University of Michigan CSE-TR-273-95

Measuring and Improving Memory's Resistance to Operating System Crashes

Wee Teck Ng, Gurushankar Rajamani, Christopher M. Aycock, Peter M. Chen
Computer Science and Engineering Division
Department of Electrical Engineering and Computer Science
University of Michigan
{weeteck,gurur,caycock,pmchen}@eecs.umich.edu

Abstract: Memory is commonly viewed as an unreliable place to store permanent data because it is perceived to be vulnerable to system crashes.[1] Yet despite all the negative implications of memory's unreliability, no data exists that quantifies how vulnerable memory actually is to system crashes. The goals of this paper are to quantify the vulnerability of memory to operating system crashes and to propose a method for protecting memory from these crashes.

We use software fault injection to induce a wide variety of operating system crashes in DEC Alpha workstations running Digital Unix, ranging from bit errors in the kernel stack to deleting branch instructions to C-level allocation management errors. We show that memory is remarkably resistant to operating system crashes. Out of the 996 crashes we observed, only 17 corrupted file cache data. Excluding direct corruption from copy overruns, only 2 out of 820 corrupted file cache data. This data contradicts the common assumption that operating system crashes often corrupt files in memory. For users who need even greater protection against operating system crashes, we propose a simple, low-overhead software scheme that controls access to file cache buffers using virtual memory protection and code patching.

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 fairly reliable places to store long-term data such as files. However, random-access memory is commonly viewed as an unreliable place to store *permanent data* (files) because it is perceived to be vulnerable to power outages and operating system crashes [Tanenbaum95, page 146].

Memory's vulnerability to power outages is straightforward to understand and fix. A simple solution is to add an uninterruptible power supply to the system. Another solution is to switch to a non-volatile memory technology such as Flash RAM [Wu94]. We do not consider power outages further in this paper.

Memory's vulnerability to OS crashes is less concrete. 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]. As evidence of this view, most systems periodically write file data to disk, and transaction processing applications view transactions as committed only when the changes are made to the disk copy of the database.

The reason most people view battery-backed memory as unreliable yet view disk as reliable 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. Functions that use the device driver are checked for errors, and functions 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 function 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 assumption that memory is unreliable hurts system performance, reliability, simplicity, semantics, and cost.

- Because memory is unreliable, systems that require high reliability, such as databases, write new data through to disk, but this slows performance to that of disks. Many systems, such as Unix file systems, mitigate the performance loss caused by extra disk writes by only writing new data to disk every 30 seconds or so, 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

1. It is also vulnerable to power loss, but this paper will not discuss this aspect of reliability. It is possible to make memory non-volatile by using an uninterruptible power supply or by using Flash RAM.

fraction of writes must eventually be written through to disk anyway. A longer delay can decrease disk traffic due to writes, but only at the cost of losing more data. The extreme approach is to use a pure write-back scheme where data is only written to disk when the memory is full. This is only an option for applications where reliability is not an issue, such as compiler-generated temporary files.

- Memory's unreliability also increases system complexity [Rahm92]. Increased disk traffic due to extra write backs forces the use of extra disk optimizations such as disk scheduling, disk reorganization, and group commit. Much of the research in main-memory databases deals with checkpointing and recovering data in case the system crashes [GM92, Eich87].
- Ideal semantics, such as atomicity for every transaction, are also sacrificed because disk accesses are slow and memory is unreliable. Finally, memory's unreliability forces systems to keep a copy of permanent memory data on disk; this shrinks the available storage capacity.

Although it is common to assume that files in memory are vulnerable to operating system crashes, there is remarkably little data on how often these crashes actually do corrupt files in memory. The objectives of this paper are as follows:

- To quantify the vulnerability of memory to OS crashes. The ideal way to measure how often system crashes corrupt files in memory would be to examine the behavior of real system crashes. Unfortunately, data of this nature is not recorded (or is not available) from production systems. We use software fault injection to induce a wide variety of operating system crashes in our target system (DEC Alphas running Digital Unix) and find that the file cache is almost *never* corrupted.
- To propose a software mechanism that protects memory even further from system crashes. We combine the system's virtual memory protection and code-patching to lower the chance that wild stores will corrupt memory.

The rest of this paper is organized as follows: Section 2 reviews the work most closely related to this research; Section 3 describes the platform and mechanisms used in the experiments; Section 4 describes the different types of faults injected into the system; Sections 5 and 6 describe and discuss the results of our experiments; and Section 7 proposes a software mechanism that protects memory from system crashes.

2 Related Work

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

2.1 Field Studies of System Crashes

Studies have shown that software has become the dominant cause of system outages [Gray90]. Many studies have investigated system software errors. The studies most relevant to this paper investigate operating system errors on production IBM and Tandem systems. Sullivan and Chillarege classify software faults in the MVS operating system and DB2 and IMS database systems; in particular, they analyze faults that corrupt program memory (overlays) [Sullivan91, Sullivan92]. Lee and Iyer study and classify software failures in Tandem's Guardian operating system [Lee93a, Lee95]. These studies provide valuable information about failures in production environments; in fact many of the fault types in Section 3 were inspired by the major error categories from [Sullivan91] and [Lee95]. However, they do not provide specific information about how often system crashes corrupt the permanent data in memory.

2.2 Using Software to Inject Faults

Software fault injection is a popular technique for evaluating how prototype systems behave in the presence of hardware and software faults. We review some of the most relevant prior work; see [Iyer95] for an excellent introduction to the overall area and a summary of much of the past fault injection techniques.

The most relevant work to this paper is the FINE fault injector and monitoring environment [Kao93]. FINE uses software to emulate hardware and software bugs and monitors the effect of the fault on the Unix operating system. Another tool, FIAT, uses software to inject memory bit faults into various code and data segments [Segall88, Barton90] of an application program. FERRARI also uses software to inject various hardware faults [Kanawati95, Kanawati92]. FERRARI is extremely flexible: it can emulate a large number of data, address, and control faults, and it can inject transient or permanent faults into user programs or the operating system.

As with field studies of system crashes, these papers on fault injection inspired many of the fault categories used in this paper. However, no paper on fault injection have specifically measured the effects of faults on permanent data in memory.

2.3 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 7 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.

The Harp file system 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 functions 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 [Banatre86, Banatre88, Banatre91]. Our work seeks to make main memory reliable without needing special-purpose hardware or dual memory banks.

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; the condition they suggest is that a certain register has been pre-loaded with the memory word's previous content. 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. Finally, 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 flash RAM, which is non-volatile [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].

3 Experimental Environment and Mechanisms

Our experiments were run on DEC Alpha 3000/600 workstations (Table 1) running the Digital Unix V3.0 operating system. Digital Unix is a monolithic kernel derived from Mach 2.5 and OSF/1.

machine type	DEC 3000
model	600
CPU chip	Alpha 21064, 175 MHz
SPECint92	114
SPECfp92	165
memory bandwidth	207 MB/s
memory capacity	128 MB (512 MB max capacity)
system bus	Turbochannel
system bus bandwidth	100 MB/s

Table 1: Specifications of Experimental Platform [Dutton92].

What data does the user expect and want to remain intact after a system crash? The user probably does not really want or expect all memory data to survive; after all if the entire state of the machine were preserved, the newly rebooted machine would likely crash again! The abstraction used to distinguish permanent data from transient data is *files*[2]. We thus want to preserve the file cache—all data in memory that relates to files, including both file data and metadata.

Digital Unix stores file data in two distinct buffers. 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 normally 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, the UBC uses about 60% of the physical memory (80 MB of the 128 MB on each computer).

We use two strategies to detect corruption of the file cache: checksums and a synthetic workload called *memTest*.

3.1 Checksum Detection of Corruption

Our primary method to detect corruption is to maintain a checksum of each memory block in the file cache [Baker92b]. We update the checksum in all functions that write the file cache; unintentional changes to file cache buffers will result in an inconsistent checksum. We identify blocks that are being modified during a crash by marking a block as *changing* (by using a special checksum value) before writing to the block. Because file cache updates are not yet atomic, blocks being modified during a crash cannot be identified as corrupt or intact.[3]

User programs that use the mmap interface pose a special problem for our mechanism for detecting corruption. Programs that mmap files into their virtual address space need not call any kernel functions to update the file image. Instead, stores into their address space *implicitly* change the file cache. Because the changes do not go through any kernel function, we have no opportunity to update the checksum. If this happens, the system will erroneously show the block as corrupted. We detect this situation by marking mmap'ed blocks in the file cache as *changing*. This limits our ability to detect corruption of these pages. Fortunately, programs that use mmap (with PROT_WRITE and MAP_SHARED[4]) are relatively rare, and the general-purpose file system workload we use to stress the system contains no programs that write to mmap'ed files.

3.2 Workload Detection of Corruption

One could argue that faults could indirectly call file caching routines with erroneous arguments. These would not be caught by the checksum mechanism, since these file cache routines would correctly manipulate the checksum of any buffers they corrupt.[5] Catching these errors requires a higher-level check on specific data, 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 corruption for any arbitrary workload; *memTest* provides a higher-level check on certain data by knowing its correct value at every instant. In practice, checksums proved to be sufficient; all crashes that memTest detected as having corruption were also detected by checksums.

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 iteration, *memTest* records its progress in a status file on disk. After the

2. File systems and databases are the two major systems that store permanent, long-term data, that is, data the user intends to survive system crashes. In this paper, we use file-system terminology, such as files and file caches, because our implementation to date in Digital Unix has focused on file systems. Many of the ideas described here apply to databases, and we plan to test our ideas in this arena as well. We use the term *file cache* to include any area of memory that caches long-term data, such as the Unix buffer cache, a database buffer cache, or the virtual memory system for operating systems that map files into memory. We also include any mapping information necessary to find and interpret the contents of files in memory.

3. Similarly, writes to disk blocks that occur when the system crashed are not guaranteed to be atomic. We are modifying the file caching code to make file cache updates atomic.

4. PROT_WRITE indicates that the page is writable. MAP_SHARED indicates that changes to the page update the permanent file image (unlike MAP_PRIVATE).

5. As an aside, note that undetected indirect errors would likely propagate corrupted data to disk.

system crashes, we reboot the system and run *memTest* until it reaches the point when the system crashed. This reconstructs the correct contents of the test directory at the time of the crash, and we then compare the reconstructed, correct contents with the file cache image in memory. To examine the file cache image in memory, we perform a *warm reboot* during which all files in memory at the time of the crash are restored to disk [Chen95].

We have two other goals in designing the workload. First, we want a general-purpose workload that calls many different programs. Second, we want to stress the file system with real programs that expanded the file cache to include most of main memory. To create a general-purpose workload, we run four copies of the Andrew benchmark [Howard88, Ousterhout90]. Andrew creates and copies a source hierarchy; examines the hierarchy using find, ls, du, grep, and wc; and compiles the source hierarchy. Since the file working space for Andrew is quite small (a few MB), we supplement this general-purpose workload with three copies of a script that copies, compresses, and uncompresses the kernel image (9.3 MB). This expands the UBC to about 60% of physical memory (80 MB out of 128 MB). Running more copies did not expand the UBC further; the machine merely started to thrash.

4 Description of Faults

This section describes the types of faults we inject to measure memory's resistance to operating system crashes. Many different hardware and software faults can cause operating system crashes. Our primary goal in designing these experiments is to generate a wide variety of system crashes. We use software to emulate both software and hardware faults because software fault injection has proven to be an easy and effective injection mechanism [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. Hardware faults are usually specific and relatively easy to model [Lee93b], and various techniques such as ECC and redundancy have been successfully used to protect against these errors [Abbott94, Banatre93]. We focus primarily on software faults because:

- Kernel programming errors are the errors most likely to circumvent hardware error correction schemes and corrupt memory.
- Software errors (like most design flaws) are difficult to model and understand. After all, if you knew exactly what was wrong with your program, you'd fix it! Our understanding of software errors is hazy, and this erodes our confidence that memory will survive a crash caused by a software bug.

In choosing what type of fault to inject, there is a tradeoff between the size of the fault universe it can generate and how realistic the fault is. Random faults such as changing memory words are very general, because almost any real fault can be expressed as a change in memory state. However, it is difficult to relate specific software or hardware errors to the changes of state that are injected. On the other hand, faults such as misallocating memory can be quite realistic. However, these can only mimic specific real faults. For example, misallocating memory can not precisely mimic the behavior of most erroneous if statements. We classify the faults we inject into three categories: random bit flips, low-level software faults, and high-level software faults. Each succeeding fault category is progressively more realistic.

4.1 Random Bit Flips

The first category of faults flips randomly chosen 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. For kernel text tests, we corrupt ten randomly chosen instructions in memory after the system is up and running. We corrupt ten instructions rather than one to increase the probability that a corrupted instruction will be executed. For kernel heap tests, we corrupt ten randomly chosen words in the kernel heap. For kernel stack tests, we corrupt one word near the top of each kernel thread's stack.

Most crashes occurred within 15 seconds after the fault was injected for all faults in this paper. If a fault does not crash the machine after ten minutes, we halt and reboot the system.

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.

4.2 Low-Level Software Faults

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]. The first fault in this category is an assignment faults. One type of assignment fault changes the destination register used by an instruction; the other changes a source register. The second fault is a condition check fault; these are injected by changing a branch to a noop. We also change random instructions (both branch and non-branch) to noops. We inject ten faults for each specific fault type (corrupt destination register, corrupt source register, change branch to noop, change random instruction to noop).

4.3 High-Level Software Faults

The last category of faults imitate specific programming errors. These are more targeted at specific programming errors than low-level software faults are. We inject an initialization fault by deleting (turning into noops) the instructions in the kernel text responsible for initializing a variable at the start of a function [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. For both initialization and pointer corruption fault tests, we inject ten faults and halt the system if it stays up for more than ten minutes.

We also inject two of the common, high-level programming errors described by [Sullivan91]: allocation management and copy overruns. In an allocation management fault, a module continues to use a region of memory after it has deallocated it. We inject this fault by modifying the kernel malloc function to occasionally prematurely free the newly allocated memory. We implement this by having malloc start a new thread that sleeps for a random interval of 0-256 ms, then calls free.[6] Malloc is set to inject this error every 1000-4000 times it is called; this occurs approximately every 15-30 seconds.

A copy overrun occurs when a module copies bytes past the end of a buffer. We inject this fault by modifying the kernel's bcopy function to occasionally increase the number of bytes it copies. The length of the overrun was distributed as follows: 50% corrupt one byte; 44% corrupt 2-1024 bytes; 6% corrupt 2-4 KB. This distribution was chosen by starting with the data gathered in [Sullivan91] and modifying it somewhat according to our specific platform and experience. bcopy is set to inject this error every 1000-4000 times it is called; this occurs approximately every 15 seconds.

5 Results

This section focuses on two major results from the fault-injection experiments: the variety of crashes and their effect on memory-resident files.

The faults listed in Section 4 were designed to generate as many different realistic crashes as possible, and they were very successful at this. We generated a total of 996 crashes (approximately 90 of each type of fault) over a one-month period on three workstations. We observed 76 unique crash error messages and used these error messages to divide the crashes into six categories. The first category is kernel memory fault/unaligned access. These accounted for the largest fraction of crashes (78%), which is consistent with prior results in the field [Lee93a, Kao93]. The second largest category is kernel consistency check (11%), followed by user process failure (4%), hardware error (2%), illegal instruction (2%), and unknown (3%). Table 2 shows the distribution of crashes for each type of fault.

Our primary goal in performing this fault-injection experiment was to measure how often crashes corrupt the file cache. The data in Table 3 show that the overwhelming majority of crashes do *not* corrupt any data in the UBC or buffer cache. The circled column indicates those runs that corrupt the file cache without completely corrupting the disk. These are the corruptions we are most concerned with, since they indicate memory's increased vulnerability to crashes. A number of other crashes irreparably corrupted the disk image of the file system; that is, the disk had to be reformatted. These are indicated by the rightmost column and illustrate that disks are similarly vulnerable to crashes. We are unable to determine if the file cache was corrupted by these disk corruptions, though this is irrelevant since the file cache does no good if the underlying disk image is destroyed. We do nothing special to detect disk corruption, so some of the crashes that corrupted file cache data may also have corrupted some data on disk.

Apart from copy overrun faults, only two crashes out of 820 (a mere 0.2%) corrupted file cache data. One of these corruptions was caused by a bit-flip in the kernel text and corrupted three UBC buffers. The other corruption was caused by a bit-flip in the kernel stack and corrupted three UBC buffers. Note that these are among the least realistic types of faults.

Most corruptions (15 out of 17) resulted from copy overrun faults. 14 of these 15 crashes each corrupted one UBC buffer; the other crash corrupted one buffer cache buffer. Copy overruns have a relatively

6. We also had to modify free to first check to make sure the original module had not already freed the block of memory.

high chance (8.5%) 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. We distinguish between *direct* corruption of the file cache, where the fault itself corrupts the file cache, and *indirect* corrupt of the file cache, where the fault triggers a chain of events that eventually corrupts the file cache. We believe all corruptions caused by bcopy were direct corruptions of the file cache rather than indirect corruptions. To support this hypothesis, we note that 1) the corrupted area of each buffer always began from the beginning of the buffer and 2) corruptions from the other types of faults were extremely rare.

Fortunately, direct corruption from faults such as copy overruns are the easiest type of corruption to protect against. Section 7 describes a method for keeping pages write-protected when not actively being written by the file cache code. Copy overruns caused by non-file cache routines will not unlock a file cache page before attempting to write to it, so these writes will trigger a protection violation.

To sum up the results of the fault-injection experiments, file cache corruptions of any sort were exceedingly rare. This stands in sharp contrast to the general feeling among computer scientists that oper-

Fault Type	**# of crashes**	**% kernel memory faults**	**% kernel consis-tency checks**	**% user process failures**	**% hard-ware errors**	**% illegal instruc-tions**	**% unknown**
bit flips in kernel text	74	67.6%	5.4%	10.8%	2.7%	13.5%	0%
bit flips in kernel heap	73	87.7%	8.2%	0%	2.7%	0%	1.4%
bit flips in kernel stack	120	85.0%	2.5%	0%	5.0%	2.5%	5.0%
change destination reg	75	85.3%	4.0%	8.0%	1.3%	1.3%	0%
change source reg	75	77.3%	6.7%	10.7%	0%	2.7%	2.7%
change branch to noop	62	51.6%	27.4%	14.5%	0%	0%	6.5%
change random instruction to noop	97	72.2%	14.4%	6.2%	2.1%	1.0%	4.1%
initialization fault	76	81.6%	14.5%	2.6%	0%	0%	1.3%
pointer corruption	65	81.5%	7.7%	4.6%	0%	0%	6.2%
allocation management	103	74.8%	23.3%	0%	0%	0%	1.9%
copy overrun	176	84.1%	6.8%	0%	2.8%	1.7%	4.5%

Table 2: Distribution of Crashes for Each Type of Fault. The faults described in Section 4 successfully generated a diverse set of crashes. We generated a total of 996 crashes and observed 76 unique crash error messages.

ating system crashes often corrupt files in memory [Tanenbaum95 page 146, Silberschatz94 page 200]. Nearly all corruptions that did occur were copy overruns that directly wrote into memory-resident files, not some mysterious chain of events where faulty modules affected other kernel modules and eventually corrupted the file cache. This is consistent with results from FINE that show most faults do not propagate to other kernel modules [Kao93]. Overall, only 1.7% of all crashes corrupted the file cache. Excluding direct corruption from copy overruns, only 0.2% of crashes corrupted the file cache.

6 Discussion

In this section, we discuss why the difference between disk and memory reliability is not as great as one might think. The first reason is that the huge majority of operating system crashes do not corrupt files in memory. Systems tend to fail quickly due to virtual memory protection and kernel consistency checks, and errors do not tend to propagate between different kernel modules [Kao93].

To illustrate memory's reliability, consider a system that crashes once per month (a pessimistic estimate for production-quality operating systems). If 1% of crashes corrupt files in memory, data will be lost only about once every eight years! This is comparable to the MTTF (mean time to failure) of disks, which indicates that memory's vulnerability to system crashes is no worse than disk's vulnerability to disk crashes.

Second, even if operating system errors did corrupt memory, these corruptions would often find their way back to the on-disk copy of the file. This is because memory serves as an intermediary between the

Fault Type	# of crashes	# of crashes that corrupt file cache without corrupting disk	# of crashes that corrupt disk (and possibly file cache as well)
bit flips in kernel text	74	**1**	4
bit flips in kernel heap	73	0	0
bit flips in kernel stack	120	**1**	0
change destination reg	75	0	0
change source reg	75	0	1
change branch to noop	62	0	0
change random instruction to noop	97	0	4
initialization fault	76	0	6
pointer corruption	65	0	0
allocation management	103	0	7
copy overrun	176	**15**	0
total	**996**	**17**	**22**

Table 3: Effect of Crashes on File Cache and Disk. This table shows that very few of the 996 crashes we generate corrupt the file cache. This stands in sharp contrast to the general feeling that operating system crashes are quite likely to corrupt files in memory. The circled column indicates those runs that corrupt the file cache without completely corrupting the disk. These are the corruptions we are primarily concerned with, since they indicate memory's increased vulnerability to crashes. A number of crashes irreparably corrupted the disk image of the file system; that is, the disk had to be reformatted. These are indicated by the rightmost column and illustrate that disks are also vulnerable to crashes. We are unable to determine if the file cache was corrupted by these crashes, though this is irrelevant since the file cache does no good if the underlying disk image was destroyed. Note that some of the crashes that corrupted file cache data may also have corrupted some data on disk.

processor and disk. Most systems have a file cache, and any corruption of the file cache will eventually get written back to disk if the system stays up long enough. Ironically, write-through systems—which strive for optimal safety by writing data immediately through to disk—are the systems most vulnerable to propagating memory errors back to disk. Even if a system crashes soon after an error (before the corrupted memory has a chance to get written to disk), many systems try to write all dirty file cache data back to disk as the last step before halting. If memory is corrupted as a result of the crash, this faulty data is guaranteed to be made permanent! We are conducting tests to precisely measure how often faults corrupt data on disk.

Some may claim that the 30-second write-back delay imposed by most Unix systems actually *increases* disk reliability by not writing back the last 30 seconds of data before a crash. Even if this is true, it is trivial to implement the same delay for memory data; simply mark memory data as permanent 30 seconds after you write it.

7 A Proposal for Protecting Memory from Operating System Crashes

Our results show that files in memory are almost always preserved without corruption during operating system crashes. Some users, however, need an even higher level of assurance that files in memory are safe. This section describes a proposal for how to protect file in memory even further from operating system crashes.

A disk is protected from operating system crashes by its interface—to change its contents, the system must go through the disk device driver (or closely imitate it). We believe memory-resident files can be protected from crashes in much the same way by strictly controlling the way memory can be written [Baker92b]. To accomplish this protection, we propose adding a *memory device driver* to check for errors and prevent misbehaving software from corrupting memory. The memory device driver is the only module in the operating system allowed to change files in memory—any write to the file cache that does not use the memory device driver should cause an exception [Chapin95]. The main question then is the *protection mechanism*: how does the system cause an exception when other modules try to change the file cache without using the memory device driver?

An ideal protection mechanism would have the following characteristics:

- **Lightweight**: the protection mechanism should add little or no overhead to file cache accesses: it should not need to be invoked on memory reads [Needham83] and should have minimal overhead on writes.
- **Enforced**: it should be extremely unlikely for a non-malicious kernel function to accidentally bypass the protection mechanism. The vast majority of errors should be trapped.
- **Simple**: the protection mechanism should require little change to the existing system. In particular, avoiding custom hardware would enable us to modify the system more quickly and make our results more widely applicable.

At first glance, the virtual memory protection of a system seems ideally suited to protect the file cache from unauthorized stores [Copeland89]. By keeping the write-permission bits in the page table entries turned off for the file cache pages, the system will cause most unauthorized stores to encounter a protection violation. To write a page, the memory device driver enables the write-permission bit in the page table, writes the page, then disables writes to the page. The only time a file cache page is vulnerable to an unauthorized store is while it is being written by the memory device driver, and disks have the same vulnerability, since the disk sector being written during a system crash can be corrupted. The memory device driver can check for corruption during this window verifying the data after the write is completed. Or the memory device driver 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 Alpha processor with the two most significant bits equal to 10_2 bypass the TLB.[7] To protect against these physical addresses, we can modify the kernel object code, inserting a check before every kernel store; this is called *code patching* [Wahbe93]. If the address is a physical address, the system checks to make sure the address is not in the file cache, or that the file cache has explicitly registered the address as writable.[8]

Initially, the idea of inserting code before every store instruction sounds prohibitively slow. However, several optimizations make the actual overhead quite reasonable. First, modifications to the stack pointer

7. It may be possible to configure the system to disallow physical addresses, but this presents other difficulties because the system uses physical addresses to initialize the virtual memory system, access I/O devices, and manipulate page tables.

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 therefore decrease the number of checks by replacing the checks on local, stack variables with a few checks on the stack pointer [Wahbe93]. Another method to lower the checking overhead is to replace individual checks in loops with a few higher-level checks. For example, functions such as bcopy modify sequential blocks of data; these blocks can be checked once rather than checking every individual store. Initial performance data for Digital Unix indicates that the overhead of code patching is only 2-10% [Chen95].

We are currently completing an initial implementation of this protection mechanism and plan to evaluate how effectively it lowers the risk of memory corruption for the faults described in Section 3.

8 Conclusions and Future Work

We have shown that memory is remarkably resistant to operating system crashes. Out of the 996 crashes we observed, only 17 corrupted any file cache data (1.7%). Excluding direct corruption from copy overruns, only 2 out of 820 corrupted file cache data (0.2%). This data contradicts the common assumption that operating system crashes often corrupt files in memory.

Thus, even without special mechanisms for protecting files in memory, battery-backed memory may already be as reliable as disks. For situations where greater protection against operating system crashes is required, we proposed a simple, low-overhead software scheme that controls access to file cache buffers using virtual memory protection and code patching.

One direction for future work is to redo this study on a different operating system or to perform a similar fault-injection experiment on a database system. We believe these will extend the applicability of our conclusions.

If main memory is indeed safe from system crashes, battery-backed main memory should be viewed as stable storage in the same way that disks are. This has striking implications for future system designers. Write-backs to disk are needed much less often; optimizations such as group commit may disappear; checkpointing data in main-memory databases may no longer be needed.

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.

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

[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.

[Banatre86] Jean-Pierre Banatre, Michel Banatre, Guy LaPalme, and Florimond Ployette. The design and building of Enchere, a distributed electronic marketing system. *Communications of the ACM*, 29(1):19–29, January 1986.

[Banatre88] Michel Banatre, Gilles Muller, and Jean-Pierre Banatre. Ensuring data security and integrity with a fast stable storage. In *Proceedings of the 1988 International Conference on Data Engineering*, pages 285–293, February 1988.

[Banatre91] Michel Banatre, Gilles Muller, Bruno Rochat, and Patrick Sanchez. Design decisions for the FTM: a

8. It is possible to use this check on every store in place of virtual memory protection, but this forces a full check for both physical and non-physical addresses. The combination of virtual memory protection and code patching allows most stores to be checked quickly; only physical addresses need a full check. We are currently measuring the performance of both alternatives.

general purpose fault tolerant machine. In *Proceedings of the 1991 International Symposium on Fault-Tolerant Computing*, pages 71–78, June 1991.

[Banatre93] Michel Banatre, Pack Heng, Gilles Muller, Nadine Peyrouze, and Bruno Rochat. An experience in the design of a reliable object based system. In *Proceedings of the 1993 International Conference on Parallel and Distributed Information Systems*, pages 187–190, January 1993.

[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.

[Chen95] Peter M. Chen, Christopher M. Aycock, Wee Teck Ng, Gurushankar Rajamani, and Rajagopalan Sivaramakrishnan. Rio: Storing Files Reliably in Memory. Technical Report CSE-TR-250-95, University of Michigan, July 1995.

[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.

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

[Dutton92] Todd A. Dutton, Daniel Eiref, Hugh R. Kurth, James J. Reisert, and Robin L. Stewart. The Design of the DEC 3000 AXP Systems, Two High-Performance Workstations. *Digital Technical Journal*, 4(4):66–81, 1992.

[Eich87] Margaret H. Eich. A classification and comparison of main memory database recovery techniques. In *Proceedings of the IEEE International Conference on Data Engineering*, pages 332–339, February 1987.

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

[GM92] Hector Garcia-Molina and Kenneth Salem. Main Memory Database Systems: An Overview. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):509–516, December 1992.

[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. The Zebra Striped Network File System. In *Proceedings of the 1993 Symposium on Operating System Principles*, pages 29–43, December 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.

[Kanawati92] Ghani A. Kanawati, Nasser A. Kanawati, and Jacob A. Abraham. FERRARI: a tool for the validation of system dependability properties. In *Proceedings of the 1992 International Symposium on Fault-Tolerant Computing*, pages 336–344, July 1992.

[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 Engi-*

neering, 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.

[Lee93a] 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.

[Lee93b] Inhwan Lee, Dong Tang, Ravishankar K. Iyer, and Mei-Chen Hsueh. Measurement-Based Evaluation of Operating System Fault Tolerance. *IEEE Transactions on Reliability*, 42(2):238–249, June 1993.

[Lee95] Edward K. Lee. Highly-Available, Scalable Network Storage. In *Proceedings of the 1995 IEEE Computer Society International Conference (COMPCON)*, 1995.

[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.

[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.

[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.

[Rahm92] Erhard Rahm. Performance Evaluation of Extended Storage Architectures for Transaction Processing. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, pages 308–317, June 1992.

[Segall88] Z. Segall, D. Vrsalovic, D. Siewiorek, D. Yaskin, J. Kownacki, J. Barton, R. Dancey, A. Robinson, and T. Lin. FIAT–Fault Injection Based Automated Testing Environment. In *18th Annual International Symposium on Fault-Tolerant Computing*, pages 102–107, 1988.

[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.

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

[Sullivan92] Mark Sullivan and Ram Chillarege. A Comparison of Software Defects in Database Management Systems and Operating Systems. In *Proceedings of the 1992 International Symposium on Fault-Tolerant Computing*, pages 475–484, July 1992.

[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.