①

AUGUST 1994

UILU-ENG-94-2231
CRHC-94-16

Center for Reliable and High-Performance Computing

Experimental Study Of Software Dependability

Wei-lun Kao

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)	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 & Space Administration	
6c. ADDRESS *(City, State, and ZIP Code)* 1308 West Main Street Urbana, IL 61801	7b. ADDRESS *(City, State, and ZIP Code)* Hampton, VA
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)*
Experimental Study of Software Dependability

12. PERSONAL AUTHOR(S) Wei-lun Kao

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT *(Year, Month, Day)*	15. PAGE COUNT
Technical	FROM ______ TO ______	1994 July 28	77

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
FIELD	GROUP	SUB-GROUP	

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

In this study, a distributed fault injection and monitoring environment (DEFINE) has been developed. It consists of a target system, a fault injector, a software monitor, a workload generator, a controller, and several analysis utilities. DEFINE can inject software faults as well as hardware faults, can trace fault propagation in software systems and among machines, can monitor whether faults are activated and when the faults are activated, and has accurate time control. The fault models used are extracted from the results of field error data analyses and fault simulations. Fault injection experiments show that the majority of no-impact faults are latent. Memory faults and software faults usually have a very long latency, while bus faults and CPU faults tend to crash the system immediately. About half of the detected errors are data faults, and they are detected while the system is trying to access a memory location it has no privilege to access. Only about 8 of faults propagate to other UNIX subsystems. Fault propagation from servers to clients occurs more frequently than from clients to servers. The fault impact depends on the workload. Transient Markov reward analysis shows that the performance losses incurred by bus faults and CPU faults are much higher than those incurred by software and memory faults. Among software faults, the impact of pointer faults is higher than that of non-pointer faults.

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

EXPERIMENTAL STUDY OF SOFTWARE DEPENDABILITY

BY

WEI-LUN KAO

B.S., National Taiwan University, 1985
M.S., University of Illinois at Urbana-Champaign, 1991

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1994

Urbana, Illinois

EXPERIMENTAL STUDY OF SOFTWARE DEPENDABILITY

Wei-lun Kao, Ph.D.
Department of Computer Science
University of Illinois at Urbana-Champaign, 1994
Ravi K. Iyer, Advisor

There is trend of increasing demand for highly dependable software systems. The factors that influence the dependability of software systems include software faults and hardware faults. To improve software dependability, it is necessary to understand the characteristics of these faults and how they affect software systems. In this study, a distributed fault injection and monitoring environment (DEFINE) has been developed. It consists of a target system, a fault injector, a software monitor, a workload generator, a controller, and several analysis utilities. DEFINE can inject software faults as well as hardware faults, can trace fault propagation in software systems and among machines, can monitor whether faults are activated and when the faults are activated, and has accurate time control. The fault models used are extracted from the results of field error data analyses and fault simulations. Fault injection experiments on the UNIX kernel (SunOS 4.1.2) and the Sun Network File System are conducted to study fault impact and to investigate fault propagation. Three kinds of fault injections are conducted: uniform fault injection, biased fault injection, and path-based fault injection. Based on the experimental results, fault propagation models have been developed for both hardware and software faults, and transient Markov reward analysis has been performed to evaluate the loss of performance after a fault is injected.

Experimental results show that the majority of no-impact faults are latent. Memory faults and software faults usually have a very long latency, while bus faults and CPU faults tend to crash the system immediately. About half of the detected errors are data faults, and they are detected while the system is trying to access a memory location it has no privilege to access. Only about 8% of faults propagate to other UNIX subsystems. Fault propagation from servers to clients occurs more frequently than from clients to servers. The fault impact depends on the workload. Transient Markov reward analysis shows that the performance losses incurred by bus faults and CPU faults are much higher than those incurred by software and memory faults. Among software faults, the impact of pointer faults is higher than that of non-pointer faults.

Acknowledgements

I would like to thank my advisor, Professor Ravi K. Iyer, for his support, guidance, and encouragement during this research. I also wish to thank Professors Gul A. Agha, Roy H. Campbell, W. Kent Fuchs, and Jane W.-S. Liu for serving on my dissertation committee and for giving me suggestions for my research.

Thanks are due to friends at the Center for Reliable and High-Performance Computing. I would especially like to thank Dong Tang, Tim Tsai, Inhwan Lee, Daniel Olson, Rene Llames, and Kumar Goswami, who were always there to answer questions, review my papers, and offer suggestions.

I would like to thank my wife, Li-Ling Chen, for her love and moral support. Finally, I am indebted to my parents and family members for their love, care, and support.

Accesion For	
NTIS CRA&I	[✓]
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution /	
Availability Codes	
Dist	Avail and / or Special
A-1	

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

There is trend of increasing demand for highly dependable software systems. The first issue that arises is how to evaluate software dependability. The second issue is how to improve software dependability if it does not meet the specified criteria. One method of improving software dependability is to use fault-tolerant mechanisms to detect and recover errors. To design effective error detection and recovery mechanisms, it is necessary to understand the characteristics of faults and how they propagate in software systems. The next issue is how to validate fault-tolerant mechanisms.

Current approaches to studying these issues include analytical modeling, field error data analysis, fault simulation, and fault injection. Accurate analytical models of software behavior under faults are difficult to construct without the information provided by experimental analysis. Field error data analysis is useful for understanding actual error behavior but is restricted to those errors already detected. Fault simulation is easy to control and is able to reveal fault characteristics and propagation in detail if no unrealistic assumptions are made; however, this is time consuming and impractical for very large systems. Fault injection is an appropriate approach for studying fault propagation if sufficient information can be extracted from experimental results. In this study, a distributed fault injection and monitoring environment (DEFINE) has been developed, and fault injection experiments have been conducted to study fault impact and to investigate fault propagation. The fault models used are extracted from the results of field error data analyses and fault simulations. Based on the experimental results, fault propagation models have been developed, and transient Markov reward analysis has been performed to evaluate the loss of performance after a fault is injected.

1.1 Motivation

Factors influencing the dependability of software systems include software faults (software design/implementation faults) and hardware faults (hardware-induced software errors). Most of the previous fault injection studies emphasized hardware faults and assumed that software faults can be removed by software testing. However, field error report analyses from IBM, AT&T, and Tandem show that about 20% to 60% of system failures are due to software faults [Iyer85, Sullivan91, Levendel90, Gray90]. Therefore, it is necessary to develop a tool with which the impact of software faults can be studied.

Although fault injection has been applied to several computer systems, in most of these experiments faults or errors are injected into systems and their final impact on the systems is observed, with an emphasis on error latency and fault coverage. What really happens after a fault is injected and how a fault propagates in a software system are not well understood. In addition, without detailed monitoring, it is difficult to identify the vulnerable modules. To study these issues, a tool that can trace fault propagation in detail is needed.

By observing only the final impact, it is impossible to distinguish between avoided faults and non-activated faults if the faults do not have any impact. Without this information, it is difficult to evaluate the dependability of software systems or the effectiveness of fault-tolerant mechanisms accurately.

Accurate time control is important when injecting intermittent faults; otherwise, the distribution of interarrival times will not follow the specified distribution. Alternating executable files or memory images cannot control activation times accurately. In addition, CPU and bus faults have very high activation rates, but the aforementioned method cannot activate most of the injected faults. Therefore, a new injection method that can guarantee fault activation and can accurately control injection time and activation time is needed.

1.2 Target of the Study

Software behavior under faults depends strongly on the nature of software. Operating systems are chosen as the study target because all the user applications need support from the operating system. If the operating system crashes, all the applications running on that machine will crash. Operating systems have some properties that most user application do not have. They are:

- continuously running: the faults will remain in the kernel until they are recovered, the system is rebooted (damage may have been caused), or the system is crashed. Faults in user applications will be removed after the applications crash or finish.
- highly parameterized: many parameters and initial files are used.
- complex: they are written in both assembly and high-level languages, and a significant portion of the data processed are pointers.
- service programs: there is no certain output, as in application programs, to determine if the result is correct.
- of high impact: the kernel has high privilege so that the impact of faults may not be restricted to one process or one machine. User applications in the UNIX system cannot access the kernel text or data except through system calls or device drivers.
- of broad spectrum of workloads: workloads vary significantly in different sites and applications.

The UNIX kernel already has many assertions to detect errors, but they are not good enough to cover most of the errors. We have to understand fault propagation to design effective software assertions and to know what kind of assertions are needed. Although the target of this study is an operating system, the methodology can also be applied to user applications.

1.3 Thesis Overview

A brief description of software-implemented fault injection and related research is presented in Chapter 2. The approaches to injecting faults by software methods are classified and discussed. The techniques for enhancing the capabilities of software-implemented fault injection are also described.

Chapter 3 describes the fault models used in this study, including hardware and software faults. The models are extracted from field error report analyses and fault simulations.

In Chapter 4, the design and implementation of DEFINE, a distributed fault injection and monitoring environment, is explained and depicted. The methodology used in DEFINE is to inject faults on-line into the UNIX kernel and to trace the execution flow and key variables of the system. A comparison is then made between the faulty trace data and the fault-free one under the same workload to identify the fault propagation. The faults injected by DEFINE included both hardware faults (which induce software errors) and software faults. DEFINE is composed of a target system, a fault injector for injecting hardware and software faults, a software monitor for tracing the execution flow and key variables of the kernel, a workload generator for activating injected faults, a controller for controlling experiments, and several analysis utilities for analyzing fault propagation.

Experiments are conducted on SunOS 4.1.2 to show the fault propagation and the impact of various types of faults. The experimental results and discussions are presented in Chapter 5. The results show that memory faults and software faults usually have a very long latency, while bus faults and CPU faults tend to crash the system immediately. About half of the detected errors are data faults, and they are detected while the system is trying to access a memory location where it has no access privilege. The majority of no-impact faults are latent. Analysis of fault propagation among the UNIX subsystems reveals that only about 8% of faults propagate to other UNIX subsystems.

Chapter 6 describes the biased fault injection and path-based fault injection experiments conducted on the Sun Network File System. The experiments are conducted on six Sun workstations, one as server and the others as clients. Faults are injected into the clients and the server. Experimental results show that the fault impact depends on the workload, and fault propagation from servers to clients occurs more frequently than from clients to servers.

Based on the experimental results, fault propagation models for hardware and software faults are built and presented in Chapter 7. Transient Markov reward analysis is performed to evaluate the loss of performance after a fault is injected.

Chapter 8 provides a summary of this research and highlights the important conclusion. Suggestions for future study are also listed.

Chapter 2

Background

In fault injection studies, faults are injected by hardware or software. The hardware fault injection approach is to inject faults to IC pins by hardware instrumentation [Lala83, Shin84, Shin86, Finelli87, Arlat89, Arlat90] or to apply radiation rays to target components [Cusick85, Karlsson89, Gunneflo89]. Software fault injection changes source code, executable files, or the contents of memory or registers to simulate the occurrence of hardware or software faults [Segall88, Barton90, Chillarege89, Devarakonda90, Young92, Kanawati92, Rosenberg93, Han93, Kao93, Kao94], or to emulate certain types of software faults [Kao93, Kao94]. Hardware fault injection requires special hardware equipment and accessibility to the target components. In addition, software faults can be injected only by the software approach, since the faults are so particular that it is almost impossible to implement by the hardware approach. Therefore, software fault injection is applied in the study.

In this chapter, the assumptions and the definitions of some terms used in this study are listed in Section 2.1. Section 2.2 presents software-implemented fault injection approaches. Section 2.3 describes related research in this area and comparison. The objective of this study is presented in Section 2.4.

2.1 Assumptions and Terminology

In this fault injection study, two assumptions are made:

- The original system, including hardware and software, is perfect.

- All the abnormal behavior is due to the injected faults.

Several terms used in this study may be different from other studies; therefore, the definitions of these terms are given here to avoid confusion.

- Hardware fault: hardware-induced software errors, that is, those caused by hardware faults that are not detected by hardware detection mechanism and therefore propagate to software and affect software execution.
- Software fault: software design or implementation fault. It is also as known as software defect or software bug.
- Fault injection: changing the correct system to a faulty system. See Figure 2.1.
- Fault activation: executing or accessing the faulty part. Some studies call this fault manifestation. See Figure 2.1.
- Error detection: the system detects the error or the user observes the abnormal behavior. See Figure 2.1.
- Fault latency: the time between fault injection and fault activation. See Figure 2.1.
- Error latency: the time between fault activation and error detection. See Figure 2.1.
- Fault propagation: the chain effect of a fault after the fault is activated. It is also called error propagation.

2.2 Software-Implemented Fault Injection Approaches

The execution of software takes several steps, and faults can be injected at these steps. Figure 2.2 depicts the steps to execute software. The occurrence of faults is simulated by executing the modified instructions. To achieve this, faults can be injected at different steps—modifying source programs, libraries, executable code, or memory image. The next five subsections describe these approaches and discuss their advantages and disadvantages.

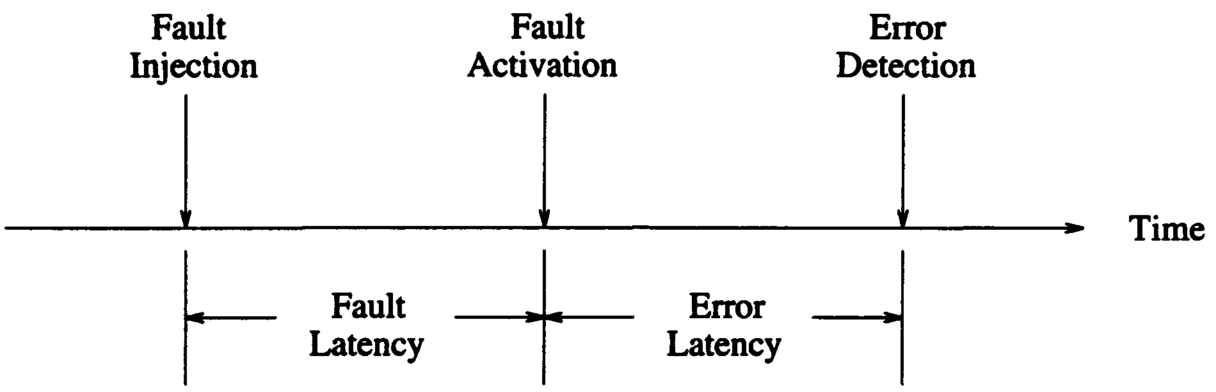

Figure 2.1 Some terminology used in fault injection.

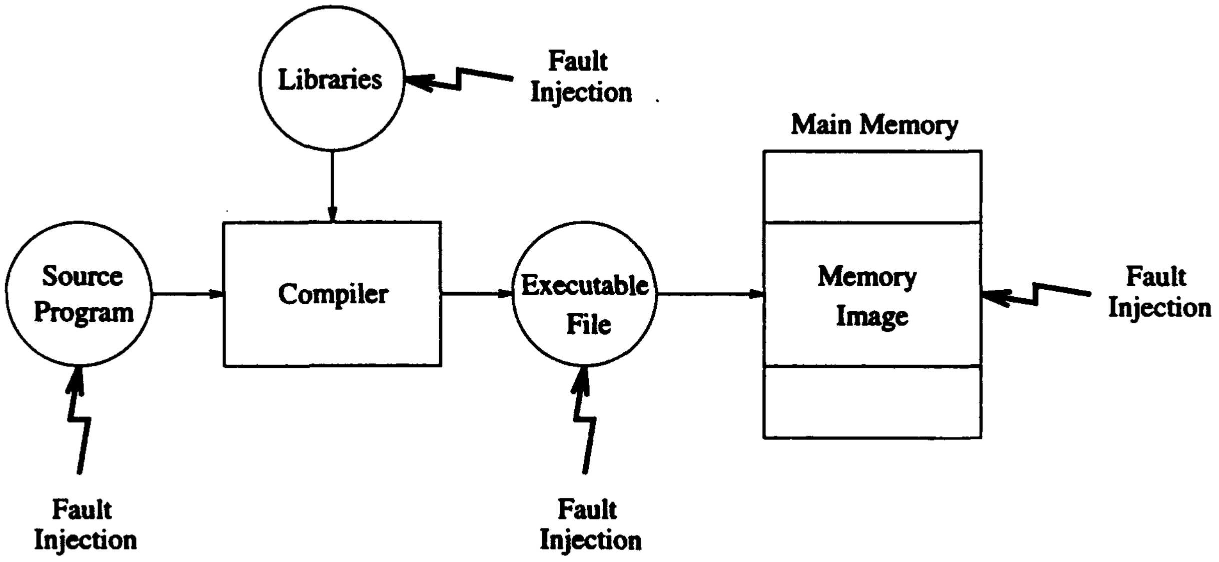

Figure 2.2 Software-implemented fault injection approaches.

2.2.1 Modification of Source Program

This approach is to change source programs to emulate certain types of faults. It is the best way to inject software faults because it is intuitive and natural. This approach is similar to mutation testing. However, it is difficult to inject certain types of hardware faults, such as CPU and bus faults. In addition, this approach requires recompilation of source code and reloading the code into memory. To inject different faults, the source programs need to be modified differently according to the fault type and then be recompiled and reloaded. The overhead is significant if the target system is a big software system, such as an operating system.

2.2.2 Modification of Library

This approach is to add fault injection modules into libraries and then link these modules into executable files. When the executable files are executed, the fault injection modules are called to inject faults.

This approach requires the target programs to be relinked with the augmented libraries. If the fault injection modules can inject various types of faults, the target programs need to be relinked only once.

2.2.3 Modification of Executable Code

This approach is to change or add instructions of executable code to inject faults. It is flexible, but the executable code needs to be modified and reloaded into memory each time a new fault is injected.

One variant of this approach is to change or add instructions of assembly code generated by a compiler and then assemble and link the assembly code into an executable file. The advantage of this variant is ease of control; it is easier to read, and libraries and symbol tables are not included. The disadvantage is the extra overhead of running assembler and linker.

2.2.4 Modification of Memory Image

This approach is to change the contents of memory and registers to inject faults. It has the lowest overhead, since no extra preparation is needed. However, it is not as flexible as modifying executable code, because the code has already been loaded into memory and all the modifications are limited in the existing space.

One variant is to use software traps to achieve better control over the target program.

2.2.5 Hybrid

It can be difficult to use only one fault injection approach when injecting more than one type of fault. Therefore, some tools use different approaches to inject different types of faults.

2.3 Related Research

In this section, recent works on software-implemented fault injection are briefly reviewed. The comparison of these works is presented in Table 2.1

FIAT [Segall88, Barton90]

FIAT (Fault Injection Automated Testing), developed in Carnegie Mellon University, injects faults by modifying memory images and is designed for distributed/parallel systems. The injection approach is hybrid: it augments fault injection library and modifies the memory image. It consists of two parts: the fault injection manager (FIM) and the fault injection receptor (FIRE). Injectable faults include memory, register, and communication faults.

Chillarege and Bowen's Work [Chillarege89]

Chillarege and Bowen of IBM T. J. Watson Research Center formalized the idea of failure acceleration and injected software errors by modifying physical memory. The injection approach is modification of memory image. The injected software errors are overlays. They have classified the failure types, measured loss of service, and identified potential candidates for repair before total failure.

Devarakonda, Goswami, and Chillarege's Work [Devarakonda90]

Devarakonda, Goswami, and Chillarege of IBM T. J. Watson Research Center injected software errors into the NFS request packages assuming that some unknown software faults corrupted the packages. The injection approach is modification of source programs. The injected software errors are overlays. They have classified the failure types and investigated the relationship between workloads and failure rates.

HYBRID [Young92]

HYBRID, developed in University of Illinois at Urbana-Champaign, injects faults by modifying physical memory and cache, and uses a hybrid monitor to reduce the overhead of monitoring. The injection approach is modification of memory image. Injectable faults include memory (main memory and cache), and register faults. HYBRID consists of a fault injection system, a hybrid monitor system, and a supervisory system. By using hybrid monitor, HYBRID can measure very short fault latency and error latency.

FERRARI [Kanawati92]

FERRARI (Fault and ERRor Automatic Real-time Injector), developed in the University of Texas at Austin, builds an extra layer, similar to a debugger, to control the experiments. The injection approach is modification of memory image. It introduced the method of using software traps to inject transient/permanent bus faults. Injectable faults include memory, control flow, bus, and condition code flag faults. communication (lost, altered, or delayed messages), and processor errors (adder or multiplier).

DOCTOR [Han93]

DOCTOR, developed in the University of Michigan at Ann Arbor, is an integrated software fault injection environment. The injection approach is hybrid: modification of source program, modification of executable code, and modification of memory image. It consists of a software fault injector (SFI [Rosenberg93]), a synthetic workload generator, and a software monitor. DOCTOR is designed for validating dependability mechanisms on an experimental distributed real-time system, HARTS. It can inject intermittent faults in addition to transient and permanent faults. The interarrival time between intermittent faults can be deterministic or follow a specified exponential distribution. Injectable faults include memory (code, global variables, or heap), communication (lost, altered, or delayed messages), and processor errors (adder or multiplier).

FINE [Kao93]

FINE, developed in The University of Illinois at Urbana-Champaign, was designed to inject software faults and hardware errors and to trace fault propagation in software systems. FINE is the predecessor of DEFINE, and all the the functions are covered in DEFINE. The injection approach is modification of memory image. It consists of a fault injector, a software monitor, a workload generator, a controller, and several analysis utilities. Injectable faults include memory (text, data, or stack segment), CPU (ALU, opcode decoder, or registers), bus (address or data lines), and communication (missing or corrupted messages) faults.

2.4 Objective of this Research

The objective of this study is to investigate the detailed (step by step) effect of hardware-induced software errors and software faults on software execution, to understand fault impact, to discover the most vulnerable portion of a system, and to provide feedback to system designers.

In this study, we emphasize and propose solutions to the problems mentioned in Section 1.1. Here is a brief description of the solution. The detailed implementation is depicted in Chapter 4.

- Software faults: substitute a set of instructions with another set on line to emulate certain types of software faults.
- Fault propagation: use a software monitor to trace control flow and key variables of software systems.
- Distinction of avoided faults and non-activated faults: use software traps to inject faults so that the time of fault activation can be recorded and avoided faults and so that non-activated faults can be distinguished.
- Accurate time control: use hardware clock interrupts to control fault injection and activation time.

Table 2.1 Comparison of software-implemented fault injections.

Tool	FIAT (1988)	Chillarege and Bowen (1989)	Devarakonda, Goswami, and Chillarege (1990)	HYBRID (1992)	FERRARI (1992)	DOCTOR (1993)	FINE (1993)
Target	O.S. User Appl.	O.S. User Appl.	O.S.	O.S. User Appl.	User Appl.	O.S. User Appl.	O.S. User Appl.
Testbed machine	IBM RT	IBM 3081-KX	IBM RT	Tandem S2	Sun SPARC	HARTS	Sun IPC
Fault Types	Memory Register Communication	Overlay (Software Errors)	NFS Request Packages (Software Errors)	Memory Register	Memory Bus CPU	Memory CPU Communication	Memory CPU Bus Software Faults
Fault Injector	Embedded in Target Task and O.S.	Special Program	Embedded in O.S.	Embedded in O.S.	Extra Layer	Embedded in Target Task and O.S.	Embedded in O.S.
Injection Techniques	Altering memory or registers	Altering physical memory	Modifying the XDR routines of the NFS	Altering physical memory, cache or memory-mapped registers	Using traps and altering memory or registers	Altering memory image, modifying O.S. routines	Using traps to alter memory or registers
Monitor	Software		Software	Hybrid	Software	Software	Software
Application	Error detection coverage	Failure characterization	Fault characterization	Fault latency Error latency Error detection Error recovery Recovery overhead	Fault latency Error detection Detection overhead	Error recovery	Fault latency Fault propagation Fault impact Error detection

Chapter 3

Fault Model

Faults that affect software execution include hardware faults that lead to software errors (hardware-induced software errors) and software faults (software design/implementation faults). Most fault injection tools do not consider the causes of software errors and inject either hardware faults or software errors. To our knowledge, there has been no tool that can emulate software design/implementation faults. DEFINE is capable of injecting both hardware-induced software errors (for simplicity, they are called hardware faults in this thesis) and software faults. This section describes the fault classes used in DEFINE. Faults are separated into hardware and software faults. The classifications of these two kinds of faults are shown in Sections 3.1 and 3.2.

3.1 Hardware Faults

Although hardware faults may occur in any part of a computer system, only those faults that can affect program execution directly or propagate to software, i.e., hardware-induced software errors, are considered. Based on the results of field error report analyses and fault simulations [Czeck92, Duba88, Iyer82, Iyer86, Yang92], such hardware faults are classified as *memory*, *CPU*, *bus*, *communication*, and *I/O faults*.

Memory Faults

Memory faults are those that corrupt the contents of a particular memory location. They can occur in text segments or data segments.

CPU Faults

CPU faults include computation, control flow, and register faults. From the software viewpoint, all these faults result in the corruption of registers. The corrupted registers can be general registers or special registers, such as the program counter (PC), the next program counter (nPC), the processor state register (PSR), or the stack pointer (SP).

Bus Faults

Bus faults can occur on address lines or data lines. They may affect bits in the instructions or data which are transmitted through the bus.

Communication Faults

Communication faults are those faults that occur during message t ansmission between machines. They may cause missing or corrupted messages. Both incoming and outgoing messages may be affected.

I/O Faults

I/O faults are from peripheral devices. Device drivers are designed to be able to handle these exception situations, so this type of fault will not be discussed in this thesis.

According to [Laprie85], all these faults are hardware errors. We treat them as fault sources from hardware that cause errors in software code or data. The faults may be permanent (stuck-at-1 or stuck-at-0) or transient (value flipped).

3.2 Software Faults

Field error reports from different software systems have been analyzed to uncover the cause, frequency, and correlation of software faults [Endres75, Chillarege91, Sullivan91, Tang92, Lee92]. These studies classify software faults according to their causes or symptoms. Automatic error logs available in several operating systems (e.g. in VAX/VMS and Tandem GUARDIAN) usually give information only on error symptoms (e.g., allocation, range, pointer, synchronization, and timing faults), because fault causes are not available from the logs. Analyses of human-collected error reports, especially from manufacturers, can usually provide insight into the causes (e.g., incorrect computation, using incorrect data, missing operation, data definition fault, and incorrect procedure interface) and their relation to the symptoms. Since fault causes and their locations are our key consideration in on-line fault injections (without recompilation for software faults), DEFINE employs a fault classification based on causes. Specifically, faults are classified into *initialization*, *assignment*, *condition check*, *function*, and *documentation faults*.

Initialization Faults

Initialization faults include uninitialized variables and wrongly initialized variables/parameters. The value of an uninitialized variable is compiler dependent. For C, the value is set to zero if the variable is global or unknown if the variable is local. A smart compiler should be able to detect most uninitialized variables. Wrongly initialized variables are similar to misassigned variables. Wrongly initialized parameters are those that are initialized to incorrect values (e.g., defining a small value to a parameter MAXPATHLEN). Incorrect arguments of function calls are also initialization faults, because the arguments are wrongly initialized for the callees.

Assignment Faults

Assignment faults can be missing assignments or incorrect assignments. A fault in an incorrect assignment may be in the right-hand side, causing one incorrect data value (e.g., using $a = b + c$ for $a = b + d$ corrupts a), or in the left-hand side, causing two incorrect data values (e.g., using $a = b + c$ for $d = b + c$ corrupts a and d).

Condition Check Faults

Condition check faults include missing condition checks (e.g., fail to check return values) and incorrect condition checks.

Function Faults

Function faults mean that the faulty parts are not single statement faults; that is, these faults are complicated, and the correction of this type of fault involves multi-statement modification or function rewriting.

Documentation Faults

Documentation faults mean that the system messages or documents are incorrect. Since these faults do not affect program execution, they are excluded from the study.

Chapter 4

Design and Implementation of DEFINE

To conduct fault injection experiments, a fault injection tool is needed. Since the existing tools cannot provide all the desired functions, a new distributed fault injection and monitoring environment (DEFINE) is developed for this study. DEFINE injects faults into the UNIX kernel by software and then monitors the system behavior by software. The source code of the operating system must be available to implement the software monitor. This chapter delineates the structure of DEFINE and then described each component of DEFINE in detail.

4.1 Overall Structure and Methodology

DEFINE consists of six parts: a target system, a controller, a software monitor, a fault injector, a workload generator, and analysis utilities. The block diagram of DEFINE is shown in Figure 4.1. The target system is a group of connected machines executing the same modified UNIX kernel. Each machine can be a server, a client, or both. The controller, fault injector, workload generator, and software monitor run on a separate host machine (other than the target system) that is connected to the target system so that they will not be affected by the faults.

The fault injector can inject both hardware and software faults into the kernel or into any user application on any machine in the target system. Each machine has a local fault injector to receive the fault specification and to handle fault injection via hardware clock interrupts and software traps.

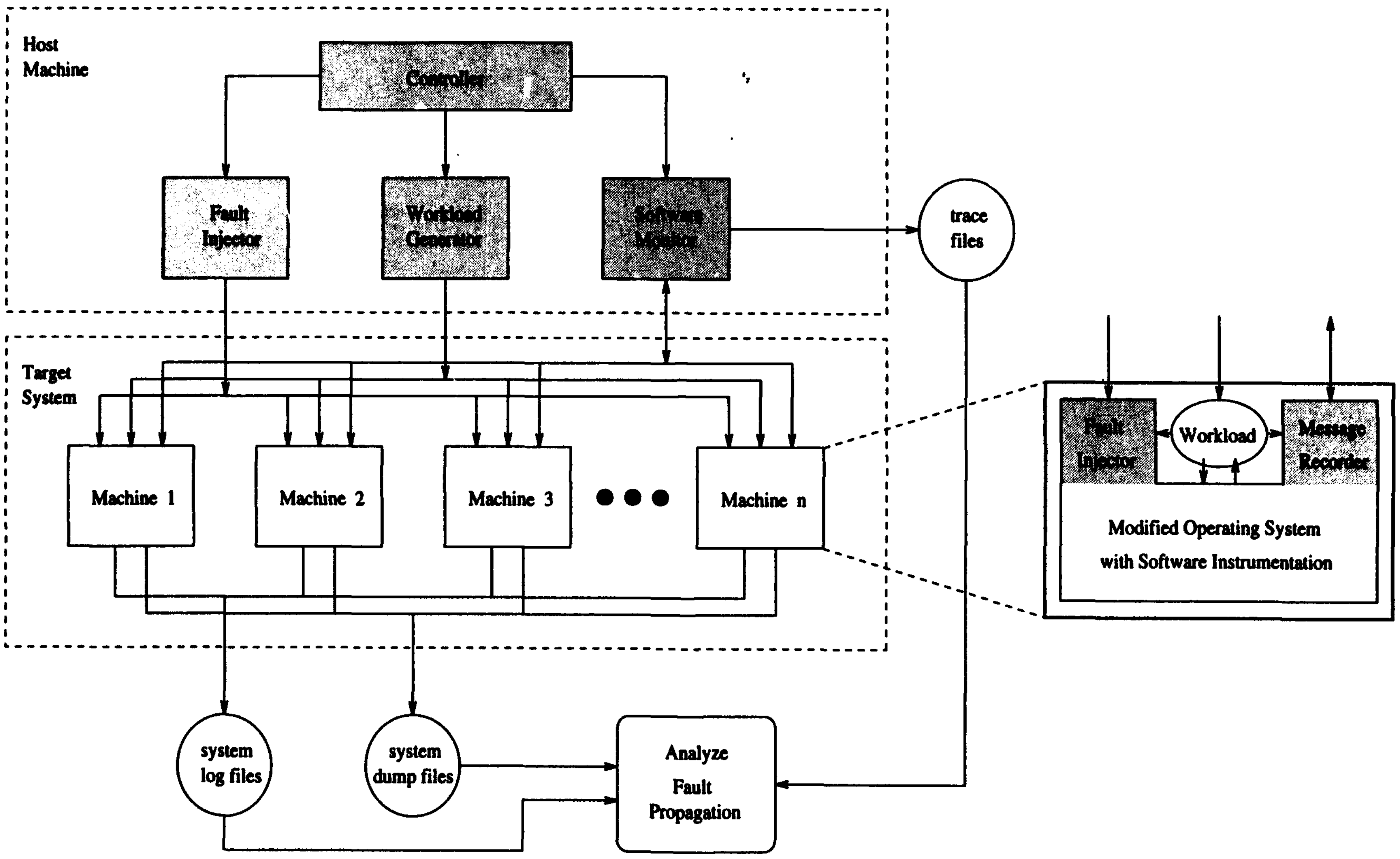

Figure 4.1 Block diagram of DEFINE.

The software monitor is a distributed software monitor to trace the execution flow and key variables of the software systems on every machine of the target system. For each machine in the target system, software instrumentation is inserted into the kernel, and a message recorder is added to collect data and send them to the software monitor. During the experiments, the UNIX kernel also writes system events to the system log file and creates dump files when it crashes.

The workload generator, an extended version of the workload generator reported in [Kao92], is a user-oriented synthetic workload generator that simulates user behavior based on real workload characterizations or any specified distributions. It issues most types of system calls to activate injected faults. It is used to accelerate the activation of injected faults, because some seldom used system calls are invoked more frequently than usual to reduce fault latency.

The controller specifies fault types and locations/machines for the fault injector, the key variables/data for the software monitor, and the workload specification (distributions) for the workload generator. The controller then initiates the experiments. The interface between the target system and the DEFINE components is Remote Procedure Call (RPC).

Several utilities have been developed to extract trace data from trace and dump files and to assist the analysis of fault propagation.

The structure just described is for distributed systems. This structure needs to be modified to conduct experiments on a stand-alone computer. The structure is shown in Figure 4.2. This is similar to the structure of FINE, the predecessor of DEFINE, which is a fault injection and monitoring environment for uni-computer systems [Kao93]. In this structure, the fault injector, the software monitor, the workload generator, and the controller have to be run on the target machine, where they are four processes on the target system. The interface between the kernel and the DEFINE components is a new system call created for this study.

4.2 Fault Injector

The fault injector consists of two parts: the host fault injector, which is running on the host machine to send requests to the target system, and the local fault injector, which is running on each machine in the target system to receive requests from the host fault injector and to inject faults into the UNIX kernel. Since user applications do not have privilege to modify the kernel,

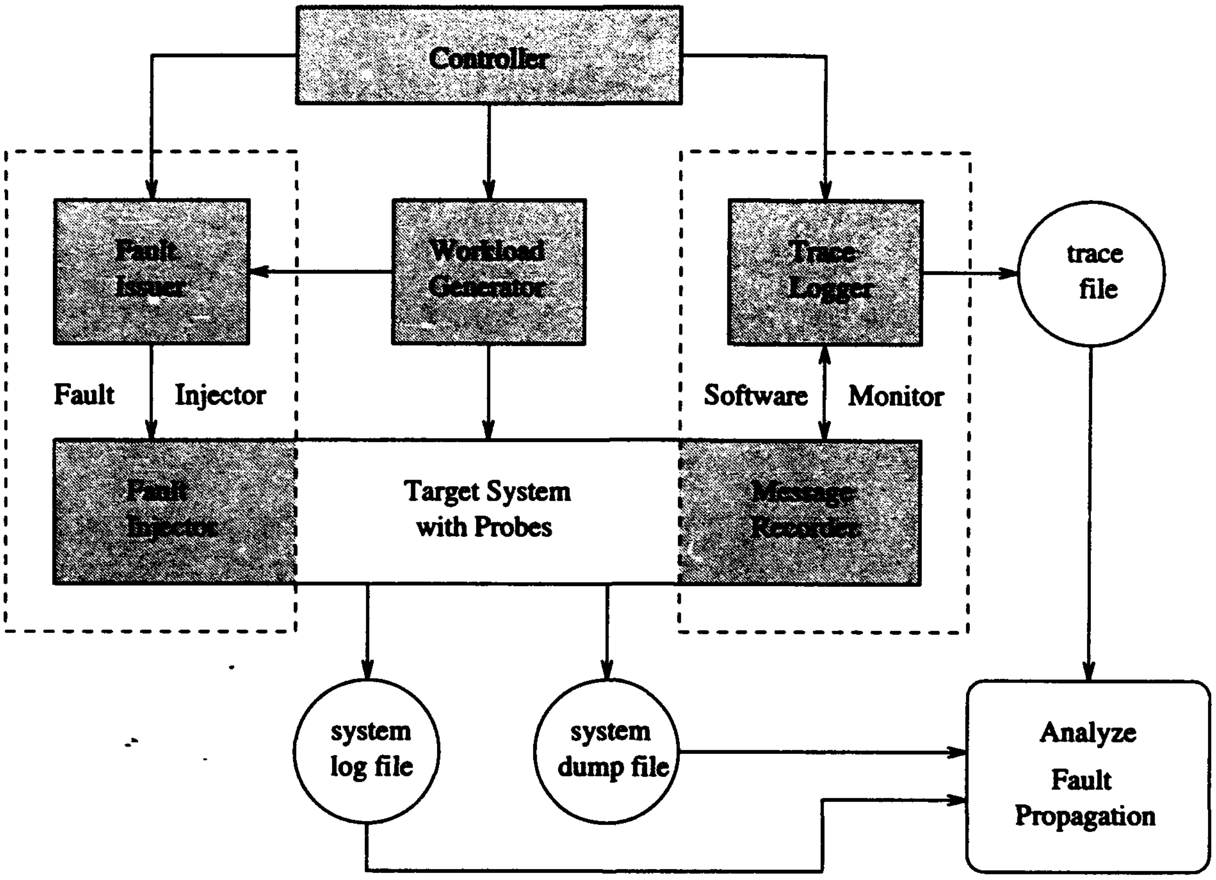

Figure 4.2 Using DEFINE on a stand-alone computer.

the local fault injector has to be implemented in the kernel. The interface between the host fault injector and the local fault injector is RPC and a new system call, `ftrace`. RPC is used in the experiments on distributed systems; `ftrace` is used in the experiments on stand-alone machines or in the experiments where the faults need to be injected by the workload to create a particular scenario. The mechanisms used to inject hardware and software faults will be described in the next two subsections. The detailed fault injection procedure is described in Section 4.2.3.

4.2.1 Hardware Faults

In Section 3.1, we classified the hardware faults into memory, CPU, bus, and communication faults. To inject these different types of faults, the following mechanisms are used:

- A memory fault is injected by changing the contents of a particular location, either in the text segment or the data segment of the kernel. To inject faults in the text segment, the system protection must be turned off temporarily, since the text segment of the kernel is write-protected.
- A CPU fault in a register is injected by issuing a new trap and changing the saved value of the register so that the value of the register is wrongly restored after the trap returns.
- A bus fault is injected by changing the instruction to be executed or the register to be used to read/write data and then restoring it back immediately after the execution. In this way, the incorrect instruction or register emulates the effect of the fault on the bus. A similar approach has been used in FERRARI [Kanawati92], which developed an extra environment like a debugger to inject faults and control experiments. FERRARI is useful for user applications, but not for operating systems, because FERRARI is also a user application. We implement the fault injector in the kernel and use a new trap to inject faults.
- For communication faults, since software-implemented fault injection cannot inject communication faults into the network directly, we mimic the consequence of communication faults by killing or corrupting messages in the User Datagram Protocol (UDP) and Transmission Control Protocol (TCP) routines. The affected messages are selected based on the fault type.

Note that these mechanisms inject transient faults. To inject permanent faults, these mechanisms should be invoked each time the faulty component is used. Table 4.1 lists the injection mechanism and location selection for each hardware fault type.

4.2.2 Software Faults

Software faults have been classified into initialization, assignment, checking, and function faults in Section 3.2. To inject these software faults on-line without recompilation, the addresses of the corresponding instructions and the semantics of the instructions must be known in advance. Therefore, the assembly code (the assembly source code and the assembly code

Table 4.1 Selection and injection of hardware faults.

Fault Type	Injection Mechanism	Location Selection
Memory Fault in Text Segment	Turn off the protection, corrupt an instruction, and turn on the protection	Any location in text segment
Memory Fault in Data/ Stack Segment	Corrupt data	Any location in data/stack segment
CPU Fault in Opcode Decoder	Corrupt the instruction to be fetched and then restore it back	Any instruction
CPU Fault in ALU	Corrupt the destination register or PSR after an arithmetic instruction	Any arithmetic instruction
CPU Fault in Registers	Corrupt a register	Any register and any location
Bus Fault at Address Line when Reading an Instruction	Change the instruction to be fetched and then restore it	Any instruction
Bus Fault at Address Line when Reading Data	Corrupt the load instruction or the indirect register and then restore it	Any load instruction
Bus Fault at Address Line when Writing Data	Corrupt the store instruction or the indirect register and then restore it	Any store instruction
Bus Fault at Data Line when Reading an Instruction	Corrupt the instruction to be fetched and then restore it	Any instruction
Bus Fault at Data Line when Reading Data	Corrupt the destination register after the load instruction	Any load instruction
Bus Fault at Data Line when Writing Data	Corrupt the source register before the store instruction and restore it	Any store instruction
Incoming Message Missing	Kill the incoming message without processing it	Any incoming message
Incoming Message Corrupted	Corrupt the incoming message	Any incoming message
Outgoing Message Missing	Kill the outgoing message without sending it	Any outgoing message
Outgoing Message Corrupted	Corrupt the outgoing message	Any outgoing message

Table 4.2 Selection and injection of software faults.

Fault Type	Injection Mechanism	Location Selection
Uninitialized Data	Change the corresponding instructions to nop	Any initialization
Incorrect Initialized Data	Change the corresponding instructions to the desired instructions	Any initialization
Missing Assignment	Change the corresponding instructions to nop	Any assignment
Incorrect Assignment	Change the corresponding instructions to the desired instructions	Any assignment
Missing Condition Check	Change the corresponding instructions to nop	Any condition check
Incorrect Condition Check	Change the corresponding instructions to the desired instructions	Any condition check

compiled from the C source code) of the kernel has to be used. The instructions are changed according to the following mechanisms to emulate various software faults:

- An initialization fault is injected by changing the instructions that initialize a particular data area such that either the initialized value is incorrect or no initial value is given (only nop's (no operations) are executed).

- An assignment fault is injected by changing the corresponding instructions such that the destination is assigned a wrong value, the evaluated expression is assigned to the wrong destination (and the right destination is not assigned the evaluated value), or the assignment is not executed (only nop's are executed).

- A condition check fault is injected by changing the branch instruction to nop's for a missing condition check or changing the condition check for an incorrect condition check.

- DEFINE can also inject function faults as long as faulty instructions fit into the space of the original instructions. There are no special fault patterns, so users have to specify the faulty instructions for the faults they want to inject.

Table 4.2 lists the injection mechanism and location selection for each software fault type. The software faults involved in the experiments are at the C language level.

4.2.3 Fault Injection Procedure

In Section 2.4, we mentioned three fault injection techniques employed by DEFINE. This section describes the fault injection procedure and implementation issues. To implement these techniques, the system hardware clock interrupt routine has been modified to allow the fault injector to control the fault injection, and two new software traps are created to help fault injection and monitoring. The detailed procedure is shown in Figure 4.3. Basically, the fault injection procedure for all the fault types except communication faults and memory faults in the data/stack segment has these three steps:

(1) Choose the location where the fault is injected. The location may depend on main memory (for memory faults), time (for CPU, bus, and communication faults), or software structure (for software faults). Replace the instruction with the first software trap.

(2) When the trap instruction is executed, send a message to the message recorder to report the time, fault type, and fault location. Then replace the trap instruction with the instruction that emulates the specified fault, replace the next instruction with a second software trap, and return. We cannot replace the next instruction in the first step because the current instruction may be the delay instruction of a branch instruction (for RISC machines) or a jump instruction (for CISC machines), so the next instruction may not be known in the first step.

(3) When the second trap is executed, replace the faulty instruction in step 2 with the first software trap (for memory and software faults) or the original instruction (for the CPU and bus faults). For CPU faults in the ALU, corrupt the destination register or the Processor Status Register (PSR) right now. Then replace the current instruction with the original instruction and return.

Memory faults in the data/stack segment are injected by corrupting data at the specified location. For intermittent memory faults in the data/stack segment, we inject them intermittently based on the interarrival time distribution. For permanent memory faults, the ideal method is to inject them after each time the location is rewritten. Currently, we emulate permanent faults by checking the location periodically with a short interarrival time (10 ms in this implementation). If the location has been rewritten, we inject the fault again.

CPU and bus faults can affect any process, and the affected process may be in user mode or supervisor mode. The fault injector allows users to specify target processes and the target mode so that faults can only be injected into those processes and/or in the target mode. To help biased fault injection, the fault injector also allows users to specify the range of memory locations in which faults will be injected. The default is no restriction. In addition, the fault injector also provides the traditional fault injection method in case users are interested in the effect of faults when they occur during the execution of particular instructions.

Although DEFINE uses hardware clock interrupts to control injection and activation time, when injecting intermittent faults the interarrival times still cannot perfectly follow the specified distribution due to that the hardware interrupts may be disabled temporarily so that the interarrival times will be lengthened a little. The impact of the overhead of fault injection on the interarrival time is removed by subtracting the average time of fault injection from the times given to the hardware timer.

4.3 Software Monitor

The software monitor traces the kernel execution flow and the values of key variables and data. Since the target system of the fault injection experiments is the UNIX operating system, we could not build an extra environment like a debugger to monitor the software behavior; instead, we developed a software instrumentation embedded in the kernel to monitor the system behavior and trace the fault propagation. In addition, core dumps are also analyzed to reveal the reasons for system crashes.

Similar to the fault injector, the software monitor consists of two parts: the host monitor and the local monitor. The local monitor has to be implemented in the kernel. It provides an interface for the host monitor to specify the probes and key variables, to turn the monitor on and off, and to retrieve the trace data. The interface is RPC and the new system call, `ftrace`. The local monitor consists of two parts: probes and a message recorder. The probes are inserted into most of the significant functions to keep track of the execution flow and arguments. The functionality of the probes is similar to the C function `printf` in that they simply report the function name and arguments. The message recorder records time (in microseconds), uid, pid, messages, and the values of the specified key variables into a trace buffer. Figure 4.4 shows five

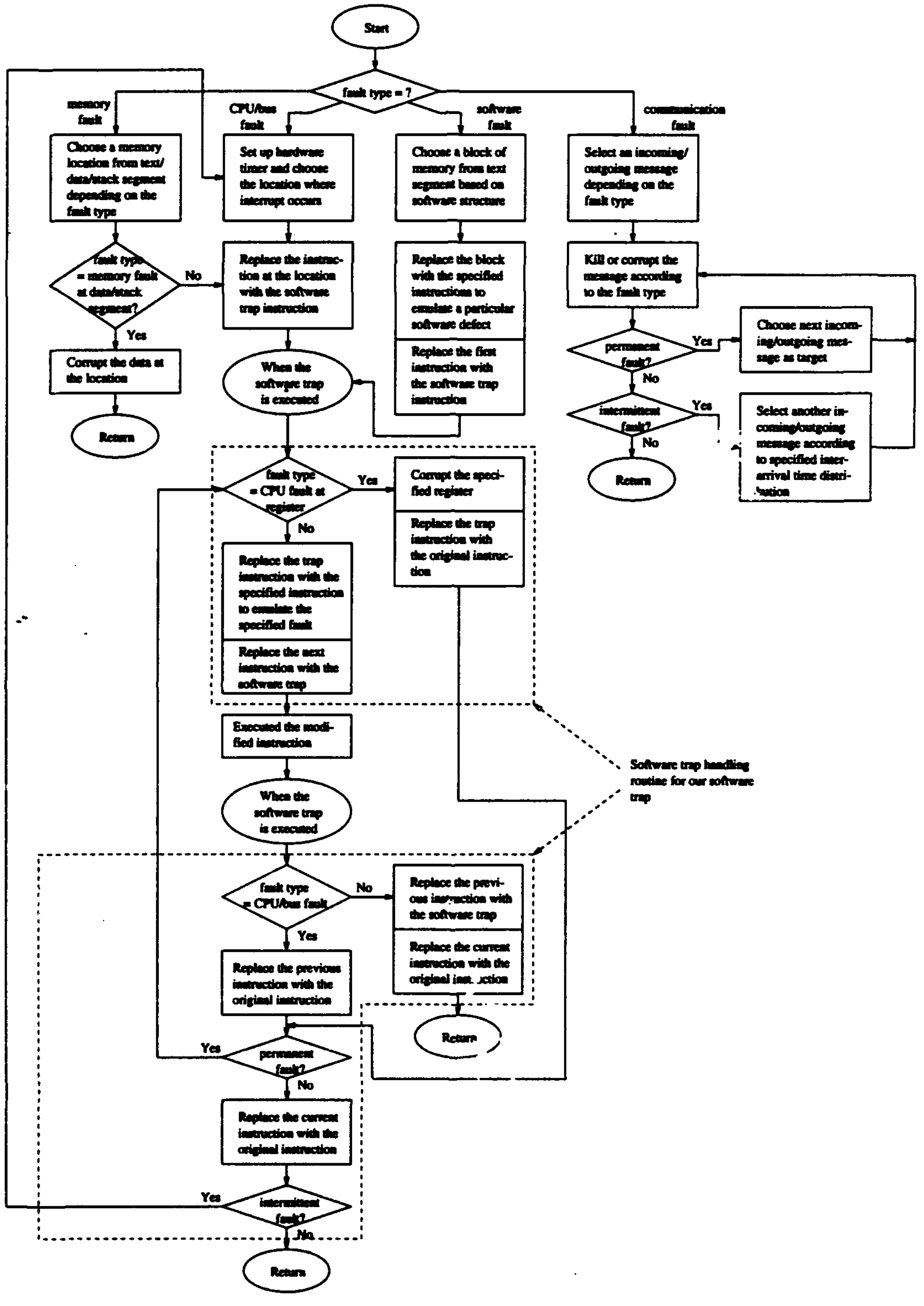
Start
fault type = ?
memory fault
CPU/bus fault
software fault
communication fault
Choose a memory location from text/ data/stack segment depending on the fault type
Set up hardware timer and choose the location where interrupt occurs
Choose a block of memory from text segment based on software structure
Select an incoming/ outgoing message depending on the fault type
fault type = memory fault at data/stack segment?
No
Yes
Replace the instruction at the location with the software trap instruction
Replace the block with the specified instructions to emulate a particular software defect
Replace the first instruction with the software trap instruction
Kill or corrupt the message according to the fault type
Corrupt the data at the location
Return
When the software trap is executed
permanent fault?
Yes
No
Choose next incoming/outgoing message as target
intermittent fault?
Yes
No
Select another incoming/outgoing message according to specified inter-arrival time distribution
Return
fault type = CPU fault at register
Yes
No
Corrupt the specified register
Replace the trap instruction with the original instruction
Replace the trap instruction with the specified instruction to emulate the specified fault
Replace the next instruction with the software trap
Software trap handling routine for our software trap
Executed the modified instruction
When the software trap is executed
fault type = CPU/bus fault
No
Yes
Replace the previous instruction with the software trap
Replace the current instruction with the original instruction
Replace the previous instruction with the original instruction
Return
permanent fault?
Yes
No
Replace the current instruction with the original instruction
intermittent fault?
Yes
No
Return

Figure 4.3 Procedure of fault injection.

```
726737109 800856   200 3039 write 3 f7ff5ca4 4522
       f8113f08: f8223000
       f812c0a8: f81794f4
       f8134470:      246
726737110 249314   201 3039 ftrace stamp: 201, Start of usim
       f8113f08: f822d000
       f812c0a8: f817966c
726737110 251077   201 3039 copen System/File007 1 135100
726737110 251439   201 3039 falloc
726737110 251693   201 3039 ufalloc 0
```

Figure 4.4 Example of trace data.

records of trace data. The fields in the first line of a record are the time in seconds since Jan. 1, 1970, microseconds, pid, uid, function name, and arguments, respectively. The remaining lines of a record are the values of key variables. For example, in the first record, the left-hand numbers are the addresses of the key variables `uunix`, `masterproc`, and `nfile` which are the, respectively, pointer pointing to the current user structure, the pointer pointing to the current process structure, and the maximum number of files can be opened. The right-hand numbers are the values of the key variables. To reduce the trace data, the value of a key variable is recorded only when it has been modified. The interface is also the new system call, `ftrace`. The host monitor is a trace logger that specifies probes and key variables, turns the monitor on and off, retrieves trace data from the trace buffer through `ftrace`, and writes the trace data to a trace file.

Since it is impractical to write down all the data at each check point (probe) and since the key variables are dependent on the injected fault, the key variables have to be specified by their addresses at the beginning of each experiment.

To trace fault propagation, the semantics of the fault and the contents of the fault location should be known when a fault is injected. The symbol table of the UNIX kernel is used to identify the faulty function/data and to specify the key variables,

In addition to tracing the kernel behavior, the software monitor can also trace user applications. The new system call, `ftrace`, can be inserted into user applications to report interesting information, such as control flow and key data.

The overhead of the software monitor may be up to 50% of system time, but the user time is not affected. If the purpose of experiments is not on fault propagation, the software monitor can be turned off to eliminate this overhead.

Other features of the software monitor are listed below.

- The interface for the software monitor is designed in such a way that users can design their experiments flexibly. The features include reallocating the trace buffer, specifying the key variables, turning the software monitor on and off, resetting the monitor, retrieving the trace data from the buffer, marking an event to the trace data, checking the protection of an area, peeking at the contents of an area, and modifying the contents of an area (while keeping the same protection).
- The buffer for trace data is dynamically allocated so that the buffer size can be changed on-line to fit different experiments.
- The probes of the software monitor are divided into several groups and can be turned on and off by group to control the trace data.
- If time is not important for the experiment, time stamps can be turned off to reduce trace data.
- The fault injector also marks fault injection events and faults in the trace buffer to make it easier to analyze fault propagation.

4.4 Workload Generator

The workload generator is an extended version of the workload generator reported in [Kao92]. It is a user-oriented synthetic workload generator that simulates user behavior based on real workload characterization or any specified distributions. The workload generator generates system calls because system calls are the interface for users to utilize system resources. The purposes of including the workload generator in the environment are:

- to accelerate the activation of injected faults and hence to be able to determine their effects,

- to generate two identical copies of worklcads (by giving the same seed to the random number generator) for comparison between fault-free trace data and faulty trace data, and
- to avoid interfering with normal work because DEFINE cannot be run while other users are performing useful work.

Commonly used applications are not good enough to activate injected faults, because not all of the system calls are exercised by these applications. Since latency is not the key point of the study, the interarrival times between system calls are set to zero to accelerate experiments. In addition, some seldom-used system calls are invoked more frequently than usual to reduce latency time. The idea of failure acceleration is proposed by Chillarege and Bowen [Chillarege89]. To obtain a realistic latency time, the distributions of system calls and interarrival times should be determined based on the real workload characterization.

4.5 Controller

The controller is a script file used to specify the fault to be injected for the fault injector, the key variables to be traced for the software monitor, and the workload to be generated for the workload generator. When the specifications are completed, it then initiates the experiment.

Since the target system may crash during experiments and may need to be handled by operators, it is impossible to automate a series of experiments unless the target system is a fault-tolerant system.

4.6 Analysis Utilities

Several utilities have been developed to help analyze fault propagation. When the target system crashes and creates dump files, an ADB debugger script is used to extract the remaining portion of trace data, the values of key variables, and the information from the stack. A utility is provided so that users can compare correct trace data with faulty trace data to identify fault propagation. However, human interpretation is needed to identify fault propagation and impact.

Chapter 5

Experiments on the UNIX Kernel

Experiments are conducted on the SunOS 4.1.2 system to evaluate the impact and propagation of various types of faults. The experiments are described in Section 5.1, and the results are shown in Section 5.2 and 5.4. The locations of very-long latency faults and the possible impact of these faults are discussed in Section 5.3

5.1 Experiment Description

The experiments are designed to evaluate the impact and propagation of various faults. One fault is injected for each experiment, and its propagation and impact are analyzed individually. Five hundred experiments are conducted: one half of them inject hardware faults, and the others inject software faults. Sections 5.1.1 through 5.1.3 depict the experiment environment, the experiment control flow, and the selection of faults and key variables.

5.1.1 Experiment Environment

The experiment environment is a Sun SPARCstation IPC connected to a local area network system in a university laboratory. The operating system is the SunOS 4.1.2, which is derived from the BSD 4.3 UNIX. The exact sizes of the kernel text and data segments depend on the system configuration. The target version of the SunOS 4.1.2, excluding the DEFINE components, the fault injector and the software monitor, has about 1.07 megabytes of text segment and 1,426 global variables (about 56 kilobytes of bss (uninitialized data) and 158 kilobytes of data that include system messages). The SPARC RISC CPU has 32 working registers (current window within a set of 136 registers) and 4 special registers. Floating point and coprocessor registers are optional.

5.1.2 Experiment Control Flow

The overall experiment consists of a series of similar individual experiments. The control flow of each experiment is shown in Figure 5.1. The software monitor starts first, and then the workload generator is run once to generate a fault-free trace data. Next, the workload generator is run again to duplicate the same workload. At this time, a fault is injected into the kernel to generate a faulty trace data. Hardware faults, except for process-related faults, are injected by the fault injector. Process-related faults are injected by the workload generator so that the workload generator will be the process that is influenced. Software faults are injected by the workload generator just before the workload is repeated. If the machine does not crash, we stop the software monitor and reboot the machine for the next experiment.

If the target system crashes before an experiment is finished, the remaining portion of the trace data is extracted from the system dump file and appended to the trace file. The fault-free trace data and faulty trace data are extracted from the trace file and compared to identify the fault propagation. If a fault does not affect the execution of the kernel, the assembly code has to be traced to understand how the fault is recovered or avoided. If the target system crashes before the next probe is reached (i.e., no information is recorded), the system dump file is analyzed and the assembly code is also traced to investigate how the fault affects the system. The system log file, the system dump file (if created), and the result of fault propagation analysis are then analyzed to identify the fault impact on the target system.

5.1.3 Faults and Key Variables

In each experiment, one fault is injected into the kernel in the following manners:

- Memory faults at the text segment are generated by selecting an address randomly from the kernel text segment and flipping a bit in the address at random.
- Memory faults at the data segment cannot be generated like memory faults at the text segment because system messages are also in the data segment and injecting faults into

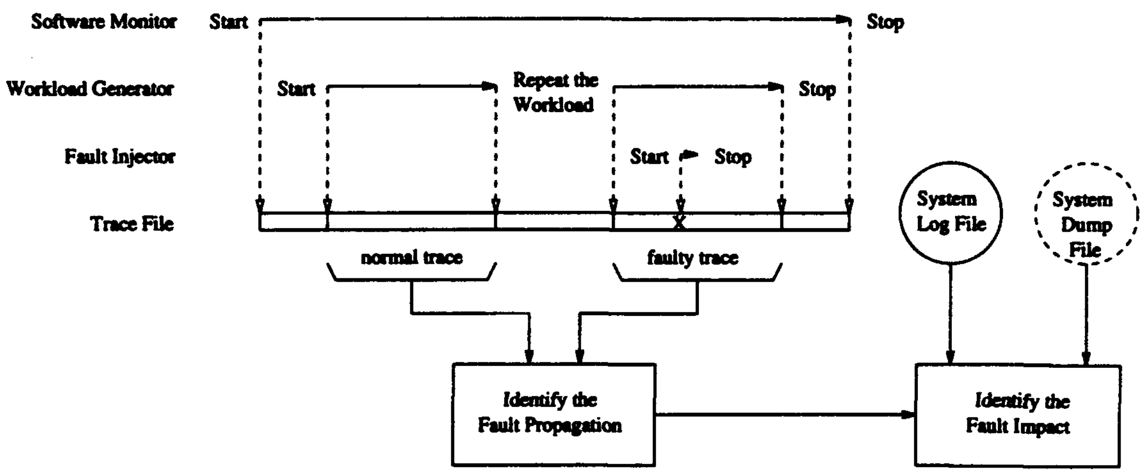

Figure 5.1 Experiment control flow.

system messages (documentation faults) is not desired. Therefore, a global variable in the kernel is randomly selected and a bit is randomly chosen to be flipped.

- Bus and CPU faults are generated by selecting and changing an instruction that will be executed in the near future. The code that will not be executed under a normal workload is not considered.

- Three types of software faults are generated by a similar procedure. First, an address in the kernel text segment is selected randomly. Then, the nearest initialization, assignment or condition check from the address is located and the corresponding instructions are modified according to the fault injection mechanisms in Section 4.2. This type of fault cannot be generated automatically because it is not trivial to identify the instructions.

Since DEFINE traces the fault propagation in the UNIX kernel, faults are not injected in the fault injector and the software monitor in the experiments because the fault injector and the software monitor are not parts of the original UNIX kernel. In addition, it allows proper injection and monitoring.

The hardware faults injected in the experiments include only those that occur in the kernel, but we can estimate the probability of faults occurring in the kernel. For the memory faults, we can calculate the probability by dividing the kernel size by the total memory size. For the bus and CPU faults, we can estimate the probability by dividing the system time by the total time. All these probabilities are machine and application dependent.

The key variables to be traced are strongly related to the injected fault and are difficult to specify thoroughly in advance. Sometimes, an experiment has to be repeated with more key variables to identify the fault propagation.

5.2 Fault Impact on the System

A fault injected into the kernel can cause three possible effects on the system:

(1) *System failure*: The fault causes one or more errors, and one of the errors is detected by hardware or software detection/protection mechanisms. The error is considered so serious that the system should not continue its work. The system may reboot itself or need to be checked by operators.

- *No self-reboot*: Hardware detects the error and returns control to the PROM (monitor). This kind of error is considered very serious since the system will not reboot itself automatically and needs operators to handle the situation. Operators usually check the hardware first, and then reboot the system.
- *Self-reboot*: Hardware or software detects the error and returns control to the system's exception handling routine to report the error and reboot the system. This kind of error is serious because the system is unavailable temporarily and all the jobs in the system have been killed.
- *System hung*: The fault causes the system to enter an endless loop or to wait for an event that will never happen. This kind of error is serious because the system is virtually unavailable and needs operators to handle the situation (usually just to reboot the system). Although this kind of error is not detected by the system, it is still included in this group since no service can be provided under this situation.

(2) *User application failure*: The fault causes one or more errors, but the system fails to detect these errors. The system survives, but user applications are affected. There are two possible impacts:

Table 5.1 Distribution of hardware fault impact.

Fault Type	System failure			Multiple User Application Failure	No Error	
	Without Self-Reboot	With Self-Reboot	System Hang		Fault Avoided	Very Long Latency
Memory Fault in Text Segment	2%*	22%	2%*	0%	6%	68%
Memory Fault in Data Segment	2%*	14%	0%	8%	18%	58%
Bus Fault on Address Line	0%	82%	0%	10%	6%	2%*
Bus Fault on Data Line	0%	76%	0%	10%	14%	0%
CPU Fault in Registers	0%	66%	0%	0%	34%	0%

*Not statistically significant.

- *Multiple program failure*: More than one user application crashes or produces incorrect results. The error still remains in the system, and may affect subsequent user applications. Usually, the system needs to be rebooted, since it cannot provide full service.
- *Single program failure*: The fault affects only one user application, and the fault is recovered after the application has crashed or finished.

(3) *No error*: The fault does not cause any damage. There are two possible reasons:

- *Fault avoided*: The fault has no effect on the execution or is recovered before it causes any propagation or damage.
- *Very long latency*: The fault will not be activated under normal execution condition. The impact of the fault is unknown. To evaluate the impact of such faults, other specially designed workloads need to be used to activate the faults.

The following sections discuss the impact of hardware and software faults, and the system error detection mechanisms.

5.2.1 Impact of Hardware Faults

The impact of hardware faults on the system is shown in Table 5.1. Rows represent injected fault types and columns represent impacts on the system. The numbers are the percentages of different impacts on the system for each fault type. There are 50 experiments for each row. Since the single-user application failure is not observed, this kind of impact is not shown in the table. Very long latency includes very long fault latency and very long error latency. Fault latency is the time between the fault occurrence and the activation, and error latency is the time from the error occurrence (a fault is activated and causes an error) to the detection. If a fault is not activated by the workload generator, which runs for about 20 minutes each time, this fault is assumed to have a very long fault latency. If an error does not propagate and is not detected during the experiment, this error is assumed to have a very long error latency. Several lessons can be learned from the experiments:

- A significant part of the UNIX kernel is not exercised although the accelerated workload is used, so most of the memory faults at the text segment (68%) have a very long fault latency. Detailed investigation of the locations and possible impact of these non-activated faults are discussed in Section 5.3. For those memory faults with very long fault latency, we can use a memory scrubber, a system routine for early error detection, to recover them before they are activated [Saleh90]. For those memory faults in the data segment, we can apply robust data structures to recover them [Taylor80, Kant90].

- Two system failures without self-reboot are observed, both of which cause watchdog reset. The incurred errors are so serious that dump files cannot be created by the monitor command **sync**, because **sync** calls the kernel's function **panic**. One of them even hangs the system when **sync**ing the file systems (trying to dump memory to dump area) and cannot be interrupted by the hardware interrupt, making power-off the only way to solve the problem.

- Only one system hang is observed. In fact, the fault causes system crash with self-reboot first, and then the system gets stuck when **sync**ing file systems. The reason is as follows: the fault is injected into the file system, a hardware trap is invoked when the workload generator issues a file operation, and then the system issues another file operation to dump memory that causes another hardware trap that has been disabled. The system is thus hung. No system dump file can be created in this situation. Although this case and the two system failures without self-reboot are not *statistically significant*, we still mention them because they are unexpected and have serious impact. In addition, no dump file can be created, which makes debugging almost impossible when these faults occur in the field.

- Bus and CPU faults have a very short error latency. Therefore, multiple instruction retry is a feasible mechanism to recover these two types of transient faults [Li91, Alewine92].
- Some global pointer variables like `uunix` are very sensitive. If their values are corrupted, the system is crashed immediately and is unable to reboot itself.
- When an experiment is repeated several times, the results are not always the same. This is because the timing of the fault injection is not fixed unless the fault is injected by the workload generator.
- Propagation through global variables may not be caught in the experiments. This is because there is no cross-reference for global variables, pointers are widely used in the kernel, and some global variables have the same name (a very confusing implementation style).
- In SPARC instruction set, some instructions do not use all the bits. Thus, if a memory fault in the text segment or a bus fault on the data lines when reading a instruction affect these "don't-care" bits, the fault is avoided.
- If there is no memory scrubber in the system, memory faults in the kernel text segment cannot be recovered. This is because the kernel text will not be swapped out, and therefore, no correct copy will be swapped in to overwrite the faulty pages. However, user applications can recover from memory faults at the text segment if the correct copy of the faulty pages are already stored in the swap area (the faulty pages will be discarded when they are swapped out, and the correct ones will be swapped in later).

5.2.2 Impact of Software Faults

The impact of software faults on the system is shown in Table 5.2. The rows, columns, and numbers represent the same things as in Table 5.1. There are 50 experiments for each row. Since no system failure without self-reboot nor single-user application failure is observed, these

Table 5.2 Distribution of software fault impact.

Fault Type	System failure: With Self-Reboot	System failure: System Hang	Multiple User Application Failure	No Error: Fault Avoided	No Error: Very Long Latency
Uninitialized Pointer	46%	0%	0%	0%	54%
Misassigned Pointer	40%	0%	0%	0%	60%
Missing Condition Check	22%	0%	2%*	20%	56%
Incorrect Condition Check	26%	0%	0%	12%	62%
Uninitialized/ Misassigned Non-Pointer Data	26%	2%*	6%	6%	60%

two kinds of impacts are not shown in the table. From the experiments, several phenomena can be observed:

- Similar to memory faults at the text segment, many of the injected software faults (54% to 62%) have a very long latency.
- Except for those with a very long latency, pointer faults cause system failures because the system tries to access a memory location where it has no privilege to access.
- One system hang is observed. In this case, the fault is injected into a signal-processing function, and the fault simulates missing initialization of a variable (signal number). The fault thus causes the system to wait for an event that will never happen.

5.2.3 System Error Detection Mechanisms

When the injected faults cause errors, the system can detect the errors by hardware (`traps`) or software (`panics`) mechanisms. Table 5.3 shows the distribution of errors detected by these detection mechanisms.

The hardware detection and protection mechanisms are the first barrier to detect errors, and they are quite effective. Nearly 90% of detected errors are detected by hardware. If an error passes the first barrier, software **`assertions`** and **`panics`** provide the second barrier to

Table 5.3 Distribution of errors detected by system error detection.

Hardware Traps				Software Panics
Text Fault	Data Fault	Memory Address Alignment	Illegal Instruction	
16%	47%	19%	6%	12%

Table 5.4 Distribution of locations of very long latency faults.

Error Handling or Other Case Handling	Device Driver	Initial Routine
40%	47%	13%

stop the error. However, about 5% of errors are not detected by the system and cause the system to hang or affect user applications. About half of the detected errors are data faults (47%), and they are detected when the system tries to read/write data from/to the area where it has no privilege to do so.

5.3 Very Long Latency Faults

Although the workload generator is designed to activated injected faults, about 60% of memory and software faults are still not activated. Thus, the locations of these very long latency faults in the source code are studied to find out why so many faults are not activated. Table 5.4 lists the distribution of locations of very long latency faults. About 47% of very long latency faults are injected into device drivers where the corresponding devices are not installed in the target system. About 40% are injected into error handling routines or those handling routines that are seldom exercised. To activate these faults, two faults need to be injected, or special inputs need to be designed. About 13% are injected into initial routines, which will not be executed again once the machine is booted. Other fault injection approaches need to be applied to activate these faults.

Assuming these very long latency faults can be activated, their possible impact is listed in Table 5.5. The impact of most of the faults (66.7%) cannot be determined before really executing the code, because the impact depends on the value of registers or memory when the

Table 5.5 Possible impact of very long latency faults.

Text Fault	Data Fault	Illegal Instruction	Fault Avoided	Non-deterministic
6.7%	6.7%	6.7%	13.3%	66.7%

faults are activated. About 13.3% of these faults will always be avoided, while 20% will always crash the system. Among these 20% of faults, one third of them (6.7%) will cause the system to execute illegal instructions. This kind of impact can be easily determined, and none of the non-deterministic faults will cause this impact. The percentage (6.7%) of this impact is very similar to that (6%) in Table 5.4.

5.4 Fault Propagation

The previous section showed the impact of various faults on the system. What really happens after a fault is injected is described in this section. After a fault propagates to another part, the propagated part is like a new fault and may propagate to other parts. Such chain effects will continue until the system crashes or is rebooted (damages may have been caused). The following sections discuss the fault propagation of hardware and software faults, and the fault propagation among the UNIX subsystems.

5.4.1 Hardware Fault Propagation

The first level of fault propagation of hardware faults is shown in Table 5.6. Rows represent injected fault types, and columns represent the first level of fault propagation. The numbers are percentages. The subsequent fault propagation (error propagation) of hardware faults is shown in Table 5.7. Rows represent current fault propagation states, and columns represent the next fault propagation states. The numbers are percentages. Several phenomena are observed from the experiments:

- A same kind of error can be caused by several different types of faults. For example, the execution of an incorrect instruction can be caused by a memory fault at the text segment, a bus fault at address lines, or a bus fault at data lines.

- The SPARC CPU is a RISC processor, and load/store instructions read/write memory indirectly (through registers). When registers are corrupted or the system reads incorrect instructions or data, there is a very high probability that the system will crash in the next few instructions.

- SPARC uses PC and nPC to implement delayed control transfers, wherein the instruction that immediately follows a control transfer may be executed before control is transferred to the target address. A CPU fault in PC register caused incorrect instruction executed, and a CPU fault in nPC causes incorrect control flow.

- nPC register faults and incorrect jumps/calls tend to distract the debugging direction, because the system `traceback` routine gives an incorrect procedure call sequence.

- Some memory faults at the data segment change the procedure call table, and their effects are the same as memory faults at the `call` instructions and CPU faults in the nPC register.

- When a bus fault on the address lines occurs during reading an instruction or data, it is possible that the contents of the wrong address happen to be the same as those in the correct address, so the fault is avoided. However, this scenario did not occur in the experiments.

5.4.2 Software Fault Propagation

The first level of fault propagation of software faults is shown in Table 5.8. The subsequent fault propagation (error propagation) of software faults is shown in Table 5.9. The rows, columns, and numbers represent the same things as in Tables 5.6 and 5.7. The results reveal several phenomena:

- The percentage of software faults with very long latency is slightly lower than that of memory faults at the text segment (Table 5.6), although they should have similar percentages. This is because the locations of the software faults are shifted to the nearest initialization, assignment, or checking instructions (determined by our fault injection methodology), and the shifting increases the probability that the faults are activated.

- Faults involving pointers and faults involving non-pointer variables/parameters have quite different properties, thus faults are divided into pointer related and non-pointer related faults.

- Pointer faults tend to crash the system immediately if they are activated. If the faults are not detected immediately, the propagation is unpredictable because they may corrupt other data that are not related to the faults. Therefore, error detection/recovery mechanisms should pay closer attention to pointers.

- In the software fault propagation, incorrect control flow is the major effect for the first level of propagation, and data corruption is the major effect for the subsequent propagation. The system crashes are usually caused by the corrupted data, and the symptom is data fault shown in Table 5.3. Robust data structures can be used to recover the data corruption caused by pointer faults [Taylor80, Kant90].

5.4.3 Fault Propagation among the UNIX Subsystems

The previous two subsections investigated the fault propagation of various faults but did not consider whether the affected parts are in the same subsystem where the faults occur. This section analyzes the fault propagation among the UNIX subsystems, that is, the locations of faults and propagated errors.

Basically, the UNIX operating system can be divided into process management, memory management, file management, and exception handling subsystems. Experiment results show that only about 8% of faults propagate to other subsystems, and that most of the propagation is caused by function calls with incorrect arguments. The reasons for the low percentage are that hardware errors tend to crash the system (detected by the system) in a few instructions and do not have chance to propagate faults, that the corrupted data are usually accessed by the same subsystem, and that the incorrect control flow caused by software faults usually affects the same subsystem.

Table 5.6 Distribution of the first level of fault propagation of hardware faults.

Fault Type	Incorrect Instruction Executed	Further Data Corruption	Incorrect Control Flow	System Failure with Self-Reboot	Fault Avoided	Very Long Latency
Memory Fault in Text Segment	26%	0%	0%	0%	6%	68%
Memory Fault in Data Segment	0%	10%	14%	0%	18%	58%
Bus Fault on Address Line	60%	32%	8%	0%	0%	0%
Bus Fault on Data Line	48%	34%	6%	0%	12%	0%
CPU Fault in Registers	4%	24%	8%	30%	34%	0%

Table 5.7 Distribution of the subsequent fault propagation of hardware faults.

Fault Type	Further Data Corruption	Incorrect Control Flow	System Failure without Self-Reboot	System Failure with Self-Reboot	System Hang	User Application Failure	Fault Avoided	Very Long Latency
Incorrect Instruction Executed	29%	35%	0%	36%	0%	0%	0%	0%
Further Data Corruption	8%	25%	0%	56%	0%	4%	5%	2%*
Incorrect Control Flow	18%	9%	3%*	58%	3%*	9%	0%	0%

Table 5.8 Distribution of the first level of fault propagation of software faults.

Fault Type	Further Data Corruption	Incorrect Control Flow	System Failure with Self-Reboot	Fault Avoided	Very Long Latency
Uninitialized Pointer	0%	4%	42%	0%	54%
Misassigned Pointer	20%	4%	20%	0%	56%
Missing Condition Check	0%	24%	0%	20%	56%
Incorrect Condition Check	0%	26%	0%	12%	62%
Uninitialized/ Misassigned Non-Pointer Data	24%	10%	0%	6%	60%

Table 5.9 Distribution of the subsequent fault propagation of software faults.

Fault Type	Further Data Corruption	Incorrect Control Flow	System Failure with Self-Reboot	System Hang	User Application Failure	Fault Avoided	Very Long Latency
Further Data Corruption	43%	24%	22%	1%*	3%*	5%	2%*
Incorrect Control Flow	75%	5%	11%	0%	6%	3%*	0%

Chapter 6

Experiments on the Sun Network File System

In Chapter 5, the fault impact and propagation in the UNIX kernel have been investigated, but the fault impact and propagation are restricted to the faulty machine. The faults may cause no error, user application failure, or system failure. However, the probability of fault propagation through the network is so low that no fault propagation among machines is observed during the experiments.

In this chapter, the strategy of fault location selection is changed in order to increase the probability of fault propagation through the network. The strategy is biased fault injection; that is, faults tend to be injected into certain parts of the kernel. The target software system is the Sun Network File System (NFS). The experiments are conducted on six Sun workstations (one as server and the others as clients) to study fault characteristics and propagation in a distributed system. The fault scenario we are simulating is that one machine is faulty due to one hardware or software fault, and the fault may affect the processes on the faulty machine, crash the faulty machine, or propagate to other machines.

The remainder of this chapter is organized as follows. Section 6.1 describes uniform fault injection and biased fault injection. A brief introduction of the Sun NFS is presented in Section 6.2. The fault propagation model of the NFS is described in Section 6.2.1. Section 6.3 explains the experiment design. In Section 6.4, the the possible impact of an injected fault on a distributed system is described, and the system behavior under biased fault injection is presented and discussed. The impact of faulty user applications is described in Section 6.5. Section 6.6 presents the path-based fault injection experiments, where faults are injected along popular control paths to increase the activation rate of software faults to 100% and to create a particular scenario.

6.1 Uniform Fault Injection and Biased Fault Injection

In the experiments described in Chapter 5, faults are injected into the UNIX kernel uniformly. This kind of fault injection is called uniform fault injection; that is, faults are injected into the UNIX kernel without any preference. On the other hand, if faults tend to be injected into certain parts of the kernel, the fault injection is called biased fault injection. The purpose of biased fault injection is to increase the probability of certain kind of fault impact. In this Chapter, the purpose is to increase the frequency of fault propagation through the network.

Since most network traffic in a distributed system is related to remote file operations, fault propagation through the network may be observed more frequently if more faults are injected into the file system, especially the distributed file system. The underlying philosophy is to increase the probability of fault propagation, which is similar to the failure acceleration proposed in [Chillarege89]. In this study, the distributed file system used is the Sun Network File System (NFS). Therefore, the target software in this Chapter is the Sun NFS.

6.2 NFS Overview

The Network File System (NFS) is a client-server mechanism whereby several computers can share file systems [Gould86]. The NFS structure is shown in Figure 6.1.

The arrows mean requests and responses, and most of them are function calls and return values (some values are passed through global variables). User applications can issue system calls (e.g., `open`, `read`, and `write`) directly or indirectly through libraries, such as the C library. The system call interface is usually implemented by software traps.

The NFS is embedded in the kernel just like the UNIX file system. NFS is an extension of the UNIX file system and provides a transparent view for local file systems and remote file systems. It adds one more layer, vnode manipulation, above the UNIX inode manipulation.

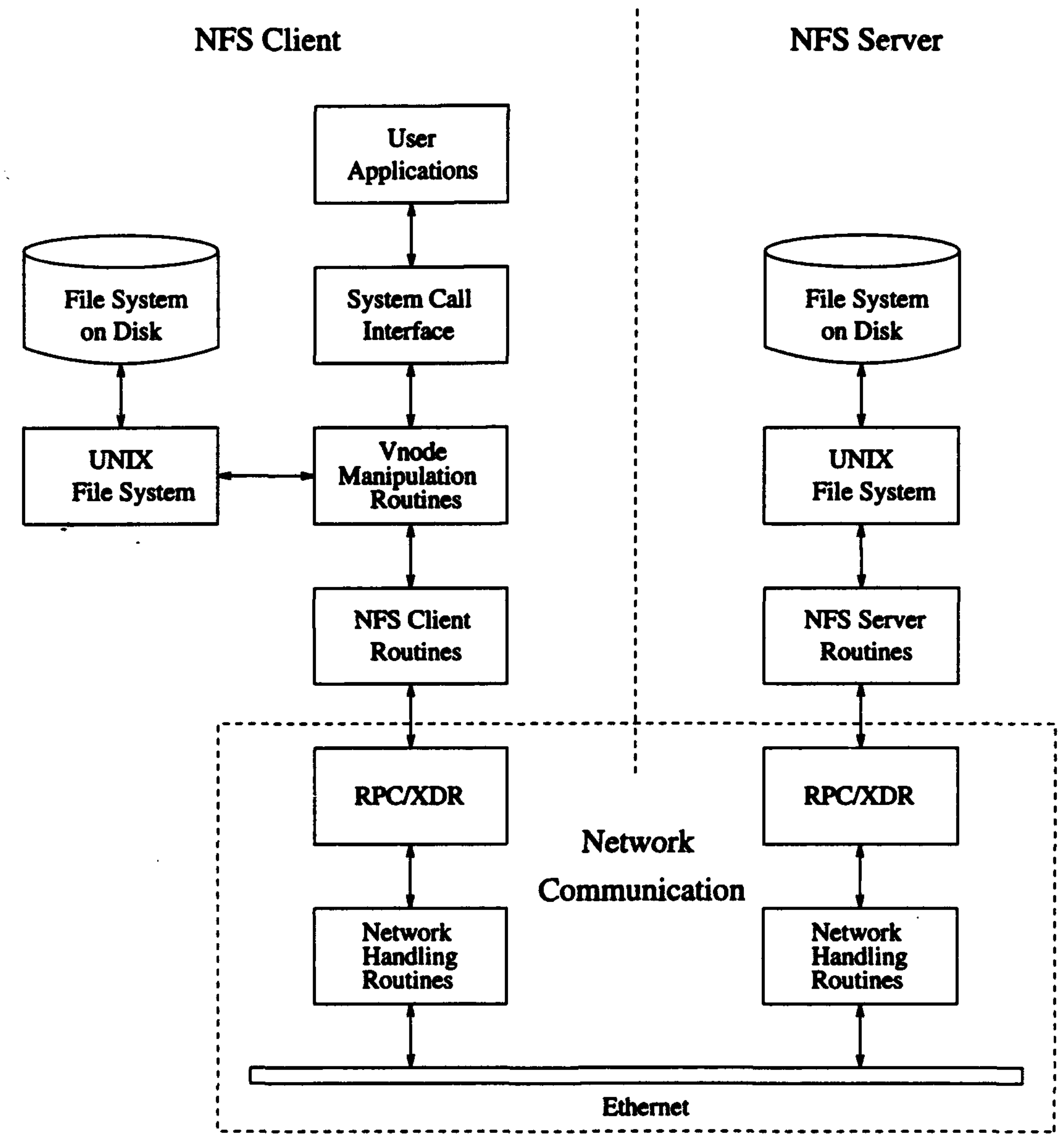

Figure 6.1 Structure diagram of NFS.

In addition to vnode manipulation routines, NFS has server routines and client routines which are initialized by `nfsd` and `biod` daemons respectively. The server and client routines can be included into the kernel configuration when the kernel is built, and thus the machine can be a server, client, or both by issuing `nfsd`, `biod`, or both. If the kernel of a machine does not have server routines, this machine cannot be a NFS server; similarly the absence of client routines presents a machine from being a client.

6.2.1 Fault Propagation Model for the NFS

To delineate the software characteristics, we develop software-structure-level fault propagation models. Because software-structure-level fault propagation models include software system structure, they are built for specific software systems and are different for different software systems. The software-structure-level fault propagation model for the NFS is shown in Figure 6.2. For simplicity, the figure just demonstrates one diskless NFS client and one NFS sever connected by network. Generally, there are several machines connected by network, and each machine may have a local disk and can be a client or server or both.

The left column in the figure shows that faults can occur at any part of the computer network system. The middle column illustrates the fault/error propagation in the NFS. The right column depicts the fault impact on the network system.

Arrows in the middle column mean fault/error propagation. The only way that a fault can propagate from user applications to the kernel is through system calls. Propagation in a client or a server can be made possible through global variables or arguments/return values of function calls. Propagation between a client and a server has to go through network communication. Propagation between two clients must go though network communication and the shared server(s).

In the right column, *fault avoided* means the fault has no effect on the execution or is recovered before it causes any propagation or damage. After an error is produced, it may be avoided, recovered, or cause further errors until the error(s) is detected or its impact is observed. The six states of fault impact on the network system are not exclusive. For instance, when a client crashes, all the user applications running on that client also crash. When a server crashes, all the user applications on clients that access files on the server will be hung until the server is up. After the server is up, the hung user applications will continue their execution, and may or

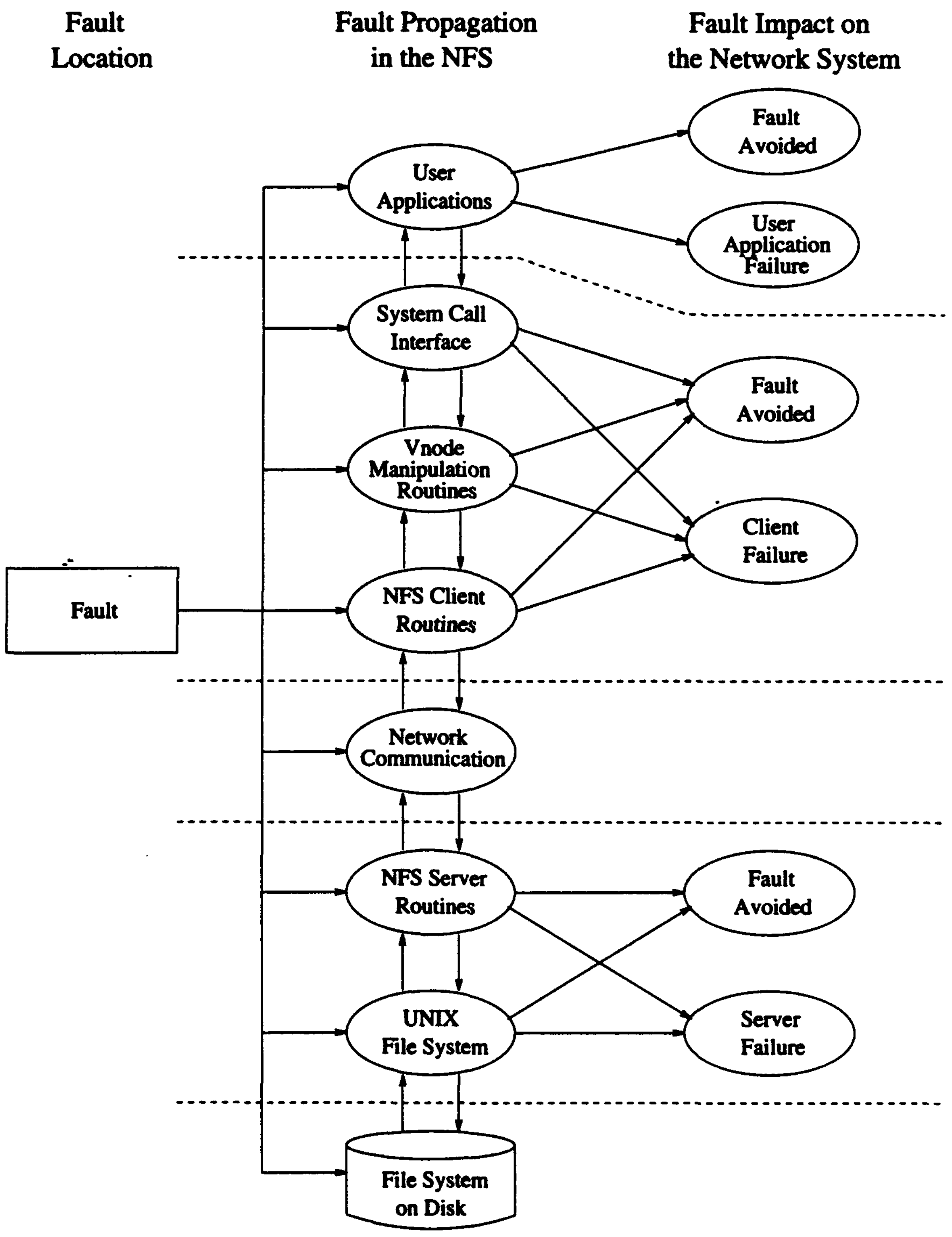

Figure 6.2 Fault propagation model for the NFS.

may not crash depending on the following situation. If an application is hung during reading, it can continue the work without crash; if an application is hung during writing, the correctness of the application depends on whether the transaction has been recovered from the crash.

6.3 Experiment Description

In each experiment, one machine is selected, either a server or a client, and a fault is injected into that machine's kernel. Although faults may occur in both the kernel and user applications, we are interested in the faults in the kernel because the kernel has higher privilege and the impact is more serious.

For communication faults, the impact of corrupted messages has been described in [Devarakonda90], and the impact of missing messages has been shown in [Rosenberg93]; therefore, we do not show them again by conducting another experiment. Instead, we show the impact of one faulty machine and the natural process of fault propagation.

The experiments are conducted on six Sun workstations (one as server and the others as clients) to study fault characteristics and propagation in a distributed system.

6.3.1 Experiment Procedure

To identify fault propagation, we need to execute the workload twice—once without any fault and once with one injected fault—and then compare the trace data collected from the two runs. The overall experiment consists of a series of similar individual experime⁻ᵗs. The following procedure is used in each experiment:

(1) Set up the experiment, and make sure that all the machines work perfectly.

(2) Start the software monitor.

(3) Run workloads on all machines, and wait until all the workloads complete.

(4) Run workloads on all machines again with the same setting to duplicate the same workloads.

(5) During the execution of the second run, select one machine and send a message to the local fault injector of that machine to inject a fault. If the fault is a software fault, it is injected before the second run of the workload generator.

(6) Observe the distributed system, and record any abnormal behavior.

(7) If any machine crashes, save the dump file for further analysis.

(8) After all the clients finish the workload, stop the software monitor.

(9) Analyze the results.

   - For crashed machines, analyze the trace files to identify fault propagation and the memory dump files to obtain the reason of the crash in detail.
   - For clients that finish the workload without failure, test their outputs for irregular results and check the trace files to identify any possible undetected errors.
   - If the faulty machine does not crash, check whether the fault is activated. If it is activated, trace the source code to understand how the fault is avoided.

6.4 Fault Impact on a Distributed System

A fault injected into the kernel of a client can cause six possible effects on the distributed system:

(1) *No impact*: The fault does not cause any damage. The two possible reasons are *fault avoided* and *very long latency*. Very long latency means the fault will not be activated under normal execution conditions and the impact of the fault is unknown.

(2) *User application failure on the faulty client*: The fault causes one or more errors, but the client fails to detect these errors. The client survives but user applications are affected. Usually, the machine needs to be rebooted since the machine cannot provide full service.

(3) *Faulty client failure*: The fault causes one or more errors, and one of the errors is detected by the hardware or software detection/protection mechanism. The error is considered so serious that the client should not continue its work. The client may reboot itself or need to be checked by operators.

(4) *User application failure on other clients*: The fault propagates to the server but the server fails to detect these errors (but the server is still working). When user applications on other clients try to get service from the server, they are affected.

(5) *Server failure*: The fault propagates to the server and causes the server to crash. The server may reboot itself or need to be checked by operators. Since the server crashes, all the clients cannot get responses from the server, and all user applications that access data in the server will be suspended until the server is up.

(6) *Other client failure*: The fault propagates to the server but the server fails to detect these errors (but the server is still working). When other clients try to get service from the server, they are affected and crash. Usually, this effect should not be experienced because fault propagation through the network twice without being detected is a very serious reliability problem.

Very long latency means the fault will not be activated under normal execution conditions and the impact of the fault is unknown.

A fault injected into the kernel of a server can cause five possible effects on the distributed system:

(1) *No impact*: The fault does not cause any damage. The two possible reasons are *fault avoided* and *very long latency*. Very long latency means the fault will not be activated under normal execution condition and the impact of the fault is unknown.

(2) *User application failure on the server*: The fault causes one or more errors, but the server fails to detect these errors. The server survives but user applications are affected. Usually, the server needs to be rebooted since the server cannot provide full service.

(3) *Server failure*: The fault causes one or more errors, and one of the errors is detected by the hardware or software detection/protection mechanism. The error is considered so serious that the server should not continue its work. The server may reboot itself or need to be checked by operators.

(4) *User application failure on clients*: The fault causes one or more errors, but the server fails to detect these errors. The fault propagates to clients and affects the user applications running on the clients.

Table 6.1 Distribution of fault impact of one faulty client.

Fault Type	No Impact: Very Long Latency	No Impact: Fault Avoided	Faulty Client Failure	User Application Failure: At Faulty Client	User Application Failure: At Other Clients	Server Failure
Memory Fault	40%	8%	46%	6%	0%	0%
CPU Fault	0%	30%	70%	0%	0%	0%
Bus Fault	0%	16%	80%	4%	0%	0%
Initialization	28%	4%	68%	0%	0%	0%
Assignment	38%	6%	56%	0%	0%	2%
Condition Check	34%	20%	42%	4%	0%	0%

(5) *Client failure*: The fault propagates to the clients and causes one or more clients to crash.

The experiments in this section are performed using biased fault injections. The next two subsections describe the system behavior when faults are injected into the NFS.

6.4.1 Impact of a Faulty Client

The impact of faults injected into a client is shown in Table 6.1. All the numbers in Tables 6.1 and 6.2 are percentages.

The distributed nature of DEFINE allows us to monitor fault propagation among machines. From the experiments, we found that the impact of all the hardware faults was restricted to the faulty client and the majority of software faults do not propagate to other machines. This is because the error latency of hardware faults tends to be very short, and thus the client usually crashes in a few instructions before the fault can propagate to other parts of the kernel or other machines. In addition, the locations of all the faults are randomly selected from the specified biased range in the experiments, so they may not hit the critical locations. The specified biased range covers the entire NFS, which includes server routines that are never accessed by a client.

By using software traps to inject faults, we can distinguish very long latency faults from avoided faults. The results showed that 40% of memory faults and 33% of software faults (average of initialization, assignment, and condition check faults in Table 6.1) still are not activated although the workload is designed to activate as many faults as possible. This is because the NFS server routines are also in the client kernel, and those faults that are injected into server routines will never be activated. Besides, there are many handling routines for special situations, and it is difficult to activate those faults that are injected into these routines.

DEFINE uses hardware clock interrupts to inject CPU and bus faults, so all the CPU and bus faults are activated (no dormant faults). From Table 6.1, we found that CPU and bus faults crash the faulty client more frequently than other faults. This is because SPARC is a load/store architecture and all the memory accesses are performed indirectly via registers. Therefore, when the contents of a register are corrupted due to a CPU or bus fault, there is a high probability that the next memory access via this register will access an inaccessible location and thus cause a client failure.

If a fault has no impact, we trace the source code to understand how the fault is avoided. Those no-memory faults that have no impact usually corrupt the don't-care bits (those bits in an instruction that are not used for decoding and execution). Table 6.1 shows that 30% of CPU faults have no impact on the execution. Most of these faults corrupt registers which are overwritten before they are referenced. Another example of faults that have no impact are those faults that occur in condition codes of the processor status register (PSR), and the corrupted condition codes are not used. For bus faults that have no impact, some of them corrupt the don't-care bits when fetching instructions, and some of them do cause errors (corrupt data), but the errors remain dormant. The NFS keeps statistical data of utilization, and these data are for reference only. The faults related to these data will not have any impact on the execution and will not propagate to other parts of the kernel or other machines. In the NFS, there are many condition checks for errors that occur rarely. If we inject missing-condition-check faults for these condition checks, there is usually no impact. This accounts for most of the fault-avoided condition check faults.

More client hangs were experienced than with uniform fault injection because more faults were injected into the file system and thus more faults could affected the system dump file creation procedure when the client crashed. When this situation occurs, the client needs an operator to interrupt it with a hardware interrupt and to reboot the client without creating a dump file. If a memory dump is forced when the operator reboots the machine, the machine will be hung again, and hardware interrupts cannot rescue the machine at this moment. Turning power off and on is the only solution. In either case, no memory dump files are created, so

Table 6.2 Distribution of fault impact of one faulty server.

Fault Type	No Impact: Very Long Latency	No Impact: Fault Avoided	Faulty Server Failure	User Application Failure: At the Server	User Application Failure: At Clients	Client Failure
Memory Fault	36%	8%	44%	8%	4%	0%
CPU Fault	0%	26%	72%	2%	0%	0%
Bus Fault	0%	16%	82%	2%	0%	0%
Initialization	28%	2%	68%	2%	0%	0%
Assignment	36%	6%	50%	6%	2%	0%
Condition Check	32%	18%	44%	2%	0%	0%

source code tracing is the only way to explore the reasons of crashes, and the detailed reason of crashes might not be well understood.

6.4.2 Impact of a Faulty Server

The same faults injected into clients in the previous experiments are injected into the server, and the impact of faults is shown in Table 6.2.

Using DEFINE to monitor fault propagation among machines, we find that more faults propagate to other machines (clients) and cause user applications on clients to crash or produce incorrect outputs when faults are injected into a server. Although the server crashes many times and user applications on clients fail several times, all the clients survive through all the experiments.

DEFINE can separate dormant faults and avoided faults. From Table 6.2, we find that about 36% of memory faults and about 35% of software faults were not activated because the NFS client routines were not exercised. The reasons for non-activated faults were similar to those in the previous section.

When faults are injected into the server, more system hangs are observed—about a quarter of server failures are system hangs. It is interesting that the same faults that caused client failures with self-reboot when the faults are injected into a client might cause server hangs when the faults are injected into the server although the server and clients used the same UNIX kernel. In addition, when faults are injected into the server, the same faults might cause server failures with self-reboot or server hangs under different workload settings. That is, the fault impact depended on workloads. Since memory dump files are not created when the server is hung, further analysis of these faults should be performed to understand the detailed reasons.

In some cases, user applications on clients crashed, but user applications on the server completed without any error, while some other cases had opposite results. This is because the user application on the server accesses data through vnode manipulation routines and then call the UNIX File System (UFS), but remote file operations are handled by the NFS server routines and then call UFS routines to finish the operations. The routines involved are not the same, although many common routines are used by both UFS and NFS. In the experiments, more user application failures on the server than user application failures on clients are observed. This implies that the error checking or detection in the UFS interface may be not as good as in the NFS interface.

There was one case that the fault even corrupted the file system on the server's disk. The fault was injected into a function dealing with buffer allocation. The server crashed but could not reboot itself properly. The operator needed to do a file system check to recover unconnected files, bad references, bad flag, etc. After the server was up, the working directory was still not recovered; therefore, all the user applications were crashed because data were not accessible. In addition, all the clients needed to unmount and then mounted the file system again to access the data in that file system. This was the most serious fault impact in the experiments.

From the experimental results, we found that the server was more vulnerable than clients because the server crashed more often than clients. In addition, the impact of a faulty server was more serious than that of a faulty client because the server could not reboot itself in about one fourth of server failures.

6.4.3 Correlated Errors

In the experiments, multiple errors in different machines due to the same fault are observed. Since these errors are caused by the same faults, they are correlated errors although they occur in different machines. In field error data analysis, same errors in different machines that occur in a short period (e.g., five minutes) are considered as correlated errors [Tang92]. For these multiple errors in different machines due to the same fault, the time difference between errors can be adjusted, in the experiments, according to the workload. Since the workload used in the experiments is designed to accelerate the experiments, the time difference is short. When we change the workload on one machine to a much lighter workload (by lengthening the interarrival time between system calls) while keeping the workload on the other machines the same, the time difference can be lengthened to more than five minutes. If this is done and we only look at the error report, this error may be considered as an isolated error caused by another faults. This scenario illustrates the fact that correlated errors are not so easy to identify from field error data.

6.5 Fault Impact of Faulty User Application on the NFS

The NFS system is supposed to be able to handle any bad arguments of system calls from user applications, and returns an error code if the arguments are invalid. However, during the experiment that simulates bad users, the simulator crashes when some arguments of system calls `pathconf`, `pipe`, `poll`, and `utime` are invalid. In addition, `poll` even prints out many garbage messages on the screen. Similar experiments have been conducted to test the reliability of UNIX utilities and C libraries [Miller90, Suh92].

Except for those system calls that may crash user applications, the rest of the system calls are used in the workload generator to activate those faults that are injected into error handling routines.

6.6 Path-Based Fault Injection

In this section, a new fault injection method, path-based fault injection, is introduced. It incorporates software testing techniques, specially path testing techniques. The procedure is as follows. First, apply path testing techniques to find the interesting path and the test data that can drive the program to execute that path. Second, inject faults into the modules in the path. Third, use the test data to drive the experiment and investigate the fault impact and propagation.

The advantages of path-based fault injection are:

- All the faults can be activated.
- All possible impact of faults can be investigated.

Table 6.3 Distribution of fault impact of faulty injection along control paths.

	No Impact		Faulty	User Application Failure		
Fault Type	Very Long Latency	Fault Avoided	Client Failure	At Faulty Client	At Other Clients	Server Failure
Initialization	0%	6%	94%	0%	0%	0%
Assignment	0%	10%	86%	2%	0%	2%
Condition Check	0%	30%	64%	6%	0%	0%

In Chapter 5, about 60% of software faults are not activated. In Section 6.4, although all the faults are injected into the Sun NFS and all the workload is related to file operations, more than 30% of software faults are still not activated. In this section, path-based fault injection is applied, and all the software faults are activated. Table 6.3 lists the distribution of fault impact. The distribution of fault impact is similar to that in Table 6.1 when the very long latency faults are filtered out. This is because the activation rate is increased to 100%, and all the other situations are kept the same. The similar results from these two experiments imply that we can conduct fewer experiments by path-based fault injection to obtain results similar to biased fault injection or uniform fault injection. Although only software faults are studied, path-based fault injection can also be used to study hardware faults.

In the previous experiments, we found that the same faults may cause different impact if we repeat the experiment with the same fault but different workloads. One possible reason is that different control paths that cover the same faulty module are executed. Path-based fault injection is an appropriate approach to investigate this case, because different paths that cover the faulty module can be exercised to understand all the possible impact of this fault. This procedure is useful to validate error detection/recovery mechanisms. The disadvantage of this application is that it is usually not easy to find the test data to cover all the possible paths.

Chapter 7

Fault Propagation Model Analysis

We have mentioned that analytical models for software system behavior under faults are difficult to build. However, using the data measured in the experiments, we can build fault propagation models. The models can give an overall picture of the system behavior under faults. They can also be used to perform Markov reward analysis to provide a better understanding of the impact of faults on the system performance. Section 7.1 presents a hardware fault propagation model and a software fault propagation model. Section 7.2 conducts transient Markov reward analysis based on the models to evaluate the performance loss after a fault is injected.

7.1 Fault Propagation Models

Based on the transition probabilities measured in Section 5.4, two models are built to illustrate the fault propagation. The fault propagation model for hardware faults is shown in Figure 7.1, and the fault propagation model for software faults is shown in Figure 7.2.

7.2 Transient Markov Reward Analysis

To calculate transient performance loss during the fault propagation, a reward function is defined for the fault propagation models. Then the models are treated as Markov models and reward analysis is conducted based on them.

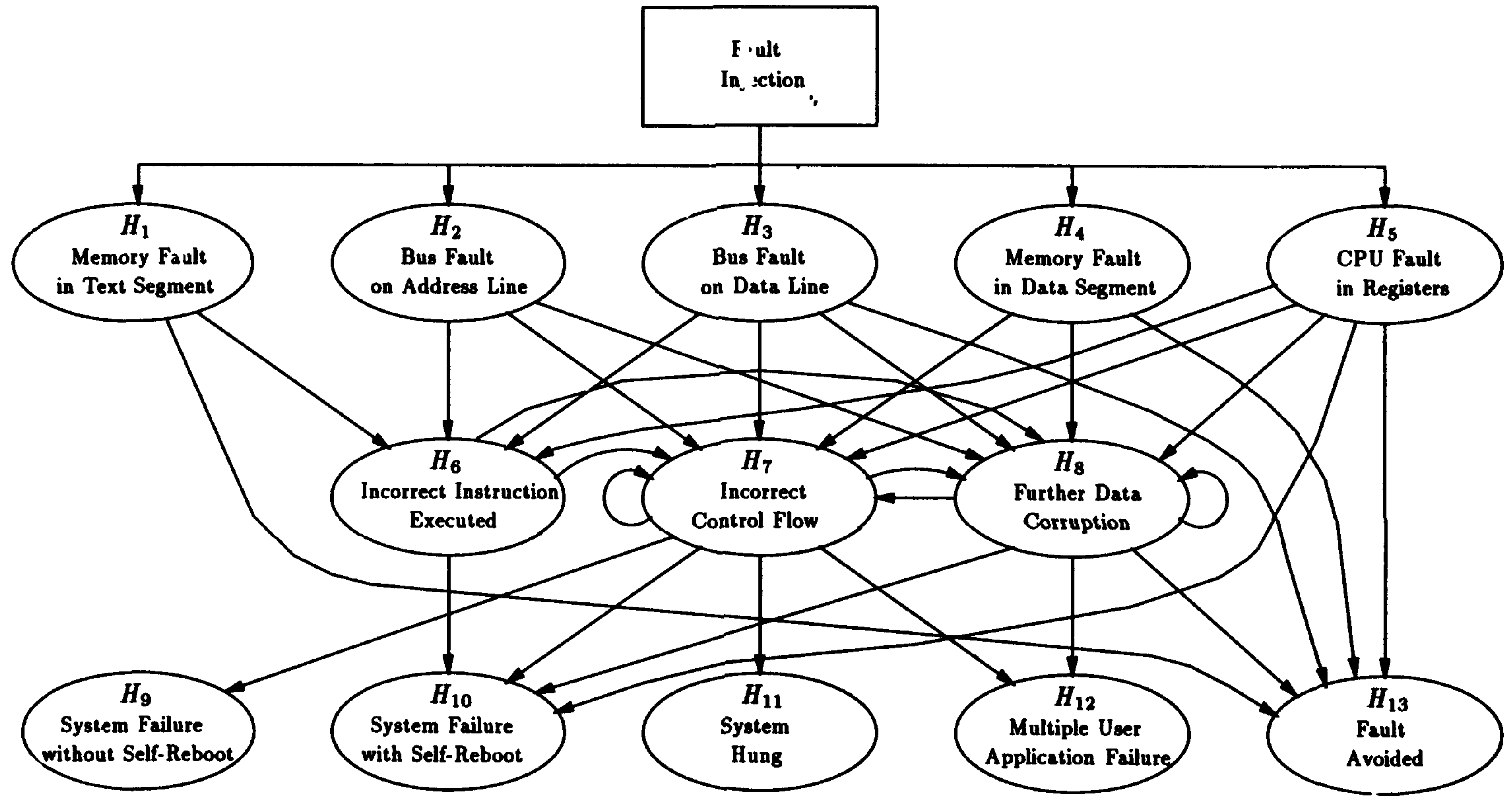
F ·ult In. ction
H_1 Memory Fault in Text Segment
H_2 Bus Fault on Address Line
H_3 Bus Fault on Data Line
H_4 Memory Fault in Data Segment
H_5 CPU Fault in Registers
H_6 Incorrect Instruction Executed
H_7 Incorrect Control Flow
H_8 Further Data Corruption
H_9 System Failure without Self-Reboot
H_{10} System Failure with Self-Reboot
H_{11} System Hung
H_{12} Multiple User Application Failure
H_{13} Fault Avoided

Figure 7.1 Fault propagation model for hardware faults.

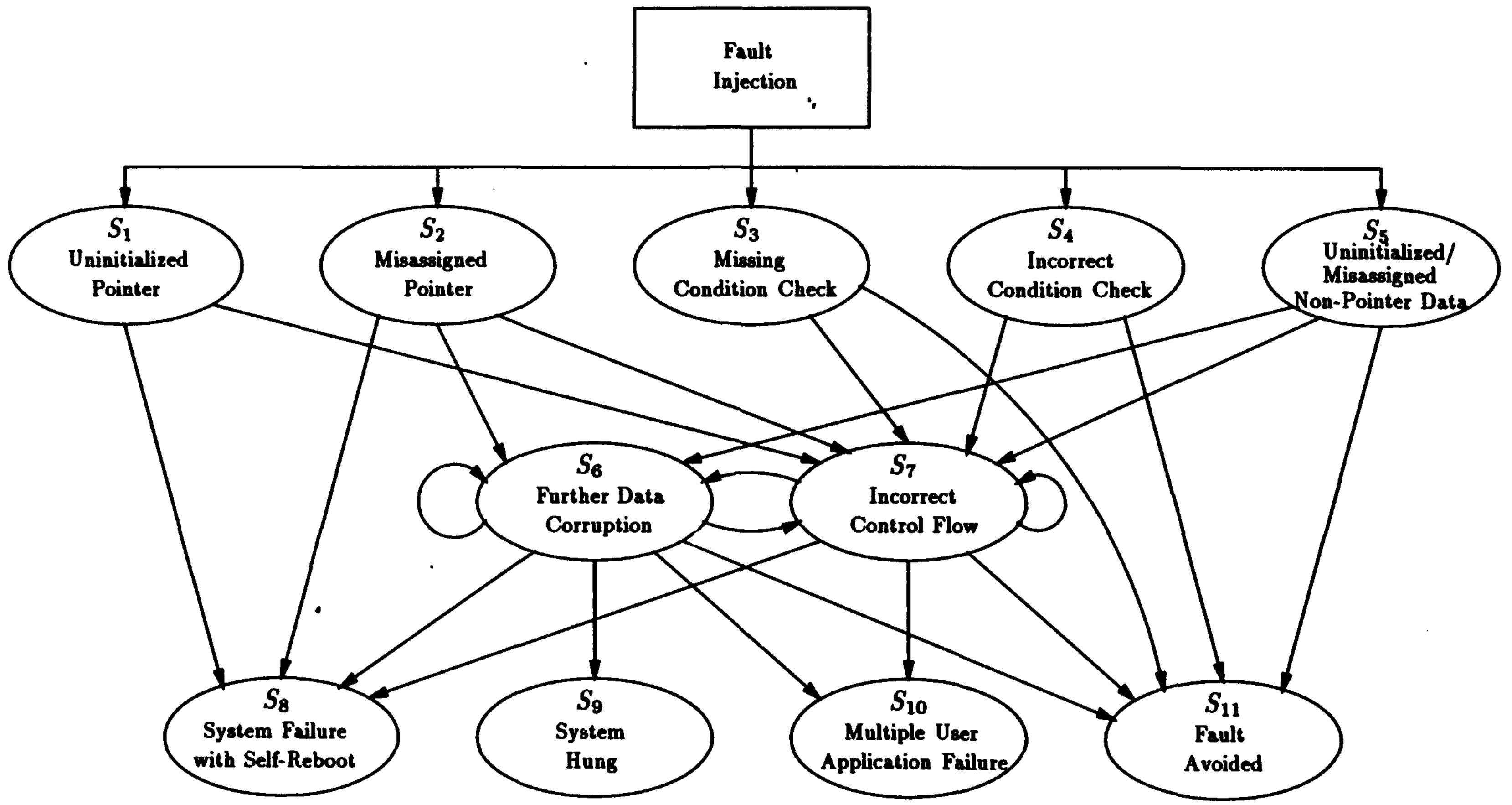
Fault Injection
S_1 Uninitialized Pointer
S_2 Missassigned Pointer
S_3 Missing Condition Check
S_4 Incorrect Condition Check
S_5 Uninitialized/ Missassigned Non-Pointer Data
S_6 Further Data Corruption
S_7 Incorrect Control Flow
S_8 System Failure with Self-Reboot
S_9 System Hung
S_{10} Multiple User Application Failure
S_{11} Fault Avoided

Figure 7.2 Fault propagation model for software faults.

State (H_i)	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8	H_9	H_{10}	H_{11}	H_{12}	H_{13}
Reward Rate (r_i)	1	1	1	1	1	0	0.5	0.3	0	0	0	0	1

Table 7.1 Reward rates for hardware fault propagation model.

State (S_i)	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}	S_{11}
Reward Rate (r_i)	1	1	1	1	1	0.5	0.3	0	0	0	1

Table 7.2 Reward rates for software fault propagation model.

The reward rate r_i for each state is defined in Tables 7.1 and 7.2. A reward rate of 1 means that the system can provide 100% of its performance; a reward rate of 0 means that the system cannot provide any service; a reward rate between 0 and 1 means that part of the performance does not contribute to useful work. After a fault is injected and before the fault is activated, the system still has 100% of performance ($r = 1$ for states H_1 to H_5 and S_1 to S_5). If the fault does not affect the system or the fault is recovered, the system returns to the normal state and provides full performance ($r = 1$ for H_{13} and S_{11}). When the system crashes or needs to be rebooted, no service is provided ($r = 0$ for H_9 to H_{12} and S_8 to S_{10}). No useful work is done when the system is executing incorrect instructions ($r = 0$ for H_6). Only partial work is useful after some data are corrupted or after the control flow is changed (H_7, H_8, S_6, and S_7). Reward rates for these states with degraded performance are estimated based on our experimental results.

To obtain expected reward rate, we need to know the holding times or latency for all transient states (non-failure and non-error states). Since the overhead of software probes is about 100 microseconds, it is impossible to measure extremely short latency. In addition, usually no more than two probes are inserted into a function. The holding time for the states into which the system goes after faults are activated cannot be measured accurately without assistance from other hardware monitors. Therefore, the mean holding time of the states are estimated based on the experimental results, as shown in Tables 7.3 and 7.4. The time unit is the execution time of one instruction. Bus faults propagate faults or are avoided immediately. CPU faults propagate faults or are avoided in a few time units. More than half of memory faults and software faults have a very long latency; the rest of the faults may take a few time units to several seconds to propagate. Executing incorrect instructions propagates faults immediately.

State (H_i)	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
Mean Holding Time	10^6	1	1	10^6	5	1	5	5

Table 7.3 Mean holding time for hardware fault propagation model.

State (S_i)	S_1	S_2	S_3	S_4	S_5	S_6	S_7
Mean Holding Time	10^6	10^6	10^6	10^6	10^6	5	5

Table 7.4 Mean holding time for software fault propagation model.

Corrupted data and incorrect control flow tend to change states in a few time units. The purpose of this study is to understand the effect and propagation of faults, not the latency; therefore, the holding times are rough estimations. The purpose of the Markov reward analysis is to show how much performance we can expect after a fault occurs. For an analysis that addresses the latency, another experiment with a hardware monitor has to be conducted to measure accurate latency. To calculate expected reward rate, system failure states (H_9, H_{10}, H_{11}, S_8, S_9), user application failure states (H_{12}, S_{10}), and non-error states (H_{13}, S_{11}) are set to absorbing states.

Given a time t, the expected instantaneous reward rate at that time $X(t)$ can be evaluated as [Goyal87, Trivedi92]:

$$E[X(t)] = \sum_i r_i p_i(t) \ ,$$

where $p_i(t)$ is the probability of the system being in state i at time t. The expected time-averaged accumulated reward over the time period $(0, t)$, i.e., the expected interval reward rate [Goyal87, Trivedi92], $Y(t)$, can be calculated by

$$E[Y(t)] = \frac{1}{t}\int_0^t \sum_i r_i p_i(x)dx \ .$$

$E[X(t)]$ is a measure of the instantaneous capacity of system performance assuming a 100% capacity at time 0 (i.e., when a fault is injected). We will call $1 - E[X(t)]$ reward loss or performance loss. $E[Y(t)]$ is a measure of the time-averaged accumulated service provided by the system. Figures 7.3 through 7.8 show the computed $E[X(t)]$ and $E[Y(t)]$ after a fault is injected.

A common feature of these figures is that after some time point, $E[X(t)]$ will be stable, approximately staying at a constant value forever. This value is an approximation of the

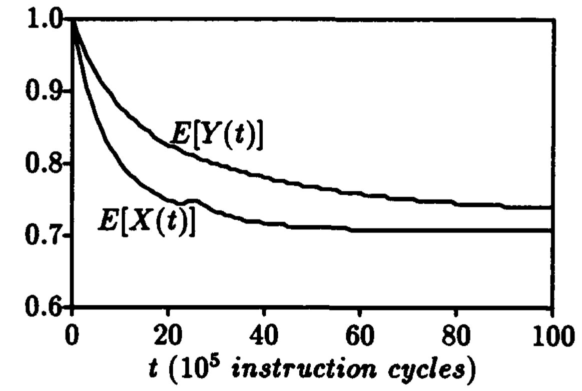

Figure 7.3 Expected reward rate after a memory fault is injected.

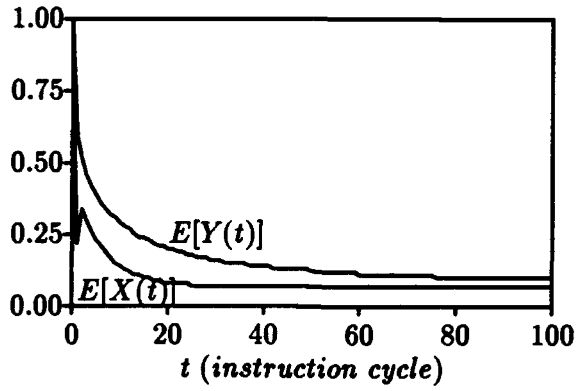

Figure 7.4 Expected reward rate after a bus fault is injected.

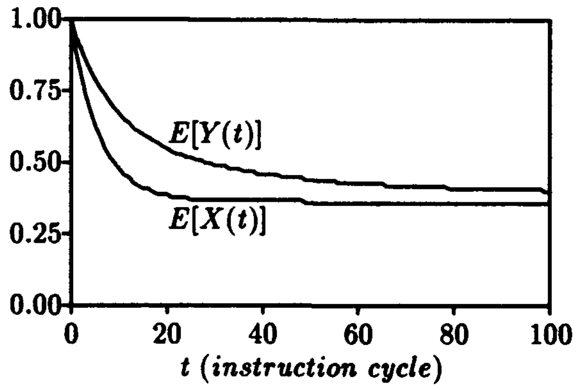

Figure 7.5 Expected reward rate after a CPU fault is injected.

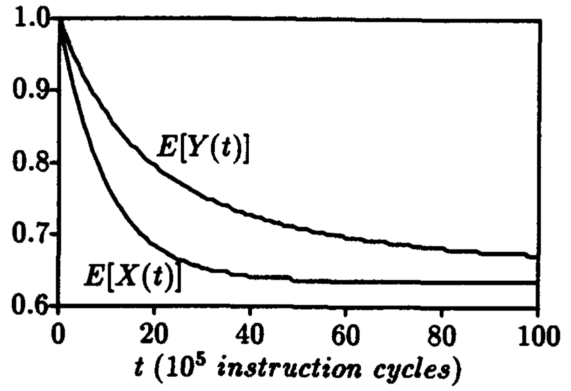

Figure 7.6 Expected reward rate after a pointer fault is injected.

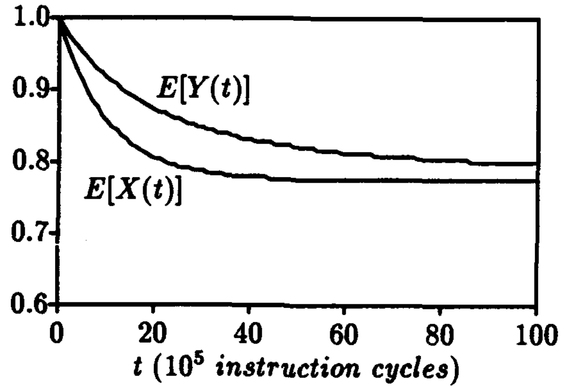

Figure 7.7 Expected reward rate after a check fault is injected.

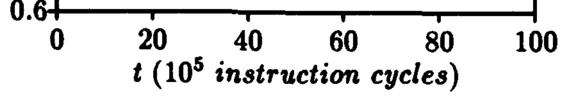

Figure 7.8 Expected reward rate after a non-pointer data fault is injected.

probability that the system will not be affected by the injected fault, assuming that a long latency has no effect on the system. The constant value also reflects the expected impact of the fault on the system in the long run. The figures show that the bus faults have the most serious impact on the system performance (reward loss = 0.93), followed by the CPU faults (0.63). All other faults have much lower performance impact (0.36 to 0.20). The impact of pointer faults (0.36) is higher than that of non-pointer faults (0.20).

The transient values of $E[X(t)]$ before they become stable quantify the system performance during the fault propagation. For instance, after a bus fault occurs, the system can be expected to function at 10% of its capacity at the sixteenth instruction. The performance loss is not so high for a CPU fault—50% of its capacity is expected at the ninth instruction. The system is expected to have better transient performance under all other faults. The time needed to reach a stable state is about 20 time units for bus and CPU faults and 2×10^6 time units for other faults. That is, compared to other faults, bus and CPU faults have a much shorter latency.

An unusual phenomenon is that $E[X(1)]$ is less than $E[X(2)]$ for the bus fault (Figure 7.4). This is because more than half of bus faults cause the system to execute incorrect instructions, and the reward rate (r_6) for executing incorrect instructions is 0, but the reward rate (r_7 and r_8) goes up after the faults propagate to "other data corrupted" or "incorrect control flow".

$E[Y(t)]$ is the expected percentage of full performance which can be delivered by the system up to the time point t, after a fault injection. If a software fault occurs, the system is expected to deliver 67% to 82% of its full performance for the 10^7 instructions of operation, while only 10% of the full performance can be delivered in 100 instruction cycles after a bus fault injection.

Chapter 8

Conclusion

Fault injection has been used to evaluate the dependability of computer systems, but some functions are not included in the previous studies. This study emphasizes the following four issues and proposes techniques to enhance the capability of software-implemented fault injection:

- Software faults are not studied.
- Detailed fault propagation in software systems are not well understood.
- The information about whether the faults are activated and when the faults are activated is not always available.
- Fault injection and activation time cannot be controlled accurately, and CPU and bus faults are not always activated.

This thesis has presented the methodology, design, and implementation of DEFINE, a distributed fault injection and monitoring environment. DEFINE injects faults into the UNIX kernel by software and traces the execution flow and key variables of the kernel by software. It solve the aforementioned issues by the following methods:

- Software faults: substitute a set of instructions with another set on line to emulate certain types of software faults.
- Fault propagation: use a software monitor to trace control flow and key variables of software systems.

- Distinction of avoided faults and non-activated faults: use software traps to inject faults so that the time of fault activation can be recorded and avoided faults and non-activated faults can be distinguished.
- Accurate time control: use hardware clock interrupts to control fault injection and activation time.

DEFINE consists of a target system, a fault injector, a software monitor, a workload generator, a controller, and several analysis utilities. The target of the study is the UNIX kernel because it is a continuously-running, highly parameterized, complex service program with high impact and a broad spectrum of workloads.

Experiments on SunOS 4.1.2 have been conducted to evaluate the impact and the fault propagation of various types of faults. Results show that memory faults and software faults usually have a very long latency, while bus faults and CPU faults tend to crash the system immediately. About half of the detected errors are data faults (47%), and they are detected while the system is trying to access a memory location where it has no privilege to access. The majority of no-impact faults are latent. Analysis of fault propagation among the UNIX subsystems reveals that only about 8% of faults propagate to other UNIX subsystems.

To increase the frequency of fault propagation through the network, biased fault injection has been applied, with more faults injected into the Sun Network File System, a distributed file system. Another approach, path-based fault injection is also applied to activate all the software faults. Experimental results show that fault propagation from servers to clients occurs more frequently than from clients to servers. The fault impact depends on the workload.

Fault propagation models are built for both software and hardware faults. Transient Markov reward analysis has been performed based on the models to evaluate the performance loss after a fault is injected. Markov reward analysis shows that the performance loss incurred by bus faults (0.93) and CPU faults (0.63) are much higher than that incurred by software and memory faults (0.20 to 0.36). Among software faults, the impact of pointer faults (0.36) is higher than that of non-pointer faults (0.20).

8.1 Limitations

DEFINE can inject software faults as well as hardware faults, can trace fault propagation, can monitor whether and when faults are activated, and has accurate time control. However, there are several limitations. First, when injecting intermittent faults, the interarrival times may not follow the specified distribution perfectly. This is because hardware interrupts may be disabled temporarily, lengthening the interarrival times. Second, the overhead of software monitor may up to 50% of system time, though the user time is not affected. If the focus is not on fault propagation, the software monitor can be turned off to eliminate this overhead.

The target software system is the UNIX kernel, whose operations are not deterministic; thus, the experiments are not always repeatable. Some interesting results, such as the one that crashed the local disk, could not be repeated for further investigation.

8.2 Suggestions for Future Work

This is the first attempt of the detailed investigation of fault propagation in the UNIX system, and DEFINE is shown to be a useful tool for this study. Possible improvements and follow-up work include the following:

- DEFINE can inject many types of hardware faults, but only a few types of software faults can be injected now. More mechanisms will be developed to cover other types of software faults.
- Using software traps to activate faults can tell us whether faults are activated for most faults, but it cannot be applied to memory faults in the data/stack segment. Another mechanism is needed to solve this problem.
- There is no special error detection/recovery mechanism embedded in the SunOS, so faults can propagate and crash the system easily. Error detection/recovery mechanisms could be added to the kernel, and DEFINE could be used to evaluate the effectiveness of the error detection/recovery mechanisms.
- The obtained understanding of fault characteristics can be used to design error detection/recovery mechanisms.

- The methodology can be applied to user applications by inserting probes into the target applications. The current probes need to be modified so that messages can be passed to the message recorder in the kernel. Possible solutions include using a new system call or using a pseudo device. If fault propagation is not the objective, probes can be ignored. Since each user application has its own user address space, the fault issuer has to be inserted into the application. The fault injector also needs to be modified to inject faults into user address spaces.

Bibliography

[Alewine92] N. J. Alewine, S.-K. Chen, C.-C. Li, W. K. Fuchs, and W.-M. Hwu, "Branch recoverv with compiler-assisted multiple instruction retry," in *The 22th Int. Symp. on Fault-Tolerant Computing*, pp. 66–73, June 1992.

[Arlat89] J. Arlat, Y. Crouzet, and J.-C. Laprie, "Fault injection for dependability validation of fault-tolerant computing systems," in *The 19th Int. Symp. on Fault-Tolerant Computing*, pp. 348–355, June 1989.

[Arlat90] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J. C. Fabre, J. C. Laprie, E. Martins, and D. Powell, "Fault injection for dependability validation: a methodology and some applications," *IEEE Tran. on Software Engineering*, vol. 16, pp. 166–182, Feb. 1990.

[Barton90] J. H. Barton, E. W. Czeck, Z. Segall, and D. P. Siewiorek, "Fault injection experiments using FIAT," *IEEE Tran. on Computers*, vol. 39, pp. 575–582, Apr. 1990.

[Beizer90] B. Beizer, *Software Testing Techniques*. New York: Van Nostrand Reinhold Company, 1990.

[Bodnarchuk91] R. R. Bodnarchuk and R. B. Bunt, "A synthetic workload model for a distributed system file server," in *Proc. 1991 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pp. 50–59, May 1991.

[Buchholz69] W. Buchholz, "A synthetic job for measuring system performance," *IBM Syst. J.*, vol. 8, no. 4, pp. 309–318, 1969.

[Cabrera81] L. F. Cabrera, "Benchmarking unix: a comparative study," in *Experimental Computer Performance Evaluation* (D. Ferrari and M. Spadoni, eds.), pp. 205–215, North-Holland, 1981.

[Calzarossa85] M. Calzarossa and G. Serazzi, "A characterization of the variation in time of workload arrival patterns," *IEEE Tran. on Computers*, vol. C-34, pp. 156–162, Feb. 1985.

[Chillarege87] R. Chillarege and R. K. Iyer, "Measurement-based analysis of error latency," *IEEE Tran. on Computers*, vol. 36, pp. 529–537, May 1987.

[Chillarege89] R. Chillarege and N. S. Bowen, "Understanding large system failures – a fault injection experiment," in *The 19th Int. Symp. on Fault-Tolerant Computing*, pp. 356–363, June 1989.

[Chillarege91] R. Chillarige, W.-L. Kao, and R. G. Conduit, "Defect type and its impact on the growth curve," in *The 13th Int. Conf. on Software Engineering*, pp. 246–255, May 1991.

[Cusick85] J. Cusick, R. Koga, W. Kolasinski, and C. King, "SEU vulnerability of the Zilog Z-80 and NSC-800 microprocessors," *IEEE Tran. on Nuclear Science*, vol. 32, pp. 4206–4211, Dec. 1985.

[Czeck92] E. W. Czeck and D. P. Siewiorek, "Observations on the effects of fault manifestation as a function of workload," *IEEE Tran. on Computers*, vol. 41, pp. 559–566, May 1992.

[DeMillo78] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: help for the practicing programmer," *IEEE Computer*, pp. 34–41, Apr. 1978.

[DeMillo91] R. A. DeMillo and A. J. Offutt, "Constraint-based automatic test data generation," *IEEE Tran. on Software Engineering*, vol. 17, pp. 900–910, Sept. 1991.

[Devarakonda86] M. V. Devarakonda and R. K. Iyer, "A user-oriented analysis of file usage in unix," in *Proc. COMSAC-86*, pp. 21–27, 1986.

[Devarakonda88] M. V. Devarakonda, *File Usage Analysis and Resource Usage Prediction: A Measurement-Based Study.* PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1988.

[Devarakonda90] M. Devarakonda, K. Goswami, and R. Chillarege, "Failure characterization of the NFS using fault-injection," Research Report RC 16342, IBM, Dec. 1990.

[Domanski82] B. Domanski, "Load driving a system," *Computer Performance*, vol. 3, no. 4, pp. 195–200, 1982.

[Duba88] P. Duba and R. Iyer, "Transient fault behavior in a microprocessor: a case study," in *Proc. 1988 IEEE Int. Conf. Computer Design: VLSI in Computers and Processors*, pp. 272–276, Oct. 1988.

[Endres75] A. Endres, "An analysis of errors and their causes in system programs," *IEEE Tran. on Software Engineering*, vol. 1, pp. 140–149, June 1975.

[Ferrari81] D. Ferrari and M. Spadoni, *Experimental computer performance evaluation.* North-Holland, 1981.

[Ferrari83] D. Ferrari, G. Serazzi, and A. Zeigner, *Measurement and tuning of computer systems.* Prentice-Hall, 1983.

[Finelli87] G. B. Finelli, "Characterization of fault recovery through fault injection on FTMP," *IEEE Tran. on Reliability*, vol. 36, pp. 164–170, June 1987.

[Frankl93] P. G. Frankl and R. J. Weyuker, "Provable improvements on branch testing " *IEEE Tran. on Software Engineering*, vol. 19, pp. 962–975, Oct. 1993.

[Gould86] E. Gould, "The network file system implemented on 4.3BSD," in *USENIX Summer Conference Proceedings*, pp. 294–298, June 1986.

[Goyal87] A. Goyal, S. S. Lavenberg, and K. S. Trivedi, "Probabilistic modeling of computer system availability," *Annals of Operations Research*, vol. 8, pp. 285–306, Mar. 1987.

[Gray90] J. Gray, "A census of tandem system availability between 1985 and 1990," *IEEE Tran. on Reliability*, Oct. 1990.

[Gunneflo89] U. Gunneflo, J. Karlsson, and J. Torin, "Evaluation of error detection schemes using fault injection by heavy-ion radiation," in *The 19th Int. Symp. on Fault-Tolerant Computing*, pp. 348–355, June 1989.

[Han93] S. Han, H. A. Rosenberg, and K. G. Shin, "DOCTOR: an integrated software fault injection environment," CSE Technical Report CSE-TR-192-93, Department of Electrical Engineering and Computer Science, The University of Michigan, Dec. 1993.

[Hedley85] D. Hedley and M. A. Hennell, "The causes and effects of infeasible paths in computer programs," in *The 8th Int. Conf. on Software Engineering*, pp. 259–266, 1985.

[Howard88] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West, "Scale and performance in a distributed file system," *ACM Tran. Computer Systems*, vol. 6, pp. 51–81, Feb. 1988.

[Howden76] W. E. Howden, "Reliability of the path analysis testing strategy," *IEEE Tran. on Software Engineering*, vol. SE-2, pp. 208–215, Sept. 1976.

[Howden82] W. E. Howden, "Weak mutation testing and completeness of test sets," *IEEE Tran. on Software Engineering*, vol. SE-8, pp. 371–379, July 1982.

[Hughes84] H. D. Hughes, "Generating a drive workload from clustered data," *Computer Performance*, vol. 5, no. 1, pp. 31–37, 1984.

[Iyer82] R. K. Iyer and D. J. Rossetti, "A statistical load dependency model for cpu errors at slac," in *The 12th Int. Symp. on Fault-Tolerant Computing*, pp. 363–372, June 1982.

[Iyer85] R. K. Iyer and D. J. Rossetti, "Effect of system workload on operating system reliability: a study on IBM 3081," *IEEE Tran. on Software Engineering*, vol. SE-11, pp. 1438–1448, Dec. 1985.

[Iyer86] R. K. Iyer, D. J. Rossetti, and M. C. Hsueh, "Measurement and modeling of computer reliability as affected by system activity," *ACM Tran. on Computer Systems*, vol. 4, pp. 214–237, Aug. 1986.

[Iyer94] R. K. Iyer and D. Tang, "Experimental analysis in computer system dependability," in *Fault-Tolerant Computing* (D. K. Pradhan, ed.), Prentice Hall, second ed., 1994.

[Kanawati92] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham, "FERRARI - a tool for the validation of system dependability," in *The 22nd Int. Symp. on Fault-Tolerant Computing*, pp. 336–344, July 1992.

[Kant90] K. Kant and A. Ravichandran, "Synthesizing robust data structures–an introduction," *IEEE Tran. on Computers*, vol. 39, pp. 161–173, Feb. 1990.

[Kao92] W.-L. Kao and R. K. Iyer, "A user-oriented synthetic workload generator," in *The 12th Int. Conf. on Distributed Computing Systems*, pp. 270–277, June 1992.

[Kao93] W.-L. Kao, R. K. Iyer, and D. Tang, "FINE: A fault injection and monitoring environment for tracing the UNIX system behavior under faults," *IEEE Trans. on Software Engineering*, vol. 19, Nov. 1993.

[Kao93a] W.-L. Kao, D. Tang, and R. K. Iyer, "Study of fault propagation using fault injection in the unix system," in *The 2nd Asian Test Symposium*, pp. 38–43, Nov. 1993.

[Kao94] W.-L. Kao and R. K. Iyer, "DEFINE: A distributed fault injection and monitoring environment," in *The 1994 IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, 1994. to appear.

[Karlsson89] J. Karlsson, U. Gunneflo, and J. Torin, "The effects of heavy-ion induced single event upsets in the MC6809E microprocessor," in *The 4th Int. Conf. on Fault-Tolerant Computing Systems, GI/ITG/GMA*, 1989.

[Lala83] J. Lala, "Fault detection, isolation and reconfiguration in FTMP: methods and experimental results," in *The 5th AIAA/IEEE Digital Avionics Systems conference*, pp. 21.3.1–21.3.9, 1983.

[Laprie85] J.-C. Laprie, "Dependable computing and fault tolerance: concepts and terminology," in *The 15th Int. Symp. on Fault-Tolerant Computing*, pp. 2–11, June 1985.

[Lee92] I. Lee and R. K. Iyer, "Analysis of software halts in the Tandem GUARDIAN operating system," in *The Third Int. Symp. on Software Reliability Engineering*, pp. 227–236, Oct. 1992.

[Lee93] I. Lee and R. K. Iyer, "Measurement-based reliability analysis of the Tandem GUARDIAN90 operating system," submitted to *IEEE Trans. on Software Engineering.*

[Li91] C.-C. J. Li, S.-K. Chen, W. K. Fuchs, and W.-M. Hwu, "Compiler-assisted multiple instruction retry," Technical Report CRHC-91-31, Coordinated Science Laboratory, Univ. of Illinois, May 1991.

[Miller90] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," in *Communications of the ACM*, pp. 32–43, Dec. 1990.

[Morell90] L. J. Morell, "A theory of fault-based testing," *IEEE Tran. on Software Engineering*, vol. 16, pp. 844–857, Aug. 1990.

[Ousterhout85] J. Ousterhout, D. Da Costa, D. Harrison, J. A. Kunze, M. Kupfer, and J. G. Thompson, "A trace-driven analysis of the unix 4.2 bsd file system," in *Proc. of the 10th ACM Symposium on Operating System Principles*, pp. 15–24, 1985.

[Randell75] B. Randell, "System structure for software fault tolerance," *IEEE Tran. on Software Engineering*, vol. 1, pp. 220–232, June 1975.

[Rosenberg93] H. A. Rosenberg and K. G. Shin, "Software fault injection and its application in distributed systems," in *The 23rd Int. Symp. on Fault-Tolerant Computing*, pp. 208–217, June 1993.

[Saleh90] A. M. Saleh, J. J. Serrano, and J. H. Patel, "Reliability of scrubbing recovery-techniques for memory systems," *IEEE Tran. on Reliability*, vol. 39, pp. 114–122, Apr. 1990.

[Schuette86] M. A. Schuette, J. P. Shen, D. P. Siewiorek, and Y. Zhu, "Experimental evaluation of two concurrent error detection schemes," in *The 16th I .t. Symp. on Fault-Tolerant Computing*, pp. 138–143, July 1986.

[Schuette87] M. A. Schuette and J. P. Shen, "Processor control flow monitoring using signatured instruction streams," *IEEE Tran. on Computers*, vol. 36, pp. 264–276, Mar. 1987.

[Segall88] Z. Segall, D. Vrsalovic, D. Siewiorek, D. Yaskin, J. Kownacki, J. Barton, R. Dancey, A. Robinson, and T. Lin, "FIAT – fault injection based automated testing environment," in *The 18th Int. Symp. on Fault-Tolerant Computing*, pp. 102–107, June 1988.

[Serazzi86] G. Serazzi, *Workload characterization of computer systems and computer networks*. North-Holland, 1986.

[Shin84] K. Shin and Y.-H. Lee, "Error detection process–model, design, and its impact on computer performance," *IEEE Tran. on Computers*, vol. 33, pp. 529–540, June 1984.

[Shin86] K. Shin and Y.-H. Lee, "Measurement and application of fault latency," *IEEE Tran. on Computers*, vol. 35, Apr. 1986.

[Sreenivasan74] K. Sreenivasan and A. J. Kleinman, "On the construction of a representative synthetic workload," *CACM*, vol. 17, pp. 127–133, Mar. 1974.

[Suh92] B.-H. Suh, J. Hudak, D. Siewiorek, and Z. Segall, "Development of a benchmark to measure system robustness: experiences and lessons learned," in *The Third Int. Symp. on Software Reliability Engineering*, pp. 237–245, Oct. 1992.

[Sullivan91] M. Sullivan and R. Chillarege, "Software defects and their impact on system availability - a study of field failures in operating systems," in *The 21th Int. Symp. on Fault-Tolerant Computing*, pp. 2–9, June 1991.

[Svensson90] A. Svensson, "History, an intelligent load sharing filter," in *The 10thInternational Conference on Distributed Computing Systems*, pp. 546–553, May 1990.

[Tang92] D. Tang and R. K. Iyer, "Analysis of the VAX/VMS error logs in multicomputer environments – a case study of software dependability," in *The Third Int. Symp. on Software Reliability Engineering*, pp. 216–226, Oct. 1992.

[Taylor80] D. J. Taylor, D. E. Morgan, and J. P. Black, "Redundancy in data structures: improving software fault tolerance," *IEEE Tran. on Software Engineering*, vol. 6, pp. 595–602, Nov. 1980.

[Trivedi92] K. S. Trivedi, J. K. Muppala, S. P. Woolet, and B. R. Haverkort, "Composite performance and dependability analysis," *Performance Evaluation*, vol. 14, pp. 197–215, Feb. 1992.

[Wood71] D. C. Wood and E. H. Forman, "Throughput measurement using a synthetic job stream," in *AFIPS Conf. Proc. FJCC*, vol. 39, pp. 51–56, 1971.

[Woodward80] M. R. Woodward, D. Hedley, and M. A. Hennell, "Experience with path analysis and testing of programs," *IEEE Tran. on Software Engineering*, vol. SE-6, pp. 278–286, May 1980.

[Yang92] F. L. Yang, *Simulation of Faults Causing Analog Behavior in Digital Circuits*. PhD thesis, Dept. of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, May 1992.

[Young92] L. T. Young, R. K. Iyer, K. K. Goswami, and C. Alonso, "A hybrid monitor assisted fault injection environment," in *The Third IFIP Working Conf. on Dependable Computing for Critical Applications*, Sept. 1992.

[Zeil83] S. J. Zeil, "Testing for perturbations of program statements," *IEEE Tran. on Software Engineering*, vol. SE-9, pp. 335–346, May 1983.

[Levendel90] Y. Levendel, "Reliability analysis of large software systems," *IEEE Tran. on Software Engineering*, vol. 16, pp. 141–152, Feb. 1990.

Vita

Wei-lun Kao was born on August 3, 1963 in Taipei, Taiwan, Republic of China. He received his B.S. degree in Computer Science and Information Engineering from the National Taiwan University in 1985, and the M.S. and Ph.D. degrees in Computer Science from the University of Illinois at Urbana-Champaign in 1991 and 1994, respectively.

He worked as an instructor and software engineer at Computer Center, Chinese Military Academy from 1985 to 1987. While pursuing graduate studies at the University of Illinois, he was a research assistant in the Center for Reliable and High-Performance Computing, Coordinated Science Laboratory, from 1988 to 1994. He was with IBM T. J. Watson Research Center, Yorktown Heights, New York, in the summer of 1989, where he analyzed software defects and applied software reliability models to the analysis.

Mr. Kao will now join Silicon Graphics, Inc. in Mountain View, California.