June 1992

UILU-ENG-92-2217
CRHC-92-11

Center for Reliable and High-Performance Computing

DEPEND: A SIMULATION-BASED ENVIRONMENT FOR SYSTEM LEVEL DEPENDABILITY ANALYSIS

Kumar Goswami and Ravishankar K. Iyer

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

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

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-92-2217 CRHC-92-11	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 National Aeronautics and Space Administ.	
6c. ADDRESS (City, State, and ZIP Code) 1101 W. Springfield Avenue Urbana, IL 61801	7b. ADDRESS (City, State, and ZIP Code) Moffitt Field, CA
8a. NAME OF FUNDING / SPONSORING ORGANIZATION 7a	8b. OFFICE SYMBOL (If applicable)
9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER	
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)
DEPEND: A simulation-based environment for system level dependability analysis

12. PERSONAL AUTHOR(S)
GOSWAMI, Kumar and Ravishankar K. Iyer

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical	FROM ______ TO ______	92-06-26	41

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
dependability analysis, fault injection, near-coincident errors, error latency, triple modular redundant systems, simulation

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The design and evaluation of highly reliable computer systems is a complex issue. Designers mostly develop such systems based on prior knowledge and experience and occasionally from analytical evaluations of simplified designs. This paper presents a simulation-based environment called DEPEND which is especially geared for the design and evaluation of fault-tolerant architectures. DEPEND is unique in that it exploits the properties of object-oriented programming to provide a flexible framework with which a user can rapidly model and evaluate various fault-tolerant systems. The paper describes the key features of the DEPEND environment and illustrates its capabilities with a detailed analysis of a real design. In particular, DEPEND is used to simulate the Unix based Tandem Integrity fault-tolerant and evaluate how well 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. Issues such as the method used by DEPEND to simulate error latency and, the time acceleration technique that provides enormous simulation speed up are also discussed. Unlike any other simulation-based dependability studies, the use of these approaches and the accuracy of the simulation model are validated by comparing the results of the simulations with measurements obtained from fault injection experiments conducted on a production Tandem Integrity machine.

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

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

DEPEND: A Simulation-Based Environment for System Level Dependability Analysis

Kumar K. Goswami
Ravishankar K. Iyer

Center for Reliable and High-Performance Computing
University of Illinois at Urbana-Champaign
1101 W. Springfield Ave., Urbana, IL 61801

Abstract

The design and evaluation of highly reliable computer systems is a complex issue. Designers mostly develop such systems based on prior knowledge and experience and occasionally from analytical evaluations of simplified designs. This paper presents a simulation-based environment called DEPEND which is especially geared for the design and evaluation of fault-tolerant architectures. DEPEND is unique in that it exploits the properties of object-oriented programming to provide a flexible framework with which a user can rapidly model and evaluate various fault-tolerant systems. The paper describes the key features of the DEPEND environment and illustrates its capabilities with a detailed analysis of a real design. In particular, DEPEND is used to simulate the Unix based Tandem Integrity fault-tolerant system[1] and evaluate how well it copes with 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. Issues such as the method used by DEPEND to simulate error latency and, the time acceleration technique that provides enormous simulation speed up are also discussed. Unlike any other simulation-based dependability studies, the use of these approaches and the accuracy of the simulation model are validated by comparing the results of the simulations with measurements obtained from fault injection experiments conducted on a production Tandem Integrity machine.

[1]The MTBF figures presented in this paper should not be construed to reflect the MTBF figures of an actual Tandem Integrity system because key parameters that have a direct bearing on this measure were *not* obtained from measurements of the Integrity system but rather from other production machines. For this reason, the results shown in this paper should only be construed to reflect the trend and behavior of a general TMR based system.

1 Introduction

The rapid growth in the demand for highly dependable systems and the increasing complexity of these systems has made system dependability issues – availability, reliability, fault tolerance – too important to be based on prior know-how, engineering changes and other fragmented approaches. The classical approach of designing for performance and then addressing dependability issues and the practice of determining a system's dependability with isolated analysis of its individual components are no longer adequate. Since dependability is a system issue, an entire system should be analyzed under various realistic stress conditions to determine the types of faults to which it is especially vulnerable, to study the dynamic interactions between the components in the system and to identify the major dependability bottlenecks.

There is a lack of automated analysis tools that facilitate such studies. This paper describes a simulation-based environment called DEPEND that is explicitly geared for system level dependability analysis. DEPEND is unique in that it exploits the properties of object-oriented programming to provide a flexible framework with which a user can rapidly model and evaluate various fault-tolerant architectures. DEPEND contains objects which inject faults, model the behavior of system components, compile fault statistics and perform other functions typically required for fault injection studies. These objects are general in nature and a user can customize them by specifying the exact behavior of the components and the fault models desired. The user can then connect the objects together to build complex simulation models with minimal effort. This approach makes DEPEND a versatile tool that can model a wide variety of architectures and fault scenarios.

The capabilities and the features of DEPEND are illustrated with a simulation-based fault injection study of the Tandem Integrity system – a TMR-based, fault-tolerant computer. It is well established that this system is very effective against single faults [Jewett 91, Young 92]. An important question is how such systems cope with near-coincident errors generally caused by correlated failures 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. To study these issues, DEPEND was used to simulate the Integrity system and evaluate the combined effect of all these factors. This comprehensive study demonstrates the capabilities of DEPEND in a realistic setting.

The paper focuses on the features of DEPEND used to conduct fault-injection and dependability studies. The general design philosophy and the key aspects of DEPEND are discussed in detail. These include the object-oriented programming environment that provides a framework for rapid modeling, simulation of correlated and latent faults and the time acceleration technique that provides significant simulation speed up. These techniques and the simulation of the Integrity system are validated by comparing the results of the simulations with measurements obtained from fault injection experiments conducted on a production Integrity machine. To our knowledge, no other simulation-based dependability study has been validated in such a fashion.

The next section presents tools and methods commonly used for dependability analysis and it surveys simulation-based tools that have appeared recently. Section 3 describes the DEPEND environment and illustrates the user programming interface with a very simple example. The details of the Tandem Integrity system, the simulation model of the system and the simulation of error latency are presented in section 4. Section 5 describes the testbed used to conduct the injection experiments on a Tandem Integrity system and it also presents the results that validate the system simulation model. Finally, section 6 details the simulation-based fault injection experiments conducted and the results obtained. Concluding remarks and an assessment of the results are presented in section 7.

2 Methods & Tools used for Dependability Analysis

Analytical tools based on continuous time Markov chains are frequently used for dependability analysis in the early design stage. SHARPE [Sahner 87], SAVE [Goyal 86], and METASAN [Sanders 86] are three well known analytical tools used for dependability analysis. METASAN also uses simulation techniques. These tools are excellent for rapid modeling and analysis of systems and they require very little computer processing time. Their drawback is that stringent restrictions and simplifying assumptions are needed to keep a model analytically tractable. This makes it difficult to model systems in detail and leads to abstractions that may not be representative of the system being analyzed. Furthermore, analysis of certain issues such as error latency and propagation, pose significant problems.

Fault injections, both hardware and software, is a method commonly used for dependability analysis at the prototype stage. Studies that have used the physical insertion of faults, via hardware, include fault injection experiments conducted on the FTMP (Fault Tolerant Multi-Processor) [Lala 83], fault latency studies on the NASA AIRLAB testbed [Shin 84a], [Shin 84b], and a validation study of the computerized interlocking system for the French railways [Arlat 88]. In [Young 92], a hybrid monitor is used to inject errors and analyze the Tandem Integrity fault-tolerant computer. The main drawback with the proposed hardware methods is that they offer no feedback to the system designers and are only useful as validation methods *after* the system has already been built.

Two tools that use software fault injection are FIAT [Segall 88] and FERRARI [Kanawati 92]. The FIAT environment utilizes software implemented fault injection to emulate various hardware faults and is being used by the Federal Aviation Authority to validate the software system of the Advanced Automation System. FERRARI is a similar tool which also allows the injection of transient errors. These tools are specifically designed to inject faults into the memory space of a program as it is executing. They have been specifically designed to evaluate and validate application software after the software has been developed.

Simulation has greater applicability than analytical techniques and unlike the fault injection techniques, simulation can be used in the early life cycle of a design to provide feedback to the designer to ensure that dependability specifications are met. Various researchers have addressed specific aspects of dependability evaluation via simulation. At the chip-level, simulation experiments have been used to determine the efficiency of error-detection mechanisms [Courtois 79] and, fault propagation [Lomelino 86]. Recently, in [Czeck 91], a simulation model of the IBM PC was developed and injected with gate-level transient faults. A simulation-based tool called FOCUS [Choi 89] has been developed to investigate reliability and fault sensitivity analysis of VLSI systems. FOCUS has been used to evaluate several chip designs, including the BDX930 Bendix flight control processor and the EC-16 Pratt/Whitney jet engine controller. These studies indicate that simulation techniques are useful and provide insight into fault-propagation and fault-impact at the chip-level.

At the system level, however, there is a lack of simulation-based dependability evaluation tools. Most simulation-based tools are designed to facilitate performance analysis (CSIM [Schwetman 86], ASPOL [MacDougall 73], SES Workbench [SES 89], RESQ [Sauer 82]). VHDL [IEEE 88] is a powerful hardware specification language but it does not contain built-in facilities to support dependability analysis. To our knowledge, there is no other simulation environment like DEPEND although some are being developed, such as NEST [Dupuy 90] and the Rainbow Net [Johnson 91]. The Rainbow Net simulator uses a petri-net like structure to model a system's behavior under fault conditions. The model is then solved via simulation to produce typical dependability measures. NEST is a simulation and proto-typing testbed for analyzing distributed networks and system protocols. It can fail links and nodes, but it is very specialized in the types of architectures it considers and the types of faults it can model.

While these tools are useful in the context of their specific applications, several important issues with regard to simulation-based, dependability evaluation of fault-tolerant systems have yet to be addressed. These include 1) evaluating a system for long periods of time to obtain statistically valid results, 2) providing extensive, automated fault injection environments, 3) providing facilities to model fault tolerant components and, 4) validating simulation results with actual data. Other issues in simulating complete systems are coordinating the injection of faults, the reactions to these faults (e.g. aborting a process) and initiating repairs that occur simultaneously in the various components in the system. As the system gets larger, the problem can become overwhelming. There are no general-purpose simulation based tools that address all these issues and provide an automated environment that makes the analysis of fault-tolerant systems more feasible. The development of such a tool has been the motivation behind DEPEND.

DEPEND has been used to model the hardware and software of a distributed system employing centralized and distributed prediction-based load sharing [Goswami 90a]. Faults were injected into the processors and the communication channels to corrupt or destroy data used by the load sharing software. The simulation helped to isolate a minor implementation detail of the load sharing software that caused destructive task scheduling. This would have been difficult to detect without simulating the software in detail and injecting faults. DEPEND is now being used to simulate the computing element of the Hubble telescope. Various architectural configurations are being modeled to find an optimal design that will be both reliable and yet afford the highest performance. Traces of actual workload are being used to gauge the performability of the system under faults.

3 The DEPEND Environment

DEPEND is a joint performability and dependability analysis tool that provides a simulation framework to facilitate modeling and analysis of fault-tolerant architectures at the system level (e.g., modeling CPUs, communication links and communication protocols). DEPEND exploits the properties of object-oriented programming to provide the framework which consists of objects that perform simple tasks but which can be easily connected together to build complex simulation models. Some objects simulate the functional behavior [2] of components commonly used in fault-tolerant systems. Others inject faults, compile fault statistics or generate detailed reports. Each object is

[2]As an example, the functional behavior of an adder is the addition operation. The gate level operation required to do the addition is not simulated. However, there is nothing that prevents using DEPEND for lower level simulation except for processing speed.

self-contained and is designed to interface with others to create a useful, functional model. For example, an object designed to inject faults can be connected to an object that models a communication link to create an object that can simulate a communication link under various fault conditions. The simplicity of the objects allows them to be used in a variety of ways and yet they are designed to be easily customizable to provide very specific functions. This makes it possible to use DEPEND to model many different types of architectures, reconfiguration and repair schemes and various fault scenarios.

The steps required to develop and execute a model are shown in figure 1. The user writes a

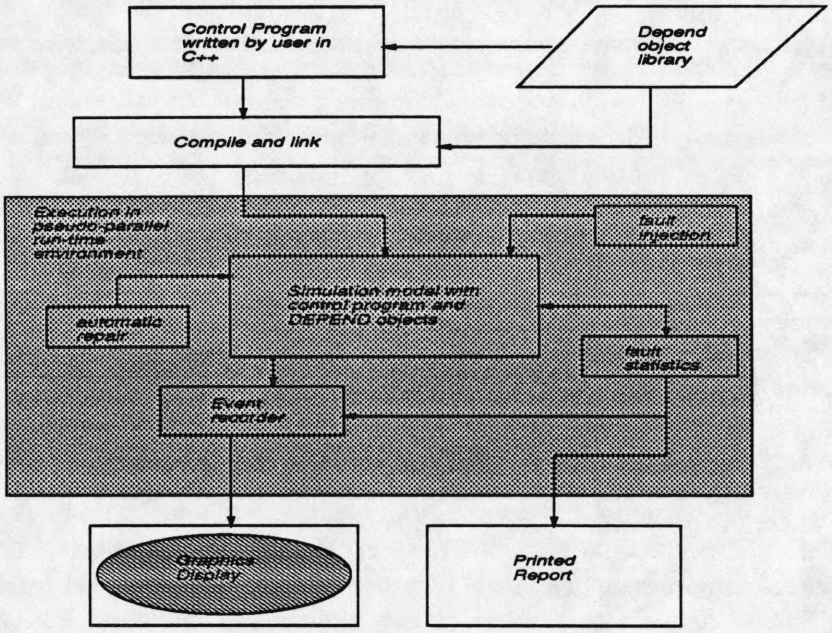

Control Program written by user in C++
Depend object library
Compile and link
Execution in pseudo-parallel run-time environment
fault injection
Simulation model with control program and DEPEND objects
automatic repair
fault statistics
Event recorder
Graphics Display
Printed Report

Figure 1: Steps in developing and simulating a model with DEPEND.

control program in C++ with the objects provided by DEPEND. Once it is written, a command is used to compile and link the program with the DEPEND objects and the run-time environment. The model is then executed in a simulated parallel environment created by the 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, all according to the user's specifications. When the simulation is complete, a report containing the essential statistics of the simulation is produced.

The next subsection presents the fault models used in DEPEND. The following subsection describes a few key DEPEND objects and the general philosophy behind their design. The programming environment of DEPEND is illustrated with a simple example in subsection 3.3 and the time acceleration technique used to reduce simulation execution time is described in subsection 3.4. The approach used by DEPEND to simulate error latency is discussed in section 4.2, along with the description of the Integrity simulation model, because it is best explained in the context of the Tandem Integrity application.

3.1 DEPEND fault models

DEPEND was designed to handle functional fault models which simulate the system level manifestation of low, 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 more on the behavior of a component rather than the exact underlying structure. DEPEND provides the mechanism for injecting faults and relies on a 'fault subroutine' provided by the user to simulate a specific fault model. Hence, the the generality of the types of faults injected is not limited by the tool and is determined by the user. Default fault subroutines are provided with each DEPEND object, but these can be overridden by the users by supplying fault subroutines of their own. The default fault models depict the functional behavior of faults in memory and I/O subsystems, in CPUs and in communication channels. For example, the default CPU fault model assumes that the processor hangs when a fault is discovered. If the fault is transient, it disappears when the CPU is restarted. If the fault is permanent, it is corrected only when the CPU is replaced. Thus this fault model represents the functional behavior of low-level, stuck-at faults and transient errors in key CPU registers and functional units. Similarly, for a communication medium, the functional fault model assumes that messages are corrupted (bits in the fields of a message are flipped) or lost due to errors. This simulates the effects of a noisy communication medium. The reaction to these faults might trigger a re-transmission of the message. Two default fault models are available for memory and I/O subsystems. Either a bit of a word is flipped or a flag is raised to represent the error. The error can be detected with a byte-by-byte comparison or by checksum comparison, if the error is a flipped bit. Otherwise, it can be detected by checking the status of the flag.

3.2 The DEPEND Object Library

DEPEND is essentially a library of objects that provide functions typically needed to model fault-tolerant architectures and conduct system level, simulation-based dependability studies. An object is perfect for encapsulating a set of related functions. The object library consists of a hierarchy of elementary and complex objects. Elementary objects provide basic functions like fault injection and the compilation of statistics. Complex objects, created with several elementary objects, perform more complex and specific tasks. The objects were designed with four criteria in mind:

- They should make very few assumptions and should be general-purpose. For example, the fault injector object does not assume any particular fault model.
- They should easily interface with other objects.
- They should be easy to customize and easy to use to create new objects.
- When possible, each object should provide default functionality to reduce the work done by a user. It should be possible to override these defaults.

These criteria make it possible to simulate a wide range of architectures, and fault scenarios with just a few key objects. These key objects model the fundamental components found in most fault tolerant architectures. The objects model fault tolerant processors, communication channels, memory, voters and other basic parts of a fault-tolerant computer system. As a result they can be combined with others or replicated to build more complex systems such as self-checking systems, N-modular redundant systems and multiprocessor systems. The rest of the subsection describes an object used to inject errors and objects used to model processors and communication mediums. Section 3.3 shows how they can be used to create a simulation program.

3.2.1 Fault Injector

The primary object used to inject faults is the *fault injector* shown in figure 2. To use the injector, a user specifies the fault arrival distribution and a fault subroutine that contains the fault model and simulates the occurrence of a fault. The injector randomly samples from the specified distribution to determine the arrival time of each fault and then calls the fault subroutine at the appropriate times. With this approach, the fault injector is not restricted by any specific fault model and can be used in a wide variety of ways. The injector is an elementary object that was expressly designed to be used by more complex objects such as the server and the link (figures 3 and 4) to inject faults specific to these components. For example, the link, which simulates a communication medium, uses the *fault injector*, along with a fault subroutine to corrupt and lose messages traversing along the communication medium.

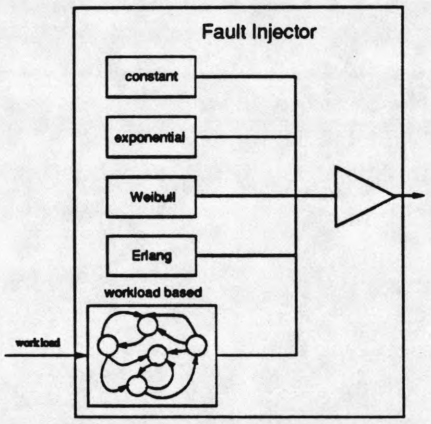


Figure 2: The Fault Injector Object.

In addition to using statistical distributions to model fault arrival (see figure 2), the injector also provides a workload-based injection scheme that varies the fault arrival rate based on a specified workload. The user provides a workload function, a set of workload states and an exponential fault arrival rate for each state. For example, the workload function may be the utilization of a server or any other function that provides a measure between 0 (low workload) and 1 (high workload). The states may be defined to consist of a high utilization state (e.g., with utilization between 0.8 to 1.0), a medium state (e.g., 0.8 to 0.4) and a low state (e.g., 0.4 to 0.0). To maintain a history of the workload behavior, the fault injector periodically polls the workload function to update a state transition diagram. This history is used to inject a larger number of faults during peak workload conditions and fewer faults when the workload is low. This technique models the workload/failure dependency observed in [Iyer 82] and [Castillo 82]. Depending on the workload function supplied to the injector, this approach can be used to model one or a combination of workload factors.

The *fault injector* illustrates what we mean by "providing a simulation framework". The injector provides the basic algorithms and mechanisms needed to inject faults and allows 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 arrival distributions and fault subroutines. Other DEPEND objects are similarly designed to provide the functionality that are commonly needed without restricting the applicability of the object.

3.2.2 Fault-tolerant server

The *server* object, which is typically used to model CPUs and other processors, is an example of a complex object (figure 3) that is built from several elementary objects. It uses a *fault injector* object to inject faults and other objects to compile and output fault statistics. The server contains functions to simulate the acquisition, the use and the release of a server. The server offers several service disciplines including, first come first serve (FCFS), round-robin and pre-emptive round-robin and, it simulates contention at the server.

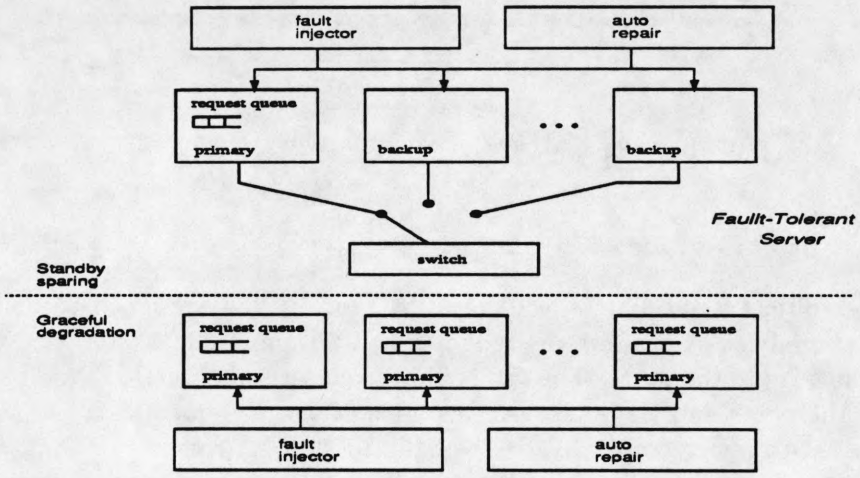


Figure 3: The Fault-Tolerant Server Object.

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 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 becomes faulty and all requests in the queue, including the one currently using the server, are dequeued or aborted to simulate that the processor is in a hung state. The remaining processing time of each request is returned so that remedial action can be taken. Other fault models can be simulated by specifying user defined faults. To simulate user defined faults, the server calls a pre-specified fault subroutine, 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 subroutines to be invoked when a fault or repair event occurs.

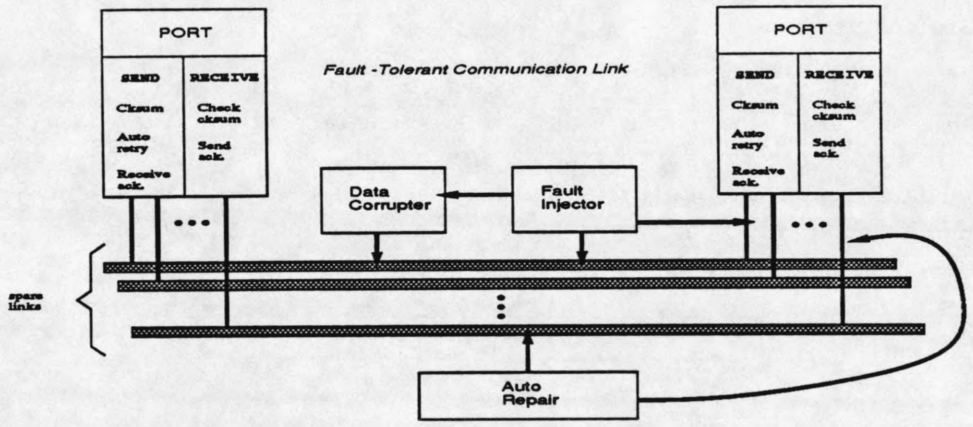


Figure 4: The Fault-Tolerant Communication Link.

3.2.3 Fault-tolerant communication link

Another complex object is the *link* object shown in figure 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 objects. Several *server* objects are used to model the ports and the links. An instance of the *injector* object is used to provide the mechanism needed to inject faults. The rest of the *link* object consists of additional software to model the behavior of a communication medium. To initialize a *link* object, 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 object 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 fault models, including faulty link, faulty connection, lost message and corrupted message are offered. If a link or connection 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. A message corruption fault flips bits in a message based on user specifications. Like the *server* object, these default fault models can be overridden by supplying user written fault subroutines. 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.

Table 1 briefly describes the major objects available in DEPEND. Each object has functions that maintain and report fault statistics, from availability and mean time between failures to a detailed report on each fault injected and repair attempted. The objects in DEPEND can be used to simulate a gamut of different architectures from hypercubes and meshes to TMR and multiprocessor systems. The level of detail of the simulation is up to the user. For example, a node

Name	Type	Description
Active_elem	Elementary	Simulates a basic server. Offers various usage disciplines: first come first serve, round robin, pre-emptive round robin. Allows manual fault injection and repair.
Injector	Elementary	Automatically injects faults based on statistical distributions. Offers workload based injections. Can inject correlated and latent faults.
Checksum	Elementary	Compute checksums.
Fault Reporter	Elementary	Compiles fault statistics. Displays MTBF, MTBR, availability and coverage. Can output details of every fault injected and repair attempted.
Voter	Elementary	Simulates a basic voter. Has timeout features. Default voting scheme: byte by byte comparison. Allows user defined voting algorithms.
RAM	Complex	Simulates a basic random access memory. Can inject permanent and transient faults (bit flips) with latencies.
Link	Complex	Simulates communication channels. Several types of faults: link dead, packet corruption, packet loss and user defined faults. Automatic retry.
Server	Complex	Simulates a server with spares. Three sparing policies: no spare, graceful degradation, stand-by sparing. Automatic repair and reconfiguration with specified coverage. Automatic injection of faults.
NMR	Complex	Simulates dual self-checking, triple-modular redundant and N-modular redundant components.
Fault Manager	Complex	Simulates software fault management schemes. Logs faults and shuts off components which exceed their fault threshold.

Table 1: DEPEND objects used for fault-based simulations

of a hypercube may be represented by a server object or, the user can use the server, link and RAM objects to simulate the internal architecture of each node in more detail.

3.3 DEPEND's 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, a user does not have 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++ 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 such as C. Finally, C++ produces efficient code, and the C++/UNIX environment is widely available.

Figures 5 and 6 show a simple, example program that illustrates the programming environment and some of DEPEND's features [3]. 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 sparing mode. There are two ports connected to the link and each port has an additional spare port. The **co_routine()** statement in routines **main()** and **receiver()** distinguish them from ordinary C++ subroutines. **main()** and **receiver()** are light-weight co-routines that execute concurrently in a pseudo-parallel environment. Each time the **co_routine()** 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 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-routines (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.

[3]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) {     co_routine("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 5: A simple example program using DEPEND – the main co-routine.

```
30) void receiver(int port_id)
31) {       co_routine("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 6: A simple example program using DEPEND – the "receiver" co-routine.

The simulation contains several concurrent threads of execution. There are the threads for the main routine and the two 'receive' routines. In addition, there are several threads that inject faults into the ports, the links and the messages traversing them.

The level of detail of the simulation model depends on the user. For example, for the computations (line 34, 35, etc.), the user may choose to simply forward the simulation clock and the data in the messages could be arbitrary values. Alternatively, the user could execute actual code and send real data back and forth, making it possible to test actual programs such as communication protocols, algorithm-based fault-tolerant schemes [Huang 84] 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 [Goswami 90a].

The simple example in figures 5 and 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 object 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 object (**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 object, one for each link in the cube. By modifying the link declaration, a comparative performability study can be conducted to determine whether the *cold sparing* or the *graceful degradation* sparing mode is best suited for a particular application.

A more realistic example in which a complete fault-tolerant system is simulated with DEPEND and injected with near-coincident errors is described in section 4.

3.4 Simulation time acceleration

Although simulation has the advantage that it can model the detailed behavior of a system, it is bound by execution time. Simulating systems with large MTBFs can require very large execution times because system failure is a rare event. This problem is further exacerbated because a simulation must be executed several times or over a large period of time to collect enough failures to acquire statistically valid results.

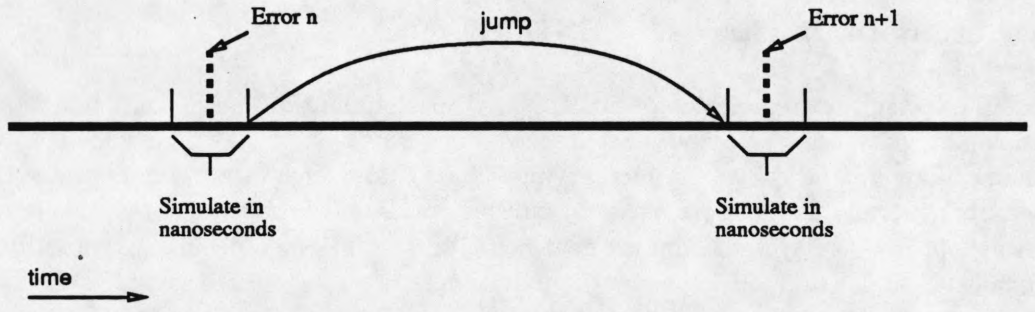


Figure 7: Time acceleration: "Error" driven simulation.

Several techniques can be used to effectively reduce simulation time explosion. In DEPEND, a time acceleration mechanism has been incorporated to reduce simulation execution times. Objects in DEPEND are designed to furnish the time of their next event. Using this feature, a list of important events which affect dependability measures, such as the time when the next error will arrive and when the next latent error will be activated, are kept in a chronologically sorted list. The simulator leaps forward to the time of the event at the head of the list and resumes processing at the granularity of the system clock until the effect of the event has subsided (figure 7). This is different from regular event-driven simulation because this approach allows certain user-specified "unimportant" events to be suspended during leaps while others continue. This acceleration technique is used with the Tandem Integrity simulation described in section 4.2.

DEPEND also allows a user to implement other acceleration techniques, such as importance sampling. Importance sampling increases the number of system failures by accelerating the failure rates (thereby reducing the number of events that have to be simulated). It then produces unbiased estimates of the dependability measure by multiplying the value with a ratio that compensates for the accelerated failure rate used [Lewis 89]. DEPEND facilitates the implementation of importance sampling techniques by allowing control over the simulation engine and providing a C++ user interface with which users can incorporate importance sampling methods directly into their simulation models. Importance sampling has been very successful in reducing simulation time when the failure and repair distributions are exponential. In [Nicola 90], the authors extend the applicability of importance sampling to non-Markovian systems with general failure and repair distributions. But, even these methods cannot be used to accurately calculate steady state MTTF and MTBF measures when the error occurrence process is not exponential [Shahabuddin 88], [Nicola 90] – as is the case in this study. However, it should be noted, that the time acceleration method described above is not a substitute for importance sampling and that it can be used in conjunction with importance sampling to further reduce simulation time when computing dependability measures other than the steady state MTBF or MTTF.

Another approach to reduce simulation execution time is to use hybrid simulation techniques which combine the detailed modeling available with simulation with the efficiency of analytical modeling (e.g., CTMC). This approach has been used on the Tandem Integrity simulations and has been shown to provide significant reduction in simulation time [Goswami 92]. An advantage of this approach is that it is not restricted to Markovian processes (so long as the CTMC is solved via Monte Carlo simulation) and is therefore more widely applicable.

4 A Case Study to Illustrate DEPEND

This section illustrates the capabilities of DEPEND with a simulation-based fault injection study of the Tandem Integrity fault tolerant computer system. This system has been shown to be very effective against single faults. An important question is how such a system copes with near-coincident faults generally caused by correlated failures and latent faults. Architectural issues that have a bearing on how the system handles near-coincident errors include memory scrubbing, reintegration policies and workload dependent repair times. To study these issues, DEPEND was used to simulate the Integrity system and evaluate the combined effect of all these factors.

The next subsection describes the salient features of the Tandem Integrity system. The following subsection describes the simulation model of the machine developed with DEPEND. The error occurrence process and the techniques used to simulate error latency and speed up the simulation are presented in the third subsection. The last subsection presents the assumptions and parameters used in the simulation.

4.1 The Tandem Integrity Fault-Tolerant System

The Tandem Integrity fault-tolerant system [Jewett 91] was introduced in 1990. The system is shown in figure 8. Each CPU board contains a MIPS R2000 RISC processor with an on-chip virtual memory mechanism, 128 Kbytes of cache (64 Kbytes of instruction cache and 64 Kbytes of data cache), a minimum of 8 Mbytes of local memory and its own separate clock. All three processors execute the same instruction stream but due to clock drift, they do not necessarily execute the instructions at the same time. Any time the global memory is accessed, the three processors are synchronized and their requests are checked by the voter. The voter waits until it

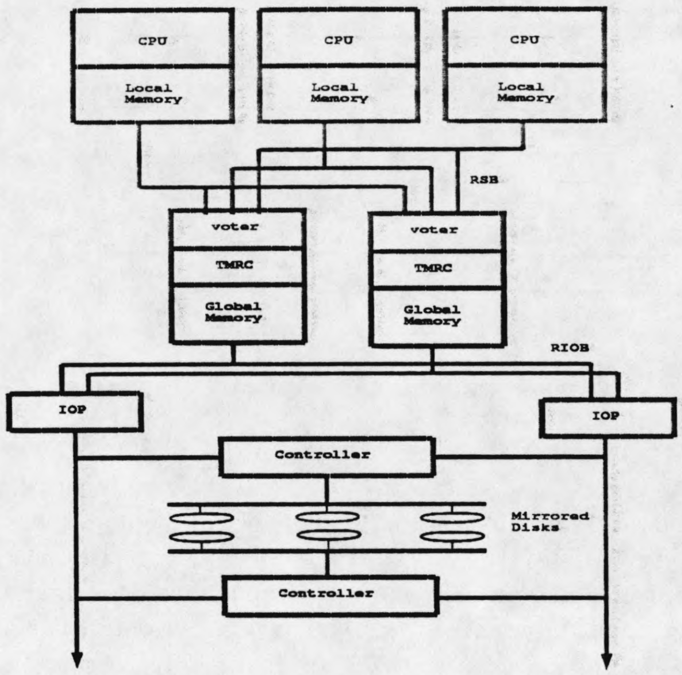


Figure 8: The Tandem Integrity processing subsystem

receives a request from each healthy processor. It then compares all three requests (the request type, the address field and the data field, if any) for agreement and then forwards the request to the global memory. If the processors do not access the global memory within 2047 instructions, the voter forces them to synchronize to prevent large clock drifts. If there is a discrepancy during voting, the processor in disagreement is shut down and the remaining processors are alerted but continue with normal operation. The faulty processor performs a power-on self-test (POST), and if it completes successfully it is re-integrated into the system. The re-integration process consists of halting all the healthy processors, copying their state information and the contents of their local memory to global memory and then copying them back again to all three processors to ensure that all of them have identical states. Re-integration takes only 1.5 seconds for a system with 8 Mbytes of local memory. The entire process takes approximately 61.5 seconds because 60 seconds are needed to execute the POST. The Tandem system can continue to function with one faulty processor. However, if a second disagreement is detected by the voter before the faulty processor is repaired, the system fails.

The local memories in the Tandem Integrity do not have parity or ECC circuitry. The system relies on memory scrubbing to correct transient memory errors. Unlike conventional systems, scrubbing in the Tandem Integrity system involves voting. Periodically, 64 byte portions of all three local memories are copied to global memory. As they are copied, they are checked to see if they are in agreement. If a disagreement is detected, the local memory's block that disagrees is rewritten with the correct data found in the other two memories.

The boards containing the voter and the global memory are referred to as the TMRC. The TMRC contains up to 128MB of global memory and interfaces with the CPUs via the Reliable System Bus (RSB). The primary function of the TMRC is to vote upon the transactions sent by

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 2: Global memory re-integration times with varying machine idle percentages

the CPUs. All processor transactions that are external to the CPUs are voted on by the TMRC. All bits of the command, address and data are voted on. The voting circuit is duplicated and compared and any self-check errors halt the board. The global memories are protected by parity, with one parity bit per byte of data. When a parity error is detected by the TMRC, the backup memory takes over and prevents the primary memory from driving the RSB. A global memory is re-integrated in the background, interleaved with ordinary processing. For a system with 32MB of global memory, the re-integration process takes approximately 30 seconds on an idle system. Since the re-integration time is load dependent, the re-integration time of an Tandem Integrity with 32MB of global memory was measured under various loads. Table 2 contains the results of these measurements. Global memory re-integration has a lower priority than CPU re-integration. A global memory undergoing re-integration will be aborted if a CPU is ready to be re-integrated. Once the CPU re-integration is complete, the global memory re-integration is restarted from the beginning. The global memory, like the local memory, is scrubbed periodically to eliminate latent errors.

The I/O subsystem is a redundant system with two I/O processors (IOP), redundant busses to the controllers (RIOB) and more than one controller per disk. In a healthy system, there are always two paths to each disk. The disks can be organized as mirrored pairs so that a hot backup is available in case a disk fails. Fault recovery for the processing core of the Tandem Integrity is a joint hardware and software effort, but for the I/O subsystem, the software is primarily responsible for recovery from I/O processor (IOP) and I/O controller faults. The IOP exception handler determines if the fault is in an IOP or in a I/O controller. If it is in a controller and retries do not succeed, the controller is shut off permanently. If the fault is in an IOP, all the I/O controllers attached to it are switched to the remaining IOP. If the IOP has a soft-error, it is re-integrated. Otherwise, it is logically removed from the system. The re-integration time of an IOP is less than 15 seconds. The re-integration time of the controllers varies among controllers but is typically around 30 seconds. If a disk fails, the backup disk accepts all disk requests. Once the fault disk is replaced, it can be mirrored on-line and takes approximately 20 minutes for a 150MB disk.

The Integrity system fails when any of the major subsystems, the processing core, the global memory or the I/O subsystem, fails. The processing core and the global memory fails if a second error occurs in the subsystem before recovering from an earlier error. The I/O subsystem fails if both controllers or both IOP boards fail. In the simulation model, described next, the I/O subsystem is also considered failed if both pairs of a mirrored disk have errors.

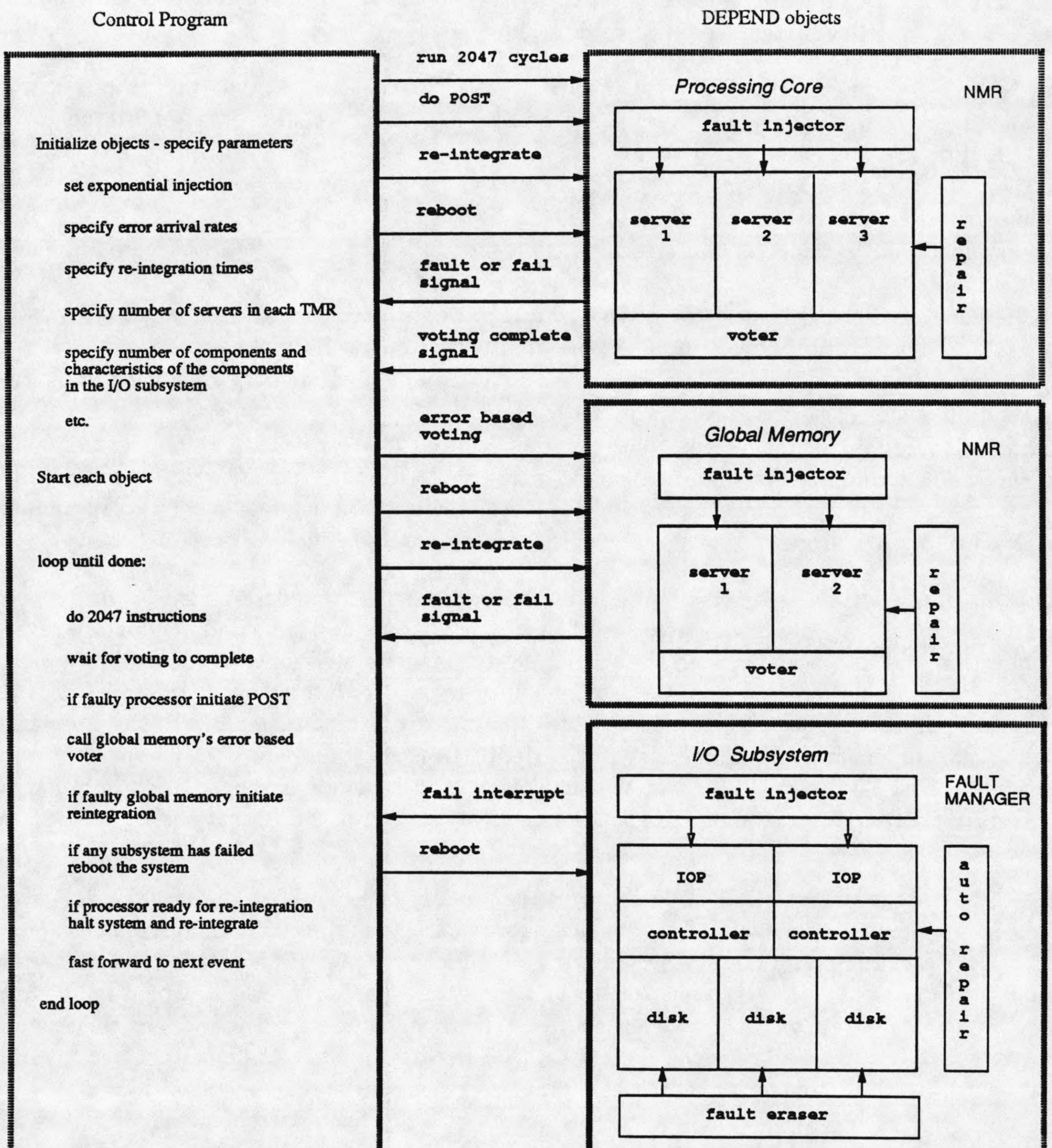
4.2 The simulation model developed with DEPEND

Recall that we are interested in evaluating the Integrity system and several of its architectural features under the stress of near-coincident errors. In order to achieve this, several key characteristics of the Tandem Integrity system were simulated. These include:

1. the loose synchronization policy of the Integrity system. The fact that the processors idle at the voters waiting for all three 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).
6. the software fault management policy for the disk and I/O subsystem which logs faults and shuts down components that receive too many faults in a short interval. Aspects of the management policy such as the specific number of faults that must arrive before a component is shutdown are proprietary. We use realistic values based on the error arrival rates.

The 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 simulation model. The simulation model of the Tandem Integrity, developed with DEPEND, is shown in figure 9. The blocks on the right are the DEPEND objects 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 object** in the DEPEND library is the primary object used to develop the simulation model. It is used to model the NMR processing core and the dual, self-checking global memory. The NMR object simulates dual self-checking, triple-modular redundant and N-modular redundant systems. It contains two, three or more servers and one voter. The servers are designed to idle until they receive a task to process. They then execute for the time period specified and feed their results to the voter. The voter waits for the servers to deposit their results and then executes a voting algorithm to detect discrepancies in the results. The voter uses a timeout condition to prevent hanging in cases where a server fails to report to the voter. The NMR object offers 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 *active* error (see below) has been injected into the server. This option was used in the simulations because the processors were not given any real data to process. In addition to these voting schemes, the NMR object also allows the voting algorithm to be specified by the user. Once the voting is complete, the NMR automatically shuts down the servers with



Control Program
DEPEND objects
Initialize objects - specify parameters
set exponential injection
specify error arrival rates
specify re-integration times
specify number of servers in each TMR
specify number of components and characteristics of the components in the I/O subsystem
etc.
Start each object
loop until done:
do 2047 instructions
wait for voting to complete
if faulty processor initiate POST
call global memory's error based voter
if faulty global memory initiate reintegration
if any subsystem has failed reboot the system
if processor ready for re-integration halt system and re-integrate
fast forward to next event
end loop
run 2047 cycles
do POST
re-integrate
reboot
fault or fail signal
voting complete signal
error based voting
reboot
re-integrate
fault or fail signal
fail interrupt
reboot
Processing Core
NMR
fault injector
server 1
server 2
server 3
repair
voter
Global Memory
NMR
fault injector
server 1
server 2
repair
voter
I/O Subsystem
FAULT MANAGER
fault injector
IOP
IOP
controller
controller
disk
disk
disk
auto repair
fault eraser

Figure 9: The simulation model of the Tandem Integrity system developed with DEPEND.

faulty results. The NMR does not offer automatic repair schemes. However, it has functions that can be called to repair the individual servers. This feature is used to simulate the automatic re-integration feature of the Tandem Integrity. The NMR object's fault injector is used to inject latent and correlated errors. The details of the error injection process are discussed in the next subsection.

The **processing core** was simulated with a NMR object containing 3 servers. Each server simulates a processor board containing a CPU and a local memory. The NMR's injection technique 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 system because parity errors in the global memory are detected when the processors synchronize at the voter to access global memory.

The **fault manager** object was used to simulate the reliability aspects of the I/O subsystem because it closely models the software controlled recovery process used with the Tandem Integrity's I/O subsystem. The fault manager object is primarily used to mimic typical software fault management schemes which log faults against components and shut down components that receive a large number of faults in a short time interval. Each component has a fault threshold. Periodically, faults logged against the component are erased so that only a burst of faults can exceed the threshold. To setup the fault manger, the user specifies: the number of components, the number of elements in each component and the number of faults that can arrive in a specified period of time without causing the element to be shutdown (fault threshold). The user can opt to apportion faults to the components probabilistically or can specify the component with each fault injected. The user can select automatic repair and the re-integration time required for each component. To simulate the I/O subsystem of the Tandem Integrity, an instance of the Fault Manager object was created with components to simulate the IOP, the controller and the three mirrored disks. The IOP and controller were each modeled with one component containing two elements. The mirrored disk was modeled with two components with each containing three elements. Once component was used to log soft errors such as errors from bad blocks. The other was used to log hard errors such as a major catastrophe that affects the entire disk and causes disk mirroring to be lost. The fault threshold for the IOP was set at 1, for the controller it was 4, for soft disk errors 8, and 1 for hard disk errors. The fault eraser was specified to wake up hourly and erase an error (if one was logged) in all healthy elements. These specifications for the I/O subsystem are not those of the Tandem Integrity system. Detailed information about the software error management scheme of the system was not available.

The **control program** in figure 9 declares instances of the three DEPEND objects and initializes and customizes them in a similar way to that shown in figure 5. Initialization consists of specifying the error arrival distribution desired, the error arrival rate, the error latency, the number of components and their re-integration times and so on. After initialization, the program "starts" each object causing it to automatically perform tasks based on the parameters specified during initialization. For example, when the fault manager object is started, it will automatically inject faults, perform repairs and erase faults until the I/O subsystem fails. 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.

After starting the objects, the control program loops for a specified number of simulated years exercising the machine. The program requests the processors to execute 2047 instructions and then synchronize at the voter. The control program then sleeps until voting is completed and then it checks to see if any processor has been shutdown by the voter. If so, a POST cycle is initiated for the faulty processor. The control program then polls the global memory boards to determine if there are any detected errors. If a board has an error, a re-integration process is started. Next, the control program checks to see if the entire system has failed and if so it initiates a reboot of the system that resets all the subsystems. If the system has not failed, the control program determines if there is a processor that has completed a POST (that was initiated at an earlier time) and is ready for re-integration. If so, any global memory re-integration that is in progress is aborted, and the system simulates the re-integration process of a processor by halting the system for 1.5 seconds. Afterwards any global memory re-integration process that was aborted is restarted from the beginning. Finally the time acceleration algorithm, described earlier, is invoked to determine the time of the next major event. The clock is advanced to that time and the whole cycle repeats.

The control program does not explicitly check for faulty I/O components. The fault manager object conducts automatic re-integration of failed elements and interrupts the control program whenever the entire I/O subsystem fails. The I/O subsystem fails if an entire component (e.g. both IOPs, both controllers or a mirrored disk pair) fails.

To summarize, this subsection describes the DEPEND objects and shows how they are used to simulate the Tandem Integrity system. The bulk of the simulation is performed by the DEPEND objects. All the statistics compilation, fault injection and repair actions are done by DEPEND. As a result, only a small control program needs to be written by the user. This makes it possible to simulate large systems with relatively little effort. Since the control program is small, it is easy to compile, debug and verify.

4.3 The Error occurrence process, Time acceleration & Experiment design

The simulation models errors in the storage elements (e.g. registers and memory) of the system. These errors are assumed to be caused by transient faults such as ionization radiation and are assumed to manifest as a single bit flip that can be detected by the voter if it is in the processor board, or detected by the parity error coding scheme if it is in the global memory. Compensating second errors to the same bit are not considered because it is a very rare event.

The TMR's fault injector is used to inject *active*, *latent* and *correlated* errors. Active errors are detected within 2047 cycles from the time they are injected. Latent errors may remain dormant for hours before being detected. As a result, a component may have several latent errors. This models the phenomenon observed when errors were injected into the Integrity system; errors injected into the section of a local memory containing the exception handler, the TLB (Translation Lookaside Buffer) miss handler or the processor register space were usually detected immediately whereas errors injected into other locations in the local or global memories had a much larger latency. A correlated error is two or more errors injected simultaneously into two or more components of a subsystem. Each correlated error can be either active or latent. This notion that the correlated errors may have different latencies is justifiable because there is a high probability that the errors will not occur in the same location in each component and hence will produce different latencies. Even if the errors occur in the same location, the latencies are still likely to be different because the processors are loosely synchronized.

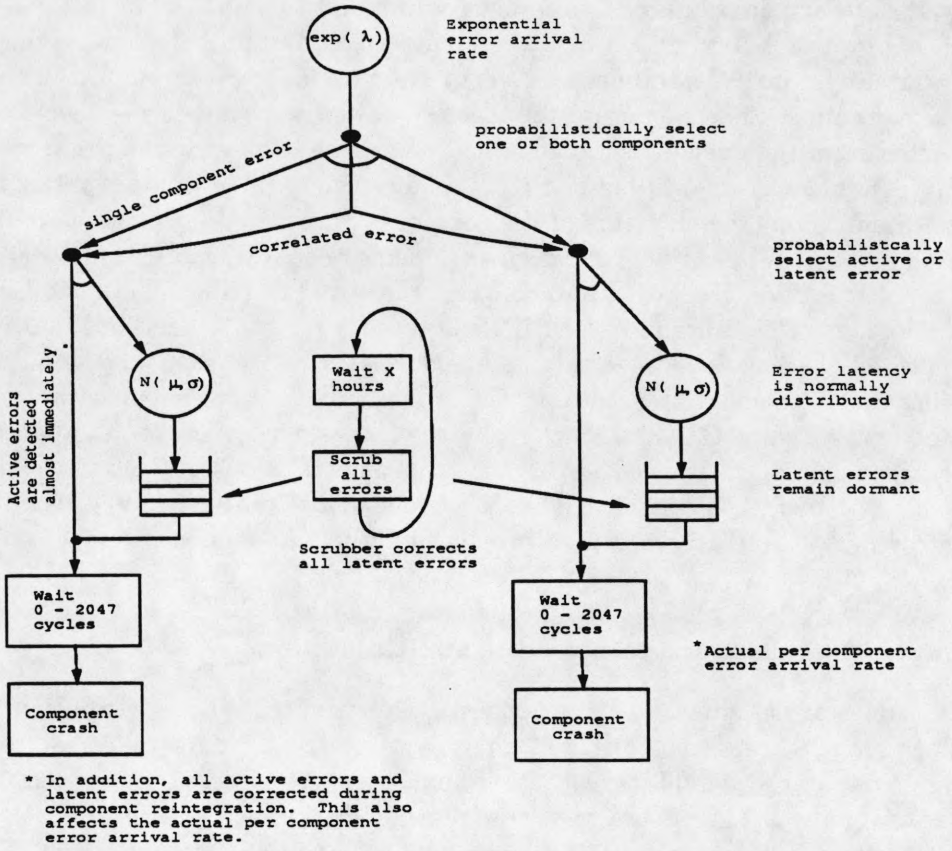

exp(λ)
Exponential error arrival rate
probabilistically select one or both components
single component error
correlated error
probabilistcally select active or latent error
Active errors are detected almost immediately
N(μ, σ)
Wait X hours
Scrub all errors
Error latency is normally distributed
Latent errors remain dormant
Scrubber corrects all latent errors
Wait 0 - 2047 cycles
*Actual per component error arrival rate
Component crash
* In addition, all active errors and latent errors are corrected during component reintegration. This also affects the actual per component error arrival rate.

Figure 10: The error occurrence process simulated with DEPEND.

The complete error occurrence process for just two CPUs is illustrated in figure 10. A similar process is used to inject errors into the global memory with one exception; only latent errors are injected. As shown in the figure, the error arrival times are exponentially distributed. When an error is injected, a probabilistic branch is used to determine whether it is a correlated error affecting 2 or more components in a subsystem, or a single error. A probabilistic branch is also used to determine whether the error is active or latent. Eventually, the error is detected by the voter and the component is shutdown and then re-integrated. The next subsection describes how error latency is simulated.

4.3.1 Simulating error latency

One way to simulate error latency is to model a memory structure and the read and write access to the words in the memory. Error latency is then the time from the corruption of a word to the time it is *read*. An excellent description of such an approach can be found in [Meyer 88]. The drawback with this approach is that in order to reduce execution time, several simplifying assumptions such as geometrical access time distribution, independence of memory accesses and simple, static read/write access distributions need to be assumed.

While such a model can be used in DEPEND, a simpler approach, which allows the user to directly specify the latency distribution, is also incorporated into DEPEND. In section 5, we show that this simpler approach is indeed valid and representative of system behavior. When the NMR object injects a latent error, it determines its latency from a statistical or empirical distribution. Based on this latency time, the error is inserted into a *dormant queue* in chronological order. The errors are extracted from the queue when their latency times have expired and are then placed on a list of *active*, detectable errors (see figure 10). This approach has several advantages. First, it makes no assumptions about the structure of the component injected and so the basic mechanism can be used to simulate error latency in CPU registers, memory boards, caches and other structures also. The latency distribution represents and contains the structural and functional differences of the various components. This is useful to DEPEND because it strives for methods that are general and widely applicable. Second, because this approach uses a distribution and does not require detailed modeling of the memory structure, it can potentially be represented with an analytical continuous time Markov chain (CTMC). Finally, and the biggest advantage of this approach is that it results in enormous simulation speed up because the time at which the next latent error will become detectable is known thus eliminating the need to simulate the system between occurrences of detectable errors. The time acceleration algorithm described in section 3.4 takes advantage of this property.

4.3.2 Assumptions and parameters used in the simulations

Error arrival times are assumed to be exponentially distributed. The distribution means are based on findings from a measurement-based analysis of real error data collected from a DEC VAXcluster multicomputer system[Tang 90]. 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. The voter is assumed to be error free. To compensate for the fact that the measurements are from a larger system and that the standard deviations are large relative to their means, three combined error arrival rates (shown in Table 5) are used in the simulations. Still, one cannot assert that the error rate of the Tandem Integrity is similar to that of the VAX cluster. Since we are not concerned with absolute reliability figures but rather with the trends and changes to system reliability due to various fault conditions, this does not pose a problem.

The error latencies used in the simulations are based on findings from a measurement study of the VAX 11/780 by Chillarege [Chillarege 87]. 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. The error latencies used in this study are approximated by normal distributions with the means and standard deviations just mentioned. A larger latency with a mean of 36 hours and a standard deviation of 18 hours is also used in the simulations to determine the effect of extremely large latencies.

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.

5 Validation of the Simulation Model

Many steps were taken to ensure that the simulation model of the Tandem Integrity adequately represents the behavior of the actual machine under error conditions. The Tandem Integrity's response to errors was determined by discussing the matter with its designers and by injecting errors into multiple components of an Integrity system, observing the interactions and incorporating them into the simulation model. For example, the prioritized re-integration policy, where a global memory re-integration is aborted to conduct a CPU re-integration, was only discovered when errors were simultaneously injected into the global memory and a CPU. To further verify the abstractions used to simulate the injection process and the system's behavior, an additional set of injection experiments were conducted and the results were compared with those obtained from the simulation. This section describes these injection experiments in detail.

Specifically, the goal of the experiments was to verify whether the simulation model is correct and detailed enough to capture the behavior of the Integrity system, especially under latent errors. We also determined if a single latency distribution was sufficient to model the workload and the dynamically changing user and system activity.

The experimental testbed takes advantage of the Integrity system's ability to re-integrate a failed component of a subsystem on-the-fly without crashing the system. The testbed was designed to repeatedly injected errors into one CPU board and collect measurements. The Integrity system's on-line re-integration also made it possible to automate the injection process and run the injection experiments continuously for several days while other users continued to use the system, unaffected. This made it possible to collect data under varying load conditions and to collect enough data to provide statistically valid comparisons. The next subsection describes the validation testbed and the injection process. The following subsection describes the validation experiments conducted. Finally, results of the experiments are presented in the third subsection.

5.1 Validation environment: Hybrid monitoring

A hybrid monitoring environment was used to conduct the injection experiments. Figure 11 shows the various parts of the environment. A Tektronix DAS 9200 digital logic analyzer was used to monitor bus activity on one CPU (the CPU that is injected with errors) of the Integrity system. The logic analyzer contains a *finite state machine* that was programmed to collect pertinent data such as the times when an injection occurs, an exception is raised by the Tandem Integrity and when POST is initiated. In addition, the types of exceptions raised and the times when the corrupted word are accessed (read and written) are also collected.

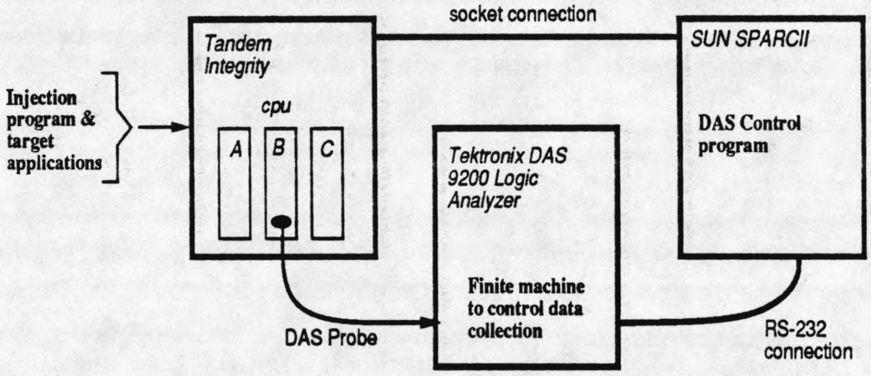

socket connection
Tandem Integrity
cpu
A
B
C
Injection program & target applications
SUN SPARCII
DAS Control program
Tektronix DAS 9200 Logic Analyzer
Finite machine to control data collection
DAS Probe
RS-232 connection

Figure 11: An injection environment using hybrid monitoring.

The logic analyzer is controlled by a *DAS control program* that runs on a Sun SparcII workstation. The control program accepts commands from the *injection program* and translates them into a sequence of DAS recognizable commands that start and stop the DAS and upload the data collected by the DAS. The *injection program* runs on the Tandem Integrity machine and injects errors into the text region (the region containing the machine instructions) of a process. It also controls the operations of the DAS remotely via the DAS control program. The specific injection programs used in the experiments are described in the next subsection. An error is injected by corrupting a bit in a word that is randomly selected from the text region of the target applications and writing that word into the memory of *one* CPU (in this case, CPUB). If the word resides in the cache, it is deleted to ensure that the corrupted version of the word is used.

An *acceleration* technique was used to increase the percent of errors detected. An executable version of a program consists of a large prologue, epilogue and support code in addition to the main body of the program. Because the prologue and epilogue mostly contain code used to initiate and terminate a program and perform other support functions, and were infrequently invoked during the execution of the target applications, only the machine instructions in the main body of the program were injected. This increased the percent of errors detected from approximately 10% to 65%. This acceleration technique may reduce the error latency values measured but our objective is validation and not error latency measurement.

The *target applications* are Gaussian elimination programs which use a random number generator to fill a square matrix and then execute the Gaussian elimination algorithm to solve the set of simultaneous equations. This process is repeated indefinitely. Two instances of the program were executed simultaneously and injected with errors. The programs generated and solved 300-by-300 element matrices.

5.1.1 The Validation Experiments

The validation was conducted in two phases. In the **first phase**, a controlled experiment was conducted to determine the *error latency distribution*, the *error isolation distribution* and the percent of errors detected. For these experiments, the error latency distribution is defined to be the time from injection to the time an exception is raised by the Tandem Integrity. The error isolation distribution is the time from the exception to the time when power-on-self-test (POST) of the CPU board is initiated. These values are subsequently input to the system simulation. The injection program used is shown in figure 12. The three minute wait time used in the injection program was

```
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 sigle bit).
6) Wait for CPU shutdown or 3 minutes – whichever comes first.
        meanwhile, the DAS looks for exceptions, POST etc.
7) If the CPU was not shutdown
        shutdown the CPU.
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 12: Injection program used for phase 1 of the validation experiment.

determined through experimentation to be sufficiently long enough so that the probability of an error being detected three minutes after the injection is negligible. Examples of errors that are not detected include errors injected into machine instructions that are never or rarely executed (e.g., an if clause that is never true) and flipping bits in an unused portion of an instruction word. This phase of the experiment is referred to as a 'controlled' experiment because only one error is injected at a time and the system is always reset to a known state before injecting the next error (step 8 in figure 12).

The purpose of the **second phase** of the experiment was to collect the CPU shutdown distribution and the distribution of the number of undetected, latent errors present in the CPU board prior to a shutdown [4]. These distributions are then compared with those generated with the DEPEND Integrity simulation model. The second phase of the experiment was an 'uncontrolled' experiment in which errors were injected randomly without waiting to determine the effect of the previous errors injected and without returning the system to a known state before each injection. Errors arrived based on an exponential distribution with a mean of 3 minutes. Such an accelerated rate was used

[4]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.

to ensure that enough CPU shutdown events would be collected for a meaningful comparison with the simulation results. The injection program used is shown in figure 13.

```
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 13: Injection program used for phase 2 of the validation experiment.

A summary of both phases of the experiment is shown in table 3. In the first phase, 734 errors

Phase	Duration (hrs.)	No. Injections	No. CPU Shutdowns
Phase 1	93.5	734	476
Phase 2	28.0	414	247

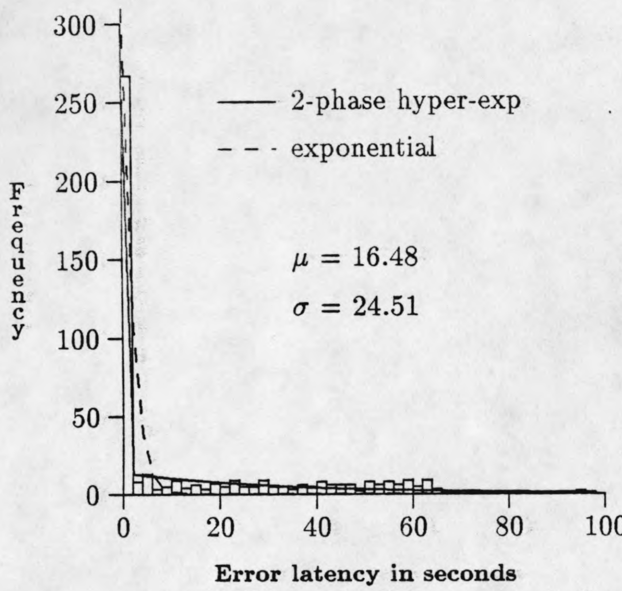
Table 3: Validation experiment statistics.

were injected continuously over a period exceeding three days (93.5 hours). During this period, other users continued to use the Integrity system. As a result, the workload was not fixed and varied with the number of users and the time of day. The errors, however, were only injected into the two target applications. The second phase of the experiment was conducted under similar conditions and lasted for 28 hours, during which 414 errors were injected.

5.1.2 Results from Phase 1: Measurement of Error Latency

The error latency and isolation latency distributions measured from the first injection experiment is shown in figure 14. The mean error latency was found to be 16.48 seconds for 300-by-300 element matrices. The mean latency was found to vary considerably depending on the size on the matrices. With 15-by-15 matrices, the mean latency was approximately 1 second.

Non-linear regression techniques from a statistical analysis package [SAS], were used to fit the empirical distributions with common exponential polynomials. The fitted functions are also shown in figure 14 and their parameters and an indication of the goodness of fit (the r^2 value) are given in table 4. Both a 2-phase hyperexponential and an exponential distribution were fitted to the empirical error latency distribution. The figure shows that the hyperexponential distribution is a reasonably good fit and approximates the percent of small and large error latencies well. The exponential distribution overestimates the number of small latent errors and underestimates the number of large latent errors. The isolation distribution, the time from the first exception to



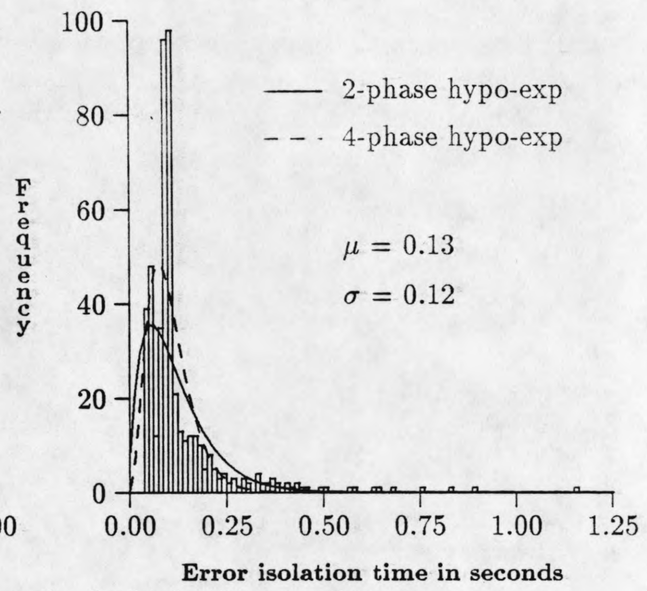


Figure 14: Measurements from the first injection experiment.

Error Latency			Error Isolation		
Type	Parameters	r^2	Type	Parameter	r^2
2-phase hyperexponential	$\alpha_1 = 0.5, \alpha_2 = 0.5$ $\lambda_1 = 0.03, \lambda_2 = 5.94$	0.89	2-phase hypoexponential	$\lambda_1 = 16.32$ $\lambda_2 = 16.36$	0.55
exponential	$\lambda = 0.5257$	0.819	4-phase hypoexponential	$\lambda_1 = 34.93$, $\lambda_2 = 31.26$ $\lambda_3 = 54.66$, $\lambda_4 = 34.13$	0.67

Table 4: The fitted exponential polynomials.

the time POST is initiated, has an unusually large peak which makes it difficult to fit. The distribution has a peak that indicates that a majority of the errors are isolated within 0.1125 seconds followed by a large tail which represents those infrequent cases that take more time. As discussed in [Geist 90], similar behavior has been observed by several experimenters, including [Finelli 87] and [McGough 83]. The empirical isolation distribution was fitted with a 2-stage and a 4-stage hypoexponential distribution. Both fail to capture the peak and indicates that a mixed distribution may be needed for a better fit [Geist 90]. However, since the isolation distribution is a small fraction of the total error latency, the total latency being the sum of error latency and isolation latency, additional effort to find a proper fit was abandoned. Instead, the simpler 2-stage hypoexponential distribution was used in the simulations and found to produce adequate results.

5.1.3 Results from Phase 2: Comparison of Simulation and Actual Injection Results

The fitted hyperexponential error latency and the 2-stage hypoexponential isolation latency distributions obtained from phase 1 of the experiment were used in the DEPEND Integrity simulation model to represent the total error latency of the injected errors. The system simulation was modified to only inject errors into one CPU. The simulation injection scenario used was identical to that outlined in figure 13. 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 15 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.

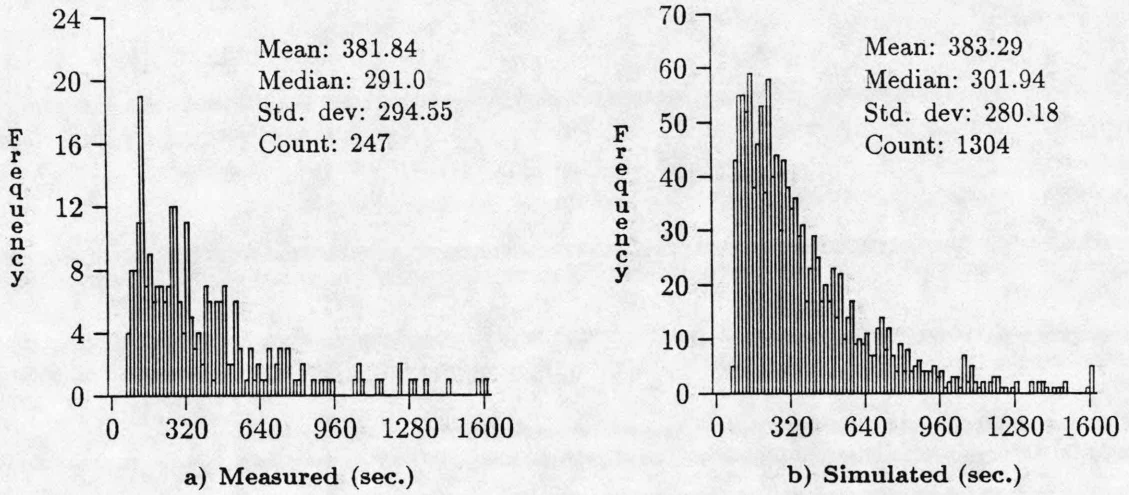


Figure 15: The Time to CPU shutdown distribution (in seconds).

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. The simulation model was also executed with the fitted exponential latency distribution (see table 4) and found to produce a mean time of 368.3 seconds between CPU shutdowns. The mean time is smaller because the fitted exponential error latency distribution overestimates the number of errors with small latencies. Figure 16 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.

These results demonstrate that the simulation model of the Integrity system is valid and that DEPEND is capable of capturing the intricacies of a real system. The results also indicate that a single error latency distribution can be used to represent the dynamically changing system activity and workload so that accurate dependability measures can be obtained. The distribution in figure 15a was obtained from an actual machine over a period of 28 hours while several users used the system, the workload changed with the time of day and paging, swapping and other system activities continued undeterred. The distribution shown in figure 15b was obtained from the simulation model in which all of the system activity was replaced by a single latency distribution. The statistically identical parameters of the CPU shutdown distributions and their similar shapes indicate that modeling system activity and memory read/write access may be eliminated and replaced with a latency distribution for the purpose of obtaining dependability measures. This simplification makes it possible to model larger systems for extended periods of time without sacrificing accuracy. Furthermore, in cases where the latency can be fitted with an exponential polynomial, it may be possible to model the error latency process with analytical techniques.

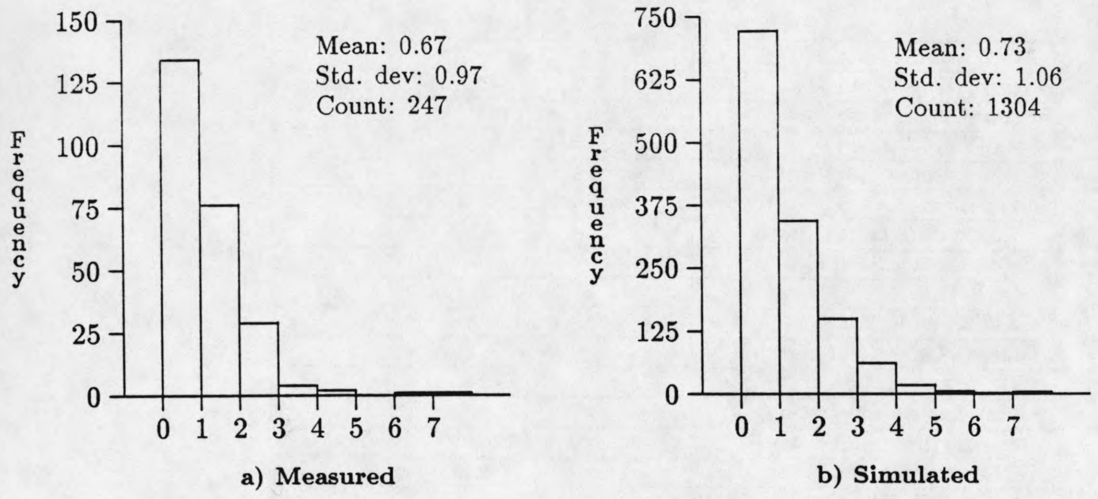


Figure 16: Distribution of the number of latent errors prior to a CPU shutdown.

6 Simulation-based Fault Injection Study

Having verified the simulation model, it is now used to analyze the behavior of the TMR-based system under correlated and latent errors and with memory scrubbing and various re-integration times. These experiments illustrate some of the capabilities of DEPEND and the sorts of analysis that is possible with such a tool. The parameters used in the simulation experiments are listed in table 5[5]. The experiments were conducted in phases to isolate and determine the impact these various parameters have on system reliability. The system was simulated for time periods ranging from 30 years to 2000 years. Each simulation was run twenty times with different random seeds and the averages of these repeated executions are shown here.

Except where explicitly stated, the simulations were 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 MTBF figures presented in this paper should not be construed to reflect the MTBF figures of an actual Tandem Integrity system because the error arrival rate and the error latency, which have a direct bearing on this measure, were *not* obtained from measurements of the Tandem Integrity system but rather from other production machines. For this reason, the results shown in this paper should only be construed to reflect the trend and behavior of a Tandem Integrity like machine.

6.1 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. To isolate the impact of correlated errors, only active errors are injected. Correlation factors of 0, 1

[5]Normally distributed error latencies based on [Chillarege 87] are used because they were collected over large periods of time whereas our latency measurements were for validation and were obtained under high acceleration conditions. Nevertheless, DEPEND can use both, as has been established.

Error Arrival Rate λ_1	1/24 hrs
Error Arrival Rate λ_2	1/72 hrs
Error Arrival Rate λ_3	1/120 hrs
Small Latency (sm)	mean = 44 min. std = 29 min.
Large Latency (lg)	mean = 8 hrs. std = 4 hrs.
Extra Large Latency (xl)	mean = 36 hrs. std = 18 hrs.
Percent Correlated Errors	0, 1, 2
Percent Latent errors	85
POST Time	1, 10, 30, 60 sec.
CPU Re-integration	1.5 sec.
Global Mem. Re-integration	0.5, 2, 5, 10 min.
IOP Re-integration	15 sec.
Controller Re-integration	30 sec.
Recovery from bad block	30 sec.
Disk Mirroring time (150 MB)	20 min.
System Reboot time	10 min.
I/O Error Arrival Intervals	5,8,12,15,18,21,24,27,48,72,96 hours

Table 5: Simulation parameters.

and 2 percent were used. Figure 17 shows the results for error arrival rates λ_1 and λ_2 for various global memory re-integration times. The figure shows that there is a significant reduction in the MTBF when correlated errors are injected. However, because only active errors are injected, all errors injected are detected and all occurrences of correlated errors result in a system failure.

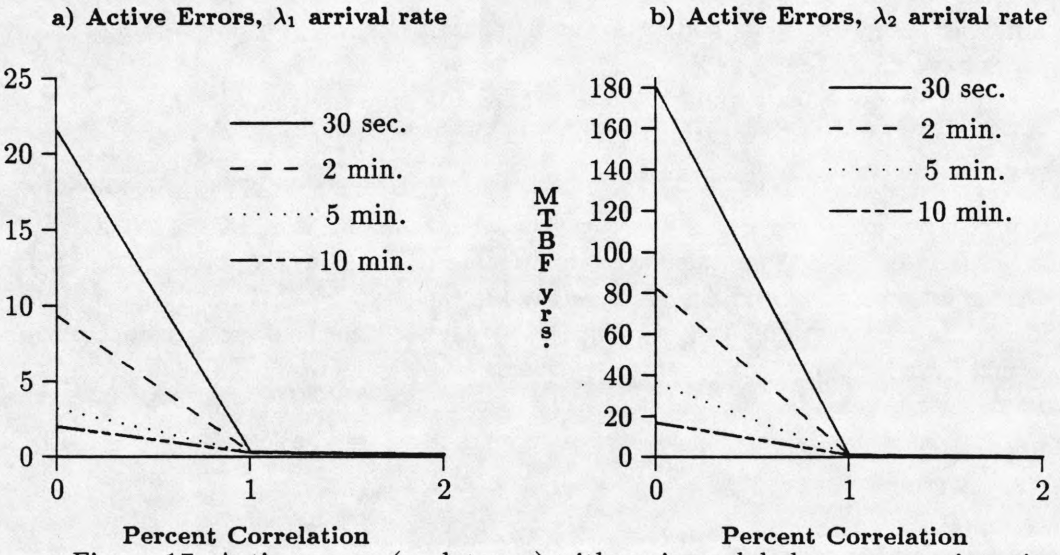


Figure 17: Active errors (no latency) with various global memory re-int. times

These results are therefore very pessimistic. Latency tends to decrease the probability of two errors occurring simultaneously and reduces the degradation caused by correlation. The next subsection presents results when *latent* correlated errors are injected.

The figures also illustrate the degradation in reliability as the global memory re-integration time is increased. Increasing the re-integration time from 30 seconds to 5 minutes (a 10-fold increase) reduces the MTBF by a factor of 5 for λ_2. The results demonstrate that the MTBF for a TMR system is quite sensitive to the re-integration times. Since the global memory re-integration times for the Integrity system increases with increasing workload, the system reliability will be much lower if the machine is heavily used. To avoid this, the priority of the background re-integration process can be increased to ensure that it always takes approximately 30 seconds (for 32Mbyte boards) and is workload independent. Whether the performance cost is acceptable will depend on the use of the S2. If the Integrity system is used as a part of a cash station system, a 30 second glitch may be acceptable whereas it may be problematic for a manufacturing system. Finally, since the re-integration times also depend on the size of the memory, larger systems will have smaller MTBFs.

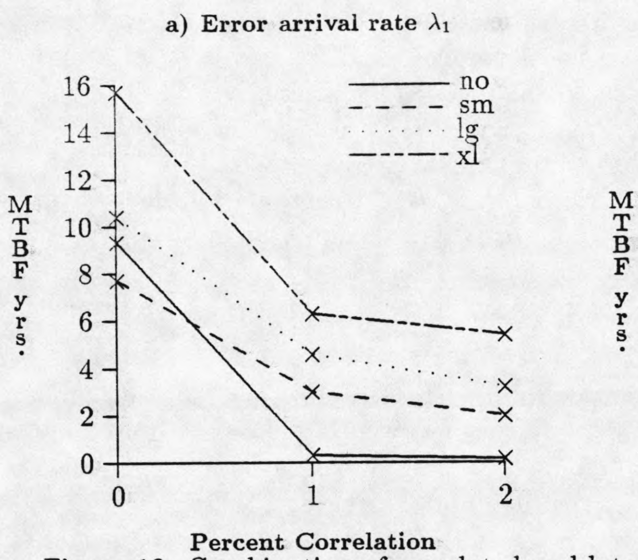
The performance overhead for voting was also measured during the experiments and was found to be 3.36% with the assumptions that the processors only vote every 2047 cycles and that they all arrive at the voter at the same time. This is the minimum voting overhead because in actual operation, the CPUs are likely to vote more often (i.e., they vote whenever they access global memory or perform I/O) and they will typically never reach the voter at the same time, leaving the early comers to idle waiting for the slowest processor.

Three different re-integration policies: 1) the CPU has priority and aborts global memory re-integration in progress, 2) global memory re-integration has priority and aborts CPU re-integration in progress, 3) and both have equal priority, were simulated and found not to have any significant impact on system MTBF. This is because the probability of simultaneous re-integrations by a CPU and a global memory was extremely small and hence all three policies behaved identically.

6.2 Impact of Latent and Correlated Errors

Since in reality errors have latency, the experiments above are repeated except this time correlated, latent errors are injected. All errors injected into the global memory have latency. Eighty-five percent of the errors injected in the CPU boards have latency and the remaining 15% are active errors. Since 2 or 3 correlated *active* errors will always cause the system to fail, the maximum system MTBF attainable under these error conditions can be approximated by multiplying the probability of injecting 2 or 3 active correlated errors with the error arrival rate of the processor subsystem. Using the equations in the appendix, the approximate maximum MTBF for λ_1 is 8.66 years and for λ_2 is 25.98 years for a correlation factor of 2%.

Figure 18 graphs the change in the MTBF for error arrival rates of λ_1 and λ_2. Scrubbing was not activated and a POST time of 60 seconds and a global memory re-integration time of 2 minutes was used. There are three things worth noting. First, the figure shows that the degradation in the MTBF when correlated *latent* errors are injected is not as significant as when correlated errors with no latency are injected. For example, for λ_2, with 1% correlated errors and small latency, the degradation in the MTBF is 5-fold, whereas the degradation is over 80-fold when the errors have no latency. These results indicate that error latency tends to counteract the effect of correlated errors. This is because latent errors remain dormant for some period of time and reduce the probability of their simultaneous detection. Similar trends were observed for the other error arrival rates. Second, the MTBF values in the graph are all less than the maximum MTBF calculated above. This indicates that correlated active errors are not the sole cause of all the system



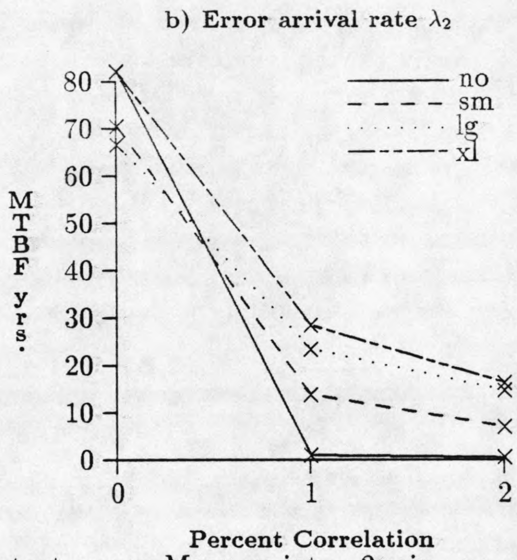
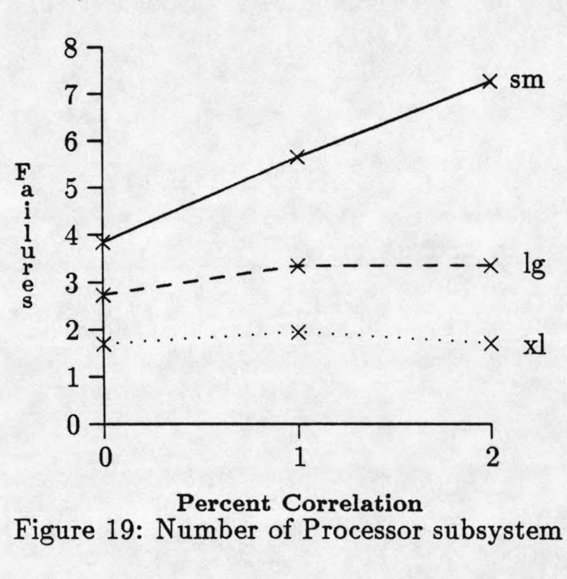


Figure 18: Combination of correlated and latent errors, Mem. re-int = 2 min., no scrubbing.

crashes and that correlated *latent* errors also impair system reliability. This is also borne out by figure 19 which shows that the number of times the entire global memory subsystem fails increases



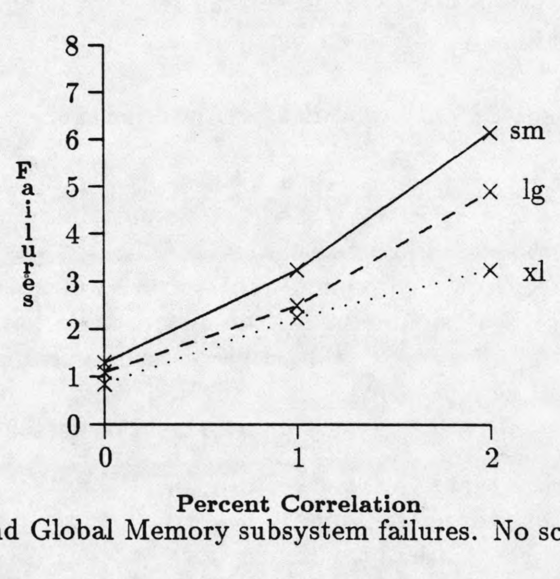


Figure 19: Number of Processor subsystem and Global Memory subsystem failures. No scrubbing.

with increasing correlation. Since no active errors are injected into the global memory, only the increase in correlated, latent errors can account for the increase. Similarly, the number of processor subsystem failures would have been identical for all latencies if correlated latent errors did not contribute to the failures (figure 19b).

6.3 Error Injection with Scrubbing

In this experiment, memory scrubbing is activated to see how well it protects the system from correlated, latent errors. Four scrubbing intervals, 1 hour, 4 hours, 8 hours and 24 hours, are used.

Figure 20 shows the impact of scrubbing on the system MTBF for the various scrubbing intervals for the three latencies with error arrival rate λ_1. The case where scrubbing is not used (scrubbing interval = 0 in the figure) is also included in the graphs for comparison purposes. Figure 20a

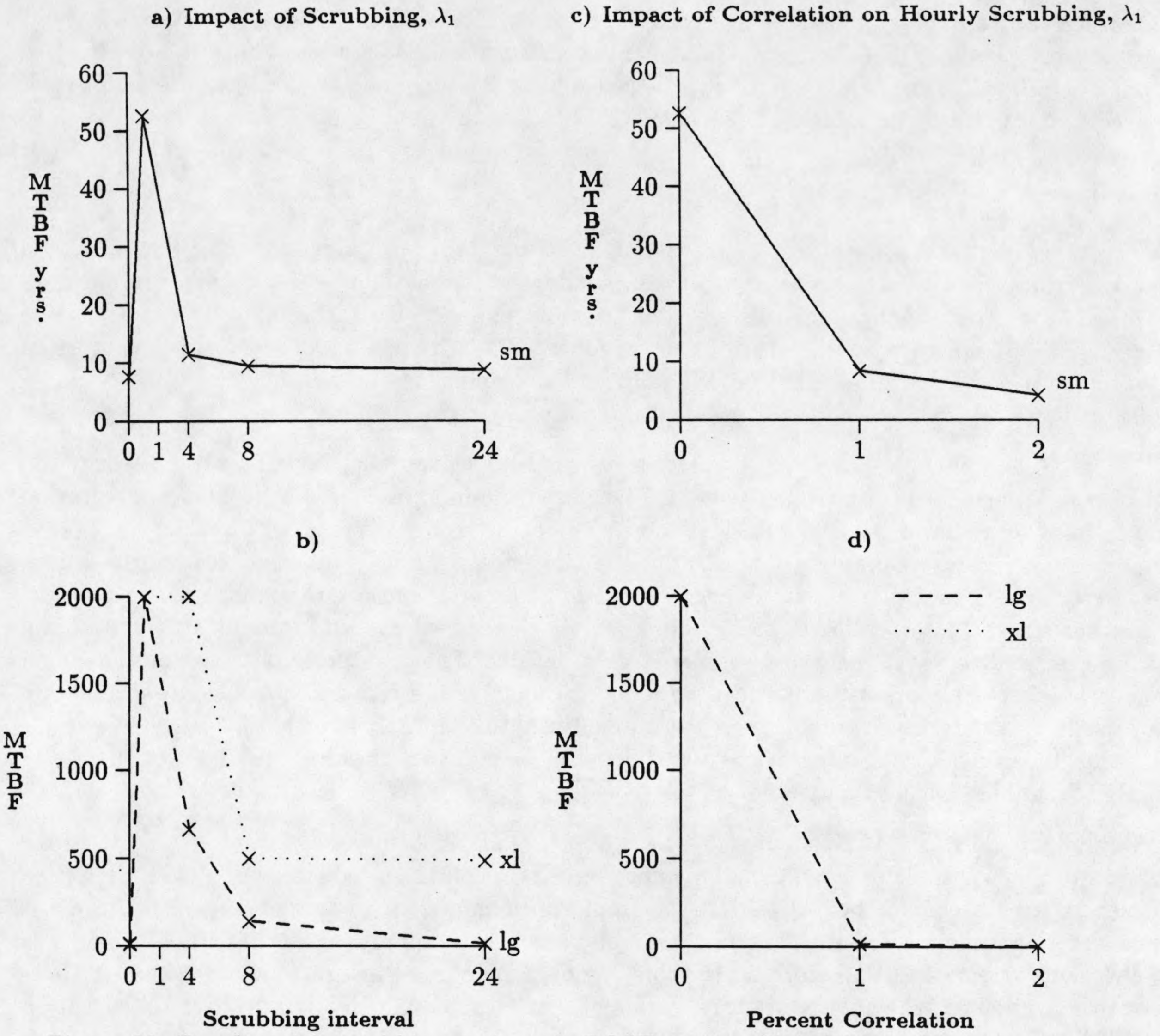


Figure 20: Results with scrubbing activated. POST = 60s. Global Mem. Re-int. = 2 min.

shows that with small latency (mean of 44 minutes) frequent scrubbing is necessary. With hourly scrubbing, the MTBF increases nearly 7 times over the non-scrubbing case. When the scrubbing frequency is decreased to once in 4 hours, there is only a 1.5 times increase in the MTBF. For larger latencies, scrubbing every 8 hours or just once a day provides good results as seen in figure 20b. Identical trends were seen for the other error arrival rates. The results indicate that the scrubbing frequency should be based on the size of the expected latency and not on the error arrival rate. Here again there is a performance/reliability tradeoff because latency decreases with increasing workload and that is precisely when frequent scrubbing is required. Fortunately, the overhead for scrubbing is not significant. The overhead for scrubbing the global memory and the local memory hourly was only 0.34% of the processing time.

Figure 20c and 20d show that the improvement in system reliability achieved by scrubbing diminishes substantially when there are correlated errors. For example, with 1% correlated, small latent errors, the MTBF falls from 52.63 years to 8.4 years. Though scrubbing is not effective against correlated errors, the experiments revealed that it still tends to decrease the number of near-coincident faults and increase the availability of the individual components.

6.4 Impact of POST Time

The system simulated 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 will fail. The previous experiments show that the MTBF of the system is quite sensitive to the size of this window and so in this experiment we simulate the Integrity system with various CPU POST and global memory re-integration times to determine which of these repair times have the greatest impact on system reliability.

So far all experiments have been conducted with a 61.5 second CPU repair time; 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 reduced but the POST time can be cut by using different self-checking programs. Since most errors are caused by transient faults, an effective policy may be to perform only a perfunctory check that takes just a few seconds and immediately initiate the re-integration process. If another error is detected in the same board shortly thereafter, a more thorough POST program can be executed to see if there are any 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 experiment, simulations are conducted with POST times of 1, 10, 30 and 60 seconds with the global memory re-integrations times listed in table 5 to determine their impact on system MTBF. Scrubbing was not activated.

Figure 21a plots the MTBF figures for the various POST times for small latent errors with error arrival rate λ_1. With a 30 second global memory re-integration time, reducing the POST time from 60 seconds to 1 second improves the MTBF from approximately 20 years to 50 years. However, when the global memory re-integration time is increased to 2 minutes, reducing POST times has no effect on system MTBF. According to table 2, global memory re-integration time for a 32Mbyte system is 2 minutes when the system is working at half capacity. Clearly, given the assumption that 62%[6] of all errors are injected into the global memory, the "reliability bottleneck" is the large re-integration time of the global memory. Rather than trying to reduce CPU POST times, the designers should focus their efforts on reducing global memory re-integration times.

The results of simulations where correlated errors are injected are shown in figure 21b. For these simulations, the global memory's re-integration time was kept fixed at 30 seconds. Again, in this experiment we see that the presence of correlated errors diminishes any gains achieved by reducing the CPU POST times.

[6]This number was determined based on the size of the global memory and the CPU local memories. See section 4.3.2.

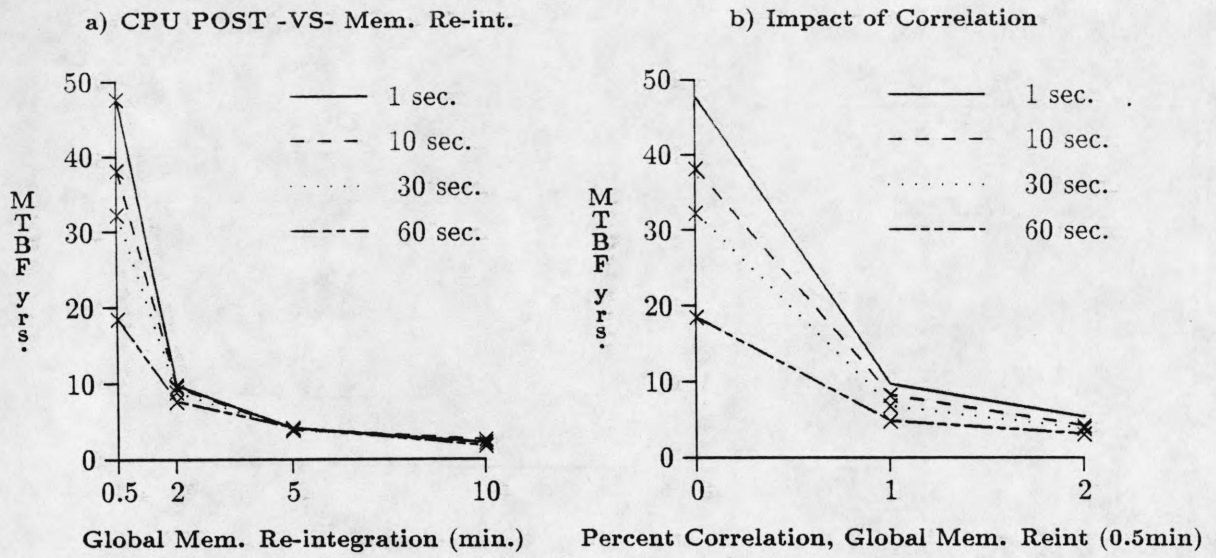


Figure 21: System reliability for various subsystem re-integration times.

6.5 Impact of the I/O Subsystem

In this experiment, the I/O subsystem is included in the simulation to determine its impact on system reliability. The I/O subsystem was injected with errors based on an exponential distribution with the means listed in table 5. For a given error arrival rate, 5% were injected into the IOPs, 34% into the controllers, 60% were injected as recoverable disk errors and 1% as major disk errors.

Figure 22 shows the results of simulations with a POST time of 60 seconds, a global memory re-integration time of 2 minutes and λ_1 as the combined error arrival rate for the CPU and the global memory. The figure shows that the original "without I/O" MTBF figure are not reached until the I/O error arrival rate is slightly slower than λ_1. Most of the I/O failures were due to controller failures which in turn caused disk mirroring to be lost. Since it takes 20 minutes to re-mirror a disk, a controller failure creates a large window of vulnerability for the system. The I/O subsystem is the reliability bottleneck until the I/O error arrival rate falls to less than 2 errors a day (figure 22b).

7 Conclusion

The objective of this paper was to introduce DEPEND and demonstrate its capabilities and its utility with a real example. The paper described the unique object-oriented approach used by DEPEND to provide a flexible framework which can model various fault-tolerant architectures. It also demonstrated how the tool can facilitate the evaluation of a complete system and help to conduct extensive fault-injection experiments with minimal effort.

The key features of DEPEND were illustrated by analyzing the Tandem Integrity fault-tolerant system. Specifically, the effect of near-coincident errors caused by correlated and latent faults was analyzed. Issues such as memory scrubbing, re-integration policies and workload dependent repair times, which affect how the system handles near-coincident errors were also evaluated. Other issues

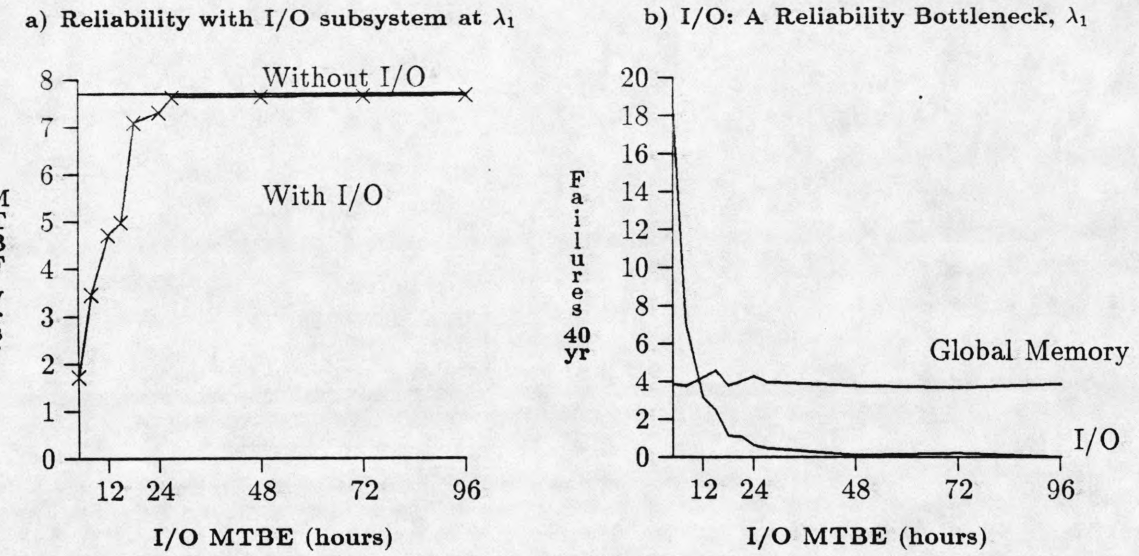


Figure 22: System Reliability with I/O Subsystem included. No Scrubbing.

such as the simulation of error latency and the time acceleration technique that provides significant simulation speed up were discussed.

Results obtained using DEPEND were validated by comparing with measurements obtained from fault injection experiments conducted on a production Tandem Integrity system. A hybrid monitoring testbed was used to conduct the injection experiments. The measured error latency distribution for a workload consisting of two Gaussian elimination programs was found to fit a hyperexponential distribution. The mean latency time was found to vary significantly depending on the size of the matrix being solved. The results of the validation experiments demonstrated that DEPEND is capable of capturing the intricacies of a real system. The results also verified that the technique used to simulate error latency adequately represents the workload and the dynamically changing system and user activity so that precise computation of dependability measures can be obtained.

The simulation model was used to inject near-coincident errors and evaluate the behavior of the Integrity system. Results from the fault injection study clearly showed that there is substantial degradation in system reliability due to transient, correlated errors. Even correlated errors with large latencies (larger latencies reduce the probability that a pair of correlated errors will be detected simultaneously or near-coincidentally) were found to significantly impair system reliability and diminish much of the benefit derived from reliability improvement strategies such as memory scrubbing and reducing re-integration times. Several re-integration schemes were modeled and found to behave similarly. Experiments in which various CPU and workload dependent global memory re-integration times were tried revealed that if the machine is working at half its capacity, the global memory re-integration time is the reliability bottleneck and that designers should try to reduce this time rather than CPU re-integration times. The weak point of the I/O subsystem is the time needed to mirror a disk, coupled with the fact that a faulty controller causes all the disks attached to it to be inaccessible thus losing disk mirroring.

8 Acknowledgements

This work was supported by NASA grant NAG-1-613 at the Illinois Computer Laboratory for Aerospace Systems and Software (ICLASS), a NASA Center of Excellence. This work would not have been possible without the help of Doug Jewett, Bob Horst and Carlos Alonso who have furnished many of the details of the Tandem Integrity system and have given useful feedback about DEPEND and the Tandem Integrity simulation. The authors would like to thank In-hwan Lee for his detailed and valuable suggestions regarding this paper and, Dong Tang for discussions that were useful in designing the experiments. Special thanks go to Bob Dimpsey, John Murphy, Krishna Subramanian, Karyn Williamson and Luke Young for their proof-reading and their helpful suggestions.

9 Appendix

The probability of 2 or 3 active correlated errors ($P[ACTERR]$) is computed as follows:

$$P[ACTERR] = P[CR] \times (P[2AE|2EI] * P[2EI] + P[23AE|3EI] * P[3EI]) \quad (1)$$

where

- $P[CR]$ is the probability of a correlated error.
- $P[2AE]$ is the probability of 2 active errors.
- $P[2EI]$ is the probability that there are 2 correlated errors given that this is a correlated error.
- $P[23AE]$ is the probability of 2 or 3 active errors.
- $P[3EI]$ is the probability that there are 3 correlated errors given that this is a correlated error.

$$P[2AE|2EI] = 0.15^2 \quad (2)$$

$$P[23AE|3EI] = 0.15^3 + 0.15^2 \times 0.85 \times 3 \quad (3)$$

Here, $P[CR]$ is simply the percent of correlated errors and is either 0, 0.01 or 0.02 and $P[2EI] = P[3EI] = 0.5$. The maximum MTBF is then:

$$MTBF_{maximum} = \frac{1}{P[ACTERR] \times \lambda_{CPU}} \quad (4)$$

where λ_{CPU} is the error arrival rate of the processor subsystem.

Bibliography

[Arlat 88] J. Arlat, Y. Crouzet and J. Laprie, "Fault-injection for dependability validation," *LAAS Research Report no. 88-363*, December 1988.

[Castillo 82] X. Castillo and D. Siewiorek, "A Workload Dependent Software Reliability Prediction Model," *12th Int. Symp. on Fault-Tolerant Computing*, Santa Monica, Ca., June, 1982.

[Chillarege 87] R. Chillarege and R. K. Iyer, "Measurement-Based Analysis of Error Latency," *IEEE Trans. on Computers*, Vol. C-36, No. 5, May 1987.

[Choi 89] G. S. Choi, R. K. Iyer and V. Carreno, "FOCUS: An Experimental Environment for Validation of Fault Tolerant Systmes: A case study of a Jet Engine Controller," *IEEE Inter. Conf. on Computer Design (ICCD)*, Cambridge, MA, October, 1989, pp. 561-564.

[Courtois 79] B. Courtois, "Some results about the efficiency of simple mechanisms for the detection of microcomputer malfunctions," *9th Int. Symp. on Fault-Tolerant Computing*, pp. 71-74, June 1979.

[Czeck 91] E. W. Czeck, "On The Prediction of Fault Behavior Based on Workload," *Ph.D. Thesis, Dept. of Electrical and Computer Engineering,* Carnegie Mellon University, April 19, 1991.

[Dupuy 90] A. Dupuy, J. Schwartz, Y. Yemini, and D. Bacon, "NEST: A Network Simulation and Prototyping Testbed", *Communications of the ACM*, Vol. 33, No. 10, October 1990, pp. 64-74.

[Finelli 87] Geaorge B. Finelli, "Characterization of Fault Recovery through Fault Injecton on FTMP," *IEEE Trans. on Reliability*, Vol. R-36, No. 2, June 1987, pg. 164-170.

[Geist 90] R. Geist, M. Smotherman and R. Talley, "Modeling Recovery Time Distributions in Ultrareliable Fault-Tolerant Systems," *20th Int. Symp. on Fault-Tolerant Computing*, June 26, 1990, pg. 499-504.

[Goswami 90a] K. K. Goswami and R. K. Iyer, "DEPEND: A Design Environment for Prediction and Evaluation of System Dependability," *9th Digital Avionics Systems Conference*, October 15, 1990.

[Goswami 90b] K. K. Goswami and R. K. Iyer, "The DEPEND Reference Manual," *CRHC Technical Report*, January 10, 1991.

[Goswami 91] K. K. Goswami and R. K. Iyer, "A Simulation-Based Study of a Triple Modular Redundant System using DEPEND", *FTRS-91*, September 1991, Nurnberg, Germany.

[Goswami 92] K. K. Goswami and R. K. Iyer, "An Investigation and Application of Hybrid Simulation for Dependability Analysis," *CRHC Technical Report*, February 21, 1992.

[Goyal 86] A. Goyal, W. C. Carter, E. de Souza e Silva, and S. S. Lavenborg, "The system availability estimator," *Proc. 16th Int. Symp. Fault-Tolerant Computing*, Vienna, Austria, July 1986, pp. 84-89.

[Hsueh 88] M. C. Hsueh, R. K. Iyer and K. S. Trivedi, "Performability Modeling Based on Real Data: A Case Study," *IEEE Trans. on Computing*, Vol. 37, No. 4, April 1988.

[Huang 84] Kuang-Hua Huang and Jacob A. Abraham, "Algorithm-Based Fault Tolerance for Matrix Operations," *IEEE Trans. on Computing*, Vol. c-33, No. 6, June 1984.

[Hwang 89] D. K. Hwang and W. K. Fuchs, "CSP-Based Object Oriented Description of Parallel Reconfigurable Architectures," *Proc. IEEE Intl. Conf. on Wafer-Scale Integration*, Jan. 1989, pp. 111-120.

[IEEE 88] IEEE Standard VHDL Language Refernce Manual -Std 1076-1987, *IEEE Press*, New York 1988.

[Iyer 82] R. K. Iyer, S. E. Butner and E. J. McCluskey, "A Statistical Failue/Load Relationship: Results of a Multicomputer Study," *IEEE Trans. on Computers*, Vol. SE-8, No. 4, July 1982, pp. 354-370.

[Jewett 91] D. Jewett, "Integrity S2: A Fault-Tolerant Unix Platform," *Proc. 21st Int. Symp. Fault-Tolerant Computing*, Montreal, Canada, June 1991.

[Johnson 91] A. M. Johnson and M. A. Schoenfelder, "Rainbow Net Analysis of VAXcluster System Availability," *IEEE Trans. on Reliability*, July 1991.

[Kanawati 92] G. Kanawait, N. Kanawati, and J. Abraham, "FERRARI: A Fault and ERRor Automatic Real-time Injector," *CERC Technical Report UT-CERC-TR-JAA92-01,* Computing Engineering Research Center, University of Texas, Austin, TX, 1992.

[Kubiak 89] K. Kubiak and W. K. Fuchs, "Reliability Analysis of Application-Specific Architectures," *Inter. Workshop on Defect and Fault Tolerance in VLSI Systems,* Oct. 1989.

[Lala 83] J. Lala, "Fault detection isolation and reconfiguration in FTMP: Methods and experimental results," *5th AIAA/IEEE Digital Avionics Systems Conference (DASC),* pp. 21.3.1-21.3.9, 1983.

[Law 82] A. M. Law and W. D. Kelton, "Simulation Modeling and Analysis", *McGraw Hill Book Company*, 1982.

[Lee 91] I. Lee, R.K. Iyer and D. Tang, "Error/Failure Analysis Using Event Logs from Fault Tolerant Systems," *Proc. of the 21st Inter. Symp. on Fault-Tolerant Computing*, Montreal, Canada, June 25-27, 1991.

[Lee 89] K. D. Lee, "PARAGRAPH: A Graphics Tool for Performance and Reliability Analysis," *UIUC Coordinated Science Laboratory Tech. Report UILU-ENG-89-2239*, Nov. 1989.

[Lewis 89] E. E. Lewis, F. Boehm, C. Kirsch, B. P. Kelkhoff, "Monte Carlo Simulation of Complex System Mission Reliability," *Proc. Winter Simulation Conf.,* pp. 497-504, 1989.

[Lomelino 86] D. Lomelino and R. Iyer, "Error propagation in a digital avionic processor: a simulation-based study," *Proc. Real Time Systems Symposium*, pp. 218-225, Dec. 1986.

[MacDougall 73] M. H. MacDougall and J.S. McAlpine, "Computer Simulation with ASPOL," *Symp. on the Simulation of Comp. Sys.*, ACM/SIGSIM, pp. 93-103, 1973.

[McGough 83] J. H. McGough, F. L. Swern and S. Bavuso, "New Results in Fault Latency Modelling," *16th Annual Electronics and Aerospace Conf.*, Washington, DC. Sept. 1983, pg. 299-306.

[Melamed 85] B. Melamed and R.J.T. Morris, "Visual Simulation: The Performance Analysis Workstation", *IEEE Computer*, vol. 18, no. 8, pp. 87-94, Aug. 1985.

[Meyer 88] J. F. Meyer and L. Wei, "Influence ow workload on error recovery in random access memories," *IEEE Trans. on Computers*, vol. C-37, no. 4, pp. 500-507, April 1988.

[Nicola 90] Victor F. Nicola, Marvin K. Nakayama, Philip Heidelberger and Ambuj Goyal, "Fast Simulation of Dependability Models with General Failure, Repair and Maintenance Processes," *Proc. of the 20st Inter. Symp. on Fault-Tolerant Computing*, Newcastle upon Tyne, England, June, 1990.

[Pawlikowski 90] K. Pawlikowski, "Steady-State Simulation of Queueing Processes: A Survey of Problems and Solutions," *ACM Computing Surveys*, Vol. 22, No. 2, June 1990.

[Sahner 87] R. A. Sahner and K. S. Trivedi, "Reliability modeling using SHARPE," *IEEE Trans. Reliability*, Vol. R-36, No. 2, June 1987, pp. 186-193.

[Sanders 86] W. H. Sanders and J. F. Meyer, "METASAN: A performability Evaluation Tool Based on Stochastic Activity Networks," *1986 Fall Joint Comp. Conf.*, Dallas, TX, Nov. 1986, pp. 807-816.

[SAS] SAS User's Guide: Statistics Version 5 Edition, *SAS Institute Inc.*, Cary, NC.

[Sauer 82] C.H. Sauer, E.A. MacNair and J.F. Kurose, "RESQ: CMS User's Guide," *IBM Research Report RA-139*, Yorktown Heights, N.Y., April 1982.

[Schwetman 86] H. Schwetman, "CSIM: A C-Based Process-Oriented Simulation Language," *Proc. Winter Simulation Conf.*, 1986.

[Segall 88] Z. Segall, D. Vrsalovic, D. Siewiorek, D. Yaskin, J. Kownacki, J. Barton, R. Dancey, A. Robinson, T. Lin, "FIAT - Fault Injection Based Automated Testing Environment," *Proc. 18th Int. Symp. Fault-Tolerant Computing*, June, 1988, pp. 102-107.

[SES 89] Scientific and Engineering Software, Inc., "SES/Sim Simulation Language Reference Manual, " Austin, TX, March 1989.

[Shahabuddin 88] Perwez Shahabuddin, Victor F. Nicola, Philip Heiderlberger, Ambuj Goyal and Peter W. Glynn, "Variance Reduction in Mean time to Failure Simulations," *Proc. Winter Simulation Conf.*, pg. 491-498, 1988.

[Shin 84a] K. Shin and Y. Lee, "Error detection process - model, design, and its impact on computer performance," *IEEE Transactions on Computers,* vol. C-33, pp. 529-540, June 1984.

[Shin 84b] K. Shin and Y. Lee, "Measurements of fault latency: methodology and experimental results," *Technical Report CRL-TR-45-84, Computing Research Laboratory,* University of Michigan, Ann Arbor, 1984.

[Shin 86] K. G. Shin and Y. H. Lee, "Measurement and Application of Fault Latency," *IEEE Transactions on Computers*, vol C-35, no. 4, pp. 370-375, 1986.

[Tang 90] D. Tang, R. K. Iyer, S. S. Subramani, "Failure Analysis and Modeling of a VAXcluster System," *Proc. 20th Int. Symp. Fault-Tolerant Computing*, Newcastle upon Tyne, England, June, 1990.

[Tang 91] D. Tang and R. K. Iyer, "Impact of Correlated Failures on Dependability in a VAXcluster System," *2nd IFIP Conf. on Dependable Computing for Critical Applications*, Tucson, Arizona, February, 1991.

[Young 92] Luke Young and R. K. Iyer, "A Hybrid MOnitor Assisted Fault Injection Environment," *Third IFIP Conf. on Dependable Computing for Critical Applications*, Sicily, Italy, Sept. 1992.