AN EVALUATION OF THE RECOVERY-RELATED PROPERTIES OF SOFTWARE FAULTS

by

Subhachandra Chandra

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2000

For Mrunalini Sasanka, Prasad, Manjeera,
Rohini, Madhusudana Rao and Naveen
and all my teachers

ACKNOWLEDGEMENTS

This dissertation is the results of several years of work, during which time a lot of people have supported me and my work. I owe the biggest thanks to my advisor Peter Chen. He helped shape the work in this dissertation during the many conversations we had in his office or over the phone while he was on either side of the "big pond".

I would also like to thank the members of my thesis committee. Their comments were very helpful and made the dissertation better and more polished.

Thanks to the other members of the Rio group namely Dave, Weeteck and George for all the good times they provided in 2003 EECS. Also, thanks to the other students in 2003 for their company during lunch hours and all the conversations we had.

I should also thank all my wonderful friends in Ann Arbor, all over the U.S. and back in India. Thank you Baljit, Richie and Gaurav especially for all the wild times we had in Ann Arbor.

Last but not the least I should thank my family. I am fortunate to have such wonderful parents, sister, my aunt and uncle and my cousin. Thanks for your support all through my life. Thanks also to the rest of my family in the USA and back home in India.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 Motivation

Computers have become an integral part of our daily life and are required to maintain the high standards of living we have become accustomed to. As we become more and more dependent on computers to do everyday tasks, the reliability of computer systems becomes that much more important. The failure of computer systems can cause loss of human lives, environmental catastrophes and the widespread disruption of business systems all over the world. A recent example is the “ILOVEYOU” virus which caused a small fraction of computers to fail and yet is estimated to have caused damage worth billions of dollars in just a couple of days.

Applications that require fault-tolerant computing can be categorized into five primary areas with decreasing requirements of fault-tolerance. They are safety-critical computations, long-life applications, high availability applications, maintenance postponement applications and general purpose applications [Pradhan95]. Safety-critical applications are critical to human safety, environmental cleanliness or equipment protection. Some examples are aircraft control systems, industrial controllers in chemical and nuclear plants and military systems. Long life applications are expected to be operational for years with some of them expected to work for tens of years. The most common examples of long-life applications are communications satellites and deep-space probes. Banking, stock market and reservations systems are well known examples of highly available systems. These systems are expected to be continuously available throughout their lifetime. Maintenance postponement applications are used where maintenance operations are expensive, inconvenient or difficult to perform. Remote processing stations, long duration unmanned space flights and telephone switching applications are some examples of such systems. Word processors, spread sheets and web browsers are examples of general purpose applications. These applications are neither safety critical nor do they need to be highly available. But these applications have a high rate of failure and faults in them are directly visible to the common man or woman.

Unfortunately, we have not yet succeeded in building fault free computer systems. Computers fail due to a variety of problems with their hardware and software. Field studies [Gray91] and everyday experience show that the dominant cause of failures today is software faults, both in the application and system layers. Reducing the number of software faults and surviving the ones that remain has been an important challenge for the fault-tolerance community. Researchers have designed a number of recovery systems to enable software systems to survive faults and continue operation. Recovery can be handled in an application-specific manner, or it may be accomplished via general recovery mechanisms, such as checkpointing or logging [Elnozahy99]. All recovery techniques for software faults write data to stable storage (storage that survives the fault) and use the saved data to recover after the crash.

Recovery systems make two assumptions about the behavior of faults to achieve a successful recovery operation. One assumption is that the recovery mechanism only saves good data to stable storage and is part of the definition of the fail-stop property. Once this assumption is made the recovery system expects to succeed in recovering from the fault every time. The second assumption is that the fault is non-deterministic. Without this assumption the recovery system will not be able to recover as the same fault will happen again after recovery.

Almost all recovery systems are dependent on the assumptions made above about the properties of software faults to be able to recover from them. There has been very little work done in evaluating these properties of faults. Further, we found no work which actually evaluates the assumptions made about these properties with respect to recovery after a failure.

Our goal with this thesis is to measure how well the two assumptions hold good in software systems.

1.2 Overview

The rest of this chapter looks at the scope of this thesis and introduces some common terminology used throughout this document. The last section of this chapter describes in brief the software we used to perform some of the evaluation work done in this thesis. This software is the work of other students during their dissertation work at the University of Michigan.

In Chapter 2, we look at the basic techniques used by recovery mechanisms to recover from faults. We describe a few commonly used mechanisms to detect and recover from faults. We then look at the unique properties exhibited by software faults and discuss the properties of software faults that we make assumptions about in recovery systems.

In Chapter 3, we evaluate the assumption made about the fail-stop property of software faults. We discuss how the technique of saving state for the purpose of recovery affects the fail-stop property. The evaluation is done by injecting faults at both the application level and the operating system level and looking at the how well the recovery mechanisms work. We also compare between recovery mechanisms built into the application and generic recovery mechanisms. We find that recovery mechanisms built into the application work a lot better than generic-recovery mechanisms.

In Chapter 4, we evaluate the assumption made about the non-deterministic property of software faults. We categorize faults based on their degree of non-determinism. We then analyze bug reports of some open-source applications to evaluate the assumption. Surprisingly, we find that a majority of software faults in open-source applications are deterministic.

In Chapter 5, we sum up all our work and the conclusions we obtained from our study of the properties of software faults. We then discuss future work that can be done to expand our study and also look at the effects of our study on research in software fault recovery mechanisms.

1.3 Scope

This dissertation investigates the following thesis: How closely do software faults follow the assumptions made about their behavior by recovery mechanisms? These assumptions need to be evaluated to ascertain the usefulness of recovery mechanisms.

Our investigation of this thesis is based on measuring the properties of software faults about which assumptions are made. We evaluate commonly used software to make our study applicable to software fault-tolerance in everyday life. Our investigation led to the following contributions

- showing that application specific recovery mechanisms work better than generic-recovery mechanisms.

- showing that there is a conflict between the need for failure-transparency and the need to make faults fail-stop while saving state.
- showing that the assumptions made by generic-recovery mechanisms about the fail-stop property of software faults are not valid 40-50% of the time.
- showing that software faults in open-source applications are mostly deterministic or in other words environment-independent.

1.4 Terminology

The three fundamental terms used in fault-tolerant design are fault, error and failure. Faults are the cause of errors and errors cause failures [Pradhan95].

A fault is a physical defect, imperfection or a design flaw that occurs in a hardware or a software component. An error is a change in the system state caused by the activation or execution of the fault. It is a deviation from the correct behavior of a system. A failure is the nonperformance or incorrect performance of some action that is expected of the system by the user. Most users of fault-tolerant systems only perceive the failure of the system and usually have no information about the fault that caused the failure. In the case of software where a faulty piece of code is present throughout the software's lifetime there is also the concept of a fault trigger. A fault trigger is an event which causes the faulty piece of code to be executed.

The cause-effect relationship between faults, errors and failures leads to the definition of the parameter: fault latency. Fault latency is the length of time between the occurrence of a fault or in the case of design faults the occurrence of a fault trigger and the appearance of the resulting failure.

A couple of attributes used to describe faults are the nature and duration of faults. The nature of a faults refers to whether the faults are software faults or a hardware faults. The duration of a fault refers to the way the faults are activated. Permanent faults are activated overtime the piece of code is executed or the piece of hardware is used until corrective action is taken. These are also referred to as bohrbugs in software. Transient or intermittent faults manifest themselves in a non-deterministic fashion. Their activation depends upon certain environmental conditions to occur and these environmental conditions can change in a non-deterministic manner. It may be hard to recreate the exact environment which activates the fault and so makes it hard to diagnose and debug the fault. Transient faults are also referred to as heisenbugs due to the difficulty in measuring and analyzing them. The recovery methods discussed later in this proposal are aimed at recovering from transient faults in software systems.

1.5 Software Used: DC and Kernel Fault Injector

This section gives an overview of two pieces of software used in some the experiments performed in this dissertation. These pieces of software were authored by other researchers at the University of Michigan.

Discount Checking (DC) [Lowell99] provides fast, user-level, full-process checkpoints. DC uses reliable main memory provided by Rio [Chen96] and lightweight and fast transactions provided by Vista [Lowell97] to provide fast checkpointing. We use Discount Checking as an example of a generic recovery mechanism in our evaluation of the fail-stop property of software.

Discount Checking uses transactions to implement checkpointing. The entire process state is mapped into persistent memory and “transaction_begin” and “transaction_end” calls are inserted to make sure that any changes to the process state are done within the body of a transaction. The commit of a transaction is equivalent to committing a checkpoint.

Discount Checking needs to save three types of state to take a complete checkpoint of a process: process’s address space, state of the process in the processor, and state in the kernel related to the process.

Discount Checking saves the process’s address space using Vista. It loads the process’s data and stack segments into the recoverable memory provided by Vista. Another segment in Vista’s recoverable memory is used to allocate heap memory. The process then executes with in the recoverable memory provided by Vista. All changes to Vista’s memory are saved in an undo log until the next commit/checkpoint is taken. In the event of a process failure, Vista uses the undo log to restore the memory image of the process to where it was at the last checkpoint.

A process’s state is also partly held in the processor in the form of registers, program counter, stack pointer etc. Discount Checking uses the `setjmp` function provided by libc to save a copy of the processor state at each checkpoint.

The basic strategy used by Discount Checking to save kernel state is to intercept system calls that modify state in the kernel and save the updated state in Vista’s memory. During recovery the system calls are redone to restore the kernel state for the recovering process. Some examples of kernel state saved by DC are state related to open files and sockets, signals and timers.

Discount Checking is very easy to incorporate into applications for which source code is available. There are two minor source modifications that need to be performed.

First, the header file `dc.h` must be included in the source file that has the function `main` defined. Second, a call to the function `dc_init` must be inserted as the first line of the function `main`. The function `dc_init` loads the program into Vista's recoverable memory and starts the first checkpoint interval. After making the two changes, the program can be compiled and needs to be linked with the two libraries `libdc.a` and `libvista.a`. The resulting executable now will take checkpoints based on the checkpointing strategy used. If the process crashes the same executable can be rerun and it will recover from the last checkpoint taken.

The kernel fault injector was developed by Weeteck Ng as part of his dissertation work[Ng99]. The fault injection tool uses object-code modification to inject faults into the kernel text. It is embedded into the kernel and can be triggered using a system call. The tool when triggered, randomly selects an instruction in the kernel text and modifies the instruction to reflect the kind of fault we intend to inject.

CHAPTER 2

SOFTWARE FAULT RECOVERY AND ASSUMPTIONS

2.1 Introduction

As computers become an integral part of today's society, making them dependable becomes increasingly important. Field studies [Gray91] and everyday experience make it clear that the dominant cause of failures today is software faults, both in the application and system layers. Reducing the number of failures caused by software faults is therefore an important challenge for the fault-tolerance community.

The best way to ensure fault-tolerance is to avoid faults in the first place. But in reality faults occur due to a variety of reasons. So in the presence of faults, the two major components of fault-tolerance are the ability to detect that a fault or an error as a manifestation of the fault has occurred and the ability to eliminate the effects of the fault and continue the correct operation of the system.

Figure 2.1 on page 10 shows the time line for a generic fault-tolerance mechanism that detects a fault and tries to recover from the resulting error or failure. Most of these mechanisms save the state of the process regularly to maintain failure transparency. Failure transparency aims to mask the effects of the failure from the user. This means that the recovery mechanism has to deal with the issue of not displaying inconsistent visible events to the outside world. The figure shows one such point in time where the state was saved. The figure then shows the occurrence of a fault and an eventual detection of the fault. A failure of the process can be thought of as a very crude fault detection mechanism. The recovery mechanism then tries to restore the process to the state that was last saved and

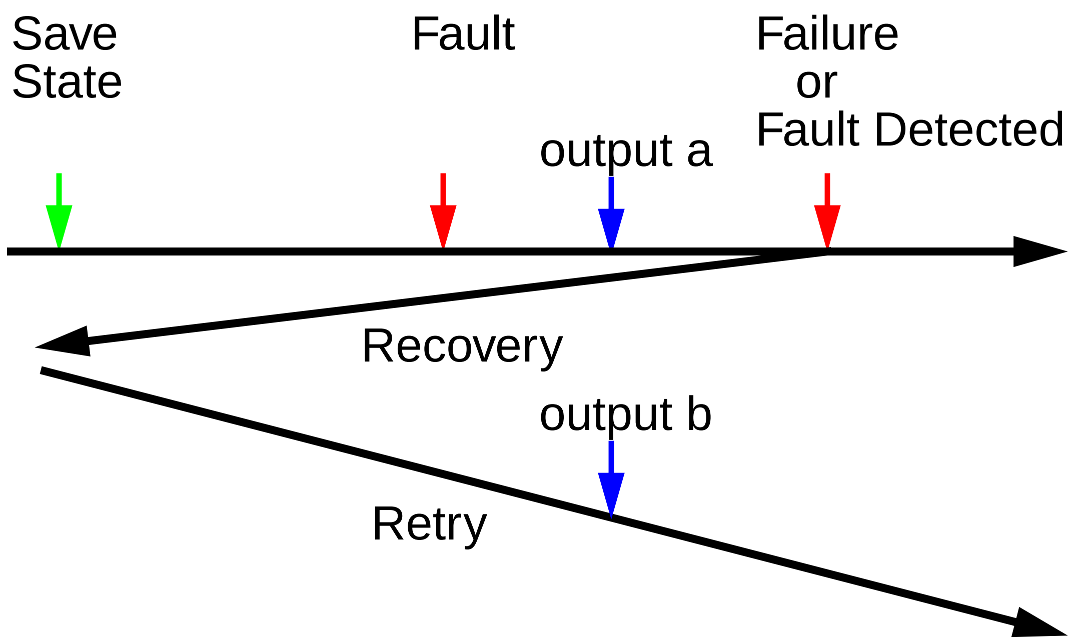

Figure 2.1 A generic time line for fault detection and recovery

retries the computation from there on. There are two assumptions made implicitly by this fault-tolerance mechanism. The first one is that the saved state does not contain the state of the fault itself or any state corrupted by the fault. The second one is that after recovery the same fault does not happen again and cause the same failure. Only when these two assumptions hold good that the recovery process is successful. The time line also shows an instance where the recovery mechanism fails to uphold failure transparency. The event "output a" happens during the initial run. But during recovery the event "output b" happens. This exposes the user to the fact that there was a failure of the application and that a recovery mechanism recovered the application. Also there is a sequence of events present and that do not appear consistent to the outside world. Hence the failure was not transparent to the user.

2.2 Fault Detection

Fault detection is the process of recognizing that a fault has occurred in the system. A lot of times the fault detection process does not detect the fault itself but detects the erroneous state caused by the fault. This is required before any fault recovery schemes can be implemented. Fault detection schemes also try to determine the location of the fault in the system to help in choosing the correct recovery method. An important property of fault detection is to detect the fault as soon as possible to isolate its effects and aid easy recovery. The property that a malfunctioning component halts before it saves the state of the error to permanent storage or transfers faulty state to other components is known as the halt-on-failure property[Schneider84] and forms the core of the fail-stop model.

All fault detection methods depend upon some form of redundancy to succeed. Redundancy allows us to check a result and reduce the probability of using an incorrect result. Without redundancy and checking of results, we have no way of deciding if a result is correct or not. Redundancy can be both physical and temporal. Physical redundancy is having multiple copies of the hardware, software or information exist simultaneously and comparing results with each other and therefore checking each other. Temporal redundancy is having the same computation being run at different points in time and comparing the results. Redundancy can also be either complete or partial. In partial redundancy only a part of the computation is redone and heuristics are used to give an answer for correctness. A few commonly used fault detection techniques and the ways in which they used redundancy are described below.

The following three techniques are based upon partial redundancy. Error detecting codes are implemented both in software and hardware.

Error detecting codes: They are based on the principle of information redundancy and are implemented both in software and hardware. They are formed by the addition of redundant information to data units or by mapping data unit words into new representations with redundant information. Some common examples of error detecting codes are parity codes, m-of-n codes and duplication codes. An application of parity codes is in memory modules which can detect parity errors in their data content and raise an exception. Checksums are another form of error detecting codes which are used on blocks of data being transferred from one point to another. They are used frequently in data transfers between mass storage devices and processors and in transfers over packet switched networks.

Self checking logic: Self checking logic has the ability to automatically detect the existence of a fault in hardware during normal operation. In general the logic presents a valid output when fault-free and generates an invalid output in the presence of a fault which can easily be recognized as faulty output. They are also designed to detect faults not only in the logic being monitored but also faults in the checking logic itself.

Software consistency checks: Consistency checks use knowledge about the characteristics of data to verify its correctness. An example is that the amount of cash withdrawn at an ATM should never exceed a certain amount. These types of checks are usually performed using assertion statements in software written in high level languages. Another example is the virtual memory system which knows that a program cannot access certain areas of its virtual memory space. Self-checking data structures are another example of consistency checks where operations on the data structure are automatically checked to ensure that they do not leave the data structure in an unstable state. This is especially important in dynamically allocated data structures.

The following two techniques are based upon complete redundancy.

Hardware redundancy: There are three forms of hardware redundancy - passive, active and hybrid. Passive hardware redundancy uses voting mechanisms to mask the occurrence of faults. This method does not truly detect the fault and requires no action from the system or an operator. The most common form of passive hardware redundancy is triple modular redundancy. The basic concept of TMR is to have three copies of the hardware and perform a majority vote to determine the output of the system. The failure of one copy of the hardware can be masked by the other two fault-free modules. Active hardware redundancy techniques try to locate and remove a faulty component after an error has been detected and enable a spare component to continue operation normally or at a degraded level. One form of active redundancy is the standby replacement technique. In standby sparing, one module is operational while one or more modules are used as spares or standbys. When a fault occurs in the operational module, one of the spares is made the operational module. Hybrid redundancy tries to combine the best features of the passive and active techniques. Fault masking is used to prevent erroneous results and fault detection and recovery are used to reconfigure the system.

N-version programming: N-version programming was designed to tolerate design and coding flaws in software[Avizienis85]. The basic concept is to design and code a software module N times and to vote the N results produced by the modules. It is hoped that by performing the N designs independently, the same mistakes will not be made in all the modules The voting module will be able to detect a fault because the same fault is not expected to occur in all the modules. In theory, this technique should be able to detect and recovery from most software faults. However it is very expensive to produce N independent designs and implementations of a software module. Moreover it is not clear if the n-modules will not have correlated faults.

2.3 Fault Recovery

Fault recovery is the process of getting the system back to an operational state after a fault has been detected or after a failure has occurred. The main goal to be achieved during recovery is failure transparency. There are two actions that a fault recovery mechanism has to perform to achieve failure transparency. The first one is that no visible events are output after recovery that are inconsistent with the events output before the failure. In other words the system should try to recover to a state as recent in time as possible to minimize the number of visible events it has to redo. The second one is correctness where the goal of the system is to recover to a state which is free of any errors caused by the fault or failure and also to recover to a state that will not lead to the same fault or failure to occur again.

All fault recovery mechanisms depend upon the state of the computation being saved in some form. Some recovery mechanisms actively save state to use it later for recovery and other mechanisms rely upon redundant state present in the system to recover from in the case of a failure. Systems can also use temporal redundancy to recompute state in the case of a failure. The state has to be saved on some form of storage that will survive the fault or a failure caused by the fault. Some common examples of such storage are hard disk drives and memory of backup or redundant machines that are not affected by the failure.

The following are examples of commonly used fault recovery mechanisms that explicitly save state during normal operation for use in the case of a failure.

Checkpoint and recover schemes: This scheme is based on temporal redundancy. The state of the system, checkpoint, is saved periodically during the failure-free operation of the system. After an error or a failure has occurred, the system rebuilds its state from a checkpoint and tries to continue execution. This scheme works on the optimism that the fault has been fixed or that the fault activation is transient or intermittent.

Recovery blocks: Recovery blocks[Randell75], used in software fault-tolerance, are a special case of checkpoint and recover schemes. N versions of a software block are provided and a set of acceptance tests are used. One version of the block is designated as the primary and the rest are used as spares or standbys'. The primary version is used until it fails the acceptance tests. Then the system rebuilds its state to a checkpoint taken before the beginning of the module. One of the spares is designated as the primary and the computation is redone using the new module.

Process pairs: Process pairs[Gray86] is a special case of recovery blocks. Process pairs comprise of a pair of processes, one designated as the primary and the other as standby. The primary periodically transfers its state to the backup. When the primary fails the backup continues execution from the last state it received from the primary. Process pairs are different from recovery blocks in the sense that the same code is executed but by a different process. This method works well with heisenbugs. A slight variation of this technique is used in the TARGON/32 system [Borg89] which executes the backup process on any one of the other processors in the system.

The following are examples of recovery mechanisms which use redundancy built into them to reconstruct the state of the computation after a failure.

Error correcting codes: They are similar to error correcting codes with enough information redundancy in them to both detect and correct the error. Some examples of error correcting codes are overlapping parity codes and Hamming error correcting codes.

Hardware redundancy: As described earlier in detection techniques, hardware redundancy is used to recover from component failures. The failed component is bypassed or masked and the system continues execution.

2.4 Software Faults

Faults may be classified into two categories: operational and design [Gray91]. Operational faults are caused by conditions such as wear-out and changes in the environment can be handled with simple replication. Faults caused by design bugs are much more difficult to handle, because simply replicating a buggy design often results in dependent failures in which all the replicas fail.

Software faults are all design bugs. They are caused due to programming errors, algorithmic errors, specification errors etc. So these faults are present in the software throughout its lifetime. In saying that a software fault occurred, we mean that the piece of code which is faulty was executed. We can also describe this other words as a software fault have been activated.

Software faults which are activated every time we run the software are usually detected during the testing phase and are corrected before the piece of software is released. Software faults which are rarely activated escape the test and debug process and end up in the production release of the piece of software. If the faults are not being activated every time we run the software, it is because that piece of code is executed when a certain external event occurs. This event can be termed as the "fault trigger". The fault trigger or the external event may happen very infrequently or can also be non-deterministic in its occurrence. An example of a rare event is the occurrence of a leap day (Feb. 29) in a year. An example of a non-deterministic event is a particular scheduling order which leads to a deadlock. During a different execution of the program the scheduling order may change and not cause the deadlock again. Faults caused by such rare or non-deterministic triggers escape the testing phase because we cannot simulate all the possible inputs to any substantial piece of software on a reasonable time. A subset of all the possible input space is chosen to test software and the subset is chosen is such a way so as to maximize the amount of code tested.

In the next few sections of this chapter we will discuss the two properties of software faults that affect recovery and see how various fault recovery methods make assumptions about these properties.

2.5 Fail-Stop Property

A faulty program may perform arbitrary tasks that make recovery complicated or impossible. For example, a malfunctioning program may overwrite critical state or send incorrect information to other processors. To ease the task of building fault-tolerant systems, most designers try to ensure that a malfunctioning program halts before it writes erroneous data to stable storage or sends incorrect information to other processes. This property is known as halt-on-failure [Schneider84] and forms the core of the fail-stop model. Many fault-tolerant systems assume that faulty application programs follow this model [Strom85, Johnson87, Birman91, Costa96].

There are a number of ways to achieve fail-stop failures in the presence of software faults. The key principle behind these methods is to stop executing the program before it commits any faulty state. By “committing faulty state”, we mean that the state is saved in such a manner that it survives the application crash and is part of the application state after recovery. There are two major factors which decide whether this principle is being upheld or not during a failure. The first one is the quality of error detection present in the program and the second one is has to do with the recovery method being used. Different recovery methods have different requirements about how frequently state needs to be saved and what part of the state of the process needs to be saved.

Figure 2.2 on page 18 shows examples of failures that uphold the fail-stop property

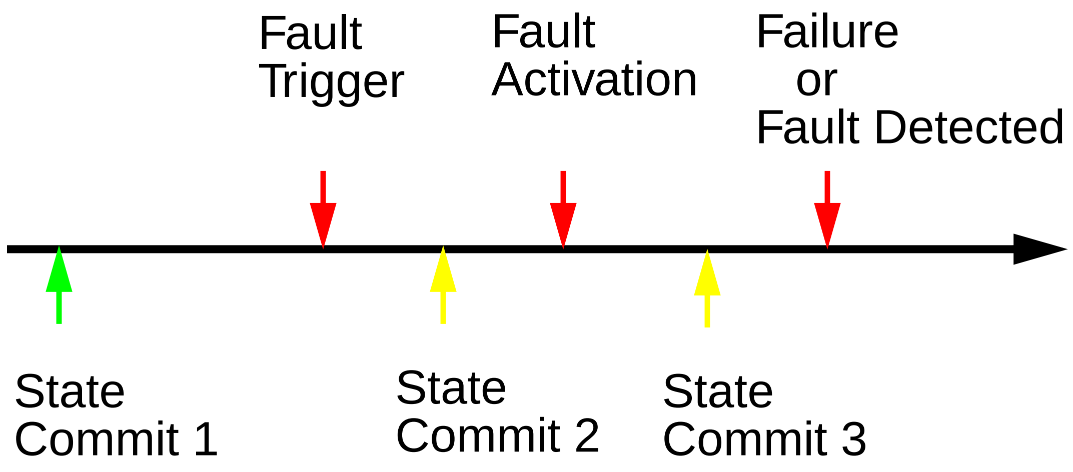

Figure 2.2 A time line showing events related to the fail-stop property

and examples of failures that potentially violate the fail-stop model. If the last state commit before the failure. happened at the point in time marked "State Commit 1" then the failure upholds the fail-stop model. This is because state was saved before the fault trigger occurred and so the committed state cannot contain any state corrupted by the fault or the state associated with the fault itself. On the other hand, if the last state commit before the failure, happened at the point in time marked "State Commit 2" then whether the failure violated the fail-stop property depends upon what was saved in the state commit. If the state commit saved state related to the trigger or recorded the occurrence of the trigger then the failure violates the fail-stop model. This is because when the recovery mechanism recovers the process, the state of the trigger will have survived the failure and will lead to the same fault being activated after recovery. If the committed state does not contain any information related to the trigger then the failure will have upheld the fail-stop property. Similarly, if the last state commit happened at the point in time marked "State Commit 3" then whether the failure violated the fail-stop property depends upon what was saved in the state commit. In addition to the state of the trigger, if the state commit saved any state changed or corrupted as a result of the fault, the failure will have violated the fail-stop model. If the state commit saved neither any state related to the trigger nor any state changed by the fault the failure upholds the fail-stop model.

2.6 Non-Determinism

A lot of recovery mechanisms rely on time redundancy to be able to successfully recover from a fault. For example, process pairs are similar to recovery blocks, but instead of retrying the operation on a different implementation of the code block, process pairs retry the operation on the same code (possibly on a different computer). Process pairs, and rollback-recovery protocols in general [Elnozahy99, Huang93], survive a specific class of software faults known as “Heisenbugs” [Gray86]. Heisenbugs are transient, non-deterministic faults that disappear when the operation is retried, even if the same code is used. The transient nature of the fault arises because some factor external to the program has changed; for example, a different interleaving order of threads may occur during retry and so avoid a race condition.

It has been hypothesized that most faults that occur in released applications are transient [Gray86]. The intuition for this hypothesis is that transient faults such as race conditions are more difficult to reproduce and hence debug than non-transient faults (so-called Bohrbugs), so transient faults are more likely to remain in released software. This is an important hypothesis for guiding research in how to recover from software faults.

2.7 Application-Specific and Generic Recovery Methods

We categorize techniques for recovering from software faults as either application-specific recovery or application-generic recovery. In application-specific recovery, a non-fault-tolerant design is made fault-tolerant by adding code that is specific to the application. This includes techniques where the programmer makes calls to fault-tolerance libraries or reconstructs part of the program state during recovery. An extreme example of application-specific recovery is N-version programming, which uses N independent implementations of the same program [Avizienis85]. Recovery blocks also use multiple implementations of a block of code, but use a passive-replication style with error-checking and rollback to avoid running multiple versions at the same time. Application-specific recovery can be very effective, but is often prohibitively expensive to implement.

Because of the cost of implementing application-specific recovery for each application, researchers have suggested various application-generic ways to survive software faults. These techniques do not add redundant, specific code for each application (we do not consider error checking code as redundant, though technically most error checks are application-specific and redundant). As a result, application-generic techniques need no information about the application, nor do they require any assistance from the application programmer in the form of extra code. For example, process pairs periodically transfer the state of the primary process to a backup process in a generic manner. They use operating system services to make a copy of the complete state of the application without any help from the application itself. When the primary process suffers a failure, the backup process takes over and executes the application. Again operating system services are used to periodically update the state of the backup process in an application independent manner and also to control the execution of the backup process. Note that a truly generic recovery mechanism must preserve all application state (e.g. by checkpointing or logging), because there is no application-specific code to reconstruct missing state. Hence only a change external to the application can allow the application to succeed on retry. Rollback-recovery protocols in general, save state periodically and rollback to the last saved state after a failure and try to recover from there.

CHAPTER 3

RE-EVALUATING THE FAIL-STOP ASSUMPTION

3.1 Introduction

This chapter evaluates the assumption that system designers make about the fail-stop property while building fault-tolerant systems. As mentioned in the last chapter, the fail-stop property ensures that a malfunctioning program halts before it writes erroneous data to stable storage or sends incorrect information to other processes. System designers assume that a faulty program does not violate this property during a failure. Interestingly, this assumption has never been comprehensively validated.

The two major factors which decide whether the fail-stop property is being upheld during a failure are the quality of error detection present in the program and the comprehensiveness and frequency of the state being saved.

All error detection methods depend upon some form of redundancy to succeed. N-version programming [Avizienis85] is a well known example of complete redundancy, which uses N independent versions of the same program. Ideally, these versions fail independently because they are written by different groups. In practice, these versions are correlated because independent groups make similar errors [Knight86, Brilliant90]. N-version programming is a powerful technique, but only a small subset of applications can afford the added expense of writing multiple versions of the same program.

Because N-version programming is prohibitively expensive, most applications use partial redundancy based on error checking. Instead of replicating a computation, the application checks the results of the computation based on a heuristic. For example, a program may verify a message's checksum after receiving the message. Assertion statements are a common form of error checking which are based on partial redundancy. They are cheap to add to the program but they can only detect the specific error for which they are designed. So there is the possibility of them not catching all the errors in the program. In the evaluation of the fail-stop property presented in this chapter, we are testing the effectiveness of partial redundancy in the form of error checking in halting the computation during a crash.

The second factor which affects the fail-stop property is the way state is saved by the application. We save state while the program is running so that a failure can be made transparent to the user. By making a program failure-transparent, we reduce the effects of the failure on the user. As an example, by periodically saving the file that is being edited, we can reduce the effects of a failure as the user can continue working from the last saved copy of the file. He does not have reissue all the editor commands and the input from the beginning. An additional benefit of saving state is the reduction in time needed to recover the application back to the point of failure. The frequency at which state is saved and the amount of state saved affect the fail-stop property of program and the ease with which the computation can be recovered. The next section discusses in detail the effects of saving state on the fail-stop property of software.

In this chapter, we address the question: "How valid is the fail-stop assumption for commonly used applications?". In addition we also answer the question: “How does the use of a generic recovery mechanism affect the fail-stop property?”.

3.2 Saving State and the Fail-Stop Property

We have not yet succeeded at writing fault-free programs. Therefore, we build recovery methods into software to shield the user from the effects of a failure and preserve the user’s data. This means that we have to save the state of the program periodically while it is running so that we can restart the program from the most recently saved state. The two factors we need to consider while saving state are the comprehensiveness of the state being saved and the frequency at which we save state. In deciding the degree of comprehensiveness and frequency to be used, there is a trade-off between failure transparency for the user and the amount of work to be done by the application programmer, versus the number of failures that are not fail-stop (fail-stop violations).

The comprehensiveness of the state saved determines how much work needs to be done to recover the process to a prior state. The more state we save, the less the amount of state that needs to be reconstructed during recovery. This makes the programmer’s job a little easier. It also means that the failure will be more transparent to the user. In other words, if we do not save the comprehensive state there are two possible effects, the first being reconstruction of more state during recovery and the second being loss of some state visible to the user. As an example, let’s look at how a word processor can save state in different ways. The word processor can periodically save the state of the file the user is working on. When a crash occurs the user can start a new instance of the word processor and load one of the automatically saved versions of the file to reconstruct the state up to the last time the file was saved. However, the user loses state like the position of the cursor, the contents of the clipboard buffers, undo buffers etc. The makes the user aware of the effects of the crash and some failure-transparency is lost. On the other hand, the application programmer can periodically save the entire state of the process including the cursor state, clipboard buffers etc. When the program crashes the user can recover the entire process to the state last saved by the application. In this case no state is lost, including any temporary state like cursor position, and the failure is transparent to the user.

The frequency at which state is saved determines how much work we lose or need to redo when the process is recovered. The more frequently we save state, the less the time between the last save and the crash. This means that very little work needs to be redone to recover to the point of failure. This also reduces the chance of needing to invalidate a visible event or needing to output a visible event twice. This becomes very important for interactive applications where losing work means that the user might have to redo some of the recent input or might be exposed to inconsistent sequences of output events. This is considered to be a highly undesirable side effect during recovery. So interactive applications generally try to save state after every user input event, so that the user never needs to repeat input in the event of a crash. Some interactive applications use a different strategy and save state before every visible event to maintain failure-transparency.

Comprehensiveness and frequency of saving state affect the fail-stop property in a straight-forward manner. The more comprehensive the state saved, the more the chance of a fail-stop violation. This is because the more state that is saved each time, the more likely that erroneous state or state related to the fault trigger will be included in that saved state.This means that after recovery the state of the activated fault will still be present in the process and hence fail-stop is violated. The greater the frequency at which state is saved, the greater the chance that state is saved after the fault has been activated and before the program is halted. Basically, more frequent saves gives less time for error detection mechanisms to detect the error and stop the program.The state saved after the fault has been activated may include the state of the activated fault and so violate the fail-stop property. So for preserving the fail-stop property it is desirable to minimize the amount of state saved and to minimize the frequency at which it is saved.

The decision about comprehensiveness and frequency of state saves is dependent upon the level at which state saves and recovery mechanisms are being implemented. By level, we mean whether the recovery process is handled by the application or by system level software transparent to the application. There are trade-offs in choosing at which level state should be saved and recovery handled after a crash.

General purpose recovery methods like checkpointing and logging address the development costs of application specific recovery. Having a general purpose recovery method for a platform means that all applications on that platform can use it. This reduces development costs as the recovery code needs to be written only once and can be better tested. It also means that the system can be more failure-transparent as the system software has control over application recovery. But, they suffer from weaknesses that are the strong points of application specific recovery. General purpose recovery methods have no information about the application. Their lack of knowledge about the application means that they cannot decide on which parts of the state need to be saved and also which visible events need to be committed. So they save all the state of the process and as frequently as needed to preserve failure-transparency. As discussed earlier this increases the possibility of the application violating the fail-stop property during a crash.

Application specific recovery can minimize the amount of state that needs to be saved and frequency at which it needs to be saved. This is possible because the application programmer has control over saving state and being the most knowledgeable about the application, he can minimize both the comprehensiveness and frequency of the state being saved. He can minimize comprehensiveness because he knows what state needs to be saved for recovery and knows what he can recompute during recovery from other state. Since the comprehensiveness of the state being saved has been minimized, saved state changes less frequently and so minimizes the frequency at which it has to be saved. The programmer also knows which outputs or visible events need to committed. In this way he can minimize the frequency of state saves. As discussed earlier, this would be ideal for upholding the fail-stop property during a crash. One of the disadvantages is that the burden of handling recovery is placed on the application programmer. The correctness of the recovery protocol is dependent on the programmer's implementation and the cooperation between multiple programmers working on the application. It is also harder to predict whether all the applications in a system will be able to recover from a total system crash because each application has a custom recovery protocol.

In the following sections of this chapter we measure the fraction of faults that violate of the fail-stop property in the presence of both application specific and general purpose recovery protocols.

3.3 Workload

We use three general purpose applications to conduct our experiments: vi, Postgres and oleo. vi is one of the earlier text editors for Unix and one of the commonly used editors. The version we use is nvi. Postgres is a database management system developed at U.C. Berkeley [Stonebraker87]. oleo is a terminal based spreadsheet program. While our results are specific to the software being tested, we believe that these programs exhibit most of the characteristics found in commonly used applications.

nvi is an interactive applications which reads in user input and displays all of its work on the screen. It saves output to a file when given a command by the user. We modified the application to simulate user input by reading characters from a text file. This allowed us to automate the thousands of experiments used in this paper. nvi consists of about 87,000 lines of code. Postgres can run in a variety of conditions ranging from accepting input directly from the user to running in a client-server configurations. We ran Postgres in the mode where it read SQL commands from a file. The SQL commands used to drive Postgres are based on TPC-B [TPC90]. Postgres consists of around 327,000 lines of code. oleo, like nvi, is also an interactive application which reads in user input and displays all its work on the terminal screen. It saves the state of the spreadsheet to a file when given a command by the user. We modified oleo to read its input from a file. oleo consists of about 53,000 lines of code.

We make these software systems recoverable by linking them with Discount Checking, a generic checkpointing library developed by our David Lowell and others at the University of Michigan [Lowell99]. Discount Checking is general enough to save the state of a wide range of applications, even those with plentiful kernel state. nvi and Postgres also have application specific recovery mechanisms built into them. nvi uses temporary files to periodically save the state of the file being edited. It uses these recovery files to recover the state of the file being edited after a crash. Postgres, like most databases, uses a transactional mechanism [Gray93] and logs to save state and recover in the event of a crash. oleo has no recovery mechanism built into it. It relies upon the user to save state and recover from it in the case of a failure.

3.4 Fault Model

Faults injected into the applications form the second part of the workload. Our primary goal in designing these faults is to generate a wide variety of software crashes. Our models are derived from studies of commercial databases and operating systems [Sullivan92, Sullivan91, Lee93] and from prior models used in fault-injection studies [Barton90, Kao93, Kanawati95, Chen96, Ng97]. The faults we inject range from low-level faults such as flipping bits in memory to high-level software faults such as memory management errors and uninitialized variables. We classify injected faults into two categories: bit flips and high-level software faults.

The first category of faults flips random bits in the database's address space [Barton90, Kanawati95]. We target the heap and stack areas of the database's address space. These faults are easy to inject, and they cause a variety of different crashes. It is difficult to relate a bit flip with a specific error in programming but they simulate the corruption of a process's address space by unknown errors and external agents.

The second category of faults introduce programming errors either into the source code of the application or dynamically into the running program. We inject memory management faults dynamically into the running program. The remaining 5 types of faults are injected into the source code of the application using a custom built parser. The parser identifies all potential pieces of code where a particular fault can be injected. That piece of code is replaced by code which injects the fault on demand A brief description of the injected faults and example code for each of the fault types follows in the next few paragraphs.

Memory management faults: These faults affect the memory dynamically allocated and deallocated by a process as per its run-time requirements.This memory is part of the heap segment of a process. In a program written in the programming language "C", the function malloc() is used to allocate memory on the heap and the function free() is used to deallocate the memory. Memory management faults free memory prematurely while the particular block of memory is still in use. The malloc() and free() routine were modified to inject the fault. The routines keep track of allocated memory that has not yet been freed. When the fault injection routine is executed at a random point in the execution time of the workload, it prematurely frees one of the allocated blocks of memory still in use.

Off by One faults: These faults simulate software bugs where the programmer uses the wrong operator for the conditional operator which determines if a loop will execute another iteration or terminate. The operator “>” is replaced by “>=” and “<” is replaced by “<=” and vice versa. In all the following code samples the string "nnn" represents an integer value which identifies the specific fault. This is used to selectively turn on a specific software fault. The variable "errornumber" indicates which specific fault to activate and is set by the fault injection routine using random number generators.

original code:

```
      for (i=0; i < 5;i++)
      {
         loop body
      }
```

faulty code:

```
      for (i=0; fault_slt(i,5,nnn);i++)
      {
```

```
   loop body
}
int fault_slt (int lside,int rside,int flag)
{
   if (errorflag == 1 && flag == errornumber)
      return(lside <= rside);
   else
      return(lside < rside);
}
```

Initialization faults: These faults simulate software bugs where the programmer fails to initialize a variable. The value with which the variable is being initialized is substituted by a zero.

original code:

```
j = 10;
```

faulty code:.

```
j = fault_init(10,nnn);
int fault_init(int value,int flag)
{
   if (errorflag == 1 && flag == errornumber)
   {
      fprintf(stderr,"Fault_init active %d\n",errornumber);
      return 0;
   }
   else
      return value;
}
```

Incorrect Branch faults: These faults simulate software bugs where the programmer declares a conditional branch statement instead of a iteration statement. The keyword "while" is substituted by a "if" statement.

original code:

```
while (i + j < len)
{
   loop body
}
```

faulty code:

```
if (errorflag == 1 && errornumber == nnn)
{
   if (i + j < len)
   {
      loop body
   }
}
else
{
   while (i + j < len)
   {
      loop body
   }
}
```

Delete Instruction faults: These faults simulate software bugs where the programmer forgets a simple expression statement. A simple expression statement is deleted.

original code:

```
keyname [ 1 ] = codes [ n ] ;
```

faulty code:

```
if (errorflag == 1 && errornumber == nnn)
 {
 }
 else
 {
  keyname [ 1 ] = codes [ n ] ;
 }
```

Change Destination Variable faults: These faults simulate software bugs where the programmer assigns a value to the wrong variable. The destination variable in an assignment statement is substituted by another variable.

original code:

```
i = k * numcols;
```

faulty code:

```
fault_assign(&i, &j, k * numcols, "=", nnn);
int fault_assign(int *orig,int *sub,int value,char *opr,
                   int flag)
{
      if (errorflag == 1 && flag == errornumber)
      {
         switch(opr[0])
         {
            case ’=’:
               *sub = value; /* j = k * numcols; */
               break;
         }
      }
      else
      {
         switch(opr[0])
         {
            case ’=’:
               *orig = value; /* i = k * numcols; */
               break;
         }
      }
   }
```

3.5 Data Collection

The faults are activated using the alarm signal generated by a timer. The timer is set to expire at the end of a random interval during the execution of the program. The signal handler for the alarm signal calls the fault injection routine. The routine randomly flips a bit in the stack and heap areas of the process or activates one of the faults injected into the source code.

At the end of the run, we check if the fault caused the program to execute erroneously. There are three possible outcomes for such a run:

- The program completes execution normally with the correct output.
- The program completes execution but with incorrect output.
- The program terminates prematurely with an error condition.

The first outcome means that the injected fault did not affect the program and so we do not consider these runs in our results. Runs with the second and third outcomes are what we consider as crashed runs. These are the runs we check for fail-stop violations. If a run completes with the second outcome, it means that error checking failed and the injected fault changed the output of the program. Since the program ran to completion, neither application specific recovery nor general purpose recovery methods will be able to recover the program and eliminate the fault. Hence these runs are considered to have violated the fail-stop model in the presence of both recovery mechanisms. Runs which end with the third outcome can be recovered using either application specific or general purpose mechanisms. The success of the recovery depends upon whether the run violated the fail-stop property while committing state as required by the recovery mechanism. The methodology used by us to decide if any of these runs violated the fail-stop property is explained in the rest of this section.

We had discussed earlier in this chapter about how comprehensiveness and frequency of saving state affect the fail-stop property. We had also discussed how these factors are affected by whether the recovery mechanism is implemented within the application or is implemented below the application in a application-independent fashion. For this study we used Discount Checking as the generic recovery mechanism. We can vary the comprehensiveness and frequency of saving state by using the two different mechanisms and varying the strategy for saving state. We used two levels of comprehensiveness and two levels of frequency to arrive at four different strategies for saving state and recovering from a fault. The four strategies for saving state and recovering from failures are described below and the way we measure fail-stop violations for each them is also explained.

Less Comprehensive / Less Frequent (LC/LF): The application-specific recovery mechanism is built by the application programmer who has full knowledge about the software. Hence this mechanism will save the minimum amount of state (less comprehensive) at the appropriate time (less frequent) to be able to recover the process after a failure. In the case of nvi, this is in the form of an auto-save mechanism and the save commands issued by the user. In the case of Postgres state is committed at the end of every transaction. In the case of oleo this is left to the user to control when to save state by issuing save commands. When a save command is issued, oleo saves the current state of the spreadsheet.

We measure the number of fail-stop violations by first generating the complete set of states, that the output files of the application can be in, during a normal or reference run. The output files of an application include all files on a disk that the applications writes to. In the case of nvi these files would be the file the user is editing, temporary files used to save state for recovery, etc. For each of the crashed runs, we compare the state of the output files at the end of the crashed run with the complete set of states from the reference run. If the state of the crashed run matches any one of the states of the reference run it means that if we let the application recover using its built-in recovery mechanism, it will recover and run to completion provided the user continues the input from the point where the application recovers to. If the state of the crashed run does not match any of the states of the reference run, then that crashed run is considered to have violated the fail-stop property. This follows from the fact that the injected fault caused some changes in the application and these changes were committed in such a way that they will survive the crash and be visible to the application after recovery.

In the case of Postgres which uses a transaction based recovery method, we do not use the raw disk files to compare state. We apply transactional recovery methods supplied by the application to obtain the state of the database and use it to compare state and measure fail-stop violations.

Less Comprehensive / Frequent (LC/F): This state saving strategy modifies the application specific mechanism to take checkpoints more frequently to provide the better failure transparency. State is saved just before every visible event. For the applications we use in our workload this means any write to the computer screen. We do this by calling the state saving routine built into nvi, every time it writes something to the screen. In this way we maintain the comprehensiveness of the saved state at the same level as the previous scheme but increase the frequency of saving state. This strategy is however hard to implement for all applications with out modifying or perturbing the way the application normally runs.

We measure the number of fail-stop violations for this strategy in exactly the same manner as the previous one (LC/LF).

Comprehensive / Less Frequent (C/LF): This strategy uses Discount Checking to save the complete state of the process as a checkpoint. The saved state is the most comprehensive state that can be saved and includes state of the process present in the processor and also the state normally present in the operating system. To save state at the same frequency as the application-specific mechanism we configure Discount Checking to save state whenever the application writes to the disk.

To measure the number of fail-stop violations, we recover the run using the last checkpoint saved by Discount Checking after a crash or a failure. During this recovered run, we deactivate the fault injection mechanism so that neither the fault is reinjected during recovery nor are any new faults injected. If the run does not finish with normal output, then that crashed run is considered to have violated the fail-stop property. The fail-stop property has been violated because the fault caused a change in the state of the process which leads to the crash and Discount Checking committed this changed or faulty state. Upon recovery, the faulty state is preserved or visible to the application and leads to a crash again. Since we only check if the application runs to completion and has the correct output, we might miss cases of fail-stop violations where the application recovered with the faulty state and still managed to finish because it overwrote the state later. This case still violates the fail-stop model and so our results looking at the number of fail-stop violations are probably on the lower side.

Comprehensive / Frequent (C/F): This strategy uses Discount Checking's default state saving strategy. Discount checking tries to achieve maximum failure transparency by saving state comprehensively and before every visible event.

We measure the number of fail-stop violations for this strategy in the same manner as we do for the previous scheme (C/LF) which also uses Discount Checking.

Ideally we would like to apply each of the state saving strategies to a particular run and check if there is a fail-stop violation. However our use of a timer signal to inject the fault introduces non-determinism into when the fault is injected in the run. Hence it is not possible to have four runs for the four strategies where the fault is injected in the same way each time. This raises the question of whether we can apply all the four strategies to a single run and check for fail-stop violations. However the interactions between the application-specific recovery mechanism and the generic recovery mechanism allow us to apply only three of the strategies simultaneously. The LC/F uses the application-specific mechanism to save state at a higher frequency than it normally does. The application files this mechanism writes to are however visible to the generic recovery mechanism and the state of the files is committed by it. So we cannot at the same time have the generic recovery mechanism commit state which has the state of the files if they were less frequently committed. The strategies *Less Comprehensive / Frequent* (LC/F) and *Comprehensive / Less Frequent* (C/LF) are mutually exclusive. So we measured the number of fail-stop violations for the LC/LF, the C/LF and the C/F strategies. In the case of nvi we also measured the number of fail-stop violations in a different set of runs for the LC/LF, the LC/F and the C/F strategies.

3.6 Results

The results presented here were collected over several machine-months of fault-injection experiments. As mentioned before, we only considered runs which corrupted the output or terminated prematurely. About 3-4% of all the runs into which we injected faults met our criteria. We repeated the fault-injection experiments till we obtained 50 candidate runs for each fault per application. This gave us a total of 400 runs for each of the three applications.

Table 3.1 on page 39 shows the number of fail-stop violations for three of the strategies used to save state. As discussed in the earlier section we could only apply three of the four strategies at one time for a single run. So we present results for the LC/LF, C/LF and the C/F strategies in this table. In the case of nvi, we also applied the LC/F strategy instead of the C/LF strategy in another set of runs. These results are presented in Table 3.2 on page 40. Since the two tables show results from two different set of runs, the common columns show some variation in the number of fail-stop violations detected. This is a

Fault	Crashes	Fail-stop Violations LC/LF	Fail-stop violations C/LF	Fail-stop Violations C/F	Undetected Errors
Stack	50	0	0	0	0
Alloc	50	24	40	50	0
Heap	50	14	20	43	8
Off by One	50	18	19	21	12
Init Errors	50	0	2	2	0
Delete Branch	50	33	35	42	8
Delete Inst	50	15	17	27	3
Change Dest Var	50	6	10	13	5
Total	**400**	**110 (27%)**	**143 (36%)**	**198 (49%)**	**36 (9%)**

Table 3.1: Fail-Stop violations for nvi - 1
The column LC/LF gives the number of fail-stop violations for the application specific state save and recover strategy. The column C/LF gives the number of fail-stop violations for the Discount Checking strategy where comprehensive but less frequent checkpoints are taken. The column C/F gives the number of fail-stop violations for the default Discount Checking strategy which saves comprehensive state very frequently. The column Undetected Errors gives the number of cases where the fault was never detected and the run ran to completion with faulty results. These errors are included in all the other three columns.

result of the randomness in when the faults are injected and where in the code the faults are injected.

The results show that a significant fraction of the runs (25%) violate the fail-stop model when using the application's default mechanism (LC/LF) to save state. They also show that when a generic-recovery scheme such as checkpointing (C/F) is used the percentage of fail-stop violations increases to about 50%. This is a disturbingly high number and shows that a generic recovery scheme aiming to be failure-transparent will actually fail to recover a crashed application half the time. The other two columns (LC/LF & C/F)

Fault	**Crashes**	**Fail-stop Violations LC/LF**	**Fail-stop violations LC/F**	**Fail-stop Violations C/F**	**Undetected Errors**
Stack	50	0	0	1	0
Alloc	50	22	24	50	1
Heap	50	8	8	40	6
Off by One	50	11	21	29	5
Init Errors	50	0	0	1	0
Delete Branch	50	30	35	37	8
Delete Inst	50	10	14	25	4
Change Dest Var	50	13	17	16	7
Total	**400**	**94 (23%)**	**119 (30%)**	**199 (50%)**	**31 (8%)**

Table 3.2: Fail-stop violations for nvi - 2
The column LC/LF gives the number of fail-stop violations for the application specific state save and recover strategy. The column LC/F gives the number of fail-stop violations for the application specific strategy where the frequency of state saves is increased. The column C/F gives the number of fail-stop violations for the default Discount Checking strategy which saves comprehensive state very frequently. The column Undetected Errors gives the number of cases where the fault was never detected and the run ran to completion with faulty results. These errors are included in all the other three columns.

show that by reducing either the frequency of the state saves or the comprehensiveness of the saved state, the number of fail-stop violations can be reduced. However, the results do not give a clear indication of which of the two factors cause more fail-stop violations.

Table 3.3 on page 41 shows the number of fail-stop violations for Postgres. We present results for Postgres for only three of the strategies discussed in the previous section. Postgres, being a database application has a very specific strategy to save state at specific times based on transactional requirements. Therefore, we believe that results for the

Fault	**Crashes**	**Fail-stop Violations LC/LF**	**Fail-stop violations C/LF**	**Fail-stop Violations C/F**	**Undetected Errors**
Stack	50	1	17	18	1
Alloc	50	0	22	24	0
Heap	50	2	2	46	2
Off by One	50	8	8	8	8
Init Errors	50	2	4	5	2
Delete Branch	50	6	6	44	6
Delete Inst	50	6	7	11	5
Change Dest Var	50	2	2	3	0
Total	**400**	**27 (7%)**	**68 (17%)**	**159 (40%)**	**24 (6%)**

Table 3.3: Fail-stop violations for Postgres
The column LC/LF gives the number of fail-stop violations for the application specific state save and recover strategy. The column C/LF gives the number of fail-stop violations for the Discount Checking strategy where comprehensive but less frequent checkpoints are taken. The column C/F gives the number of fail-stop violations for the default Discount Checking strategy which saves comprehensive state very frequently. The column Undetected Errors gives the number of cases where the fault was never detected and the run ran to completion with faulty results. These errors are included in all the other three columns.

LC/F strategy which increase the frequency of state saves do not make any sense for this application.

The results for Postgres show that the fraction of fail-stop violations (7%) is much smaller than what was observed for nvi. Postgres being a database application has a lot of error detection built into it and also has a very good recovery mechanism in the form of transactions. The fraction of fail-stop violations is still significant for a database application which might be used to store critical data. The numbers for checkpointing (C/F) show a huge increase in the number of fail-stop violations. This number is a little less than what

.

Fault	Crashes	Fail-stop Violations LC/LF	Fail-stop violations C/LF	Fail-stop Violations C/F	Undetecte d Errors
Stack	50	0	0	3	0
Alloc	50	9	11	43	9
Heap	50	19	19	31	19
Off by One	50	7	7	17	7
Init Errors	50	8	11	23	8
Delete Branch	50	7	7	26	7
Delete Inst	50	18	20	27	18
Change Dest Var	50	23	23	25	20
Total	**400**	**91 (23%)**	**98 (24%)**	**195 (49%)**	**88 (22%)**

Table 3.4: Fail-stop violations for oleo
The column LC/LF gives the number of fail-stop violations for the application specific state save and recover strategy. The column C/LF gives the number of fail-stop violations for the Discount Checking strategy where comprehensive but less frequent checkpoints are taken. The column C/F gives the number of fail-stop violations for the default Discount Checking strategy which saves comprehensive state very frequently. The column Undetected Errors gives the number of cases where the fault was never detected and the run ran to completion with faulty results. These errors are included in all the other three columns.

was observed for nvi. This is due to the presence of better error detection mechanisms in Postgres as compared to nvi. The column for C/LF shows that the number of fail-stop violations are closer to LC/LF than C/F. This is because the frequency of saves in Postgres is very small and so in the event of a failure there is a high probability that the fault was triggered after the last state commit and so comprehensiveness will not have a large effect on successful recovery.

Table 3.4 on page 42 shows the number of fail-stop violations for oleo for the three strategies used to save state. We could not use the fourth strategy (LC/F) to save state for oleo without modifying the application significantly

The results for oleo show that a significant fraction of the runs (23%) violate the fail-stop model for the LC/LF strategy. It should be noted that oleo does not have a recovery mechanism built into it. So the state commits were done whenever the user gave a save command to the application which was four times in the input sequence we used. They also show that when a generic-recovery scheme such as checkpointing (C/F) is used the percentage of fail-stop violations increases to about 50%. This is a high number and shows that a generic recovery scheme aiming to be failure-transparent will actually fail to recover a crashed application half the time. The column for C/LF shows that the number of fail-stop violations are closer to LC/LF than C/F. Again as in Postgres, this is because the frequency of saves in oleo was very small and so in the event of a failure there is a high probability that the fault was triggered after the last state commit and so comprehensiveness will not have a large effect on successful recovery. oleo exhibits a very high number of undetected errors. Almost all the fail-stop violations exhibited by application-specific recovery are due to undetected errors. This is because oleo has very little error detection built into it.

3.7 Discussion of Injecting Faults at OS Level

The results presented in the previous section reflect the number of fail-stop violations when faults are injected into the application. We found that generic recovery mechanisms violate the fail-stop property for a large fraction of the faults. We know that generic recovery mechanisms were traditionally designed to recover from transient hardware faults. So we decided to look at how well generic recovery mechanisms work when faults happen below the application and recovery layers at the operating system level. We would expect generic recovery mechanisms to work well for faults occurring at the system level.

We used the fault injector developed by Weeteck Ng [Ng99] to inject faults into the operating system. We injected the same types of faults as that were injected into the application for the results presented in the previous section.

At the end of the run, we check if the fault caused the program or the operating system to execute erroneously. As discussed in the section “Data Collection” on page 33, there are three similar outcomes for each run:

- The program completes execution normally with the correct output and the operating system does not exhibit any errors.
- The program completes execution but with incorrect output.
- The program or the operating system terminates prematurely with an error condition.

We consider all runs with outcomes 2 and 3 as crashed runs and measure the number of fail-stop violations present in these runs.

3.8 Results

The results presented here were collected over several machine-months of fault-injection experiments. As mentioned before, we only considered runs which corrupted the output or terminated prematurely. About 10% of all the runs into which we injected faults met our criteria. We repeated the fault-injection experiments till we obtained 50 candidate

Fault	**Crashes**	**Fail-stop Violations LC/LF**	**Fail-stop violations C/LF**	**Fail-stop Violations C/F**	**Undetected Errors**
Stack	50	0	1	6	0
Alloc	50	1	5	19	0
Heap	50	2	3	4	0
Off by One	50	0	6	11	0
Init Errors	50	4	3	9	1
Delete Branch	50	1	2	12	0
Delete Inst	50	0	1	6	0
Change Dest Var	50	2	0	5	0
Total	**400**	**10 (2%)**	**21 (5%)**	**72 (18%)**	**1 (0%)**

Table 3.5: Fail-stop violations for nvi - Faults in the OS
The column LC/LF gives the number of fail-stop violations for the application specific state save and recover strategy. The column LC/F gives the number of fail-stop violations for the application specific strategy where the frequency of state saves is increased. The column C/F gives the number of fail-stop violations for the default Discount Checking strategy which saves comprehensive state very frequently. The column Undetected Errors gives the number of cases where the fault was never detected and the run ran to completion with faulty results. These errors are included in all the other three columns.

runs for each fault per application. This gave us a total of 400 runs for each of the three applications.

Table 3.5 on page 45 shows the number of fail-stop violations for nvi when faults are injected into the operating system. We notice that the fraction of fail-stop violations in all the three columns has gone down substantially from what was observed in the previous results section where the faults were injected into the application. About 2% of the faulty runs violated the fail-stop model for application specific recovery and 18% of the runs violated the fail-stop model for generic recovery. This is because for a fault injected in the

.

Fault	Crashes	Fail-stop Violations LC/LF	Fail-stop violations C/LF	Fail-stop Violations C/F	Undetected Errors
Stack	50	0	5	5	0
Alloc	50	NA	NA	NA	NA
Heap	50	1	3	3	0
Off by One	50	0	0	0	0
Init Errors	50	0	0	0	0
Delete Branch	50	1	2	2	0
Delete Inst	50	0	1	2	0
Change Dest Var	50	1	1	1	1
Total	**400**	**3 (1%)**	**12 (3%)**	**13 (3%)**	**1 (0%)**

Table 3.6: Fail-stop violations for Postgres - Faults in the OS
The column LC/LF gives the number of fail-stop violations for the application specific state save and recover strategy. The column LC/F gives the number of fail-stop violations for the application specific strategy where the frequency of state saves is increased. The column C/F gives the number of fail-stop violations for the default Discount Checking strategy which saves comprehensive state very frequently. The column Undetected Errors gives the number of cases where the fault was never detected and the run ran to completion with faulty results. These errors are included in all the other three columns.

operating system, to affect the fail-stop nature of the application, it has to corrupt the state of the application. The operating system primarily interacts with the application state through a well defined interface implemented as system calls. System call interfaces are well tested and also have a lot of error detection built into them. This is the reason we see fewer fail-stop violations when faults are injected in to the operating system.

Table 3.6 on page 46 shows the results for Postgres when faults are injected into the operating system. Again we notice that the number of fail-stop violations have decreased from what was observed when faults are injected into the application itself.

About 1% of the runs exhibit fail-stop violations for application-specific recovery and 3% of the runs violate the fail-stop property for generic recovery. As explained before, Postgres shows fewer violations than nvi because of the presence of better error detection. It was also observed that Postgres makes fewer system calls than nvi does and so this reduces the probability that a fault in the operating system can affect the state of the application. nvi makes 147,155 system call during each run and Postgres makes 64619 during each run for our workload.

Table 3.7 on page 48 shows the results for oleo when faults are injected into the operating system. Again we notice that the number of fail-stop violations have decreased from what was observed when faults are injected into the application itself. About 4% of the runs exhibit fail-stop violations for application-specific recovery and 3% of the runs violate the fail-stop property for generic recovery. In oleo’s case application-specific recovery causes more fail-stop violations than generic recovery. We had mentioned earlier that our numbers for generic recovery are on the lower side because of the way we measure the number of fail-stop violations for generic recovery. If the injected faults only affect the code which writes the spreadsheet to a file, then these faults will be recovered by generic recovery. The corrupted file will be overwritten with correct data when the file is written to later. This is a fail-stop violation for generic recovery too but is not reflected in the above table. Further, oleo uses the operating system functionality a lot less than nvi does. This is reflected in the overall reduction in the number of fail-stop violations compared to nvi.

.

Fault	Crashes	Fail-stop Violations LC/LF	Fail-stop violations C/LF	Fail-stop Violations C/F	Undetected Errors
Stack	50	4	0	3	0
Alloc	50	0	0	0	0
Heap	50	1	1	1	0
Off by One	50	3	0	0	0
Init Errors	50	0	1	1	0
Delete Branch	50	1	3	4	0
Delete Inst	50	5	0	1	0
Change Dest Var	50	3	4	4	0
Total	**400**	**17 (4%)**	**9 (2%)**	**14 (3%)**	**0 (0%)**

Table 3.7: Fail-stop violations for oleo - Faults in the OS
The column LC/LF gives the number of fail-stop violations for the application specific state save and recover strategy. The column LC/F gives the number of fail-stop violations for the application specific strategy where the frequency of state saves is increased. The column C/F gives the number of fail-stop violations for the default Discount Checking strategy which saves comprehensive state very frequently. The column Undetected Errors gives the number of cases where the fault was never detected and the run ran to completion with faulty results. These errors are included in all the other three columns.

3.9 Related Work

Many prior studies have used software fault injection. Hsueh et.al, [Hsueh97] provide an excellent introduction to the overall area and a summary of past fault injection techniques. Most fault injection studies focus on the final effect of the error (e.g. fault latency). Very few studies seek to understand and measure if the faults violate the fail-stop model. We know of no prior study which looks at fail-stop violations from the point of recovery after a failure.

Mark Sullivan and Ram Chillarege looked at the field failures reported for the MVS operating system [Sullivan91] and the DB2 and IMS database systems [Sullivan92]. They classified the errors into types of programming mistakes that caused them. The fault model we use in this fail-stop study is largely derived from their work.

A prior paper from the Rio project measures corruption in the Postgres database system by running a pre-determined workload, monitoring the progress of the workload, and comparing the results after a crash with the pre-determined, correct answers at the point of the crash [Ng97]. This study shows that transactions can be used to recover from software failures with very few fail-stop violations as our results point out too. This study however does not look at other kinds of applications like interactive general-purpose applications and also does not look at the implications of using generic -recovery.

A couple of studies have looked at the effectiveness of n-version programming. Knight et.al, [Knight86] evaluated the assumption made in n-version programming that programs developed independently will fail independently. They concluded that the assumption of independence of faults in N-version programming does not hold. Brilliant et.al, [Brilliant90] analyzed the faults found in an N-version software experiment to find the cause of correlated failures. They concluded that there do not seem to exist any simple methods to reduce correlated failures in N-version programming. They found that the failures were not caused by the use of a specific tool or programming language, and even the use of diverse algorithms did not eliminate correlated faults. Both the studies evaluated only the error detection capabilities of N-version programming. They did not evaluate whether N-version programming can detect errors early enough to prevent fail-stop violations.

A study done on the reliability of the IBM MVS/XA [Mourad85] operating system looked at effectiveness of recovery routines. They found that recovery routines could recover the system from 88%-92% of the failures. These recovery routines were specifically written for the operating system and the operating system software was also better tested than general purpose applications. These results are comparable to what we obtained for Postgres.

A study done by Madeira et. al, [Madeira94] at the University of Coimbra looked at the effectiveness of error masking in the presence of hardware faults at the pin level. The results they obtained showed that 46% of the faults violated the fail-silent model. In the presence of error detection about 0.4% to 2.3% of the faults violated the fail-silent model. They also found the number of fail-stop violations is very dependent on the target system and the workload being run. If we consider the 46% fail-silent violations in the absence of error-detection as equivalent to the crashed runs we considered in our study then the about 1%-5% (0.4-2.3/46) violated the fail-silent model in the presence of error-detection. This result is comparable to what we obtained when faults were injected in the operating system instead of the application software itself. We do this normalization because our study always had error detection turned on in the workload. A later study [Rela96] looked at the effectiveness of software consistency checks in the presence of hardware faults at the pin level. Their results show that software consistency checks can prevent 40% of the fail-stop violations that escape simple error detection mechanisms like memory protection. They also found that using techniques like software based control flow checking could prevent another 40% of the fail-stop violations that escaped the simple error detection schemes. These results show that simple consistency checks as used by many software applications are not enough to prevent fail-stop violations. It should be noted that both these studies looking at hardware faults. Our study looks at software faults which cause a majority of the failures. Further these studies do not look at recovery mechanisms.

Work also done at the University of Coimbra [Costa00] measures corruption in the Oracle database system by injecting faults into the database system and monitoring it for any inconsistencies in the database. Their fault model is very similar to what we use, although they use object code modification [Carreira98] to inject faults into the application instead of source code level injection as we do. The results from this study confirm the fact that software faults are the major cause of failures in applications. They found that about 1% of the injected faults caused data corruption. However, if only the runs in which the faults were activated are considered, 5% of the injected faults caused data corruption. This number is comparable to what we observed for the Postgres database system (7%) in our study. This study is limited to looking at failures for a single class of software and also does not look at other issues like generic recovery and faults in software outside the application.

3.10 Conclusion

Computer users experience software faults very commonly. Moreover a majority of computer failures are due to software faults. Recovery mechanisms designed to make the failures transparent to users assume that the failures are fail-stop. Our goal in this chapter is to evaluate empirically how well software faults follow the fail-stop model.

Our study has shown that a significant number of application failures violate the fail-stop model. Some applications like Postgres, a database application, have very good error detection and recovery mechanisms built into them. These applications in general cause few fail-stop violations. Our study also shows that generic recovery mechanisms are worse than application specific recovery mechanisms.

However, this does not mean that generic recovery mechanisms have no role in recovering applications from software failures. Generic recovery mechanisms were originally designed to handle hardware failures which occur outside the application software. Our study shows that for faults in the operating system, generic recovery mechanisms are acceptable to recover from failures.

Our hope is that this work will encourage the design of better fault detection and recovery mechanisms. Recovery mechanisms instead of assuming that failures will be fail-stop should try to handle failures which may not be fail-stop. Recovery mechanisms that offer standard programming constructs which can be used by the application programmer to develop application-specific recovery mechanisms will help in better recovery from failures. We also hope that this work will lead to the design of better fault detection mechanisms which can detect an error and stop the application before it commits any faulty state.

CHAPTER 4

RE-EVALUATING THE NON-DETERMINISTIC NATURE OF SOFTWARE FAULTS

4.1 Introduction

We evaluated the fail-stop property of software in the last chapter. Upholding the fail-stop property ensures that we did not commit the state of the fault before the failure and that we have clean state to use for recovery. However there is another important property of software that influences the application's ability to execute beyond the point of failure. This property describes whether the same fault will be activated and lead to the same failure again. In other words, this property describes whether the fault is triggered in a deterministic manner or in a transient manner.

The evaluation of this property requires us to analyze the faults that occur in production software in the field. This means we need access to detailed information about the faults, their triggers and the way they were fixed. Unfortunately, most production software in the past was proprietary and therefore hard to analyze. In particular, most companies were understandably reluctant to release information on the exact failings of their software.

However, the recent trend toward open-source software may make this information available. There are now widely used, open-source programs of all types, including operating systems, databases, web servers, web browsers, word processors, spreadsheets, and a host of others. Open-source programs share a number of important characteristics. First, they are widely used and hence form a relevant base of software to study. For example, the Apache web server is used by 54% of all web sites [Netcraft99]. Second, their development process is open. Open development allows others to peruse the history of bugs and software revisions, information which is normally not public. Last, while there is little hard data comparing the quality of open-source with proprietary software, the rapid increase in the use of open source software like Apache and Linux indicates that the quality of open-source software is good enough for mainstream use. Both open-source and proprietary software are released quickly and with plentiful bugs, perhaps as a result of the current pace of innovation in the computer industry. We believe this trend toward open-source, open-development software presents a treasure-trove of information for research in software reliability.

In this chapter, we use information in bug reports and source code to understand and classify faults that occur in three widely used, open-source programs. The goal of this study is to provide information to guide research on how to survive application faults. In particular, we wish to test the hypothesis that generic recovery techniques, such as process pairs, can survive a majority of application faults. Our methodology differs from that of prior studies [Lee93]. We reason from bug reports and source code as to whether a purely generic recovery system would have recovered from application faults, while past studies examine the field behavior of implemented, mostly generic recovery systems. This comparison is valuable because of our focus on purely generic recovery and because there is no data available on the effectiveness of recovery on widely used, open-software systems.

4.2 Recovery from Software Faults

Faults may be classified into two categories: operational and design [Gray91]. Operational faults are caused by conditions such as wear-out and can be handled with simple replication. Faults caused by design bugs are much more difficult to handle, because simply replicating a buggy design often results in dependent failures in which all the replicas fail. Software faults are difficult to survive because they are all caused by design bugs.

Faults may occur in application or system software. In this chapter, application software refers to any software that runs on top of the recovery system, that is, software for which the recovery system is responsible for recovering. System software refers to software that runs below the recovery system, that is, software that recovers itself. The scope of this study is limited to faults that occur in application software and how often a recovery system can recover from these faults.

Generic recovery protocols like process pairs, and rollback-recovery [Elnozahy99, Huang93], survive a specific class of software faults known as “Heisenbugs” [Gray86] provided they do not violate the fail-stop model. Heisenbugs are transient, non-deterministic faults that may not occur again when the operation is retried, even if the same code is used. The transient nature of the fault arises because some factor external to the program has changed; for example, a different interleaving order of threads may occur during retry and so avoid a race condition. Note that a truly generic recovery mechanism must preserve all application state (e.g. by checkpointing or logging), because there is no application-specific code to reconstruct missing state. Hence only a change external to the application can allow the application to succeed on retry.

It has been hypothesized that most faults that occur in released applications are transient [Gray86]. The intuition for this hypothesis is that transient faults such as race conditions are more difficult to reproduce and hence debug than non-transient faults (so-called Bohrbugs), so transient faults are more likely to remain in released software. This is an important hypothesis for guiding research in how to recover from software faults, because it encourages work on application-generic recovery. The primary goal of this study is to test this hypothesis by analyzing bug reports and bug fixes of several large, widely used, open-source programs.

4.3 Categorizing Software Faults

We classify software faults based on how they depend on the operating environment. By operating environment, we mean states or events that occur outside of the application being studied. The operating environment includes both software and hardware. Software examples of operating environment include other programs, such as the DNS name server and user-level applications, or the kernel, such as the number of available slots in the kernel's process table. Hardware examples include transient hardware conditions such as disk ECC errors and events such as clock interrupts to the thread scheduler. The operating environment also includes the timing of the workload requests to the program (e.g. the user's typing speed). However, we consider the sequence of workload requests made to the program as part of the program, rather than as part of the operating environment, because the sequence of requests is usually fixed for any given program task. That is, we assume the user is not willing to aid recovery by avoiding certain input sequences.

Non-deterministic execution is always due to a change in the operating environment. For example, a race condition is non-deterministic because of the different times a clock interrupt is delivered to the thread scheduler. This connection between changing operating environment and non-deterministic execution is why we classify faults based on their dependence on the operating environment.

We classify software faults into two main categories: environment-independent and environment-dependent.

Environment-independent faults occur independent of the operating environment. Given a specific workload (e.g. requested operations from the user), an environment-independent fault will always occur. As an example, one release of Apache had an environment-independent fault that caused it to fail whenever a browser submitted a long URL. Because environment-independent faults do not depend on the operating environment, they are completely deterministic (non-transient). Hence application-generic recovery techniques will not survive these faults.

Environment-dependent faults depend on the operating environment. For these faults, the operating environment plays some role in triggering the design bug in the program. For example, the program may not deal gracefully with a slow network, a full disk, or an operating system that has run out of file descriptors. Because environment-dependent faults depend on the operating environment, the program may behave differently when the requested operation is retried. Hence, application-generic recovery techniques may survive these faults if the environment changes enough to avoid the fault when the operation is retried.

Environment-dependent faults can be further classified into transient and non-transient, depending on how likely it is for the operating environment to be fixed in the absence of application-specific recovery. In *environment-dependent-nontransient* faults, the environment is unlikely to be changed enough to avoid the bug during retry. For example, the program may fail if the disk is full. We consider this an environment-dependent-nontransient bug because most current systems do not fix this condition automatically. Of course, some systems may provide a way to automatically increase the disk capacity and hence avoid the bug during retry. If this becomes common, we would re-classify this as an environment-dependent-transient fault.

Like environment-dependent-nontransient faults, *environment-dependent-transient* faults depend on the environment. Unlike environment-dependent-nontransient faults, however, the environmental dependency is such that simply retrying the operation is likely to encounter a different environment and hence succeed. For example, a race condition will typically be triggered by a specific interleaving of threads by the thread scheduler. If the operation is retried, the environmental condition (the specific interleaving of threads) is likely to be different, and the operation may succeed. As another example of an environment-dependent-transient fault, a program may create child processes but not kill them. These programs typically fail when the operating system runs out of processes. A typical generic recovery system would kill all processes related to the application (thereby changing the operating environment), recover the program, and successfully continue.

4.4 Software and Faults Targeted

Every piece of software goes through a huge number of bugs over its lifetime of development and use. In this study we look at a subset of the faults that were detected by the users of released versions of the software. Fault-recovery techniques are generally directed at this subset of faults. Among this subset of faults we concentrate on high-impact faults, i.e. those that cause the software to crash, return an error condition, cause security problems, or stop responding. There are many other types of faults that we do not examine, such as those encountered during compilation and installation. These faults do not cause critical outages because they occur before the software is being used in a production setting. We assume that users test new versions of software before incorporating them into their production environment.

Our primary source of data for analyzing faults is the on-line bug reports that are maintained for open-source software. These bug reports contain information about each reported fault, including the symptoms accompanying the fault, the results of the fault, the operating environment and workload that induces the fault, and, in most cases, how the underlying bug was fixed. A key field in all the bug reports we study is the “How To Repeat” field. We use the information supplied in this field along with the comments entered by the developers of the software to decide which fault class the fault belonged to. The developers also provide information on how the bug was fixed and whether they could repeat the failure on their development machines.

We analyze three large, open-source applications: Apache, GNOME, and MySQL. Apache is a robust and commercially used open-source implementation of an HTTP (web) server. As of November 1998, Apache was being used by 36% of the 53,265 Internet domains owned by U.S businesses with annual revenue of $10 million or more [SiteMetrics98], and as of October 1999, Apache was being used as a web server by 54% of over 8 million sites polled in a survey [Netcraft99]. The Apache bug reports are available at http://bugs.apache.org. Of all the bugs reported, we consider bugs on production versions of the software that were categorized as severe or critical. The site has a total of 5220 bug reports listed, and in our study we narrow these to 50 unique bug reports meeting these criteria.

GNOME (GNU Network Object Model Environment) is an open-source desktop environment for users and a powerful application framework for software developers.

GNOME, along with KDE, is included in most of the Linux distributions available today. GNOME is also used by users running other flavors of UNIX on their hardware. We use two sources of fault information for our survey. The first (http://bugs.gnome.org) provides us with bug reports and the second (http://cvs.gnome.org) provides us with information about how the bugs were fixed. The GNOME package comes with a set of core files and libraries and a number of applications. We look at faults in the core files and libraries and four commonly used GNOME applications: panel (a user customizable toolbar), gnome-pim (a personal information manager), gnumeric (a spreadsheet application), and gmc (a file management utility). We looked at about 500 bug reports and narrowed them to 45 unique bugs meeting our criteria.

MySQL is a multi-user, multi-threaded SQL database server designed for client-server implementations. It is a fast and robust server aimed at handling small to medium amounts of data. These features make it the database server of choice for web applications and ISP's offering database-hosting services. All the fault data used in our study was obtained from the archives of the mysql mailing list at http://www.geocrawler.com. There are about 44,000 messages archived at the website. In this study, we use all the messages from the archives that matched one of the following keywords: "crash", "segmentation", "race", and "died" (we looked at a few hundred messages and found that these keywords were the ones commonly used to describe serious bugs). We then narrowed these messages to 44 unique bugs.

4.5 Results

We categorize the faults for each of the three applications into the three classes discussed earlier and describe more fully the causes of all the environment-dependent faults.

Since the number of environment-independent faults is very high we only describe a few of them in this chapter. The complete list and descriptions of the environment-independent faults can be found in the appendix “LIST OF ENVIRONMENT-INDEPENDENT FAULTS” on page 86

4.5.1 Apache

Class	**# Faults**
environment-independent	36
environment-dependent-nontransient	7
environment-dependent-transient	7

Table 4.1: Classification of faults for Apache
Environment-independent faults do not depend on the operating environment and are therefore deterministic. Environment-dependent-nontransient faults depend on the operating environment, and the environmental condition that triggers the fault is likely to persist during retry. Environment-dependent-transient faults depend on the operating environment, and the environmental condition that triggers the fault is likely to be fixed during retry.

Table 4.1 on page 61 contains the results of classifying 50 faults for Apache. The overwhelming majority of faults do not depend on the operating environment. These deterministic bugs are relatively easy to repeat, so it is perhaps surprising that Apache suffered from so many in released software. Even for deterministic faults, testing all the boundary conditions is notoriously difficult.

The following are some of the environment-independent bugs reported for Apache:

- Dies with a segfault when the submitted URL is very long. This problem was a result of an overflow in the hash calculation.
- SIGHUP kills apache on Solaris and Unixware. Normally, this should gracefully restart/rejuvenate Apache.

- Dumps core on Linux/PPC if handed a nonexistent URL. The problem is that ap_log_rerror() uses a va_list variable twice without an intervening va_end/va_start combination.
- This error occurs when directory listing is turned on and the directory has zero entries. The palloc() call used in index_directory() doesn’t handle size zero properly.
- Shared memory segment keeps growing and reaches sizes exceeding 100 Mbytes in less than 5 hours of operation.
- When a HUP signal is sent to rotate logs, Apaches freezes or dies. This is caused by memory leaks in the application.

Of the 14 faults that do depend on the operating environment, half are due to conditions that persist during recovery. The conditions of the operating environment that trigger the 7 environment-dependent-nontransient bugs are:

- High load leading to an unknown resource leak. Resource leaks in the application will persist during recovery, assuming a generic recovery mechanism that saves and recovers all application state.
- Lack of file descriptors. As above, truly generic recovery mechanisms will recover all application resources such as file descriptors, so this condition will persist during recovery.
- Disk cache used by the application gets full and the application cannot store any more temporary files.
- Size of log file is greater than maximum allowed file size.
- Full file system.

- Unknown network resource.exhausted.
- Removal of PCMCIA network card from the computer

The remaining faults are triggered by conditions in the operating environment that are likely to be fixed during recovery. The conditions of the operating environment that trigger the 7 environment-dependent-transient bugs are:

- Call to Domain Name Service returns an error. This is likely to change when the DNS server is restarted.
- Child processes hangs during peak load and consume all available slots in the process table. As part of automatic recovery, the recovery system is likely to kill all processes associated with the application.
- User presses stop on the browser in the midst of a page download. This fault depends on the exact timing of the requested workload, which is not likely to be repeated during recovery.
- Hung child processes hang onto required network ports. These will likely be killed during recovery and the ports will be freed.
- Slow Domain Name Service response. The cause of the slow DNS response will likely be fixed eventually without application-specific recovery, either by restarting DNS, or by fixing the network.
- Slow network connection. The network may be fixed by the time Apache recovers.
- Lack of events to generate sufficient random numbers in /dev/random. During recovery, it is likely that more events will be generated for /dev/random.

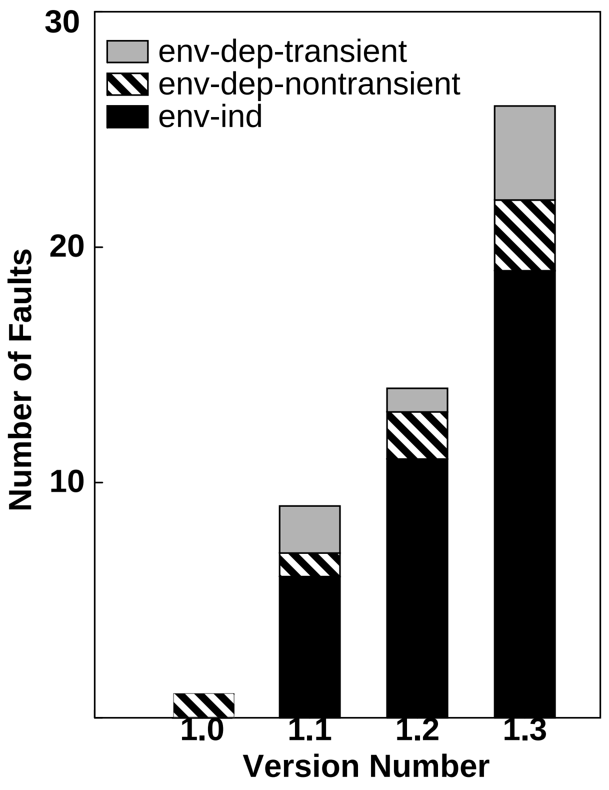

Figure 4.1: : Distribution of faults for Apache over software releases.

Figure 4.1 on page 64 shows the distribution of faults over releases of the Apache software. The fault distribution exhibits two distinct properties. First, the relative proportion of environment-independent bugs stays about the same even for new releases of the software. Second, the total number of bugs reported increases with newer releases of software. This is probably due to an increase in the number of users of the newer releases.

4.5.2 Gnome

Table 4.2 on page 65 contains the results of classifying 45 faults for GNOME. As with Apache, the overwhelming majority of faults do not depend on the operating environment.

The following are some of the environment-independent bugs reported for Gnome:

- Clicking on the “tasklist” tab in gnome-pager settings, causes the pager to die.

Class	# Faults
environment-independent	39
environment-dependent-nontransient	3
environment-dependent-transient	3

Table 4.2: Classification of faults for GNOME
Environment-independent faults do not depend on the operating environment and are therefore deterministic. Environment-dependent-nontransient faults depend on the operating environment, and the environmental condition that triggers the fault is likely to persist during retry. Environment-dependent-transient faults depend on the operating environment, and the environmental condition that triggers the fault is likely to be fixed during retry.

- Clicking on the “prev” button in the “year” view of the gnome calendar application causes it to crash. This was due to assigning a value to a local copy of the variable instead of the global copy.
- The spreadsheet application “gnumeric” crashes if a tab is pressed in the “define name” dialog or in the “File/Summary” dialog. This was caused by initializing a variable to an incorrect value.
- Double-clicking on a “tar.gz” file that is lying as an icon on the desktop crashes gmc, the gnome file manager. This was caused due to the declaration of a variable as “long” instead of “unsigned long”.
- After clicking the main button once to pop up the main menu, a click again on the desktop in order to remove the menu freezes the desktop.

The conditions in the operating environment that trigger the 3 environment-dependent-nontransient bugs are:

- Hostname of the machine was changed while the application was running.

- Open sockets left around by sound utilities while exiting. Each open socket consumes a file descriptor and the application runs out of file descriptors.
- File has an illegal value in the owner field. Application crashes when trying to edit the file or its properties

The conditions in the operating environment that trigger the 3 environment-dependent-transient bugs are:

- Unknown failure of application which works on a retry
- Race condition between a image viewer and a property editor. Race conditions depend on the exact timing of thread scheduling events, and these are likely to change during retry.
- Race condition between a request for action from an applet and its removal.

Figure 4.2 on page 67 shows the distribution of faults reported over time. We used time as opposed to releases because of the nature of GNOME. GNOME is a collection of modules, each of which are released independently. There is an occasional major release of all the modules put together. But there was only one release over the period we performed the fault study. The distribution shows that the proportion of environment-independent bugs is very high over all periods. GNOME shows a decrease in the number of faults reported for a short interval before increasing again. This was probably a period of few changes in the software.

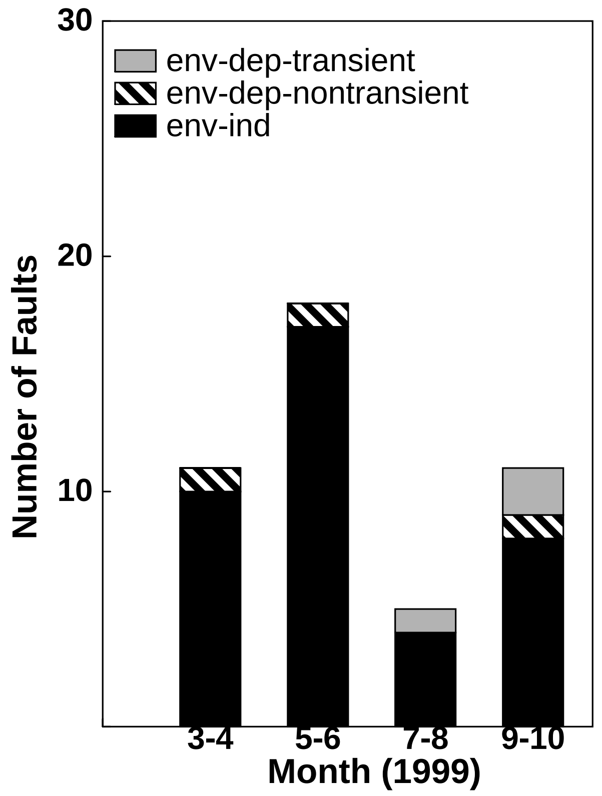

Figure 4.2: : Distribution of faults for GNOME over time.

4.5.3 MySQL

Class	# Faults
environment-independent	38
environment-dependent-nontransient	4
environment-dependent-transient	2

Table 4.3: Classification of faults for MySQL

Environment-independent faults do not depend on the operating environment and are therefore deterministic. Environment-dependent-nontransient faults depend on the operating environment, and the environmental condition that triggers the fault is likely to persist during retry. Environment-dependent-transient faults depend on the operating environment, and the environmental condition that triggers the fault is likely to be fixed during retry.

Table 4.3 on page 67 contains the results of classifying 44 faults for MySQL. As with the other two applications, the overwhelming majority of faults do not depend on the operating environment.

The following are some of the environment-independent bugs reported for MySQL:

- Updating an index to a value that will be found later while scanning the index tree and hence creating duplicate values in the index will crash MySQL. This was solved by first scanning for all matching rows and then updating the found rows.
- A query which selects zero records and has an "order by" clause will cause the server to crash. This was due to some missing initialization statements.
- The use of a "count" clause on an empty table causes MySQL to crash. This was cause due to missing check for empty tables.
- An "OPTIMIZE TABLE" query crashes the server. This was caused by a missing initialization statement.
- A "FLUSH TABLES" command after a "LOCK TABLES" command crashes the server.

The conditions in the operating environment that trigger the 4 environment-dependent-nontransient bugs are:

- Shortage of file descriptors due to competition between MySQL and a web server.
- Server crashes when it receives a connection request from a remote machine if reverse DNS is not configured for the remote host.

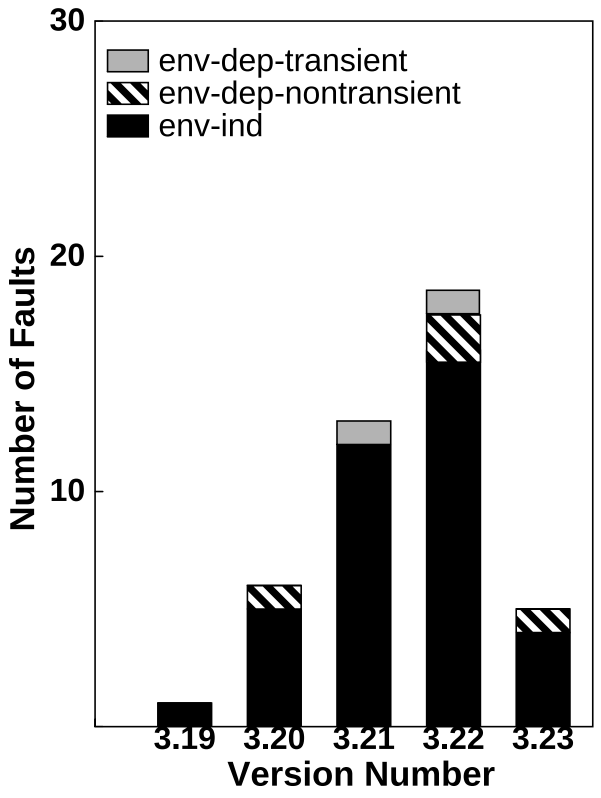

Figure 4.3: : Distribution of faults for MySQL over software releases.

- Size of database file is greater than the maximum allowed file size.
- Full file system prevents all operations on the database.

The conditions in the operating environment that trigger the 2 environment-dependent-transient bugs are:

- Race condition between the masking of a signal and its arrival. race conditions depend on the exact timing of thread scheduling events, and these are likely to change during retry.
- Race condition between a new user login and commands issued by the administrator.

Figure 4.3 on page 69 shows the distribution of faults over releases of the MySQL software. The fault distribution exhibits two distinct properties similar to what Apache exhibits. One, the relative proportion of environment-independent bugs stays about the same even with new releases of the software. Two, the total number of bugs reported increases with newer releases of software. This is probably due to increase in the number of users of the newer releases. The last release has a substantially lower number of faults because the release is very new and hence very few users are using the software and reporting bugs.

4.6 Discussion

The distributions of faults for the three pieces of software we looked at show that there are very few environment-dependent faults. Of the 139 bugs we looked at, we found 14 (10%) environment-dependent-nontransient faults and 12 (9%) environment-dependent-transient faults. We acknowledge that classifying bugs between environment-dependent-transient and environment-dependent-nontransient classes is subjective and depends upon the recovery system in place. However, this does not affect the fact that the number of environment-independent faults is very high.

This result differs from the general perception that a majority of bugs in released software are Heisenbugs [Gray86]. According to this perception, most Bohrbugs are caught during development, or perhaps during early releases of the software. As these bugs are fixed and the software becomes more stable, the relative percentage of Heisenbugs in the remaining bugs should increase. In today’s software culture, however, new features and code are added very quickly, and this rapid rate of change may prevent the application from reaching stability.

Another possible explanation for why Heisenbugs are so rare is that they occur infrequently and so do not appear in bug reports. However, recovery efforts should be directed at the faults that occur most frequently in practice, and these appear to primarily consist of environment-independent Bohrbugs.

As with any case study based on reported data, a few limitations apply to our work. First, results may differ widely for other applications; for example, mission-critical applications undergo more rigorous testing than most software, and this may change the distribution of faults. Second, the reported data may be biased; for example, people may tend not to report faults they cannot duplicate. Third, we analyze the distribution of bugs, rather than the distribution of actual failures. Finally, we reason from bug reports about the ability for generic recovery mechanisms to recover from specific faults. An important avenue of future research is to implement generic recovery and verify its ability to survive a specific fault [Lee93]. This would serve as an end-to-end check on whether the bug report had a complete list of environmental dependencies for that fault.

4.7 Surviving Software Faults

In this section we look at various techniques to survive software faults belonging to different classes. Some of these techniques are well known, especially for environment-dependent-transient faults and some categories of environment-dependent-nontransient faults. For other categories of faults we suggest some techniques to survive them and refer to techniques suggested by researchers.

4.7.1 Environment-Independent Faults

Since environment-independent faults are guaranteed to reoccur given a specific workload there is no easy or general technique to recover applications after the fault has manifested itself. The best way to survive such faults is to prevent them from occurring in the first place. However, this is not always possible due to the difficulties associated with testing all of the boundary conditions the software may encounter in the field. Formal code inspections and thorough testing are highly effective to remove software faults before releasing software [Weller93].

There also exist languages and tools to help solve some of the problems in this category. Languages like Java, which are type-safe and allocate/deallocate memory automatically, can be used solve problems like buffer overflows and memory leaks. Tools like Purify can also help find problems related to memory allocation and deallocation. Tools like Ballista [Kropp98] test functions for boundary conditions and place wrapper code around them to prevent failure. Using standard libraries like POSIX helps prevent problems related to inconsistent function behavior on different operating systems.

4.7.2 Environment-Dependent-Nontransient Faults

Most environment-dependent-nontransient faults are due to some resource being exhausted, such as file descriptors, sockets, or disk space. There are two general approaches for solving resource exhaustion. One way is to detect the problem and automatically increase the resources available to the application. For example, the operating system may be able to dynamically increase the number of file descriptors available to a process. This approach works when the application only temporarily exceeds the resources available (e.g. during a peak load).

The second approach is to try to automatically decrease the amount of resources used by the application. One way to do this is to use garbage collection techniques to discern which resources are no longer needed and reclaim these. For example, the system may monitor which file descriptors are used and automatically close the unused ones. Or the system may provide “virtual sockets” to an application by multiplexing the application’s sockets onto the system’s sockets.

Some applications implement application-specific solutions to prevent some environment-dependent-nontransient faults. One example is Apache, which can be rejuvenated by sending it a special signal. During rejuvenation, Apache kills all its child processes and thereby reclaims any process structures used up by zombie processes. This technique is widely used by web administrators to reduce failures in Apache. A more detailed discussion of this type of rejuvenation can be found in [Huang95].

4.7.3 Environment-Dependent-Transient Faults

Process pairs [Gray86] and rollback-recovery protocols [Elnozahy99, Huang93] in general solve faults in this class very well. Faults in this class are related to either time or to an environmental condition which is expected to change very frequently. So retrying the same operation at a later time will usually succeed. Some techniques have also been suggested to induce changes in the environment to increase the success of a retry without affecting the program correctness. One such technique changes the message ordering to simulate changes in the environment [Wang93].

The fact that recovery protocols can recover from this class of faults should not preclude us from developing techniques to prevent these faults from happening in the first place. One such technique [Savage97] looks at detecting data race conditions in lock-based multithreaded programs.

4.8 Related Work

Several prior studies have examined the faults that occur in released software. Most of these studies do not discuss the interaction between the faults and a recovery system. Because of this, their error descriptions are not sufficiently detailed to classify faults precisely into transient and non-transient faults. For comparison purposes, however, we can infer a rough classification based on the information they do provide. Our rough classification of faults studied in related papers supports our conclusion that most faults in released software are non-transient, and therefore, that application-specific recovery is needed to survive these faults.

Sullivan and Chillarege study faults in the MVS operating system and the DB2 and IMS database systems [Sullivan91, Sullivan92]. They categorize some errors as being timing or synchronization related, either in terms of the error type or the error trigger. These errors are likely to be environment-dependent-transient faults. They found that 5-13% of the faults were timing or synchronization related. Sullivan and Chillarege were surprised (as we were) that most faults were non-transient bugs (e.g. boundary conditions) that would have been relatively easy to catch during the testing phase.

Lee and Iyer study faults in the Tandem GUARDIAN operating system [Lee93]. They also have a category for errors related to timing and race conditions, which comprise 14% of the faults. Again, these results match ours conclusion that most faults do not depend on the operating environment and so are non-transient.

Lee and Iyer also examined how often the Tandem process-pair mechanism enabled the system to recover from a software fault. They found that 82% of the software faults could be recovered with process pairs. This is a much higher fraction than our estimates and requires some explanation. First, we are assuming a pure application-generic recovery system, and the Tandem operating system's process pair implementation uses some application-specific information. For example, Lee and Iyer report that many errors were recovered because the backup process did not start from the same state as the failed primary (this eliminates their “memory state” and “error latency” categories). Second, a large fraction of recovered faults were due to the backup not re-executing the requested task (perhaps because the task was directed at a specific processor rather than to the process-paired application). In our model, all requested tasks need to be executed; we do not assume a user will generously avoid the fault trigger, nor do we consider faults below the process-paired application. Third, many recovered faults in [Lee93] only affected the backup process. This results from bugs introduced by the process-pair system. Lee and Iyer count these as transient bugs; we are only concerned with bugs in the application itself. After eliminating these sources of differences from consideration, only 29% of the software faults are transient bugs in the operating system. This is still somewhat higher than we found, and we conjecture that this is due to two reasons. First, Tandem software is probably tested more thoroughly than most current software, and this testing likely eliminates more non-transient faults than transient ones. Second, operating system software interacts more closely with the hardware than the application-level software we study. This interaction creates more dependencies on the environment, which increases the fraction of environment-dependent-nontransient and environment-dependent-transient faults.

Several schemes have been proposed to enable generic recovery to work for a larger class of faults. For example, some recovery techniques seek to increase the non-determinism in the application by re-ordering events such as message receives [Wang93]: these are basically techniques to induce change to the external environment. These do not transform environment-independent faults into environment-dependent faults. Rather, they increase the chance that a environment-dependent fault will experience a different operating environment (order of message receives, in this case) during recovery. A second technique that seeks to reduce the impact of software bugs is software rejuvenation [Huang95]. Software rejuvenation takes advantage of recovery code that is already present in the application, e.g. code to re-initialize the application's state. Software rejuvenation seeks to prevent failures by invoking this application-specific recovery code before the program crashes.

4.9 Conclusion

Our growing dependence on computers increases the need for tolerating faults in general and software faults in particular as they cause the majority of failures. A better understanding of their properties and behavior will help us design better recovery mechanisms. Our goal in this chapter is to evaluate one such property of software faults namely their non-deterministic nature. This evaluation also tests the hypothesis that generic recovery techniques can survive most software faults without using application-specific information as they assume that most software faults are transient or non-deterministic.

We examined in detail the faults that occurred in three, large, open-source applications: the Apache web server, the GNOME desktop environment, and the MySQL database. Using information contained in the bug reports and source code, we classified faults based on how they depended on the operating environment. We found that 72-87% of the faults were independent of the operating environment and were hence deterministic (non-transient). Recovering from these faults requires the use of application-specific knowledge. Half of the remaining faults depended on a condition in the operating environment that was likely to persist on retry. These faults are also likely to require application-specific recovery. Unfortunately, only 5-14% of the faults were caused by transient conditions, such as timing and synchronization, that would naturally fix themselves during recovery. Our data indicate that classical application-generic recovery techniques, such as process pairs, will not be sufficient to enable these applications to survive most software faults.

Our hope is that this study will encourage the design of mechanisms which can recover from the majority of faults that are non-deterministic or environment independent. Recovery mechanisms which can fix the environment will help transform some of the environment-dependent-nontransient faults into transient faults and improve the recovery rate of generic recovery mechanisms. Better error detection techniques and programming tools can help reduce the number of environment-independent faults. Recovery techniques which can take better advantage of the non-determinism present in the environment will be able to recover from more faults.

CHAPTER 5

CONCLUSIONS AND FUTURE WORK

5.1 Conclusions

We have computers on our desktop today that are more powerful than what we termed as supercomputers some years back. Their capabilities are still increasing at an enormous rate as predicted by Moore's Law. This gives them the capability to run complex software which pervades our daily life. We use computers to do everything from buying everyday necessities to controlling infrastructure like power and water to entertaining ourselves. Yet most people do not trust software as they have been bitten by software failures at one time or another. Developing reliable software is still one of the bigger challenges in computer science.

What have we achieved so far in building reliable computer systems? We have been successful in building very reliable computer hardware. We have succeeded in building highly reliable custom-built software systems. We use these systems to run applications like our stock exchange, banking, air traffic control etc. However, the applications we use in our every day life are not reliable yet. Desktop operating systems, web browsers, office suites etc. are prone to failures. These failures directly affect us and may hide the fact that very reliable systems are running in the back ground. Having a highly reliable stock exchange is of no use if we cannot connect to it because of a browser failure.

Researchers have proposed a wide variety of generic recovery mechanisms to recover from failures. These mechanisms work well for hardware failures which tend to be operational in nature. However, these mechanisms rely upon two assumptions for them to work for software failures. One is that they assume that software faults are fail-stop and the second is that they assume that most of the software faults are activated in a non-deterministic fashion.

Our study of the fail-stop property shows that general-purpose applications show a lot of variation in how fail-stop they are. Applications like Postgres, a database application, have very good error detection and recovery mechanisms. These applications violate the fail-stop property very rarely. However, the recovery mechanisms used are complex to build and so are expensive. So common applications like word processors and spread sheets do not use these kind of recovery mechanisms. They use much simpler mechanisms to detect errors and recover from failures. This is reflected in the much higher number of fail-stop violations detected in our study.

Researchers have been working on using generic recovery mechanisms for these kind of applications which cannot afford custom built recovery mechanisms. Generic recovery mechanisms were originally developed to handle hardware failures at which they were very successful. Since generic recovery mechanisms are designed to work for any kind of application and their development costs can be amortized over all applications, their use appears attractive for general purpose applications. However, our study shows that generic recovery mechanisms are very bad at upholding the fail-stop property during a failure. It is precisely their generic nature that makes them bad at recovering from application software failures. Their lack of knowledge about the application causes them to save more state at a higher frequency than what is needed for recovery. This makes them more susceptible to saving the state of the fault itself in the committed state. This is reflected in our results where using a generic recovery mechanism meant that about half the runs did not recover from a failure. We also measured which of the two factors, comprehensiveness or frequency of state commits, causes more fail-stop violations in generic recovery. Our data does not provide a conclusive answer for which one of the two factors causes more fail-stop violations.

Generic recovery mechanisms were originally designed for recovery from hardware failures which occur outside the application software. So we looked at how well they work when faults are present in the operating system rather than in the application itself. Our study shows that there are a lot fewer fail-stop violations when faults are injected into the operating system. Fail-stop violations still occur as there is some transfer of state between the operating system and the applications through system calls.

Our second study looks at the non-deterministic nature of software faults. We looked at software bug reports for three commonly used open-source applications. Open-source applications are being widely used today and so form a good basis for looking at software faults. Further, their open nature gives us access to valuable data about faults present in them, their effects and fixes for them. This data has not been traditionally available for proprietary software. We can also follow their evolution over multiple releases and look at how the types of faults present in them change over time.

The activation of software faults is non-deterministic only due to changes in the environment. So we classified all software faults based on their dependence on the environment into environment-dependent faults and environment-independent faults. We further divided environment dependent faults into environment-dependent-transient faults and environment-dependent-nontransient faults depending on whether the environmental conditions that triggered the fault are likely to change during the recovery process. We analyzed the bug-reports of three open-source applications and classified all the critical and serious faults in them into these three categories.

The results from this study show that 75% or more of the faults in these applications are independent of the environment. So these faults will not be recovered by generic recovery mechanisms as these faults will be activated again after recovery. Of the remaining faults about half the faults are dependent on the environment but the environmental conditions which triggered these faults are unlikely to change. So these faults will not be recovered by generic recovery mechanisms either. The remaining 10-15% of the faults are what generic recovery mechanisms will be successful at recovering applications from after a failure.

We also wanted to evaluate the belief that over time most of the faults remaining in a piece of software will be environment-dependent faults. It is assumed that environment-independent faults will be detected easily compared to environment-dependent ones and so they will be fixed leaving a higher proportion of environment-dependent faults in the application. We took the same faults used in the above study and classified them based upon which release of software they were detected in. We found that the proportion of environment-dependent faults versus environment-independent faults remained about same over multiple versions of the software. We attribute this to the constant addition of new features to each new release of software and the rate at which new versions of the software are released nowadays. All this new code contributes more environment-independent bugs to the software. So even mature versions of the software will not be recovered by generic recovery mechanisms from most of the software faults.

The goal of this thesis was to evaluate assumptions made about the behavior of software faults by recovery mechanisms. Our evaluation shows empirically that the assumptions about the fail-stop and non-deterministic nature of software faults are not safe assumptions. Our hope is that, in the future, recovery mechanisms will try to recover from software failures without making these assumptions about the behavior of software faults and thus improve the reliability of general purpose software applications.

5.2 Future Work

This section looks at a couple of research directions to improve and extend the work done in this thesis. The first one is to look at ways to improve the methodology we used to measure properties of software faults. The second one is to use the results obtained in this thesis to improve the mechanisms used for recovering from software faults.

Our fail-stop study used a fault model which was derived from prior studies of faults that have been found in software. We injected faults into the application based upon this fault model and evaluated their fail-stop properties. We have tried to make the injected faults as realistic as possible. However, there is always room to improve our methodology. Recently, open-source software applications have started maintaining source code repositories accessible to the general public along with detailed bug reports. Using these resources we can obtain any specific version of the software which had a bug and activate this specific bug in a similar way as it would have been activated in field use. We can then evaluate whether the failure caused by this fault was fail-stop or not. This methodology can also be used for the non-determinism study. This way we can activate a certain fault and check whether the fault is activated again during or after recovery. This can give us a better understanding of the properties of software faults and their effect on recovery mechanisms than we can get by emulating these faults using fault injection.

The results we obtained in this thesis are basically negative results. They show that generic recovery mechanisms do not work very well for recovering software applications from software faults. Even application-specific recovery mechanisms violate the fail-stop model a significant number of times. We also found that a majority of software faults are environment-independent which means they are hard to recover from whether we use application-specific or generic recovery mechanisms. So the best solution is to avoid having faults in software in the first place. This is of course easier said than done. This requires the development of techniques which help programmers write code without faults in them. There have been some developments in this area like the development of type-safe languages and languages with built-in garbage collectors. Another direction for future research would be into the area of better error detection. Better error detection will reduce the number of fail-stop violations. Currently most of the burden of implementing error detection is left to the application programmer and this leads to insufficient error detection in general purpose software. Research into providing error detection tools like standard libraries which the programmer can just make calls to would help improve the amount of error detection built into software. Similarly providing recovery mechanisms [Huang93] as standard components which an application programmer can use in according to the needs of the application will take advantage of the benefits of both application-specific and generic recovery methods.

The non-determinism study showed that half of the environment-dependent bugs are dependent on environmental conditions which are not transient. Techniques can be developed which can sense changes in the environment and prevent those changes or can fix the environment after a failure. An example of one such technique is software rejuvenation [Huang95]. Better application-specific recovery techniques as discussed in the earlier paragraph will also reduce the reliance upon bugs being environment-dependent to recovery from them.

We hope that this thesis will provide better insight into the properties of software faults. We also hope that the results will spur the development of reliable software for general purpose use.

APPENDIX

APPENDIX A
LIST OF ENVIRONMENT-INDEPENDENT FAULTS

A.1 Fault Study for Apache

- On some versions of unix fopen with the O_APPEND flag behaves differently. This causes errors in the fast_cgi module while logging.
- Dumps core due to an uninitialized pointer. Happens when there is a bad directive in the config file and request first goes to the directory specified in the bad directive.
- Connection reset by peer. Apache does not drop connection and keeps retrying for ever and hangs.
- Server logs get mixed up when several virtual servers are running on the same physical machine
- Dies with a segfault when the submitted URL is very long. This problem was a result of an overflow in the hash calculation.
- SIGHUP kills apache on Solaris and Unixware. Normally, this should gracefully restart/rejuvenate Apache.
- Piping large amounts of data to the logs causes a SIGPIPE signal to be sent to apache. Apaches stops responding to http requests after receiving the signal though it will respond to server status requests and report itself as alive.
- On win32 versions an url with a lot of "../" substrings causes buffer overflows and poses a security risk as well as causing server crashes.
- Memory was being allocated from the wrong pool and so was being prematurely freed. This caused fields like Remote UserID being corrupted.

- A denial of service attack causes a high load on the apache server and brings it down.
- The server does not service the client request and gives an error message "Invalid method on request". This occurs because of incompatibilities between apache and the MSIE 4.71 browser.
- An empty line between content and date in the request field of an ftp request causes apache to send corrupted data back to Netscape 4.05, GNU wget 1.4.3 and lynx 2.81dev5 browsers.
- Win32 and OS/2 use case-blind file systems. This causes corruption of data and security problems if Apache is accessing directories which differ only in case.
- Apache fails to execute 16bit CGI scripts on Win32 systems when the script is executed more than once.
- When an user clicks on the STOP button on the browser while a CGI script is sending data, the CGI script turns into a zombie. This is especially a problem when the CGI scripts are run within the server.
- Some CGI scripts cause internal server errors when syslog is used to log Errors.
- Whenever Apache tries to print an error message about a file not being found and the name of the file contains a "%" sign, then Apache goes into an infinite loop in the "printf routine" trying to resolve the "%".
- Apache shows random data from memory while displaying directory listings. Occasionally the listing also showed data from the password file.

- Shared memory segment keeps growing and reaches sizes exceeding 100 Mbytes in less than 5 hours of operation. When a regularly scheduled signal is sent to Apache to rotate the logs, the server either freezes or crashes.
- Dumps core on Linux/PPC if handed a nonexistent URL. The problem is that ap_log_rerror() uses a va_list variable twice without an intervening va_end/ va_start combination.
- Using the setMaxAge(0) method does not delete a cookie as it should on the release and platform 1.3.6/jserv1.0 on WindowsNT.
- Server hangs after running for some time on HPUX-11. This happens because the program is in an infinite loop trying to open a non-existent mutex file.
- Apache dies after a restart with "Invalid argument: accept: (client socket)". Happens on a variety of linux machines.
- Problem caused when maximum number of clients is greater than 256. This happens because the variable which keeps track of number of connections on the scoreboard is of 8 bits in size.
- This error occurs when directory listing is turned on and the directory has zero entries. The palloc() call used in index_directory() doesn't handle size zero properly.
- Server times out while sending HTTP documents to client. This happens because "Content-Length" in the header is being incorrectly set due to a wrong variable being passed to a function.
- Files whose transmission was interrupted are being cached by Apache acting as a proxy even though they are incomplete and should be discarded.

- A map file with an entry like "../../sales/index.html" called from a URL like "http://server.com/graphics/cell.map" hangs the server. The server gets stuck in a loop trying to remove the second "../" because of an incorrect use of "while" instead of an "if".
- Apache sends incorrect Content-length headers upon receiving Range headers that are negative in value or greater then legal value.
- Apache crashed due to a segmentation fault if the first variant being treated by the function "is_variant_better_na" is unacceptable. This is due to some pointers being set incorrectly.
- Apache running as a proxy crashes for every 50MB of data passing through it or upon an aborted retrieval of a long document. This due to a missing null pointer check.
- The function "Jserv.getParameter(String name) throws a null pointer exception if a page is requested without parameter.
- Directory listings passing through a proxy were missing the first character of filenames in the directory. This was due to a design flaw in the code extracting filenames out of directory entries.
- Apache has memory leaks while running virtual servers.
- Apache crashes in mod_jserv if redirected to a servlet by an ErrorDocument directive. This happens due to an uninitialized pointer being null.
- Apache goes into an infinite loop after a CGI error. This happens because apache is making an incorrect assumption about how Win32 uses the variable "errno".

A.2 Fault Study for GNOME

- The pager disappears if the "tasklist" tab in gnome-pager settings is clicked upon.
- The foreign language informations gets dropped from the desktop entries if the system menus are edited as root. The entries are not saved to the desktop configuration files. This is due to a design error where a data structure has no entries for languages other than English.
- Gimp crashes when it is launched from a panel menu or from gnome-run and hangs all user functionality. Process is trying to read from closed "stdin". It should be remapped to /dev/null to prevent gnome from hanging.
- Panel will not restart after a crash. It says that it is already running when gnome/Enlightenment is loaded.
- Desk Guide and Tasklist applets running under Enlightenment crash when a file is deleted using gmc.
- The following sequence of inputs will crash gnome

  1) Run gmenu as an normal user

  2) Go to "User Menus", press the right mouse button and choose "New Item"

  3) In "Name" type a name in capital letters and for "Type" choose "directory"

  4) Click on "Save", Click on "File" and then on "Exit"

  Now clicking on the newly created menu item will crash gnome. This is due to a memory leak.

- Using a GL applet like "Quake 2" as superuser hangs GNOME. After a restart, the GNOME desktop switcher and clock disappear and any number of restarts do not restore them.
- Accessing the dialogue options in the gnome control center causes the computer to stop responding soon because of swapping. This is caused by a memory leak in the process "ui-properties".
- Using the disk space monitor crashes the computer.
- Leaving the gnome-help-browser running with a large HTML file being displayed causes gnome to leak memory and make the system unresponsive.
- Placing the "Another Clock" applet on the panel and using the right mouse button to move it causes gnome to crash.
- Starting a terminal window from gmc and closing it with icewin's close button on the window list causes gnome-session to die.
- Icon browser dialog does not release memory after loading a thumbnail. Continued use of the dialog box causes system to run out of memory.
- If an invalid time is entered into gnomecal while adding a new appointment, gnomecal crashes. A string like "8" instead of "8:00" is considered invalid.
- Gnome-gen-mimedb causes segment violations while processing a mime information file. This is caused by an array overflowing because it is too small.
- Using gnome-libs-1.0.10 and gnome-core-1.0.6, with glib/gtk+- 1.2.3 to change the icon for a launcher on the panel causes the panel to crash with a segment violation.

- After opening the main menu, clicking on the desktop to remove the menu freezes the desktop and needs the X server to be restarted.
- Clicking on "Log Out" selecting "No" and then clicking on "Log Out" again crashes the panel.
- Selecting a png file as a background with the panel background selector kills the selector and the panel itself. This is caused due to the incorrect use of a function.
- Adding the "Fish: Amusements" applet to the panel will cause the panel to crash.
- The panel crashes whenever an applet exits due to a crash or a successful finish.
- Right clicking on an applet 5-10 minutes after using its submenus causes the applet to crash.
- Selecting "Normal Menu" as properties by right clicking on the main menu causes the panel to crash.
- Clicking on the "prev" toolbar option in the year view of the calendar causes gnomecal to crash. This is caused due to using a local copy of a variable instead of a global copy.
- The following sequence of inputs will crash gnomecal

  1) Create a recurring monthly appointment

  2) Restart gnomecal

  3) Go to the second occurrence of the monthly appointment in the daily view

4) Right click on the appointment and choose "Make this appointment movable".

- The following sequence of inputs will crash gnumeric

  1) Start with a new spreadsheet

  2) Input "A1:=B1"

  3) Copy A1 to the clipboard

  4) Paste the contents of the clipboard into A1 on 2nd sheet.

  5) Paste the contents of the clipboard into A2 on 2nd sheet.

  The clipboard is freeing memory which had already been freed.

- Loading an Excel spreadsheet into gnumeric and printing it after changing the font to a smaller one causes gnumeric to crash.

- Resizing a column in gnumeric-0.34 causes it to crash. This is caused due to wrong variable being passed to a function.

- Pressing the "Tab" key in the "define name" dialog or the "File/Summary" dialog causes gnumeric to crash. This is caused due to initializing a variable with the wrong value.

- Performing a shift-left mouse click to select a file after a filter has been applied in the presence of a previously selected file that does not match the filter causes gmc to crash. This is caused due to calling a couple of functions in the wrong order.

- Removing "Name" from Edit->Preferences->Custom View->Displayed Columns causes gmc to crash. This caused by a missing condition check.

- When a copy operation is selected to be done in the background, gmc goes into an infinite loop. This is caused due to a disabled feature being used.
- Using the viewer by right clicking on a text file and then closing it by clicking on the close button on the title bar causes gmc to crash.
- Double clicking on a "tar.gz" file that is lying as an icon on the desktop causes gmc to crash. This is caused due to using the wrong type for a variable.
- Selecting files in a directory, changing the tree selection and then attempting to delete the old directory causes gmc to crash. This is caused due to a missing function call that should reset the selection and also due to wrong arguments being passed to another function.
- Editing the name of a desktop url by double clicking on it and then saving the name by pressing "return" causes gmc to crash.
- Opening a file on the desktop by double-clicking on the icon in the midst of editing its name causes gmc to crash.
- Accessing NFS mounted directories causes I/O errors in gmc.
- Selecting "Rescan Directory" after changing the mime type of a file in the directory causes gmc to crash.

A.3 Fault Study for MySQL

- A query of the type "...where A and (B or C or D or E)..." causes MySQL to crash. The crash is due to a missing null pointer check and a missing initialization statement.
- A complex select query involving a LEFT JOIN crashes the server due to a design error.
- A query with IN (null) in the ON part of an outer join crashes the server.
- An update query from an user without update permission crashes the server.
- A query with "SELECT HIGH_PRIORITY" clause in it crashes the server. This is due to using the wrong field of a structure as the destination.
- "Alter table add column" causes the server to crash.
- Using "alter table" while the table is in use causes the table to be corrupted and crashes the server. This happens even if the table is being properly locked.
- Updating an index with a value that will be found later while scanning the index tree crashes MySQL. This was caused by a design error in the code for updating an index.
- Using the czech character set causes MySQL to crash.
- Issuing a lock command on a table before selecting a database crashes the server.
- Issuing a flush command on a table after locking it causes MySQL to crash.
- Issuing a reverse operation on empty BLOB (string) columns causes the server to crash. This is caused due to a missing check for a null string.

- Sending the content of a field of type BLOB (string) crashes the server. This is caused due to a pointer problem.
- Performing a LEFT JOIN on a table with 0 matching rows causes MySQL to crash with a "divide by 0" error. This is due to a missing null pointer check and floating point roundoff errors.
- MySQL client loses connection to the server when a simple error occurs. This is due to incorrect error handling.
- Performing a query on a table without a selected database while update logging is turned on, crashes the server. This is caused by a null pointer problem.
- MySQL crashes when using "order by" or "distinct" on left-joined tables.
- Connecting from a host that does not have access rights crashes the server.
- Performing an "ORDER BY" operation on zero records from a JOIN operation causes the server to crash on Solaris 2.6. The failure was caused by a bug in the code that caches rows while performing a JOIN operation.
- Flushing tables while some of them are locked causes MySQL to crash.
- Performing a COUNT(DISTINCT col_name) operation on a table with zero rows cause the server to crash. This is due to missing null pointer and empty table checks.
- Performing a GROUP BY operation on a sum function causes MySQL to crash. This caused by a missing condition check.
- A query selecting distinct fields from a column of type text where one of the values is "NULL" crashes the server.

- Accessing the server through the "C" programming language api caused the server to crash. This was due to a missing call to an initialization function.
- The sum() function always returns a zero or otherwise false data.
- Performing a delete operation on a locked table without a WHERE clause crashes the server. This is due to a missing check for NULL condition value.
- MySQL does not allow a field to be declared of type BLOB without a length field. This was caused by a wrong bounds check.
- Lock contention causes the application to use up all processor time. This is in spite of no lock commands issued by the user. This is caused by some unnecessary synchronization calls.
- MySQL is caught in an infinite loop when a thread tries to throw a SEGV exception which is being masked. So the instruction is forever being restarted.
- MySQL crashes when a client connects to it using TCP/IP. This is due to a wrong value being passed to a function.
- MySQL crashes when it is asked to sort records while checking database integrity. This is caused due to a missing initialization statement.
- Performing an optimization query on a table causes MySQL to crash. This is caused by a missing initialization statement.
- MySQL dies on the Linux/Alpha platform when a SELECT operation is done on a text field.
- Switching databases after trying to use a table or database without access permissions crashes the server.
- Removing an index from a heap table crashes the server.

- Running a SELECT query without selecting a database crashes the server.
- Submitting an invalid command crashes the server. It should just return an error. This is caused by a missing check for a null table pointer.

BIBLIOGRAPHY

[Avizienis85] Algirdas Avizienis. The N-Version Approach to Fault-Tolerant Software. *IEEE Transactions on Software Engineering*, SE-11(12):1491–1501, December 1985.

[Barton90] James H. Barton, Edward W. Czeck, Zary Z. Segall, and Daniel P. Siewiorek. Fault injection experiments using FIAT. *IEEE Transactions on Computers*, 39(4):575–582, April 1990.

[Birman91] Kenneth Birman, Andre Schiper, and Pat Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.

[Borg89] Anita Borg, Wolfgang Blau, Wolfgang Graetsch, Ferdinand H errman, and Wolfgang Oberle. Fault Tolerance Under UNIX. *ACM Transactions on Computer Systems,* 7(1):1–24, February 1989.

[Brilliant90] Susan S. Brilliant, John C. Knight, and Nancy G. Leveson. Analysis of Faults in an N-Version Software Experiment. *IEEE Transactions on Software Engineering*, SE-16(2):238–247, February 1990.

[Chen96] Peter M. Chen, Wee Teck Ng, Subhachandra Chandra, Christopher M. Aycock, Gurushankar Rajamani, and David Lowell. The Rio File Cache: Surviving Operating System Crashes. In *Proceedings of the 1996 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 74–83, October 1996.

[Costa96] Manuel Costa, Paulo Guedes, Manuel Sequeira, and Nuno Neves a nd Miguel Castro. Lightweight Logging for Lazy Release Consistent Distributed Shared Memory. *Operating Systems Design and Implementation (OSDI),* October 1996.

[Costa00] Diamantino Costa, Tiago Rilho, and Henrique Madeira. Joint Evaluation of Performance and Robustness of a COTS DBMS through Fault-Injection. In *Proceedings of the 2000 Symposium on Fault-Tolerant Computing (FTCS)*, June 2000.

[Elnozahy99] E. N. Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. Technical Report CMU-CS-99-148, Carnegie Mellon University, June 1999.

[Gray86] Jim Gray. Why do computers stop and what can be done about it? In *Proceedings of the 1986 Symposium on Reliability in Distributed Software and Database Systems*, pages 3–12, January 1986.

[Gray91] Jim Gray and Daniel P. Siewiorek. High-Availability Computer Systems. *IEEE Computer*, 24(9):39–48, September 1991.

[Gray93] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, Inc., 1993.

[Hsueh97] Mei-Chen Hsueh, Timothy K. Tsai, and Ravishankar K. Iyer. Fault Injection Techniques and Tools. *IEEE Computer*, 30(4):75–82, April 1997.

[Huang93] Yennun Huang and Chandra R. Kintala. Software Implemented Fault Tolerance: Technologies and Experience. In *Proceedings of the 1993 International Symposium on Fault-Tolerant Computing*, pages 2–9, June 1993.

[Huang95] Y. Huang, C.M.R. Kintala, N. Kolettis, and N.D. Fulton. Software Rejuvenation: Analysis, Module and Applications. In *Proceedings of the 1995 International Symposium on Fault-Tolerant Computing*, pages 381–390, June 1995.

[Johnson87] David B. Johnson and Willy Zwaenepoel. Sender-Based Message Logging. In *Proceedings of the 1987 International Symposium on Fault-Tolerant Computing*, pages 14–19, July 1987.

[Kanawati95] Ghani A. Kanawati, Nasser A. Kanawati, and Jacob A. Abraham. FERRARI: A Flexible Software-Based Fault and Error Injection System. *IEEE Transactions on Computers*, 44(2):248–260, February 1995.

[Kao93] Wei-Lun Kao, Ravishankar K. Iyer, and Dong Tang. FINE: A Fault Injection and Monitoring Environment for Tracing the UNIX System Behavior under Faults. *IEEE Transactions on Software Engineering*, 19(11):1105–1118, November 1993.

[Knight86] John C. Knight and Nancy G. Leveson. An Experimental Evaluation of the Assumption of Independence in Multiversion Programming. *IEEE Transactions on Software Engineering*, SE-12(1):96–109, January 1986.

[Kropp98] Nathan P. Kropp, Philip J. Koopman, and Daniel P. Siewiorek. Automated Robustness Testing of Off-the-shelf Software Components. In *Proceedings of the 1998 International Symposium on Fault-Tolerant Computing*, pages 230–239, June 1998.

[Lee93] Inhwan Lee and Ravishankar K. Iyer. Faults, Symptoms, and Software Fault Tolerance in the Tandem GUARD IAN Operating System. In *International Symposium on Fault-Tolerant Computing (FTCS)*, pages 20–29, 1993.

[Lowell97] David E. Lowell and Peter M. Chen. Free Transactions with Rio Vista. In *Proceedings of the 1997 Symposium on Operating Systems Principles*, pages 92–101, October 1997.

[Lowell99] David E. Lowell. *Theory and Practice of Failure Transparency*. University of Michigan, 1999.

[Madeira94] Henrique Madeira and Joao G. Silva. Experimental Evaluation of the Fail-Silent Behavior in Computers Without Error Masking. In *Proceedings of the 1994 Symposium on Fault-Tolerant Computing (FTCS)*, June 1994.

[Mourad85] Samiha Mourad and Dorothy Andrews. The Reliability of the IBM MVS/XA Operating System. In *Proceedings of the 1985 Symposium on Fault-Tolerant Computing (FTCS)*, June 1985.

[Netcraft99] The Netcraft Web Server Survey. At *http://www.netcraft.com/survey/*

[Ng97] Wee Teck Ng and Peter M. Chen. Integrating Reliable Memory in Databases. In *Proceedings of the 1997 International Conference on Very Large Data Bases (VLDB)*, pages 76–85, August 1997.

[Ng99] Wee Teck Ng. *Design and Implementation of Reliable Main Memory*. University of Michigan, 1999.

[Pradhan95] Dhiraj K. Pradhan. *Fault-Tolerant Computer System Design*. Prentice-Hall, Inc., 1995.

[Randell75] Brian Randell. System Structure for Software Fault Tolerance. *IEEE Transactions on Software Engineering*, 1(2):220–232, June 1975.

[Schneider84] Fred B. Schneider. Byzantine Generals in Action: Implementing Fail-Stop Processors. *ACM Transactions on Computer Systems*, 2(2):145–154, May 1984.

[SiteMetrics98] Internet Server Survey. At *http://www.sitemetrics.com/serversurvey/ss_98_q3/revseg.htm*.

[Stonebraker87] M. Stonebraker. The design of the POSTGRES storage system. In *Proceedings of the 1987 International Conference on Very Large Data Bases*, pages 289–300, September 1987.

[Savage97] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems*, 15(4):391–411, November 1997.

[Strom85] Robert E. Strom and Shaula Yemini. Optimistic Recovery in Distributed Systems. *ACM Transactions on Computer Systems,* 3(3):204–226, August 1985.

[Sullivan91] Mark Sullivan and R. Chillarege. Software Defects and Their Impact on System Availability–A Study of Field Failures in Operating Systems. In *Proceedings of the 1991 International Symposium on Fault-Tolerant Computing*, June 1991.

[Sullivan92] Mark Sullivan and Ram Chillarege. A Comparison of Software Defects in Database Management Systems an d Operating Systems. In *Proceedings of the 1992 International Symposium on Fault-Tolerant Computing*, pages 475–484, July 1992.

[TPC90] TPC Benchmark B Standard Specification. Technical Report, Transaction Processing Performance Council, August 1990.

[Wang93] Yi-Min Wang, W. Kent Fuchs, and Yennuan Huang. Progressive Retry for Software Error Recovery in Distributed Systems. In *Proceedings of the 1993 Symposium on Fault-Tolerant Computing*, June 1993.

[Weller93] Edward F. Weller. Lessons from Three Years of Inspection Data. *IEEE Software*, 10(5):38–45, September 1993.