28th International Symposium on Fault-Tolerant Computing, June 1998

How Fail-Stop are Faulty Programs?

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

Most fault-tolerant systems are designed to stop faulty programs before they write permanent data or communicate with other processes. This property (halt-on-failure) forms the core of the fail-stop model. Unfortunately, little experimental data exists on whether or not program failures follow the fail-stop model. This paper describes a tool, based on the SimOS complete-machine simulator, that can trace how faults propagate through memory, disk, and functions. Using this tool on the Postgres database system, we conduct a controlled experiment to measure how often faulty programs violate the fail-stop model. We find that a significant number of faults (7%) violate the fail-stop model by writing incorrect data to stable storage before halting. We then apply Postgres' transaction mechanism to undo recent changes before a crash and find that transactions reduce fail-stop violations by a factor of 3.

1. Introduction

Building fault-tolerant systems is a difficult task. A malfunctioning 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 processes. 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].

General mechanisms exist to make systems fail-stop. For example, Schneider describes a system where each program runs on multiple processors [Schneider84]. The processors vote before writing to stable storage and halt if their results disagree. These types of mechanisms are used primarily to tolerate independent hardware faults. Unfortunately, software bugs are currently the dominant cause of failures, accounting for 60-90% of all failures [Gray91].

There are two major ways to make programs fail-stop in the presence of software bugs, both of which use replication. The first technique is *N-version programming* [Avizienis85], 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. 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 a different form of replication 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. *Recovery blocks* are a fault-tolerant scheme that uses error checking at the end of each block of code (a recovery block) [Randell75]. When one recovery block fails an error check, the state of the system is rolled back to the beginning of the recovery block and another version of the recovery block is used. *Process pairs* [Gray86] are similar to recovery blocks. Whereas recovery blocks invoke a different version of the code, process pairs re-invoke the same code (possibly on a different computer) and hope the bug does not repeat itself on the second try. Process pairs

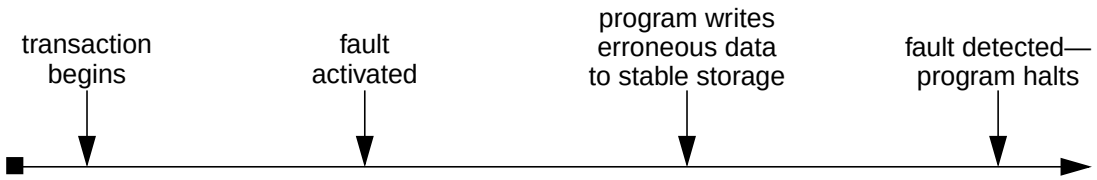

Figure 1: Timeline of a program failure. Without transactions, the above timeline violates the fail-stop model, because the program writes erroneous data to stable storage before detecting the fault and halting. Transactions undo the changes to stable storage performed during the current transaction and hence make the system appear to have halted at the beginning of the current transaction. When using transactions, the program will be fail-stop if all erroneous writes occur after the beginning of the current transaction.

work well in practice because most bugs are non-deterministic (i.e. Heisenbugs).

Both recovery blocks and process pairs use *atomic transactions* [Gray93] to roll the state of the system back to the beginning of the block of code. Transactions help make the program fail-stop by undoing recent changes to stable storage (Figure 1). Specifically, transactions undo the changes to stable storage performed during the current transaction and hence make the system appear to have halted at the beginning of the current transaction. When using transactions, the program will be fail-stop if all erroneous writes occur after the beginning of the current transaction.

Although most applications use error checking to approximate the fail-stop model, we know of no study that quantifies experimentally how well this works in practice. In this paper, we address the question: “For a large software system, how often do software bugs cause erroneous data to be written to stable storage?”. Using the Postgres database as our target software system, we measure the frequency of fail-stop violations both with and without transactions. We find that 7% of software bugs violate the fail-stop model without transactions. After using Postgres’ transaction mechanism to roll back recent, uncommitted changes, we find that only 2% of software bugs violate fail-stop.

Our secondary goal in this paper is to present a tool that can monitor fault propagation through the program and measure how quickly the system detects the fault and halts.

2. A tool to trace fault propagation

2.1. High-level description

Our goal is to monitor the effects of the injected faults on the program’s behavior. To measure the effect of a fault, we run the program twice: once without faults (*good*) and once with faults (*buggy*) (Figure 2). Periodically (e.g. points A/A’ and B/B’ in Figure 2) we compare the states (memory and disk contents) of the two runs and note the differences.

Two factors complicate this simple strategy. First, the runs must be perfectly deterministic and repeatable. Otherwise, differences between the two runs might be due to non-determinism instead of faults. Second, the two runs must be at the same point in their execution whenever they are compared. Matching execution points is difficult if the fault affects the control flow of the program. Sections 2.2 and 2.3 describe our strategy for overcoming these two complications.

2.2. Achieving repeatability

We conduct our experiments on a simulator to make the runs deterministic and repeatable. Using simulators to conduct fault experiments has been infeasible in the past. Some past simulators have been too slow to run large software programs; other simulators achieved high speed but did not simulate the machine in enough detail to run system software.

Our tool is based on SimOS, a complete-machine simulator developed at Stanford University [Rosenblum95]. SimOS (Digital Alpha version) simulates an entire Digital Alpha workstation at enough detail to run unmodified application programs and a slightly modified Digital Unix kernel. The modifications are limited to low-level device drivers, such as disk drivers and the system console. Thus the applications and most of the subsystems in the kernel cannot distinguish between running on a real workstation and running on top of SimOS. Because SimOS simulates the complete machine, our tool can monitor fault propagation through both application and system software. In addition, SimOS uses advanced techniques such as binary translation to provide very fast simulation.

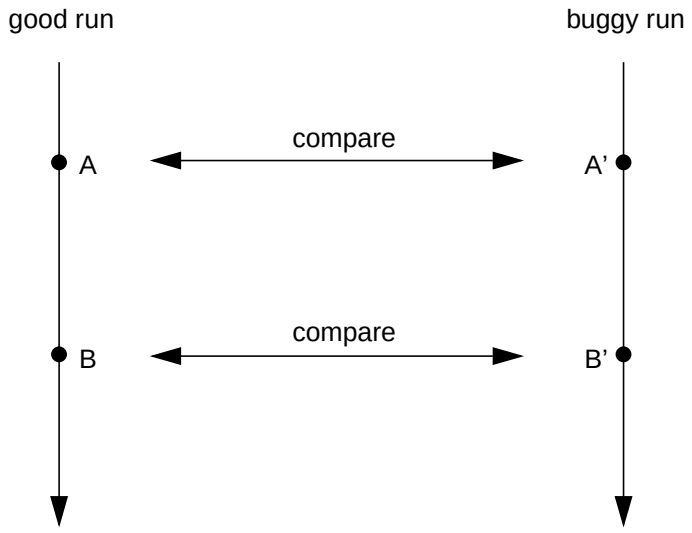

Figure 2: High-level description of tool. To measure the effect of a fault, we run the program twice: once without faults (good), once with faults (buggy). Periodically (e.g. points A and B), we compare the states (memory and disk contents) of the two runs and note the differences.

We run SimOS in a mode that achieves 1/10 the speed of actual hardware.

Our tool takes advantage of two additional SimOS features: annotations and checkpoints. Annotations are Tcl scripts that can be run at specified simulation events without perturbing the program's execution. We use SimOS annotations to control the simulation mode and to start application programs. We use SimOS's checkpoint mechanism in two ways. First, we use checkpoints to collect relevant machine state (memory and disk contents). Second, we use checkpoints to speed up simulation by starting from a checkpoint taken after the simulated machine has booted.

2.3. Matching execution points

Determining the effect of a fault means comparing the states (checkpoints) of a good run and a buggy run. To do this comparison accurately, the two runs must be at the same point in their execution. For certain types of faults, the good run and the buggy run proceed in lockstep; that is, each instruction in the good run has a corresponding instruction at the same instruction address in the buggy run. For example, a fault that causes an bank account balance to be off by $1 will not affect the execution path of the program. For these types of faults, execution points can be matched very easily—a point in the buggy run matches a point in the good run if both have executed the same number of instructions up to that point.

Unfortunately, many faults directly or indirectly alter the execution path of the program. For example, a loop may execute one fewer iteration in the buggy run than in the good run. Or a branch instruction might be taken in the buggy run but not in the good run, as shown in Figure 3. Often, the execution paths of buggy runs and good runs will diverge for a number of instructions (e.g. one makes an extra system call), then converge again. We would like to find these points of divergence and convergence so we can match execution points during periods of convergence. Meaningful comparisons of the checkpoints may then be done at matching execution points. In Figure 3, execution points A/A' match, as do points C/C' and D/D'. In general, it may be impossible to match certain points in the execution of the two runs. In Figure 3, for example, execution point B' in the buggy run has no corresponding execution point in the good run.

Our heuristic for finding periods of convergence is to match traces of instruction addresses. If two runs execute the same series of instructions (i.e. instructions at the same addresses), we assume that each instruction in one run corresponds to the same instruction in the other run. We use the Unix diff utility to match these traces. Diff preserves the ordering of an address trace; if execution points C and C' are found to match in Figure 3, then D' cannot match an instruction above C. This method works well for our method of injecting faults, which never inserts or deletes instructions and hence preserves instruction addresses between the good program and the buggy program.

We use two passes to collect these traces and checkpoints. In the first pass, we run the good program and the buggy program and collect traces of instruction addresses. We then diff the two traces, calculate all matching execu-

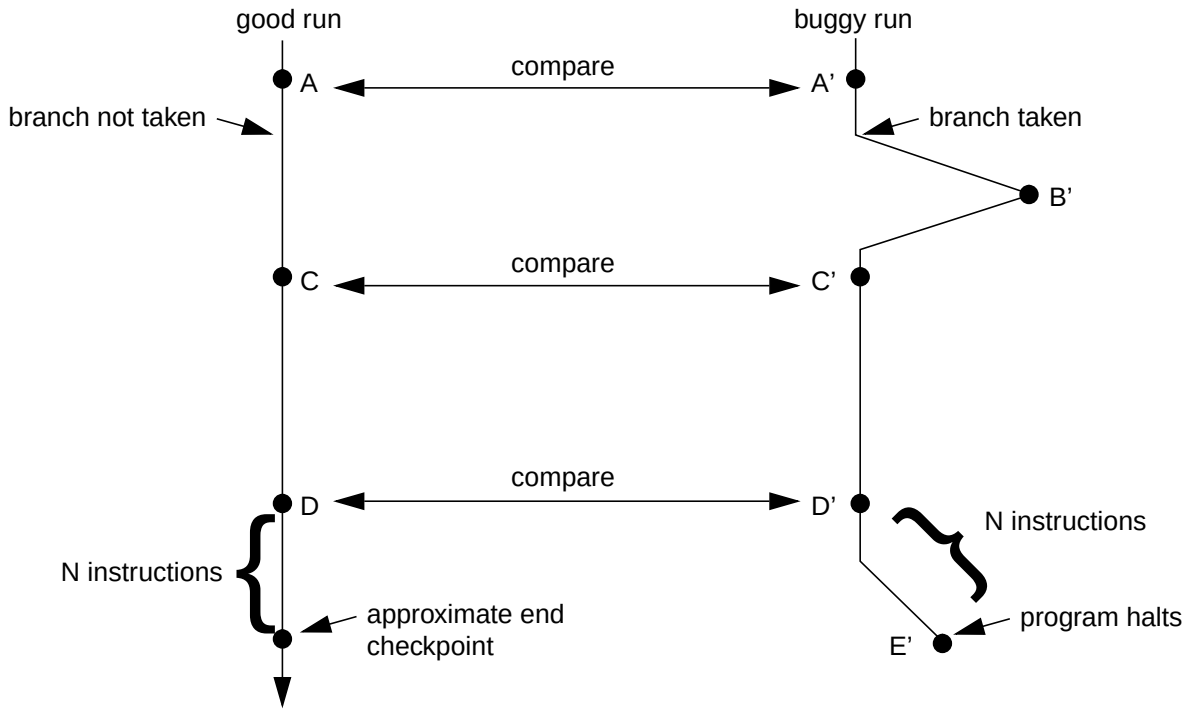

Figure 3: Matching execution points. Many faults alter the execution path of the program. Section 2.3 describes how we match the execution points of good and buggy runs.

tion points, and choose a number of points to take checkpoints. In the second pass, we re-run the good program and the buggy program and take checkpoints at the points chosen during the first pass.

Recording all instruction addresses generates extremely long traces. We reduce the size of the trace by only recording the address of every 256th instruction in the static program (i.e. instructions whose word address is divisible by 256). This works well in practice because periods of convergence or divergence tend to be thousands of instructions in length. Note that sampling every 256th instruction *executed* would not work because one extra instruction executed would offset the entire trace.

2.4. Detecting corruption

The prior section describes how we match execution points between the good run and the buggy run. The result of the second pass is a sequence of checkpoint pairs. Each pair consists of a checkpoint of the good run and a checkpoint (at the matching execution point) of the buggy run. By comparing the two checkpoints in the pair, we can enumerate memory and disk corruption caused by the fault. We can examine how this list of corruptions grows over the run to trace fault propagation.

Our tool also remembers which application functions last wrote to each memory location. We use this to show how faults propagate through application modules. At each checkpoint, our tool notes which application function last wrote each corrupted memory location; this yields a list of functions whose behavior was corrupted during the preceding interval. For example, Table 1 shows a series of three snapshots of the Postgres function space taken at various points in a run.

2.5. Detecting fail-stop violations

The focus of this paper is detecting fail-stop violations. Fail-stop violations are defined by the application files on disk at the end of the buggy run (E' in Figure 3). Finding a matching execution point to compare with the end of the buggy run is non-trivial. Often, no point in the execution of the good run matches the end of the buggy run, because the buggy run usually ends in some error routine (or system signal handler, e.g. core dump). We take a two-pronged approach to find an execution point in the good run that approximately matches the end of the buggy run.

First, we take a checkpoint in the good run at a point that roughly matches the end of the buggy run. To do this, we count the number of instructions executed in the buggy run after the last matching execution points (N instructions). We then take an approximate-end checkpoint in the good run N instructions after the last match.

Snapshot 1 (at fault activation)	Snapshot 2 (middle of run)	Snapshot3 (when program crashes)
PostgresMain	PostgresMain	PostgresMain
	FileWrite	FileWrite
	ProcessQueryDesc	ProcessQueryDesc
	mdcommit	mdcommit
	mywrite2	mywrite2
	showatts	showatts
		_bt_getroot
		_bt_search
		_bt_binsrch
		ExclusiveLock
		ExclusiveUnlock
		LockRelease
		SingleLockPage
		hash_search

Table 1: Propagation of fault through Postgres functions. Each snapshot lists the functions whose behavior has been corrupted at that point in the run. By viewing a sequence of snapshots, our tool can describe the fine-grained propagation of the fault through the application functions. This particular run was for an interface error (Section 3) and did not violate the fail-stop model.

Second, we categorize a buggy run as obeying the fail-stop model if its final disk contents match *any* of the checkpoints in the good run. For example, if the disk contents at the end of the buggy run match the disk contents at checkpoint C in Figure 3, it is as though the buggy run had a fail-stop crash at point C. For 85% of the runs in Section 4, the disk contents at the end of the buggy run matched the approximate-end checkpoint; 8% of the runs matched an earlier checkpoint, and 7% were fail-stop violations that did not match any checkpoint.

Detecting fail-stop violations with transactions is similar to detecting fail-stop violations without transactions. The only difference is that the comparison of disk contents must use the *transactional state* of the files. The transactional state of a database is the state of the database after in-progress transactions have been aborted. Aborting current transactions rolls the disk state back to the point before the current transaction began. Because of this rollback process, runs that violate fail-stop with transactions are usually a subset of the runs that violate fail-stop without transactions.

3. Workload and fault models

We use the Postgres95 database management system developed at U.C. Berkeley as the software system in our experiments [Stonebraker87]. While our results are specific to the software system being tested, we believe Postgres is a good example of a large software system with abundant error checking.

As with all databases, Postgres can abort the transactions in progress at the time of the crash. We use this feature to measure how many fail-stop violations are masked by the transaction mechanism. We drive Postgres with a transaction workload based on TPC-B [TPC90]. The default configuration of Postgres prints a warning but does not stop if it detects minor errors. To make Postgres more fail-stop, we halt the program immediately after any warning message.

Faults injected into Postgres form the second part of the workload. Our primary goal in designing these faults is to generate a *wide variety* of database 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 hardware faults such

as flipping bits in memory to high-level software faults such as memory allocation errors. We classify injected faults into three categories: bit flips, low-level software faults, and high-level software faults. Unless otherwise stated, we inject 5 faults for each run to increase the chances that a fault will be triggered. We only consider runs in which a fault was activated.

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. They are the least realistic of our bugs, however. It is difficult to relate a bit flip with a specific error in programming, and most hardware bit flips would be caught by parity on the data or address bus.

The second category of fault changes individual instructions in the database text segment. These faults are intended to approximate the assembly-level manifestation of real C-level programming errors [Kao93]. We corrupt assignment statements by changing the *source* or *destination* register. We corrupt conditional constructs by deleting *branches*. We also delete *random instructions* (both branch and non-branch) by replacing them with NOPs.

The last and most extensive category of faults imitate specific programming errors in the database [Sullivan91]. These are targeted more at specific programming errors than the previous fault category. We inject an *initialization* fault by deleting instructions responsible for initializing a variable at the start of a procedure [Kao93, Lee93]. We inject *pointer* corruption by 1) finding a register that is used as a base register of a load or store and 2) deleting the most recent instruction before the load/store that modifies the base register [Sullivan91, Lee93]. We do not corrupt the stack pointer register, as this is used to access local variables instead of as a pointer variable. We inject an *allocation management* fault by modifying the database's malloc procedure to occasionally free the previously allocated block of memory. Malloc is set to inject this error every 0-5000 times it is called. We inject a *copy overrun* fault by modifying the database's bcopy procedure to occasionally increase the number of bytes it copies. The length of the overrun was distributed as follows: 50% corrupt one byte; 44% corrupt 2-1024 bytes; 6% corrupt 2-4 KB. This distribution was chosen by starting with the data gathered in [Sullivan91] and modifying it somewhat according to our specific platform and experience. bcopy is set to inject this error every 0-5000 times it is called. We inject *off-by-one* errors by changing conditions such as > to >=, < to <=, and so on. We mimic common *synchronization* errors by randomly causing the procedures that acquire/free a lock to return without acquiring/freeing the lock. We inject *interface errors* by corrupting one of the arguments passed to a procedure.

Fault injection cannot mimic the exact behavior of all real-world database system crashes. However, the wide variety of faults we inject (13 types), the random nature of the faults, and the number of runs we performed (650) give us confidence that our experiments cover a wide range of real-world faults. Table 2 shows examples of how real-world programming errors can manifest themselves as the faults we inject in our experiments.

4. Results

The results in this paper were collected over several machine-months of fault-injection experiments. We only consider runs in which a fault was activated (50 of these runs for each type of fault). In 55% of the runs, Postgres or the operating system detected the error and halted the program. In 44% of the runs, Postgres ran to completion. In the remaining 1% of the runs, Postgres enters an infinite loop; we killed these runs after 100 million instructions (normal runs finish after 57 million instructions).

The main goal of our paper is to measure how often program failures violate the fail-stop model. Table 3 shows that fail-stop violations occur for a significant fraction of the runs (7%). In these runs, Postgres detects the error and halts *after* stable storage has been corrupted by the fault. This frequency is disturbingly high, because it implies that any fault-tolerant protocol that depends on the program being fail-stop will not work 7% of the time.

Transactions are able to make a program more fail-stop by undoing recent changes to stable storage. The right-hand column of Table 3 shows that transactions are quite effective at reducing fail-stop violations, lowering the frequency of fail-stop violations from 7% to 2%. This implies that most erroneous writes to stable storage occur during the transaction that detected the error. It also implies that errors tend to affect the data being modified by the current transaction, because modifications outside the current transaction are not undone by the transaction mechanism.

Figure 4 plots the fault detection latency (how long the program runs after the fault is activated) versus the number of corrupted memory words at the end of the run. Runs that violate the fail-stop model are circled. Runs that violate the fail-stop model with transactions are represented by filled-in circles.

Table 4 shows the median fault latencies for various types of runs: all runs, runs that violate fail-stop, runs that violate fail-stop when using transactions. As expected, runs that continue long after the fault is activated have more opportunity to damage permanent data. Hence the

Fault Type	Example of Programming Error	
	Correct Code	**Faulty Code**
destination reg.	**numFreePages** = count(freePageHeadPtr)	**numPages** = count(freePageHeadPtr)
source reg.	numPages = **physicalMemorySize** / pageSize	numPages = **virtualMemorySize** / pageSize
delete branch	**while** (flag) {body}	**if** (flag) {body}
delete random inst.	for (i=0; i<10; i++,**j++**) {body}	for (i=0; i<10; i++) {body}
initialization	function () {int i**=0**; ...}	function () {int i; ...}
pointer	ptr = ptr->next**->next**;	ptr = ptr->next;
allocation	ptr = malloc(N); use ptr; use ptr; **free(ptr);**	ptr = malloc(N); use ptr; **free(ptr)**; use ptr
copy overrun	for (i=0; i<**sizeUsed**; i++) {a[i] = b[i]};	for (i=0; i<**sizeTotal**; i++) {a[i] = b[i]};
off-by-one	for (i=0; i**<**size; i++)	for (i=0; i**<=**size; i++)
synchronization	**getWriteLock;** write(); **freeWriteLock;**	write();
interface error	insert(**buf**, index);	insert(**buf1**,index);

Table 2: Relating faults to programming errors. This table shows examples of how real-world programming errors can manifest themselves as the faults we inject in our experiments. None of the errors shown above would be caught during compilation.

Fault Category	# of Runs	Fail-stop Violations	Fail-stop Violations with Transactions
heap	50	5	0
stack	50	1	1
destination reg.	50	6	2
source reg.	50	2	1
delete branch	50	3	2
delete random inst.	50	6	3
initialization	50	1	0
pointer	50	3	0
allocation	50	3	0
copy overrun	50	4	1
off-by-one	50	8	3
synchronization	50	0	0
interface error	50	4	2
Total	**650**	**46 (7%)**	**15 (2%)**

Table 3: Frequency of fail-stop violations in Postgres. This table shows that fail-stop violations occur about 7% of the time without transactions. By undoing recent changes to stable storage, transactions are able to reduce fail-stop violations by a factor of 3.

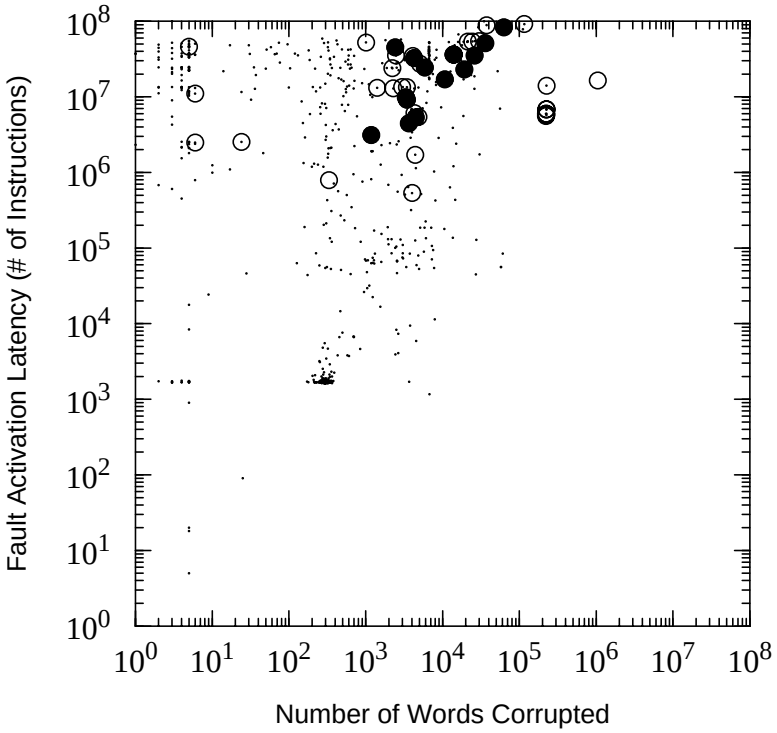

Figure 4: Fault detection latency versus amount of corruption. Each run is shown as a dot. Fail-stop violations are circled. Fail-stop violations with transactions are shown as filled-in circles.

	Median Fault Latency
All runs	4.3 million instructions
Fail-stop violations	13.8 million instructions
Fail-stop violations with transactions	24.5 million instructions

Table 4: Median fault latencies. Runs that violate fail-stop tend to have longer fault latency than average, and only very long latency faults cause fail-stop when using transactions. A complete run of Postgres on our workload executes 57 million instructions.

fault latency for runs that violate the fail-stop model is three times as high as the overall average. Using transactions prevents fail-stop violations in most cases. Fail-stop violations with transactions occur only for very long latency faults in which the program continues beyond the transaction that wrote corrupted data.

5. Related work

Many prior studies have used software fault injection; see [Iyer95] for 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 trace how errors propagate during the crash, and we know of no prior study that measures how often program failures violate the fail-stop model. More work is clearly needed in this area, especially in light of how many fault-tolerant systems are based on the fail-stop model.

The most relevant work to this paper is the FINE fault injector and monitoring environment [Kao93]. FINE uses software to emulate hardware and software bugs into the Unix operating system and monitors how the fault propagates through the kernel. To trace fault propagation, a

FINE user manually specifies key variables at the beginning of each experiment. These variables are printed out at user-specified probe points. Traces from faulty and good runs can be compared to identify how the fault is propagating. Our tool differs from FINE in several key ways:

- We monitor all memory locations, while FINE traces only those memory locations specified by the user.
- We monitor disk corruption, which is crucial to detecting fail-stop violations.
- By matching execution timelines (Section 2.3), we are able to match corresponding checkpoints between good and faulty runs.

In summary, FINE is appropriate for tracing fault propagation at a high level. The user of FINE needs intimate knowledge of how the system works in order to specify the variables and probe points. Our tool traces fault propagation at a low level and is able to monitor comprehensively the entire system state.

A prior paper from the Rio project measures database corruption 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]. The prior study compares the transactional state of the data (after changes from in-progress transactions are undone), so these results correspond to the right-most column of Table 3 (fail-stop violations with transactions). Our results match those in the prior study: 2% of crashes violate fail-stop when transactions are used. The methodology used in the current paper is much more general than that used in prior paper, however. We can measure fail-stop violations for arbitrary applications at arbitrary points in the computation. The method used in the prior study works only when correct answers can be pre-determined for each point in the computation. We also compare results with and without transactions to determine how effectively transactions hide fail-stop violations, something that could not be easily done in [Ng97].

6. Conclusions and future work

While many systems depend on programs being fail-stop, no prior study has measured how often programs actually stop before corrupting permanent data. We have described a tool that can trace how faults propagate through memory, disk, and functions. Using this tool on the Postgres database system, we have shown that a significant number of faults (7%) violate the fail-stop model. Transactions undo recent changes to stable storage and are able to reduce the number of fail-stop violations by a factor of 3. Based on these findings, we recommend that more system designers use transactions to increase their system's reliability.

This paper evaluates the fail-stop property on a single application. In the future, we plan to broaden the scope of our conclusions by repeating this study on more applications. In particular, we plan to measure how often distributed applications violate the fail-stop model by sending corrupted messages to other processes. We also plan to use our tool's ability to trace fault propagation to understand more fully what happens during a program failure.

7. Acknowledgments

We would like to thank the Stanford SimOS team for providing and supporting the SimOS complete machine simulator. In particular, Edouard Bugnion wrote the Alpha port of SimOS that forms the basis of our tool. Wee Teck Ng (University of Michigan) provided the fault-injector used in this paper.

This research was supported in part by NSF grant MIP-9521386, Digital Equipment Corporation, and the University of Michigan. Peter Chen was also supported by an NSF CAREER Award (MIP-9624869).

8. References

[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.

[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, Nuno Neves, and Miguel Castro. Lightweight Logging for Lazy Release Consistent Distributed Shared Memory. *Operating Systems Design and Implementation (OSDI)*, October 1996.

[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.

[Iyer95] Ravishankar K. Iyer. Experimental Evaluation. In *Proceedings of the 1995 International Symposium on Fault-Tolerant Computing*, pages 115–132, July 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.

[Lee93] Inhwan Lee and Ravishankar K. 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.

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

[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.

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

[Rosenblum95] Mendel Rosenblum, Stephen A. Herrod, Emmett Witchel, and Anoop Gupta. Complete computer system simulation: the SimOS approach. *IEEE Parallel & Distributed Technology: Systems & Applications*, 3(4):34–43, January 1995.

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

[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.

[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 and 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.