Proceedings of the 2000 International Conference on Dependable Systems and Networks /
Symposium on Fault-Tolerant Computing (FTCS)

Whither Generic Recovery from Application Faults? A Fault Study using Open-Source Software

Subhachandra Chandra and Peter M. Chen
Computer Science and Engineering Division
Department of Electrical Engineering and Computer Science
University of Michigan
{schandra,pmchen}@eecs.umich.edu
http://www.eecs.umich.edu/Rio

Abstract

This paper tests the hypothesis that generic recovery techniques, such as process pairs, can survive most application faults without using application-specific information. We examine in detail the faults that occur 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 classify faults based on how they depend on the operating environment. We find that 72-87% of the faults are independent of the operating environment and are hence deterministic (non-transient). Recovering from the failures caused by these faults requires the use of application-specific knowledge. Half of the remaining faults depend on a condition in the operating environment that is likely to persist on retry, and the failures caused by these faults are also likely to require application-specific recovery. Unfortunately, only 5-14% of the faults were triggered by transient conditions, such as timing and synchronization, that naturally fix themselves during recovery. Our results indicate that classical application-generic recovery techniques, such as process pairs, will not be sufficient to enable applications to survive most failures caused by application faults.

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 software faults and surviving the ones that remain is therefore an important challenge for the fault-tolerance community.

One key component of this quest to avoid and survive software faults is understanding what types of faults occur in released applications. This understanding can help us develop recovery techniques for surviving software faults and develop new languages and tools for avoiding software faults in the future. 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 paper, 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 paper is to provide information to guide research on how to survive application faults. In particular, we wish to test the hypothesis that generic (i.e. not application-specific) 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. For this software, we find that only a small fraction (5-14%) of faults are triggered by transient conditions (so-called Heisenbugs) which can be survived with generic recovery techniques such as rollback and retry.

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 paper, 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 paper is limited to faults that occur in application software and how often a recovery system can recover from these faults.

We categorize techniques for recovering from software faults as either application-specific recovery or application-generic recovery. Both styles of recovery involve redundancy. 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 [Randell75]. 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. Instead, application-generic recovery typically relies on time redundancy. For example, process pairs [Gray86] 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. 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 paper is to test this hypothesis by analyzing bug reports and bug fixes of several large, widely used, open-source programs.

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.

It is important to note that given a fixed operating environment, a set of concurrent, sequential processes is completely deterministic [Dijkstra72]. 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 (Section 5.1). 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. 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.

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. There are too many environment-independent faults to describe each one fully in the limited space available, so we give descriptions of several representative ones.

5.1. Apache

Table 1 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.

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

Table 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.

- 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.

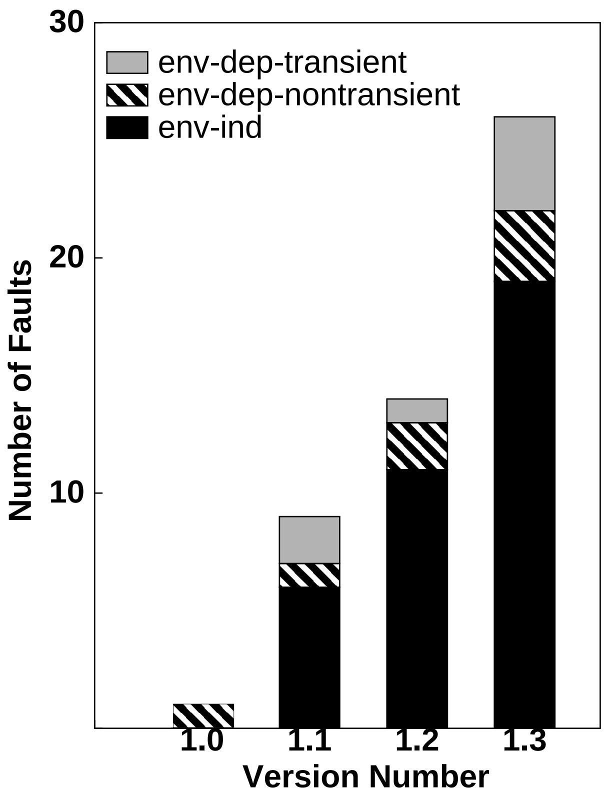

Figure 1: **Distribution of faults for Apache over software releases.**

- 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 1 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.

5.2. Gnome

Table 2 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 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.

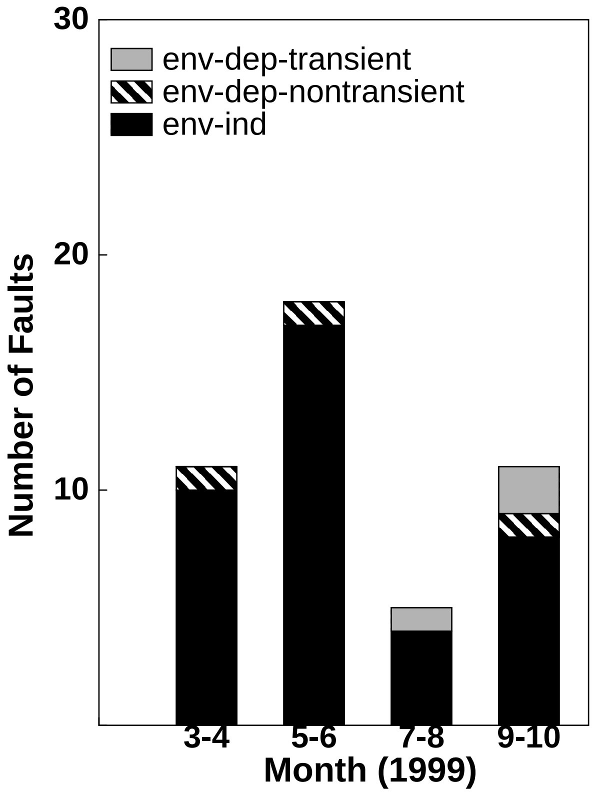

Figure 2: **Distribution of faults for GNOME over time.**

Figure 2 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.

5.3. MySQL

Table 3 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.

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

Table 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.

- 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.
- 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 3 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.

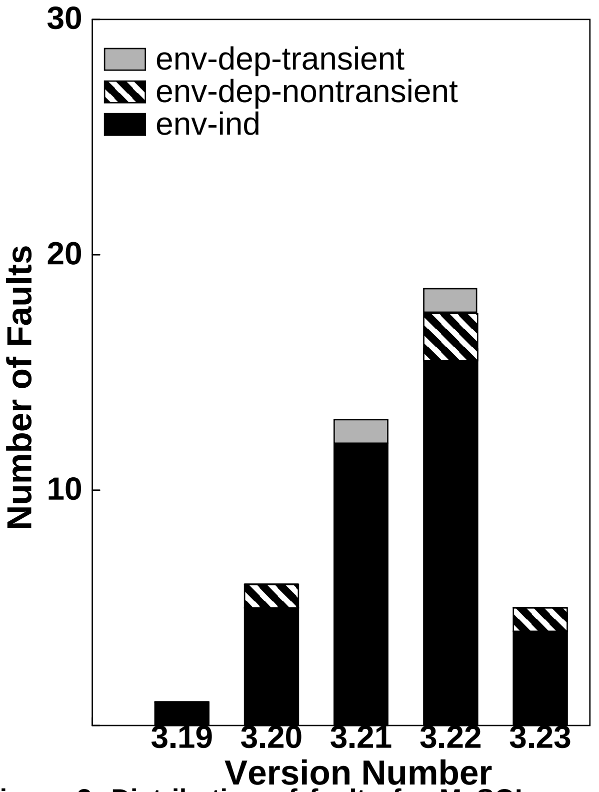

Figure 3: **Distribution of faults for MySQL over software releases**.

5.4. 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.

6. 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.

6.1. 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.

6.2. 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].

6.3. 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].

7. 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.

8. Conclusions and Future Work

In this paper, we have tested the hypothesis that generic recovery techniques, such as process pairs, can survive most software faults without using application-specific information. 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.

In the future, we hope to implement applications like Apache and MySQL using various fault-tolerant techniques and test how well they recover from the bugs reported in error logs. This will allow us to verify the conclusions we drew from information in the error logs; for example, it will allow us to verify that a fault did not depend on an unreported environmental condition.

9. Acknowledgments

We are grateful to the open-source community for providing useful, robust software and for providing a wealth of information to help make future programs even more useful and robust. We thank the program committee and reviewers, particularly Ravi Iyer and Chandra Kintala, for their insightful feedback.

This research was supported in part by NSF grant MIP-9521386, NSF CAREER Award MIP-9624869, IBM University Partnership Program #19981020024, Intel Technology for Education 2000, and AT&T Labs.

10. References

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

[Dijkstra72] Edsger W. Dijkstra. Hierarchical Ordering of Sequential Processes. Technical report, Academic Press, 1972. In Operating Systems Techniques, Hoare and Perrott eds.,.

[Elnozahy99] E.N. Elnozahy, L.Alvisi, Y.-M. Wang, and D. 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.

[Huang93] Y. Huang and C. 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. Kintala, N. Kolettis, and N. Fulton. Software Rejuvenation: Analysis, Module and Applications. In *Proceedings of the 1995 International Symposium on Fault-Tolerant Computing*, pages 381–390, June 1995.

[Kropp98] N. P. Kropp, P. J. Koopman, and D.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] I. Lee and R. Iyer. Faults, Symptoms, and Software Fault Tolerance in the Tandem GUARDIAN Operating System. In *International Symposium on Fault-Tolerant Computing (FTCS)*, pages 20–29, 1993.

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

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

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

[Sullivan91] M. 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] M. Sullivan and R. 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.

[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.