Proceedings of the 1999 Symposium on Fault-Tolerant Computing (FTCS)

The Systematic Improvement of Fault Tolerance in the Rio File Cache

Wee Teck Ng and Peter M. Chen
Computer Science and Engineering Division
Department of Electrical Engineering and Computer Science
University of Michigan
{weeteck,pmchen}@eecs.umich.edu
http://www.eecs.umich.edu/Rio

Abstract

Fault injection is typically used to characterize failures and to validate and compare fault-tolerant mechanisms. However, fault injection is rarely used for all these purposes to guide the design and implementation of a fault-tolerant system. We present a systematic and quantitative approach for using software-implemented fault injection to guide the design and implementation of a fault-tolerant system. Our system design goal is to build a write-back file cache on Intel PCs that is as reliable as a write-through file cache. We follow an iterative approach to improve robustness in the presence of operating system errors. In each iteration, we measure the reliability of the system, analyze the fault symptoms that lead to data corruption, and apply fault-tolerant mechanisms that address the fault symptoms. Our initial system is 13 times less reliable than a write-through file cache. The result of several iterations is a design that is both more reliable (1.9% vs. 3.1% corruption rate) and 5-9 times as fast as a write-through file cache.

1 Introduction

Software-implemented fault injection (SWIFI) is a common technique in the fault-tolerant community [12]. Software fault injection can be used for many purposes, such as comparing the robustness of different systems [24, 29, 16], understanding how systems behave during a fault [8, 4, 15], and validating fault-tolerant mechanisms [3, 11, 21, 6]. However, there are very few case studies that use fault injection for all three of these purposes to guide the design and implementation of a fault-tolerant system.

In this paper, we present a systematic approach for using software-implemented fault injection to guide the design and implementation of a fault-tolerant system, focusing on the file system and file cache modules of the FreeBSD operating system. Our system design goal is to enable data in memory to survive operating system crashes as reliably as data on disk. Specifically, we want to build a software file cache that leaves dirty file data in memory (a write-back file cache), yet loses file data as seldomly as if it wrote data immediately to disk (a write-through file cache). Normal write-back file caches are very fast but are much less reliable than write-through caches. For example, the default write-back file cache in FreeBSD loses data during 39% of operating system crashes, while a write-through file cache loses data during 3% of the crashes. Write-back file caches are less reliable because they are often unable to write dirty file data to disk during a crash.

We follow an iterative approach to improve the robustness of a write-back file cache in the presence of operating system errors. In each iteration, we measure the reliability of the system, analyze the fault symptoms that lead to data corruption, and apply fault-tolerant mechanisms that address the fault symptoms. The result of several iterations is a design that improves reliability by a factor of 21. The resulting write-back file cache is both more reliable (1.9% vs. 3.1% corruption rate) and 5-9 times as fast as a write-through file cache.

This paper makes two main contributions:

- We describe the design and implementation of a reliable write-back file cache on Intel PCs. We call this the *Rio file cache* (Rio stands for RAM I/O). An earlier study showed how to implement a write-back file cache on Digital Alpha workstations that is as robust against software errors as a write-through file cache [7]. The earlier study uses *warm reboot*, which writes file cache data to disk during reboot. Unfortunately, warm reboot relies on several Alpha-specific hardware features, such as a reset button that does not erase memory. In this paper, we use a new software technique called *safe sync* that writes dirty file cache data reliably to disk during the last stage of a crash. Safe sync requires no hardware support and can be used on a wide variety of platforms.
- We present a detailed case study of using software fault injection to systematically improve the robustness of a large software system through several iterations. A key feature of our methodology is using quantitative data to guide the design and implementation of the system. At each iteration, we use fault injection to evaluate the reliability of our design, identify vulnerabilities, and provide quantitative results that help select techniques to address these vulnerabilities. We find that several design iterations are needed to reach our reliability goal, because the first iteration may introduce new bugs or leave secondary vulnerabilities hidden. Another feature of our methodology is that we use fault injection to remove faults on real systems, unlike prior simulation-based fault removal studies [12]. Since we inject faults into real systems, we can accurately characterize the system failure process and fault propagation without the

performance overheads of simulation-based tools.

2 Experimental Methodology

Our experiments are performed on PCs running the FreeBSD 2.2.7 operating system [18]. Each PC has an Intel Pentium processor, 128 MB of memory, a 2 GB IDE hard drive, and the Phoenix 4.0 BIOS. We quantify reliability for a design by injecting various faults into a running operating system, letting it crash and reboot, and measuring how frequently the file system is corrupted. *Corruption rate* is the fraction of crashes that corrupt file data. The following sections describe the faults we inject into the operating system, how we inject faults, and how we detect file system corruption.

2.1 Description of Faults

This section describes the types of faults we inject into the operating system. Our primary goal in designing our fault model is to generate a *wide variety* of operating system crashes. Our models are derived from studies of commercial operating systems and databases [27, 26, 17] and from prior models used in fault-injection studies [4, 15, 14, 7]. 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 concentrate on software faults because studies have shown that software has become the dominant cause of system outages [10]. We classify injected faults into three categories: bit flips, low-level software faults, and high-level software faults.

The first category of faults flips random bits in the kernel's address space [4, 14]. We target three areas of the kernel's address space: the *text*, *heap*, and *stack*. 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 kernel text segment. These faults are intended to approximate the assembly-level manifestation of real C-level programming errors [15]. 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).

The last and most extensive category of faults imitate specific programming errors in the operating system [26]. 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 [15, 17]. We inject *pointer* corruption by corrupting the addressing bytes of instructions which access operands in memory [26, 17]. We either flip a bit within the addressing-form specifier byte (ModR/M) or the scale, index or base (SIB) byte following the instruction opcode [2]. We do not corrupt the stack pointer registers (i.e. *esp* and *ebp* registers) as these are used to access local variables instead of as a pointer variable. We inject an *allocation management* fault by modifying the kernel's malloc procedure to occasionally free the newly allocated block of memory after a delay of 0-64 ms. Malloc is set to inject this error every 1000-4000 times it is called; this fault occurs approximately every 10 seconds on our system. We inject a *copy overrun* fault by modifying the kernel's data copy procedures to occasionally increase the number of bytes they copies. The length of the overrun is 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 [26] and modifying it according to our specific platform and experience. The copy routines are set to inject this error every 1000-4000 times it is called; this fault occurs approximately every 5 seconds on our system. 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 *memory leaks* by modifying free() to occasionally return without freeing the block of memory. We inject *interface errors* by corrupting one of the arguments passed to a procedure.

We collect data on 100 crashes (each using a different random seed) for each of the 15 fault types above for each of the five designs in this paper—this represents about 8 machine-months of continuous operating system crashes. Fault injection cannot mimic the exact behavior of all real-world operating system crashes. However, the wide variety of faults we inject (15 types), the random nature of the faults, and the sheer number of crashes we performed (7500) give us confidence that our experiments cover a wide range of real-world crashes.

2.2 Injecting Faults

Our fault injection tool uses object-code modification to inject bugs into the kernel text. It is embedded into the kernel and, when triggered, will select an instruction in the kernel text and corrupt it. Some fault types, such as memory leaks, are implemented by modifying the relevant kernel routines (e.g. malloc) to occasionally fail when the fault is triggered. The fault trigger and injection location within the kernel text are determined by a random seed. We do not inject fault into the recovery and fault-tolerant code we added into the system.

Unless otherwise stated, we inject 10 faults for each run to increase the chance of triggering a fault. Most crashes occur within 10 seconds from the time the fault was injected. If a fault does not crash the operating system after fifteen minutes, we restart the system and discard the run; this happens about 40% of the time. Note that faults that leave the system running will corrupt data on disk for both write-back and write through file caches, so these runs do not change the relative reliability between file caches.

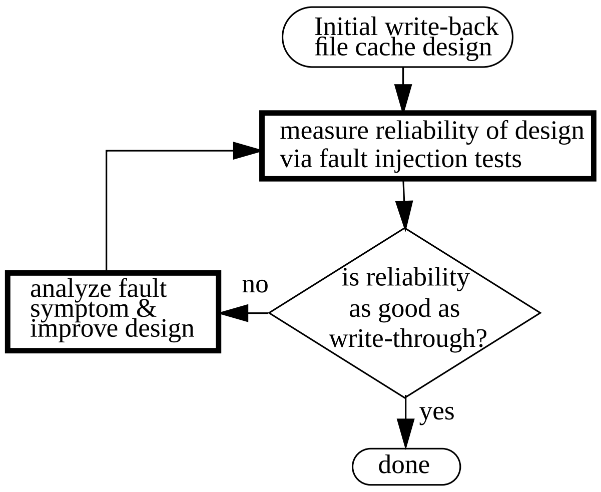
Initial write-back file cache design
measure reliability of design via fault injection tests
is reliability as good as write-through?
no
analyze fault symptom & improve design
yes
done

Figure 1: Design Process. The bold boxes represent the stages that use fault injection data to guide the design process.

2.3 Detecting Corruption after a Crash

We run a repeatable, synthetic workload called *memTest* to detect file system corruption. *memTest* generates a repeatable stream of file and directory creations, deletions, reads, and writes, reaching a maximum file set size of 128 MB. Actions and data in *memTest* are controlled by a pseudo-random number generator. After each iteration, *memTest* records its progress in a status file on a network disk that is not affected by the fault injection experiments. After the system crashes, we reboot the system and run *memTest* until it reaches the point when the system crashed. This reconstructs the correct contents of the test directory at the time of the crash, and we then compare the reconstructed, correct contents with the rebooted file system. The experiments are controlled by a host connected to each test system via a serial link. The control host logs relevant data (crash latencies, fault symptoms, etc.) for subsequent analysis.

3 Design Process

We follow an iterative approach, as described in [23], to improve the robustness of a write-back file cache in the presence of operating system errors (Figure 1). In each iteration, we measure the reliability of the system using the approach described in Section 2, analyze the fault symptoms that lead to data corruption, and apply fault-tolerant mechanisms that address the fault symptoms. We use software fault injection at each stage to provide quantitative data to help guide our design. For example, we evaluate the reliability of our system quantitatively to decide if it meets our design goal. We also use the data collected during fault injection to analyze and fix faults. Note that we do not merely fix the faults we ourselves have injected. Rather, we use the bugs we inject to reveal categories of faults, then we fix the entire category.

Our goal is to make Rio (our write-back file cache) as reliable as a write-through file cache. Write-through file caches are considered very reliable against software crashes because they propagate data immediately to disk, and disks are not easily corrupted by operating system crashes [25, 28, 7]. We configure FreeBSD to use a write-through file cache, then measure the corruption rate to be 3.1% using the method described in Section 2. That is, 3.1% of the crashes corrupt some data in the file system. Table 1 summarizes the corruption rate by fault category of all our designs in this paper.

Fault Type	Write-Thru File Cache	Write-Back File Caches			
		Default FreeBSD Sync	Basic Safe Sync	Enhanced Safe Sync	BIOS Safe Sync
text	3	51	7	5	2
stack	0	3	3	2	0
heap	5	28	8	3	1
initialization	10	45	9	7	4
delete random inst.	4	43	8	2	4
dest. reg.	4	42	9	5	2
source reg.	4	43	10	3	1
delete branch	4	51	14	4	5
pointer	3	38	5	4	2
allocation	0	100	5	0	0
copy overrun	4	36	1	3	2
synchronization	0	3	1	0	0
off-by-one	4	59	16	9	3
mem. leak	0	0	0	0	0
interface	1	47	8	3	2
Total of 1500	**46 (3.1%)**	**589 (39%)**	**104 (6.9%)**	**50 (3.3%)**	**28 (1.9%)**

Table 1: Comparing Reliability. This table shows how often each type of fault corrupts data for a write-through file cache (our reliability target) and the four designs for a reliable write-back file cache.

4 Design Iterations

This section describes the four design iterations we went through to arrive at the final system. Each subsection (4.1-4.4) describes the write-back file cache used in a design iteration, presents results from the fault-injection tests on that design, then analyzes the results to select techniques to fix the revealed fault categories.

In all our designs, we modify the FreeBSD file cache in two ways to be a pure write-back file cache. First, FreeBSD normally writes dirty file data to disk every 30 seconds or when a full file block is written. We disable this reliability-induced write-back, so the system only writes data back to disk when dirty blocks are replaced in the file cache. Second, FreeBSD normally limits the amount of dirty file cache data to 10% of available system memory. We increase this limit by allowing dirty file data to migrate from the file cache to the virtual memory system, as is done in memory-mapped file systems [5].

4.1 Design Iteration 1: Default FreeBSD Sync

4.1.1 Design

We start the design process with the default sync used in FreeBSD. Sync refers to the routine that writes dirty file-cache data to disk during a crash. FreeBSD's default sync routine examines all blocks in the file cache and writes dirty blocks to disk using normal file system routines.

4.1.2 Results and Analysis

Unfortunately, the default FreeBSD sync is not very robust during operating system crashes. As shown in Table 1, 39% of crashes corrupted some file system data when using the default FreeBSD sync. This corruption rate is 13 times as high as that of a write-through file cache.

We next examine the corrupted runs in greater detail, focusing on where the faults are injected into the system and how the system crashes. We determine how the system crashes by looking at the crash messages and tracing fault propagation with the aid of the FreeBSD kernel debugger. Our fault-injection tool helps by printing the kernel routine name and location of corrupted code. We use this information to divide the fault symptoms into categories:

- *Hang before sync*: Most data corruptions occur because the system hangs and fails to call the sync routine. Many workstations have a reset key that allows the user to drop the system into the console prompt. The user can then issue a sync command directly or initiate recovery using a user-written routine. But most PCs do not have such a feature, and those that are equipped with a reset switch typically erase memory (including dirty file cache data).
- *Page fault during sync*: Sync often fails because it encounters a page fault while trying to write dirty file cache data to disk. The page fault occurs when the operating system accesses unmapped data or mapped data with the wrong permission settings. It can also happen when the code is invalid or unmapped. The FreeBSD sync routine uses many different kernel routines and data structures (e.g. mounted file system list, vnode data structures, buffer hash list), so this fault is quite common.
- *Buffer locked during sync*: FreeBSD's sync routine obeys the locking protocol used during normal operation. It does not write to disk any file cache blocks that are locked, so data in these blocks are lost.
- *Double fault*: The Pentium processor calls a double-fault handler if it detects an exception while servicing a prior exception [2]. The processor will reset and abandon sync if another exception occurs when the double fault handler is being serviced.
- *File system errors*: Our tool may inject faults into any part of the kernel. Faults that are injected into file system routines often cause data corruption. For example, the file system's write routine might be changed to write to the wrong part of the file. We do not attempt to fix this fault symptom because the write-through file cache is also susceptible to these errors.
- *Device Timeout*: Sync sometimes fails because it experiences repeated device timeouts when writing to the hard drive.
- *Unknown*: A few data corruptions are due to unknown causes. For these corruptions, the sync routine appears to be successful, and the injected bugs appear to be benign. We do not attempt to overcome this problem because of the lack of information and the low frequency of this fault symptom.

Fault Symptom	**# of Corruptions**	**Solutions Used in the Next Design (Section 4.2.1)**
hang before sync	268 (17.9%)	software reset key
page fault during sync	163 (10.9%)	registry, safe sync
buffer locked during sync	89 (5.9%)	registry, safe sync
double fault	39 (2.6%)	disable interrupt in safe sync
file system error	25 (1.7%)	
device timeout	3 (0.2%)	
unknown	2 (0.1%)	
Total of 1500	**589 (39.3%)**	

Table 2: Fault Symptoms for Default FreeBSD Sync.

Table 2 summarizes the categories of fault symptoms and some potential solutions we developed (discussed in the next section) to reduce the vulnerability of the system to that fault symptom.

4.2 Design Iteration 2: Basic Safe Sync

4.2.1 Design

A write-back file cache must do two steps to write dirty data back to disk during a crash. First, the system must transfer control to the sync routine. Second, the sync routine must write dirty file data successfully to disk. Most of the corruptions experienced using the default FreeBSD sync fail one of these two steps. *Hang before sync* fails to transfer control to the sync routine during a crash. Most of the other fault symptoms transfer control to sync but experience an error during sync.

We address errors in these two steps separately. First, we must make it more likely that the system will successfully transfer control to the sync routine during a crash. To fix *hang before sync*, we use a software reset key that calls sync when pressed. We modify the low-level keyboard interrupt handler of FreeBSD to call sync whenever it detects a certain key sequence (e.g. control-alt-delete). This addresses the dominant fault symptom in Table 2.

Second, we must make it more likely that the sync routine, once called, will write dirty file data successfully to disk. Default FreeBSD sync fails this step because it depends on many parts of the kernel. The default FreeBSD sync calls many routines and uses many different data structures. Sync fails if *any* of the routines or data structures are corrupted. To make sync more robust, we must minimize the scope of the system that it depends on.

To minimize data dependencies, we implement informational redundancy [13] by creating a new data structure called the *registry*. The registry contains all information needed to find, identify, and write all file cache blocks. For each block in the file cache, the registry contains the physical memory address, file ID (device number and inode number), file offset, and size. The registry allows sync to operate without using previously needed kernel data structures, such as file system and disk allocation data. The registry is wired in memory to reduce the likelihood of page faults during sync. Registry information changes relatively infrequently during normal operation, so the overhead of maintaining it is low.

We replace FreeBSD's default sync routine with a new routine (called *safe sync*) that uses the registry when writing data to disk. Safe sync examines all valid entries in the registry and writes dirty file cache data directly to disk. By using information in the registry, safe sync does not depend on normal file system routines or data structures. Safe sync also takes additional precautions to increase its chances of success. First, safe sync operates below the locking protocol to avoid being stymied by a locked buffer. Second, safe sync disables interrupts to reduce the likelihood of double faults while writing to disk.

In addition to adding the registry and using a new sync routine, we also use the virtual memory system to protect file cache data from wild stores [7]. We turn off the write-permission bits in the page table for file cache pages, causing the system to generate protection violations for unauthorized stores. File cache procedures must enable the write-permission bit in the page table before writing a page and disable writes afterwards.

4.2.2 Results and Analysis

Fault injection tests on the new design show substantial improvement over the default FreeBSD sync. Table 1 shows that the new design has a corruption rate of 6.9%, which is six times better than the default FreeBSD sync. However, it still has twice as many corruptions as a write-through file cache. Table 3 breaks down the fault symptoms for our current design. The fault symptoms are very similar to those in Table 2, but the corruption rates are reduced significantly due to the fault-tolerant measures introduced in Section 4.2.1. We analyze the fault symptoms from this design to see how we can make safe sync more robust in the next iteration:

- *Hang before sync*: Table 3 shows that the reset button reduces the corruption rates substantially from the default FreeBSD sync (from 17.9% to 3.0%). We examine the remaining cases and the keyboard interrupt handler to determine what causes the reset key to fail. The dominant reason is that FreeBSD sometimes masks keyboard interrupts. If the system hangs while keyboard interrupts are masked, the reset key will not transfer control to safe sync. To fix this, we add a watchdog timer to the system timer interrupt handler [13]. The system timer interrupt handler watches for pending keyboard interrupts and calls safe sync if the keyboard interrupt does not get serviced for a long time. For five of the corruptions, the fault was injected into the terminal output routine, and safe sync failed when it tried to print some debugging information. We fix this fault by disabling debugging print statements during safe sync.

Fault Symptom	**# of Corruptions**	**Solutions Used in the Next Design (Section 4.3.1)**
hang before sync	45 (3.0%)	watchdog timer, no print
file system error	24 (1.6%)	
page fault during sync	18 (1.2%)	read-only text, private stack, restore segment registers
data corruption	11 (0.7%)	fix VM protection
double fault	4 (0.3%)	private stack
device timeout	2 (0.1%)	
Total of 1500	**104 (6.9%)**	

Table 3: Fault Symptoms for Basic Safe Sync.

- *File system error*: Again, we do not attempt to fix this fault symptom because the write-through file cache is also susceptible to these errors. Note that the corruption rate for file system errors is similar between Table 2 and Table 3. The slight differences are due to the non-determinism inherent to testing a complex, timing-dependent system.
- *Page fault during sync*: This fault symptom occurs for a variety of reasons, and we develop a variety of solutions to fix it. For example, some faults corrupt the Intel segment registers that are used by some instructions in safe sync. To fix this error, we can re-initialize the segment registers to their correct value at the beginning of safe sync. Other faults cause wild stores to write over kernel code. To fix this error, we can map the kernel code as read-only (of course, our fault injector can still modify kernel code).
- *Data corruption*: Some faults corrupted data in the file cache before crashing the system. For these runs, safe sync completed successfully but wrote out the corrupted data. While investigating the source of this corruption, we uncovered a bug in FreeBSD's protection code that sometimes allowed wild stores to overwrite the file cache and kernel code.
- *Double fault*: This bug occurs when part of the stack segment is unmapped by the injected fault but the TLB is not invalidated. The system will continue to function until the stack pointer advances beyond the valid page in the TLB, and encounter multiple page faults when it tries to fault in subsequent pages from the bogus stack. To fix this, we pre-allocate a stack for safe sync during

Fault Symptom	# of Corruptions	Solutions Used in the Next Design (Section 4.4.1)
hang before sync	21 (1.4%)	BIOS I/O, real-mode addressing
file system error	20 (1.3%)	
page fault during sync	4 (0.3%)	real-mode addressing
device timeout	3 (0.2%)	BIOS I/O
data corruption	2 (0.1%)	
Total of 1500	**50 (3.3%)**	

Table 4: Fault Symptoms For Enhanced Safe Sync.

bootup and wire it in memory. Safe sync's first action is to switch to this private stack.

4.3 Design Iteration 3: Enhanced Safe Sync

4.3.1 Design

Our next design improves on the basic safe sync design from the last iteration using the fixes suggested in Section 4.2.2. First, we add a watchdog timer to call safe sync if the system hangs with keyboard interrupts disabled. Second, we disable print statements during safe sync to remove dependencies on the print routines. Third, we re-initialize the segment registers to their proper value. Fourth, we map the kernel code as read-only and fix a bug in FreeBSD's protection code. Finally, we switch to a pre-allocated, wired stack at the beginning of safe sync to remove dependencies on the system stack.

4.3.2 Results and Analysis

We conduct fault injection tests on the new design and find that it has a corruption rate of 3.3%, versus 6.9% for the basic safe sync design of iteration 2. Enhanced safe sync is nearly as reliable as a write-through file cache. Table 4 breaks down the fault symptoms of our current design.

There are two basic dependencies remaining in our system. First, all kernel code, including safe sync, runs in virtual-addressing mode with paging enabled [2], which uses virtual addresses to access code and data. Because safe sync accesses virtual addresses, it depends on the FreeBSD virtual memory code and data (such as the doubly linked address map entries [20]). To fix this dependency, we must configure the processor to use physical addresses during safe sync.

Second, safe sync uses the low-level kernel device drivers to write data to disk. The FreeBSD disk device drivers are quite complex, and there is no simple disk device driver routine to initialize the device driver state or reset the hard drive and disk controller card. Our safe sync code can thus hang or timeout whenever it access the disk. To remove this dependency, we must bypass the complex device driver for a simpler disk interface.

4.4 Design Iteration 4: BIOS Safe Sync

4.4.1 Design

In our final design, we want to remove dependencies on the virtual memory system and device drivers. We remove dependencies on the virtual memory system by switching the processor to use physical addresses [2]. We remove dependencies on the kernel device drivers by using the BIOS interface to the disk [9]. BIOS (Basic Input/Output Service) routines are implemented in the firmware of the I/O controller. Both physical addressing and BIOS routines have limited features and are used normally to load the operating system from disk during system boot. Modern operating systems like FreeBSD use virtual addressing and replace the BIOS with their own device drivers.

Our final design replaces the safe sync code used in design iteration 3. The new safe sync procedure is summarized below (the full source code is available at http://www.eecs.umich.edu/Rio):

- *Part 1: Initial setup*: we followed the instructions outlined in Section 8.8.1 of [2], which includes setting up a linearly mapped segments for data and code, setting the global (code/data) and interrupt descriptor table registers for real-mode operation, and making a long jump to the real-mode switch code.
- *Part 2: Mode switching*: the real-mode switch code disables paging, loads the segment registers with real-mode segments, and clears the paging enable bit before making a jump to the real-mode safe sync code. This jump brings the processor to real-mode operation.
- *Part 3: Real-mode setup*: BIOS safe sync begins by initializing the remaining segment registers, setting the interrupt controllers to real-mode operation [1], and initializing the video console and disk controller using the BIOS interface [9]. The rest of BIOS safe sync is fairly straightforward and is generated from the C version of enhanced safe sync. We modify the resulting assembly code by adding address/data overrides [2] and using a large data segment (i.e. big real-mode [22]) to access data beyond the first 1 MB of memory. During sync, we copy the file cache data into the lower 1 MB of memory because the BIOS disk interface uses 16-bit segment addressing.

Part 1 of BIOS safe sync is a C function and can be invoked directly by the FreeBSD kernel. The rest of BIOS safe sync code is written in assembly and is not accessible to the FreeBSD kernel, because it resides in unmapped physical memory pages that are hidden from the FreeBSD page allocator.

4.4.2 Results and Analysis

We conduct fault injection tests on our BIOS safe sync. Table 5 breaks down the fault symptoms for our latest design. The overall corruption rate is 1.9%, which is 40% more reliable than a write-through file cache. We speculate that Rio is able to achieve higher reliability than a write-through file cache because the virtual memory protection

Fault Symptom	**# of Corruptions**	**Possible Solutions**
file system errors	17 (1.1%)	
hang before sync	5 (0.3%)	hardware reset
data corruption	4 (0.3%)	
device timeout	2 (0.1%)	warm reboot
Total of 1500	**28 (1.9%)**	

Table 5: Fault Symptoms For BIOS Safe Sync.

described in Section 4.2.1 can protect the file cache (and thus indirectly protect the disk) from wild stores.

It may be possible to improve upon BIOS safe sync by adding a hardware reset key and modifying the PC firmware and motherboard to not initialize memory on reset/reboot. This would allow the system to do a complete reset, then to perform a warm reboot as was done in [7]. Doing so should fix the remaining hangs before sync and device timeouts, but it would incur significant system cost.

5 Discussion

We would like to address the scalability, portability, and cost of our design approach.

- *Scalability*: Our target system is fairly representative of a medium-size software development project. The relevant operating system code (file system, VM, interrupt, etc.) spans approximately 40 files and 20,000 lines of code. We added 3 files and 2000 lines of code. The development effort took one man-year. This includes the substantial time it took to understand FreeBSD and the Intel PC architecture (microarchitecture, assembly language, system BIOS). Since our experimental setup was fully automated and we had sufficient machines to run the experiments in parallel, most of our time was devoted to uncovering fault symptoms and debugging code. The analysis process was largely manual, though we wrote several tools to expedite this process. Our design approach is applicable to many software development projects as long as there are enough resources to perform the fault injection experiments and sufficient expertise to implement the fault-tolerant mechanisms.
- *Portability*: We demonstrated our design methodology on FreeBSD running on Intel PCs. We have also tried this approach on a limited scale when implementing a reliable write-back file cache on Digital Alpha workstations [7] and the Postgres database [19]. We are confident that our approach is portable to other systems.
- *Cost*: Our design took four iterations, requiring 8 machine-months of testing. We tackled the dominant fault symptoms in the first iteration, with diminishing returns on successive iterations. Using software fault injection provided quantitative data on when our system reached our reliability goal. For example, we could have stopped after enhanced safe sync, because its corruption rate was statistically indistinguishable from a write-through file cache. Choosing a reliability goal is a tradeoff between design cost and application requirements.

File System	**cp+rm**	**Andrew**
UFS	54.0 seconds	1.9 seconds
write-through file cache	186.3 seconds	6.7 seconds
write-back file cache (Rio)	**19.9 seconds**	**1.3 seconds**

Table 6: Performance Comparison. All performance measurements were made on a PC with a 400 MHz Pentium-II processor, 128 MB of 100 MHz SDRAM, and a IBM DCAS-34330W SCSI disk (with the disk write-cache enabled).

6 Performance

Table 6 compares the performance of our reliable write-back file cache (Rio) with different Unix file systems (UFS), each providing different guarantees on when data is made permanent. UFS is the default FreeBSD Unix file system. It writes data asynchronously to disk when 64 KB of data has been collected, when the user writes non-sequentially, or when the update daemon flushes dirty file data (once every 30 seconds). UFS writes metadata synchronously to disk to enforce ordering constraints. The write-through file cache writes data and metadata synchronously to disk. The last row shows the performance of the Rio write-back file cache.

We run two workloads, cp+rm and Andrew. cp+rm recursively copies then removes the FreeBSD source tree (23 MB). Andrew models a software development workload. All results represent an average of 20 runs.

Table 6 shows that our Rio file cache prototype is 5-9 times as fast as a write-through file cache. It is also 1.5-2.7 times as fast as the standard Unix file system. Rio is roughly equivalent in reliability to a write-through file cache. Both Rio and a write-through file cache are more reliable than standard UFS. UFS loses up to 30 seconds of data on a crash, while Rio and a write-through file cache typically lose no data on a crash.

7 Conclusions

We have presented a systematic and quantitative approach for using software-implemented fault injection to guide the design and implementation of a fault-tolerant system. Our goal was to build a write-back file cache on Intel PCs that was as reliable as a write-through file cache. We followed an iterative approach to improve the robustness of a write-back file cache in the presence of operating system errors. In each iteration, we measured the reliability of the system, analyzed the fault symptoms that led to data corruption, and applied fault-tolerant mechanisms that addressed the fault symptoms. The result of several iterations was a design that improved reliability by a factor of 21. The resulting write-back file cache is both more reliable (1.9% vs. 3.1% corruption rate) and 5-9 times as fast as a write-through file cache.

8 Acknowledgments

This research was supported in part by NSF grant MIP-9521386 and Intel Technology for Education 2000. Peter Chen was also supported by an NSF CAREER Award (MIP-9624869).

9 References

[1] *Intel 82371AB PCI ISA IDE Xcelerator (PIIX4) Datasheet*. Intel Corporation, 1997.

[2] *Intel Architecture Software Developer's Manual: Volume 1-3*. Intel Corporation, 1997.

[3] Jean Arlat, Martine Aguera, Louis Amat, Yves Crouzet, Jean-Charles Fabre, Jean-Claude Laprie, Eliane Martins, and David Powell. Fault Injection for Dependability Validation: A Methodology and Some Applications. *IEEE Transactions on Software Engineering*, 16(2):166–182, February 1990.

[4] 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.

[5] A. Bensoussan, C.T. Clingen, and R.C. Daley. The Multics Virtual Memory: Concepts and Design. *Communications of the ACM*, 15(5):308–318, May 1972.

[6] Joao Carreira, Henrique Madeira, and Joao Gabriel Silva. Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers. *IEEE Transactions on Software Engineering*, 24(2):125–136, February 1998.

[7] 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.

[8] R. Chillarege and N. S. Bowen. Understanding Large System Failure–A Fault Injection Experiment. In *Proceedings of the 1989 International Symposium on Fault-Tolerant Computing (FTCS)*, pages 356–363, 1989.

[9] Frank Van Gilluwe. *The Undocumented PC: A Programmer's Guide to I/O, CPUs, and Fixed Memory Areas*. Addison-Wesley Developer Press, 1997.

[10] Jim Gray. A Census of Tandem System Availability between 1985 and 1990. *IEEE Transactions on Reliability*, 39(4), October 1990.

[11] John Hudak, Byung-Hoon Suh, Dan Siewiorek, and Zary Segall. Evaluation and Comparison of Fault-Tolerant Software Techniques. *IEEE Transactions on Reliability*, 42(2), June 1993.

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

[13] Barry W. Johnson. *Design and Analysis of Fault-Tolerant Digital Systems*. Addison-Wesley Publishing Co., 1989.

[14] 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.

[15] 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.

[16] Nathan P. Kropp, Philip J. Koopman, and Daniel P. Siewiorek. Automated Robustness Testing of Off-the_shelf Software Components. In *Proceedings of the 1998 Symposium on Fault-Tolerant Computing (FTCS)*, June 1998.

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

[18] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley Publishing Company, 1996.

[19] 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.

[20] Richard F. Rashid, Jr. Avadis Tevanian, Michael Young, David Golub, Robert Baron, David Black, Jr. William J. Bolosky, and Jonathan Chew. Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. *IEEE Transactions on Computers*, 37(8):896–908, August 1988.

[21] Mario Zenha Rela, Henrique Madeira, and Joao G. Silva. Experimental Evaluation of the Fail-Silent Behavior in Programs with Consistency Checks. In *Proceedings of the 1996 Symposium on Fault-Tolerant Computing (FTCS)*, June 1996.

[22] Tom Shanley. *Protected Mode Software Architecture*. Addison-Wesley Developer Press, 1996.

[23] Daniel P. Siewiorek. *Reliable Computer Systems: Design and Evaluation*. A K Peters, 1998.

[24] Daniel P. Siewiorek, John J. Hudak, Byung-Hoon Suh, and Zary Segal. Development of a Benchmark to Measure System Robustness. In *Proceedings of the 1993 International Symposium on Fault-Tolerant Computing*, pages 88–97, June 1993.

[25] Abraham Silberschatz and Peter B. Galvin. *Operating System Concepts*. Addison-Wesley, 1994.

[26] 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.

[27] 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.

[28] Andrew S. Tanenbaum. *Distributed Operating Systems*. Prentice-Hall, 1995.

[29] Timothy K. Tsai, Ravishankar K. Iyer, and Doug Jewett. An Approach towards Benchmarking of Fault-Tolerant Commercial Systems. In *Proceedings of the 1996 Symposium on Fault-Tolerant Computing (FTCS)*, June 1996.