University of Michigan CSE-TR-250-95

Rio: Storing Files Reliably in Memory

Peter M. Chen, Christopher M. Aycock, Wee Teck Ng, Gurushankar Rajamani, Rajagopalan Sivaramakrishnan

Computer Science and Engineering Division
Department of Electrical Engineering and Computer Science
University of Michigan
{pmchen,caycock,weeteck,gurur,sraja}@eecs.umich.edu

Abstract: Memory is currently a second-class citizen of the storage hierarchy because of its vulnerability to power failures and software crashes. Designers have traditionally sacrificed either reliability or performance when using memory as a cache for disks; our goal is to do away with this tradeoff by making memory as reliable as disks. The Rio (RAM I/O) project at Michigan is modifying the Digital Unix (formerly OSF/1) kernel to protect the file cache from operating system crashes. If successful, making memory as reliable as disks will 1) improve file cache performance to that of a pure write-back scheme by eliminating all reliability-caused writes to disk; 2) improve reliability to that of a write-through scheme by making memory a reliable place to store files long term; and 3) simplify applications such as file systems and databases by eliminating write-back daemons and complex commit and checkpointing protocols.

1 Introduction

As processors continue to double in speed every year or two, a system's storage hierarchy becomes increasingly important to the system's overall performance. Storage hierarchies combine random-access memory, magnetic disk, and possibly optical disk and magnetic tape to try to achieve the performance of the fastest level (memory) at the cost per bit of the cheapest level. Because disks and tapes are not improving in performance nearly as fast as processors are, an increasing fraction of accesses must be satisfied in main memory for the storage system to keep pace with processor performance.

Unfortunately, memory's unreliability limits the fraction of accesses it can satisfy without needing to go to disk. Because systems assume that memory is not a reliable storage area, they must do one of several things when writing to memory:

1. Sacrifice performance by writing new data immediately through to disk (or some other reliable storage medium). Systems that require durability for all writes, such as transaction processing systems, choose this option.
2. Sacrifice reliability by storing data in memory until capacity forces the data to disk. This is only possible for storing temporary files, since other files are too precious to take this kind of reliability risk with them.
3. Compromise by writing data to disk after a fixed delay, typically 30 seconds [Ousterhout85]. This is commonly done on systems that can tolerate the loss of recently-written data, such as many Unix systems. Unfortunately, 1/3 to 2/3 of newly written data lives longer than 30 seconds [Baker91, Hartman93], so a large fraction of writes must eventually be written through to disk. A longer delay can decrease disk traffic due to writes, but only at the cost of lower reliability.

In addition to hurting performance and reliability, the assumed unreliability of memory increases system complexity for applications such as main-memory databases, file systems, and transaction processing systems [Rahm92]. Much of the research in main-memory databases deals with checkpointing and recovering data in case the system crashes [GM92, Eich87]. File systems must obey complex ordering constraints when forcing metadata to disk to ensure file system consistency [Ganger94]. The unreliability of memory and the slow speed of disk accesses slows the commit speed of transactions and forces other optimizations such as group commit in applications such as transaction processing and distributed message logging [DeWitt84, Copeland89].

The assumed unreliability of memory also increases cost by a small amount. Since data must eventually be stored on disk, the file cache can only serve as a *copy* of disk data. For example, the total amount of data that can be stored using a 1000 MB disk and a 100 MB file cache is 1000 MB, not 1100 MB.

An attractive solution to all these problems is to enable memory to store files as reliably as disks store them [Copeland89]. This would allow the reliability of write-through with the performance of a pure write-back scheme, substantially simplify system design for a variety of applications, and moderately increase usable capacity by allowing the use of non-inclusive caching schemes.

Figure 1 demonstrates the potential performance improvement of this approach. It compares the performance of the Digital Unix V3.0 operating system with and without reliability-induced writes. The hardware platform is a DEC 3000/600 with 128 MB of memory, and the workload is to repeatedly write and rewrite a file of varying size in 1 MB units for five minutes. The performance with reliability writes is limited to disk speeds, while the modified kernel without reliability writes allows files that fit in memory to be accessed at memory speeds.

This paper explores the idea of reliable file caches[1]—being able to store files in main memory as reliably as storing them on disk. The two main factors limiting the reliability of memory are power outages and software corruption. Options for protecting memory against power outages include uninterruptible power supplies and switching to an inherently non-volatile memory technology such as Flash RAM [Wu94]. This paper focuses on protecting memory against software-induced corruption, which can account for 2/3 of all system outages [Gray90].

2 Why Is Memory Less Reliable Than Disks?

If someone surveyed system administrators and asked them if they would trust the contents of memory after a system crash, most of them would likely give a resounding "no!". This intuition is backed by field studies of MVS and Guardian (Tandem's operating system), which show that between 1/4 to 1/2 of all software-induced system crashes could corrupt memory [Sullivan91, Lee93]. It is not yet clear how often these crashes corrupt the file cache; we are conducting experiments to measure this.

Memory is vulnerable to software corruption because writes to memory invoke no protocols and hence are not scrutinized by any error checking—a simple store instruction by any kernel function can

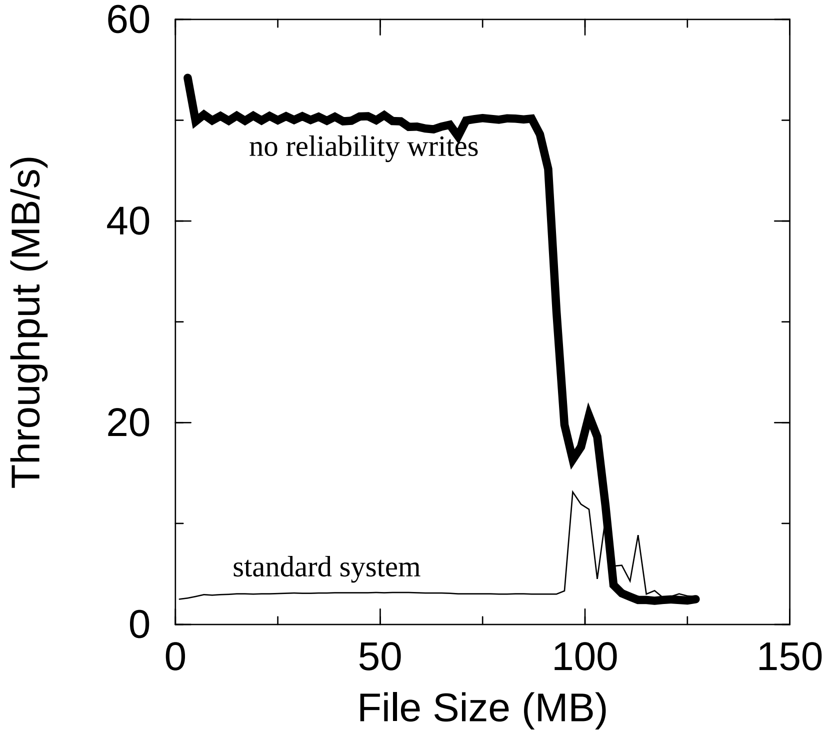

Figure 1: Performance with and without Reliability-Induced Writes. Dramatically better performance is possible if memory is made as reliable as disks, because no extra writes to disk need be done to keep the disk copy up to date. These measurements were taken on a DEC 3000/600 running Digital Unix V3.0. Data for the "no reliability writes" case was taken by modifying the Digital Unix kernel. The workload for this graph was repeatedly writing an entire file in 1 MB units.

1. By file cache we include any area of memory that caches files, such as the Unix 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.

change any data in memory. Consequently, it is easy for a random software error, such as following a bad pointer, to corrupt the contents of memory [Baker92a]. Since any operating system module can access memory, ensuring that no memory corruption can occur is a daunting task.

On the other hand, most users assume the contents of disk are intact after a system crash. Users trust the disk because the process of writing to the disk uses complex protocols, and a system crash is unlikely to mimic these protocols. Writing to a disk involves error checking at many levels, including the user-kernel library interface, the device driver, and the I/O controller. It is hence unlikely for a software error to corrupt the contents of disk because many types of errors will be caught by the strict interface.

Of course, a malicious or pathological kernel bug can always navigate through arbitrarily complex protocols, both for disk interfaces and memory interfaces, so the probability of corruption can never be zero. We are concerned only with lowering the probability that a non-malicious bug will corrupt memory, so that the overall reliability of the file cache is made as high as that of disk.

3 A Reliable File Cache

A disk is protected from software errors by its interface—to change its contents, the system must go through the disk device driver (or closely imitate it). We believe memory can be protected from software errors 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 software errors 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. 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?

Ideally, the 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 that a non-malicious kernel function could 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.

3.1 Protecting the File Cache from Unauthorized Stores

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.

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 are called KSEG addresses and bypass the TLB.[2] To protect against these physical addresses, we modify the kernel object code, inserting a check before every kernel store (this is called code patching in [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.[3]

2. It may be possible to configure the system to disallow physical addresses, but this presents other difficulties because the system uses physical addresses to start the virtual memory system, access I/O devices, and manipulate the page tables.
3. 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.

The check before every kernel store sometimes calls a procedure. If the stack pointer were corrupted, the act of calling a procedure could itself corrupt the file cache. To prevent this, we check the stack pointer whenever it is modified. Checking the stack pointer obviates the need to check stores that use the stack pointer to form an address [Wahbe93]. Since this includes all stores to local variables, this significantly lowers the number of stores that need to be checked. There are many other ways to lower the overhead of this check. For example, functions such as bcopy modify sequential blocks of data; these blocks could be checked once rather than checking every individual store.

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). Both these buffers need to be protected, but the method of protection differs. The buffer cache, which is usually only a few megabytes, is stored in virtual memory, so we use virtual memory protection and code patching to protect it. The UBC, which contains the bulk of the data, is accessed using physical addresses to avoid paging out the file cache (the virtual memory system and the UBC dynamically trade off pages depending on system workload). Because it is stored in physical memory, no virtual pages point to UBC pages, and virtual memory protection is not needed. We thus protect the UBC by solely using code patching.

The verdict is still out on whether the protection mechanism described here (VM memory protection plus code patching) meets our goals of being lightweight, enforced, and simple, though we are optimistic that it does. Code patching has slowed overall performance by only 10%, and we think that adding VM protection will not significantly affect performance. We are conducting studies to measure effective enforcement against random errors. Code patching has been relatively simple to implement, and the VM protection should also be straightforward. See Section 5 for more details on current status.

3.2 Warm Reboot

The protection scheme described above protects the file cache from being corrupted during system crashes. When the system is rebooted, it must read the file cache contents present in physical memory and update the file system with the data present in memory before the crash. We call this process a warm reboot. Because system crashes are (hopefully) infrequent, our first priority in designing the warm reboot is ease of implementation, rather than reboot speed.

Two issues that arise when doing a warm reboot are 1) what additional data should the system maintain during normal operation and 2) when in the reboot process should the system restore the file cache contents.

Maintaining additional data during normal operation makes it easier to find, identify, and restore the file cache contents in memory during the warm reboot. Without additional data, the system would need to analyze a series of data structures, such as internal file cache lists and page tables, and all these intermediate data structures would need to be protected. Instead of understanding and protecting all intermediate data structures, we keep and protect a separate area of memory, which we call the *registry*, that contains all essential pieces of information needed to find, identify, and restore files in memory. For each buffer in the UBC, we note the physical memory address, file id (device number and inode number), file offset, and size.[4] Registry information changes relatively infrequently during normal operation, so the overhead of maintaining it is low. It is also quite small; only 32 bytes of information are needed for each 8 KB file cache page.

The second issue is when to restore the file cache contents during reboot. One method is to modify the virtual memory and the UBC bootup code to restore the file cache contents instead of starting the system with an empty file cache. This method presents several difficulties. First, it requires an intimate understanding of the boot process. Second, the process must be done very early in the boot sequence so that the memory contents remain intact, but restoring the file cache is very difficult without a fully functional system. The file cache would need to be restored before the file system is up and running, for example.

To avoid these difficulties, we perform the warm reboot in two steps. Very early in the boot, we dump all of physical memory to the swap partition on disk. This saves the contents of the file cache and registry from before the crash. After the system is completely booted, we analyze the memory dump and restore the file cache using normal system calls such as open and write.

4. Rather than a logical identification such as file id and offset, we could store a physical location, such as disk block number. Storing this low-level information would allow the system to directly update the disk upon reboot, but it seems dangerous to bypass the file system when restoring data.

3.3 Memory-Mapped Files

Our protection scheme forces all writes to the file cache to go through the memory device driver. This protocol assumes writes to the file cache explicitly invoke kernel routines, which in turn call the memory device driver. Memory-mapped files, on the other hand, allow the user to do implicit I/O. The user first issues a command (such as mmap) to map a file into the user's virtual address space. The user can then execute normal loads and stores to those addresses. Because the user does not invoke explicit I/O functions for each store, there is no opportunity for the kernel to call the memory device driver, so these stores will cause protection violations. Even if it were possible to call the memory device driver on every store, the overhead would be prohibitive.

A possible solution is to create a new copy for each file cache page the user modifies (using copy-on-write). The user is then free to modify the copy. The original page remains part of the protected file cache, but the copy is not protected, so individual stores to the copy would not be considered reliable. The reliable, original page can be updated periodically (or upon msync) from the unreliable, memory-mapped page to make the changes permanent. This preserves the original reliability semantics of memory-mapped files without needing to write pages back to disk. However, it is not as reliable as explicit I/O, where each write is immediately safe.

3.4 Caveats

Perfect reliability is not possible for either memory or disks, since a software error in the kernel could accidentally mimic a memory or disk device driver. Our goal is simply to make the probability of corrupting memory as low as the probability of corrupting disk. That said, it is helpful to examine some of the errors that could corrupt memory by bypassing the combination of virtual memory protection and code patching.

The first and most obvious error that can bypass our protection scheme is an error in the file system or memory device driver. An equivalent problem exists for disk, so this does not detract from our goal of making memory as reliable as disk.

Double errors can also bypass our protection scheme. For example, a routine could first corrupt a pointer, then branch around the checking code directly to the store instruction. Or a routine could corrupt the page table (either by changing the permission bit for the buffer cache, or by mapping the UBC into a virtual address), then corrupt the file cache by writing to the corrupted entry.

4 Alternative Protection Schemes

Our first prototype combines virtual memory protection and code patching to protect the file cache from unauthorized writes, but there are other protection mechanisms as well [Copeland89].The following paragraphs discuss three alternative protection mechanisms and how well each mechanism meets the three goals of being lightweight, enforced, and simple.

One way to protect memory from software crashes is to restructure how memory is connected to the system so that it looks like a disk. This is a complete change to the memory interface—instead of performing load/stores to memory, the operating system performs disk I/Os to memory, which is then called a solid-state disk. The operating system views and accesses a solid-state disk in exactly the same way as a magnetic disk. Theoretically then, a solid-state disk should be as immune to software errors as magnetic disks are. Besides providing good enforcement, solid-state disks require no software changes; the memory device driver can be exactly the same as a disk device driver.

The weakness of solid-state disks is performance. The hardware and software interfaces used to access disks were designed with the long latencies of mechanical devices in mind, so they tend to be slower than those designed for random-access, solid-state memories. For example, overhead on a SCSI access can be as high as 1 ms. This is inconsequential compared to a 10-20 ms disk access, but it is huge compared to a 100 ns memory latency. Forming I/O control blocks, using memory-mapped registers to issue I/Os, and always dealing with sectors is fine for accessing devices with mechanical latencies, but it is inappropriate for random-access memories. In addition, both reads and writes are forced to go through the device driver. We would like to improve memory's reliability without losing the ability to access memory randomly.

Using redundancy can be used to protect memory in much the same way as with redundant disk arrays [Patterson88, Chen94]. Instead of trapping illegal accesses, this method allows a file cache page to survive some amount of corruption. For example, the system could store two copies of each file cache page or use lower-overhead error-correction such as Hamming codes. By placing different copies in different memory regions, we can make it less likely that a system crash would corrupt all copies of the page. However, this method will not protect memory against serious system crashes, which could potentially corrupt all of memory. In addition, redundancy lowers effective memory capacity and slows performance for writes.

A third approach is to modify the memory controller so that it can disallow writes to certain pages. One simple way to implement this is for the controller to store a write-permission bit for each memory page and map the write-permission bits into the processor's address space. As in the virtual memory protection scheme, file cache pages are kept write-protected. When the memory device driver wishes to write to the page, it temporarily unlocks the page, performs the write, then re-locks it. This scheme is identical to using the virtual memory system's page tables, but unlike a page table, the kernel cannot bypass it. Modifying the memory controller provides good protection and low overhead, but it does require custom hardware.

5 Status

The Rio (RAM I/O) project is currently implementing a reliable file cache on DEC Alpha workstations running Digital Unix V3.0. This section describes the status of our implementation at the time of this writing.

We use the ATOM code-modification system [Eustace95, Srivastava94] to check kernel stores to the UBC and buffer cache. For the UBC, VM protection is not needed because the UBC is stored in physical memory and cannot be accessed using virtual addresses. Thus the protection system for the UBC is nearly complete. The current system counts the number of stores to the UBC rather than raising a protection violation; this will be changed when the memory device driver is running. Checking kernel stores increases the running time of the Andrew benchmark [Howard88] by 10%. We do not check the stack pointer in Digital Unix because the stack pointer cannot be a physical address.[5]

Warm reboot is up and running. It successfully restores the UBC data present before the crash. One potential difficulty is that the default firmware initializes the entire memory during reboot. Fortunately, a simple console variable prevents this initialization. We are currently modifying the registry and warm reboot to save the buffer cache data.

In order to detect corruption, we store the checksum for each buffer in the UBC and buffer cache. This checksum is computed whenever a buffer in the file cache is modified (for example, by the ufs_write system call). On reboot, the system compares each buffer's contents with its checksum to detect any corruption. We are currently injecting memory faults to crash the operating system and measure how often crashes corrupt the file cache, both with and without our protection mechanisms.

We have not yet implemented VM protection for the buffer cache, the memory device driver, or memory-mapped files.

6 Related Work

Nearly all modern file systems use a file/buffer cache to speed up disk accesses [Nelson88, Leffler89]. Many have a memory file system that stores a complete, though temporary, file system in memory [McKusick90]. To our knowledge, however, the only file system that attempts to make its 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 the deletes obsolete pages in the original version. Rio differs from Phoenix in several ways: 1) Phoenix does not ensure the reliability of every write; instead, writes are only made permanent at periodic checkpoints; 2) Phoenix keeps multiple copies of modified pages, while Rio keeps only one copy; 3) Rio makes the file cache reliable and hence automatically deals with file systems larger than the main-memory. We plan on measuring the effective reliability of the virtual-memory protection used in Rio and Phoenix.

Several projects attempt to protect certain information from failures. 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. [Horn91] describes an

5. After the initial bootstrap, the stack pointer is a virtual address. In Digital Unix, the stack pointer is modified only by small increments, and these increments can not change it to a physical address without first causing a memory exception.

implementation of stable memory that uses dual memory banks. Hardware block copies between the two banks periodically checkpoint the memory image, but it is unclear what prevents a processor from corrupting the stable image.

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]. 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].

7 Conclusions

Memory is currently a second-class citizen of the storage hierarchy because of its unreliability against software crashes and power loss. Designers have traditionally sacrificed reliability or performance when using memory as a cache for disks; our goal is to do away with this tradeoff by making memory as reliable as disks. We have described how Rio protects memory from software crashes by forcing every file cache update to go through a memory device driver. This enforcement is done with a combination of virtual memory protection and code patching to check every kernel store.

If successful, making memory as reliable as disks will:

- improve file cache performance to that of a pure write-back scheme by eliminating all reliability-caused writes to disk;
- improve reliability to that of a write-through scheme by making memory a reliable place to store files long term;
- simplify applications such as file systems and databases by eliminating write-back daemons and complex commit and checkpointing protocols.

Imagine how much easier it would be to design a system if you could assume that your data was safe as soon as it reached memory!

8 References

[Abbott94] M. Abbott, D. Har, L. Herger, M. Kauffmann, K. Mak, J. Murdock, C. Schulz, T. B. Smith, B. Tremaine, D. Yeh, and L. Wong. Durable Memory RS/6000 System Design. In *Proceedings of the 1994 International Symposium on Fault-Tolerant Computing,* pages 414–423, 1994.

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

[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.

[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.

[Chen94] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. RAID: High-Performance, Reliable Secondary Storage. *ACM Computing Surveys*, 26(2):145–188, June 1994.

[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.

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

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

[Eustace95] Alan Eustace and Amitabh Srivastava. ATOM: A Flexible Interface for Building High Performance Program Analysis Tools. In *Proceedings of the Winter 1995 USENIX Conference*, pages 303–314, January 1995.

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

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

[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. Letter to the Editor. *Operating Systems Review*, 27(1):7–9, January 1993.

[Horn91] Chris Horn, Brian Coghlan, Neville Harris, and Jeremy Jones. Stable memory–another look. In A. Karshmer and J. Nehmer, editors, *Operating Systems of the 90s and Beyond. International Workshop Proceedings*, pages 171–177, July 1991.

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

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

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

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

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

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

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

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

[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.

[Nelson88] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout. Caching in the Sprite Network File System. *ACM Transactions on Computer Systems*, 6(1):134–154, February 1988.

[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.

[Patterson88] David A. Patterson, Garth Gibson, and Randy H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *International Conference on Management of Data (SIGMOD)*, pages 109–116, June 1988.

[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.

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

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

[Sullivan91] Mark Sullivan and R. Chillarege. Software Defects and Their Impact on System Availability–A Study of Field Failures in Operating Systems. In *Digest 21st International Symposium on Fault-Tolerant Computing*, June 1991.

[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.