1996 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)

The Rio File Cache: Surviving Operating System Crashes

Peter M. Chen, Wee Teck Ng, Subhachandra Chandra, Christopher Aycock, Gurushankar Rajamani, David Lowell

Computer Science and Engineering Division
Department of Electrical Engineering and Computer Science
University of Michigan
rio@eecs.umich.edu

Abstract: One of the fundamental limits to high-performance, high-reliability file systems is memory's vulnerability to system crashes. Because memory is viewed as unsafe, systems periodically write data back to disk. The extra disk traffic lowers 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 memory enables a system to achieve the best of both worlds: reliability equivalent to a write-through file cache, where every write is instantly safe, and performance equivalent to a pure write-back cache, with no reliability-induced writes to disk. To achieve reliability, we protect memory during a crash and restore it during a reboot (a "warm" reboot). Extensive crash tests show that even without protection, warm reboot enables memory to achieve reliability close to that of a write-through file system. Adding protection makes memory even *safer* than a write-through file system while adding essentially no overhead. By eliminating reliability-induced disk writes, Rio performs 4-22 times as fast as a write-through file system, 2-14 times as fast as a standard Unix file system, and 1-3 times as fast as an optimized system that risks losing 30 seconds of data and metadata.

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 long-term data such as files. However, random-access memory is viewed as an unreliable place to store permanent data (files) because it is vulnerable to power outages and operating system crashes.

Memory's vulnerability to power outages is easy to understand and fix. A $119 uninterruptible power supply can keep a system running long enough to dump memory to disk in the event of a power outage [APC96], or one can use non-volatile memory such as Flash RAM [Wu94]. We do not consider power outages further in this paper.

Memory's vulnerability to OS 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 [Tanenbaum95, Silberschatz94, Hennessy90]. Consequently, file systems periodically write data to disk, and transaction processing applications view transactions as committed only when data is written to disk. The goal of the Rio file cache is to enable memory to survive operating system crashes without writing data to disk.

Memory's perceived unreliability forces a tradeoff between performance and reliability. Applications requiring high reliability, such as transaction processing, write data synchronously through to disk, but this limits throughput to that of disk. While optimizations such as logging and group commit can increase effective disk throughput [Rosenblum92, Hagmann87, DeWitt84], disk throughput is still far slower than memory throughput.

Most Unix file systems mitigate the performance lost in synchronous, reliability-induced writes by *asynchronously* writing data 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. Users have learned to live with the fact that their data may not be safe on disk when a write or close finishes, and applications that require more safety, such as editors, explicitly flush the file to disk with an fsync call.

Some file systems improve performance further by delaying some writes to disk in the hopes of the data being deleted or overwritten [Ousterhout85]. 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 [Baker91, Hartman93], so a large fraction of writes must eventually be 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. This can only be done by applications for which reliability is not an issue, such as compilers that write temporary files.

This research was supported in part by NSF grant MIP-9521386, Digital Equipment Corporation, and the University of Michigan. Peter Chen was also supported by an NSF CAREER and Research Initiation Award (MIP-9624869 and MIP-9409229).

The goal of the Rio (RAM I/O) file cache is to achieve the *performance of main memory with the reliability of disk:* write-back performance with write-through reliability. We achieve memory performance by eliminating all reliability-induced writes to disk [McKusick90, Ohta90]. We achieve write-through reliability by protecting memory during a crash and restoring it during a reboot (a "warm" reboot). Extensive crash tests show that even without protection, warm reboot enables memory to achieve reliability close to that of a write-through file system. Adding protection makes memory even *safer* than a write-through file system while adding essentially no overhead. By eliminating reliability-induced writes to disk, Rio performs 4-22 times as fast as a write-through file system, 2-14 times as fast as a standard, delayed-write file system, and 1-3 times as fast as an optimized system that risks losing 30 seconds of data and metadata.

2 Design and Implementation of a Reliable File Cache

This section describes how we modify an existing operating system to enable the files in memory (the file cache) to survive crashes.

We use DEC Alpha workstations (DEC 3000/600) running Digital Unix V3.0 (OSF/1), a monolithic kernel based on Mach 2.5. Digital Unix stores file data in two distinct buffers in memory. Directories, symbolic links, inodes, and superblocks are stored in the traditional Unix buffer cache [Leffler89], while regular files are stored in the Unified Buffer Cache (UBC). The buffer cache is stored in wired virtual memory and is usually only a few megabytes. To conserve TLB slots, the UBC is not mapped into the kernel's virtual address space; instead it is accessed using physical addresses. The virtual memory system and UBC dynamically trade off pages depending on system workload. For the I/O-intensive workloads we use in this paper, the UBC uses 80 MB of the 128 MB on each computer.

2.1 Protection

The first step in enabling the file cache to survive a crash is to ensure that the system does not accidentally overwrite the file cache while it is crashing. The reason most people view battery-backed memory as vulnerable during a crash yet view disk as protected is the *interface* used to access the two storage media. The interface used to access disks is explicit and complex. Writing to disk uses device drivers that form I/O control blocks and write to I/O registers. Calls to the device driver are checked for errors, and procedures that do not use the device driver are unlikely to accidentally mimic the complex actions performed by the device driver. In contrast, the interface used to access memory is simple—any store instruction by any kernel procedure can easily change any data in memory simply by using the wrong address. It is hence relatively easy for many simple software errors (such as de-referencing an uninitialized pointer) to accidentally corrupt the contents of memory [Baker92a].

The main issue in protection is how to control accesses to the file cache. We want to make it unlikely that non-file-cache procedures will accidentally corrupt the file cache, essentially making the file cache a protected module within the monolithic kernel. To accomplish this, we use ideas from existing protection techniques such as virtual memory [Sullivan91a] and sandboxing [Wahbe93].

At first glance, the virtual memory protection of a system seems ideally suited to protect the file cache from unauthorized stores [Copeland89, Sullivan91a]. By turning off the write-permission bits in the page table for file cache pages, the system will cause most unauthorized stores to encounter a protection violation. File cache procedures must enable the write-permission bit in the page table before writing a page and disable writes afterwards. The only time a file cache page is vulnerable to an unauthorized store is while it is being written, and disks have the same vulnerability, because a disk sector being written during a system crash can be corrupted. File cache procedures can check for corruption during this window by verifying the data after the write. Or the file cache procedures can create a shadow copy in memory and implement atomic writes.

Unfortunately, many systems allow certain kernel accesses to bypass the virtual memory protection mechanism and directly access physical memory [Kane92, Sites92]. For example, addresses in the DEC Alpha processor with the two most significant bits equal to `10` bypass the TLB; these are called *KSEG* addresses. This is especially significant on the DEC Alpha because the bulk of the file cache (the UBC) is accessed using physical addresses. We have implemented two different methods to protect against these physical addresses.

Our current method disables the ability of the processor to bypass the TLB, that is, all addresses are mapped through the TLB. This can be done on the Alpha 21064, Intel x86, Sparc, PowerPC, and possibly other CPUs. On the Alpha 21064, a bit in the ABOX CPU control register can be set to map all KSEG addresses through the TLB. The page tables must be expanded to map these KSEG addresses to their corresponding physical address so the kernel can still access data such as page tables and the UBC. While issuing a KSEG address accesses the same memory location as before, this method enables the system to write-protect file cache pages. Disabling KSEG addresses in this manner adds essentially no overhead (Section 4).

A second method called *code patching* can be used for processors that cannot prevent physical addresses from bypassing the TLB. Code patching modifies the kernel object code by inserting a check before every kernel store [Wahbe93]. If the address is a physical address, the inserted code checks to make sure the address is not in the file cache, or that the file cache has explicitly registered the address as writable. Even after a number of optimizations to reduce the number of checks, the performance of code patching is 20-50% slower than with our current protection method [Chen96]. Hence code patching should be used only when the processor cannot be configured to map all addresses through the TLB.

Kernels that use memory-mapping to cache files must be modified to map the file read-only. Kernel procedures that write to the memory-mapped file must be modified as above to first enable writes to memory. Digital Unix does not use memory-mapping in the kernel. User memory-mapped files require no changes to the kernel because these files are not mapped into the kernel address space and hence the kernel cannot corrupt them.

This scheme protects memory solely from kernel crashes. Naturally, a faulty user program can still corrupt any file to which it has write access.

2.2 Warm Reboot

The second step in enabling the file cache to survive a crash is to do a *warm reboot*. When the system is rebooted, it must read the file cache contents that were present in physical memory before the crash and update the file system with this data. Because system crashes are infrequent, our first priority in designing the warm reboot is ease of implementation, rather than reboot speed.

Two issues arise when doing a warm reboot: 1) what additional data the system maintains during normal operation, and 2) when in the reboot process the system restores the file cache contents.

Maintaining additional data during normal operation makes it easier to find, identify, and restore the file cache contents in memory during the warm reboot. Without additional data, the system would need to analyze a series of data structures, such as internal file cache lists and page tables, and all these intermediate data structures would need to be protected. Instead of understanding and protecting all intermediate data structures, we keep and protect a separate area of memory, which we call the *registry*, that contains all information needed to find, identify, and restore files in memory. For each buffer in the file cache, the registry contains the physical memory address, file id (device number and inode number), file offset, and size. Registry information changes relatively infrequently during normal operation, so the overhead of maintaining it is low. It is also quite small; only 40 bytes of information are needed for each 8 KB file cache page.

The second issue is when to restore the dirty file cache contents during reboot. To minimize the changes needed to the VM and file system initialization procedures, we perform the warm reboot in two steps. Before the VM and file system are initialized, we dump all of physical memory to the swap partition. This saves the contents of the file cache and registry from before the crash and is similar to performing a crash dump as the system is going down. While a standard crash dump often fails, however, this dump is performed on a healthy, booting system and will always work. We also restore the metadata to disk during this step, using the disk address stored in the registry, so that the file system is intact before being checked for consistency by fsck.

After the system is completely booted, a user-level process analyzes the memory dump and restores the UBC using normal system calls such as open and write.

2.3 Effects on File System Design

The presence of a reliable file cache changes some aspects of the file system. First, reliability-induced writes to disk are no longer needed, because files in memory are as permanent and safe as files on disk. Digital Unix includes tunable parameters to turn off reliability writes for the UBC. We disable buffer cache writes as in [Ohta90] by turning most bwrite and bawrite calls to bdwrite; we modify sync and fsync calls to return immediately[1]; and we modify the panic procedure to avoid writing dirty data back to disk before a crash. With these changes, writes to disk occur only when the UBC or buffer cache overflows, so dirty blocks can remain in memory indefinitely. Less extreme approaches such as writing to disk during idle periods may improve system responsiveness, and we plan to experiment with this in the future. The focus of this paper is reliability, hence we take the extreme approach of delaying writes to disk as long as possible.

Second, metadata updates in the buffer cache must be as carefully ordered as those to disk, because buffer cache data is now permanent.

Third, memory's high throughput makes it feasible to guarantee atomicity when updating critical metadata information. When the system wants to write to metadata in the buffer cache, it first copies the contents to a shadow page and changes the registry entry to point to the shadow. When it finishes writing, it atomically points the registry entry back to the original buffer.

3 Reliability

The key to Rio is reliability: can files in memory truly be made as safe from system crashes as files on disk? To answer this, we measure how often crashes corrupt data on disk and in memory. For each run, we inject faults to crash a running system, reboot, then examine the file data and measure the amount of corruption.

3.1 Fault Models

This section describes the types of faults we inject. Our primary goal in designing these faults is to generate a *wide variety* of system crashes. Our models are derived from studies of commercial operating systems [Sullivan91b, Lee93] and from prior models used in fault-injection studies [Barton90, Kao93, Kanawati95]. The faults we inject range from low-level hardware faults such as flipping bits in memory to high-level software faults such as memory allocation errors. We classify injected faults into three categories: bit flips, low-level software faults, and high-level software faults. Unless otherwise stated, we inject 20 faults for each run to increase the chances that a fault will be triggered. Most crashes occurred within 15 seconds after the fault was injected. If a fault does not crash the machine after ten minutes, we discard the run and reboot the system; this happens about half the time. Note that faults that leave the system running will propagate data to disk and hence not change the relative reliability between memory and disk.

The first category of faults flips random bits in the kernel's address space [Barton90, Kanawati95]. We target three areas of the kernel's address space: the *kernel text*, *heap*, and *stack*. These faults are easy to inject, and they cause a

1. We do provide a way for a system administrator to easily enable and disable reliability disk writes for machine maintenance or extended power outages.

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. These faults are intended to approximate the assembly-level manifestation of real C-level programming errors [Kao93]. We corrupt assignment statements by changing the *source* or *destination* register. We corrupt conditional constructs by deleting *branches*. We also delete *random instructions* (both branch and non-branch).

The last and most extensive category of faults imitate specific programming errors in the kernel [Sullivan91b]. These are more targeted at specific programming errors than the previous fault category. We inject an *initialization* fault by deleting instructions responsible for initializing a variable at the start of a procedure [Kao93, Lee93]. We inject *pointer* corruption by 1) finding a register that is used as a base register of a load or store and 2) deleting the most recent instruction before the load/store that modifies that register [Sullivan91b, Lee93]. We do not corrupt the stack pointer register, as this is used to access local variables instead of as a pointer variable. We inject an *allocation management* fault by modifying the kernel malloc procedure to occasionally start a thread that sleeps 0-256 ms, then prematurely frees the newly allocated block of memory. Malloc is set to inject this error every 1000-4000 times it is called; this occurs approximately every 15 seconds. We inject a *copy overrun* fault by modifying the kernel's bcopy procedure to occasionally increase the number of bytes it copies. The length of the overrun was distributed as follows: 50% corrupt one byte; 44% corrupt 2-1024 bytes; 6% corrupt 2-4 KB. This distribution was chosen by starting with the data gathered in [Sullivan91b] and modifying it somewhat according to our specific platform and experience. bcopy is set to inject this error every 1000-4000 times it is called; this occurs approximately every 15 seconds. 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.

Fault injection cannot mimic the exact behavior of all real-world operating system crashes. However, the wide variety of faults we inject (13 types), the random nature of the faults, the large number of ways the system crashed in our experiments (e.g. 74 unique error messages, including 59 different kernel consistency error messages), and the sheer number of crashes we performed (1950) give us confidence that our experiments cover a wide range of real-world crashes.

3.2 Detecting Corruption

File corruption can occur in two ways. In *direct* corruption, a series of events eventually causes a procedure (usually a non-I/O procedure) to accidentally write to file data. Memory is more vulnerable than disks to direct corruption, because it is nearly impossible for a non-disk procedure to directly overwrite the disk drive. However, direct memory corruption can affect disk data if the system stays up long enough to propagate the bad memory data to disk. In *indirect* corruption, a series of events eventually causes a procedure to call an I/O procedure with the wrong parameters. The I/O procedure obediently carries out the request and corrupts the file cache. Disks and memory are both vulnerable to indirect corruption.

We are interested primarily in protecting memory from direct corruption, because this is the weak point of random-access memories. Note that the mechanisms described in Section 2.1 protect only against direct corruption; indirect corruption will circumvent our protection mechanism.

We use two strategies to detect file corruption: checksums detect direct corruption, and a synthetic workload called *memTest* detects direct and indirect corruption.

The first method to detect corruption maintains a checksum of each memory block in the file cache [Baker92b]. We update the checksum in all procedures that write the file cache; unintentional changes to file cache buffers result in an inconsistent checksum. We identify blocks that were being modified while the crash occurred by marking a block as *changing* before writing to the block; these blocks cannot be identified as corrupt or intact by the checksum mechanism. Files mapped into a user's address space for writing are also marked changing as long as they are in memory, though this does not occur with the workloads we use.

Catching indirect corruption requires an application-level check, so we create a special workload called *memTest* whose actions and data are repeatable and can be checked after a system crash. Checksums and *memTest* complement each other. The checksum mechanism provides a means for detecting direct corruption for any arbitrary workload; *memTest* provides a higher-level check on certain data by knowing its correct value at every instant.

memTest generates a repeatable stream of file and directory creations, deletions, reads, and writes, reaching a maximum file set size of 100 MB. Actions and data in *memTest* are controlled by a pseudo-random number generator. After each step, *memTest* records its progress in a status file across the network. 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 contents with the file cache image in memory (restored during the warm reboot).

As a final check for corruption, we keep two copies of all files that are not modified by our workload and check that the two copies are equal. These files were not corrupted in our tests.

In addition to *memTest*, we run four copies of the Andrew benchmark [Howard88], a general-purpose file-system workload. Andrew creates and copies a source hierarchy; examines the hierarchy using find, ls, du, grep, and wc; and compiles the source hierarchy. As with all file activity besides *memTest*, the correctness of Andrew's files is checked only with the checksum mechanism.

3.3 Reliability Results

Table 1 presents reliability measurements for three systems: a disk-based (write-through) file cache, Rio without protection (just warm reboot), and Rio with protection. We conducted 50 tests for each fault category for each of the three systems (disk, Rio without protection, Rio with protection); this represents 6 machine-months of testing.

Rio's goal is to match the reliability of disk, so we start by measuring the reliability of a write-through file cache. We use the functionality and setup of the default Digital Unix kernel. That is, we do not use warm reboot or protection, nor do we turn off reliability-induced disk writes. To achieve write-through semantics, *memTest* calls fsync after every write—without this, many runs would lose asynchronously written data that had not yet made it to disk. Only *memTest* is used to detect corruption on disk, because data on disk is not subject to direct corruption, so the checksum is guaranteed to be correct.

Table 1 shows that corruption is quite infrequent, which agrees with our intuition that disks are usually safe from operating system crashes. Of 650 crashes, only seven crashes (1.1%) corrupted any file data, and each of those runs corrupted only a few (1-4) files/directories.[2]

Fault Type	Disk-Based	Rio without Protection	Rio with Protection
kernel text	2	1	
kernel heap			
kernel stack		1	1
destination reg.			
source reg.	2		
delete branch	1	1	1
delete random inst.	1		
initialization			1
pointer		1	
allocation			
copy overrun		4	
off-by-one	1	2	1
synchronization			
Total	**7 of 650 (1.1%)**	**10 of 650 (1.5%)**	**4 of 650 (0.6%)**

Table 1: Comparing Disk and Memory Reliability. This table shows how often each type of error corrupted data for three systems. We conducted 50 tests for each fault type for each of three systems. The disk-based system uses fsync after every write, achieving write-through reliability. The two Rio systems test memory reliability by turning off reliability writes to disk and using warm reboot to recover the in-memory data after a crash. Blank entries had no corruptions. Even without protection, Rio's reliability is nearly as high as a write-through system. With protection, Rio achieves even higher reliability than a write-through system.

The middle section of Table 1 shows the reliability of the Rio file cache *without* the protection mechanisms described in Section 2.1. We turn off all reliability-related disk writes (Section 2.3) and use warm reboot (Section 2.2) to recover the files in memory after a crash. These runs thus measure how often files in memory are corrupted during an operating system crash if no provisions are made to protect them. Out of 650 crashes, ten crashes (1.5%) corrupted any file data. As with the disk tests, each corruption affected a small number of files/directories, usually just a small portion of one file. *memTest* detected all ten corruptions, and checksums detected five of the ten. Interestingly, the corrupted data in the other five corruptions resided on disk rather than the file cache. This implies that the system remained running long enough to propagate the corruption to disk. Copy overruns have a relatively high chance of corrupting the file cache because the injected fault directly overwrites a portion of memory, and this portion of memory has a reasonable chance of overlapping with a file cache buffer.

While slightly less reliable than disks, Rio without protection is *much* more reliable than we had expected and is probably reliable enough for most systems. To illustrate, consider a system that crashes once every two months (a somewhat pessimistic estimate for production-quality operating systems). If these crashes were the sole cause of data corruption, the MTTF (mean time to failure) of a disk-based system would be 15 years, and the MTTF of Rio without protection would be 11 years. That is, if your system crashes once every two months, you can expect to lose a few file blocks about once a decade with Rio, even with no protection! Thus, *warm reboot enables a file cache to be nearly as reliable as disk, even with no protection*.

These results stand in sharp contrast to the general feeling among computer scientists (including the authors) that operating system crashes often corrupt files in memory. We believe the results are due to the multitude of consistency checks present in a production operating system, which stop the system very soon after an injected fault is encountered and thereby limit the amount of damage. In addition to the standard sanity checks written by programmers, the virtual memory system implicitly checks each load/store address to make sure it is a valid address. Particularly on a 64-bit machine, most errors are first detected by issuing an illegal address [Kao93, Lee93].

Thus, even without protection, Rio stores files nearly as reliably as a write-through file system. However, some applications will require even higher levels of safety. The rightmost section of Table 1 shows the reliability of the Rio file cache with protection turned on. Out of 650 crashes, we

2. We plan to trace how faults propagate to corrupt files and crash the system instead of treating the system as a black box. This is extremely challenging, however, and is beyond the scope of this paper [Kao93].

measured only four corruptions (0.6%). Thus Rio with protection provides reliability even higher than a write-through file cache while issuing no reliability-induced writes to disk! We recorded eight crashes where the Rio protection mechanism was invoked to prevent an illegal write to the file cache (six for copy overrun and two for initialization); these indicate cases where the file cache would have been corrupted had the protection mechanism been off. Rio's protection mechanism provides higher reliability than a write-through file cache because it halts the system when it detects an attempted illegal access to the file cache. Write-through file caches, in contrast, may continue to run and later propagate the corrupted memory data to disk.

4 Performance

The main benefit of Rio discussed 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 2 compares the performance of Rio with different Unix file systems, each providing different guarantees on when data is made permanent.

UFS is the default Digital 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 [Ganger94].

UFS's poor performance is due in large part to its synchronous metadata updates. To eliminate this bottleneck, we enhanced UFS to delay all data and metadata until the next time update runs; this is the optimal "no-order" system in [Ganger94]. This improves performance significantly over the default UFS; however, the optimization risks losing 30 seconds of both data and metadata.

We measure the behavior of two file systems that write data synchronously to disk. UFS with write-through-on-close makes data permanent upon each file close by calling fsync. UFS with write-through-on-write makes data permanent upon each file write by mounting all file systems with the "sync" option and also calling fsync after each close. Note that only UFS with write-through-on-write achieves the same reliability as Rio.

The Memory File System, which is completely memory-resident and does no disk I/O, is shown to illustrate optimal performance [McKusick90]. AdvFS is a journalling file system that reduces the penalty of metadata updates by writing metadata sequentially to a log.

We run three workloads, cp+rm, Sdet, and Andrew. cp+rm recursively copies then recursively removes the Digital Unix source tree (40 MB). Sdet is one of SPEC's SDM benchmarks and models a multi-user software development environment [SPE91]. Andrew also models software development but is dominated by CPU-intensive compilation [Howard88]. All results represent an average of at least 5 runs.

The last two rows of Table 2 show that Rio's protection mechanism adds almost no performance penalty, even on very I/O intensive workloads such as cp+rm. Since Section 3.3 shows that Rio's protection mechanism enables memory to be even safer than a write-through file system, we recommend that protection be turned on.

Table 2 shows that Rio performs as fast as a memory file system and significantly faster than all other file systems. As expected, Rio's performance improvement is largest over systems that provide similar reliability guarantees—Rio performs 4-22 times as fast as UFS write-through-on-write and write-through-on-close.

	Data Permanent	**cp+rm (seconds)**	**Sdet (5 scripts) (seconds)**	**Andrew (seconds)**
Memory File System	never	21 (15+6)	43	13
UFS with delayed data and metadata	after 0-30 seconds, asynchronous	81 (76+5)	47	13
AdvFS (log metadata updates)	after 0-30 seconds, asynchronous	125 (110+15)	132	16
UFS	data after 64 KB, asynchronous metadata synchronous	332 (245+87)	401	23
UFS with write-through after each close	after close, synchronous	394 (274+120)	699	49
UFS with write-through after each write	after write, synchronous	539 (419+120)	910	178
Rio without protection	**after write, synchronous**	**24 (18+6)**	**42**	**12**
Rio with protection	**after write, synchronous**	**25 (18+7)**	**42**	**13**

Table 2: Performance Comparison. This table compares the running time of Rio with different Unix file systems, each providing different guarantees on when data is made permanent. cp+rm recursively copies then recursively removes the Digital Unix source tree (40 MB); Sdet is one of SPEC's SDM benchmarks and models a multi-user software development environment; Andrew also models software development but is dominated by CPU-intensive compilation. Rio achieves performance comparable to a memory-resident file system while providing the reliability of a write-through file system. Rio's protection mechanism adds essentially no overhead, yet enables Rio to surpass the reliability of a write-through file system. Rio is 2-14 times as fast as the default Unix file system. Delaying metadata writes by 30 seconds enables UFS to match Rio's speed on some workloads, but Rio is still 3 times as fast on cp+rm. Rio is 4-22 times as fast as systems that guarantee data permanence after each file write or close.

Other file systems can shrink the gap in performance by sacrificing reliability. Rio is 2-14 times as fast as the standard UFS file system, yet Rio provides synchronous data updates. Rio is 1-3 times as fast as UFS with delayed data and metadata. Yet while the optimized UFS system risks losing 30 seconds of data and metadata on a crash, Rio loses no data or metadata.

5 Architectural Support for Reliable File Caches

We have shown that memory can safely store permanent data in the presence of operating system crashes. This has several implications for computer architects. First, designers of memory-management hardware should continue to provide the ability to force all accesses through the TLB, as is done in most microprocessors today. Without this ability, the processor can bypass the TLB at any time, and code patching must be used to protect the file cache from corruption.

Second, since memory contains long-term data, the system should treat memory like a peripheral that can be removed from the rest of the system. If the system board fails, it should be possible to move the memory board to a different system without losing power or data [Moran90, Baker92a]. Similarly, the system should be able to be reset and rebooted without erasing the contents of memory or CPU caches containing memory data. DEC Alphas allow a reset and boot without erasing memory or the CPU caches [DEC94]; the PCs we have tested do not.

Storing permanent data both on disk and in memory makes data more vulnerable to hardware failures than simply storing data on disk. Being able to remove the memory system without losing data can reduce but not eliminate the increased vulnerability. Because software crashes are the dominant cause of failure today [Gray90], we do not consider the increased vulnerability to hardware failures a serious limitation of Rio. However, if memory or CPU failures becomes the most common cause of system failure, extra redundancy may need to be added to compensate for the larger number of components holding permanent data.

6 Related Work

Several researchers have proposed ways to protect memory from software failures [Copeland89], though to our knowledge none have evaluated how effectively memory withstood these failures.

Many commercial I/O devices contain memory. This memory is assumed to be reliable because of the I/O interface used to access it. These devices include solid-state disks, non-volatile disk caches, and write-buffers such as Prestoserve [Moran90]. While these can improve performance over disks, their performance is limited by the low bandwidth and high overhead of the I/O bus and device interface. Being able to use ordinary main memory to store files reliably would be much better: systems already have a relatively large amount of main memory and can access it very quickly. Further, main memory is random-access, unlike special-purpose devices.

Phoenix is the only file system we are aware of that attempts to make all permanent files reliable while in main memory [Gait90]. 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. Rio 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 Rio keeps only one copy.

Harp protects a log of recent modifications by *replicating* it in volatile, battery-backed memory across several server nodes [Liskov91]. Harp designers considered using warm reboot to protect against software bugs that crash both nodes. Unfortunately, the MicroVax's used to run Harp overwrote memory during a reboot, making warm reboot impossible [Baker94].

The Recovery Box stores system state used in recovery in a region of memory [Baker92b]. Recovery box memory is preserved across crashes and used during the reboot of file servers. Similar to results found in Rio, Baker and Sullivan expect few crashes to corrupt the contents of the Recovery Box and so rely primarily on checksums to verify that data is intact. They lower the chance of corruption by 1) writing to the Recovery Box through a careful interface that checks for errors and 2) storing the Recovery Box within the kernel text segment, where it is less likely to be corrupted by random pointer errors. If checksums indicate that the Recovery Box is corrupted despite these precautions, the system discards the data and performs a full recovery.

Rio differs from Recovery Box in the degree to which the system depends on memory being intact, as well as the use and size of the data. Data in the Recovery Box is seen strictly as a hint; if the data is wrong, the system can recover all information. In contrast, Rio sees memory as reliable enough to store the sole copy of data. This allows Rio to store a wider range and larger amount of data. In particular, Rio stores file data and can thus improve file system performance under normal operation.

Rio's protection mechanism is similar to the scheme in [Baker94] and the "expose page" scheme in [Sullivan91a], but Rio additionally protects against physical addresses that would otherwise bypass the TLB. Sullivan and Stonebraker measure the overhead of "expose page" to be 7% on a debit/credit benchmark. The overhead of Rio's protection mechanism, which is negligible, is lower for two reasons. First, Rio is implemented in the kernel and needs no system call to change a page's protection. Second, data in the file cache is written in larger blocks than in debit/credit; this amortizes the cost of changing protection over more bytes.

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 [Banatre91]. In contrast, Rio makes all files in memory reliable without special-purpose hardware or replication.

A variety of general-purpose hardware and software mechanisms may be used to help protect memory from software faults. Papers by Johnson and Wahbe suggest various hardware mechanisms to trap updates to certain memory locations [Johnson82, Wahbe92]. Hive uses the Flash firewall to protect memory against wild writes by other processors in a multiprocessor [Chapin95]. 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 [Kessler90, Wahbe92] and fault isolation between software modules [Wahbe93].

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 non-volatile, flash RAM [Wu94]. 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 [Abbott94]. These schemes complement Rio, which protects memory from operating system crashes.

7 Conclusions

We have made a case for reliable file caches: main memory that can survive operating system crashes and be as safe and permanent as disk. Our reliability experiments show that, even without protection, warm reboot enables memory to achieve reliability close to that of a write-through file system. Adding protection makes memory even safer than a write-through file system while adding essentially no overhead. Eliminating all reliability-induced writes to disk enables Rio to run 4-22 times as fast as systems that give comparable reliability guarantees, 2-14 times as fast as a standard, delayed-write file system, and 1-3 times as fast as an optimized system that risks losing 30 seconds of data and metadata.

Reliable file caches have striking implications for future system designers:

- Write-backs to disk are no longer needed except when the file cache fills up. This changes the assumptions about the dominance of write traffic underlying some file system research such as LFS [Rosenblum92, Baker91]. Delaying writes to disk until the file cache fills up enables the largest possible number of files to die in memory and enables remaining files to be written out more efficiently. Thus Rio improves performance over existing delayed-write systems.
- All writes are synchronously and instantly permanent, improving reliability over current systems. Fast, synchronous writes improve performance by an order of magnitude for applications that require synchronous semantics. Applications that do not require synchronous semantics for reliability may become simpler because synchronous events are easier to deal with than asynchronous events. For example, the order that synchronous writes become permanent matches the order in which writes are issued.

To further test and prove our ideas, we have installed a departmental file server using the Rio file cache with protection and with reliability-induced writes to disk turned off. Among other things, this file server stores our kernel source tree, this paper, and the authors' mail. We plan to redo this study on a different operating system and to perform a similar fault-injection experiment on a database system. We believe these will show that our conclusions about memory's resistance to software crashes apply to other large software systems.

The Rio file cache provides a new storage component for system design: one that is as fast, large, common, and cheap as main memory, yet as reliable and stable as disk. We look forward to seeing how system designers use this new storage component.

8 Acknowledgments

We would like to thank Mary Baker, Greg Ganger, Peter Honeyman, Trevor Mudge, Margo Seltzer, and the anonymous reviewers for helping improve the quality of this work.

9 References

[Abbott94] 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. In *Proceedings of the 1994 International Symposium on Fault-Tolerant Computing*, pages 414–423, 1994.

[APC96] The Power Protection Handbook. Technical report, American Power Conversion, 1996.

[Baker91] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a Distributed File System. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 198–212, October 1991.

[Baker92a] Mary Baker, Satoshi Asami, Etienne Deprit, John Ousterhout, and Margo Seltzer. Non-Volatile Memory for Fast Reliable File Systems. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pages 10–22, October 1992.

[Baker92b] Mary Baker and Mark Sullivan. The Recovery Box: Using Fast Recovery to Provide High Availability in the UNIX Environment. In *Proceedings USENIX Summer Conference*, June 1992.

[Baker94] Mary Louise Gray Baker. *Fast Crash Recovery in Distributed File Systems*. PhD thesis, University of California at Berkeley, January 1994.

[Banatre91] Michel Banatre, Gilles Muller, Bruno Rochat, and Patrick Sanchez. Design decisions for the FTM: a general purpose fault tolerant machine. In *Proceedings of the 1991 International Symposium on Fault-Tolerant Computing*, pages 71–78, June 1991.

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

[Chapin95] John Chapin, Mendel Rosenblum, Scott Devine, Tirthankar Lahiri, Dan Teodosiu, and Anoop Gupta. Hive: Fault Containment for Shared-Memory Multiprocessors. In *Proceedings of the 1995 Symposium on Operating Systems Principles*, December 1995.

[Chen96] Peter M. Chen, Wee Teck Ng, Gurushankar Rajamani, and Christopher M. Aycock. The Rio File Cache: Surviving Operating System Crashes. Technical Report CSE-TR-286-96, University of Michigan, March 1996.

[Copeland89] George Copeland, Tom Keller, Ravi Krishnamurthy, and Marc Smith. The Case for Safe RAM. In *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, pages 327–335, August 1989.

[DEC94] DEC 3000 300/400/500/600/700/800/900 AXP Models System Programmer's Manual. Technical report, Digital Equipment Corporation, July 1994.

[DeWitt84] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. Wood. Implementation Techniques for Main Memory Database Systems. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, pages 1–8, June 1984.

[Gait90] Jason Gait. Phoenix: A Safe In-Memory File System. *Communications of the ACM*, 33(1):81–86, January 1990.

[Ganger94] Gregory R. Ganger and Yale N. Patt. Metadata Update Performance in File Systems. *1994 Operating Systems Design and Implementation (OSDI)*, November 1994.

[Gray90] Jim Gray. A Census of Tandem System Availability between 1985 and 1990. *IEEE Transactions on Reliability*, 39(4), October 1990.

[Hagmann87] Robert B. Hagmann. Reimplementing the Cedar File System Using Logging and Group Commit. In *Proceedings of the 1987 Symposium on Operating Systems Principles*, pages 155–162, November 1987.

[Hartman93] John H. Hartman and John K. Ousterhout. Letter to the Editor. *Operating Systems Review*, 27(1):7–9, January 1993.

[Hennessy90] John. L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach, 2nd Edition*. Morgan Kaufmann Publishers, Inc., 1990. page 493.

[Howard88] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[Johnson82] Mark Scott Johnson. Some Requirements for Architectural Support of Software Debugging. In *Proceedings of the 1982 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 140–148, April 1982.

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

[Kane92] Gerry Kane and Joe Heinrich. *MIPS RISC Architecture*. Prentice Hall, 1992.

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

[Kessler90] Peter B. Kessler. Fast breakpoints: Design and implementation. In *Proceedings of the 1990 Conference on Programming Language Design and Implementation (PLDI)*, pages 78–84, June 1990.

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

[Leffler89] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD Unix Operating System*. Addison-Wesley Publishing Company, 1989.

[Liskov91] Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, Liuba Shrira, and Michael Williams. Replication in the Harp File System. In *Proceedings of the 1991 Symposium on Operating System Principles*, pages 226–238, October 1991.

[McKusick90] Marshall Kirk McKusick, Michael J. Karels, and Keith Bostic. A Pageable Memory Based Filesystem. In *Proceedings USENIX Summer Conference*, June 1990.

[Moran90] J. Moran, Russel Sandberg, D. Coleman, J. Kepecs, and Bob Lyon. Breaking Through the NFS Performance Barrier. In *Proceedings of EUUG Spring 1990*, April 1990.

[Ohta90] Masataka Ohta and Hiroshi Tezuka. A Fast /tmp File System by Delay Mount Option. In *Proceedings USENIX Summer Conference*, pages 145–150, June 1990.

[Ousterhout85] John K. Ousterhout, Herve Da Costa, et al. A Trace-Driven Analysis of the UNIX 4.2 BSD File System. In *Proceedings of the 1985 Symposium on Operating System Principles*, pages 15–24, December 1985.

[Rosenblum92] Mendel Rosenblum and John K. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.

[Silberschatz94] Abraham Silberschatz and Peter B. Galvin. *Operating System Concepts*. Addison-Wesley, 1994. page 200.

[Sites92] Richard L. Sites, editor. *Alpha Architecture Reference Manual*. Digital Press, 1992.

[SPE91] SPEC SDM Release 1.0 Technical Fact Sheet. Technical report, Franson and Haggerty Associates, 1991.

[Sullivan91a] M. Sullivan and M. Stonebraker. Using write protected data structures to improve software fault tolerance in highly available database management systems. In *Proceedings of the 1991 International Conference on Very Large Data Bases (VLDB)*, pages 171–180, September 1991.

[Sullivan91b] 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.

[Tanenbaum95] Andrew S. Tanenbaum. *Distributed Operating Systems*. Prentice-Hall, 1995. page 146.

[Wahbe92] Robert Wahbe. Efficient Data Breakpoints. In *Proceedings of the 1992 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 1992.

[Wahbe93] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient Software-Based Fault Isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 203–216, December 1993.

[Wu94] Michael Wu and Willy Zwaenepoel. eNVy: A Non-Volatile, Main Memory Storage System. In *Proceedings of the 1994 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 1994.