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.

[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.

[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.

[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.

[Lee95] Inhwan Lee and Ravishankar K. Iyer. Software Dependability in the Tandem GUARDIAN System. *IEEE Transactions on Software Engineering*, 21(5):455–467, May 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.

[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.

[Powell95] David Powell, Eliane Martins, Jean Arlat, and Yves Crouzet. Estimators for Fault Tolerance Coverage Evaluation. *IEEE Transactions on Computers*, 44(2):261–273, February 1995.

[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.

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

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

[Thakur95] Anshuman Thakur, Ravishankar K. Iyer, Luke Yong, and Inhwan Lee. Analysis of Failures in the Tandem NonStop-UX Operating System. In *Proceedings of 1995 International Symposium on Software Reliability Engineering*, pages 40-49, Nov1995.

[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.

is also an inherent uncertainty in the data collection and classification process that is not explicitly addressed in these studies.

Our platform consists of a modern 64-bit RISC processor, and a monolithic kernel derived from the popular UNIX and Mach OS. We believe our results apply to a broad range of research and commercial environments, but further experiments are required to prove this.

9 Conclusions and future work

We have shown that memory is remarkably resistant to operating system crashes. Only 10 of the 650 crashes we observed (1.5%) corrupted any file cache data; disk was corrupted at a similar rate (1.1%). This data contradicts the common assumption that operating system crashes corrupt files in memory much more often than files on disk.

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 have implemented a simple, low-overhead software scheme that controls access to file cache buffers based on virtual memory protection [Chen96].

To gather more realistic data on crashes, we have installed a departmental file server that uses warm reboot and disables reliability-induced disk writes. Among other things, this file server stores this paper and the sole copy of the authors' mail. Another 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. We believe the availability of a fast, safe storage medium will provide new opportunities for any application that uses persistent data.

10 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. Technical report, American Power Conversion, 1996.

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

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

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

[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.

[Chen96] Peter M. Chen, Wee Teck Ng, Gurushankar Rajamani, and Christopher M. Aycock. The Rio File Cache: Surviving Operating System Crashes. To appear, *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, October 1996.

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

[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.

[GM92] Hector Garcia-Molina and Kenneth Salem. Main

and the MVS and Tandem studies, together with the resulting weights.

Table 6 shows the mean corruption rate (95% confidence interval). In all cases, memory has higher mean corruption rate than disk. However, we note that the absolute difference in mean corruption rate is very small (0.21% to 0.46%), and thus, the mean memory corruption rates for both MVS and Tandem fault distributions are less than 1.5%. We have also implemented a low overhead memory protection scheme that lowers memory's corruption rate to be even better than that of disk [Chen96].

These results stand in sharp contrast to the general feeling among computer scientists that operating system crashes often corrupt files in memory. While slightly less reliable than disk, memory is *much* more reliable than we had expected and is reliable enough for most systems. To illustrate this level of reliability, 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 memory 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 memory! Even though the faults we inject probably do not perfectly represent real-world crashes, the conclusion is clear: *warm reboot enables memory to be about as reliable as disk.*

7 Discussion

In this section, we discuss why memory reliability is nearly as good as disk reliability.

First, the huge majority of operating system crashes do not corrupt files in memory. This is 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, Lee93]. Thus, 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].

Weight Distribution	**Disk**		**Memory**	
	Mean	**Half-Width**	**Mean**	**Half-Width**
Equal	1.08%	0.87%	1.54%	1.03%
Proportional MVS	0.18%	0.17%	0.60%	0.68%
Proportional Tandem	1.25%	1.29%	1.46%	1.71%

Table 6: Comparing disk and memory reliability. This table shows the average corruption rate (95% confidence). Our tests show that memory's reliability is nearly as high as a write-through system. The corruption rate of memory can be reduced to even better than that of disk using a simple memory protection scheme [Chen96].

Second, even if an operating system error does corrupt memory, the corruption will often find its way back to the on-disk copy of the file. This is because memory serves as an intermediary between the 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. 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 (e.g. Linux, NetBSD, Digital Unix, Sprite). If memory is corrupted as a result of the crash, this faulty data is guaranteed to be made permanent! For example, half the corrupted files in the memory tests were not in memory at all—they had already been written back to disk.

Some may claim that the 30-second write-back delay imposed by most Unix systems actually *increases* disk reliability by deliberately throwing away data written within 30 seconds of 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.

8 Limitations

This paper describes a controlled experiment to measure disk and file cache corruption. No prior studies have directly quantified memory's vulnerability to software crashes. However, it would be dangerous to indiscriminately apply our results, and hence we address in this section some possible limitations of our work.

We attempt to model realistic faults by implementing common programming errors and those faults reported in field studies of system crashes. We also generate a *large number* and *wide variety* of system crashes in an attempt to capture a significant portion of the fault space. We ensure that this sampling of the fault universe is independent of the workload [Powell95] by selecting the fault randomly from the fault set, injecting it at some random delay after initiating the test, and allowing the workload to warm up. The only major uncertainty is how accurately our fault model represents the real world. We attempt to provide realistic weights by mapping our fault models to published results on field crash analysis. But this mapping is limited as we cannot possibly model all faults identified, and there

As it is difficult to prove that our fault model represents real faults, we present three different interpretations of Table 4 by varying the weights associated with each fault type: equal, proportional MVS and proportional Tandem. Equal weighting assumes that each type of fault is equally likely to occur, while proportional weighting assigns a probability to each fault category according to its likelihood to occur in real environment. The confidence interval is computed using the approach outlined in [Powell95], by partitioning the fault space into disjoint sets corresponding to the fault types.

Fault Type	Number of Corruptions	
	Disk	**Memory**
kernel text	2	1
kernel heap		
kernel stack		1
destination reg.		
source reg.	2	
delete branch	1	1
delete random inst.	1	
initialization		
pointer		1
allocation		
copy overrun		4
off-by-one	1	2
synchronization		
Total	**7 of 650 (1.1%)**	**10 of 650 (1.5%)**

Table 4: Disk and memory reliability. This table shows how often each type of error corrupted data for disk and memory. Blank entries had no corruptions.

Ideally, the weights assigned should be derived from logs of Digital UNIX machines in daily use, but such data are hard to come by. For example, we only found 2 instances of system crashes on our two non-fault injected machines over the past year. Moreover, published data about the frequency of different types of faults is rare. We rely on the fault distributions published on MVS [Sullivan91b] and Tandem NonStop-UX [Thakur95] to determine the weight of each fault category. The MVS study includes a detailed breakdown of high-impact errors, with the following classification: deadlock and synchronization (47%), undefined state (25%), pointer management (11%), register reuse (4%), uninitialized data (4%), copy overrun (2%), allocation management (2%), statement logic (2%), data error (2%), unknown (2%) and other error (11%). Undefined state, unknown and other errors are too vague to be precisely modeled, so we distribute their weights evenly across other categories. The Tandem study classifies programming mistakes: pointer (28%), missing check for exception (26%), uninitialized data (4%), memory allocation (11%), and other instruction errors (31%). Table 5 shows the mapping between our fault categories

Fault Type	**MVS**		**Tandem NonStop-UX**	
	Classification	**Weight**	**Classification**	**Weight**
kernel text	instruction	0.7%	instruction	6.2%
kernel heap	data	1.4%		
kernel stack	data	1.4%		
destination reg.	register reuse	2.7%	instruction	6.2%
source reg.	register reuse	2.7%	instruction	6.2%
delete branch	instruction	0.7%	missing check	26.0%
delete random inst.	instruction	0.7%	instruction	6.2%
initialization	initialization	5.4%	initialization	4.0%
pointer	pointer	14.9%	pointer	28.0%
allocation	allocation	2.7%	allocation	11.0%
copy overrun	copy overrun	2.7%		
off-by-one	instruction	0.7%	instruction	6.2%
synchronization	synchronization	63.5%		

Table 5: Proportional mapping. This table shows how we map between the fault type in our study and those of [Sullivan91b] and [Thakur95], and the weight assign to each fault type.

then examine the file data to determine the amount of corruption.

The faults listed in Section 5 were designed to generate as many different realistic crashes as possible, and they were very successful at this. We generated a total of 1300 crashes (100 of each type of fault), which took approximately four machine-months. We observed 74 unique crash error messages, including 59 different kernel consistency error messages. These error messages are a superset of those found on logs of non-injected machines. We used these error messages to divide the crashes into six categories. The first category is kernel memory fault/unaligned access. This accounted for the largest fraction of crashes (57%), which is consistent with prior results in the field [Lee93, Kao93]. The second largest category is kernel consistency check (26%), which is the major cause of crashes in [Thakur95]. The rest of the category are deadlock (11%), application failure (4%), illegal instruction (1%), and hardware error (1%). Table 3 shows the distribution of crashes for each type of fault.

.Our primary goal in performing this fault-injection experiment is to measure how often crashes corrupt files on disk and in memory. We start by measuring disk reliability by using 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, nor do we turn off reliability-induced disk writes. Our only tool for detecting corruption on disk is *memTest*, because our checksum method cannot detect disk corruption. 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.

Table 4 shows that disk 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 these runs corrupted only a few (1-4) files/directories. 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]

The right section of Table 4 shows the reliability of memory. We turn off all reliability-related disk writes and use warm reboot 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. 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 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 in the file cache. This implies that the system remained running long enough to propagate the corruption to disk. Copy overruns have a relatively high chance of corrupting the file cache because the injected fault directly overwrites a portion of memory, and this portion of memory has a reasonable chance of overlapping with a file cache buffer.

Fault Type	**kernel memory faults**	**kernel consistency checks**	**application failures**	**hardware errors**	**illegal instructions**	**deadlock**
kernel text	60%	23%	1%	1%	11%	4%
kernel heap	79%	18%	0%	0%	0%	3%
kernel stack	80%	15%	0%	1%	2%	2%
destination reg.	69%	22%	1%	2%	0%	6%
source reg.	69%	23%	1%	2%	0%	5%
delete branch	19%	79%	2%	0%	0%	0%
delete random inst.	45%	43%	3%	3%	1%	4%
initialization	46%	51%	0%	2%	0%	1%
pointer	80%	5%	2%	2%	1%	10%
allocation	97%	2%	1%	0%	0%	0%
copy overrun	85%	13%	0%	0%	0%	2%
off-by-one	12%	31%	37%	0%	0%	20%
synchronization	0%	17%	0%	0%	0%	83%

Table 3: Distribution of crashes for each type of fault. The faults described in Section 5 successfully generated a diverse set of crashes. We generated a total of 1300 crashes and observed 74 unique crash error messages, including 59 different kernel consistency error messages.

is up and running. 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.

5.2 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 (Table 2) [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).

5.3 High-level software faults

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

6 Results

This section focuses on two major results from the fault-injection experiments: the variety of crashes and their effect on disk and memory-resident files. For each run, we inject faults to crash a running system, reboot,

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

Table 2: Relating faults to programming errors. This table shows examples of how real-world programming errors can manifest themselves as the faults we inject in our experiments. None of the errors shown above would be caught during compilation.

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 file system contents at the time of the crash (restored during the warm reboot in the memory tests).

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.

5 Description of faults

Conducting a thorough experiment on operating system crashes is a vexing problem. Data on real crashes is rarely on the platform of interest, usually describes relatively few crashes, and almost never contains enough detail to repeat the crash (and most crashes are not easily repeatable). Yet many experiments require a controlled environment with hundreds of crashes, and measuring how often files are corrupted is one of these experiments. We therefore chose to conduct a fault-injection experiment.

The primary issue in fault injection is how accurately the faults represent real-world faults. This is an especially difficult problem with software faults. Software errors are by nature design flaws, and it is notoriously difficult to precisely model human errors. Hardware faults such as bit flips are easier to model because we can more easily understand the physical processes behind the faults. Since it is difficult to prove that injected software faults represent real faults, our approach is to focus instead on generating a *large number* and *wide variety* of system crashes. Our results in Section 6 represent four machine-months of continuous testing, which is a huge improvement over currently available data on memory and disk corruption due to system crashes.

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. 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 and race conditions. Hardware faults are usually specific and relatively easy to model [Lee93], and various techniques such as ECC and redundancy have been successfully used to protect against these errors [Abbott94]. 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, but 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. See Table 2 for examples of how real-world programming errors can manifest themselves as the faults we inject in our experiments.

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. Accelerating the rate at which faults are triggered should not alter the results (other than producing them more quickly) because the behavior we are most interested in is how the system crashes after the fault is activated. If a fault does not crash the machine after ten minutes, we discard the run and reboot the system. This potentially discards faults that are activated quickly but allow the system to keep running for a long time. These long-latency faults can corrupt files, but will likely corrupt memory and disk equally because any corrupted file data cached in main memory will propagate to disk after 15-30 seconds. Thus these faults will not affect the relative reliabilities of disk and memory. In practice, we have seen only two faults (one in the disk tests, one in the memory tests) that corrupt data and leave the system running for very long.

5.1 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 memory that relates to files, including both file data and metadata. We also include any mapping information necessary to find and interpret the contents of files in memory.

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 wired in virtual memory so that it cannot be paged out, and is usually only a few megabytes. To conserve space in the address translation cache, 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 in this paper, the UBC uses 80 MB of the 128 MB on each computer.

We use different methods to measure the effects of a crash on disk and memory. To measure disk reliability, we use a write-through file cache and the default Digital Unix kernel. To achieve write-through, we force the data to be written to disk synchronously via the fsync system call after every write. Without this, many runs would lose data written within 30 seconds of the crash.

To measure memory reliability, we modify the Digital Unix kernel in several ways. First, we turn off all reliability-induced writes to disk so that the tests measure memory corruption rather than disk corruption. Reliability-induced writes include all disk writes that maintain consistency between file data cached in memory and file data 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 both synchronous and asynchronous disk writes (i.e. bwrite and bawrite) system calls to delayed disk writes (i.e. bdwrite); we take out file system calls that explicitly synchronize the file cache and disk (i.e. sync and fsync); and we modify the panic system call 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. To recover the dirty memory contents at the time of the crash, we implement *warm reboot*—a special reboot that restores to disk the dirty files in memory at the time of the crash [Chen96]. This is similar to performing a crash dump as the system is going down. While a standard crash dump often fails, however, warm reboot is performed on a healthy, booting system and will always work.

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)
system bus	Turbochannel
system bus bandwidth	100 MB/s

Table 1: Specifications of experimental platform [Dutton92].

4 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 file data cached in main memory to disk. In *indirect* corruption, a series of events eventually causes an I/O procedure to be called 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 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

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 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 (the file cache) are more vulnerable to operating system crashes than files on disk, there is remarkably little data on how often crashes actually do corrupt files in memory or on disk. The main objective of this paper is to quantify the vulnerability of disk and memory to OS crashes. The ideal way to measure how often system crashes corrupt files on disk and in memory would be to examine the behavior of a large number of real system crashes. Unfortunately, data of this nature is not recorded (or is not available) from production systems. Instead, we use software fault injection to induce a wide variety of operating system crashes in our target system (DEC Alphas running Digital Unix). We find that only 1.1% of crashes corrupt files on disk, which agrees with our intuition that disk is safe from software crashes. Surprisingly, we find that a similar fraction of crashes (1.5%) corrupt files in memory, implying that memory is much safer from operating system failures than people assume.

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 used in the experiments; Section 4 describes the way we detect file corruption; Section 5 describes the different types of faults injected into the system; Sections 6 and 7 describe and discuss the results of our experiments.

2 Related work

We divide the research related to this paper into three areas: field studies, fault injection, and protection schemes. Many papers have also examined the performance advantages and management of reliable memory [Copeland89, Baker92a, Biswas93, Akyurek95].

2.1 Field studies of system crashes

Studies have shown that software has become the dominant cause of system outages in fault-tolerant systems [Gray90, Lee95]. 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) [Sullivan91b]. Lee and Iyer study and classify software failures in Tandem's Guardian operating system [Lee93]. 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 [Sullivan91b] and [Lee93]. 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 of an application program [Barton90]. FERRARI also uses software to inject various hardware faults [Kanawati95]. 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, we know of no paper that has measured the effects of faults on permanent data in memory.

3 Experimental setup

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.

The user expects file data to survive system crashes, and hence we seek to preserve the file cache—all data in

1996 International Symposium on Software Reliability Engineering

Comparing Disk and Memory's Resistance to Operating System Crashes

Wee Teck Ng, Christopher M. Aycock, Gurushankar Rajamani, Peter M. Chen
Computer Science and Engineering Division
Department of Electrical Engineering and Computer Science
University of Michigan
rio@eecs.umich.edu

Abstract

Memory is commonly viewed as an unreliable place to store permanent data (files) because it is perceived to be vulnerable to system crashes. Yet despite all the negative implications of memory's unreliability, no data exists that quantifies how vulnerable memory actually is to system crashes. This paper quantitatively compares the vulnerability of disk and memory to operating system 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 find that files on disk are rarely corrupted (1.1% corruption rate), which agrees with our intuition. We also find that, surprisingly, files in memory are nearly as safe as files on disk. Only 10 of the 650 crashes we observed (1.5%) corrupt any files in memory. Our data contradicts the common assumption that operating system crashes often corrupt files in memory and suggests that memory can be used to store permanent data rather than needing to write it back to disk.

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 $119 uninterruptible power supply can keep a system running long enough to dump memory to disk in the event of a power outage [APC96]. 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 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]. 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. Procedures that use 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 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, and cost, and sacrifices ideal semantics.

- Because memory is unreliable, systems that require high reliability, such as databases, write new data through to disk (i.e. write-through file cache), but this slows performance to that of disks. Many systems, such as Unix file systems, wait 30 seconds before writing new data to disk (delayed writes). This improves performance by decreasing the number of writes to disk but sacrifices reliability by ensuring the loss of data written within 30 seconds of a crash. In addition,

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