The Rio File Cache: Surviving Operating System Crashes

Peter M. Chen, Wee Teck Ng, Gurushankar Rajamani, Christopher M. Aycock
Computer Science and Engineering Division
Department of Electrical Engineering and Computer Science
University of Michigan
{pmchen,weeteck,gurur,caycock}@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 while performing 20 times faster. Rio makes all writes immediately permanent, yet performs faster than systems that lose 30 seconds of data on a crash: 35% faster than a standard delayed-write file system and 8% faster than a system that delays both data and metadata. For applications that demand even higher levels of reliability, Rio's optional protection mechanism makes memory even safer than a write-through file system while while lowering performance 20% compared to a pure write-back system.

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 [Tanenbaum95, page 146].

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 [DEC95, Tanenbaum95 page 146, Silberschatz94 page 200]. Consequently, file systems periodically write data to disk, and transaction processing applications view transactions as committed only when data is written to disk. The focus of this paper is enabling 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 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, Chutani92, DeWitt84], disk throughput is still far slower than memory throughput.
- Unix file systems mitigate the performance lost in reliability-induced disk writes by waiting 30 seconds before writing data, but this ensures the loss of data written within 30 seconds of a crash [Ousterhout85]. In addition, 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. A longer delay decreases disk traffic due to writes but risks losing even 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 is an option only for applications where reliability is not an issue, such as compiler-generated temporary files.

Existing choices for reliable memory are attached via an I/O or backplane bus rather than the memory bus. These special-purpose devices include solid-state disks, non-volatile disk controllers, 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 ordi-

nary 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.

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 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 while performing 20 times faster. Rio makes all writes immediately permanent, yet performs faster than systems that lose 30 seconds of data on a crash: 35% faster than a standard delayed-write file system and 8% faster than a system that delays both data and metadata. For applications that demand even higher levels of reliability, Rio's optional protection mechanism makes memory even safer than a write-through file system while while lowering performance 20% compared to a pure write-back system.

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.[1] 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 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 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]. 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 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

1. We will see in Section 3.3 that even without protection, most crashes do not corrupt files in memory. Hence we recommend that protection be turned off for most systems. We describe Rio's optional protection mechanism first because most people (including the authors) assume it is needed.

Alpha processor with the two most significant bits equal to `10` bypass the TLB. Rio uses two different methods to protect against these physical addresses.

Our current method, called *code patching*, is to modify 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. The idea of inserting code before every store instruction sounds prohibitively slow, but several optimizations make the actual overhead only 20% (Section 4).

- The checking code is very efficient: 6 instructions for a virtual address (the normal case), 28 instructions for a physical address. We gain efficiency over more general tools such as ATOM [Srivastava94] by inlining the check for virtual addresses and by increasing each procedure's stack rather than creating a temporary stack frame for each check.
- Modifications to the stack pointer occur much less frequently than stores to memory that use the stack pointer. In addition, the stack pointer is almost always modified in small increments, and these small increments cannot change a virtual address to a physical address. We can hence replace the checks on local, stack variables with a few checks on the stack pointer [Wahbe93].
- We replace individual checks in commonly used loops with a few higher-level checks. For example, procedures such as bcopy modify sequential blocks of data; these blocks can be checked once rather than checking every individual store.
- Further optimizations are possible, such as recognizing loop invariants and eliminating redundant checks within a basic block. Trends toward moving functionality out of the kernel and relatively faster CPUs will further lower the overhead of code patching.

A second method to protect against physical addresses is specific to the Alpha processor. The Alpha CPU can be set to trap to a special handler if a physical addresses is issued. This handler can then validate the address and issue or deny the request. This method gains efficiency over code patching by avoiding the check on virtual addresses. However, the handler must be invoked on both reads and writes to physical addresses. Which method will be faster depends on the mix of physical and virtual addresses and on the mix of reads and writes. We believe the second method could lower the overhead to a mere 1-2%.

Kernels that use memory-mapping to cache files must be modified to map the file read-only. Procedures that write to the memory-mapped file must be modified as above to first enable writes to memory. The Digital Unix kernel does not use memory-mapping in the kernel. User memory-mapped files, which are supported by Digital Unix, require no changes to the kernel, because we protect memory solely from kernel crashes; users are responsible for their own errors.

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 initialization procedures are run, we dump all of physical memory to the swap partition. This saves the contents of the file cache and registry from before the crash[2]. 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 fsck runs. 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 reliable 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[3]; 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 overflow, so dirty blocks can remain in memory indefinitely. One could also take a less extreme approach, such as writing to disk during idle periods.

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. 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 the faults we inject 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.[4]

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 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 [Sullivan91]. 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, Lee93a]. 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 [Sullivan91, Lee93a]. 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

2. This 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.

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

4. These long-latency faults will propagate data to disk and hence not change the relative reliability between memory and disk.

follows: 50% corrupt one byte; 44% corrupt 2-1024 bytes; 6% corrupt 2-4 KB. This distribution was chosen by starting with the data gathered in [Sullivan91] and modifying it somewhat according to our specific platform and experience. bcopy is set to inject this error every 1000-4000 times it is called; this occurs approximately every 15 seconds. We inject *off-by-one* errors by changing conditions such as > to >=, and < 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 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 on 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, Ousterhout90], 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.

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 data written within 30 seconds of the crash. Our only tool for detecting corruption on disk is *memTest*, because our checksum method cannot detect disk corruption[5]. Table 1 shows that corruption is quite rare, which agrees with our intuition that disks are usually safe from operating system crashes. Of 650 crashes, only seven (1.1%) corrupted any file data, and each of those runs corrupted only a few (1-4) files/directories.[6]

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. We experienced ten corruptions out of 650 crashes (1.5%). 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

Fault Type	**Disk-Based**		**Rio without Protection**		**Rio with Protection**	
	# crashes	**# corruptions**	**# crashes**	**# corruptions**	**# crashes**	**# corruptions**
kernel text	50	2	50	1	50	
kernel heap	50		50		50	
kernel stack	50		50	1	50	1
destination reg.	50		50		50	
source reg.	50	2	50		50	1
delete branch	50	1	50	1	50	
delete random inst.	50	1	50		50	
initialization	50		50		50	
pointer	50		50	1	50	
allocation	50		50		50	
copy overrun	50		50	4	50	
off-by-one	50	1	50	2	50	1
synchronization	50		50		50	
Total	**650**	**7 (1.1%)**	**650**	**10 (1.5%)**	**650**	**3 (0.5%)**

Table 1: Comparing Disk and Memory Reliability. This table shows how often each type of error corrupted data for 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. We calculate the normalized corruption rate by first calculating the % corruption for each category, then averaging across all categories. This weights each fault category equally, independent of the number of runs in the category. Even without protection, Rio's reliability is nearly the same as a write-through system, and this is the system we recommend. With protection, Rio achieves the same or higher reliability as a write-through system.

5. Checksumming the disk data would be done immediately before writing to disk. Data on disk is not subject to direct corruption, so the checksum is guaranteed to be correct.

6. 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].

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 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! Even though the faults we inject probably do not perfectly represent real-world crashes, the qualitative conclusion is clear: *warm reboot enables a file cache to be about as reliable as disk, even with no protection*.

These results stand in sharp contrast to the general feeling among computer scientists 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 a fault is injected 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, Lee93a].

Thus, even without protection, Rio stores files about as reliably as a write-through file system, and this is the configuration we recommend for most systems. 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 measured only three corruptions (0.5%). Thus Rio with protection provides reliability even higher than a write-through file cache while issuing no reliability-induced writes to disk! We recorded six crashes where the Rio protection mechanism was invoked to prevent an illegal write to the file cache (three for copy overruns, three for pointer); these indicate cases where the file cache would have been corrupted if the protection mechanism had been off. We believe that 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. Write-through file caches, in contrast, may continue to run and thus propagate corrupted memory data to disk.

4 Performance

We consider the main benefit of Rio to be reliability: all writes to the file cache are immediately as permanent and safe as files on disk. Rio also improves performance by eliminating all reliability-induced writes to disk. Table 2 compares the performance (on the Andrew file system benchmark) of Rio with several variations on the Unix file system, each providing different guarantees on when data is made permanent [Howard88, Ousterhout90]. Rio without protection performs 3-20 times faster than systems with comparable reliability guarantees (write-through on write, write-through on close). Rio also performs 35% faster than the standard Unix file system. Much of this advantage is due to UFS's synchronous (write-

	Data Permanent	**Running Time**
Memory File System	never	12.3 seconds
UFS with delayed metadata updates	after 30 seconds	15.4 seconds
UFS	after 30 seconds	21.8 seconds
UFS with write-through after each close	after close	49.0 seconds
UFS with write-through after each write	after write	305.4 seconds
Rio without protection	**after write**	**14.2 seconds**
Rio with protection	after write	17.0 seconds

Table 2: Performance Comparison. This table compares the performance (on the Andrew file system benchmark) of Rio with several variations on the Unix file system, each providing different guarantees on when data is made permanent. Rio without protection makes data permanent after each write, yet has performance 35% better than the standard Unix file system and 8% better than a system where metadata updates are delayed by 30 seconds before being written to disk [Ganger94]. Adding protection slows Rio performance down by 20%, but some applications may require the extra margin of safety this provides. Other file systems that guarantee data permanence after each file write or close perform 3-20 times slower than Rio. MFS, which is completely memory-resident and does no disk I/O, is shown for comparison.

through) metadata updates, so we also measure UFS after delaying all metadata updates by 30 seconds (the optimal "no-order" system in [Ganger94]). Rio still performs 8% faster than this system due to the sync every 30 seconds. Yet while these systems lose 30 seconds of recently written data on a crash, Rio loses none. MFS, which is completely memory-resident and does no disk I/O, is shown to illustrate optimal performance [McKusick90]. Though suitable only for temporary files, MFS achieves superior performance because of its simplicity—its code is 1/10 the size of UFS's! Adding protection slows Rio performance down by 20%, but some applications may require the extra margin of safety this provides.

5 Architectural Support for Reliable File Caches

The conclusion that memory can be considered a safe place for permanent data has several implications for architects. A small amount of hardware support at the memory level would make protection easier. An ideal memory controller would enable file system procedures to prevent writes to certain physical pages [Banatre91]. One simple way to implement this is for the controller to store a write-permission bit for each memory page and map the write-permission bits into the processor's address space. The system could then use these write-permission bits to provide fine-grained protection at the physical page level, replacing the virtual memory and code-patching schemes described in Section 2.1.

There are several engineering implications as well if memory contains permanent data. A system should be able to be reset without erasing memory; and CPU caches, because they contain memory data, should also preserve their contents on a normal reset. DEC Alphas allow a reset and boot without erasing memory or the CPU caches [DEC94]; the PCs we have tested do not. To make data accessible during a hardware failure, it should be possible to move a memory board to a different machine without losing power (just as disks can be moved without losing data) [Moran90, Baker92a].

6 Related Work

We divide the research related to this paper into two areas: field studies/fault injection and protection schemes.

6.1 Field Studies and Fault Injection

Studies have shown that software is the dominant cause of system outages [Gray90], 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) [Sullivan91]. Lee and Iyer study and classify software failures in Tandem's Guardian operating system [Lee93a]. These studies provide valuable information about failures in production environments; in fact, many of the fault types in Section 3.1 were inspired by the major error categories from [Sullivan91] and [Lee93a]. 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 [Sullivan95].

Software fault injection is a popular technique for evaluating the behavior of prototype systems in the presence of hardware and software faults. See [Iyer95] for an excellent introduction to the overall area and a summary of much of the past work on fault injection, such as FINE [Kao93], FIAT [Barton90], and FERRARI [Kanawati95]. As with field studies of system crashes, these papers on fault injection inspired many of the fault categories used in this paper. However, we know of no paper on fault injection that has specifically measured the effects of faults on permanent data in memory.

6.2 Protecting Memory

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.

The only file system we are aware of that attempts to make all permanent files reliable while in memory is Phoenix [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. Our proposed mechanism in Section 2.1 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.

Harp protects a log of recent modifications by *replicating* it in volatile, battery-backed memory across several server nodes [Liskov91]. The Recovery Box keeps special system state in a region of memory accessed only through a rigid interface [Baker92b]. 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 [Banatre91]. 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. [Needham83] suggests 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 [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].

Finally, several papers have examined the performance advantages and management of reliable memory [Copeland89, Baker92a, Biswas93, Akyurek95], and countless papers have sought to improve disk performance via data placement, logging, scheduling, and so forth.

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 extra protection, warm reboot makes files in memory about as safe as files written through to disk while performing 20 times faster than write-through file systems. Rio makes all writes immediately permanent, yet performs faster than systems that lose 30 seconds of data on a crash: 35% faster than a standard delayed-write file system and 8% faster than a system that delays both data and metadata. We recommend Rio without protection for most situations. For applications that demand even higher levels of reliability, Rio's optional protection mechanism makes memory even safer than a write-through file system while while lowering performance 20% compared to a pure write-back system.

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, changing the assumptions about write traffic behind 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 efficiently in arbitrarily large units. Thus Rio improves performance moderately over delayed-write systems.
- Applications requiring instant permanence need no longer write synchronously to disk; this vastly improves performance over write-through systems.
- Applications need no longer lose 30 seconds of data on a crash, because all updates are permanent as soon as they reach the file cache. Thus Rio improves reliability significantly over delayed-write systems. For systems without battery backup, warm reboot can be used to eliminate the 30 seconds of data often lost when systems crash.

To further test and prove our ideas, we have installed a departmental file server using the Rio file cache without protection and with reliability-induced writes to disk turned off. Among other things, this file server stores the active copy of this paper and the sole copy of 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 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.

[Akyurek95] Sedat Akyurek and Kenneth Salem. Management of partially safe buffers. *IEEE Transactions on Computers*, 44(3):394–407, March 1995.

[APC96] The Power Protection Handbook. 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.

[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.

[Biswas93] Prabuddha Biswas, K. K. Ramakrishnan, Don Towsley, and C. M. Krishna. Performance Analysis of Distributed File Systems with Non-Volatile Caches. In *Proceedings of the 1993 International Symposium on High Performance Distributed Computing (HPDC-2)*, pages 252–262, July 1993.

[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.

[Chutani92] Sailesh Chutani, Owen T. Anderson, Michael L. Kazar, Bruce W. Leverett, W. Anthony Mason, and Robert N. Sidebotham. The Episode File System. In *Proceedings of the 1992 Summer USENIX Conference*, pages 43–60, January 1992.

[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.

[DEC95] August 1995. Digital Unix development team, Personal Communication.

[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.

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

[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.

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

[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.

[Needham83] R. M. Needham, A. J. Herbert, and J. G. Mitchell. How to Connect Stable Memory to a Computer. *Operating System Review*, 17(1):16, January 1983.

[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.

[Ousterhout90] John K. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *Proceedings USENIX Summer Conference*, pages 247–256, June 1990.

[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.

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

[Srivastava94] Amitabh Srivastava and Alan Eustace. ATOM: A System for Building Customized Program Analysis Tools. In *Proceedings of the 1994 Conference on Programming Language Design and Implementation (PLDI)*, pages 196–205, June 1994.

[Sullivan91] Mark Sullivan and R. Chillarege. Software Defects and Their Impact on System Availability–A Study of Field Failures in Operating Systems. In *Proceedings of the 1991 International Symposium on Fault-Tolerant Computing*, June 1991.

[Sullivan95] Mark Sullivan, December 1995. personal communication.

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

[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.