The Design and Verification of the Rio File Cache

Wee Teck Ng and Peter M. Chen, *Senior Member*, *IEEE*

Abstract—Today's file systems are limited in speed and reliability by memory's vulnerability to operating system crashes. Because memory is viewed as unsafe, systems periodically write modified file data back to disk. These extra disk writes lower system performance and the delay period before data is safe lowers reliability. The goal of the Rio (RAM I/O) file cache is to make ordinary main memory safe for persistent storage by enabling memory to survive operating system crashes. *Reliable main memory* enables the Rio file cache to be as reliable as a write-through file cache, where every write is safe instantly, and as fast as a pure write-back file cache, with no reliability-induced writes to disk. This paper describes the systematic, quantitative process we used to design and verify the Rio file cache on Intel PCs running FreeBSD and the reliability and performance of the resulting system.

Index Terms—File systems, reliable main memory, software fault injection.

1 Introduction

A modern storage hierarchy combines random-access memory, magnetic disk, and possibly optical disk or magnetic tape to try to keep pace with rapid advances in processor performance. I/O devices such as disks and tapes are considered reliable places to store persistent data such as user files. However, random-access memory is viewed as an unreliable place to store persistent data because it is vulnerable to power outages and operating system crashes.

Memory's vulnerability to power outages is straightforward to understand and fix. A $100 uninterruptible power supply can keep a system running long enough to dump memory to disk in the event of a power outage [3], or one can use non-volatile memory such as Flash RAM [73]. We do not consider power outages further in this paper.

Memory's vulnerability to operating system crashes is more challenging. Most people would feel nervous if their system crashed while the sole copy of important data was in memory, even if the power stayed on [68], [63], [28]. Consequently, file systems write data periodically to disk and transaction processing applications view transactions as committed only when data is written to disk.

Memory's perceived unreliability forces a trade-off between performance and reliability (Fig. 1):

- Applications requiring high reliability, such as transaction processing, write data synchronously through to disk, but this limits performance to that of disk. While optimizations such as logging and group commit can increase effective *throughput* [58], [59], [26], [18], they work well only when there are concurrent or delayed operations that can be grouped together and they cannot improve the *latency* of individual operations.
- Most file systems mitigate the performance lost in synchronous, reliability-induced writes by writing data *asynchronously* to disk. This allows a greater degree of overlap between CPU time and I/O time. Unfortunately, asynchronous writes make no firm guarantees about when the data is safe on disk; the exact moment depends on the disk queue length and disk speed. On these systems, users must resign themselves to the fact that their data may not be safe on disk when a write or close finishes.
- Many file systems improve performance further by delaying some writes to disk in the hopes of the new data being deleted or overwritten [54]. This delay is often set to 30 seconds, which risks the loss of data written within 30 seconds of a crash. Unfortunately, 1/3 to 2/3 of newly written data lives longer than 30 seconds [5], [27] and this data is written through to disk under this policy. File systems differ in how much data is delayed. For example, BSD 4.4 only delays partially written blocks and then only until the file is closed. Systems that delay more types of data and have longer delay periods are better able to decrease disk traffic, but risk losing more data.
- Applications that desire maximum performance use a pure write-back scheme where data is written to disk only when the memory is full [48]. This can only be done by applications for which reliability is not an issue, such as compilers that write temporary files.

It is common for file systems to use a combination of write-back strategies. For example, many Unix file systems delay partially written file blocks while initiating asynchronously writes immediately for complete file blocks. However, all these strategies suffer from the same basic trade-off: Avoiding disk writes to achieve good performance

- *W.T. Ng is with Bell Laboratories, Rm. 2A-318, 600 Mountain Ave., Murray Hill, NJ 07974. E-mail: weeteck@research.bell-labs.com.*
- *P.M. Chen is with the Computer Science and Engineering Division, Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109-2122. E-mail: pmchen@eecs.umich.edu.*

Manuscript received 13 July 1999; revised 10 July 2000; accepted 23 Jan. 2001.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 110440.

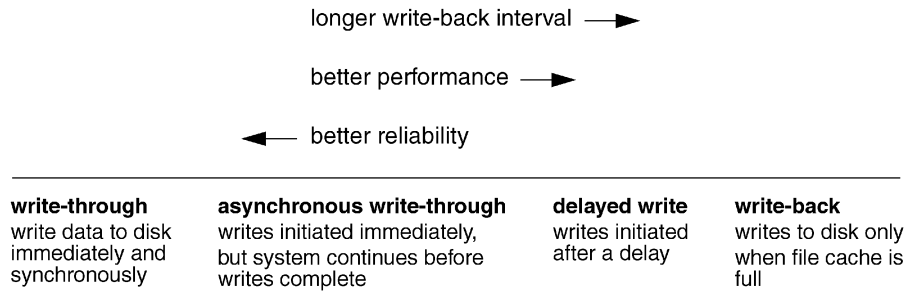


Fig. 1. Memory's unreliability forces trade-off between performance and reliability. The longer the interval between when new data enter the file cache and when it is written to disk, the better the performance, but the worse the reliability.

inevitably leads to a loss of reliability. The goal of the Rio (RAM I/O) file cache is to break this fundamental trade-off by improving the reliability of memory to be comparable to the reliability of disk. Reliable main memory allows Rio to use a pure write-back strategy (no reliability-induced writes to disk) while achieving reliability equivalent to that of a write-through file cache.

We implement the Rio file cache in the FreeBSD operating system (version 2.2.7) by starting with the default write-back file cache and following an iterative, quantitative approach to improve its robustness during operating system crashes. In each iteration, we measure the reliability of the system using fault injection, 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 over the default write-back cache. The resulting Rio file cache is at least as reliable as a write-through file cache (1.9 percent vs. 3.1 percent corruption rate) and 4-37 times as fast as the standard Unix file system.

This paper makes two main contributions:

- We describe the design, implementation, and performance of a reliable, write-back file cache on Intel PCs, using software techniques we develop to make file cache data reliable against software errors. These techniques are applicable to many platforms. An earlier study showed how to implement a reliable file cache on Digital Alpha workstations [15]. The earlier study used *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, and is not applicable to most Intel PCs. 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 is a general technique that requires no hardware support and can be used on a wide variety of platforms, including those with write-back CPU caches and that erase memory during reboot. It is also potentially less robust than warm reboot as it writes important file cache data to disk while the system is crashing. In this paper, we demonstrate how we can systematically make safe sync as reliable as a write-through file cache.
- We present a detailed case study of using software fault injection to improve systematically the robustness of a large software system through several iterations. Software-implemented fault injection [34] is used commonly to compare the robustness of different systems [61], [69], [42], understand how systems behave during a fault [16], [10], [39], and validate fault-tolerant mechanisms [4], [31], [57], [13]. 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 our study, we use fault injection for all three of these purposes to improve iteratively the fault tolerance of our reliable file cache. 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.

2 Quantifying Reliability

A crucial aspect of our design process (Section 3) is its quantitative nature. At each step in the design process, we quantify reliability for the current 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 where there is any corrupted file data. This section describes the faults we inject into the operating system and how we detect file system corruption. We implement our system on PCs running the FreeBSD 2.2.7 operating system [49]. Each PC has an Intel Pentium processor, 128 MB of memory, a 2 GB IDE hard drive, and the Phoenix 4.0 BIOS.

2.1 Description of Faults

We first describe the types of faults we inject into the operating system. Our primary goal in designing these faults is to generate a wide variety of operating system crashes. Our models are derived from studies of commercial operating systems and databases [66], [65], [43] and from prior models used in fault-injection studies [10], [39], [38], [15]. 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. Table 1 shows examples of how how real-world programming errors can manifest themselves as some of the faults we inject in our experiments.

TABLE 1
Relating Faults to Programming Errors

Fault Type	Example of Programming Error	
	Correct Code	**Faulty Code**
initialization	function () {int i=**0**; ...}	function () {int i; ...}
delete random inst.	for (i=0; i<10; i++,**j++**) {body}	for (i=0; i<10; i++) {body}
destination reg.	**numFreePages** = count(freePageHeadPtr)	**numPages** = count(freePageHeadPtr)
source reg.	numPages = **physicalMemorySize**/pageSize	numPages = **virtualMemorySize**/pageSize
delete branch	**while** (flag) {body}	**while** (!flag) {body}
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 again;
copy overrun	for (i=0; i<**sizeUsed**; i++) {a[i] = b[i]};	for (i=0; i<**sizeTotal**; i++) {a[i] = b[i]};
synchronization	**getWriteLock;** write(); **freeWriteLock;**	write();
off-by-one	for (i=0; i<size; i++)	for (i=0; i<**=**size; i++)
memory leak	function () {ptr = malloc(N); use ptr; **free(ptr);** return;}	function () {ptr = malloc(N); use ptr; return;}
interface error	results = strcmp(str1, **str2**);	results = strcmp(str1, **str3**);

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

We concentrate on software faults because studies have shown that software has become the dominant cause of system outages [23], [24]. We classify injected faults into three categories: bit flips, low-level software faults, and high-level software faults. Unless otherwise stated, we inject 10 faults for each run to increase the chances that a fault will be triggered. Most crashes occurred within 10 seconds from the time the fault was injected. If a fault does not crash the operating system after 15 minutes, we restart the system and discard the run; this happens about 40 percent 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.

The first category of faults flips random bits in the kernel's address space [10], [38]. 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 [39]. 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 nonbranch).

The last and most extensive category of faults imitate specific programming errors in the operating system [65]. These are targeted more at specific programming errors than the previous fault category. Table 1 provides a summary of the programming errors we inject. The implementation details can be found in [52].

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 eight 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 (7,500) give us confidence that our experiments cover a wide range of real-world crashes.

2.2 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 pseudorandom 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.

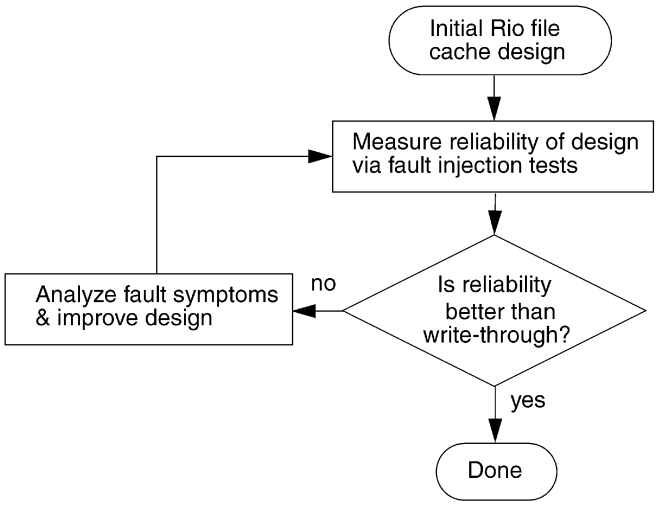


Fig. 2. Design process.

3 Design Process

We follow an iterative approach [62] to improve the robustness of a write-back file cache in the presence of operating system errors (Fig. 2). 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 the Rio 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 [63], [68], [15]. We configure FreeBSD to use a write-through file cache, then measure the corruption rate to be 3.1 percent using the method described in Section 2. That is, 3.1 percent of the crashes corrupt some data in the file system. Our reliability goal for the Rio file cache is thus to achieve a corruption rate as low or lower than 3.1 percent.

Table 2 summarizes the corruption rate by fault category of all our designs in this paper.

TABLE 2
Comparing Reliability

Fault Type	Write-Through 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
destination 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
memory leak	0	0	0	0	0
interface error	1	47	8	3	2
Total	**46 of 1500 (3.1%)**	**589 of 1500 (39.3%)**	**104 of 1500 (6.9%)**	**50 of 1500 (3.3%)**	**28 of 1500 (1.9%)**
95% Confidence Interval	**2.2 - 3.9%**	**36.8 - 41.8%**	**5.6 - 8.2%**	**2.4 - 4.3%**	**1.2 - 2.6%**

This table shows how often each type of fault corrupts data for a write-through file cache (out reliability target) and the four designs for a reliable write-back file cache. We conduct 1,500 crashes for each system (100 for each fault type). The last row shows the 95 percent confidence interval for the mean corruption rate.

TABLE 3
Design Iterations for the Rio File Cache

Section	Design Iteration	Fault-Tolerant Mechanisms
4.1	default FreeBSD sync	default sync used in FreeBSD during crashes
4.2	basic safe sync	add VM protection, reset key, registry, safe sync
4.3	enhanced safe sync	fix VM protection bugs; add watchdog timer, private stack; disable debugging prints; map kernel code read-only; initialize segment registers
4.4	BIOS safe sync	physical addressing, BIOS disk I/O

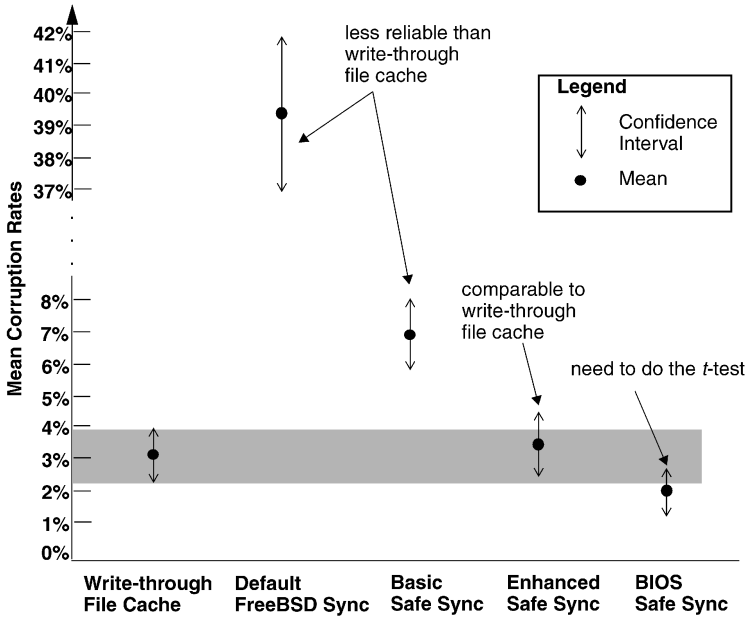


Fig. 3. Comparing Design Alternatives. This figure plots the 95 percent confidence interval of mean corruption rate for the write-through file cache and all our designs. We use the approximate visual test [35] to compare our designs.

4 Design Iterations

This section describes the four iterations we go through to design the Rio file cache. 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. Table 3 summarizes the four design iterations. Each design iteration is labeled by the method used to write data back to disk (a task known as "syncing" the data to disk) during a crash because this is the major determiner for the reliability of a write-back file cache.

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, 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 percent 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 [11].

4.1 Design Iteration 1: Default FreeBSD Sync

4.1.1 Design

We start the design process with the default sync used in FreeBSD. 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 2, 39 percent 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 also observe from Table 2 that there is no overlap between the 95 percent confidence intervals of default FreeBSD sync and the write-through file cache. This can be seen in Fig. 3, which depicts the mean corruption rate and 95 percent confidence intervals for the write-through file cache and all our designs. Thus, we conclude with 95 percent confidence that a write-back file cache that uses the default FreeBSD sync is less reliable than the 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

TABLE 4
Categories of Fault Systems for Default FreeBSD Sync

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	**589 of 1500 (39.3%)**	

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 [33]. 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 equally 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 run successfully and the injected bugs appear to be benign. We do not attempt to overcome this problem because we lack sufficient information and the frequency of this fault symptom is very low.

Table 4 summarizes the categories of faults symptoms and some potential solutions we developed (discussed in Section 4.2.1) to reduce the vulnerability of the system to each 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 the 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 most common fault symptom in Table 4.

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

TABLE 5
Categories of Fault Symptoms for Basic Safe Sync

Fault Symptom	# of Corruptions	Solutions Used in the Next Design (Section 4.3.1)
hang before sync	45 (3.0%)	watchdog timer, disable 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	**104 of 1500 (6.9%)**	

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 [37] by creating a new data structure called the *registry*. The registry contains all information needed to find, identify, and write to disk 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 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 [15]. 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 basic safe sync show substantial improvement over the default FreeBSD sync. Table 2 shows that basic safe sync has a corruption rate of 6.9 percent, which is six times better than the default FreeBSD sync. However, it still has twice as many corruptions as a write-through file cache. Fig. 3 shows that the 95 percent confidence intervals of basic safe sync and the write-through file cache do not overlap. We thus conclude, with 95 percent confidence, that this design is less reliable than a write-through file cache.

Table 5 breaks down the fault symptoms for our current design. The fault symptoms are very similar to those in Table 4, 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 5 shows that the reset button reduces the corruption rates substantially from the default FreeBSD sync (from 17.9 percent to 3.0 percent). 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 [37]. 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 can fix this fault by disabling debugging print statements during 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 4 and Table 5. The slight differences are due to the nondeterminism inherent in 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

TABLE 6
Categories of Fault Symptoms for Enhanced Safe Sync

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	**50 of 1500 (3.3%)**	

reinitialize 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 write over 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 will encounter multiple page faults when it tries to fault in subsequent pages from the bogus stack. To fix this, we preallocate a stack for safe sync during 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 reinitialize 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 preallocated, 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 percent, versus 6.9 percent for the basic safe sync design of iteration 2. Fig. 3 shows that there is significant overlap between the 95 percent confidence intervals of enhanced safe sync and the write-through file cache and the mean corruption rate of our new design falls in the confidence interval of the mean corruption rate of the write-through file cache. We thus conclude, with 95 percent confidence, that our new design has comparable reliability as a write-through file cache; we have thus reached our design goal of creating a write-back file cache that is as reliable as a write-through file cache.

Although we have reached our reliability goal, we choose to carry out one more design iteration to explore the limits of reliability for write-back file caches. Table 6 breaks down the fault symptoms of enhanced safe sync. Enhanced safe sync depends on two areas of kernel functionality. First, all kernel code, including safe sync, runs in virtual-addressing mode with paging enabled [33], 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 [56]). 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 accesses 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 [33]. We remove dependencies on the kernel device drivers by using the BIOS interface to the disk [22]. The BIOS (Basic Input/Output Service) interface is 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.

TABLE 7
Categories of Fault Symptoms for BIOS Safe Sync

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

Our final design (BIOS safe sync) replaces the enhanced safe sync code. The new safe sync procedure is summarized below:

- *Part 1: Initial setup*: We follow standard procedure to switch to real mode [33], which includes setting up 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. The 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 [32], and initializing the video console and disk controller using the BIOS interface [22]. 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 [33] and using a large data segment (i.e., big real-mode [60]) to access data beyond the first 1 MB of memory. During sync, we copy the file cache data into the lower 1MB 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 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 BIOS safe sync. Table 7 breaks down the fault symptoms for our latest design. The overall corruption rate is down to 1.9 percent, which is about 40 percent more reliable than a write-through file cache. Fig. 3 shows that the confidence intervals of BIOS safe sync and the write-through file cache overlap slightly. Because BIOS safe sync's mean corruption rate is not in the confidence interval of the write-through file cache, we use the *t*-test statistical procedure to compare the two designs [35]. We compute the mean difference in corruption rates between the two designs and the standard deviation of the mean difference. This allows us to derive the 95 percent confidence interval for the difference in mean corruption rates, which is (1.0-2.3 percent). Because the resulting confidence interval does not include zero, we conclude, with 95 percent confidence, that our BIOS safe sync design is more reliable than the write-through file cache.

While the difference in reliability between BIOS safe sync and the write-through file cache is not large, it is surprising that any write-back file cache is able to achieve even marginally higher reliability than a write-through file cache because write-back file caches have additional sources of vulnerability (e.g., hanging before sync). To understand how BIOS safe sync can achieve higher reliability, we compare the frequency of different fault symptoms for BIOS safe sync and the write-through file cache (Fig. 4). The difference in reliability is due primarily to an increase in file system errors and VM errors:

- Rio with BIOS safe sync is less vulnerable to *file system errors* than the write-through file cache because Rio rarely uses the full file system code. For most user-level write calls, Rio needs only to copy data to the file cache. Copying data to the file cache is a simple operation that uses very little of the file system code. This memory data is often overwritten or deleted before needing to be written to disk [5]; hence, Rio rarely uses the bulk of the complex file system code. In contrast, with a write-through file cache, each user-level write call initiates a complex traversal of file system functions to write the data to disk. When bugs are injected into the file system code (but before the system crashes), a write-through file cache can easily corrupt disk data, especially before the system begins to crash noticeably. Thus, errors in the file system code more easily corrupt data when using a write-through file cache than when using a write-back file cache.
- Rio with BIOS safe sync is less vulnerable to *VM errors* than the write-through file cache because Rio uses VM protection and relies on the registry to write data to disk during safe sync. In contrast, the write-through file cache uses virtual addressing and assumes that it has a valid VM mapping for the file cache data. So, a VM error may corrupt the VM mappings for a buffer page before the data is written to disk, causing a write-through file cache to write the wrong data to disk. Rio does not depend on these

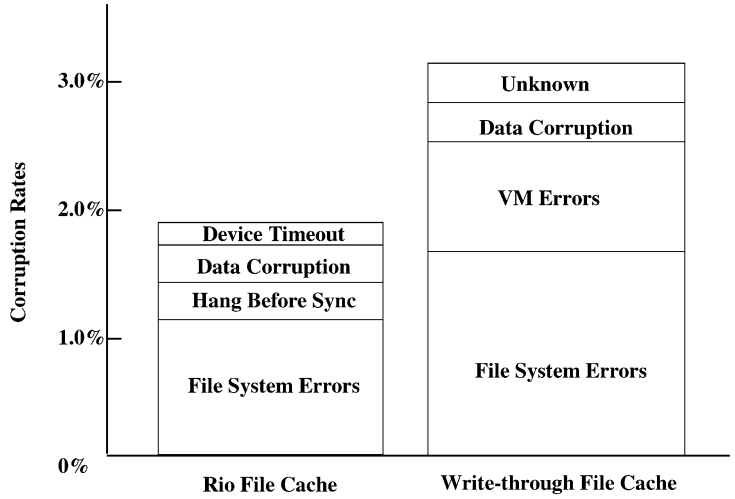


Fig. 4. Why is Rio with BIOS Safe Sync More Reliable? We breakdown the fault symptoms for both Rio file cache with BIOS safe sync and write-through file cache. The y-axis shows the mean corruption rates.

VM mappings. It already has the correct information in the registry, which contains the physical address of the buffer page and is protected by its restricted interface and by consistency checks performed when updating its contents. Its content is updated at the earliest instance when the page is written to and valid.

A potentially offsetting factor to Rio's robustness to file system and VM errors is the robustness of writing data to disk at the end of a crash. As noted before in this paper, a write-through file cache does not need to write data to disk at the end of a crash because it has written all data through to disk before the crash. In contrast, Rio must write data to disk using the safe sync routine. With BIOS safe sync, the code to write this data is small (250 lines of code), simple, and, therefore, robust. It is specialized to do a single type of I/O (block-level, synchronous write) and uses a protected BIOS I/O interface that resides in EEPROM. Because of this, the additional vulnerability of Rio during the end of the crash is less significant than the vulnerability of a write-through file cache to file system and VM errors before the crash.

It is important to note that Rio has marginally higher reliability than write-through file cache against *software* errors. We will elaborate on the limitations of Rio against hardware errors in Section 8.

We observe from Table 7 that it may be possible to improve 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 [15]. Doing so should fix the remaining hangs before sync and device timeouts, but it would incur significant system cost.

5 Effects of Large Memory and Virtual Memory Protection

In this section, we look more specifically at two related factors that may affect the relative reliability of Rio and write-through file caches. The first factor is the amount of physical memory. More memory allows Rio to cache more dirty file data and filter write traffic more effectively. Keeping more dirty file data in memory renders it more vulnerable to wild stores, so Rio's reliability may suffer when memory sizes increase. The second factor is virtual memory protection. VM protection may benefit write-through file caches, and this may also change the relative reliabilities of Rio and the write-through file cache. These two factors interact because VM protection reduces the likelihood of wild stores.

In this section, we use a slightly different hardware platform than in Section 4 (Intel Pentium processor, 128 or 512 MB memory, SCSI disk drive) due to our inventory of large memory configurations. As a result of the different hardware configuration and device drivers, the reliability data in this section cannot be compared directly with the data in Section 4 (in general, the reliability of the SCSI systems is slightly better, which we speculate is due to the increased intelligence in the SCSI controller, e.g., error checking). The Rio design used in this section is enhanced safe sync because of the difficulty of programming a portable BIOS safe sync for SCSI.

Table 8 shows the effect of main memory size on four types of file cache: Rio file cache without VM protection, Rio file cache with VM protection, write-through file cache without VM protection, and write-through file cache with VM protection. As expected, without protection, the corruption rate increases for the Rio file cache. Interestingly, the corruption rate also increases for the write-through file cache without protection.

We also observe that VM protection mitigates the reliability impact of larger memory sizes. With VM protection, both the write-through file cache and Rio maintain comparable reliability for different memory sizes. However, Rio needs VM protection to approach the reliability of the write-through file cache, in addition to needing it to maintain reliability for larger memories.

TABLE 8
Effect of Physical Memory Size on Reliability

Fault Type	Rio File Cache				Write-though File Cache			
	No Protection		Protect Buffer		No Protection		Protect Buffer	
	128 MB	512 MB	128 MB	512 MB	128 MB	512 MB	128 MB	512 MB
text	4	5	1	1	3	3	0	1
heap	0	0	0	0	0	1	3	1
stack	2	6	1	2	1	2	3	3
initialization	4	3	5	1	7	11	0	4
delete ran. inst.	6	12	5	6	2	8	4	3
source reg.	3	6	3	3	1	1	2	3
dest. reg.	1	4	4	2	1	2	1	2
delete branch	6	10	4	4	2	2	4	5
pointer	1	7	2	6	2	4	2	2
allocation	8	0	0	0	0	0	1	0
copy overrun	0	16	2	8	4	1	2	0
synch.	15	0	0	0	0	0	0	0
off-by-one	0	15	5	6	4	12	4	5
memory leak	0	0	0	0	0	0	0	1
interface error	2	0	1	1	0	1	2	0
Total (out of 1500)	**52 3.5%**	**84 5.6%**	**33 2.2%**	**40 2.7%**	**27 1.8%**	**48 3.2%**	**28 1.9%**	**30 2.0%**
95% Confidence Interval	**2.6 - 4.3%**	**4.8 - 6.4%**	**1.4 - 3.0%**	**1.8 - 3.5%**	**1.1 - 2.5%**	**2.3 - 4.1%**	**1.0 - 2.8%**	**1.3 - 2.7%**

This table shows the reliability of different file cache systems on a system with 128 MB and 512 MB main memory. The systems studied are Rio file cache with and without VM protection and the write-through file cache with and without VM protection.

6 Performance

The main benefit of Rio measured so far is reliability: All writes to the file cache are immediately as permanent and safe as files on disk. In this section, we show that Rio also improves performance by eliminating all reliability-induced writes to disk.

Table 9 compares the performance of our Rio file cache with different types of file systems, each providing different guarantees on when data is made permanent.[1] The file systems in Table 9 are ordered from least reliable (Memory File System) to most reliable (Rio). The *Memory File System* is shown to illustrate optimal performance [48]. The Memory File System is very fast because it is completely memory-resident, does no disk I/O, and is hence very simple. However, data stored in the Memory File System is never made permanent; it is simply discarded when the system shuts down or crashes. UFS (Unix File System) is the default FreeBSD Unix file system. We measure several variants. *UFS pure write-back* is UFS without any reliability-induced writes. It demonstrates the performance of a pure write-back file system (whereas MFS demonstrates the performance of a memory-only file system). *UFS pure write-back* can lose arbitrarily old data during a crash. *UFS delayed data and metadata* is similar to *UFS pure write-back*, but syncs its data and metadata to disk every 30 seconds. This is the optimal "no-order" system in [21] and is faster than soft update [50] file systems as it does not need to maintain dependencies between updates. *UFS delayed data and metadata* loses up to 30 seconds of data and metadata during a crash. The standard UFS uses a combination of write-back policies. It writes data asynchronously to disk when 64 KB of data has been collected, when the user writes nonsequentially, or when the sync daemon flushes dirty file data (once every 30 seconds). UFS writes metadata synchronously to disk to enforce ordering constraints [21]. We also measure the behavior of a write-through file cache which writes all user data synchronously to disk. This achieves comparable reliability as Rio, but at significant performance penalty. The Rio file cache uses the enhanced safe sync (with protection) described in Section 5.

We run two workloads, cp+rm and Andrew. cp+rm recursively copies then removes the FreeBSD source tree (32 MB). Andrew models a software development workload. All results represent an average of 20 runs.

1. The data in Table 9 is different from the data in [52] for the following reasons: The configuration in [52] limited the amount of metadata stored in the file cache, used a smaller workload for cp+rm, and incorrectly uses the "async" version of UFS, which performs most metadata writes to disk asynchronously and does not guarantee file system integrity.

TABLE 9
Performance Comparison

File System	cp+rm			Andrew		
	time (sec)	# sync disk writes	#async disk writes	time (sec)	# sync disk writes	#async disk writes
Memory File System	2.80	0	0	1.20	0	0
UFS pure write-back	3.05	0	0	1.20	0	0
UFS with delayed data and metadata	5.95	0	354	1.30	0	5
UFS	119.85	5725	9320	5.15	4725	4031
write-through file cache	175.11	19673	8310	14.60	24150	4366
Rio file cache	**3.20**	**0**	**0**	**1.25**	**0**	**0**

This table compares the running time and number of synchronous and asynchronous disk writes for different file systems. Each file system provides different reliability guarantees and they are ordered from least reliable (Memory File System) to most reliable (Rio). cp+rm recursively copies, then recursively removes the FreeBSD source tree (32 MB); Andrew models software development. All performance measurements were made on a PC with a 400 MHz Pentium-II processor, 128 MB of 100 MHz SDRAM, and an IBM DCAS-34330W SCSI disk (with the disk write-cache enabled). For all file systems, FreeBSD was configured to enable the file cache to store all metadata used in each workload.

Table 9 shows that our Rio file cache is 12-55 times as fast as a write-through file cache which has comparable reliability. It is also significantly faster (4-37 times) than the standard Unix file system, yet provides much stronger reliability guarantees. Rio's performance approaches that of an optimal memory-based file system and pure write-back file system.

Rio is significantly faster than the write-through and delayed write file caches because it eliminates reliability-induced writes to disk. Table 9 breaks down the number of synchronous and asynchronous disk writes for each type of file systems. File systems with no synchronous writes (Rio and the pure write-back and delayed write systems) perform significantly better than the other file systems that have a large number of synchronous writes. Rio also does not need to periodically sync its file cache data to disk, which enables Rio to outperform the delayed write file system.

7 Related Work

We divide the research related to this paper into four areas: other reliable memory systems, field studies of failures, fault injection, and low-latency writes.

7.1 Reliable Memory Systems

Several researchers have proposed ways to protect memory from software failures [17], though, to our knowledge, none have evaluated how effectively memory withstood these failures.

The only file system we are aware of that attempts to make all permanent files reliable while in memory is Phoenix [20]. Phoenix keeps two versions of an in-memory file system. One of these versions is kept write-protected; the other version is unprotected and evolves from the write-protected one via copy-on-write. At periodic checkpoints, the system write-protects the unprotected version and deletes obsolete pages in the original version. Our proposed mechanism in Section 4 differs from Phoenix in two major ways: 1) Phoenix does not ensure the reliability of every write; instead, writes are only made permanent at periodic checkpoints; 2) Phoenix keeps multiple copies of modified pages, while we keep only one copy.

Rio strives to make the memory on a single machine reliable. An orthogonal approach to improving the reliability of memory is to replicate its contents in the memory of several independent computers, as was done in Harp [44] and Network RAM [55]. The Recovery Box keeps a special system state in a region of memory accessed only through a rigid interface [7]. No attempt is made to prevent other procedures from accidentally modifying the recovery box, although the system detects corruption by maintaining checksums. Banatre et al. implement stable transactional memory, which protects memory contents with dual memory banks, a special memory controller, and explicit calls to allow write access to specified memory blocks [8], [9]. Our work seeks to make all files in memory reliable without special-purpose hardware or replication.

General mechanisms may be used to help protect memory from software faults. Needham et al. [51] suggest changing a machine's microcode to check certain conditions when writing a memory word. This is similar to modifying the memory controller to enforce protection, as are Johnson's and Wahbe's suggestions for various hardware mechanisms to trap the updates of certain memory locations [36], [70]. Hive uses the Flash firewall to protect memory against wild writes by other processors in a multiprocessor [14]. Hive preemptively discards pages that are writable by failed processors, an option not available when storing permanent data in memory. Object code modification has been suggested as a way to provide data breakpoints [40], [70] and fault isolation between software modules [71]. One can also isolate memory from the operating system by hiding the memory [19] via a device driver interface or by physically isolating the memory via a hardware interface. The former approach requires user applications and kernel to access memory via a protected

device driver interface. The latter requires a hardware interface to memory (e.g., solid state disks). These approaches are significantly slower than Rio as memory references are several order of magnitude faster than I/O instructions [33].

Other projects seek to improve the reliability of memory against hardware faults such as power outages and board failures. eNVy implements a memory board based on nonvolatile, flash RAM [73]. eNVy uses copy-on-write, page remapping, and a small, battery-backed, SRAM buffer to hide flash RAM's slow writes and bulk erases. The Durable Memory RS/6000 uses batteries, replicated processors, memory ECC, and alternate paths to tolerate a wide variety of hardware failures [1].

Finally, many papers have examined the performance advantages and uses of reliable memory [17], [6], [12], [2], [45], [46].

7.2 Field Studies of Failures

Studies have shown that software is the dominant cause of system outages [23], [24] and several studies have investigated system software errors. Sullivan and Chillarege classify software faults in the MVS operating system; in particular, they analyze faults that corrupt program memory (overlays) [65]. Lee and Iyer study and classify software failures in Tandem's Guardian operating system [43]. These studies provide valuable information about failures in production environments; in fact, many of the fault types in Section 2.1 were inspired by the major error categories from [65] and [43]. However, these studies do not provide data on how often system crashes corrupt the file cache, which may have different failure characteristics than randomly accessed data structures [67].

7.3 Fault Injection

Software fault injection can be used for many purposes, such as understanding how systems behave during a fault, validating fault-tolerant mechanisms, and comparing the robustness of different systems. See [34] for an excellent introduction to the overall area and a summary of much of the past work on fault injection.

Fault injection is traditionally used to understand how systems behave during a fault. Chillarege and Bowman [16] use fault injection to characterize large system failures. They inject software bugs on a commercial transaction processing system, and analyze the crash data to measure system component failure rates and fault latency. Barton et al. [10] use the FIAT fault injection tool to inject memory bit faults into data and code of two different applications. They produce detailed statistics on fault manifestations and error detection latencies. Kao et al. [39] use the FINE fault injection tool to study fault propagation in the UNIX operating system and construct a fault propagation model based on Markov chains.

Fault injection is also widely used to validate fault-tolerant mechanisms and system dependability. Arlat et al. [4] use fault injection to validate the dependability of fault-tolerant systems against transient hardware/software faults. Hudak et al. [31] conduct fault injection tests to determine the effectiveness of various fault-tolerant software techniques, such as n-version programming, against design and hardware faults. Rela et al. [57] use fault injection to evaluate the effectiveness of software consistency checks against transient hardware faults. They inject pin-level faults into a processor and measure its effect on software applications. Silva et al. [64] use the Xception fault injection tool [13] to injection transient faults into parallel computers running large applications. They measure the effectiveness of various fault tolerant techniques (e.g., memory protection, assertions) in increasing the system's error detection coverage.

Several papers use fault injection to compare the robustness of different systems. Siewiorek et al. [61] propose a benchmark suite to measure system robustness using fault injection. The system components under study include the file system, memory management, fault-tolerant mechanisms in user application, and C library functions. Tsai et al. [69] extend the benchmark approach to dependable systems. They used the FTAPE tool to evaluate two fault-tolerant computers. Koppman et al. measure the robustness of various commercial UNIX operating systems [42] and different releases of the same operating systems [41]. They inject faults into system calls by changing the input data values and observe the system response.

7.4 Low-Latency Writes

The main techniques for doing low-latency writes to disk is by staging the writes to a faster nonvolatile device like NVRAM or by minimizing disk head movements during disk writes. The former technique is used extensively in commercial products like Prestoserve [47] and NetApp Filer [29]. But, it is expensive as it uses SRAM or dual battery-backed DRAM. It is also slower than main memory as the NVRAM typically resides on an I/O bus. The latter approach is traditionally done by writing data to sequential locations in disk. Write-ahead logging systems [25] accumulate small updates in a log and replay the modifications later by updating in place. These systems typically use a separate log disk to avoid conflict with reads. The Log-structured File System [58], [59] writes data to disk sequentially in a log-like structure. The log contains indexing information so that file data can be read back from the log efficiently. Another way to minimize disk head movement is to write to a disk location that is close to the current disk head location [72].

There are also hybrid approaches that combine NVRAM with disk head minimizing techniques during writes. In DCD [30], disk writes are first staged to NVRAM and later written sequentially to a cache disk. The data in the cache disk is subsequently updated in place in the data disk. The two level cache (NVRAM+cache disk) thus appear to the host as a large NVRAM cache with a size close to the cache disk size. The NVRAM can be replaced by main memory with a slight trade-off in reliability [53].

The main difference between Rio and other low-latency write schemes is that Rio writes directly to main memory, rather than to disk or an NVRAM board on an I/O bus. Main memory is much faster than even sequential disk I/O and main memory is typically larger and faster to access than NVRAM boards on the I/O bus. We view Rio as adding a new layer (reliable main memory) to the storage

hierarchy. Logging and other disk optimizations can then be added to accelerate the transfers between Rio and disk.

An important contribution of our work relative to other uses of nonvolatile memory is that we quantify the reliability implications of our design choices. This is especially important for us to do because we are potentially exposing memory to more errors (because it is not hidden behind a device driver interface). Most NVRAM systems preserve a disk-like interface and thus have similar reliability as a write-through file cache.

8 Limitations

The main focus of Rio is to protect data against software errors. It would be dangerous to apply our results indiscriminately against other types of system failures and, hence, we address in this section some possible limitations of our work.

We focus on hardening our design against software errors as many studies have shown that software errors have become a major cause of system crash [23], [24], [19] and because many system designers [1], [62] feel it is easier to make fault-tolerant hardware than software. Rio without any complementary hardware techniques will *not* protect against hardware errors like equipment failures, processor faults, bus failures, etc. We have explored various architecture support for reliable file cache in [15] to make Rio as reliable as write-through file cache against hardware errors.

Our fault injection experiments use synthetic faults derived from field studies of system crashes and the system we designed may not be fault-tolerant against "real" system crashes. To further test and prove our ideas, we have installed a file server in our department using the Rio file cache with protection and with reliability-induced writes to disk turned off. Among other things, this file server (*rio.eecs.umich.edu*) stores our kernel source tree, the only copy of this paper, and our mail. The server has been operational for the past four years and we experienced more than 10 software crashes. The Rio file cache worked remarkably well during these crashes and we did not lose any user data. The source code for our system can be found at http://www.eecs.umich.edu/Rio.

9 Conclusions

We have described the design and verification of the Rio file cache. By using a systematic, quantitative approach, we were able to improve the reliability of a write-back file cache by a factor of 21. The resulting system had comparable reliability to a write-through file system while outperforming it by a factor of 12-55. Rio also outperformed the standard Unix file system by a factor of 4-37 while providing much stronger reliability guarantees.

The Rio file cache provides a new type of persistent storage with intriguing characteristics. It is as fast as main memory, directly addressable, and as reliable as disk from operating system crashes. We have found this type of storage very useful in our work on lightweight transactions and distributed recovery. In particular, the direct addressability of memory enables applications to save persistent data without system calls or extra copying and the random-access, high-speed nature of memory enables applications to save data using much simpler logging and layout strategies than those required for disk. We look forward to seeing how other system designers use this new storage medium.

Acknowledgments

This research was supported in part by US National Science Foundation (NSF) grant MIP-9521386, AT&T Labs, the IBM University Partnership Program, and Intel Technology for Education 2000. Peter Chen was also supported by NSF CAREER Award MIP-9624869.

References

[1] M. Abbott, D. Har, L. Herger, M. Kauffmann, K. Mak, J. Murdock, C. Schulz, T.B. Smith, B. Tremaine, D. Yeh, and L. Wong, "Durable Memory RS/6000 System Design," *Proc. 1994 Int'l Symp. Fault-Tolerant Computing (FTCS),* pp. 414-423, June 1994.

[2] S. Akyurek and K. Salem, "Management of Partially Safe Buffers," *IEEE Trans. Computers,* vol. 44, no. 3, pp. 394-407, Mar. 1995.

[3] "The Power Protection Handbook,"technical report, Am. Power Conversion, 1996.

[4] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell, "Fault Injection for Dependability Validation: A Methodology and Some Applications," *IEEE Trans. Software Eng.,* vol. 16, no. 2, pp. 166-182, Feb. 1990.

[5] M.G. Baker, J.H. Hartman, M.D. Kupfer, K.W. Shirriff, and J.K. Ousterhout, "Measurements of a Distributed File System," *Proc. 13th ACM Symp. Operating Systems Principles,* pp. 198-212, Oct. 1991.

[6] M. Baker, S. Asami, E. Deprit, J. Ousterhout, and M. Seltzer, "Non-Volatile Memory for Fast Reliable File Systems," *Proc. Fifth Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS-V),* pp. 10-22, Oct. 1992.

[7] M. Baker and M. Sullivan, "The Recovery Box: Using Fast Recovery to Provide High Availability in the UNIX Environment," *Proc. USENIX Summer Conf.,* June 1992.

[8] M. Banatre, G. Muller, B. Rochat, and J. Banatre, "Ensuring Data Security and Integrity with a Fast Stable Storage," *Proc. 1988 Int'l Conf. Data Eng.,* pp. 285-293, Feb. 1988.

[9] M. Banatre, G. Muller, B. Rochat, and P. Sanchez, "Design Decisions for the FTM: A General Purpose Fault Tolerant Machine," *Proc. 1991 Int'l Symp. Fault-Tolerant Computing,* pp. 71-78, June 1991.

[10] J.H. Barton, E.W. Czeck, Z.Z. Segall, and D.P. Siewiorek, "Fault Injection Experiments Using FIAT," *IEEE Trans. Computers,* vol. 39, no. 4, pp. 575-582, Apr. 1990.

[11] A. Bensoussan, C.T. Clingen, and R.C. Daley, "The Multics Virtual Memory: Concepts and Design," *Comm. ACM,* vol. 15, no. 5, pp. 308-318, May 1972.

[12] P. Biswas, K.K. Ramakrishnan, D. Towsley, and C.M. Krishna, "Performance Analysis of Distributed File Systems with Non-Volatile Caches," Proc. 1993 Int'l Symp. High Performance Distributed Computing (HPDC-2), pp. 252-262, July 1993.

[13] J. Carreira, H. Madeira, and J.G. Silva, "Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers," *IEEE Trans. Software Eng.,* vol. 24, no. 2, pp. 125-136, Feb. 1998.

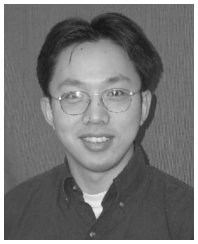
[14] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta, "Hive: Fault Containment for Shared-Memory Multiprocessors," *Proc. 1995 Symp. Operating Systems Principles,* Dec. 1995.

[15] P.M. Chen, W.T. Ng, S. Chandra, C.M. Aycock, G. Rajamani, and D. Lowell, "The Rio File Cache: Surviving Operating System Crashes," *Proc. 1996 Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS),* pp. 74-83, Oct. 1996.

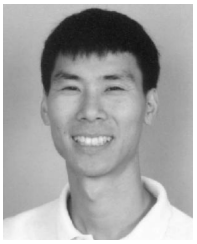
[16] R. Chillarege and N.S. Bowen, "Understanding Large System Failure—A Fault Injection Experiment," *Proc. 1989 Int'l Symp. Fault-Tolerant Computing (FTCS),* pp. 356-363, 1989.

[17] G. Copeland, T. Keller, R. Krishnamurthy, and M. Smith, "The Case for Safe RAM," *Proc. 15th Int'l Conf. Very Large Data Bases,* pp. 327-335, Aug. 1989.
[18] D.J. DeWitt, R.H. Katz, F. Olken, L.D. Shapiro, M.R. Stonebraker, and D. Wood, "Implementation Techniques for Main Memory Database Systems," *Proc. 1984 ACM SIGMOD Int'l Conf. Management of Data,* pp. 1-8, June 1984.
[19] F. Eskesen, M. Hack, A. Iyengar, R.P. King, and N. Halim, "Software Exploitation of a Fault-Tolerant Computer with a Large Memory," *Proc. 1998 Symp. Fault-Tolerant Computing (FTCS),* pp. 336-345, June 1998.
[20] J. Gait, "Phoenix: A Safe In-Memory File System," *Comm. ACM,* vol. 33, no. 1, pp. 81-86, Jan. 1990.
[21] G.R. Ganger and Y.N. Patt, "Metadata Update Performance in File Systems," *Proc. 1994 Operating Systems Design and Implementation (OSDI),* Nov. 1994.
[22] F. Van Gilluwe, *The Undocumented PC: A Programmer's Guide to I/O, CPUs, and Fixed Memory Areas.* Addison-Wesley Developer Press, 1997.
[23] J. Gray, "A Census of Tandem System Availability between 1985 and 1990," *IEEE Trans. Reliability,* vol. 39, no. 4, Oct. 1990.
[24] J. Gray and D. Siewiorek, "High-Availability Computer Systems," *Computer,* vol. 24, no. 9, Sept. 1991.
[25] T. Haerder and A. Reuter, "Principles of Transaction-Oriented Database Recovery," *ACM Computing Surveys,* vol. 15, no. 4, pp. 287-317, Dec. 1983.
[26] R.B. Hagmann, "Reimplementing the Cedar File System Using Logging and Group Commit," *Proc. 1987 Symp. Operating Systems Principles,* pp. 155-162, Nov. 1987.
[27] J.H. Hartman and J.K. Ousterhout, "Letter to the Editor," *Operating Systems Review,* vol. 27, no. 1, pp. 7-9, Jan. 1993.
[28] J.L. Hennessy and D.A. Patterson, *Computer Architecture: A Quantitative Approach,* second ed., p. 493. Morgan Kaufmann, 1990.
[29] D. Hitz, J. Lau, and M. Malcolm, "File System Design for an NFS File Server Appliance," *Proc. 1994 USENIX Winter Conf.,* Jan. 1994.
[30] Y. Hu and Q. Yang, "DCD—Disk Caching Disk: A New Approach for Boosting I/O Performance," *Proc. 1996 Int'l Symp. Computer Architecture,* May 1996.
[31] J. Hudak, B.-H. Suh, D. Siewiorek, and Z. Segall, "Evaluation and Comparison of Fault-Tolerant Software Techniques," *IEEE Trans. Reliability,* vol. 42, no. 2, June 1993.
[32] "Intel 82371AB PCI ISA IDE Xcelerator (PIIX4) Datasheet," Intel Corp., 1997.
[33] "Intel Architecture Software Developer's Manual: Volumes 1-3," Intel Corp., 1997.
[34] R.K. Iyer, "Experimental Evaluation," *Proc. 1995 Int'l Symp. Fault-Tolerant Computing,* pp. 115-132, July 1995.
[35] R. Jain, *The Art of Computer Systems Performance Analysis.* John Wiley & Sons, 1991.
[36] M.S. Johnson, "Some Requirements for Architectural Support of Software Debugging," *Proc. 1982 Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS),* pp. 140-148, Apr. 1982.
[37] B.W. Johnson, *Design and Analysis of Fault-Tolerant Digital Systems.* Addison-Wesley, 1989.
[38] G.A. Kanawati, N.A. Kanawati, and J.A. Abraham, "FERRARI: A Flexible Software-Based Fault and Error Injection System," *IEEE Trans. Computers,* vol. 44, no. 2, pp. 248-260, Feb. 1995.
[39] W.-L. Kao, R.K. Iyer, and D. Tang, "FINE: A Fault Injection and Monitoring Environment for Tracing the UNIX System Behavior under Faults," *IEEE Trans. Software Eng.,* vol. 19, no. 11, pp. 1105-1118, Nov. 1993.
[40] P.B. Kessler, "Fast Breakpoints: Design and Implementation," *Proc. 1990 Conf. Programming Language Design and Implementation (PLDI),* pp. 78-84, June 1990.
[41] P.J. Koopman and J. DeVale, "Comparing the Robustness of POSIX Operating Systems," *Proc. 1999 Symp. Fault-Tolerant Computing (FTCS),* June 1999.
[42] N.P. Kropp, P.J. Koopman, and D.P. Siewiorek, "Automated Robustness Testing of Off-the-Shelf Software Components" *Proc. 1998 Symp. Fault-Tolerant Computing (FTCS),* June 1998.
[43] I. Lee and R.K. Iyer, "Faults, Symptoms, and Software Fault Tolerance in the Tandem GUARDIAN Operating System," *Proc. 1993 Int'l Symp. Fault-Tolerant Computing (FTCS),* pp. 20-29, 1993.
[44] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams, "Replication in the Harp File System," *Proc. 1991 Symp. Operating System Principles,* pp. 226-238, Oct. 1991.
[45] D.E. Lowell and P.M. Chen, "Free Transactions with Rio Vista," *Proc. 1997 Symp. Operating Systems Principles,* pp. 92-101, Oct. 1997.
[46] D.E. Lowell, S. Chandra, and P.M. Chen, "Exploring Failure Transparency and the Limits of Generic Recovery," *Proc. 2000 Operating Systems Design and Implementation (OSDI),* Oct. 2000.
[47] B. Lyon and R. Sandberg, "Breaking through the NFS Performance Barrier," technical report, Legato Systems, Inc., 1990.
[48] M.K. McKusick, M.J. Karels, and K. Bostic, "A Pageable Memory Based Filesystem," *Proc. USENIX Summer Conf.,* June 1990.
[49] M.K. McKusick, K. Bostic, M.J. Karels, and J.S. Quarterman, *The Design and Implementation of the 4.4BSD Operating System.* Addison-Wesley, 1996.
[50] M.K. McKusick and G.R. Ganger, "Soft Updates: A Technique for Eliminating Most Synchronous Writes in the Fast Filesystem," *Proc. 1999 USENIX Ann. Technical Conf.: FREENIX Track,* June 1999.
[51] R.M. Needham, A.J. Herbert, and J.G. Mitchell, "How to Connect Stable Memory to a Computer," *Operating System Review,* vol. 17, no. 1, p. 16, Jan. 1983.
[52] W.T. Ng and P.M. Chen, "The Systematic Improvement of Fault Tolerance in the Rio File Cache," *Proc. 1999 Symp. Fault-Tolerant Computing (FTCS),* June 1999.
[53] T. Nightingale, Y. Hu, and Q. Yang, "The Design and Implementation of a DCD Device Driver for Unix," *Proc. 1999 USENIX Technical Conf.,* June 1999.
[54] J.K. Ousterhout et al., "A Trace-Driven Analysis of the UNIX 4.2 BSD File System," *Proc. 1985 Symp. Operating System Principles,* pp. 15-24, Dec. 1985.
[55] D. Pnevmatikatos, E.P. Markatos, G. Magklis, and S. Ioannidis, "On Using Network RAM as a Non-Volatile Buffer," *Cluster Computing,* vol. 2, pp. 295-303, 1999.
[56] R.F. Rashid Jr., A. Tevanian, M. Young, D. Golub, R. Baron, D. Black Jr., W.J. Bolosky, and J. Chew, "Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures," *IEEE Trans. Computers,* vol. 37, no. 8, pp. 896-908, Aug. 1988.
[57] M.Z. Rela, H. Madeira, and J.G. Silva, "Experimental Evaluation of the Fail-Silent Behavior in Programs with Consistency Checks," *Proc. 1996 Symp. Fault-Tolerant Computing (FTCS),* June 1996.
[58] M. Rosenblum, "The Design and Implementation of a Log-structured File System," PhD thesis, Univ. of California at Berkeley, June 1992.
[59] M. Rosenblum and J.K. Ousterhout, "The Design and Implementation of a Log-Structured File System," *ACM Trans. Computer Systems,* vol. 10, no. 1, pp. 26-52, Feb. 1992.
[60] T. Shanley, *Protected Mode Software Architecture.* Addison-Wesley Developer Press, 1996.
[61] D.P. Siewiorek, J.J. Hudak, B.-H. Suh, and Z. Segal, "Development of a Benchmark to Measure System Robustness," *Proc. 1993 Int'l Symp. Fault-Tolerant Computing,* pp. 88-97, June 1993.
[62] D.P. Siewiorek, *Reliable Computer Systems: Design and Evaluation.* A K Peters, 1998.
[63] A. Silberschatz and P.B. Galvin, *Operating System Concepts.* Addison-Wesley, 1994.
[64] J.G. Silva, J. Carreira, H. Madeira, D. Costa, and F. Moreira, "Experimental Assessment of Parallel Systems," *Proc. 1996 Symp. Fault-Tolerant Computing (FTCS),* June 1996.
[65] M. Sullivan and R. Chillarege, "Software Defects and Their Impact on System Availability—A Study of Field Failures in Operating Systems," *Proc. 1991 Int'l Symp. Fault-Tolerant Computing,* June 1991.
[66] M. Sullivan and R. Chillarege, "A Comparison of Software Defects in Database Management Systems and Operating Systems," *Proc. 1992 Int'l Symp. Fault-Tolerant Computing,* pp. 475-484, July 1992.
[67] M. Sullivan personal communication, Dec. 1995.
[68] A.S. Tanenbaum, *Distributed Operating Systems.* Prentice Hall, 1995.
[69] T.K. Tsai, R.K. Iyer, and D. Jewett, "An Approach towards Benchmarking of Fault-Tolerant Commercial Systems," *Proc. 1996 Symp. Fault-Tolerant Computing (FTCS),* June 1996.
[70] R. Wahbe, "Efficient Data Breakpoints," *Proc. 1992 Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS),* Oct. 1992.
[71] R. Wahbe, S. Lucco, T.E. Anderson, and S.L. Graham, "Efficient Software-Based Fault Isolation," *Proc. 14th ACM Symp. Operating Systems Principles,* pp. 203-216, Dec. 1993.

[72] R.Y. Wang, T.E. Anderson, and D.A. Patterson, "Virtual Log Based File Systems for a Programmable Disk," *Proc. 1995 Symp. Operating Systems Principles,* pp. 29-43, Feb. 1999.
[73] M. Wu and W. Zwaenepoel, "eNVy: A Non-Volatile, Main Memory Storage System," *Proc. 1994 Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS),* Oct. 1994.



Wee Teck Ng received a BS degree in electrical engineering from the University of Houston in 1986, an MEng degree in electrical engineering from the National University of Singapore, Singapore, in 1994, and MS and PhD degrees in computer science and engineering from the University of Michigan, Ann Arbor, in 1996 and 1999, respectively. He is currently a researcher at Bell Laboratories, Murray Hill, New Jersey. His research interests are in the area of storage systems, operating systems, and main memory databases.



Peter M. Chen received a BS degree in electrical engineering from the Pennsylvania State University in 1987 and MS and PhD degrees in computer science from the University of California at Berkeley in 1989 and 1992. He is currently an associate professor in the Department of Electrical Engineering and Computer Science at the University of Michigan at Ann Arbor. His research interests include operating systems, distributed systems, and fault-tolerant computing. He is a senior member of the IEEE.