latency times. The latency times obtained in chapter 6 follows this behavior. With such distributions, their will not be a pronounced increase in the MTBF as the mean of the latency distribution is increased. Furthermore, correlation will noticeably degrade system MTBF.

8.4 Evaluation of Memory Scrubbing

Recall that the Tandem Integrity S2 relies on memory scrubbing to correct latent faults in the CPU and Global memory boards. In this section, experiments are conducted to determine the effectiveness of the existing scrubber. In chapter 6, the software environment was used to determine the application specific coverage of the existing scrubber and that of a proposed "dual" scrubber, for a single processor. In this section, the hierarchical approach, described in chapter 7, is used to extend the application specific analysis to consider the whole system and perform a more complete analysis of the two scrubbing schemes.

Memory scrubbing has been evaluated analytically in [55]. The authors assume that all words in memory are accessed with equal probability and access times are exponentially distributed. They provide an exact upper and lower bound on the MTBF for exponentially distributed scrubbing and an approximate upper bound for deterministic scrubbing. Deterministic scrubbing is shown to provide MTBFs that are twice as large. The approach is very useful when evaluating a single memory system under the assumption of uniform memory access. In our simulation-based study, we consider several memory boards, their different repair times, error latency, inter-component dependence inherent in the architecture and the effect of a specific application running on the system, when evaluating the MTBF. Furthermore, with the simulation-based approach we are also able to provide coverages of the scrubbers. Some of

these aspects can be included in a CTMC model. To understand how much can be accurately captured, a general stochastic Petri-net tool was used to model just the three processors and the memory scrubbing scheme. Issues such as application specific analysis and intercomponent dependence were not addressed. The Petri-net model shown in Figure 8.1 resulted in a 2000 state CTMC for $n = 6$, the maximum number of accumulated latent errors in a board. Table

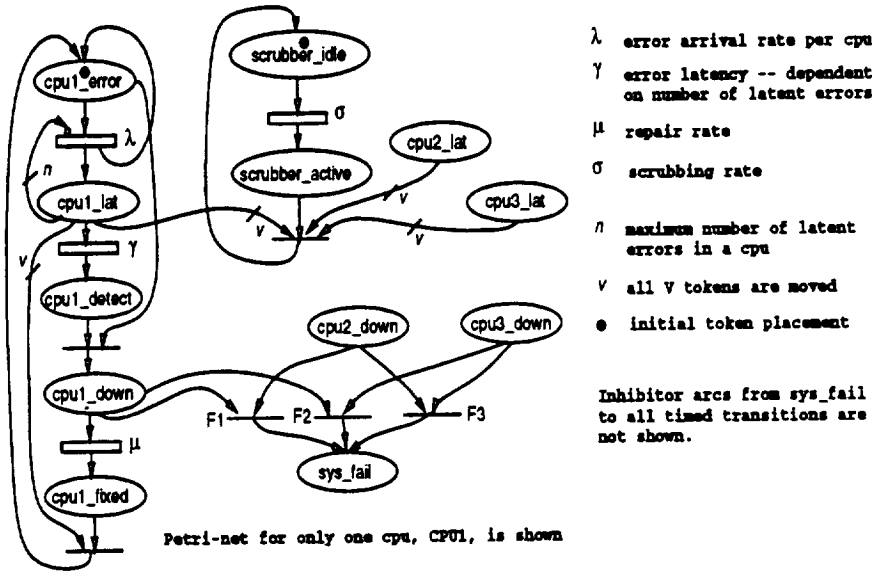


Figure 8.1: Petri-net model of the three CPUS with memory scrubbing.

8.7 shows the MTBF figures obtained with the Petri-net model, a DEPEND simulation model, and the relative difference of the Petri-net model. The large difference in the results of the Petri-net model are due to its inability to model the deterministic progression of the scrubber through the memory and because it assumes an exponential scrubbing rate which does not bound the scrubbing interval to 60 minutes. The DEPEND simulation, however, models the location at which each error is injected, the location of the scrubber at the time of each injection and the exact time the scrubber takes to detect and correct each error. A 5-stage Erlang distribution was used to replace the exponential scrubbing distribution, but this provided only a small improvement in the results at a cost of increasing the number of states to 10,000. Interestingly, because the percent difference varies with the latency, the designer can never be sure how close the model results are to the real values. This example shows that a CTMC can be used to provide a rough estimate of a deterministic process. It also justifies the need for a simulation-based study which can provide more accurate results, permit the use of

non-exponential distributions, represent dependency caused by latency, and allow application specific analysis. What follows is a step by step analysis of the existing single and a new dual scrubber.

Mean Latency (min.)	MTTF (yrs.)		Percent Difference
	DEPEND	Petri-net	
2	6.002	6.46	7.67
4	6.54	6.80	3.98
8	7.59	7.47	1.58
16	11.05	8.9	19.45
32	20.21	12.07	40.28
64	45.19	19.75	56.63
128	141.05	40.34	71.39

Table 8.7: Comparison of MTBF obtained with DEPEND and the Petri-net model.

The effectiveness of the scrubber is intimately tied to the error latency distribution. Figure 8.2 shows the MTBF of the system, with and without scrubbing, for exponentially distributed latency times with various means. The scrubber sweeps through the entire memory every

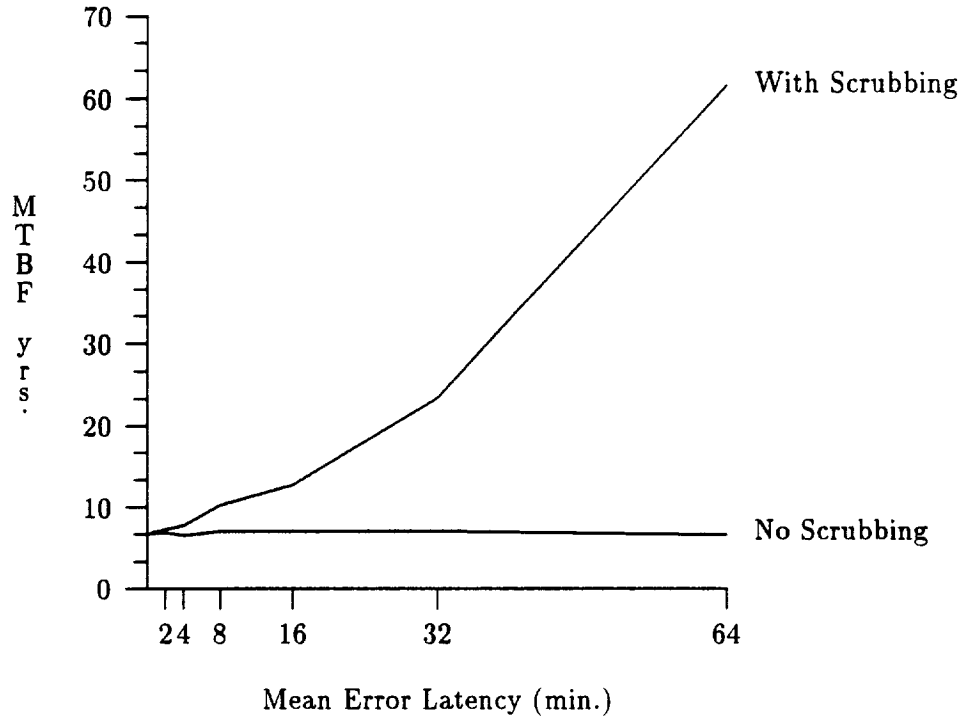


Figure 8.2: System MTBF for various latency distributions (Hourly Scrubbing).

Mean Latency in minutes	Scrubber Coverage
2	0.033
4	0.066
8	0.132
16	0.258
32	0.45
64	0.65

Table 8.8: Scrubber coverage for various exponential error latency distributions.

hour. As the mean of the latency distribution increases, the coverage of the scrubber increases rapidly (Table 8.8). For small latency distributions with means of 2 to 4 minutes, the scrubber improves the MTBF by only 7% from the base case with no scrubber. But for means of 32 and 64 minutes, the scrubber increases the MTBF by 232% and 827%, respectively. Hence the scrubber is most effective against errors with large latency times. These errors mostly reside in unused or seldomly used parts of memory. Errors within actively used parts of memory are typically detected by the process of execution and causes the board containing the error to be shutdown. The ineffectiveness of the scrubber against small error latencies was also found in [76] in which the author used memory traces and simulation to estimate the scrubber's coverage.

Next we simulate the *intercomponent dependence* caused by latent errors is to test the effectiveness of the scrubber. Figure 8.3 shows the MTBF of the system, with and without scrubbing. Without the scrubber, the system MTBF degrades monotonically as the mean of the latency distribution increases. Even with a small mean of 2 minutes[2] there is more than a 50% degradation in the MTBF (3.14 years) when intercomponent dependence is modeled. With hourly scrubbing, the MTBF increases to 3.47 years, and with a scrubbing cycle time of 30 minutes, the MTBF reaches only 3.56 years. As the figure shows, except for the case with a mean latency of 128 minutes and a 30 minute scrubbing cycle time, intercomponent dependence caused by latency reduces the system MTBF significantly. It also illustrates that memory scrubbing, at realistic cycle times that do not impose a noticeable performance overhead, are quite ineffective against this phenomenon. The MTBF curves in the figure, for all three scrubbing periods, do

[2]The smaller the latency time, the smaller the probability of a system failure due to intercomponent dependence caused by latency.

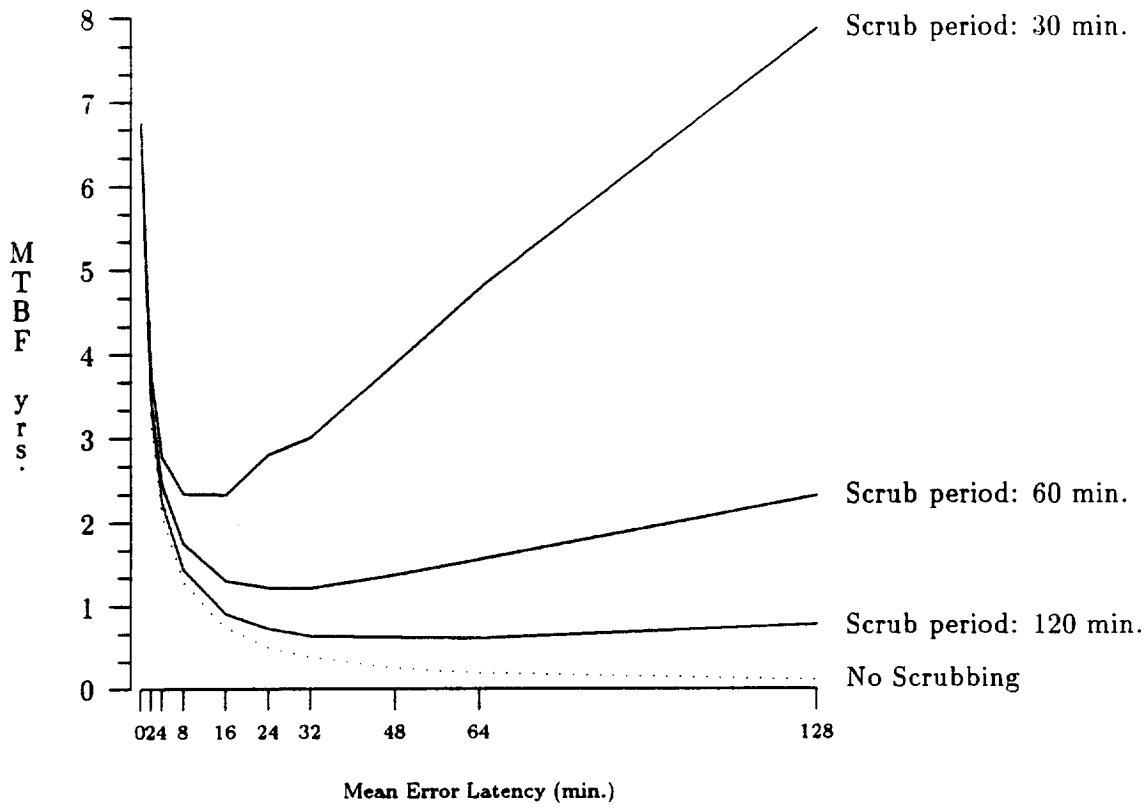


Figure 8.3: System MTBF when inter-component dependence is modeled with hourly scrubbing.

not decrease monotonically. This is due to the trade-off between small and large latency times. Though errors with small latencies are less likely to be corrected by the scrubber they are also less likely to cause a near-coincident errors due to intercomponent dependence. Errors with large latency are more likely to cause near-coincident errors, but are more likely to be detected and corrected by the scrubber. The minimum MTBF occurs for a mean latency that is half the scrubber's cycle time. For example, the smallest MTBF for the scrubber with a 30 minute cycle time occurs for the latency distribution with a mean of 16 minutes. This information is very useful to the designers because if they have some general idea of the latency distribution of the system, they can avoid using a scrubbing period at its minimum effectiveness point.

In chapter 6, an abstract representation of the **Gauss** program was executed to obtain detection latency times and to evaluate two scrubbing schemes. The analysis focussed on the coverage of the existing and the newly proposed 'dual' scrubbing scheme with respect to *active* errors that are detected by the process of program execution. In the experiments that follow,

the analysis of the scrubbers is also extended to consider the rest of the memory and the overall impact they have on system MTBF. Without such a complete analysis it is difficult to determine which scheme is more effective in improving system reliability. As it is not possible to directly perform the detailed evaluation in chapter 6 while also considering the rest of the system, the hierarchical approach described in chapter 7 is used for this extended analysis. Executing several instances of the **Gauss** PCFGs on the three processors while also modeling all the scrubbers, the global memory system and the re-integration process will require astronomically large simulation times. Instead, the execution of the PCFGs is replaced with their collected detection latency times. For the experiments, the CPU memory is logically divided into two segments: the *application space* and the *system space*. The size of the application space is a multiple of the of the size of the memory image of the actual Gaussian elimination program times the number of **Gauss** programs assumed to be executing. The detection latency of the errors injected into the application space is sampled from the collected empirical latency distribution file. For instance, to model the execution of four **Gauss** programs, the latency distribution collected when four **Gauss** PCFGs were executed is used in this simulation. The CPU memory's system space and the global memory are injected with errors having exponentially distributed latency times with a mean of 16 minutes. Both the single scrubber and the dual scrubber are tested under this configuration. The ratio of the overheads of the dual scrubber (see Equation 6.4) versus the single scrubber used are 1, 2, 4, 8, and 16. Recall that the dual scrubber has two scrubbers. One scrubs system space and the other scrubs the application space. By making the system space scrubber operate at a rate four times slower than the single scrubber, it is possible to scrub the application space frequently and still maintain an overhead ratio of 1. The other overhead ratios are obtained by further speeding up the rate of the application scrubber. The question is what happens to system MTBF with this scrubbing configuration?

Figure 8.4 plots the application specific MTBFs obtained with the single scrubber and the dual scrubber for 1, 4 and 16 simultaneous executions of the **Gauss** program. The percent of the CPU memory allocated to the application space and the mean of the empirical distribution which represent the execution of the programs are listed in Table 8.9. The results show that the single scrubber provides similar or better MTBF figures than the dual scrubber. Table 8.10, which shows the coverage of the single scrubber and the dual scrubber with an overhead ratio of 1, indicates why this is the case. Even though the application scrubber's coverage is

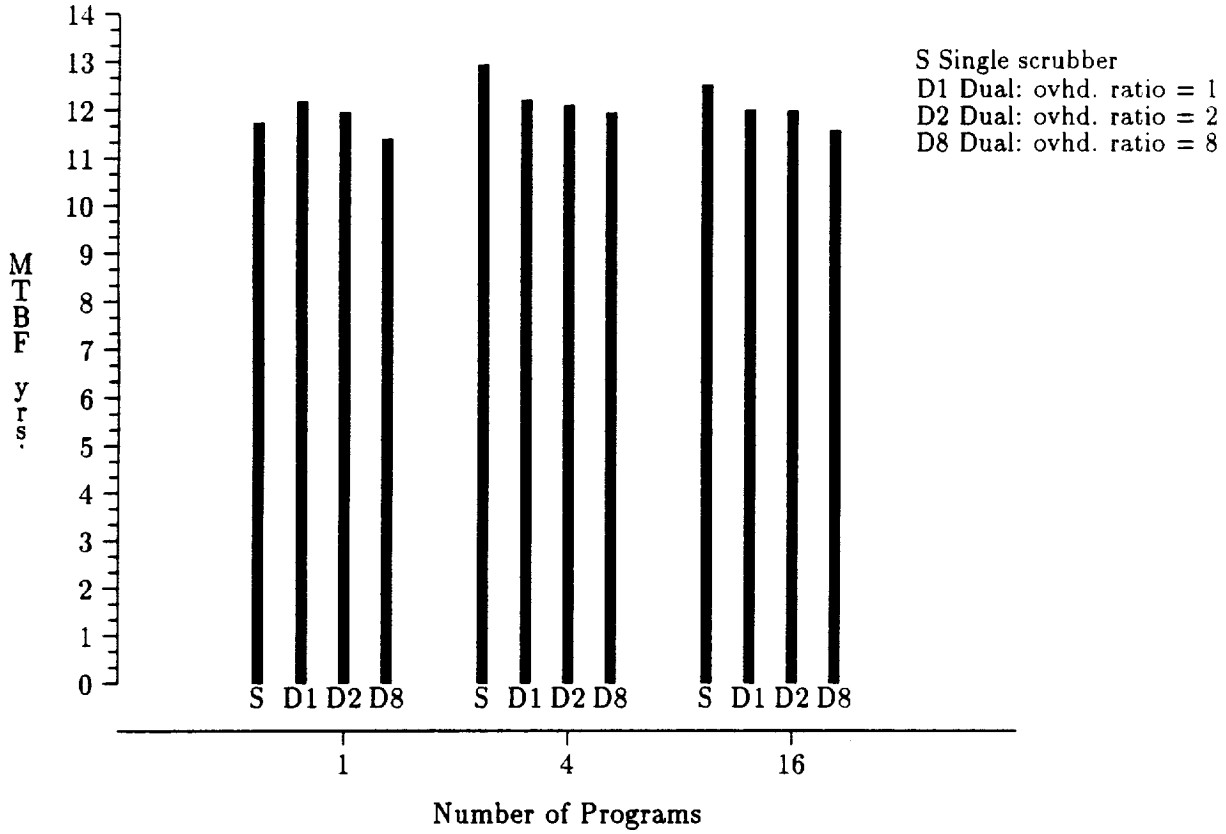


Figure 8.4: System MTBF obtained with the single and dual scrubbing schemes.

Number of Programs	Percent of CPU Memory	Mean Latency in Seconds
1	0.5	8.8
4	1.64	31.34
16	6.6	116.95

Table 8.9: Parameters of the experiment.

No. Programs	Dual Scrubber (Ovhd. = 1)			Single Scrubber
	Appl. Space Coverage	Sys. Space Coverage	Overall Coverage	Overall Coverage
1	0.33	0.066	0.068	0.26
4	0.32	0.067	0.07	0.26
16	0.31	0.066	0.08	0.24

Table 8.10: Coverage of the two scrubbing schemes.

Scrubber	System MTBF in years					
	Application Space (Percent of CPU memory)					
	5%	10%	20%	30%	40%	50%
Single	12.19	12.01	12.56	11.95	12.18	12.76
Dual (Ovhd. = 1)	11.54	12.34	11.96	11.75	12.31	11.85
Dual (Ovhd. = 2)	11.8	12.37	12.23	12.33	11.85	11.76
Dual (Ovhd. = 4)	11.63	11.95	11.86	11.71	11.57	12.13

Table 8.11: System MTBF obtained for various application memory space sizes.

very high, the overall coverage of the dual scrubbing scheme is low because most of the errors are injected into system space, where the scrubbing occurs with a cycle time of 4 hours. As a result, the single scrubber's coverage is on average four times higher.

Next, we arbitrarily increase the size of the application space to see how the scrubbers perform. Application space sizes that are 5, 10, 20, 30, 40 and 50 percent of the total CPU memory are tried. The empirical latency distribution obtained for 4 **Gauss** programs is used for all errors injected into the application space. Table 8.11 contains the MTBF figures obtained for the two scrubbing schemes. In spite of increasing the application space, there is no statistically significant improvement in the MTBF figures obtained with the dual scrubbers. This is because as the size of the application space increases, the scrubbing frequency of the application space scrubber must be reduced to maintain reasonable overheads. Hence, as the application space increases, the coverage of its scrubber decreases thus offsetting any gains.

Since the introduction of the Integrity S2, the memory configuration has changed to improve the performance of the machine. The newer Integrity S2 machines come with larger CPU memory and smaller global memory. The experiments in which we vary the application memory space size is repeated with a configuration where the CPU memory size is 64Mbytes and the

Scrubber	System MTBF in years					
	Application Space (Percent of CPU memory)					
	5%	10%	20%	30%	40%	50%
Single	13.79	12.15	12.17	11.52	10.60	10.18
Dual (Ovhd. = 1)	8.8	8.97	8.73	8.66	8.19	8.3
Dual (Ovhd. = 2)	8.8	8.87	8.77	8.95	8.08	8.62
Dual (Ovhd. = 4)	9.03	9.15	9.16	9.23	8.95	8.73

Table 8.12: System MTBF obtained for various application memory space sizes with the new memory configuration.

global memory size is 16Mbytes. The error arrival rate is kept fixed (for comparative reasons) but the percent of errors injected into the CPU memory has increased to 85% due to its larger size. Table 8.12 lists the MTBF figures obtained with the new memory configuration. With a larger percent of errors injected into the CPU, the difference between the two scrubbers is even more apparent. Here, the dual scrubber produces MTBF values that are up to 25% lower.

To summarize, the evaluation of memory scrubbing shows that it is only effective if the error latency is large. For mean latency times exceeding 30 minutes, hourly scrubbing can improve the MTBF by over 200%. Increasing the frequency of the scrubber can improve its coverage of errors with smaller latency times but at a cost of high performance overhead. Furthermore, as shown in chapter 6, increasing the scrubbing frequency beyond a certain point provides diminishing increase in the coverage. The scrubbing scheme was found to be very ineffective against near-coincident errors caused by error latency. However, if the latency times are very large, the scrubber provides reasonable improvement in the MTBF. When the possibility of near-coincident errors caused by latency is modeled, the MTBF curves no longer increase monotonically with an increase in the mean of the latency times (compare Figures 8.2 and 8.3). The experiments show that the minimum MTBF point for the scrubber occurs as the mean latency time approaches half the scrubber's cycle time.

The entire memory will not display the same detection distribution because it varies with time and with the frequency and nature of use. To model applications exercising the memory, detection latency times obtained from detailed simulations in chapter 6 are used. Portions of the CPU memory, the application space, allocated to the applications are injected with latency times sampled from these empirical distributions. This configuration was then used to evaluate

the existing single and the proposed dual scrubbers from a system perspective. The nature of the Integrity S2's re-integration mechanism makes the dual scrubber ineffective relative to the single scrubber. This is in spite of the fact that it provides orders of magnitude improvement in coverage for the application space. The problem is twofold. First, the application space is small compared to the rest of the memory. Second, to limit the overhead of the dual scrubber, the rest of the memory is scrubbed at a much slower rate than the existing scrubber. By slowing down the scrubbing of the rest of the memory, the system becomes more vulnerable to near-coincident errors. This extended analysis of the scrubbing schemes, possible with the hierarchical technique, made it possible to perform a complete analysis of the scrubbers and quantify their coverage and the system MTBF figures they would produce. With this information, it was realized that the dual scrubber, which seemed so promising with respect to just the application, reduces the dependability of *this* system as a whole.

8.5 Impact of Repair Times

The Integrity S2 is designed to tolerate single faults. For such systems, the time needed to repair a faulty component is referred to as its *window of vulnerability.* If a second fault arrives within this window, the system fails. The CPU re-integration time consists of 60 seconds to perform a power-on self-test (POST) and 1.5 seconds to re-integrate the CPU. The re-integration time cannot be easily reduced but the POST time can be cut by using different self-checking programs. Since most errors are caused by transient faults, reliability can be improved by performing a perfunctory check that takes a few seconds and immediately initiating a re-integration. If another error is detected in the same board shortly thereafter, a more thorough POST program can be executed to check for permanent defects. The re-integration time for the global memory varies with workload, but it can be reduced by increasing the priority of the re-integration process. In this section, simulations are conducted with various POST times and global memory re-integration times to quantify their impact on system MTBF. Specifically, POST times of 10, 20, 30, 40, 50 and 60 seconds and the global memory re-integrations times of 1, 2, 5 and 10 minutes are used in the simulations. Memory scrubbing is not activated and a exponential error latency distribution with a mean of 16 minutes is used.

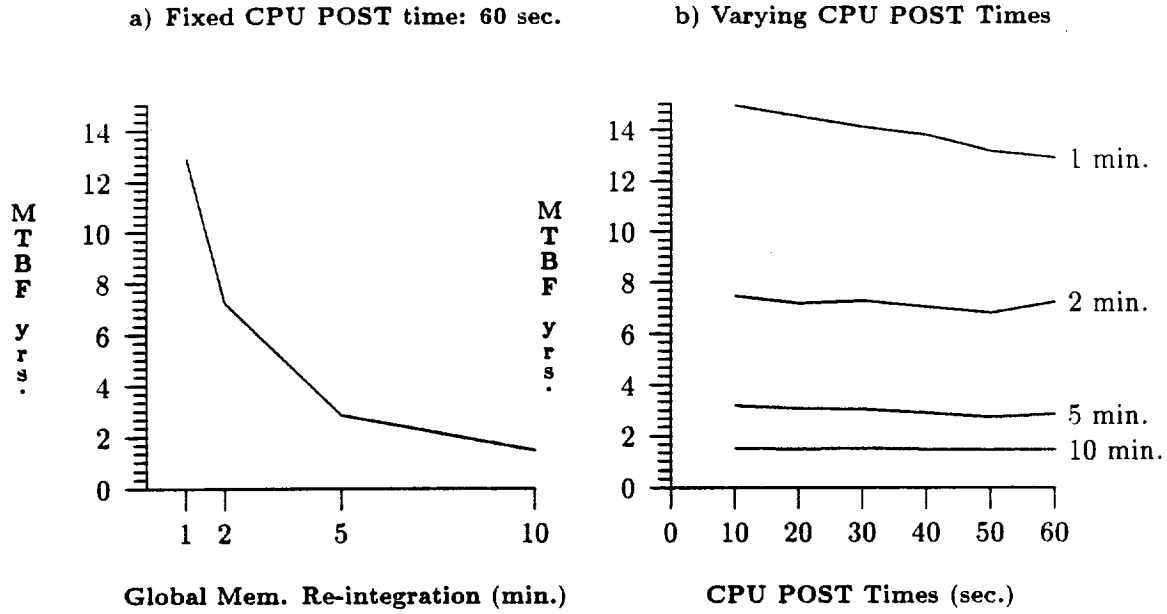


Figure 8.5: System MTBF for various subsystem re-integration times.

The simulations are first executed with the original memory configuration (8MBytes of CPU memory and 32Mbytes of global memory). With this configuration, where most of the errors are injected into the larger global memory, the global memory time is the dominant factor. Figure 8.5a shows that the system MTBF decreases monotonically as the global memory re-integration time is increased. Since the global memory is re-integrated on the fly, its re-integration time varies depending on the workload on the system (see Table 5.1). Figure 8.5a can also be interpreted to reflect the system MTBF for varying workloads. As the workload increases, the reliability of the system decreases. Figure 8.5b illustrates what little impact changing the CPU time has on system MTBF. This is especially the case as the global memory re-integration time is increased. Clearly, for this memory configuration, the global memory re-integration time is the bottleneck.

The experiments are repeated in which the inter-component dependence due to latency is modeled. The system MTBF for mean latency times of 16 and 8 minutes are listed in Table 8.13. Note that when the dependence is modeled, varying the global memory re-integration time does not have as significant an effect on system MTBF. This is because, for the Integrity S2, each latent error injected into the system creates a window of vulnerability. For instance, if a latent error X is injected into CPUA, an error detected in the two other CPUs before X is corrected will cause a system failure. So each latent error creates a window of vulnerability.

MTBF in years		
Global Memory Re-int Times (min.)	Mean Latency 8 minutes	16 minutes
1	1.47	0.76
2	1.3	0.74
5	1.0	0.63
10	0.79	0.54

Table 8.13: System MTBF with modeling of near-coincident errors.

If the error latency is large, they more so than the POST and global memory re-integration times, become the dominant factor that determine system reliability.

The first experiment was repeated with the new memory configuration in which each CPU contains 64Mbytes and each global memory has 16Mbytes. For ease of comparison, the error arrival rate was not varied, however, the ratio of errors injected into the CPU and global memory were changed to reflect their change in relative sizes. With this configuration, 85% of the errors are injected into the CPUs. Figure 8.6 shows that now the CPU POST time is the more dominant factor. Varying the global memory re-integration time does not have as dramatic an affect as seen in Figure 8.5a. Comparing Figures 8.5b and 8.6b, one sees that the system is more reliable with the new configuration. Given two systems with the same amount of total memory (CPU and global memory), the system which has apportioned more memory to the CPUs will have a higher reliability. This is because CPU re-integration takes less time and is not done in the background. Of course, global memory re-integration can be performed at a high priority like the CPU, but since the system also performs better when the local CPU memory is large it makes more sense to re-apportion the memory. Part of the reason for the higher performance is that access time to the local memory is smaller and there is no synchronization overhead to be paid.

To summarize, reducing the global memory re-integration time and the CPU POST time can improve system MTBF. Which re-integration time to reduce depends on the system configuration, with the component with the highest error arrival rate being the reliability bottleneck. Since CPU re-integration times are smaller than global memory re-integration times, apportioning more memory to the CPUs improves system reliability. It also improves its performance. Experiments in which inter-component dependence caused by latency is modeled show that

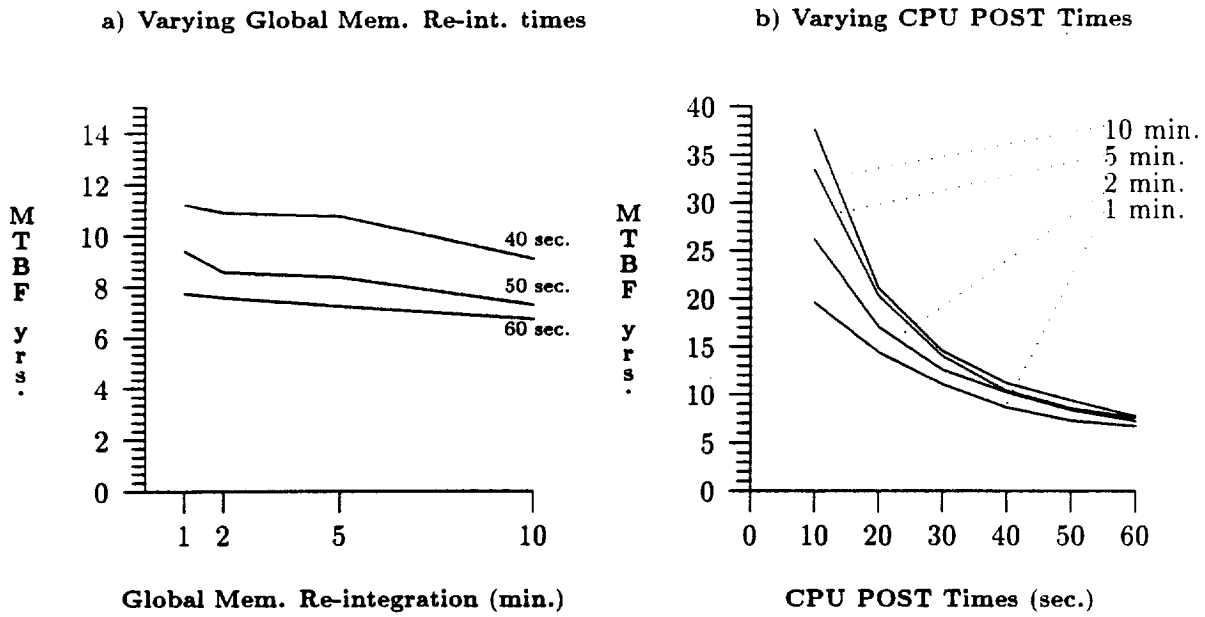


Figure 8.6: System MTBF for various subsystem re-integration times – New Memory Configuration.

error latency creates a window of vulnerability for this architecture, and this has the most dominant affect on system reliability. When modeling inter-component dependence, reducing CPU and global memory re-integration times does not noticeably improve system MTBF.

Chapter 9

Conclusion

The contribution of this thesis is the development of a methodology for functional simulation-based system level dependability analysis of fault-tolerant computer systems. Simulation-based approaches have mostly been used at lower levels, for example at the gate level where the functionality of the basic components are simple and their interconnections and inter-dependencies are well defined. At the system level, simulation-based approaches have been either Petri-net based extensions of analytical tools or have been designed to analyze only a small set of architectures. Our proposed methodology is unique in that it addresses many of the difficulties and issues that have limited the applicability and utility of these existing simulation-based approaches. The proposed methodology is designed to be generally applicable to a wide variety of architectures and can model various fault conditions. It permits detailed functional modeling of architectural features such as sparing policies, repair schemes, routing algorithms as well as other fault-tolerant mechanisms. The approach permits actual execution of application software for more realistic modeling. In addition, it allows the abstract representation of software algorithms when actual code does not exist. Through this abstraction it is possible to study the behavior and effect of the software on hardware faults and perform application specific analysis of system level fault-tolerant mechanisms in the early design phase. Finally, unlike any other existing simulation-based approach, an acceleration technique is proposed and implemented that makes it possible to study fault-tolerant systems for extended periods of time.

The effectiveness of any methodology is best determined when actually implemented and tested. The methodology has been incorporated into a software tool called DEPEND. Since

its development, the tool has been used (and is being used) to analyze the Tandem Integrity S2 system, the CM-5 Connection Machine, a distributed system executing a load balancing algorithm, the computing element of the Hubble Telescope, a proto-type switching system developed by Raynet, the Parsytec distributed computer being developed in conjunction with the European Esprit Project and the Data Management System of the Space Station. In addition, the tool has been ported to Tandem Inc., Thinking Machines, Raytheon Company, IBM Federal Systems, NASA Langley, and NASA Ames for the purposes of evaluation and use in studying prototype designs.

The section below summarizes the main facets of the methodology and the experiments used to illustrate and validate the features of DEPEND. The thesis ends with a section that suggests possible extensions of this work.

9.1 Summary

Issues that impede the development of a general-purpose, functional simulation-based system level dependability analysis tool include:

- Modeling a large variety of components.
- Coping with the large fault model domain.
- Reducing model development time and model complexity.
- Incorporating software behavior under faults.
- Reducing simulation time explosion.

9.1.1 The Approach

Modern software engineering techniques, and in particular the object-oriented design paradigm are used to tackle the first three issues. Two somewhat competing criteria of *modular decomposition* and *composition* along with inheritance and reusability are used to develop a structured class hierarchy. This hierarchy provides a skeletal framework that can be readily customized to create detailed simulation models of a wide variety of fault-tolerant architectures for fault injection and dependability studies. The key idea was to encapsulate general mechanisms in

a set of fundamental objects and through inheritance and clientship allow them to drive other objects that simulate the architecture specific behavior. Thus a set of fundamental objects was developed that facilitates and automates the development of simulation models. In addition, a default set of objects that model many common architectures was created to further simplify the development of simulation models. These fundamental and architecture specific objects can in turn be inherited and developed to create still more elaborate objects.

Four basic fault models are provided which can be used to simulate specific fault behaviors: status faults, process faults, server faults and data faults. A fault injector is used to inject faults based on commonly used distributions: exponential, Weibull and constant. In addition, user specified empirical distributions are also supported. The injector uses a fast table-based injector to keep track of all components, their status and the time of their next injection. It also automatically takes component aging into account and averts the problem of modeling local and global times found in most analytical tools. A workload-based injection scheme has also been developed. It varies the rate of injection based on the load on the system and permits modeling of the workload/failure relationship found by other researchers.

Two examples are used to highlight the benefits of the functional simulation tool. Unlike analytical and Petri-net tools, a system's fault behavior does not have to be pre-defined with a set of probabilities. Rather, using detailed modeling of the architecture and its system software, the failure modes of the system can be determined using the tool. In one example, actual system software is executed and a particular design feature that made it very susceptible to a particular fault type was identified.

9.1.2 Software Modeling

DEPEND addresses the need to study the effect of software under hardware faults and to perform application specific analysis. This is especially important to designers because the effectiveness of many system level detection and fault tolerance mechanisms depend on the class of applications running on the system. DEPEND allows the execution of actual software on a simulated hardware platform, and it provides a software model that can be used to evaluate the behavior and the effect of software on hardware faults. The model represents application programs by decomposing them into *graph models* consisting of a set of nodes, a set of edges that probabilistically determine the flow from node to node, and a mapping of the nodes to memory.

The software model simulates the execution of the programs while errors are injected into their memory space. The result provides application dependent parameters such as detection and propagation times. The model is especially useful in the early design stages because it allows designers to make application dependent evaluation of function and system level error detection and recovery schemes.

One use of the software model was illustrated with a case study. The model was used to obtain error detection latency times of the **Gauss** and **sort** programs running on a Tandem Integrity S2 system and to evaluate the coverage of two memory scrubbing schemes. The applicability of the model for different programs was evaluated by studying its sensitivity to the detection parameter ρ_t. Several detection equations, which model the low level detection process of a CPU, were derived empirically from studying the behavior of the actual machine under faults. Of these, the EXP detection equation was shown to be applicable to both **Gauss** and **Sort** which use widely different sets of instructions. We feel that this detection equation is generally applicable to most compute bound programs running on RISC processors. Error detection latency times obtained with the model were validated with measurements from an actual Integrity S2 system. Formulae which were derived to estimate application dependent *active coverage* values of the scrubbing schemes were verified with the software model. The application dependent coverage values obtained with the model were compared with those obtained via traditional schemes that assume uniform or ramp memory access patterns. For the **Gauss** program, some coverage values obtained using the traditional approach were found to be more than 100% larger than those obtained with the software model. This result emphasizes the importance of accurate application specific evaluation – especially when evaluating the dependability of application specific systems.

9.1.3 Acceleration Technique

Unlike any other simulation-based dependability analysis tool, DEPEND provides an acceleration technique to reduce simulation time explosion. The acceleration technique proposed uses a unique combination of three schemes to provide speed-up. It uses hierarchical simulation, a time acceleration algorithm, and hybrid simulation. First, detailed functional simulations of segments of the system are executed to obtain statistical models that represent and characterize their behavior. These statistical models are then used by higher level simulations that simulate

and analyze larger segments of the system. Speed-up is achieved with this hierarchical approach through a time acceleration algorithm. The time acceleration algorithm, used by the higher level simulations, samples from the statistical models to determine the time of the next event (represented by the statistical model) *apriori*. With this apriori knowledge, the simulation leaps forward to a point in time just before the event and resumes detailed simulation. Once the effect of the event has subsided, it leaps forward to the next chronological event. To achieve further speed-up, a hybrid simulation approach is used. Using the notion of variable aggregation and decomposition, the entire dependability simulation is divided into two submodels: the failure occurrence submodel and the repair submodel. Either or both submodels are simulated in detail while statistical models to represent them are collected. Then, the statistical models are used to drive either a Monte Carlo simulation or a Markov or Semi-Markov model to obtain solutions for the entire system. The statistical models are typically distributions such as detection latency distributions or propagation time distributions. But they may also be sets of probabilities such as a fault dictionary used to represent the failure modes of an application software.

The proposed acceleration technique differs from those used by others in that

- It does not rely solely on analytical techniques to reduce simulation time explosion.
- It is designed to work with detailed functional simulations.
- It is widely applicable and does not impose any particular solution method. As such, it caters to the general design philosophy of DEPEND.
- It can be used in conjunction with other acceleration techniques such as importance sampling.

The acceleration technique was illustrated with a case study of the Tandem Integrity S2. Using hierarchical simulation and time acceleration techniques, error latency distributions were obtained. These distributions were validated with measured distributions collected from the actual machine. The empirical distributions were then used in a hybrid model of the entire system to obtain system MTBF figures. The results obtained were compared with those from a simulation model that did not use the hybrid acceleration technique. The results from the hybrid model were found to fall within the 99% confidence interval of the non-accelerated sim-

ulation. Where the original simulation took 6 to 36 hours to generate results, the hybrid model required only 50 seconds to 7 minutes.

9.1.4 Analysis of the TMR-based System

The DEPEND tool, including the software environment and the acceleration technique, is used to analyze a TMR system that is based on the design of the Tandem Integrity S2. The purpose of the study was to investigate issues which include: correlated errors, accurate modeling of correlated errors, latent errors, inter-component dependencies during automatic repair, memory scrubbing heuristics, application specific analysis of scrubbing schemes, impact of repair times, impact of different configurations, and isolation of dependability bottlenecks. This study illustrates many of the capabilities of DEPEND in a realistic setting. A few of the findings are now summarized:

- Under the assumptions of the study, the accumulation of latent errors does not, in and of itself, increase the probability of near-coincident errors. But once the inter-component dependence imposed by the Integrity S2's architecture and its re-integration scheme is modeled, error latency can reduce system MTBF by orders of magnitude, with the system MTBF decreasing monotonically with increasing latency times.

- Simple analytical models that fail to consider error latency exaggerate the impact of correlated errors. Correlated errors, modeled in a similar fashion to the 'partial coverage' technique commonly used with analytical tools were shown to reduce system MTBF by orders of magnitude. However, once correlated *latent* errors were injected to mimic the phenomenon of "staggered machine failures" found to be caused by correlated errors [73], the impact of correlation was noticeably less and became negligible when the error latency was very large. Hence, the impact of correlated errors depends on the error latency times associated with the errors.

- The distribution of the latency times for correlated errors has a bearing on system reliability. Errors with exponentially and normally distributed latency times with various means were tried and found to produce statistically significant differences in system MTBF. A normal distribution with the same mean as an exponential distribution, but with a

coefficient of variation $C_x = 2$, was found to produce MTBF figures that were nearly double in size. Hence, the specific latency distributions used is important and can produce significantly different results.

- Memory scrubbing was found to be effective only if the error latency is large. For mean latency times of 2 to 4 minutes, hourly scrubbing increases system MTBF by only 7%. For mean latency times exceeding 30 minutes, hourly scrubbing can improve the MTBF by over 200%. The MTBF curve increases monotonically as the mean of the latency distribution increases.

- The scrubbing scheme was also found to be very ineffective against near-coincident errors caused by error latency. However, if the latency times are very large, the scrubber provides reasonable improvement in the MTBF. When the possibility of near-coincident errors caused by latency is simulated, the MTBF curves no longer increase monotonically with an increase in the mean of the latency times. There is a dip in the MTBF curve with the minimum MTBF point occurring at the point where the mean latency time is equal to half the scrubber's cycle time.

- The acceleration technique was used to perform application specific analysis of the single scrubber and the proposed dual scrubber while executing one or more instances of the **Gauss** program. The dual scrubber was found to reduce system reliability compared to the single scrubber in spite of the fact that its coverage of errors injected within the application space is orders of magnitude higher. This is a function of the Integrity S2 architecture and its re-integration scheme. On another machine the dual scrubber may improve the reliability.

- Reducing global memory re-integration time and CPU POST time can improve system MTBF. However, simulations show that reducing repair time of only the component with the highest error arrival rate is necessary. Reducing the other component's repair time has negligible impact on system MTBF. Which re-integration time to reduce depends on the system configuration. With the original configuration which had large global memories, the global memory re-integration time was the dominant factor. With the new configuration containing large local memory, the CPU POST time is the dominant factor.

- Experiments modeling inter-component dependence caused by latency show that error latency creates a window of vulnerability for this architecture, and has the most dominant affect on system reliability. When inter-component dependence is considered, reducing CPU and global memory re-integration times did not noticeably improve system MTBF for the error arrival rate used.

9.2 Future Extensions

The methodology and design philosophy incorporated into the existing DEPEND simulation tool provides the foundation upon which significant future extensions can be built. Future extensions that will enhance the state-of-the-art in functional simulation-based system level dependability analysis can be divided into three categories: compiler-based enhancements, complete fault injection and debugging environment and graphical interface.

The current implementation of DEPEND is essentially a super set of C++. It was specifically designed this way because a special compiler did not have to be developed thus allowing rapid implementation and testing of the proposed methodology. However, by designing a specific compiler it is possible to embed intelligence as well as features that will allow the implemention of two major capabilities. Recall that DEPEND is a process based simulation tool. Process based simulation facilitates developing large detailed simulation models but has the drawback of a very large context switch overhead. This is especially true for the Sparc architecture which has a large register file that must be copied out and in for each context switch. Using a special compiler, it is possible to translate a process based simulation model into an event-driven simulation model. This approach will then provide the ease of programming without the context switch overhead. Preliminary analysis with simple models has shown that this approach is very promising and can attain 60 times speedups [3]. As mentioned in chapter 2, the crux of parallel and distributed simulation lies partly in the way in which processes (or co-routines) are assigned to the various processors. Yet, this issue is seldom addressed in the distributed simulation literature. Using a special compiler that is cognizant of the functionality of DEPEND objects, the DEPEND communication paradigms, and also aided by compiler directives, it will be possible to make intelligent placement decisions to substantially reduce the communication messages used by the processors to maintain a synchronized global clock. The compiler's placement decisions

will most likely be specific to a synchronization scheme. An existing synchronization scheme, aided by the intelligent placement algorithm may make the approach less application sensitive and thus capable of providing good speedup across a large domain of simulation models.

In its present state, a DEPEND simulation model can execute actual C++ and C application programs as a part of the simulation and corrupt any data values using the *data* fault type described in chapter 3. The tool can be made more useful if the entire software image of the application program, its static data space, text space, stack space and its heap space is corruptible. Since random corruption of a program's software image (which is ultimately just a co-routine of DEPEND) can cause the co-routine and hence DEPEND simulation to abort, an approach to capture and handle exceptions is needed. This must be coordinated by another process that operates independently and outside of the DEPEND simulation. C++ provides some exception handling facilities but they are still in their infant stages. Hence, standard operating system signal handling routines will need to be used. While existing tools such as FERRARI [36] and FIAT [18] already provide such an injection capability, implementing it *within* a simulation environment provides a significant advantage. The application program is executed on a simulated hardware platform rather than an actual one making it possible to test the software on various types of architectures (e.g. single processor, shared memory multiprocessor, distributed system) with minimal effort. For example, an actual operating system can be executed and tested on a single processor initially and then executed on a distributed system, by simply changing the underlying simulated hardware platform. With little additional work the injection environment can be extended to permit debugging. Note that since both the DEPEND simulation model as well as the application programs executed within the simulation environment are just DEPEND processes, both can be debugged and tested under normal operating conditions (that is without fault injection).

A graphical user interface can significantly facilitate the entry of simulation models. Currently, C++ programming is used to develop simulation models. As the system becomes large and complex, faster and more efficient input schemes are necessary so that the models can be developed with minimal effort and within a reasonable period of time. A major issue is how to provide a user friendly data entry environment without losing the power of C++ programming. A key to this problem will be to develop a methodology for explicitly specifying the physical architecture of a system, the component inter-connections and their inter-dependencies. By

inter-dependencies, we mean information such as what components fail when a given component is shutdown and what components must be healthy in order to initiate a repair of a faulty component. Currently, system interconnections and the physical architecture are specified implicitly via the functionality of the simulation model. Inter-dependencies are specified with "notification functions", which though powerful are tedious and difficult to use for large complex systems.

All three areas are fundamental research topics that can substantially enhance the state-of-the-art in functional simulation-based system level dependability analysis and help to make such a tool useful to industry.

Appendix A

Automation of the Acceleration Technique

The object-oriented paradigm facilitates the automation of the acceleration technique described in chapter 7. Some objects used to automate the process are presented and described. Only objects that are used to collect statistical models consisting of distributions are shown. Figure A.1 shows the fundamental class **store_dist** used to collect a distribution. The method **record()** is used to store (collect) samples obtained from a simulation model. The remaining methods return commonly required statistics of the distribution. Method **get_sample()**, randomly samples from the distribution and returns a value. This process involves creating a cumulative distribution, using an inverse mapping [40] and a binary search.

The **store_dist** class is the fundamental class used to develop statistical model classes consisting of one or more distributions. Figure A.2 shows how it is used to define a class, **stat_model2**, that contains two distributions. The failure and repair submodels are developed from the statistical model classes. The user inherits a class that defines a statistical model they need and then adds a method that contains the functional simulation that models the failure or repair process. Figure A.3 shows the skeletal framework of a derived class that inherits **stat_model2** and defines the failure process. The failure submodel depicted simulates the failure occurrence process and records the failure times of the CPU and global memory. When enough samples have been collected, the **finish()** routine is invoked to wake up any process awaiting its completion. The repair submodel is not shown. It is developed in an identical fashion. That is, it

```
class store_dist {
    store_dist();
    store_dist(int num);          // number of samples stored
    void init(int num);           // number of samples stored
    void record(double x);        // store sample point x
    double mean();                // return mean
    double median();              // return median
    double max();                 // return maximum
    double min();                 // return minimum
    double std();                 // return standard deviation
    double get_sample();          // randomly sample from distribution
};
```

Figure A.1: Definition of class used to collect distributions.

```
class stat_model2 {
private:
    store_dist x,y;                          // Each stores a distribution
public:
    stat_model2(int numx, int numy);         // initialize each collector
    void record(int type, double value);     // place value in to x or y based on type
    double mean(int type);                   // return mean
    double median(int type);                 // return median
    double max(int type);                    // return maximum
    double min(int type);                    // return minimum
    double std(int type);                    // return standard deviation
    double get_sample(int type);             // randomly sample from distribution
    void finish();                           // collection is finished - wakes up sleeping processes
    void sleep();                            // sleep until collection is done
};
```

Figure A.2: Definition of statistical model consisting of two distributions.

```
class failure_model : public stat_model2 {
    void run_failure_model();
};

void failure_model::run_failure_model()
{
        .
        .
    while(!done) {
            .
    record(cpu, fail_time);
            .
    record(global_memory, fail_time);
            .
    }
    finished();
}
```

Figure A.3: Definition of statistical model consisting of two distributions.

inherits a statistical model class and defines the repair submodel.

The Monte Carlo program which uses the statistical models obtained from the failure and repair submodels is shown in Figure A.4. The program starts each submodel and then sleeps until both have completed. It then executes the Monte Carlo simulation program that models the Markov model in Figure 7.9. Only the logic used to determine the transition from the first state, $0C0G$, is shown.

The DEPEND objects provide all the mechanism needed to collect and sample from the statistical models and use them in Monte Carlo simulations. The user supplies the definition of the failure and repair submodel and the Monte Carlo program. Though the example shows an example of hybrid simulation, the same objects are used with hierarchical simulation.

```
main()
{
    failure_model X;
    repair_model Y;

    Run each submodel and collect statistical model
    X.run_failure_model();
    Y.run_repair_model();
    X.sleep();
    Y.sleep();

    Monte Carlo model of whole system
    state = 0C0G;
    while(!done) {
        switch(state) {
            case 0C0G: cpufailtime = X.get_sample(cpu);
                       gmemfailtime = Y.get_sample(gmem);
                       if(cpufailtime ¡ gmemfailtime) {
                           state = 1C0G;
                           time += cpufailtime;
                       } else {
                           state = 0C1G;
                           time += gmemfailtime;
                       }
                       .
                       .
                       .
        }
    }
}
```

Figure A.4: The Monte Carlo program that uses the failure and repair submodel.

Bibliography

[1] J. Arlat, Y. Crouzet, and J. Laprie. Fault-injection for dependability validation of fault tolerant computing systems. In *Proc. 19th Int. Symp. Fault Tolerant Computing, Chicago, IL*, Jun. 1989.

[2] R. L. Bagrodia, K. M. Chandy, and J. Misra. A message-based approach to discrete-event simulation. *IEEE Trans. on Software Eng.*, SE-13(6):654–665, Jun. 1987.

[3] J. Barnette, Mar. 1993. Private communications with Mr. Barnette during his preliminary analysis.

[4] S. J. Bavuso, J. B. Dugan, K. S. Trivedi, E. M. Rothman, and W. E. Smith. Analysis of typical fault-tolerant architectures using HARP. *IEEE Trans. on Reliability*, R-36(2):176–185, Jun. 1987.

[5] G. Booch. *Object Oriented Design.* Benjamin Cummings, 1991.

[6] R. Breu. *Algebraic Specification Techniques in Object Oriented Programming Environments.* Lecture Notes in Computer Science. Springer-Verlag, 1991.

[7] X. Castillo and D. Siewiorek. A workload dependent software reliability prediction model. In *12th Int. Symp. on Fault-Tolerant Computing*, Jun. 1982.

[8] R. Chillarege and N. S. Bowen. Understanding large system failures – a fault injection experiment. In *Proc. 19th Int. Symp. on Fault-Tolerant Computing*, pages 356–363, Jun. 1998.

[9] R. Chillarege and R. K. Iyer. Measurement-based analysis of error latency. *IEEE Trans. on Computers*, C-36(5), May 1987.

[10] G. Choi and R. K. Iyer. Fault dictionary stuff. In *23rd Int. Symp. on Fault Tolerant Computing*, Jun. 1993.

[11] G. S. Choi, R. K. Iyer, and V. Carreno. Focus: An experimental environment for validation of fault tolerant systems: A case study of a jet engine controller. In *IEEE Inter. Conf. on Computer Design (ICCD)*, pages 561–564, Oct. 1989.

[12] J. A. Clark and D. K. Pradhan. React: A synthesis and evaluation tool for fault-tolerant multiprocessor architectures. In *Proc. Ann. Reliability and Maintainability Symp.*, pages 428–435, 1993.

[13] P. J. Courtois. Decomposability, instabilities, and saturation in multiprogramming systems. *Communications of the ACM*, 18(7):371–377, Jul 1975.

[14] E. W. Czeck. *On The Prediction of Fault Behavior Based on Workload.* PhD thesis, Carnegie Mellon University – Dept. of Electrical Eng., April 1991.

[15] M. Devarakonda and D. Ferguson. Coroutines in process-oriented simulation laguages: Implications of multiple stacks implementation. In *5th International Conference on Modeling Techniques and Tools*, Feb. 1990.

[16] J. B. Dugan and K. S. Trivedi. Coverage modeling for dependability analysis of fault-tolerant systems. *IEEE Trans. on Computers*, 38(6):775–787, Jun. 1989.

[17] A. Dupuy, J. Schwartz, Y. Yemini, and D. Bacon. NEST: A network simulation and prototyping testbed. *Communications of the ACM*, 33:64–74, Oct. 1990.

[18] Z. Segall et. al. Fiat - fault injection based automated testing environment. In *Proc. 18th Int. Symp. Fault-Tolerant Computing*, pages 102–107, Jun. 1988.

[19] R. Geist and K. Trivedi. Reliability estimation of fault-tolerant systems: Tools and techniques. *IEEE Computer*, 23(7):52–61, Jul. 1990.

[20] K. K. Goswami, M. Devarakonda, and R. K. Iyer. Prediction-based dynamic load-sharing heuristics. *IEEE Trans. on Parallel and Distributed Systems*, Jun. 1993.

[21] K. K. Goswami and R. K. Iyer. Depend: A design environment for prediction and evaluation of system dependability. In *9th Digital Avionics Systems Conference*, pages 87–92, Oct. 1990.

[22] K. K. Goswami and R. K. Iyer. *The DEPEND Reference Manual.* University of Illinois – Center for Reliable and High Performance Computing, Urbana, Illinois 61801, Oct. 1990.

[23] K. K. Goswami and R. K. Iyer. DSIM: A distributed simulator. Technical Report NCA-2-385, NASA Ames Research Center, May 1990.

[24] K. K. Goswami and R. K. Iyer. DEPEND: A simulation-based environment for system level dependability analysis. Technical Report CRHC Report #92-11, CRHC – University of Illinois, Jun. 1992.

[25] K K. Goswami and R. K. Iyer. Simulation of software behavior under hardware faults. In *Proc. of the 23rd Inter. Symp. on Fault-Tolerant Computing, Toulouse*, pages 218–227, Jun. 1993.

[26] K. K. Goswami, R. K. Iyer, and M. Devarakonda. Load-sharing based on task resource prediction. In *22nd Hawaii Int. Conf.*, pages 921–927, Jan. 1989.

[27] J. Gray. A census of tandem system availability between 1985 and 1990. *IEEE Trans. on Reliability*, 39(4):409–418, 1990.

[28] J. M. Hammersley and C. C. Handscomb. *Monte Carlo Methods.* Methuen and Co. LTD., London, England, 1964.

[29] M. C. Hsueh, R. K. Iyer, and K. S. Trivedi. Performability modeling based on real data: A case study. *IEEE Trans. on Computing*, 37(4), Apr. 1988.

[30] IEEE Press. *IEEE Standard VHDL Language Reference Manual*. std 1076-1987 edition, 1988.

[31] R. K. Iyer, S. E. Butner, and E. J. McCluskey. A statistical failure/load relationship: Results of a multicomputer study. *IEEE Trans. on Computers*, SE-8:354–370, Jul. 1982.

[32] D. Jefferson and H. Sowizral. Fast concurrent simulation using the time warp mechanism. In *Proc. of the SCS Distributed Simulation Conf.*, San Diego, Jan. 1985.

[33] D. R. Jefferson. Virtual time. *ACM Trans. Programming Language System*, 7(3):404–425, Jul. 1985.

[34] D. Jewett. Integrity S2: A fault-tolerant unix platform. In *Proc. 21st Int. Symp. Fault-Tolerant Computing*, Jun. 1991.

[35] A. M. Johnson and M. A. Schoenfelder. Rainbow net analysis of VAXcluster system availability. *IEEE Trans. on Reliability*, Jul. 1991.

[36] G. Kanawati, N. Kanawati, and J. Abraham. FERRARI: A fault and error automatic real-time injector. In *Proc. 22nd Int. Symp. Fault-Tolerant Computing*, Jul. 1992.

[37] H. Kobayashi. *Modeling and Analysis: An Introduction to System Performance Evaluation Methodology Simulation Modeling and Analysis*. Addison-Wesley Publishing Co., 1978.

[38] J. Lala. Fault detection isolation and reconfiguration in ftmp: Methods and experimental results. In *5th AIAA/IEEE Digital Avionics Systems Conference*, pages 21.3.1–21.3.9, 1983.

[39] J. Laprie. Dependability evaluation of software systems in operation. *IEEE Trans. on Software Engineering*, SE-10:701–714, Nov. 1984.

[40] A. M. Law and W. D. Kelton. *Simulation Modeling and Analysis*. McGraw Hill Book Company, 1982.

[41] E. E. Lewis, F. Boehm, C. Kirsch, and B. P. Kelkhoff. Monte carlo simulation of complex system mission reliability. In *Proc. Winter Simulation Conf.*, pages 497–504, 1989.

[42] N. A. Lynch and M. J. Fischer. On describing the behavior and implementation of distributed systems. *Theoretical Computer Science*, 13:17–43, 1981.

[43] M. H. MacDougall and J.S. McAlpine. Computer simulation with aspol. In *Symposium on the Simulation of Comp. Sys., ACM/SIGSIM*, pages 93–103, 1973.

[44] A. Mahmood and E. J. McCluskey. Watchodg processors: Error coverage and overhead. In *Proc. 15th Int. Symp. Fault-Tolerant Computing*, pages 214–219, Jun. 1985.

[45] J. H. McGough, F. L. Swern, and S. Bavuso. New results in fault latency modelling. In *16th Annual Electronics and Aerospace Conf.*, pages 299–306, Sep. 1983.

[46] B. Meyer. *Object-oriented Software Construction.* International Series in Computer Science. Prentice Hall, 1988.

[47] J. F. Meyer and L. Wei. Influence of workload on error recovery in random access memories. *IEEE Trans. on Computers*, C-37:500–507, Apr. 1988.

[48] G. Miremadi, J. Karlsson, U. Gunneflo, and J. Torin. Two software techniques for on-line error detection. In *Proc. 22nd Int. Symp. on Fault-Tolerant Computing*, pages 328–335, June 1992.

[49] J. Misra. Distributed discrete-event simulation. *Computing Surveys*, 18(1):39–65, Mar. 1986.

[50] V. F. Nicola, M. K. Nakayama, P. Heidelberger, and A. Goyal. Fast simulation of dependability models with general failure, repair and maintenance processes. In *Proc. 20th Int. Symp. on Fault-Tolerant Computing*, Jun. 1990.

[51] R. R. Oldehoeft. Program graphs and execution behavior. *IEEE Trans. on Software Eng.*, SE-9:103–108, 1983.

[52] D. K. Pradhan, editor. *Fault-Tolerant Computing: Theory and Techniques*, volume I. Prentice-Hall, 1986.

[53] T. W. Pratt. *Programming Languages: Design and Implementation.* Prentice-Hall, Inc., Engelwood Cliffs, New Jersey, 1975.

[54] R. A. Sahner and K. S. Trivedi. Reliability modeling using SHARPE. *IEEE Trans. on Reliability*, R-36(2):186–193, Jun. 1987.

[55] A. M. Saleh, J. J. Serrano, and J. H. Patel. Reliability of scrubbing recovery-techniques for memory systems. *IEEE Trans. on Reliability*, 39:114–122, April 1990.

[56] W. H. Sanders and J. F. Meyer. METASAN: A performability evaluation tool based on stochastic activity networks. In *Fall Joint Comp. Conf.*, pages 807–816, Nov. 1986.

[57] SAS Institute Inc., Box 8000, Cary, NC 27511-8000. *SAS Manual*, 1985.

[58] C.H. Sauer, E.A. MacNair, and J.F. Kurose. Resq: Cms user's guide. Technical Report RA-139, IBM T.J. Watson Research Center, Yorktown Heights, N.Y., Apr. 1982.

[59] H. Schwetman. Hybrid simulation models of computer systems. *Communications of the ACM*, 21(9):718–723, Sep. 1978.

[60] H. Schwetman. Csim: A C-based process-oriented simulation language. In *Proc. Winter Simulation Conf.*, 1986.

[61] SES, Inc., Austin, TX. *SES/Sim Simulation Language Reference Manual*, Mar. 1989.

[62] P. Shahabuddin, V. F. Nicola, P. Heidelberger, A. Goyal, and P. W. Glynn. Variance reduction in mean time to failure simulations. In *Proc. Winter Simulation Conference*, pages 491–498, 1988.

[63] K. G. Shin and Y. H. Lee. Measurement and application of fault latency. *IEEE Transactions on Computers*, C-35:370–375, 1986.

[64] K. G. Shin and T. Lin. Modeling and measurement of error propagation in a multimodule computing system. *IEEE Trans. on Computers*, 37:1053–1066, Sep. 1988.

[65] S. Y. H. Su and T. Lin. Functional testing techniques for digital lsi/vlsi devices. In *21st Design Automation Conf. (DAC)*, pages 517–528, 1984.

[66] F. L. Swern, S. J. Bavuso, A. L. Martensen, and P. S. Miner. The effects of latent faults on highly reliable computer systems. *IEEE Trans. on Computers*, C-36(8):1000–1005, Aug. 1987.

[67] D. Tang and R. K. Iyer. Impact of correlated failures on dependability in a vaxcluster system. In *2nd IFIP Conf. on Dependable Computing for Critical Applications*, Feb. 1991.

[68] D. Tang, R. K. Iyer, and S. S. Subramani. Failure analysis and modeling of a vaxcluster system. In *Proc. 20th Int. Symp. Fault-Tolerant Computing*, June 1990.

[69] S. M. Thatte and J. Abraham. Test generation for microprocessors. *IEEE Trans. on Computers*, C-29(6):429–441, Jun. 1980.

[70] K. S. Trivedi. *Probability and Statistics with Reliability, Queueing, and Computer Science Applications*. Prentice-Hall, 1982.

[71] K. S. Trivedi and R. M. Geist. Decomposition in reliability analysis of fault-tolerant systems. *IEEE Trans. on Reliability*, R-32(5):463–468, Dec. 1983.

[72] J. Walrand. Quick simulation of queueing networks: An introduction. In G. Iazeolaa, P. J. Courtois, and O.J. Boxma, editors, *Computer Perf. and Reliability - Proc. 2nd Int. MCPR Workshop*, pages 275–286. Elsevier Science Publishers B.V. (North-Holland), May 1988.

[73] A. S. Wein and A. Sathaye. Validating complex computer system availability models. *IEEE Trans. Reliability*, 39(4):468–479, Oct. 1990.

[74] M. H. Woodbury and K. G. Shin. Measurement and analysis of workload effects on fault latency in real-time systems. *IEEE Trans. on Software Engineering*, 16(2):212–216, Feb. 1990.

[75] L. Young, R. K. Iyer, K. K. Goswami, and C. Alonso. A hybrid monitor assisted fault injection environment. In *Third IFIP Conf. on Dependable Computing for Critical Applications*, Sep. 1992.

[76] L. T. Young. *Hybrid Fault Injection Environment for Measuring System Dependability*. PhD thesis, University of Illinois at Urbana Champaign – Center for Reliable and High Performance Computing, January 1993.

Vita

Kumar K. Goswami received his B.S. degree in Computer Science from Embry-Riddle Aeronautical University in 1982. From 1982 to 1984 he was a software engineer in General Electric's Simulation and Control Systems Division. From 1984 to 1986 he was employed by Raytheon's Equipment Division in Sudbury, Massachusetts where he was one of the project leaders responsible for the design of a distributed, fault-tolerant operating system for the AOSP project. He has had summer internships at the Jet Propulsion Laboratory in California and at IBM's T.J. Watson Research Center. He received his M.S. degree in Computer Science from the University of Illinois at Urbana-Champaign in 1988. He is a candidate for the Ph.d in Computer Science at the University of Illinois and is supported by a NASA Graduate Student Researcher's Fellowship. Upon completion of his disseration, he will join Tandem Computers in Cupertino, California.