Design and Implementation of Reliable Main Memory

by

Wee Teck Ng

A thesis submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy (Computer Science & Engineering)
in The University of Michigan
1999

Doctoral Committee:

Associate Professor Peter M. Chen, Chair
Assistant Professor Stephen Chick
Adjunct Professor Peter Honeyman
Associate Professor Farnam Jahanian
Professor Trevor Mudge

© Wee Teck Ng 1999
All Rights Reserved

ACKNOWLEDGMENTS

I would like to thank my parents and sisters, whose love and support made this all possible.

I would also like to thank my research advisor Peter M. Chen. Pete suggested this research topic, secured the funds to support my research, and pitched in with the coding and writing when I was working overtime to meet a paper deadline. I could not have finished my dissertation without his support and encouragement.

I am grateful to my friends in the department. Professor Toby Teorey, my database instructor and co-author in a journal paper, was instrumental in getting me involved in database research early in my career. My dissertation committee members, especially Professors Peter Honeyman and Stephen Chick, have made many suggestions that helped improve my dissertation. The Rio project team members, David Lowell, Subhachandra Chandra, George Dunlap, Olga Kornievskaia, Gurushankar Rajamani, and Chris Aycock, have provided a fun and productive environment for research. I have benefited immensely from my discussions with Dave, Chandra, and Guru on distributed systems and reliable computing. Victor Kravets, my good friend and marathon training companion, was always pushing me to go the extra mile. I am sad to say that after 1.5 years of training, I am still not ready to compete in a marathon. Charles Lefurgy, Professor Sugih Jamin, and the staff of Department Computing Organization were very generous in their equipment loan.

ABSTRACT

Design and Implementation of Reliable Main Memory

by

Wee Teck Ng

Chair: Peter M. Chen

One of the fundamental limits to high-performance, high-reliability applications is memory's vulnerability to system crashes. Because memory is commonly viewed as unsafe, applications requiring high reliability need to write data synchronously back to disk. This extra disk traffic lowers performance and necessitates complex strategies to minimize the impact of disk I/O. The goal of the Rio (RAM I/O) project is to make ordinary main memory safe for persistent storage by enabling memory to survive system crashes. Reliable memory enables a system to achieve the best of both worlds: reliability equivalent to a write-through file cache, where every write is instantly safe, and performance equivalent to a pure write-back cache, with no reliability-induced writes to disk. We apply an iterative design methodology to design systematically reliable main memories, and use it to design and implement reliable main memories in different platforms (Intel PC and Digital Alpha workstation) and application domains (operating system and databases). Our main design techniques are to protect memory during a crash and restore it either during a crash (a "safe" sync) or during a reboot (a "warm" reboot). We conduct extensive crash tests to demonstrate that our ideas are sound, and to formulate the design principles for reliable main memory.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem Statement

A modern storage hierarchy (see Figure 1.1) 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 (RAM) 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 (OS) crashes [Tanenbaum95, page 146].

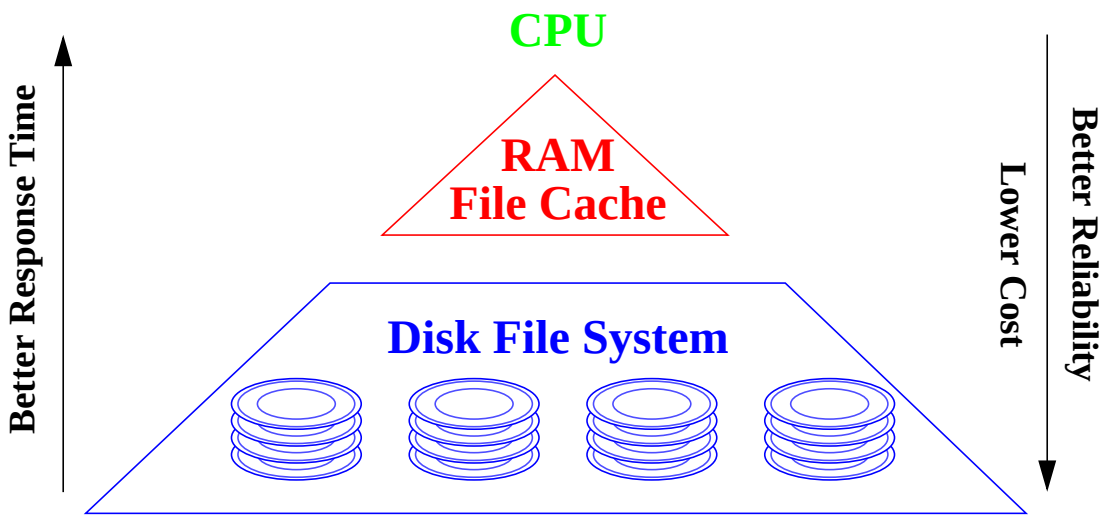

Figure 1.1 Storage Hierarchy

Memory's vulnerability to power outages is easy to understand and fix. A $100 uninterruptible power supply (UPS) can keep a system running long enough to dump memory to disk in the event of a power outage [APC96], or one can use non-volatile memory such as Flash RAM [Wu94]. We do not consider power outages further in this dissertation.

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 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 or its log 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 writing only new data to disk every 30 seconds or so, but this risks the loss of data written within 30 seconds of a crash [Ousterhout85]. In addition, 1/3 to 2/3 of newly written data lives longer than 30 seconds [Baker91, Hartman93], so a large fraction of writes must eventually be written through to disk anyway. A longer delay can decrease disk traffic due to writes, but only at the risk of losing more data. The extreme approach is to use a pure write-back scheme where data is written to disk only when the memory is full. This is an option only for applications where reliability is not an issue, such as compiler-generated temporary files.

- Memory's unreliability also increases system complexity. Increased disk traffic due to extra write backs forces the use of extra disk optimizations such as disk scheduling and disk reorganization [Ganger94b]. It is also common in file and database systems to have several levels of cache to reduce the need for disk access [Rahm92]. This hierarchy of storage complicates system design as there are many system parameters (cache size, replacement policy, etc.) to consider, and no settings will satisfy all applications.

  Many of these complex techniques also do not scale well with faster CPU speeds. For example, transaction systems overlap I/O to increase the system's multiprogramming level. While this increases the system's throughput, it does not reduce the individual transaction's response time. It does not scale well with faster CPU speeds as higher concurrency increases data contention and lock synchronization.

- Ideal semantics, such as atomicity for every transaction, are also sacrificed. Because disk accesses are slow and memory is unreliable, application programmers commonly forsake a simpler synchronous programming model for a harder asynchronous model. This allows the programmer to schedule disk I/O asynchronously and move on to other tasks, instead of waiting for the I/O to complete. An interrupt service routine is subsequently invoked when the disk I/O completes. While this approach gives better performance, it can complicate program development as an asynchronous program has a non-sequential program flow that is counter-intuitive for most programmers. Moreover, an asynchronous program is non-deterministic, further complicating performance analysis. It is also less reliable, as it could potentially lose data during system crashes.

- Finally, memory's unreliability increases system cost. It forces systems to keep a copy of permanent memory data on disk; this shrinks the available storage capacity. Applications requiring fast reliable memory have to rely on expensive hardware solutions like solid state disks.

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 mimic accidentally 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 easier for many simple software errors (such as de-referencing an uninitialized pointer) to accidentally corrupt the contents of memory [Baker92a].

1.2 Two Approaches for Reliable Memory

Many commercial I/O devices contain memory. This memory is assumed to be reliable because of the I/O interface used to access it. These devices include solid-state disks, non-volatile disk caches, and write-buffers such as Prestoserve [Moran90]. While these can improve performance over disks, their performance is limited by the low bandwidth and high overhead of the I/O bus and device interface. There is also no experimental data that show that disk I/O interface offers better protection against software errors than main memory on processor bus.

A better approach is to use ordinary main memory to store files and protect main memory using fault tolerant techniques. Most systems already have a relatively large amount of main memory and can access it very quickly. Further, main memory is random-access, unlike special-purpose devices. The main obstacle to using main memory as stable store is that it is vulnerable to software errors and system crashes [Tanenbaum95, Silberschatz94]. Several researchers attempt to overcome this by incorporating fault tolerant techniques into main memory to protect it from various hardware and software faults [Banatre91, Gait90, Abbott94]. These fault tolerant techniques include redundancy (duplicated data, memory controllers, etc.), protection (hardware controllers to enforce protection on memory access), and watchdog timers (to detect processor crashes). While most of these fault-tolerant devices are attached to the memory bus, they are still several times slower than conventional RAM [Banatre91]. They are also much more expensive than main memory due to the need for specialized hardware and redundant configuration. Moreover, with the exception of the Hive firewall described in [Chapin95], none of these papers attempt to evaluate the effectiveness of their proposals with experiments.

1.3 Our Solution: Rio (RAM I/O)

Our approach to reliable memory design is to use existing main memory as the underlying storage, and protect it from software errors and system crashes using software-based techniques. This approach incurs very little additional cost as it uses main memory that is already available on a computer. It is also portable as it does not require specialized hardware. The main potential drawbacks are performance and reliability. Software-based protection may have very high overheads [Wahbe92], and there are very little experimental data on its effectiveness against software errors.

We address these concerns by implementing reliable memories on two different hardware platforms (DEC Alpha and Intel PC) and application domains (operating systems and databases). We present benchmark results to illustrate the substantial performance benefits of our reliable memory design. We also study the reliability of our design by analyzing its reliability during system crashes. A major concern in using main memory to store permanent data is its vulnerability to software errors. For example, a pointer containing an invalid address can easily overwrite data in memory [Sullivan91b]. Although it is common to assume that files in main memory are vulnerable to operating system crashes, there is remarkably little data on how often these crashes actually do corrupt files in memory. We first quantify the vulnerability of memory and disk to OS crashes. This allows us to compare quantitatively our reliable memory design to memory and disk-based system, and to provide insights into developing effective fault tolerant techniques.

The ideal way to measure how often system crashes corrupt files in memory or disk 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. Other approaches, such as analytical modeling, are not appropriate for very large systems like UNIX operating systems or database management systems [Kao93, Kanawati95]. Our approach is to use a software fault injection tool to study system reliability. We use software fault injection to induce a wide variety of operating system crashes in our target system (DEC Alphas running Digital UNIX and Intel PC running FreeBSD) and measure the amount of corruption after the crashes.

1.4 Contributions

In summary, the main contributions of this dissertation are:

- We design and implement three reliable main memories on Intel PC running the FreeBSD operating system (OS), Digital Alpha workstation running the Digital UNIX OS, and Postgres database management system.
- We apply an iterative design methodology to design reliable main memory.
- We evaluate the critical design factors that make main memory reliable against software errors.
- We evaluate the effectiveness of our fault-tolerant techniques in a systematic and quantitative manner.

1.5 Outline of Dissertation Proposal

The section outlines the dissertation proposal and lists its overall results by chapters.

Chapter 2 reviews the work most closely related to this research. First, it gives more motivation for our research by describing the benefits of reliable memory. The next section surveys major work on field studies of system crashes. Although these papers do not provide specific information on how often system crashes corrupt file cache, they provide valuable information on software errors that cause system crashes. The next two sections describes a variety of software fault injection tools, and how these tools are used to design fault-tolerant systems. The last section lists various techniques used to protect memory from hardware and software faults.

Chapter 3 describes our design methodology and fault model. The first section describes our design methodology. We use an iterative approach to design our reliable main memory. In each iteration we measure the reliability of our design, identify categories of weakness in the design, and improve on our design. We use fault injection tools to

quantify the reliability of our design. The next section describes the fault model we developed to drive the fault injection tests. We have built a flexible fault injection tool that dynamically injects software faults into a running kernel or application. This is complemented by several tools to accurately measure corruption in memory. The last section discusses how we conduct our fault injection tests.

Chapter 4 describes the design and implementation of a reliable file cache on Digital UNIX operating system on Digital Alpha workstation. Our baseline system is a write-through file cache. We require two design iterations to make our Digital UNIX Rio file cache as reliable as a write-through file cache. The first section describes the warm reboot technique we used in the first design iteration. Warm reboot ensures that data in memory is restored to disk after system crashes. We address two main design issues: what additional data should be maintained during normal operation, and when to restore the file cache content. We present experimental results on the reliability of our first design, and analyze how we can limit the corruptions due to copy overrun faults. The next section explains how we use virtual memory protection to protect against software errors and describes two different techniques to deal with physical memory access. We find that the file cache is almost *never* corrupted during software crashes, and protecting memory makes it more reliable than disk. The last section presents the performance of Rio file cache against various file system benchmarks. Our results show that reliable memory significantly improves file system performance.

Chapter 5 describes the design and implementation of a reliable file cache on the FreeBSD operating system on Intel-based PCs. We discuss the challenges in implementing reliable memory on Intel PCs in the first section, and contrast the differences with the

Digital Alpha platform. We require four design iterations to make FreeBSD Rio file cache as reliable as a write-through file cache. We describe our design and analysis in the next four sections. We present detailed performance results in the last section.

Chapter 6 describes different ways to integrate reliable memory in databases. We show that our approach can be generalized to other application domains. We conduct fault injection tests to measure the reliability of three different designs. We find that mapping reliable memory into the database address space does not significantly affect reliability. These results mean that it is possible to create an area of main memory that is as safe as disk from process or operating system crashes.

Chapter 7 examines the design dimensions for reliable memory. We categorize the various design techniques we implemented in Chapters 4, 5 and 6, and analyze these techniques in greater detail. We describe various fault detection techniques in the first section. Our results show that we can quickly detect corruption by protecting the buffer data using VM protection mechanism, and correctly recover from system hang by using a reset key and a watchdog timer. We describe various fault recovery techniques in the next section. We examine the effect of system memory size on the design techniques in the last section. Our results show that it is critical to protect buffer data for large memory system.

The concluding chapter summarizes our findings and proposes further research to extend our work.

Chapter 2

Related Work

This chapter reviews the work most closely related to our research. The first section provides the motivation for our research. It describes the benefits of reliable memory, and explains the growing importance of disk I/O in computer system performance. Section 2.2 surveys major work on field studies of system crashes, which provide valuable insights on software errors that cause system crashes. These studies inspire the fault model used in our study to model realistic software crashes. Section 2.3 describes various software fault injection tools. We use fault injection to simulate operating system crashes and to quantify system reliability. Section 2.4 examines three different ways in which fault injection tools are used in fault-tolerant system design and describes how we integrate these different approaches into a unified design methodology for reliable memory design. The last section describes various techniques used to protect memory from hardware and software faults. We rely on some of these techniques to protect the data in our reliable main memory from software errors.

2.1 Benefits of Reliable Memory

Reliable memory can improve a computer system's performance, availability and cost. Any application that writes data to disk will run faster on reliable memory as main memory is much faster than disk. This will especially benefit applications requiring high reliability, like many types of UNIX file systems [McKusick96] which write metadata synchronously to disk to maintain file system integrity, and transaction processing system

[Gray93] which write either the log or data synchronously to disk before committing a transaction. Reliable memory can also be used to help a system recover quickly by preserving the system state across system crashes. It can either preserve the system state used in recovery [Baker92_1, Abbott94], file caches, or the whole main memory [Chen96a]. This allows a system to avoid retrieving these data from disk after a system crash and reboot quickly. Using reliable memory can ultimately result in cheaper system as it greatly improve system performance and availability on general purpose PCs and workstations. This allows system designer to support high performance applications on low cost commodity systems (e.g. a PC with a single PCI bus is much cheaper than a PC with multiple PCI buses). It can also support new applications like lightweight transactions [Lowell97] and fast checkpoints [Chen98]. These applications need the high bandwidth and low latency of reliable memory, and they cannot be easily supported on disk-based system.

Most prior work on reliable memory tend to focus on the performance advantage of reliable memory, and not the cost or availability advantages of reliable memory. This is because most commercial and experimental reliable memories are built from costly proprietary hardware parts [Banatre91, Abbott94]. Most also have inadequate protection from software hazards [Abbott94] such as wild pointers and copy overruns, and are not suitable for storing system state. We will examine in greater detail prior work that quantify the performance benefits of using reliable memory in file systems and databases in the following sections.

2.1.1 Benefits of Reliable Memory for File Systems

Baker et al. analyze how non-volatile memory can improve I/O performance in a distributed file system [Baker92a]. Two configurations are analyzed: non-volatile file caches

on client workstations to reduce write traffic to file servers, and write buffers for write-optimized file systems to reduce server disk accesses. The results show that a megabyte of NVRAM on diskless clients reduces the amount of file data written to the server by 40 to 50%. On the server side, providing a one-half megabyte write-buffer per file system reduces disk accesses by about 90% on one heavily-used file system.

Akyurek et al. [Akyurek95] analyze the management of partially safe buffers using trace-driven simulation. Partially safe buffers consist of both reliable memory and volatile memory. Akyurek studies different buffer management policies, including cache replacement strategies, safe buffer cache size, workload characteristics, and buffer write-back strategies. The main design parameter is safety of data in memory. The study found that a small amount of reliable memory (~3% of total memory) is sufficient for significant performance gains, and under lightly loaded hard disks, partially safe buffer performs as well as volatile buffers.

2.1.2 Benefits of Reliable Memory for Databases

Reliable memory can be used to store the log (or the tail of the log). Keeping the log in reliable memory removes all synchronous disk writes from the critical path of a transaction [Copeland89]. This decreases transaction commit time and can help to reduce lock contention and increase concurrency [DeWitt84]. It also removes the need for group commit, which improves log throughput at the cost of increased transaction commit time. Storing the log in reliable memory can also decrease disk bandwidth waste due to logging, because many log records can be removed before being written to the log disk [DeWitt84, Hagmann86]. For example, undo records may be removed if they belong to transactions that have committed, and redo records may be removed if they belong to transactions that

have aborted. Finally, critical information may be stored in the stable memory to help improve recovery time. For example, storing an appropriate pointer in reliable memory can save scanning the log to find the last checkpoint [DeWitt84].

A more aggressive use of reliable memory is to store the database buffer cache, or to store an entire main-memory database [GM92, Bohannon97]. This makes all buffer cache changes permanent without writing to disk. Like the force-at-commit policy, this eliminates the need for checkpoints and a redo log in recovering from system crashes (partial redo) [Haerder83, Akyurek95]. This simplifies and accelerates recovery, because there is no need to redo incomplete operations; each commit is essentially a transaction consistent checkpoint. The redo log is still needed in recovering from media failures (global redo); however, redundant disk storage makes this scenario less likely [Chen94]. Because undo records can be eliminated after a transaction commits, removing the redo log implies that *no* log records need be written if memory is large enough to contain the undo records for all transactions in progress [Agrawal89]. In addition, storing the database buffer cache in reliable memory allows the system to begin operation after a crash with the contents present prior to the crash (a warm cache) [Sullivan93, Elhardt84, Bhide93].

Several studies have looked at the performance advantage of a main memory database versus a traditional database system with a large buffer cache. Lehman compares the performance of a memory resident DBMS, which store the entire database in memory, and a disk-oriented DBMS that keeps the entire database cached in its memory buffer in [Lehman92]. The memory resident DBMS out-performs the disk-oriented DBMS by a factor of four, based on the Wisconsin benchmark [Bitton83]. A newer study by Oracle on VLM (Very Large Memory) database systems compares the performance of conventional

disk-based and memory-resident DBMSs on a variety of benchmarks, ranging from sequential scan to four-way joins. The performance improvement for the memory-resident benchmark ranges from 3 (sequential scan) to 140 times (four-way joins). A related study [Kawaf96] investigates why VLM database systems perform better than a conventional database systems. The study indicates that VLM improves the use of hardware and software caches, main memory, and I/O systems. It also results in fewer processor stalls and provides faster locking in multiprocessor configuration.

Storing the log and/or the buffer cache in reliable memory can thus simplify and accelerate database systems. A recent study shows that using a persistent database buffer cache can yield a system 40 times faster than using a non-persistent buffer cache, even when both run on reliable memory [Lowell97]. Figure 2.1 compares the performance of

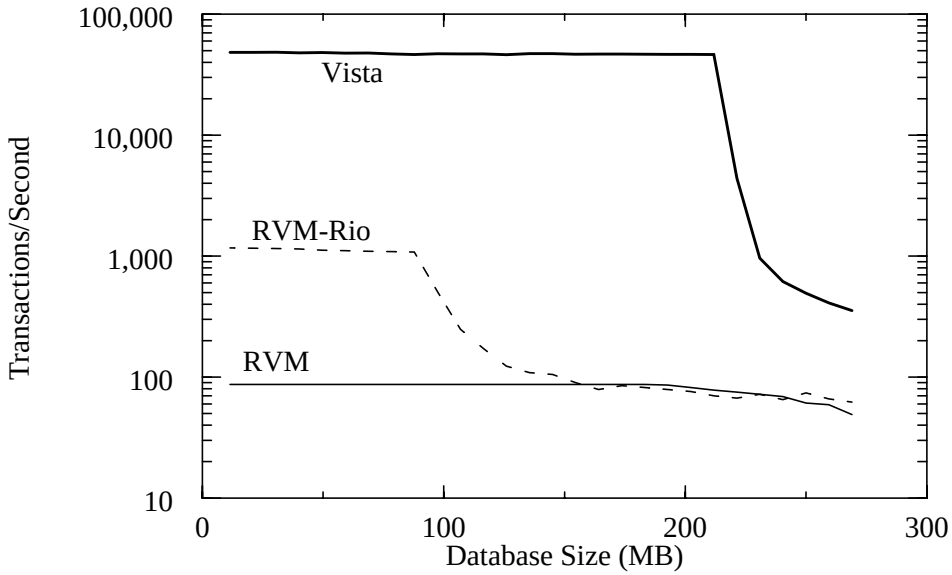

Figure 2.1 Performance Improvements with Reliable Memory. This figure shows the performance of three different transaction systems on a DEC 3000/600 with 256 Mbyte memory, running a workload based on TPC-B. RVM is a simple transaction system without reliable memory. Running RVM on Rio (RVM-Rio) provides an I/O interface to reliable memory and speeds RVM up by a factor of 13. Vista uses a memory interface to reliable memory and achieves a factor of 40 speedup over RVM, even though both run on Rio.

three systems on a workload based on TPC-B. RVM is a simple transaction system with a redo log and achieves about 100 transactions/second without reliable memory [Satyanarayanan93]. Running RVM on Rio with an I/O interface to reliable memory speeds it up by a factor of 13. Vista is a transaction system tailored to run on Rio. By using a persistent buffer cache, Vista achieves a factor of 40 improvement over RVM, even though both run on Rio. Vista also avoids the double buffering that causes RVM-Rio performance to drop at 100 MB.

2.1.3 Benefits of Reliable Memory for Future Applications

Reliable memory is going to be even more important for future applications. Figure 2.2 [Rosenblum95] shows the simulated normalized execution time of database and program development workloads on three different machine model: 1994, 1996 and 1998. It is can be seen that disk I/O is a first order bottleneck for application performance. For the database workload, the fraction of execution time spent in disk I/O increases from 36% of

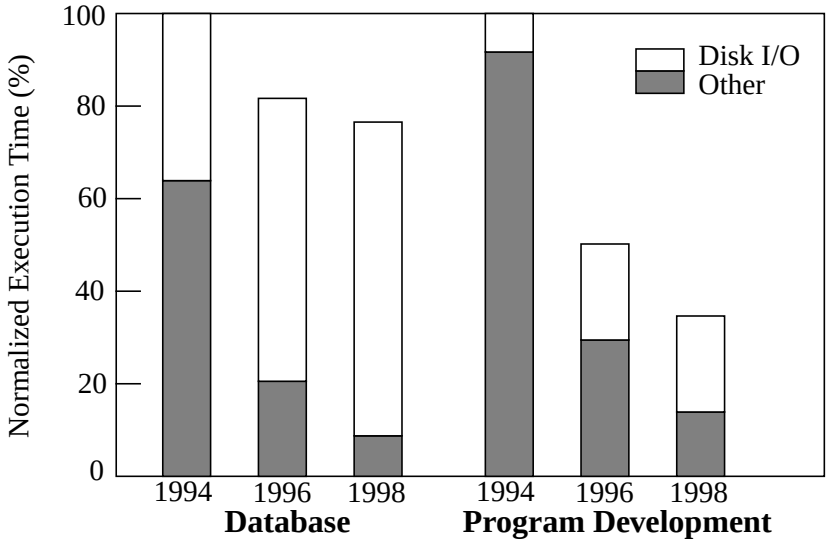

Figure 2.2 Execution Time Profile on Next-Generation Machines. This figure is taken from [Rosenblum95] and shows execution time of a database (Sybase SQL server running the TPC-B benchmark) and program development workload on three machine models. The time is normalized to the speed of the 1994 model. Without reliable memory, disk I/Os will be the first-order bottleneck to higher performance.

the workload on the 1994 model to 75% of the time in the 1996 model and over 90% of the 1998 model. Conventional techniques to hide disk latencies, like prefetching and write buffers, are not effective, as all disk accesses bypass the file system interface, and there is little that the operating system can do to reduce the I/O time. There is thus a strong need to look into reliable memory now.

2.2 Field Studies of System Crashes

Studies have shown that software has become the dominant cause of system outages [Gray90, Gray91]. Gray analyzed the field data for fault-tolerant Tandem systems in [Gray90], and found that system outage caused by software grew from 33% to 60% from 1985 to 1990. In comparison, system outage caused by hardware decreased from 50% to 10%. This trend is likely to continue given the rise in program size and features, increased use of parallel and distributed applications, improvements in hardware and maintenance, and inadequacy of current techniques to tolerate software design faults [Gray91].

Many studies have investigated system software errors. The studies most relevant to this dissertation investigate operating system errors on production IBM and Tandem systems. Sullivan and Chillarege classified software faults in the MVS operating system and DB2 and IMS database systems; in particular, they analyzed faults that corrupt program memory (overlays) [Sullivan91b, Sullivan92]. Their study found that most overlay errors are due to boundary conditions and allocation problems, and not timing or synchronization problems. They also found that most overlays are small, and corrupt data near the data that the programmer meant to update. Their research provides valuable insights into common programming mistakes that cause software to fail, events that cause latent errors in programs to surface in the field, and failure symptoms.

Lee and Iyer studied and classified software failures in Tandem's Guardian operating system using memory dumps collected from field software failures [Lee93a, Lee93b]. They also identified the effects of software faults on the system, and traced the propagation of the effects to other subsystems. Their results indicate that the fault-tolerant Tandem system tolerates 82% of the reported field software faults, that 72% of reported field software failures are recurrences of known software faults, and that the majority of the faults (82%) are either quickly detected or do not propagate to other subsystems.

A recent study by Thakur and Iyer provides a detailed classification of faults on a modern UNIX operating system (Tandem's NonStop-UX) [Thakur95]. Their results identify the modules (e.g. device drivers, memory subsystem, streams mechanism, etc.) in the operating system that generate the most faults and the modules in which most errors are detected. The paper also presents the distributions of the failure and repair times, which can offer insights into software quality.

These studies provide valuable information about failures in production environments; in fact many of the fault types analyzed in Chapter 4 were inspired by the major error categories from [Sullivan91b], [Lee93a] and [Thakur95]. However, they do not provide specific information about how often system crashes corrupt the permanent data in memory.

2.3 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 dissertation is the FINE fault injector and monitoring environment [Kao93]. FINE uses software to emulate *both* hardware (memory, CPU and bus) and software (initialization, assignment, condition check and function) bugs. Both types of faults can be dynamically injected into user program or operating system via kernel trap. The results from fault injection experiments on UNIX operating system indicate that memory and software faults have longer latency than bus faults and CPU faults. The latter category of faults tend to crash the system immediately. Only 8% of faults were found to propagate to other UNIX subsystems.

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 permanent or transient faults into user program. As in FINE, the user can control the type, location and time of injection of the faults, and the faults are injected using traps and system calls. The results from fault injection experiments on user programs demonstrate that the tool is effective, and the system behaves differently under transient faults than permanent faults. This implies that a permanent fault injection method, such as bit flips in program memory image, cannot model the full spectrum of hardware faults.

Another tool, FIAT, uses software to emulate hardware faults. It injects memory bit faults into various code and data segments [Segall88, Barton90] of an application program or operating system. The fault injection mechanism is emulated using information from the compiler and loader to corrupt the program memory image. Unlike FERRARI, FIAT cannot inject transient faults.

All the above tools inject faults by modifying the target application, inserting software traps into the target code, or running the target application in special trace mode. In contrast, Xception [Carreira95] uses the processor's debugging and performance monitoring features to inject faults and monitor the activation of the faults and their impact on the target system behavior. This approach has several desirable features that makes it suitable to evaluate a wide variety of applications. It does not modify the target application, and allows the application to run at full speed. It can also simulate transient faults in the processor, memory and address bus. However, Xception can only emulate hardware transient faults, and it is difficult to extend it to inject high-level software errors.

As with field studies of system crashes, these papers on fault injection inspired many of the fault categories and injection techniques used in our research. However, no papers on fault injection have specifically measured the effects of faults on permanent data in memory. Thus, we have created a new fault injection tool to measure memory's resistance to operating system crashes. We have implemented our tool on the Digital UNIX and FreeBSD operating systems.

Table 2.1 compares the features of various software fault injection tools. Our tool differs from the above tools in the following ways:

- We need a flexible tool to cover most interesting faults, including high-level software faults like memory leak and copy overrun. FIAT, FERRARI and Xception can only inject hardware faults, while FINE can inject both hardware and software faults.
- Our detection mechanism can accurately measure memory reliability by a combination of checksumming and comparison with known data. The other tools are tailored towards detecting faults, and measuring fault latency or propagation characteristics.

- Our target platform are modern multithreaded UNIX kernels (Digital UNIX and FreeBSD). The other platforms are single-threaded UNIX OS (IBM RT or Sun OS4.X), or experimental OS (PARIX in Xception).
- Most of the existing tools are used to characterize the system failure process (e.g. fault propagation, error detection latency). Some are used to validate the dependability and fault tolerance properties of a fault-tolerant system, or to compare the robustness of different designs. We use our tool for all these purposes to design and implement our reliable memory.

Tool	**FIAT**	**FERRARI**	**FINE**	**Xception**	**Rio**
Hardware	PC RT	Sun SPARC	Sun SPARC	PowerPC	DEC Alpha and Intel PC
Injection Target	OS, Applications	Applications	OS, Applications	OS, Applications	OS, Applications
Monitor	Software	Software	Software	Software	Software
Fault types	Memory CPU Communication	Memory CPU Bus Control flow	Memory CPU Bus Software	Memory CPU	Memory CPU Control flow Software
To evaluate	Characterize failure	Characterize failure	Characterize failure	Validation	Characterize Failures, Validation, Robustness benchmark

Table 2.1 Features of Software-Implemented fault Injection Tools. This table extends Table 3.3 from [Iyer95]. The hardware platform is relevant here since most fault injection tool is tailored to a specific platform. Injection target refers to where the bug is injected into the system: the operating system (OS) or applications. Monitor refers to the mechanism used to monitor the progress of the fault injection experiments. A software monitor is more flexible but it may not be able to monitor processor states not visible to higher level software (e.g. bus signal). Fault type refers to the type of faults that the tool can inject. Most tools are used to characterize the failure process and validate dependability properties.

2.4 Using Fault Injection to Design Fault-tolerant Systems

Software fault injection can be used for many purposes, such as understanding how systems behave during a fault, validating fault-tolerant mechanisms, and comparing the robustness of different systems.

Fault injection is traditionally used to understand how systems behave during a fault. Chillarege et al. [Chillarege89] use fault injection to characterize large system failures. They inject software bugs on a commercial transaction processing system, and analyze the crash data to measure system component failure rates and fault latency. Barton et al. [Baron90] use the FIAT fault injection tool to inject memory bit faults into data and code of two different applications. They produce detailed statistics on fault manifestations and error detection latencies. Kao et al. [Kao93] use the FINE fault injection tool to study fault propagation in the UNIX operating system and construct a fault propagation model based on Markov chains.

Fault injection is also widely used to validate fault-tolerant mechanisms and system dependability. Arlat et al. [Arlat90] use fault injection to validate the dependability of fault-tolerant systems against transient hardware/software faults. They develop a validation methodology, and demonstrate it on two different systems. Hudak et al. [Hudak93] conduct fault injection tests to determine the effectiveness of various fault-tolerant software techniques, such as n-version programming, against design and hardware faults. Rela et al. [Rela96] use fault injection to evaluate the effectiveness of software consistency checks against transient hardware faults. They inject pin-level faults into a processor and measure its effect on software applications. Silva et al. [Silva96] uses Xception fault injection tool [Carreira98] to injection transient faults into parallel computers running large

applications. They measure the effectiveness of various fault tolerant techniques (e.g. memory protection, assertions, etc) in increasing the system's error detection coverage.

There are many recent work that use fault injection to compare the robustness of different systems. Siewiorek et al. [Siewiorek93] propose a benchmark suite to measure system robustness using fault injection. The system components under study include file management system, memory access, fault tolerant mechanisms in user application and C library functions. Tsai et al [Tsai96] extend the benchmark approach to dependable systems. They used the FTAPE tool to evaluate two fault tolerant computers. Koopman et al. measure the robustness of various commercial UNIX operating systems [Kropp98] and different releases of the same operating systems [Koopman99]. They inject faults into system calls by changing the input data values, and observe the system response. Their benchmark results can be use to identify weakness of different release of the same operating system, or operating systems from different vendors.

Our design methodology also uses fault injection. However, unlike the above studies, we use fault injection to quantify the reliability of our design, understand why our design fails, and evaluate the effectiveness of different fault-tolerant design techniques. We are not aware of any prior work that uses fault injection for all three of these purposes to guide the design and implementation of a fault-tolerant system.

2.5 Protecting Memory

This section reviews several proposals to protect memory from software and hardware failures. With the exception of [Chapin95], none of these papers attempt to evaluate their proposals with experiments.

Phoenix is the only file system we are aware of that attempts to make all permanent files reliable while in main memory [Gait90]. Phoenix keeps two versions of an in-memory file system. One of these versions is kept write-protected; the other version is unprotected and evolves from the write-protected one via copy-on-write. At periodic checkpoints, the system write-protects the unprotected version and deletes obsolete pages in the original version. Our proposed reliable memory, Rio, differs from Phoenix in two major ways: 1) Phoenix does not ensure the reliability of every write; instead, writes are made permanent only at periodic checkpoints; 2) Phoenix keeps multiple copies of modified pages, while Rio keeps only one copy.

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

The Recovery Box stores system state used in recovery in a region of memory [Baker92_1]. Recovery box memory is preserved across crashes and used during the reboot of file servers. Baker and Sullivan expect few crashes to corrupt the contents of the Recovery Box and so rely primarily on checksums to verify that data is intact. They lower the chance of corruption by 1) writing to the Recovery Box through a careful interface that checks for errors and 2) storing the Recovery Box within the kernel text segment, where it is less likely to be corrupted by random pointer errors. If checksums indicate that the Recovery Box is corrupted despite these precautions, the system discards the data and performs a full recovery.

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

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

Banatre, et. al. implement stable transactional memory, which protects memory contents with dual memory banks, a special memory controller, and explicit calls to allow write access to specified memory blocks [Banatre91]. In contrast, Rio makes all files in memory reliable without special-purpose hardware or replication.

A variety of general-purpose hardware and software mechanisms may be used to help protect memory from software faults. Papers by Johnson and Wahbe suggest various hardware mechanisms to trap updates to certain memory locations [Johnson82, Wahbe92]. Hive uses the Flash firewall to protect memory against wild writes by other processors in a multiprocessor [Chapin95]. Hive preemptively discards pages that are writable by failed

processors, an option not available when storing permanent data in memory. Object code modification has been suggested as a way to provide data breakpoints [Kessler90, Wahbe92] and fault isolation between software modules [Wahbe93].

Other projects seek to improve the reliability of memory against hardware faults such as power outages and board failures. eNVy implements a memory board based on non-volatile, flash RAM [Wu94]. eNVy uses copy-on-write, page remapping, and a small, battery-backed, SRAM buffer to hide flash RAM's slow writes and bulk erases. The Durable Memory RS/6000 uses batteries, replicated processors, memory ECC, and alternate paths to tolerate a wide variety of hardware failures [Abbott94]. These schemes complement Rio, which protects memory from operating system crashes.

2.6 Summary

In this chapter we survey relevant research that will help us design and implement our reliable file cache. We look at past studies on the importance of reliable main memory, and survey field studies on large system failures. The field studies allow us to understand how large system fails and to construct a fault model that emulates realistic software crashes. We then examine various fault injection tools, and study how these tools are used to design fault-tolerant system. This allow us to apply a systematic approach to design our reliable main memory using fault injection. We also look at various memory protection techniques, which can be used to protect data in memory against software corruptions.

Chapter 3

Reliable Memory Design Methodology

This chapter describes the design methodology we used to design our reliable main memory. Our reliable main memory is called the Rio file cache. Our goal is to make Rio file cache as reliable as a write-through file cache. Write-through file caches are considered very reliable against software crashes because they propagate data immediately to disk, and disks are not easily corrupted by operating system crashes [Silberschatz94, Tanenbaum95, Chen96a]. Rio file cache is a pure write-back file cache, it writes to the disk only when the file cache overfills. Normal write-back file caches are very fast but are much less reliable than write-through caches [Chen96a]. Our challenge is to choose a strategy that will allow us to make *systematically* our design more reliable than a write-back file cache. Section 3.1 describes our design methodology, which uses fault injection to guide the design and implementation our reliable file cache. Section 3.2 describes the fault model we develop to model common software programming errors. Section 3.3 describes our experiment setup.

3.1 Design Methodology

We follow an iterative approach, as described in [Siewiorek98], to improve the robustness of a reliable file cache in the presence of operating system errors (Figure 3.1). At each iteration, we use fault injection to evaluate the reliability of our design, identify vulnerabilities, and provide quantitative results that help select techniques to address these vulnerabilities. We quantify reliability for a design by injecting various faults into a running operating system, letting it crash and reboot, and measuring how frequently the file system

is corrupted. We repeat the fault injection tests until we have collected sufficient data, and we define the *corruption rate* as the fraction of crashes that corrupt file data. We consider a particular design to be more reliable than another design if it has a lower corruption rate. We also use the data collected during fault injection to analyze and fix faults. Note that we do not merely fix the faults we ourselves have injected. Rather, we use the bugs we inject to reveal categories of faults, then we fix the entire category.

We continue to iteratively improve on our design until we achieve a design that is as reliable as our baseline system, a write-through file cache. We find that several design iterations are needed to reach our reliability goal, because the first iteration may introduce new bugs, or may leave secondary vulnerabilities hidden. A key feature of our methodology is that we use fault injection to remove faults on real systems, unlike prior simulation-based fault removal studies [Iyer95]. Because we inject faults into real systems, we can

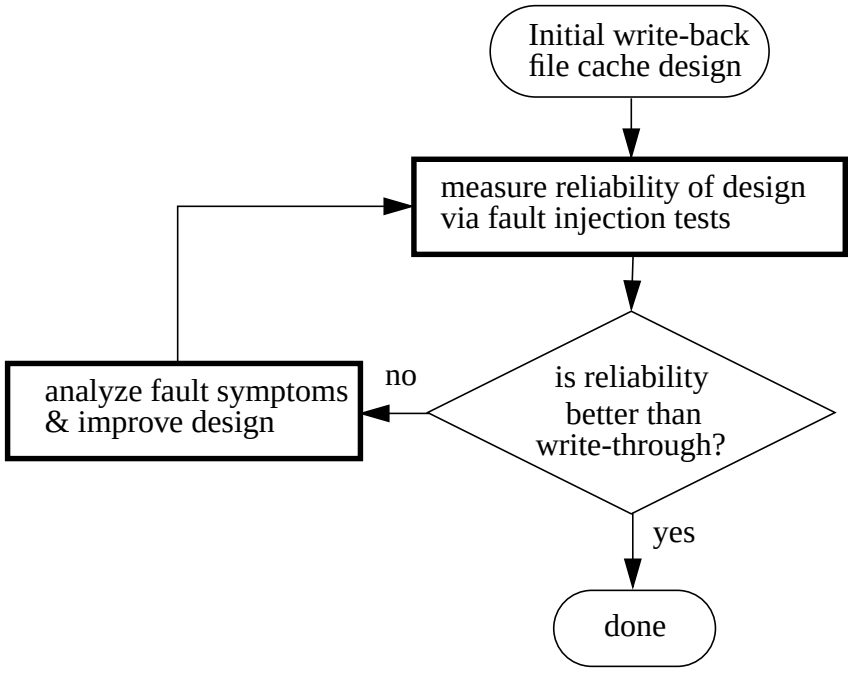

Figure 3.1 Design Process. The bold boxes represent the stages that use fault injection data to guide the design process.

accurately characterize the system failure process and fault propagation without the performance overheads of simulation-based tools.

3.2 Fault Model

An important component of our design methodology is the fault model we use to drive our fault inject tests. We want our fault model to model common software programming errors and to generate realistic operating system crashes. We focus on modeling software errors because:

- Past field studies on commercial operating systems [Gray90, Lee93a, Thakur95] and databases [Sullivan92] failures show that software errors are the dominant cause of failures.
- Kernel programming errors are most likely to circumvent hardware error correction schemes and to 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, which erodes our confidence that memory will survive a crash caused by a software bug.

Ideally, we would like to inject real software bugs into our system, using bug reports and field data collected from real system crashes. An alternative approach is to use field data to generate an error set that emulates realistic software faults [Christmansson96]. However, 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 focus instead on generating a *large number* and *wide variety* of system crashes. Our fault models are derived from studies of commercial operating systems and databases [Sullivan92, Sullivan91b, Lee93a] and from prior models used in fault-injection studies [Barton90, Kao93, Kanawati95, Chen96a]. 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. Each succeeding fault category is progressively more realistic. See Table 3.1 for examples of how real-world programming errors can manifest themselves as the faults we inject in our experiments.

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** (flag) {body}	**while** (!flag) {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 3.1 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 (except for uninitialized variable) would be caught during compilation.

3.2.1 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. 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.

3.2.2 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]. 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).

3.2.3 High-Level Software Faults

The last and most extensive category of faults imitate specific programming errors in the operating system [Sullivan91b]. These are targeted more at specific programming errors than the previous fault category. We inject an *initialization* fault by deleting instructions responsible for initializing a variable at the start of a procedure [Kao93, Lee93a]. We inject *pointer* corruption on Intel Pentium processors by corrupting the addressing bytes of instructions which access operands in memory [Sullivan91b, Lee93a]. We either flip a bit within the addressing-form specifier byte (ModR/M) or the scale, index or base (SIB) byte following the instruction opcode [Int97c]. On Digital Alpha processors, 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. We do not corrupt the stack pointer registers (i.e. *esp* and *ebp* registers on Intel processors) as these are used to access local variables instead of as a pointer variable. We inject an *allocation management* fault by modifying the kernel's malloc procedure to occasionally free the newly allocated block of memory after a delay of 0-64 ms. Malloc is set to inject this error every 1000-4000 times it is called; this fault occurs approximately every 10 seconds on our system. We inject a *copy overrun* fault by modifying the kernel's data copy procedures to increase occasionally the number of bytes they copy. The length of the overrun is 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 according to our specific platform and experience. The copy routines are set to inject this error every 1000-4000 times it is called; this fault occurs approximately every 5 seconds on our system. We inject *off-by-one* errors by changing conditions such as > to >=, < to <=, and so on. We mimic common *synchronization* errors by randomly causing the procedures that acquire/free a lock to return without acquiring/freeing the lock. We inject *memory leaks* by modifying free() to return occasionally without freeing the block of memory. We inject *interface errors* by corrupting one of the arguments passed to a procedure.

3.3 Experiment Setup

This section describes how we inject faults into our design, measure its reliability, and automate our experiments.

3.3.1 Injecting Faults

Our fault injection tool uses object-code modification to inject bugs into the kernel text. It is embedded into the kernel and, when triggered, selects an instruction in the kernel text and corrupt it. Some fault types, such as memory leaks, are implemented by modifying the relevant kernel routines (e.g. malloc) to fail occasionally when the fault is triggered. The fault trigger and injection location within the kernel text are determined by a random seed. We do not inject fault into the recovery and fault-tolerant code we added into the system.

Unless otherwise stated, we inject 10 faults for each run to increase the chance of triggering a fault. Most crashes occur within 10 seconds from the time the fault was injected. If a fault does not crash the operating system after fifteen minutes, we restart the system and discard the run; this happens about 40% of the time. Note that faults that leave the system running will corrupt data on disk for both write-back and write through file caches, so these runs do not change the relative reliability between file caches.

3.3.2 Detecting Corruption after a Crash

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

We 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 [Baker92_1]. We update the checksum in all procedures that write the file cache; unintentional changes to file cache buffers result in an inconsistent checksum. We identify blocks that were being modified while the crash occurred by marking a block as *changing* before writing to the block; these blocks cannot be identified as corrupt or intact by the checksum mechanism. Files mapped into a user's address space for writing are also marked changing as long as they are in memory, though this does not occur with the workloads we use.

Catching indirect corruption requires an application-level check, so we create a special workload called *memTest* whose actions and data are repeatable and can be checked after a system crash. Checksums and *memTest* complement each other. The checksum mechanism provides a means for detecting direct corruption for any arbitrary workload; *memTest* provides a higher-level check on certain data by knowing its correct value at every instant. Our experimental results (see Chapter 4) indicated that the corruptions detected by *memTest* are a superset of those detected by checksum. Thus the checksum mechanism is used only in our first reliable memory implementation (see Chapter 4) to verify the effectiveness of *memTest.*

memTest generates a repeatable stream of file and directory creations, deletions, reads, and writes, reaching a maximum file set size equal to the system's physical memory size. Actions and data in *memTest* are controlled by a pseudo-random number generator. After each step, *memTest* records its progress in a status file across the network. After the system

crashes, we reboot the system and run *memTest* until it reaches the point when the system crashed. This reconstructs the correct contents of the test directory at the time of the crash, and we then compare the reconstructed contents with the file cache image in memory.

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

3.3.3 Test Setup

We automate many aspects of our experiment to expedite gathering results. We conduct our experiments in parallel on several test systems (see Figure 3.2). Each test system runs a specific implementation of our file cache design. The test system contains a file partition that is reconstructed for each experiment (e.g. using the FreeBSD *newfs* command on the file system), and several script files that start a workload, select a type of fault and inject it, and log data to a network disk attached to a control host.

The experiments are controlled by a control host located on the same Local Area Network (LAN) as the test systems. The control host stores all configuration data on a shared disk that is exported via Network File System (NFS) protocol to the test systems. We also redirect the console from the test system to its serial port, and link it to the control host via a serial cable. This allows us to log crash data (crash latencies, fault symptoms, etc.) to the control host for subsequent analysis and debug the test system remotely without being physically present at the test system.

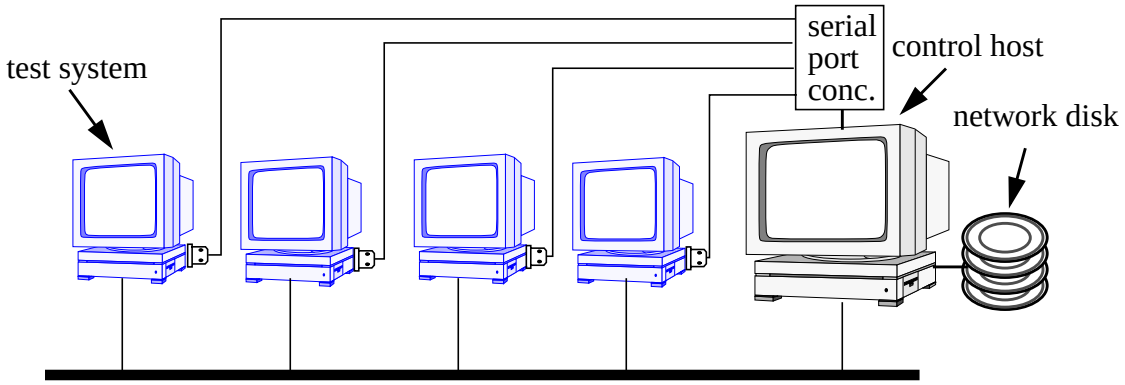

Figure 3.2 Test Equipment Setup.

3.4 Summary

We describe a systematic design methodology to design our reliable file cache. A key feature of our methodology is using quantitative data to guide the design and implementation of the system. At each iteration, we use fault injection to evaluate the reliability of our design, identify vulnerabilities, and provide quantitative results that help select techniques to address these vulnerabilities. We develop a fault model to model common software programming errors. The fault model is used in our fault injection experiments to simulate a wide variety of realistic system crashes. We also develop a test setup that automates our injection experiments.

Chapter 4

Reliable File Cache in Digital UNIX

This chapter describes our implementation of Digital UNIX Rio file cache. We use the design methodology described in Chapter 3 to iteratively refine our reliable file cache design. Section 4.1 describes our implementation platform. Section 4.2 describes the two design iterations we need to achieve our design goal. Our first design uses warm reboot to ensure that the memory content survives an operating system crash. We found that this design is less reliable than the write-through file cache. This is largely due to a significant number of corruptions caused by the copy overrun fault type. Our next design protects the data in memory and eliminates the data corruptions due to copy overrun fault type. Section 4.3 quantifies the performance advantages of our Rio file cache. Section 4.3 discusses the architecture support for reliable file cache. The material presented in this chapter first appeared in the 1996 International Conference on Architectural Support for Programming Languages and Operating Systems [Chen96a].

4.1 Implementation Platform

We implement Rio file cache on DEC Alpha 3000/600 (see Table 4.1) workstations running the Digital UNIX V3.0 operating system. Digital UNIX is a monolithic kernel derived from Mach 2.5 and OSF/1.

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. The UBC is not normally mapped into the kernel's virtual address space to conserve space in the address translation cache; 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 chapter, the UBC uses 80 MB of the 128 MB on each computer.

4.2 Design Iterations

This section describes the two design iterations we went through to arrive at the final system. Each subsection (4.2.1-4.2.2) describes the write-back file cache used in a design iteration, presents the fault-injection test results, analyzes the results to understand why our design fails, and select fault-tolerant techniques to fix the revealed weakness.

In all our designs, we modify the Digital UNIX file cache in three ways to be a pure write-back file cache. First, we disable all reliability-induced writes to disk. Digital UNIX includes tunable parameters to turn off reliability writes for the UBC. We disable buffer cache writes as in [Ohta90] by turning most bwrite and bawrite calls to bdwrite; we modify sync and fsync calls to return immediately[1]; and we modify the panic procedure to

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 4.1 Specifications of Experimental Platform [Dutton92]**.**

avoid writing dirty data back to disk before a crash. With these changes, writes to disk occur only when the UBC or buffer cache overflows, so dirty blocks can remain in memory indefinitely. Less extreme approaches such as writing to disk during idle periods may improve system responsiveness, and we plan to experiment with this in the future. The focus of this chapter is reliability, hence we take the extreme approach of delaying writes to disk as long as possible. Second, metadata updates in the buffer cache must be as carefully ordered as those to disk, because buffer cache data is now permanent. Third, memory's high throughput makes it feasible to guarantee atomicity when updating critical metadata information. When the system wants to write to metadata in the buffer cache, it first copies the contents to a shadow page and changes the registry entry to point to the shadow. When it finishes writing, it atomically points the registry entry back to the original buffer.

We configure our test systems and measure the corruption rate using the method described in Chapter 3. Table 4.2 summarizes the corruption rate by fault category of all our designs in this chapter. It presents reliability measurements for three systems: a disk-based (write-through) file cache, Rio without protection (just warm reboot), and Rio with protection. We conducted 50 tests for each fault category for each of the three systems; this represents 6 machine-months of testing.

Table 4.2 shows that corruption in write-through file cache is quite infrequent, which agrees with our intuition that disks are usually safe from operating system crashes. Of 650 crashes, only seven crashes (1.1%) corrupted any file data, and each of those runs cor-

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

rupted only a few (1-4) files/directories. Our design goal for Digital UNIX Rio file cache is to achieve a reliability that is less than or equal to 1.1%.

4.2.1 Design Iteration 1: Rio without Protection

4.2.1.1 Design

Our main technique to achieve persistent memory is to do a *warm reboot*. That is, when the system is rebooted, it must read the file cache contents that were present in physical memory before the crash and update the file system with this data. Because system

Fault Type	Disk-Based	Rio without Protection	Rio with Protection
kernel text	2	1	
kernel heap			
kernel stack		1	1
destination reg.			
source reg.	2		
delete branch	1	1	1
delete random inst.	1		
initialization			1
pointer		1	
allocation			
copy overrun		4	
off-by-one	1	2	1
synchronization			
Total	**7 of 650 (1.1%)**	**10 of 650 (1.5%)**	**4 of 650 (0.6%)**
95% Confidence Interval	**0.1%-2.0%**	**0.1%-2.9%**	**0.0%-1.2%**

Table 4.2 Comparing Disk and Memory Reliability. This table shows how often each type of error corrupted data for three systems. We conducted 50 tests for each fault type for each of three systems. The disk-based system uses fsync after every write, achieving write-through reliability. The two Rio systems test memory reliability by turning off reliability writes to disk and using warm reboot to recover the in-memory data after a crash. Blank entries had no corruptions. The last row shows the 95% confidence interval for the mean corruption rate.

crashes are infrequent, our first priority in designing the warm reboot is ease of implementation, rather than reboot speed.

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

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

The second issue is when to restore the dirty file cache contents during a system crash. Figure 4.1 shows the time-line of events during a system crash and recovery process. We can restore the file cache content before or after the system reboots. The former approach is harder, since it assumes that we are able to get control of the system when it crashes, and the restore process will proceed flawlessly during the system crash. However, this approach is attractive as it is applicable for systems that do not preserve memory content during reset. For example, Intel PC BIOS lacks a console interface, and we cannot

prevent the BIOS from erasing memory. We have to restore dirty data in memory to disk before memory is re-initialized.

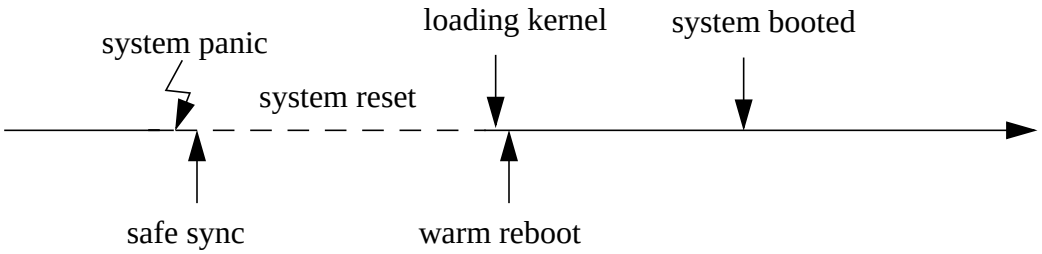

Figure 4.1 Warm Reboot Options

Since it is sometimes impossible to have control of the system when it crashes (e.g. the system may be deadlocked or seriously crippled by the fault), it is usually preferable to restore while the system boots up. In our implementation, we restored file cache after the system crashes to minimize the changes needed to the VM and file system initialization procedures. This is possible on Digital Alpha workstation as the system console has an option to turn off initializing of memory during bootup. We currently perform the restoration very early in the system bootup process (right after the kernel is loaded into memory) to ensure that file cache content is not modified by the booting kernel. After the kernel text is loaded, we scan the registry, and write all dirty pages to disk using console (PROM) routines.

4.2.1.2 Results and Analysis

The middle section of Table 4.2 shows the reliability of the Rio file cache with warm reboot. Out of 650 crashes, ten crashes (1.5%) corrupted any file data. As with the disk tests, each corruption affected a small number of files/directories, usually just a small portion of one file. *memTest* detected all ten corruptions, and checksums detected five of the ten. Interestingly, the corrupted data in the other five corruptions resided on disk rather

than the file cache. This implies that the system remained running long enough to propagate the corruption to disk.

While less reliable than disks, Rio without protection is *much* more reliable than we had expected. These results stand in sharp contrast to the general feeling among computer scientists that operating system crashes often corrupt files in memory. We believe the results are due to the multitude of consistency checks present in a production operating system, which stop the system very soon after an injected fault is encountered and thereby limit the amount of damage. In addition to the standard sanity checks written by programmers, the virtual memory system implicitly checks each load/store address to make sure it is a valid address. Particularly on a 64-bit machine, most errors are first detected by issuing an illegal address [Kao93, Lee93a].

We also observe from Table 4.2 that the fault type with the highest corruption rate is copy overruns. It is responsible for 40% (4/10) of the corruptions. 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. We attempt to eliminate these corruptions in our next design iteration by protecting the file cache buffer.

4.2.2 Design Iteration 2: Rio with Protection

To protect the memory content from erroneous writes, we need to ensure that the system does not accidentally overwrite the file cache while it is crashing. In hard disk, this protection is usually achieved by having an explicit and complex I/O interface. Writing to disk uses device drivers that form I/O control blocks and write to I/O registers. Calls to the device driver are checked for errors, and procedures that do not use the device driver are unlikely to accidentally mimic the complex actions performed by the device driver. In con-

trast, the interface used to access memory is simple—any store instruction by any kernel procedure can easily change any data in memory simply by using the wrong address. It is hence relatively easy for many simple software errors (such as de-referencing an uninitialized pointer) to accidentally corrupt the contents of memory [Baker92a].

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

4.2.2.1 Virtual Memory (VM) Protection

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

Unfortunately, many systems allow certain kernel accesses to bypass the virtual memory protection mechanism and directly access physical memory [Kane92, Sites92]. For example, addresses in the DEC Alpha processor with the two most significant bits equal to

10 bypass the TLB; these are called *KSEG* addresses. This is especially significant on the DEC Alpha because the bulk of the file cache (the UBC) is accessed using physical addresses to conserve TLB slots. We have implemented two different methods to protect against bad accesses via physical addresses.

Our current method disables the ability of the processor to bypass the TLB, that is, all addresses are mapped through the TLB. This can be done on the Alpha 21064, Intel x86, Sparc, PowerPC, and possibly other CPUs. On the Alpha 21064, a bit in the ABOX CPU control register can be set to map all KSEG addresses through the TLB. The page tables must be expanded to map these KSEG addresses to their corresponding physical addresses so the kernel can still access data such as page tables and the UBC. While issuing a KSEG address accesses the same memory location as before, the system is able to write-protect file cache pages. Disabling KSEG addresses in this manner adds essentially no overhead (see Table 4.3).

4.2.2.2 Code Patching

For processors that cannot prevent physical addresses from bypassing the TLB, a second method called *code patching* can be used, which modifies the kernel object code by inserting a check before every kernel store [Wahbe93]. If the address is a physical address, the inserted code checks to make sure the address is not in the file cache, or that the file cache has explicitly registered the address as writable. The idea of inserting code before every store instruction sounds prohibitively slow, but we have implemented several optimizations to minimize the actual overhead:

- The checking code is very efficient: 6 instructions for a virtual address (the normal case), 28 instructions for a physical address. We gain efficiency over more general tools

such as ATOM [Srivastava94] by inlining the check for virtual addresses and by increasing each procedure's stack rather than creating a temporary stack frame for each check.

- Modifications to the stack pointer occur much less frequently than stores to memory that use the stack pointer. In addition, the stack pointer is almost always modified in small increments, and these small increments cannot change a virtual address to a physical address. We can hence replace the checks on local and stack variables with a few checks on the stack pointer [Wahbe93].
- We replace individual checks in commonly used loops with a few higher-level checks. For example, procedures such as bcopy modify sequential blocks of data; these blocks can be checked once rather than checking every individual store.
- Further optimizations are possible, such as recognizing loop invariants and eliminating redundant checks within a basic block. Trends toward moving functionality out of the kernel and toward relatively faster CPUs will further lower the overhead of code patching.

However, even after a number of optimizations to reduce the number of checks, the performance of code patching is still 20-50% slower than with our current protection method [Chen96b]. Hence code patching should be used only when the processor cannot be configured to map all addresses through the TLB.

Another issue involves mmap (memory map) data. When a kernel mmaps a file, it can write to the file via a simple store instruction to the corresponding mmapped address space. Thus, kernels that use memory-mapping to cache files must be modified to map the file read-only. Kernel procedures that write to the memory-mapped file must be modified

(e.g. using code patching) to first enable writes to memory. Some OS, like Digital UNIX and Linux, does not use memory-mapping in the kernel. User memory-mapped files require no changes to the kernel because these files are not mapped into the kernel address space and hence the kernel cannot corrupt them.

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

4.2.2.3 Results and Analysis

The rightmost section of Table 4.2 shows the reliability of the Rio file cache with protection turned on. Out of 650 crashes, we measured only four corruptions (0.6%). This implies that the VM protection technique we implemented in this design iteration is very successful, and we have eliminated the corruptions due to copy overrun fault type. We use the visual test approach proposed in Section 13.4.3 of [Jain91] to compare our design with the write-through file cache. Figure 4.2 plots the 95% confidence intervals of the corruption rates and the mean corruption rates for the write-through file cache and both Rio file

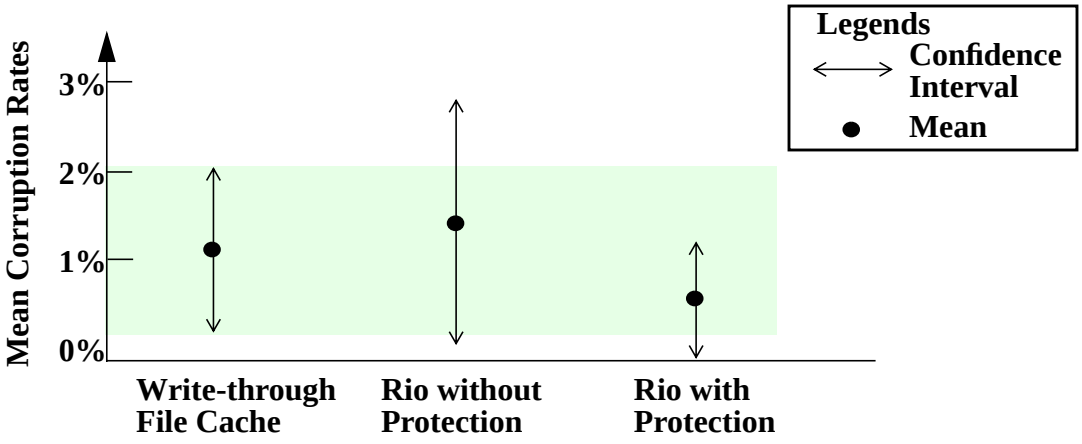

Figure 4.2 Comparing Design Alternatives. This figure plots the 95% confidence interval of mean corruption rate for all three file caches. The mean corruption rates for both Rio designs are in the confidence interval of the write-through file cache, and we conclude with 95% confidence that all three file caches have comparable reliability.

cache designs. We observe significant overlap in confidence intervals between our designs and a write-through file cache, and the mean corruption rates of both Rio file caches are in the confidence interval of the write-through file cache (shaded box in Figure 4.2). We thus conclude that both Rio file cache designs have comparable reliability as a write-through file cache.

We recorded eight crashes where the Rio protection mechanism was invoked to prevent an illegal write to the file cache (six for copy overrun and two for initialization); these indicate cases where the file cache would have been corrupted had the protection mechanism been off. Rio's protection mechanism provides higher reliability than a write-through file cache because it halts the system when it detects an attempted illegal access to the file cache. Write-through file caches, in contrast, may continue to run and later propagate the corrupted memory data to disk.

4.3 Performance

The main benefit of Rio discussed so far is reliability: all writes to the file cache are immediately as permanent and safe as files on disk. In this section, we show that Rio also improves performance by eliminating all reliability-induced writes to disk. The main results were reported in [Chen96a], and are reproduced here.

Table 4.3 compares the performance of Rio with different UNIX file systems, each providing different guarantees on when data is made permanent. UFS is the default Digital UNIX file system. It writes data asynchronously to disk when 64 KB of data has been collected, when the user writes non-sequentially, or when the update daemon flushes dirty file data (once every 30 seconds). UFS writes metadata synchronously to disk to enforce ordering constraints [Ganger94a].

UFS's poor performance is due in large part to its synchronous metadata updates. To eliminate this bottleneck, we enhanced UFS to delay all data and metadata until the next time update runs; this is the optimal "no-order" system in [Ganger94a]. This improves

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

Table 4.3 Performance Comparison. This table compares the running time of Rio with different UNIX file systems, each providing different guarantees on when data is made permanent. cp+rm recursively copies then recursively removes the Digital UNIX source tree (40 MB); Andrew models software development but is dominated by CPU-intensive compilation. Rio achieves performance comparable to a memory-resident file system while providing the reliability of a write-through file system. Rio's protection mechanism adds essentially no overhead, yet enables Rio to surpass the reliability of a write-through file system. Rio is 2-14 times as fast as the default UNIX file system. Delaying metadata writes by 30 seconds enables UFS to match Rio's speed on some workloads, but Rio is still 3 times as fast on cp+rm. Rio is 4-22 times as fast as systems that guarantee data permanence after each file write or close.

performance significantly over the default UFS; however, the optimization risks losing 30 seconds of both data and metadata.

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

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

We run two workloads, cp+rm and Andrew. cp+rm recursively copies then recursively removes the Digital UNIX source tree (40 MB). Andrew models software development but is dominated by CPU-intensive compilation [Howard88]. All results represent an average of at least 5 runs.

The last two rows of Table 4.3 show that Rio’s protection mechanism adds almost no performance penalty, even on very I/O intensive workloads such as cp+rm. Since Section 4.2.2.3 shows that Rio’s protection mechanism enables memory to be even safer than a write-through file system, we recommend that protection be turned on.

Table 4.3 shows that Rio performs as fast as a memory file system and significantly faster than all other file systems. As expected, Rio’s performance improvement is largest over systems that provide similar reliability guarantees—Rio performs 4-22 times as fast as UFS write-through-on-write and write-through-on-close.

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

4.4 Discussion

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

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

Storing permanent data both on disk and in memory makes data more vulnerable to hardware failures than simply storing data on disk. Being able to remove the memory system without losing data can reduce but not eliminate the increased vulnerability. Because software crashes are the dominant cause of failure today [Gray90], we do not consider the

increased vulnerability to hardware failures a serious limitation of Rio. However, if memory or CPU failures becomes the most common cause of system failure, extra redundancy may need to be added to compensate for the larger number of components holding permanent data.

4.5 Summary

We present a systematic and quantitative approach for using software-implemented fault injection to guide the design and implementation of a fault-tolerant system. Our goal was to build a write-back file cache on Digital Alpha workstation that was as reliable as a write-through file cache. We followed an iterative approach to improve the robustness of a write-back file cache in the presence of operating system errors. In each iteration, we measured the reliability of the system, analyzed the fault symptoms that led to data corruption, and applied fault-tolerant mechanisms that address the fault symptoms. Our first design using warm reboot enables memory to achieve reliability close to that of a write-through file system. We observed that the major cause of corruption is due to the copy overrun fault type. In our next design iteration, we protect the buffer data using VM protection mechanism to limit these corruptions. Adding protection makes memory even safer than a write-through file system while adding essentially no overhead. Our statistical analysis indicates that Rio file cache has comparable reliability as a write-through file cache. Eliminating all reliability-induced writes to disk enables Rio to run 4-22 times as fast as systems that give comparable reliability guarantees, 2-14 times as fast as a standard, delayed-write file system, and 1-3 times as fast as an optimized system that risks losing 30 seconds of data and metadata.

Chapter 5

Reliable File Cache in FreeBSD

This chapter describes the implementation of our reliable file cache on an Intel PC running the FreeBSD OS. Section 5.1 describes the challenges posed by the PC platform, and why we cannot use the warm reboot technique we develop for our Digital UNIX Rio. Section 5.2 describes how we develop systematically a new technique, *safe sync*, to safely restore memory content *during* a system crash. Section 5.3 compares the performance of our reliable file cache versus different UNIX file systems (UFS). Section 5.4 discusses the scalability, portability and cost of our design methodology. The material presented in this chapter first appeared in the 1999 Symposium on Fault-Tolerant Computing [Ng99].

5.1 Implementation Platform

Our new platform is a PC running the FreeBSD 2.2.7 OS [McKusick96]. Each PC has an Intel Pentium processor, 128 MB of memory, a 2 GB IDE hard drive, and a Phoenix 4.0 BIOS. The PC platform differs from the Digital Alpha workstation described in Chapter 4 in several ways. First, most PC console (BIOS) firmware clears memory during the initial phase of boot. Second, the PC reset button (if it exists) often erases memory and processor cache. Third, the Intel CPU cache uses a write-back policy and is also reset at the beginning of boot. This prevent us from using warm reboot described in Section 4.2.1 to write file cache data to disk during reboot. The Pentium processor is also a 32-bit processor, compare to the 64-bit Digital Alpha processor. Thus there is a greater chance that an errant pointer will actually point to a valid and dirty buffer page in the 32-bit address space.

5.2 Design Iterations

Our goal is to make Rio (our write-back file cache) as reliable as a write-through file cache. We configure FreeBSD to use a write-through file cache, then measure the corruption rate to be 3.1% using the method described in Chapter 3. That is, 3.1% of the crashes corrupt some data in the file system. Table 5.1 summarizes the corruption rate by fault category of all our designs in this chapter.

Fault Type	Write-Through File Cache	Write-Back File Caches			
		Default FreeBSD Sync	Basic Safe Sync	Enhanced Safe Sync	BIOS Safe Sync
text	3	51	7	5	2
stack	0	3	3	2	0
heap	5	28	8	3	1
initialization	10	45	9	7	4
del. random inst.	4	43	8	2	4
destination reg.	4	42	9	5	2
source reg.	4	43	10	3	1
delete branch	4	51	14	4	5
pointer	3	38	5	4	2
allocation	0	100	5	0	0
copy overrun	4	36	1	3	2
synchronization	0	3	1	0	0
off-by-one	4	59	16	9	3
memory leak	0	0	0	0	0
interface error	1	47	8	3	2
Total	**46 of 1500 (3.1%)**	**589 of 1500 (39.3%)**	**104 of 1500 (6.9%)**	**50 of 1500 (3.3%)**	**28 of 1500 (1.9%)**
95% Confidence Interval	**2.2%-3.9%**	**36.8%-41.8%**	**5.6%-8.2%**	**2.4%-4.3%**	**1.2%-2.6%**

Table 5.1 Comparing Reliability. This table shows how often each type of fault corrupts data for a write-through file cache and the four designs for a reliable write-back file cache. We conduct 1500 crashes for each system (100 for each fault type). The last row shows the 95% confidence interval for the mean corruption rate.

This section describes the four design iterations we went through to arrive at the final system. Each subsection (5.2.1-5.2.4) describes the write-back file cache used in a design iteration, presents results from the fault-injection tests on that design, then analyzes the results to select techniques to fix the revealed fault categories. Table 5.2 summarizes the four design iterations.

In all our designs, we modify the FreeBSD file cache in two ways to be a pure write-back file cache. First, FreeBSD normally writes dirty file data to disk every 30 seconds or when a full file block is written. We disable this reliability-induced write-back, so the system writes data back to disk only when dirty blocks are replaced in the file cache. Second, FreeBSD normally limits the amount of dirty file cache data to 10% of available system memory. We increase this limit by allowing dirty file data to migrate from the file cache to the virtual memory system, as is done in memory-mapped file systems [Bensoussan72].

5.2.1 Design Iteration 1: Default FreeBSD Sync

5.2.1.1 Design

We start the design process with the default sync used in FreeBSD. Sync refers to the routine that writes dirty file-cache data to disk during a crash. FreeBSD's default sync routine examines all blocks in the file cache and writes dirty blocks to disk using normal file system routines.

Section	Design Iteration	Fault-Tolerant Mechanisms
5.2.1	default FreeBSD sync	default sync used in FreeBSD during crashes
5.2.2	basic safe sync	add VM protection, reset key, registry, safe sync
5.2.3	enhanced safe sync	fix VM protection bugs; add watchdog timer, private stack; disable debugging prints; map kernel code read-only; initialize segment registers
5.2.4	BIOS safe sync	physical addressing, BIOS disk I/O

Table 5.2 Design Iterations for FreeBSD Rio File Cache.

5.2.1.2 Results and Analysis

Unfortunately, the default FreeBSD sync is not very robust during operating system crashes. As shown in Table 5.1, 39% of crashes corrupted some file system data when using the default FreeBSD sync. This corruption rate is 13 times as high as that of a write-through file cache. We also observe from Table 5.1 that there is no overlap in confidence interval between the our first design and the write-through file cache. This can be clearly seen in Figure 5.1, which depicts the mean corruption rate and 95% confidence intervals for the write-through file cache and all our designs. Thus, we conclude with 95% confidence that our basic safe sync design is less reliable than the write-through file cache.

We next examine the corrupted runs in greater detail, focusing on where the faults are injected into the system and how the system crashes. We determine how the system

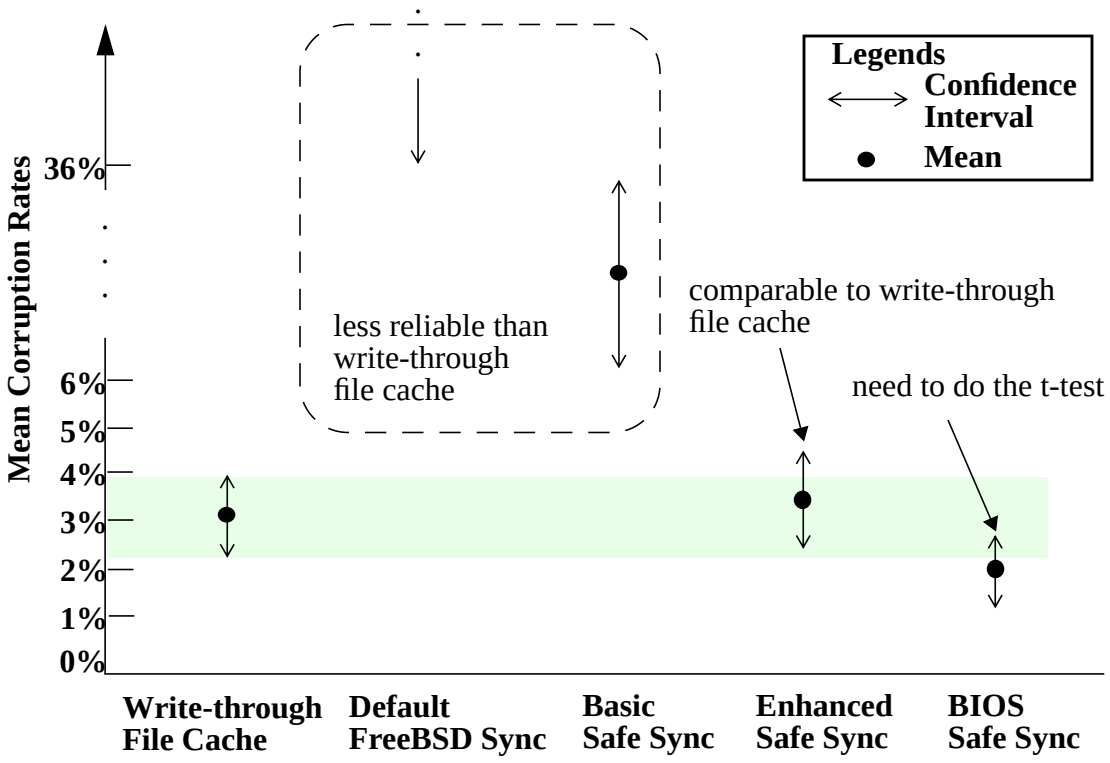

Figure 5.1 Comparing Design Alternatives. This figure plots the 95% confidence interval of mean corruption rate for the write-through file cache and all our designs. We use the approximate visual test [Jain91] to compare our designs.

crashes by looking at the crash messages and tracing fault propagation with the aid of the FreeBSD kernel debugger. Our fault-injection tool helps by printing the kernel routine name and location of corrupted code. We use this information to divide the fault symptoms into categories:

- *Hang before sync*: Most data corruptions occur because the system hangs and fails to call the sync routine. Many workstations have a reset key that allows the user to drop the system into the console prompt. The user can then issue a sync command directly or initiate recovery using a user-written routine. But most PCs do not have such a feature, and those that are equipped with a reset switch typically erase memory (including dirty file cache data).
- *Page fault during sync*: Sync often fails because it encounters a page fault while trying to write dirty file cache data to disk. The page fault occurs when the operating system accesses unmapped data or mapped data with the wrong permission settings. It can also happen when the code is invalid or unmapped. The FreeBSD sync routine uses many different kernel routines and data structures (e.g. mounted file system list, vnode data structures, buffer hash list), so this fault is quite common.
- *Buffer locked during sync*: FreeBSD's sync routine obeys the locking protocol used during normal operation. It does not write to disk any file cache blocks that are locked, so data in these blocks are lost.
- *Double fault*: The Pentium processor calls a double-fault handler if it detects an exception while servicing a prior exception [Int97b]. The processor will reset and abandon sync if another exception occurs when the double fault handler is being serviced.
- *File system errors*: Our tool may inject faults into any part of the kernel. Faults that are

injected into file system routines often cause data corruption. For example, the file system's write routine might be changed to write to the wrong part of the file. We do not attempt to fix this fault symptom because the write-through file cache is also susceptible to these errors.

- *Device Timeout*: Sync sometimes fails because it experiences repeated device timeouts when writing to the hard drive.
- *Unknown*: A few data corruptions are due to unknown causes. For these corruptions, the sync routine appears to be successful, and the injected bugs appear to be benign. We do not attempt to overcome this problem because of the lack of information and the low frequency of this fault symptom.

Table 5.3 summarizes the categories of fault symptoms and some potential solutions we develop (discussed in the next section) to reduce the vulnerability of the system to that fault symptom.

Fault Symptom	**# of Corruptions**	**Solutions Used in the Next Design (Section 5.2.2.1)**
hang before sync	268 (17.9%)	software reset key
page fault during sync	163 (10.9%)	registry, safe sync
buffer locked during sync	89 (5.9%)	registry, safe sync
double fault	39 (2.6%)	disable interrupt in safe sync
file system error	25 (1.7%)	
device timeout	3 (0.2%)	
unknown	2 (0.1%)	
Total	**589 of 1500 (39.3%)**	

Table 5.3 Categories of Fault Symptoms for Default FreeBSD Sync.

5.2.2 Design Iteration 2: Basic Safe Sync

5.2.2.1 Design

A write-back file cache must perform two steps to write dirty data back to disk during a crash. First, the system must transfer control to the sync routine. Second, the sync routine must write dirty file data successfully to disk. Most of the corruptions experienced using the default FreeBSD sync fail one of these two steps. *Hang before sync* fails to transfer control to the sync routine during a crash. Most of the other fault symptoms transfer control to sync but experience an error during sync.

We address errors in these two steps separately. First, we must make it more likely that the system will successfully transfer control to the sync routine during a crash. To fix *hang before sync*, we use a software reset key that calls sync when pressed. We modify the low-level keyboard interrupt handler of FreeBSD to call sync whenever it detects a certain key sequence (e.g. control-alt-del). This addresses the dominant fault symptom in Table 5.3.

Second, we must make it more likely that the sync routine, once called, will write dirty file data successfully to disk. Default FreeBSD sync fails this step because it depends on many parts of the kernel. The default FreeBSD sync calls many routines and uses many different data structures. Sync fails if *any* of the routines or data structures are corrupted. To make sync more robust, we must minimize the scope of the system that it depends on.

To minimize data dependencies, we implement informational redundancy [Johnson89] by creating a new data structure called the *registry.* The registry contains all information needed to find, identify, and write all file cache blocks. For each block in the file cache, the registry contains the physical memory address, file ID (device number and inode number), file offset, and size. The registry allows sync to operate without using previously needed kernel data structures, such as file system and disk allocation data. The registry is wired in

memory to reduce the likelihood of page faults during sync. Registry information changes relatively infrequently during normal operation, so the overhead of maintaining it is low.

We replace FreeBSD's default sync routine with a new routine (called *safe sync*) that uses the registry when writing data to disk. Safe sync examines all valid entries in the registry and writes dirty file cache data directly to disk. By using information in the registry, safe sync does not depend on normal file system routines or data structures. Safe sync also takes additional precautions to increase its chances of success. First, safe sync operates below the locking protocol to avoid being stymied by a locked buffer. Second, safe sync disables interrupts to reduce the likelihood of double faults while writing to disk.

In addition to adding the registry and using a new sync routine, we also use the virtual memory system to protect file cache data from wild stores [Chen96a]. We turn off the write-permission bits in the page table for file cache pages, causing the system to generate protection violations for unauthorized stores. File cache procedures must enable the write-permission bit in the page table before writing a page and disable writes afterwards.

5.2.2.2 Results and Analysis

Fault injection tests on the new design show substantial improvement over the default FreeBSD sync. Table 5.1 shows that the new design has a corruption rate of 6.9%, which is six times better than the default FreeBSD sync. However, it still has twice as many corruptions as a write-through file cache. We observe from Figure 5.1 that there is no overlap in confidence interval between the our first design and the write-through file cache. We thus conclude with 95% confidence that our new design is less reliable than a write-through file cache.

Table 5.4 breaks down the fault symptoms for our current design. The fault symptoms are very similar to those in Table 5.3, but the corruption rates are reduced significantly due to the fault-tolerant measures introduced in Section 5.2.2.1. We analyze the fault symptoms from this design to see how we can make safe sync more robust in the next iteration:

- *Hang before sync*: Table 5.4 shows that the reset button reduces the corruption rates substantially from the default FreeBSD sync (from 17.9% to 3.0%). We examine the remaining cases and the keyboard interrupt handler to determine what causes the reset key to fail. The dominant reason is that FreeBSD sometimes masks keyboard interrupts. If the system hangs while keyboard interrupts are masked, the reset key will not transfer control to safe sync. To fix this, we add a watchdog timer to the system timer interrupt handler [Johnson89]. The system timer interrupt handler watches for pending keyboard interrupts and calls safe sync if the keyboard interrupt does not get serviced for a long time. For five of the corruptions, the fault was injected into the terminal output routine, and safe sync failed when it tried to print some debugging information. We fix this fault by disabling debugging print statements during safe sync.

Fault Symptom	**# of Corruptions**	**Solutions Used in the Next Design (Section 5.2.3.1)**
hang before sync	45 (3.0%)	watchdog timer, disable print
file system error	24 (1.6%)	
page fault during sync	18 (1.2%)	read-only text, private stack, restore segment registers
data corruption	11 (0.7%)	fix VM protection
double fault	4 (0.3%)	private stack
device timeout	2 (0.1%)	
Total	**104 of 1500 (6.9%)**	

Table 5.4 Categories of Fault Symptoms for Basic Safe Sync.

- *File system error*: Again, we do not attempt to fix this fault symptom because the write-through file cache is also susceptible to these errors. Note that the corruption rate for file system errors is similar between Table 5.3 and Table 5.4. The slight differences are due to the non-determinism inherent to testing a complex, timing-dependent system.
- *Page fault during sync*: This fault symptom occurs for a variety of reasons, and we develop a variety of solutions to fix it. For example, some faults corrupt the Intel segment registers that are used by some instructions in safe sync. To fix this error, we re-initialize the segment registers to their correct value at the beginning of safe sync. Other faults cause wild stores to write over kernel code. To fix this error, we map the kernel code as read-only (of course, our fault injector can still modify kernel code).
- *Data corruption*: Some faults corrupted data in the file cache before crashing the system. For these runs, safe sync completed successfully but wrote out the corrupted data. While investigating the source of this corruption, we uncovered a bug in FreeBSD's protection code that sometimes allowed wild stores to overwrite the file cache and kernel code.
- *Double fault*: This bug occurs when part of the stack segment is unmapped by the injected fault but the TLB is not invalidated. The system will continue to function until the stack pointer advances beyond the valid page in the TLB, and encounter multiple page faults when it tries to fault in subsequent pages from the bogus stack. To fix this, we pre-allocate a stack for safe sync during bootup and wire it in memory. Safe sync's first action is to switch to this private stack.

5.2.3 Design Iteration 3: Enhanced Safe Sync

5.2.3.1 Design

Our next design improves on the basic safe sync design from the last iteration using the fixes suggested in Section 5.2.2.2. First, we add a watchdog timer to call safe sync if the system hangs with keyboard interrupts disabled. Second, we disable print statements during safe sync to remove dependencies on the print routines. Third, we re-initialize the segment registers to their proper value. Fourth, we map the kernel code as read-only and fix a bug in FreeBSD's protection code. Finally, we switch to a pre-allocated, wired stack at the beginning of safe sync to remove dependencies on the system stack.

5.2.3.2 Results and Analysis

We conduct fault injection tests on the new design and find that it has a corruption rate of 3.3%, versus 6.9% for the basic safe sync design of iteration 2. Enhanced safe sync is nearly as reliable as a write-through file cache. We observe from Figure 5.1 that there is significant overlap in confidence interval between the our new design and the write-through file cache, and the mean corruption rate of our new design falls in the confidence interval of the mean corruption rate of the write-through file cache. We thus conclude with 95% confidence that our new design has comparable reliability as a write-through file cache.

We choose to improve on our current design until we can conclude with a high degree of confidence that our design is more reliable than the write-through file cache. Table 5.5 breaks down the fault symptoms of our current design. There are two basic dependencies remaining in our system. First, all kernel code, including safe sync, runs in virtual-addressing mode with paging enabled [Int97d], which uses virtual addresses to access

code and data. Because safe sync accesses virtual addresses, it depends on the FreeBSD virtual memory code and data (such as the doubly linked address map entries [Rashid88]). To fix this dependency, we must configure the processor to use physical addresses during safe sync.

Second, safe sync uses the low-level kernel device drivers to write data to disk. The FreeBSD disk device drivers are quite complex, and there is no simple disk device driver routine to initialize the device driver state or reset the hard drive and disk controller card. Our safe sync code can thus hang or timeout whenever it access the disk. To remove this dependency, we must bypass the complex device driver for a simpler disk interface.

5.2.4 Design Iteration 4: BIOS Safe Sync

5.2.4.1 Design

In our final design, we want to remove dependencies on the virtual memory system and device drivers. We remove dependencies on the virtual memory system by switching the processor to use physical addresses [Int97d]. We remove dependencies on the kernel device drivers by using the BIOS interface to the disk [Gilluwe97]. BIOS (Basic Input/Output Service) routines are implemented in the firmware of the I/O controller. Both

Fault Symptom	# of Corruptions	Solutions Used in the Next Design (Section 5.2.4.1)
hang before sync	21 (1.4%)	BIOS I/O, real-mode addressing
file system error	20 (1.3%)	
page fault during sync	4 (0.3%)	real-mode addressing
device timeout	3 (0.2%)	BIOS I/O
data corruption	2 (0.1%)	
Total	**50 of 1500 (3.3%)**	

Table 5.5 Categories of Fault Symptoms for Enhanced Safe Sync.

physical addressing and BIOS routines have limited features and are used normally to load the operating system from disk during system boot. Modern operating systems like FreeBSD use virtual addressing and replace the BIOS with their own device drivers.

Our final design replaces the safe sync code used in design iteration 3. The new safe sync procedure is summarized below (the full source code is available at http://www.eecs.umich.edu/Rio):

- *Part 1: Initial setup*: we followed the instructions outlined in Section 8.8.1 of [Int97d], which includes setting up a linearly mapped segments for data and code, setting the global (code/data) and interrupt descriptor table registers for real-mode operation, and making a long jump to the real-mode switch code.
- *Part 2: Mode switching*: the real-mode switch code disables paging, loads the segment registers with real-mode segments, and clears the paging enable bit before making a jump to the real-mode safe sync code. This jump brings the processor to real-mode operation.
- *Part 3: Real-mode setup*: BIOS safe sync begins by initializing the remaining segment registers, setting the interrupt controllers to real-mode operation [Int97a], and initializing the video console and disk controller using the BIOS interface [Gilluwe97]. The rest of BIOS safe sync is fairly straightforward and is generated from the C version of enhanced safe sync. We modify the resulting assembly code by adding address/data overrides [Int97d] and using a large data segment (i.e. big real-mode [Shanley96]) to access data beyond the first 1 MB of memory. During sync, we copy the file cache data into the lower 1 MB of memory because the BIOS disk interface uses 16-bit segment addressing.

Part 1 of BIOS safe sync is a C function and can be invoked directly by the FreeBSD kernel. The rest of BIOS safe sync code is written in assembly and is not accessible to the FreeBSD kernel, because it resides in unmapped physical memory pages that are hidden from the FreeBSD page allocator.

5.2.4.2 Results and Analysis

We conduct fault injection tests on our BIOS safe sync. Table 5.6 breaks down the fault symptoms for our latest design. The mean corruption rate is 1.9%, which is 40% lower than the mean corruption rate of the write-through file cache. We also observe from Figure 5.1 that the confidence interval of our new design overlaps slightly with that of the write-through file cache, and Rio's mean corruption rate is not in the confidence interval of the write-through file cache. We use a more elaborate *t*-test statistical procedure explained in Section 13.4.2 of [Jain91] to compare the two designs. We compute the mean difference in corruption rates between the two designs, and the standard deviation of mean difference. This allows us to derive the 95% confidence interval for the difference in mean corruption rates, which is (1.0%, 2.3%). Because the resulting confidence interval does not include zero, we conclude with 95% confidence that our BIOS safe sync design is more reliable than the write-through file cache.

Fault Symptom	# of Corruptions	Possible Solutions
file system errors	17 (1.1%)	
hang before sync	5 (0.3%)	hardware reset/warm reboot
data corruption	4 (0.3%)	
device timeout	2 (0.1%)	hardware reset/warm reboot
Total	**28 of 1500 (1.9%)**	

Table 5.6 Categories of Fault Symptoms for BIOS Safe Sync.

Rio is able to achieve higher reliability than a write-through file cache because we have less corruptions due to the fault symptoms file system errors and VM errors. Figure 5.2 breaks down the fault symptoms for our Rio file cache with BIOS safe sync and a write-through file cache. A write-through file cache writes all data synchronously to hard disk, and performs several order of magnitude more disk writes than our Rio file cache. It also has a larger code path compare to Rio as it needs to write data to stable storage. In contrast, Rio executes less code when writing data to persistent storage in main memory. Thus, there is a higher probability that a write-through file cache will write corrupted data to disk whenever the file system code is corrupted.

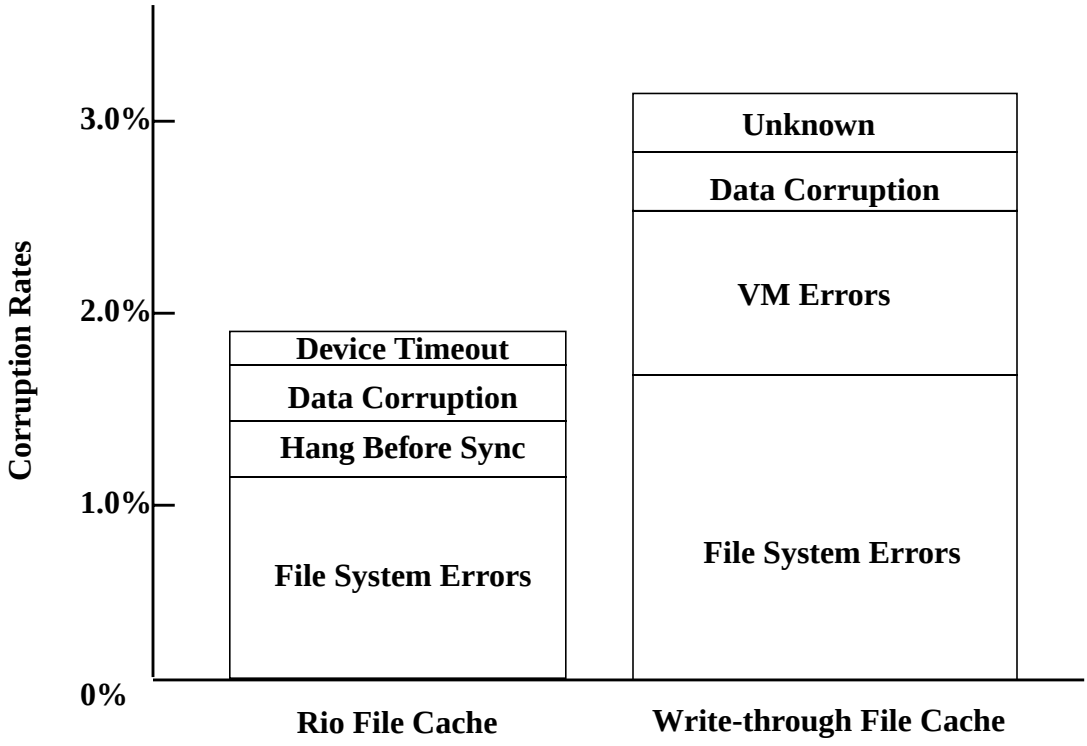

Figure 5.2 Why is Rio with BIOS Safe Sync More Reliable? We breakdown the fault symptoms for both Rio file cache with BIOS safe sync and write-through file cache. The y-axis shows the mean corruption rates.

The write-through file cache also suffers from corruptions due to VM errors. This occurs when the injected bug corrupts the address mappings for the buffer page, and causes the wrong data to be written to disk. Rio is not vulnerable to this fault type because we use the information in the registry to restore file cache data. The registry contains the physical address of the buffer page. It also has a restricted interface. We perform consistency checks before we update the registry content.

We observe from Figure 5.2 that it may be possible to improve BIOS safe sync by adding a hardware reset key and modifying the PC firmware and motherboard to not initialize memory on reset/reboot. This would allow the system to do a complete reset, then to perform a warm reboot as was done in [Chen96a]. Doing so should fix the remaining hangs before sync and device timeouts, but it would incur significant system cost.

5.3 Performance

The main benefit of Rio discussed so far is reliability: all writes to the file cache are immediately as permanent and safe as files on disk. In this section, we show that Rio also improves performance by eliminating all reliability-induced writes to disk.

Table 5.7 compares the performance of our Rio file cache with different types of file systems, each providing different guarantees on when data is made permanent. The Memory File System, which is completely memory-resident and does no disk I/O, is shown to illustrate optimal performance [McKusick90]. UFS (UNIX File System) is the default FreeBSD UNIX file system. It writes data asynchronously to disk when 64 KB of data has been collected, when the user writes non-sequentially, or when the sync daemon flushes dirty file data (once every 30 seconds). UFS writes metadata synchronously to disk to enforce ordering constraints [Ganger94a].

UFS's poor performance is due in large part to its synchronous metadata updates. To eliminate this bottleneck, we measure the behavior of a file system that write both metadata and data asynchronously to disk. We delay all data and metadata writes until the next time update runs. This is the optimal "no-order" system in [Ganger94a] and is faster than soft update [Mckusick99] file systems as it does not need to maintain dependencies between updates. Note that we did not use the async mount option in FreeBSD as the FreeBSD async file system implementation still does a significant number of synchronous metadata disk writes. This improves performance significantly over the default UFS; however, the optimization risks losing 30 seconds of both data and metadata. We also measure the behavior of a write-through file cache which writes all data synchronously to disk. This achieves the same reliability as Rio but at significant performance penalty.

File System	Data Permanent	cp+rm	Andrew
Memory File System	never	2.80 seconds	1.20 seconds
Rio file cache	**synchronous**	**3.20 seconds**	**1.25 seconds**
UFS with delayed data and metadata	after 0-30 seconds, asynchronous	5.95 seconds	1.30 seconds
UFS	data asynchronous metadata synchronous	119.85 seconds	5.15 seconds
write-through file cache	synchronous	175.11 seconds	14.60 seconds

Table 5.7 Performance Comparison. This table compares the running time of our reliable write-back file cache (Rio) with different UNIX file systems, each providing different guarantees on when data is made permanent. cp+rm recursively copies then recursively removes the FreeBSD source tree (32 MB); Andrew models software development but is dominated by CPU-intensive compilation. The results are based on an average of 40 runs (except for write-through file cache and UFS, which is based on an average of 10 runs). All performance measurements were made on a PC with a 400 MHz Pentium-II processor, 128 MB of 100 MHz SDRAM, and a IBM DCAS-34330W SCSI disk (with the disk write-cache enabled) and FreeBSD OS.

We run two workloads, cp+rm and Andrew. cp+rm recursively copies then removes the FreeBSD source tree (32 MB). Andrew models a software development workload. All results represent an average of 20 runs.

Table 5.7 shows that our Rio file cache is an order of magnitude faster (12-55 times) than a write-through file cache. It is also significantly faster (4-37 times) than the standard UNIX file system. Rio's performance approaches that of an optimal memory-based file system. It achieves excellent performance without sacrificing reliability. Rio is roughly equivalent in reliability to a write-through file cache. Both Rio and a write-through file cache are more reliable than standard UFS. UFS loses up to 30 seconds of data on a crash, while Rio and a write-through file cache typically lose no data on a crash.

Rio is significantly faster than a write-through file cache because it eliminates all synchronous writes to disk. Table 5.8 breaks down the number of synchronous and asynchronous disk writes for each type of file systems. We observe that file systems with no synchronous writes, such as Rio, perform significantly better than the other file systems

File System	**cp+rm**			**Andrew**		
	time (sec)	**sync**	**async**	**time (sec)**	**sync**	**async**
Memory File System	2.80	0	0	1.20	0	0
UFS with delayed data and metadata (no sync daemon)	3.05	0	0	1.20	0	0
Rio file cache	**3.20**	**0**	**0**	**1.25**	**0**	**0**
UFS with delayed data and metadata	5.95	0	354	1.30	0	5
UFS	119.85	5725	9320	5.15	4725	4031
write-through file cache	175.11	19673	8310	14.60	24150	4366

Table 5.8 Why is Rio Faster? This table breaks down the type of disk writes, synchronous (sync) or asynchronous (async), for Rio and different file systems.

that have synchronous writes. Rio also does not need to sync periodically its file cache data to disk. All the UNIX file systems run a sync process that periodically (e.g. every 30 seconds) writes dirty file cache data asynchronously to disk. This will increase the number of asynchronous writes. We disable the sync daemon on a UFS with delay data and meta-data. This removes all asynchronous writes and allow the file system's performance to approach that of MFS.

5.4 Discussion

We have designed two reliable file caches on two separate platforms, Digital Alpha workstation and Intel PC, using different design techniques (e.g. warm reboot on Alpha and safe sync on Intel PC). Both platforms also require more than one design iterations. This allows us to understand the scalability, portability, and cost of our design approach. We would like to address these issues in this section.

- *Scalability*: Our target system is fairly representative of a medium-size software development project. The relevant operating system code (file system, VM, interrupt, etc.) spans approximately 40 files and 20,000 lines of code. We added 3 files and 2000 lines of code. The development effort took one man-year. This includes the substantial time it took to understand FreeBSD and the Intel PC architecture (microarchitecture, assembly language, system BIOS). Our experimental setup was fully automated and we had sufficient machines to run the experiments in parallel, so most of our time was devoted to uncovering fault symptoms and debugging code. The analysis process was largely manual, though we wrote several tools to expedite this process. Our design approach is applicable to many software development projects as long as there are enough resources to perform the fault injection experiments, and sufficient expertise to diagnose the faults

and implement the fault-tolerant mechanisms. The development effort for the Digital UNIX Rio file cache is comparable in magnitude.

- *Portability*: We demonstrated our design methodology on FreeBSD running on Intel PCs. We have also tried this approach on a limited scale when implementing a reliable write-back file cache on Digital Alpha workstations (see Chapter 4) and the Postgres database (see Chapter 6). We are confident that our approach is portable to other systems.
- *Cost*: Our FreeBSD Rio design took four iterations, requiring 8 machine-months of testing. Our Digital UNIX Rio design took two iterations and 6 machines-months of testing. In both studies, we tackled the dominant fault symptoms in the first iteration, with diminishing returns on successive iterations. Using software fault injection provided quantitative data on when our system reached our reliability goal. For example, we could have stopped after enhanced safe sync, because its corruption rate was statistically indistinguishable from a write-through file cache. Choosing a reliability goal is a tradeoff between design cost and application requirements.

5.5 Summary

We have implemented the Rio file cache on a more challenging platform, Intel PC running the FreeBSD OS. We followed an iterative approach described in Chapter 3 to improve the robustness of a write-back file cache in the presence of operating system errors. In each iteration, we measured the reliability of the system, analyzed the fault symptoms that led to data corruption, and applied fault-tolerant mechanisms that address the fault symptoms. The result of several iterations was a design that improved reliability by a factor of 21. The resulting write-back file cache is both more reliable (1.9% vs. 3.1%

corruption rate) and an order of magnitude faster (12-55 times) than a write-through file cache for workload that fits in main memory.

Chapter 6

Application-Level Reliable Memory

This chapter describes how we implement reliable memory in an application, using the warm reboot and VM protection techniques described in the previous chapter. We choose database system as it is representative of large, I/O bound applications that can benefit significantly from Rio. Section 6.1 briefly describes the database management software (Postgres) used in our experiments. Section 6.2 examines three different approaches of using reliable memory in database systems. Section 6.3 analyzes the reliability of the our prototypes, and demonstrates that Rio can be applied to database systems without diminishing the application's reliability. We conclude by discussing some limitations of our work. The material presented in this chapter first appeared in the 1997 International Conference on Very Large Data Bases [Ng97].

6.1 Postgres Database Management System

We use the Postgres95 database management system developed at U.C. Berkeley as the database in our experiments [Stonebraker87]. Postgres has a few unique features which are relevant to our work; however our results should apply to more conventional databases as well.

One novel aspect of Postgres is that it *appends* new data rather than overwriting the old version of the data. In contrast, conventional databases with write-ahead logs log undo/redo records, then later write new data in-place over the old version of the data. Post-

gres' scheme forces new data to disk at commit, whereas a conventional scheme forces only the log at commit (a no-force policy for the actual data). A force-at-commit policy decreases the amount of time database buffers are vulnerable to database crashes (Section 6.2.1).

As with nearly all database systems, Postgres keeps a database buffer cache in main memory to store frequently used data. Transactions modify the buffer cache data, then force the modified data to disk on commit. Because Postgres appends new data rather than overwriting it, a steal policy may be used without an explicit undo log. If a transaction aborts, the old copy of the data can be recovered from disk. Our second software design (Section 6.2.3) makes the database buffer cache persistent and hence does not force data to disk at commit.

6.2 Software Designs for Integrating Reliable Memory

In this section, we describe three ways databases can include reliable memory and the implication of each design on reliability and performance.

The goal of this chapter is to explore how to use the Rio file cache to provide reliable memory for databases. Database systems traditionally encounter two problems in trying to use buffer caches managed by the operating system (the file cache) [Stonebraker81].

First, buffer caches managed by the file system do not easily allow the database to order updates to disk. These writes to disk need to be done in order to obey the constraints imposed by write-ahead logging [Gray78]. Because of this, databases must use fsync to force data to disk or use direct I/O, bypassing the file cache entirely. The Rio file cache solves this problem completely, because data is persistent as soon as it enters the file

cache. Thus, databases control the order of persistence by controlling the order that I/O is done to the file cache; no fsync is needed.

Second, databases manage memory better than a generic file system can, because databases know more about their access patterns than the file system. Our second software design addresses this problem by mapping the Rio file cache into the database address space. This exposes reliable memory to database crashes. Our third design incorporates virtual memory protection to minimize this vulnerability.

6.2.1 I/O Interface to Reliable Memory (Non-Persistent Database Buffer Cache)

Our baseline design minimizes the changes needed to the database system by hiding the reliable memory under the file system interface (see Figure 6.1). The Rio file cache is

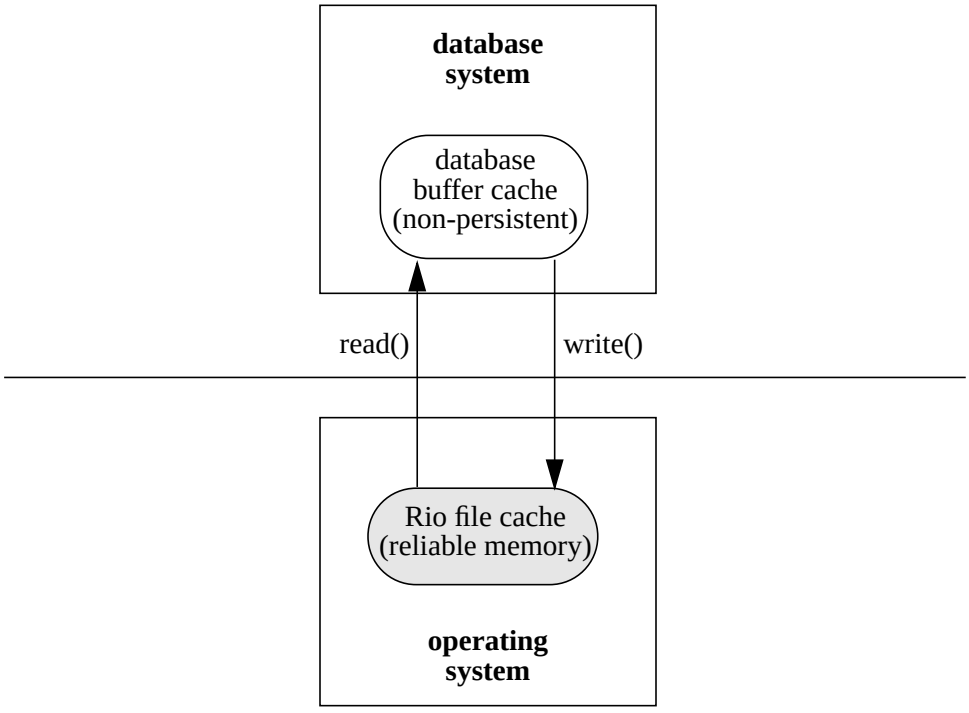

Figure 6.1 I/O Interface to Reliable Memory. This design hides the reliable memory under the file system interface. The database uses read() and write() system calls to write data to the reliable memory. This design uses a traditional, non-persistent database buffer cache and thus minimizes changes required to the database. Because the interface to stable storage has not changed, this design is as safe as a standard database system from database crashes.

used automatically when accessing the file system, so the database need only write persistent data to the file system instead of to the raw disk (or via direct I/O). No fsync is needed, because all file system writes to Rio are persistent immediately as soon as the data enters the file cache. In fact, Rio implements fsyncs as null operations. This removes all synchronous writes from the critical path of any transaction. This design requires no changes to the database; it need only run on top of the Rio file system.

Because the interface to stable storage has not changed, this design is as safe as a standard database from database crashes. Recall that the Rio file cache is responsible for protecting and restoring the file system data if the operating system should crash. This transparency and reliability incurs some costs, however.

Using an I/O interface to reliable memory partitions main memory between the Rio file cache and the database buffer cache. The operating system would like the reliable memory area (the Rio file cache) to be large so it can schedule disk writes flexibly and allow the largest number of writes to die in the file cache. But larger reliable memory areas reduce the size of memory available to the database. This partitioning creates two copies of data in memory (double buffering), one in the Rio file cache and one in the database buffer cache. Not only does this waste memory capacity, it also causes extra memory-to-memory copies.

One possible solution to these problems is to eliminate the database buffer cache and have the database use the file cache to store frequently used data. This is likely to make memory less effective at buffering database data, however, because a database can manage its buffer cache more effectively than file systems can (databases have more information on usage patterns). Researchers have proposed various ways for applications to control

memory [Harty92, Patterson95, Bershad95, Seltzer96], and eventually this may enable the file cache to be as effective as a database buffer cache. But for the time being at least, the best performance will be achieved by databases that wire down memory in their buffer caches and control it completely.

6.2.2 Memory Interface to Reliable Memory (Persistent Database Buffer Cache)

Our second design maps the Rio file cache directly into the database system's address space using the mmap system call (see Figure 6.2). The database system allocates the database buffer cache from this area and locks these pages in memory to avoid paging.

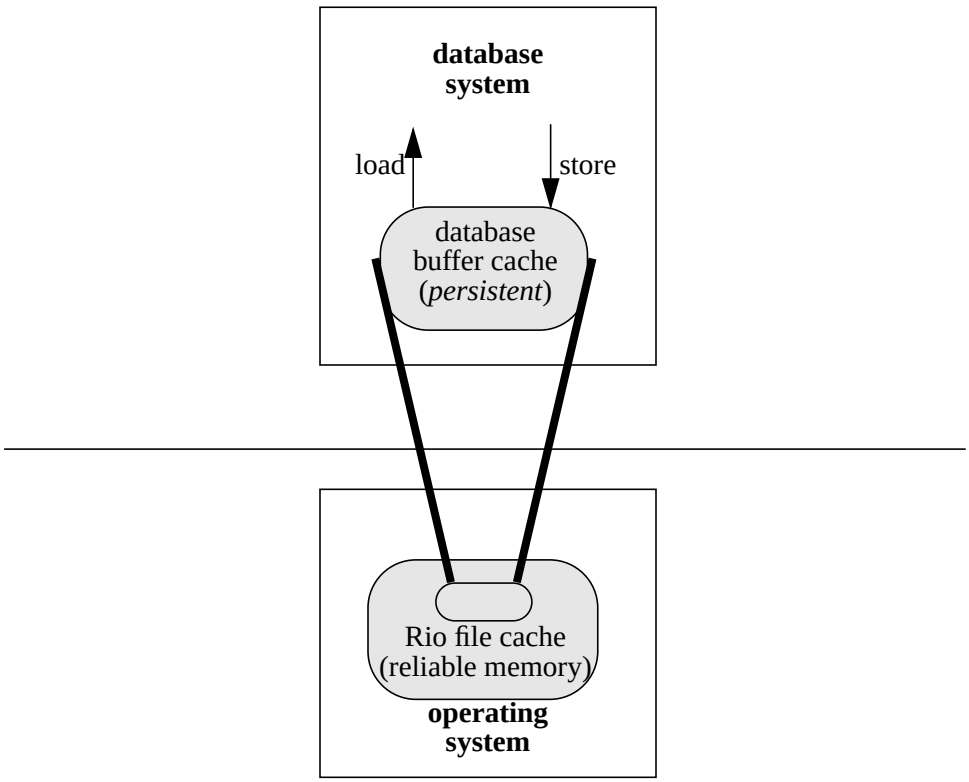

Figure 6.2 Memory Interface to Reliable Memory. This design *maps* the Rio file cache directly into the database system's address space using the mmap system call. The database system can allocate its buffer cache from this region to make a persistent buffer cache. Access to stable storage (the persistent database buffer cache) takes place using load/store instructions to memory. This design eliminates the double buffering problem but exposes stable storage to software errors more than the design using an I/O interface to reliable memory.

This design allows the database to manipulate reliable memory directly using ordinary load/store instructions.

Using a memory interface to reliable memory has several advantages over the first design. First, management of this area of memory is completely under database control; the database system treats this area of memory exactly like a normal database buffer cache (that happens to be persistent). Hence no special support is required from the operating system to allow the database to help manage replacement and prefetching policies.

Second, this design eliminates double buffering and extra memory-to-memory copies. The database simply manipulates data in its buffer cache and these changes are automatically and instantly permanent. Hence this design performs better than the non-persistent buffer cache.

Making the database buffer cache persistent leads to a few changes to the database. These changes are essentially the same as those needed by a database using the steal policy [Haerder83]. The steal policy allows dirty buffers to be written back to disk (that is, made persistent) at any time. In particular, buffers may be made persistent before the transaction commits. This policy requires an undo log so the original values may be restored if the transaction aborts. Persistent database buffer caches require an undo log for the same reason, because *all* updates to the buffer cache are instantly persistent, just as if they had been stolen immediately.

Other designs are possible that map the Rio file cache into the database address space. For example, the database log could be stored in reliable memory. Or an entire database could be mmapped, and the database could trust the virtual memory and file system of the operating system to page in and out appropriate pages. This latter approach may be appro-

priate for situations where the database program is one of several concurrent jobs (perhaps a machine running a client database program). In general, however, we believe that databases prefer to manage their own memory. Because of this, we mmap only the database buffer cache, and we lock these pages in memory to prevent paging.

The main disadvantage to using a memory interface to reliable memory is a slightly increased vulnerability to software errors. The interface to stable storage with this design is now much simpler: load/store instructions instead of read/write system calls. Hence it is easier for a software bug in the database to accidentally overwrite persistent data [Rahm92, Sullivan91a]. This section discusses the increased vulnerability conceptually; Section 6.3 compares quantitatively the chances of data corruption among the different designs

Consider the possible states of a database buffer (see Figure 6.3) during a transaction. For both persistent and non-persistent database buffer cache designs, a transaction begins with a clean buffer. This buffer will become dirty and uncommitted when the transaction modifies the buffer. Dirty means the memory version of the buffer is different than the disk version. For the non-persistent database buffer cache design, the dirty and uncommitted buffer will be written to disk when the transaction commits, and become clean buffer again. However, for the persistent database buffer cache design, the dirty buffers are not written out to disk when the transaction commits. Instead, these buffers are marked as committed data. It will be written out to disk when the buffer is replaced due to capacity overflow.

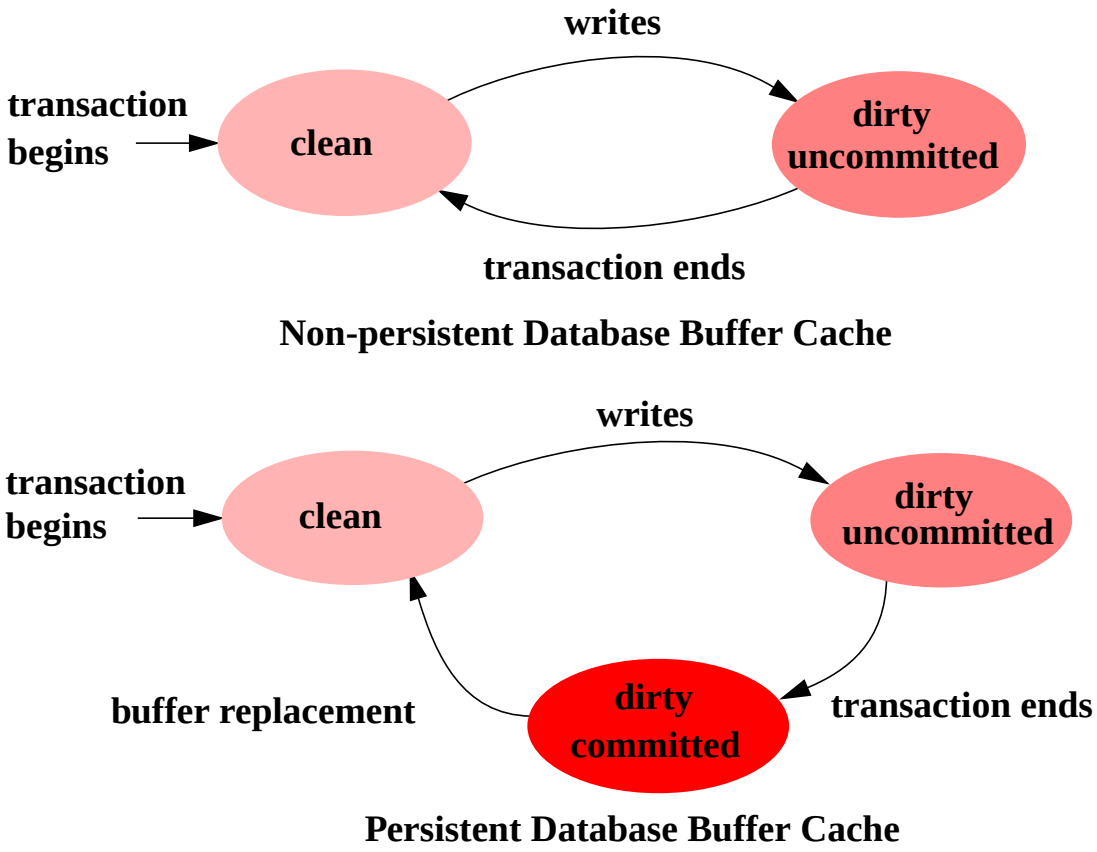

Figure 6.3 Database buffer cache state.

In our modification of Postgres95, we keep commit and dirty flags for each database buffer. After a database crash, we restore to disk only those pages that are marked as committed and dirty. Dirty pages that are not yet committed are restored to their before-image using the undo log. The following compares the vulnerabilities of different buffer states for persistent and non-persistent database buffer caches.

- **clean**: This state (see Figure 6.3) occurs when a piece of data is read from disk or when a dirty buffer is written back to disk. Buffers in this state are safe from single errors for both designs. To corrupt stable storage with a non-persistent buffer cache, the system would need to corrupt the buffer and later force it to disk (a double error). To corrupt stable storage with a persistent buffer cache, the system would need to corrupt the

buffer and mark it as dirty (a double error). With either design, errant stores to buffers in this state may lead to corruption if other transactions read the corrupted data.

- **dirty, uncommitted**: This state occurs when a buffer has been modified by a transaction that is still in progress. Buffers in this state are equally vulnerable for both designs. In either design, stable storage will be corrupted if and only if the buffer is corrupted *and* the transaction commits.

- **dirty, committed**: This state indicates the buffer has been changed by a transaction and that the transaction has committed, but that the data has not yet been written to disk.

  Dirty, committed buffers can exist in a persistent database buffer cache, because data is not written to disk until the buffer is replaced. Buffers in this state are vulnerable to software corruption; any wild store by another transaction can corrupt these buffers, and any change is instantly made persistent.

  With non-persistent buffer caches, dirty, committed buffers can exist if the database uses a no-force policy. Buffers are dirty and committed until being forced to disk, at which time they are marked as clean. However, non-persistent buffer caches keep these buffers safer than persistent buffer caches. This is because the recovery process for non-persistent buffer caches discards memory buffers and uses the redo log to recover the data. Hence if the database system corrupts a buffer in this state and crashes soon afterwards, the corrupted data will not be made persistent. Corruption occurs only if the system stays up long enough to write the affected buffer to disk.

  It is clear from Figure 6.3 that the main difference between the persistent and non-persistent database buffer cache designs is the dirty, committed buffers. Dirty, committed buffers make systems with persistent buffer caches more vulnerable to software corrup-

tion than systems with non-persistent buffer caches. These buffers are vulnerable for a longer period of time in a system with persistent buffer caches, particularly compared to systems using a force policy (such as Postgres). And systems with non-persistent buffer caches experience corruption due to these buffers only if the system remains up long enough to propagate them to disk.

6.2.3 Memory Interface to Reliable Memory with Protection (Persistent, Protected Database Buffer Cache)

Our third design also uses a memory interface to reliable memory but uses virtual memory protection to protect against wild stores to committed, dirty buffers (this scheme was suggested in [Sullivan91a]). In this system, buffers are normally kept write protected. When a transaction locks an object, the page containing the object is unprotected; when the transaction commits, the page is reprotected. If multiple transactions use objects on the same page, the system reprotects the page when all transactions release their locks.

Figure 6.4 shows the buffer cache state transition diagram for the persistent protected database buffer cache. It is very similar to the persistent database buffer cache design. However, this scheme protects buffers (dirty committed data) in the state that our analysis in Section 6.2.1 showed are more vulnerable for persistent buffer caches. It also protects clean buffers as these buffers are not being used. Dirty and uncommitted data has the same vulnerability in all three designs. Thus, introducing protection makes persistent database buffer cache more reliable as clean and dirty/committed buffers (clear bubbles with dotted outlines in Figure 6.4) are now protected from software errors.

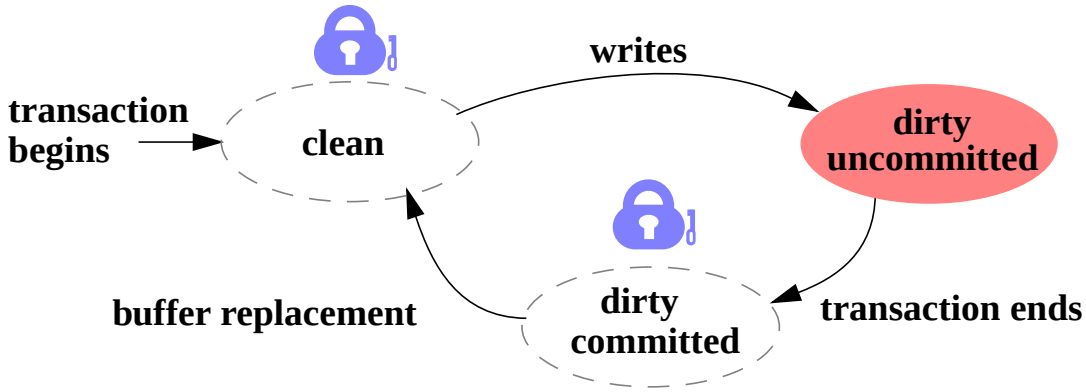

Figure 6.4 Protected Persistent Database buffer cache state.

Because the virtual memory hardware uses a page granularity, this scheme exposes unrelated objects that happen to be on the same page as the locked object. The scheme does not prevent object-level locking, however, since this locking can be accomplished independently from our protection mechanism.

Section 6.3 measures how effectively the virtual memory protection scheme protects committed, dirty pages from wild stores. The disadvantage of this scheme is that the extra protection operations may lower performance.

6.3 Reliability Evaluation

Persistent database buffer caches solve the double buffering problem by placing stable storage under database control. As discussed in Section 6.2.3, however, this design may be more vulnerable to database crashes. This section compares quantitatively the reliability of the different designs by injecting software bugs into Postgres to crash it, then measuring the amount of corruption in the database. We detect database corruption by running a repeatable set of database commands modeled after TPC-B [TPC90] and comparing the database image after a crash with the image that *should* exist at the point at which the crash occurred. Our fault model and experiment are described in Chapter 3.

Table 6.1 presents reliability measurements of our three designs. We conducted 50 tests for each fault category for each of the three systems; this represents 2 machine-months of testing. Individual faults are not directly comparable because of non-determinism in the timing of the faults and differences in code between the different systems; instead we look primarily at trends between the different systems.

Fault Type	I/O Interface (Non-Persistent Database Buffer Cache)	Memory Interface (Persistent Database Buffer Cache)	Memory Interface with Protection (Persistent, Protected Database Buffer Cache)
text	1	1	1
heap	0	0	0
stack	0	0	0
destination reg.	4	5	5
source reg.	2	2	2
delete branch	1	1	0
delete random inst.	2	2	2
initialization	0	1	1
pointer	0	0	0
allocation	0	0	0
copy overrun	0	0	0
off-by-one	5	5	3
synchronization	0	0	0
memory leak	0	0	0
interface error	4	3	3
Total	**19 of 750 (2.5%)**	**20 of 750 (2.7%)**	**17 of 750 (2.3%)**
95% Confidence Interval	**1.2%-3.8%**	**1.4%-4.0%**	**1.0%-3.5%**

Table 6.1 Comparing Reliability. This table shows how often each type of error corrupted data for three designs. We conducted 50 tests for each fault type for each of three systems using the TPC-B benchmark as the workload. The last row shows the 95% confidence interval. Even without protection, the reliability of the persistent database buffer cache is about the same as a traditional, non-persistent buffer cache. We have observed similar results in an earlier experiment with 100 tests for each fault type and the Wisconsin benchmark [Bitton83] as the workload.

Overall, all three designs experienced few corruptions—only 2.3-2.7% of the crashes corrupted permanent data over 2250 tests. This argues that, even with research database such as Postgres, software bugs may crash the system but usually do not corrupt permanent data. There are several reasons contributing to Postgres's robustness. First, Postgres has many programmer assertions that quickly stop the system after a fault is activated. Detecting the fault quickly and stopping the system prevents erroneous data from being committed to the permanent database image. Second, the operating system provides built-in assertions that check the sanity of various operations. For example, dereferencing a NULL pointer will cause the database to stop with a segmentation violation. In general, faults that left the system running for many transactions (such as off-by-one and interface) tended to corrupt data more frequently than faults that crashed the system right away (such as heap and stack).

We next compare the reliabilities of the three designs. The amount of corruption in all three systems is about the same. The differences are not large enough to make firm conclusions distinguishing the three systems, however we note that, as expected,

- the traditional I/O interface (non-persistent buffer cache) achieved the highest reliability (2.5%)
- the memory interface (persistent buffer cache) showed slightly worse reliability (2.7%)
- adding protection to the persistent buffer cache improved its reliability (2.3%)

As it is difficult to prove that our fault model represents real faults, we present three different interpretations of Table 6.1 by varying the weights associated with each fault type according to fault distributions published on DB2 and IMS [Sullivan91b]. Sullivan's study includes a detailed breakdown of software errors according to the following classifi-

cation: deadlock and synchronization, pointer management, memory leak, uninitialized data, copy overrun, allocation management, statement logic, data error, interface error, undefined state, unknown and other error. Table 6.2 shows the mapping between our fault categories and Sullivan's studies, together with the resulting weights. Undefined state, unknown and other errors are too vague to be precisely modeled, so we distribute their weights evenly across other categories.

Table 6.3 shows the mean corruption rate and the 95% confidence interval for all three designs. The confidence intervals are computed according to the approach suggested by Powell et al. [Powell95]. A memory interface generally has a higher mean corruption rate than an I/O interface. Adding protection to memory interface lowers the mean corruption

Fault Type	**Classification**	**Weight**	
		DB2	**IMS**
kernel text	statement logic	1.8%	2.1%
kernel heap	data error	2.1%	1.1%
kernel stack	data error	2.1%	1.1%
destination reg.	data error	2.1%	1.1%
source reg.	data error	2.1%	1.1%
delete branch	statement logic	1.8%	2.1%
delete random inst.	statement logic	1.8%	2.1%
initialization	initialization	6.3%	6.0%
pointer	pointer	10.4%	10.9%
allocation	allocation	8.1%	5.0%
copy overrun	copy overrun	5.4%	3.5%
off-by-one	statement error	1.8%	2.1%
synchronization	synchronization	20.3%	4.5%
memory leak	memory leak	3.6%	3.5%
interface	interface	6.8%	7.5%

Table 6.2 Proportional Mapping. This table shows how we map between the fault type in our study and those of [Sullivan92], and the corresponding weight assigned to each fault type.

rates. Because all three designs have significant overlap in confidence interval and their means are within the confidence intervals of other designs, we conclude with 95% confidence interval that all three designs have comparable reliability.

Our main conclusion is that mapping reliable memory directly into the database address space does not diminish the overall reliability of the system. This is consistent with the estimates given in [Sullivan91a, Sullivan93]. There are several factors that minimize the reliability impact of persistent buffer caches. First, most stores in Postgres are not to the buffer cache. Using the ATOM program analysis tool [Srivastava94], we found that only 2-3% of stores executed during a run were to the buffer cache. Second, store instructions that are not intended to access the buffer cache have little chance of accidentally wandering into buffer cache space, especially with the vast, 64-bit virtual address space on DEC Alphas. As a result, most corruptions are due to corrupting the *current* transaction's data. These uncommitted buffers are vulnerable to the same degree in all three systems (Section 6.2).

Weight Distribution	I/O Interface		Memory Interface		Memory Interface with Protection	
	Mean	95% Conf. Intvl	Mean	95% Conf. Intvl	Mean	95% Conf. Intvl
equal	2.5%	1.2%-3.8%	2.7%	1.4%-4.0%	2.3%	1.0%-3.5%
proportional DB2	1.5%	0.6%-2.3%	1.5%	0.7%-2.4%	1.4%	0.5%-2.2%
proportional IMS	2.1%	0.8%-3.3%	2.1%	0.8%-3.3%	1.8%	0.6%-3.0%

Table 6.3 Weighted Mean Corruption Rate. This table shows the average corruption rate and the 95% confidence interval. Our tests show that accessing persistent data via a memory interface does not significantly hurt reliability compared to accessing it via an I/O interface. The corruption rate of memory interface can be reduced even further by using a simple memory protection scheme. There is significant overlap in the confidence intervals of all three designs, indicating that the designs are comparable in reliability.

Thus, mapping reliable memory directly into the database address space does not lower reliability. Combined with the advantages of persistent buffer caches (reliable memory under database control, no double buffering, simpler recovery), these results argue strongly for using a memory interface to reliable memory. Stated another way, the high-overhead I/O interface to reliable memory is not needed, because wild stores are unlikely to corrupt non-related buffers.

6.4 Discussion

This chapter describes a controlled experiment to measure database corruption. No prior studies have directly quantified database'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.

- **How can we extrapolate our results to commercial database systems?** We modified the Postgres95 [Stonebraker87] database management system for our experiment. Postgres differs from most commercial database systems in several ways. For example, it does not use conventional write-ahead logging (WAL), and it only provides a flat transaction model [Gray93]. Our work is largely independent of these differences—we focus on the reliability implications of persistent database buffer caches, and all databases use a buffer cache. Our results are also consistent with a prior study on a commercial database [Sullivan92]. We hope to conduct further experiments on commercial database systems to further support our hypothesis.
- **Can state-of-the-art database recovery schemes use reliable memory?** As described in Section 2.1, any database can run unchanged on reliable memory. A reliable file cache such as Rio simply makes I/O appear faster. Reliable memory also allows state-

of-the-art recovery schemes (e.g. ARIES [Mohan92]) to become significantly simpler. For example, the need for fuzzy checkpoints is reduced because transactions can commit faster. Also, using a force-at-commit policy reduces the need for a redo log.

6.5 Summary

We have proposed three designs for integrating reliable memory into databases. Keeping an I/O interface to reliable memory requires the fewest modifications to an existing database but wastes memory capacity and bandwidth with double buffering. Mapping reliable memory into the database address space allows a persistent database buffer cache. This places reliable memory under complete database control, eliminates double buffering, and simplifies recovery. However, it also exposes the buffer cache to database errors. This exposure can be reduced by write protecting buffer pages.

Extensive fault tests show that mapping reliable memory into the database address space does not significantly hurt reliability. This is because wild stores rarely touch dirty, committed pages written by previous transactions. Combined with the advantages of persistent buffer caches, these results argue strongly for using a memory interface to reliable memory.

Chapter 7

Design Dimensions for Reliable Memory

We have described various reliable memory design techniques for three different platforms in Chapters 4, 5 and 6. In this chapter we will examine each technique in greater detail. We categorize the techniques according to how they are used to protect memory content during a system crash, and present experimental results on the reliability and performance implications of the techniques. Our key design principles are *early* fault detection and *correct* fault recovery: we detect fault very early in the crash before it has a chance to corrupt file cache data, and we correctly restore file cache data by depending on as little system state as possible. Figure 7.1 shows the series of events that occur during a system crash. When the system experiences a software fault, the error initially propagates within the system. The error is subsequently detected by our fault detection techniques, and our fault recovery routine correctly recovers the file cache data in main memory to disk. Section 7.1 describes the fault tolerant techniques we use to detect faults, which include VM protection, reset key and watchdog timers. Section 7.2 describes the fault recovery techniques we use to recover the data in main memory after a fault is detected, and how we decouple the file cache restore process from kernel state. Section 7.3

Figure 7.1 Effects of Faults During a System Crash. This figure presents a simplified version of the system-failure response stages found in [Siewiorek98].

describes the impact of the system configuration, particularly the physical memory address size, on the effectiveness of our fault tolerant techniques.

Some of the design techniques are platform specific, and we provide a summary of the applicability of the design techniques to the three platforms in Table 7.1.

Unless otherwise stated, the results reported in this chapter are measured on PCs equipped with Intel VS440FX motherboard, 266 MHz Pentium II processor, 128 MB DRAM, 4GB Seagate SCSI disk, AMIbios v1.00.06.CS1, and FreeBSD OS v2.2.7.

7.1 Fault Detection Techniques

Fault detection techniques are used to detect anomalous conditions in the system. Most systems already have some mechanisms to detect anomalous conditions. For example, most operating systems utilize the exception handling mechanisms offered by modern processors to detect many types of faults (e.g. divide errors, invalid instruction [Alp92, Int97d]). In both our FreeBSD and Digital UNIX Rio file caches, we use the default system trap handler to detect system fault. We also evaluated three fault detection techniques, VM protection, reset key and watchdog timer, to complement the default trap handler.We use VM protection mechanism to detect accidental corruption of file cache data. Reset key and watchdog timer are used to recover from system hang.

Design Techniques	FreeBSD Rio	Digital UNIX Rio	Postgres DBMS
VM Protection	yes	yes	yes
Software Reset Key	yes	no (has halt button)	no (use UNIX signal)
Watchdog Timer	yes	no (has halt button)	no (use UNIX signal)
Registry	yes	yes	yes
Physical Addressing	yes	no (use warm reboot)	no (application)
Device Driver	BIOS	BIOS	UNIX
Recovery Technique	safe sync	warm reboot	warm reboot

Table 7.1 Applicability of Reliability Memory Design Techniques.

Fault detection techniques can be described in four dimensions: detection latency, redundancy, recovery mechanism, and implementation overheads. Detection latency refers to the latency between file cache data corruption and detection. Ideally, we want zero latency to prevent data corruption. This can be achieved by using VM protection mechanism to protect dirty file cache data immediately after we have modified it. Other techniques often detect corruption long after the fault has corrupted data and are less effective. Redundancy refers to the amount of extra code or data that is required to implement the fault detection mechanism. VM protection mechanism requires both extra code (e.g. to protect data before we modify it) and data (e.g. a byte to indicate the protection setting of the buffer page), while both reset key and watchdog timer use extra code to detect system hang. Recovery mechanism refers to whether any operator supervision is required to initiate fault recovery. VM protection mechanism can automatically recover the file cache data by running safe sync or warm reboot after a fault is detected, while reset key require the user to manually detect system hang. Overheads refer to the performance overheads required to implement the fault detection mechanisms. Table 7.2 characterizes the various fault detection techniques we investigated in our study.

7.1.1 VM Protection

VM protection is a general mechanism offered by modern processors [Alp92, Int97b] to limit access to code or data based on privilege levels. It can be applied to many parts of

Fault Detection Dimensions	Detection Latency	Redundancy	Recovery Mechanism	Overheads
VM Protection	Immediate	Code/Data	Automatic	High
Reset Key	Delayed	Code	Manual	Low
Watchdog Timer	Delayed	Code	Automatic	Low

Table 7.2 Characterization of Fault Detection Techniques.

the system address space: user code and data, kernel code and data (e.g. stack, heap, buffer page, registry, user-level data, etc). In our study the most critical component is that kernel code and data needed in the fault recovery phase, so we focus on evaluating the effectiveness of protecting the kernel code, buffer cache and registry data.

7.1.1.1 Protecting Kernel Code

We protect the kernel code by mapping the whole kernel text as read-only during kernel initialization. This scheme protects against faults that modify the kernel code, which can either corrupt the file cache restore code or cause incorrect data to be written to the file cache. For example, some of the injected faults in our FreeBSD Rio study that corrupts the FreeBSD kernel text will also zero out the whole page. This can potentially wipe out safe sync code residing in the same page as the corrupted code, causing the safe sync process to fail. We can prevent this type of fault from occurring by either mapping the kernel text as read-only, or implementing the warm reboot mechanism used in Digital UNIX Rio (see Section 4.2.2.1). Warm reboot does not use any kernel code that exists in memory before the crash. It reloads the kernel image from disk after the crash and thus always runs uncorrupted code.

Protecting the kernel text helps to improve reliability by 2% (from 3.47% to 3.53%) in FreeBSD Rio. Although this type of fault is infrequent, we advocate protecting the kernel text as this scheme improves system reliability with no performance penalty; the performance overhead is incurred only during system startup.

7.1.1.2 Protecting Buffer Cache

We protect the file system buffer by setting the VM protection bits of the buffer page as read-write before modifying the page, and resetting the protection to read-only after

modifying the page. This substantially limits the number of dirty buffer pages that are writable at any time in the kernel, and prevents accidental corruption of the buffer cache by faults such as copy overruns and wild pointers [Sullivan91a]. Table 7.3 compares the reliability of the FreeBSD Rio file cache with and without buffer protection, and categorizes the corruptions according to fault symptoms. We observe from Table 7.3 that the overall corruption rate is reduced by 37% (from 3.5% to 2.2%) once we protect the buffer page. We observe similar improvement in reliability on Digital UNIX Rio file cache (43%, see Table 4.2) and Postgres reliable buffer cache (11%, see Table 6.1).

The most dramatic improvement comes from the reduction of data corruptions fault symptom, which mainly affects the copy overrun fault type. The number of corruption due to data corruption is reduced from 6 to 0 with buffer protection for the FreeBSD Rio file cache (see Table 7.3), and 4 to 0 for the Digital UNIX Rio file cache (see Table 4.2). Buffer protection is especially important for Rio file cache as Rio can cache significantly more dirty data in memory than the write-through file cache (see Table 7.4), and is thus more vulnerable to copy overrun fault type.

Fault Symptoms	No Protection	Protect Buffer	Protect Registry	Protect Buf & Reg
File system errors	25	20	20	17
Page fault during sync	11	2	7	5
VM errors	6	9	15	10
Data corruption	6	0	4	0
System hang	4	2	5	5
Total	**52 of 1500 (3.5%)**	**33 of 1500 (2.2%)**	**50 of 1500 (3.3%)**	**36 of 1500 (2.4%)**

Table 7.3 Breakdown of Corruptions according to Fault Symptoms for the FreeBSD Rio File Cache with 128 MB main memory.

Another reason why buffer protection is effective is that it tends to exercise the VM system frequently. In the FreeBSD OS, the VM protection code potentially needs to modify the kernel address map [McKusick96] six times when protecting or unprotecting a single buffer page. This is because the target buffer has different protection settings from the other pages in its neighboring virtual address space, so we need to create a new address map entry for the target buffer. We need to extract the buffer page from its parent map using the FreeBSD *vm_map_clip_start()* and *vm_map_clip_end()* routines, change the new address map entry's protection setting, and integrate it into the parent map using the FreeBSD *vm_map_simplify_entry()* routine. The integrity of the kernel address map is implicitly checked in each address map modifications. We thus detect corruptions in the kernel address map earlier, before the buffer with the corrupted address map entry is committed to stable store. As shown in Table 7.3, this helps to reduce the corruptions due to the fault symptoms page fault during sync from 11 to 2.

Protection can also be applied to the write-through file cache. Table 7.4 compares the reliability of the FreeBSD write-through file cache with and without buffer protection, and categorizes the corruptions according to fault symptoms. We see from Table 7.4 that protecting the buffer cache helps to reduce the corruption due to copy overrun from 3 to 0. However, protection is less effective here because VM errors corruptions outnumber data corruptions. Unlike the Rio file cache, the write-through file cache does not need to sync its data to disk during a crash, so it does not suffer from corruptions that can benefit from VM protection, such as the fault symptoms page fault during safe sync or system hang.

Most of the VM errors (4/6 in Table 7.4) are caused by a class of faults that affects only the write-through file cache. These faults occur when the input parameter or code in the FreeBSD protection routines *vm_map_protect()* and *pmap_protect()* is corrupted. As *pmap_protect()* is a void routine and *vm_map_protect*() only detects a small number of failure scenarios, both routines can potentially corrupt the page table entries (PTE) of a buffer page. A write-through file cache can thus corrupt data on disk when it writes data to disk, as each *write* function call invokes the protection routines at least once. The Rio file cache is unaffected by this type of fault as we update the registry *before* changing the protection settings. The registry thus contains a valid physical address for the buffer. Other routines that are vulnerable to this class of faults include void address map manipulation routines that are invoked by the protection routines, like the FreeBSD *vm_map_clip_start()* and *vm_map_simplify_entry()* routines. This bug affects 16% (0.3% of 1.9%) of the corruptions in the write-through file cache. We note that this type of bug can be overcome by delaying the protection change until we have written the buffer to disk. We do not attempt to fix this as we currently protect the buffer at the earliest instance (i.e. right after it is written to disk) to minimize the fault detection latency.

Fault Symptoms	No Protection	Protect Buffer
File system errors	21	19
VM errors	2	6
Data corruption	3	0
Unknown	1	3
Total	**27 of 1500 (1.8%)**	**28 of 1500 (1.9%)**

Table 7.4 Breakdown of Corruptions according to Fault Symptoms for the FreeBSD Write-through File Cache with 128 MB main memory.

Protecting the buffer cache adds negligible performance overheads on both FreeBSD and Digital UNIX Rio file caches. Table 7.5 compares the performance of various FreeBSD Rio file cache with different protection settings. The benchmark and experiment configurations can be found in Section 4.3. We observe from Table 7.5 that protecting the buffer incurs very little performance overheads on both benchmarks (<4%). We also observe similar overheads on Digital UNIX Rio file cache (see Table 4.3). We manage to achieve such low overheads because we change a page's protection settings by updating its PTE directly instead of using the more expensive OS protection change routine (e.g. *vm_map_protect* in FreeBSD).

However, the performance overheads to protect application-level data is quite high for transaction processing workload. For example, adding page-level protection on Postgres DBMS increases the transaction latency by 22% to 49% on TPC-A benchmarks. Most of these overheads are caused by the frequent invocations of *mprotect()* system call on fine-grain data common in TPC transactions (e.g. incrementing the account balance field in an account record). In contrast, file system workloads tend to work on larger data sets, so the performance overhead of the *mprotect()* system call is amortized over a full page. Other

Protection Settings	cp+rm (seconds)	Andrew (seconds)
Rio with no protection	3.10	1.20
Rio with buffer protection only	3.20	1.25
Rio with registry protection only	3.70	1.30

Table 7.5 Performance Comparison of FreeBSD Rio File Cache with Different Protection settings. The performance results are based on an average of 40 runs, measured on a PC running FreeBSD OS 2.2.7. The system configuration is similar to that described in Chapter 5.

researchers have looked at different approaches to minimize the performance overheads of protecting user-level data for database workload [Sullivan91a, Bohannon97].

7.1.1.3 Protecting Registry

The registry is another important data structure needed for fault recovery, and is used by both the safe sync and warm reboot data recovery mechanisms. The registry is an array in memory that contains, for each buffer page, the buffer's address in memory and disk, and a status word. We update the registry whenever we modify a buffer page. Each time we modify a buffer page, we will also set the VM protection bits of the page containing the registry as read-write before modifying the buffer, and reset the protection of the same page to read-only after modifying the buffer.

We can see from Table 7.3 that it is not effective to do registry protection as it offers minimum reliability improvement (3.3% vs. 3.5%). The performance overhead is also higher here (14% on Andrew benchmark, see Table 7.5). There are several reasons why we do not improve system reliability when we protect the registry. First, the granularity of protection is too coarse. Unprotecting a single registry entry also unprotects the page containing many other registry entries (more than 200 entries in FreeBSD Rio). This leaves many registry entries unprotected. Second, because we do not protect the buffer data, this results in a number of data corruptions due to copy overrun fault type. Third, registry entries are also modified much more frequently (e.g. at least twice for Andrew benchmark) than data buffers. This increases the chance that a fault might occur while the kernel address map is being manipulated, thus corrupting the kernel address map or the PTE for the registry. Our safe sync routine uses virtual addressing and can fail if the VM subsystem is compromised. We observe from Table 7.3 that the VM error corruptions increase

from 6 to 15 when we start protecting the registry. Note that this would not be a problem if safe sync used physical addressing.

7.1.2 Reset Key and Watchdog Timer

System hang is a major cause of corruption in our fault injection study, it affects 12-18% of the crash. The ideal solution is to use a reset or halt key that allows us to get control of the system when it hangs. For example, the Digital Alpha workstation has a hardware halt button that allows us to get control of the system without affecting the processor cache or main memory. However, it is more difficult to recover from system hang on Intel PCs. Most PCs wired the reset switch directly to the INIT or RESET [Int97b] pins on the processor. This complicates data recovery as we can lose file cache data in both the processor's cache and main memory. In most PCs, pressing the reset switch initializes the processor's cache and reboots the system. The ensuing reboot also wipes out the system main memory, as most PC BIOS do not preserve the main memory content during reboot.

We implemented two techniques to recover from system hang on Intel PCs. Most system hang occurs when the injected bug causes the system to enter a deadlock, and we cannot get control of the system from either the keyboard or network. To recover from system hang, we modify the low-level keyboard interrupt handler (*scgetc()* in FreeBSD) to intercept keyboard inputs. The keyboard handler invokes safe sync when it detects a "reset key" input from the keyboard. This is a manual process as we still need to detect that the system has hung and press the reset key. Table 7.6 compares the effectiveness of utilizing reset key on two different FreeBSD Rio file cache designs: Rio file cache without buffer protection, and Rio file cache with buffer protection. We observe that on both designs,

implementing the reset key helps to reduce the corruption rate by 82-83% (i.e. from 21.8% to 4.0% and from 14.9% to 2.6%, see Table 7.6).

Our reset key approach fails when the fault also disables the keyboard software interrupt mask. This prevents the system from servicing any keyboard inputs.We can implement a watchdog timer [Johnson89] to recover from this fault. We modify the low-level interrupt handler routine (*INTR* in FreeBSD) to start a timer once we receive a reset key

Fault Type	Rio without Protection			Rio with Buffer Protection		
	Rio	**Add Reset Key**	**Add Watch-dog Timer**	**Rio**	**Add Reset Key**	**Add Watch-dog Timer**
text	7	6	4	3	1	1
heap	4	0	0	2	0	0
stack	2	2	2	2	1	1
initialization	11	4	4	6	5	5
delete random inst.	15	8	6	22	7	5
source reg.	15	3	3	10	5	3
dest. reg.	11	1	1	4	5	4
delete branch	22	8	6	13	4	4
pointer	13	1	1	9	2	2
allocation	100	9	8	100	0	0
copy overrun	9	0	0	2	3	2
synchronization	81	16	15	12	0	0
off-by-one	32	0	0	29	5	5
memory leak	0	0	0	1	0	0
interface error	5	2	2	9	1	1
Total	**327/1500 21.8%**	**60/1500 4.0%**	**52/1500 3.5%**	**224/1500 14.9%**	**39/1500 2.6%**	**33/1500 2.2%**

Table 7.6 Evaluating the Effectiveness of Reset Key and Watchdog Timer. This table shows the reliability of Rio File Cache, Rio File Cache with Reset Key, and Rio File Cache with Reset Key *and* Watchdog Timer. Two types of Rio File Cache designs are evaluated: Rio File Cache without Buffer Protection, and Rio File Cache with Buffer Protection.

input. The counter times out if it is not reset after a certain threshold. Table 7.6 shows that adding a watchdog timer to both Rio file cache designs reduces the corruption rates on both designs by an additional 13-15%.

We tried two alternative designs to the watchdog timer: fast interrupts and watchdog event counter. The former approach forces the keyboard handler to use fast interrupt handler (*FAST_INTR* in FreeBSD), which allows keyboard interrupts to bypass the software interrupt mask. This approach is unstable as it may cause the system to miss other crucial interrupts like system timer interrupts during periods of heavy keyboard activities. The latter approach counts the number of reset key input events, and triggers safe sync if these events have reached a certain threshold and have not been serviced. This approach does not work on systems with hardware interrupt controllers (such as the Intel 8259), which limit the number of unserviced interrupts on the processor.

The reset key and watchdog timer techniques allow us to recover from most types of system hang. However, both approaches fail if the injected bug also disables the hardware interrupt mask. This occurs in 0.3% of the crashes in our FreeBSD Rio study (5/1500, see Table 5.6). In this case, we need a hardware reset key similar to the halt button in the Digital Alpha workstation. This can be implemented in Intel PC in hardware by using a peripheral card to generate a non-maskable interrupts (NMI) [Int97d]. We did not pursue this approach since this type of fault is relatively infrequent, and it requires a hardware solution.

7.2 Fault Recovery

Fault recovery techniques are used to ensure that we can recover the file cache data in memory after we detect a system fault. There are four design dimensions to data recovery:

when to restore, what information do we need, how to access the data and code, and how to write to disk.

7.2.1 When to Restore: Warm Reboot vs. Safe Sync

We can either restore the file cache data during a system crash or reboot. It is safer to restore file cache data during reboot as the system has been reset and is in a "clean" state. This approach, warm reboot, is used in both our Digital UNIX Rio file cache and Postgres reliable buffer cache. However, it is difficult to implement warm reboot on platforms such as Intel PCs, which have a reset key that erases memory. Our second approach, called safe sync, writes dirty file cache data reliably to disk during the last stage of a crash. Both approaches work well: we demonstrated in Chapters 4 and 5 that a Rio file cache design with either safe sync or warm reboot has better reliability than a write-through file cache. It is difficult to compare the two techniques directly as we did not implement both techniques on the same platform. However, we can analyze the fault symptoms for a safe sync design and extrapolate how much warm reboot will help to improve system reliability. The fault symptom classification for BIOS safe sync design is shown in Table 5.6. Warm reboot can potentially eliminate the fault symptom device timeout as it performs the disk write after the system has initializes the disk subsystem. Warm reboot can thus potentially improve the reliability of a safe sync design by 7% (from 1.9% to 1.7%, see Table 5.6). Warm reboot is also easier to implement as it does not have the complex setup that safe sync needs to run successfully during a crash. Warm reboot is thus preferable on platform that supports it.

7.2.2 What Information Do We Need?

We need three types of information to safely restore all the file cache data from memory to disk. We need to locate all the buffers in memory, and determine the buffer's loca-

tion on disk. We also need the status of the buffer, as we only want to restore valid and dirty buffer to disk. This information is already contained in memory in data structures such as the mounted file system list, buffer cache hash list, page tables, etc. However, it is difficult to gather the information directly from all these diverse data structures. Most of these data structures are large and are indexed by a hash table or linked list, making them vulnerable to corruptions during a software crash. Instead, we duplicate key kernel data structures in an array in memory called the registry. Our experience indicates that the registry is a critical component in our design. For example, using the registry accounts for a significant portion of the improvement in reliability between design iterations 1 and 2 in our FreeBSD Rio file cache design. We observe from Table 5.3 and Table 5.4 that having a registry and safe sync help to reduce the corruption by approximately 40% (from 39.3% to 23.7%).

7.2.3 How to Access the Data and Code?

Most systems use virtual addressing by default. This works fine if we use warm reboot, as the restore is performed after a system crash. But virtual addressing may not work well for safe sync designs. Safe sync runs during a system crash, and the VM system may be corrupted. We can bypass the VM by accessing the code and data directly using its physical address. For example, we observe that using physical addressing accounts for a significant portion of the improvement in reliability between design iterations 3 and 4 in our FreeBSD Rio file cache design. We observe from Table 5.5 and Table 5.6 that using physical addressing and BIOS drivers helps to reduce the corruption rate by approximately 42% (3.3% to 1.9%).

7.2.4 How to write the data to disk?

We can either use the operating system or BIOS device drivers to write file cache data to disk during fault recovery. OS device driver is usually large and complex because it needs to cater to many different failure scenarios. The device driver code and configuration data may be dynamically loaded and allocated to allow greater flexibility in system configuration. In contrast, the BIOS device driver is usually simpler because it offers only limited functionality. For example, both the PC BIOS and Digital Alpha PROM disk device drivers limit the size of the input data. On most systems BIOS drivers are mainly used to bootstrap the OS. BIOS code and data also resides in protected non-volatile memory such as EEPROM or CMOS, and are not vulnerable to corruption by faulty kernel or user code. Since safe sync and warm reboot routines do not need the advanced functionalities offered by the OS device drivers, it preferable to use BIOS drivers during fault recovery.

7.3 System Configuration

We have demonstrated that we can build reliable file cache on three diverse platforms, and developed design techniques that are suitable for a wide range of systems. In this section we look at two system configuration parameters, physical memory size and virtual address size, that affect some of the design techniques. We also discuss the sensitivity of our design techniques against these configuration parameters.

7.3.1 Physical Memory Size

The effectiveness of the Rio file cache is directly proportional to the size of the file cache. This is because we can cache more dirty data and filter write traffic more effectively when we have a larger file cache, i.e. more physical memory. Having more dirty data in memory makes Rio more vulnerable to software errors. We can limit this vulnerability by

protecting the data in memory. This implies that the effectiveness of VM protection of buffer data may be affected by the physical memory size. The rest of the design techniques are largely independent of memory size.

Table 7.7 shows the effect of main memory size on four types of file cache: Rio file cache with no buffer protection, Rio file cache with buffer protection, write-through file cache with no buffer protection, and write-through file cache with buffer protection. We

Fault Type	Rio File Cache				Write-though File Cache			
	No Protection		Protect Buffer		No Protection		Protect Buffer	
	128 MB	512 MB	128 MB	512 MB	128 MB	512 MB	128 MB	512 MB
text	4	5	1	1	3	3	0	1
heap	0	0	0	0	0	1	3	1
stack	2	6	1	2	1	2	3	3
initialization	4	3	5	1	7	11	0	4
delete ran. inst.	6	12	5	6	2	8	4	3
source reg.	3	6	3	3	1	1	2	3
dest. reg.	1	4	4	2	1	2	1	2
delete branch	6	10	4	4	2	2	4	5
pointer	1	7	2	6	2	4	2	2
allocation	8	0	0	0	0	0	1	0
copy overrun	0	16	2	8	4	1	2	0
synch.	15	0	0	0	0	0	0	0
off-by-one	0	15	5	6	4	12	4	5
memory leak	0	0	0	0	0	0	0	1
interface error	2	0	1	1	0	1	2	0
Total (out of 1500)	**52 3.5%**	**84 5.6%**	**33 2.2%**	**40 2.7%**	**27 1.8%**	**48 3.2%**	**28 1.9%**	**30 2.0%**

Table 7.7 Effect of Physical Memory Size on Reliability. This table shows the reliability of different file cache system on a system with 128 MB and 512 MB main memory. The systems studied include Rio file cache with and without buffer protection, Write-through file cache with and without buffer protection.

observe that the corruption rate increases on all four file systems when the memory size is increased from 128 MB to 512 MB. The increase is especially pronounced on file cache without VM protection. The corruption rates increase by 62% (from 3.5% to 5.6%) and 78% (from 1.8% to 3.2%) on Rio file cache without VM protection and write-through file cache without VM protection, respectively. This is largely due to the fact that there are more dirty buffers in memory, and this increases the likelihood that an injected fault will corrupt an unprotected dirty buffer. We can limit the probability of corruption by protecting the buffer.

We also observe that the effectiveness of VM protection increases with larger memory size. For example, for the Rio file cache with 128 MB memory, protecting the buffer reduces the corruption rate by 37% (from 3.5% to 2.2%). The reduction is more pronounced (52%, from 5.6% to 2.7%) for Rio file cache with 512 MB memory. Similarly, there is no reduction in corruption rate (1.8% vs. 1.9%) for a write-through file cache with 128 MB memory when we protect the buffer. However, the corruption rate reduces by 40% (from 3.2% to 2.0%) once the memory size increases to 512 MB.

Our results thus indicate that it is crucial to protect buffer data, particularly for system with large amount of memory.

7.3.2 Virtual Address Size

Another factor that affects reliability is virtual address size. In this dissertation we study systems with 64-bit address space (Digital UNIX Rio file cache and Postgres DBMS) and 32-bit address space (FreeBSD Rio file cache). It is very difficult to explicitly analyze the effect of virtual address size on system reliability, since all our systems do not

provide a virtual address size switch. In this section we will speculate on some of the effect of a larger address size on system reliability.

We speculate that one of the main reason why we observe low corruption rates on Digital UNIX Rio file cache (0.6%-1.5%) is that the Digital UNIX OS's 64-bit address space help to improve system reliability. It significantly reduces the chance that a random software bug will generate a valid and mapped address in the large 64-bit address space, thus offering significant protection from wild pointer and copy overrun fault types. We can also achieve more distance and better fault isolation by placing the kernel code, stack, heap, and buffer data in different part of the address space [Baker92_1]. In contrast, FreeBSD kernel code, stack, heap, and data are more closely packed in virtual address space than Digital UNIX. For example, the kernel heap typically occupies the address range 0xf3cff000-0xf5ea3000, while the buffer cache range from 0xf6730000-0xfc4f0000. A single bit flip in the seventh address byte will cause the heap pointer to point to the buffer cache. We have observed this type of corruption in one of the crashes during our fault injection study.

This issue is likely to be less important in future since the trend is towards 64-bit processors.

7.4 Summary

We categorize the reliable memory design techniques in this chapter into fault detection and fault recovery techniques. Our analysis shows that to detect fault early, it is critical to implement reset key and warm reboot (or to use the halt button if the platform supports it), and to protect buffer data in memory using VM protection mechanism. The effectiveness of the latter technique increases with larger physical memory. We also

observe that warm reboot is a more effective technique than safe sync for fault recovery because it does not depend on any pre-crash system state. We can make safe sync nearly as reliable as warm reboot by minimizing our dependence on system state (i.e. by using physical addressing and BIOS device drivers for I/O).

Chapter 8

Conclusions

We have made a case for reliable file caches: main memory that can survive operating system crashes and be as safe and permanent as disk. Our reliability experiments show that our Rio file cache is at least as reliable than a write-through file cache. This concluding chapter summarizes the results of this dissertation and lists future work in this area.

8.1 Summary of Results

We have presented a systematic and quantitative approach for using software-implemented fault injection to guide the design and implementation of a fault-tolerant system. Our goal was to build a write-back file cache that is as reliable as a write-through file cache. We followed an iterative approach to improve the robustness of a write-back file cache in the presence of operating system errors. In each iteration, we measured the reliability of the system, analyzed the fault symptoms that led to data corruption, and applied fault-tolerant mechanisms that address the fault symptoms. The result of several iterations was a design that improved reliability by a factor of 21 (on FreeBSD Rio) and 2.5 (on Digital UNIX Rio). The resulting write-back file cache is both more reliable (1.9% vs. 3.1% corruption rate on FreeBSD Rio, 0.6% vs. 1.1% on Digital UNIX Rio) and significantly faster than a write-through file cache (12-55 times on FreeBSD Rio, 4-22 times on Digital UNIX Rio). Our results also indicate that it is crucial to protect buffer cache data using VM protection techniques, especially for systems with large amounts of physical memory.

Reliable file caches have striking implications for future system designers:

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

To further test and prove our ideas, we have installed a departmental file server using the Rio file cache with protection and with reliability-induced writes to disk turned off. Among other things, this file server stores our kernel source tree, the only copy of this dissertation, and our mail. The server has been operational for the past four years, and we experienced more than ten software crashes. The Rio file cache worked remarkably well during these crashes, and we did not lose any user data.

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

8.2 Future Work

There are several topics related to reliable memory that are not covered by this dissertation. This section lists a few of these topics.

8.2.1 Error Coverage

We have developed a "reasonable" fault model to model common software errors in our study, but there is always a concern on the validity of our fault model (i.e. do our injected faults really represent software faults that cause system crashes?). We believe that our experiments cover a wide range of real-world crashes due to the wide variety of faults we inject (15 types), the random nature of the faults, the large number of ways the system crashed in our experiments, and the sheer number of crashes we performed (e.g. more than 7500 crashes for the FreeBSD study). A more direct approach to address this concern is to inject real operating system bugs into our system. This approach is viable for the FreeBSD operating system, which maintains a source code repository with detailed bug reports. The kernel source code repository has recently been made accessible via the internet. We can access the bug report on-line using Anonymous CVS (Concurrent Versions System) [Cranor99]. We plan to extract real kernel bugs from the repository and inject these bugs into the systems to understand how real bugs affect our file cache data. This will also allow us to determine the fault coverage of our designs. We are not aware of any study that evaluates fault coverage of large software systems.

8.2.2 Protecting Against Hardware Faults

Rio's focus is reliability (not losing data). It achieves this by protecting memory during system crashes and recovering memory using warm reboot. In general, achieving high availability (being able to access data continuously) requires replication of the storage devices and/or computers [Gray93, Cardoza96]. We may want to explore providing high

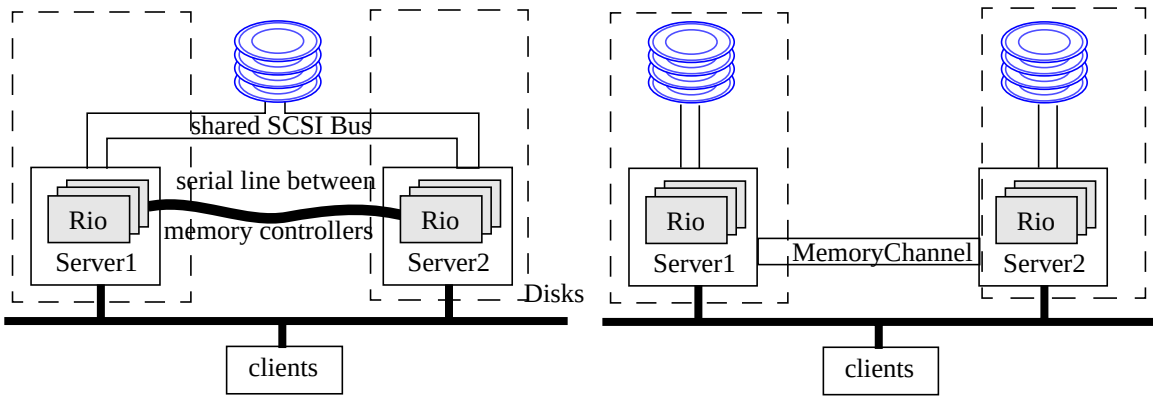

Figure 8.1 Increasing the Availability of Reliable Memory Systems.

availability in Rio by replicating the memory or computer. This replication can be done lazily or eagerly. An inexpensive example of lazy replication is to connect a serial line between the memory controllers of two computers, and to connect disks to both I/O controllers (see Figure 8.1a). This provides low-bandwidth access to the data in memory while rebooting or during hardware failures. A more expensive solution is to eagerly replicate the file cache data on a remote computer [Papathanasiou98] using a high-speed network such as Memory Channel [Gillett96], Scalable Coherent Interface [Scott92], or a LAN (see Figure 8.1b). This provides higher availability if one machine is permanently disabled.

8.2.3 Performance Characterization of Commercial Databases

Recent performance studies [Barroso98, Keeton98, Ranganathan98] on commercial databases running OLTP (On-line Transaction Processing) applications indicated that most database engines are not I/O bound. This is in contrast with past studies [Cvetanovic94, Rosenblum95] which observed that database applications are largely I/O bound and spend most of their execution time in the operating system. Modern commercial database engines can tolerate I/O latencies and incur much less operating system over-

heads [Barroso98]. There is thus some doubts as to how much performance gain we can achieve by using reliable memory on commercial databases. We did not attempt to address this question in Chapter 6 as we use an experimental database in our study. We did conduct some experiments on Informix database server DS7.3 running the TPC-A benchmark. We observed 7-12% performance gain when we disable disk writes on the operating system. A future area of research is to understand how commercial databases can exploit reliable memory to reduce the logging, synchronization and I/O overheads required to support advanced disk I/O techniques [Lowell97].

8.2.4 Fault-tolerant System Design

We have focused on the systematic design and implementation of reliable main memory in this dissertation. We believe that our approach is also applicable to other fault-tolerant system designs. We plan to investigate applying our fault-tolerant design methodology to other systems, such as embedded control systems.

Bibliography

[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 (FTCS)*, pages 414–423, June 1994.

[Agrawal89] Rakesh Agrawal and H. V. Jagadish. Recovery algorithms for database machines with nonvolatile main memory. In *Database Machines. Sixth International Workshop, IWDM '89 Proceedings.*, June 1989.

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

[Alp92] Alpha Architecture Handbook. Technical report, Digital Equipment Corporation, February 1992.

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

[Arlat90] Jean Arlat, Martine Aguera, Louis Amat, Yves Crouzet, Jean-Charles Fabre, Jean-Claude Laprie, Eliane Martins, and David Powell. Fault Injection for Dependability Validation: A Methodology and Some Applications. *IEEE Transactions on Software Engineering*, 16(2):166–182, February 1990.

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

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

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

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

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

[Baron90] Robert V. Baron, David Black, William Bolosky, Jonathan Chew, Richard P. Draves, David B. Golub, Richard F. Rashid, Jr.

Avadis Tevanian, and Michael W. Young. Mach Kernal Interface Manual. Technical Report CMU unpublished report, Carnegie Mellon University, August 1990.

[Barroso98] Luiz Andre Barroso, Kourosh Gharachorloo, and Edouard Bugnion. Memory System Characterization of Commercial Workloads. In *Proceedings of the 1998 International Symposium on Computer Architecture*, pages 3–14, June 1998.

[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.

[Bensoussan72] A. Bensoussan, C.T. Clingen, and R.C. Daley. The Multics Virtual Memory: Concepts and Design. *Communications of the ACM*, 15(5):308–318, May 1972.

[Bershad95] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gun Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the 1995 Symposium on Operating Systems Principles*, pages 267–283, December 1995.

[Bhide93] Anupam Bhide, Daniel Dias, Nagui Halim, Basil Smith, and Francis Parr. A Case for Fault-Tolerant Memory for Transaction Processing. In *Proceedings of the 1993 International Symposium on Fault-Tolerant Computing*, pages 451–460, June 1993.

[Bitton83] Dina Bitton, David J. DeWitt, and Carolyn Turbyfill. Benchmarking Database Systems–A Systematic Approach. In *Very Large Database Conference*, pages 8–19, October 1983.

[Bohannon97] Philip Bohannon, Daniel Lieuwen, Rajeev Rastogi, S. Seshadri, Avi Silberschatz, and S. Sudarshan. The Architecture of the Dali Main-Memory Storage Manager. *Journal of Multimedia Tools and Applications*, 1997.

[Cardoza96] Wayne M. Cardoza, Frederick S. Glover, and William E. Snaman Jr. Design of the TruCluster Multicomputer System for the Digital UNIX Environment. *Digital Technical Journal*, (1):5–17, June 1996.

[Carreira95] Joao Carreira, Henrique Madeira, and Joao Gabriel Silva. Xception: Software Fault Injection and Monitoring in Processor Functional Units. In *Proceedings of the 1995 IFIP Working Conference on Dependable Computing for Critical Applications*, pages 135–149, September 1995.

[Carreira98] Joao Carreira, Henrique Madeira, and Joao Gabriel Silva. Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers. *IEEE Transactions on Software Engineering*, 24(2):125–136, February 1998.

[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.

[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.

[Chen96a] Peter M. Chen, Wee Teck Ng, Subhachandra Chandra, Christopher M. Aycock, Gurushankar Rajamani, and David Lowell. The Rio File Cache: Surviving Operating System Crashes. In *Proceedings of the 1996 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 74–83, October 1996.

[Chen96b] Peter M. Chen, Wee Teck Ng, Gurushankar Rajamani, and Christopher M. Aycock. The Rio File Cache: Surviving Operating System Crashes. Technical Report CSE-TR-286-96, University of Michigan, March 1996.

[Chen98] Peter M. Chen, David E. Lowell, and George W. Dunlap. Discount Checking: Transparent, Low-Overhead Recovery for General Applications. In *submitted for publication to the Symposium on Fault-Tolerant Computing*, November 1998.

[Chillarege89] R. Chillarege and N. S. Bowen. Understanding Large System Failure–A Fault Injection Experiment. In *Proceedings of the 1989 International Symposium on Fault-Tolerant Computing (FTCS)*, pages 356–363, 1989.

[Christmansson96] Jorgen Christmansson and Ram Chillarege. Generation of an Error Set that Emulates Software Faults Based on Field Data. In *Proceedings of the 1996 Symposium on Fault-Tolerant Computing (FTCS)*, June 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.

[Cranor99] Charles D. Cranor and Theo de Raadt. Opening the Source Repository with Anonymous CVS. In *Proceedings of the 1996 USENIX Annual Technical Conference: FREENIX Track*, June 1999.

[Cvetanovic94] Z. Cvetanovic and D. Bhandarkar. Characterization of Alpha AXP Performance Using TP and SPEC Workloads. In *Proceedings of the 1994 International Symposium on Computer Architecture*, pages 60–70, April 1994.

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

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

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

[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.

[Elhardt84] Klaus Elhardt and Rudolf Bayer. A Database Cache for High Performance and Fast Restart in Database Systems. *ACM Transactions on Database Systems*, 9(4):503–525, December 1984.

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

[Ganger94a] Gregory R. Ganger and Yale N. Patt. Metadata Update Performance in File Systems. *1994 Operating Systems Design and Implementation (OSDI)*, November 1994.

[Ganger94b] Gregory R. Ganger, Bruce L. Worthington, and Yale N. Patt. Scheduling Algorithms for Modern Disk Drives. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1994.

[Gillett96] R. B. Gillett. Memory Channel Network for PCI. *IEEE Micro*, 16(1):12–18, February 1996.

[Gilluwe97] Frank Van Gilluwe. *The Undocumented PC: A Programmer's Guide to I/O, CPUs, and Fixed Memory Areas*. Addison-Wesley Developer Press, 1997.

[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.

[Gray78] J. N. Gray. *Operating Systems: An Advanced Course*. Springer-Verlag, 1978. Notes on Database Operating Systems.

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

[Gray91] Jim Gray and Daniel P. Siewiorek. High-Availability Computer Systems. *IEEE Computer*, 24(9):39–48, September 1991.

[Gray93] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, Inc., 1993.

[Haerder83] Theo Haerder and Andreas Reuter. Principles of Transaction-Oriented Database Recovery. *ACM Computing Surveys*, 15(4):287–317, December 1983.

[Hagmann86] Robert B. Hagmann. A Crash Recovery Scheme for a Memory-Resident Database System. *IEEE Transactions on Computers*, C-35(9):839–843, September 1986.

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

[Harty92] Kieran Harty and David R. Cheriton. Application-Controlled Physical Memory using External Page-Cache Management. In *Proceedings of the 1992 International Conference on Architectural Support for Programming*

Languages and Operating Systems (ASPLOS), pages 187–197, 1992.

[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.

[Hudak93] John Hudak, Byung-Hoon Suh, Dan Siewiorek, and Zary Segall. Evaluation and Comparison of Fault-Tolerant Software Techniques. *IEEE Transactions on Reliability*, 42(2), June 1993.

[Int97a] *Intel 82371AB PCI ISA IDE Xcelerator (PIIX4) Datasheet*. Intel Corporation, 1997.

[Int97b] *Intel Architecture Software Developer's Manual: Volume 1: Basic Architecture*. Intel Corporation, 1997.

[Int97c] *Intel Architecture Software Developer's Manual: Volume 2: Instruction Set Reference*. Intel Corporation, 1997.

[Int97d] *Intel Architecture Software Developer's Manual: Volume 3: System Programming Guide*. Intel Corporation, 1997.

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

[Jain91] Raj Jain. The Art of Computer Systems Performance Analysis. Technical report, John Wiley & Sons, Inc., 1991.

[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.

[Johnson89] Barry W. Johnson. *Design and Analysis of Fault-Tolerant Digital Systems*. Addison-Wesley Publishing Co., 1989.

[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 Engineering*, 19(11):1105–1118, November 1993.

[Kawaf96] Tareef S. Kawaf, D. John Shakshober, and David C. Stanley. Performance

Analysis Using Very Large Memory on the 64-bit AlphaServer System. *Digital Technical Journal*, (3):58–65, December 1996.

[Keeton98] Kimberly Keeton, David A. Patterson, Yong Qiang He, Roger C. Raphael, and Walter E. Baker. Performance Characterization of a Quad Pentium Pro SMP using OLTP Workloads. In *Proceedings of the 1998 International Symposium on Computer Architecture*, pages 15–26, June 1998.

[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.

[Koopman99] Philip J. Koopman and John DeVale. Comparing the Robustness of POSIX Operating Systems. In *Proceedings of the 1999 Symposium on Fault-Tolerant Computing (FTCS)*, June 1999.

[Kropp98] Nathan P. Kropp, Philip J. Koopman, and Daniel P. Siewiorek. Automated Robustness Testing of Off-the_shelf Software Components. In *Proceedings of the 1998 Symposium on Fault-Tolerant Computing (FTCS)*, June 1998.

[Lee93a] Inhwan Lee and Ravishankar K. Iyer. Faults, Symptoms, and Software Fault Tolerance in the Tandem GUARDIAN Operating System. In *Proceedings of the 1993 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.

[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.

[Lehman92] Tobin J. Lehman, Eugene J. Shekita, and Luis-Felipe Cabrera. An Evaluation of Starburst's Memory Resident Storage Component. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):555–566, December 1992.

[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.

[Lowell97] David E. Lowell and Peter M. Chen. Free Transactions with Rio Vista. In *Proceedings of the 1997 Symposium on Operating Systems Principles*, pages 92–101, October 1997.

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

[McKusick96] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley Publishing Company, 1996.

[McKusick99] Marshall Kirk McKusick and Gregory R. Ganger. Soft Updates: A Technique for Eliminating Most Synchronous Writes in the Fast Filesystem. In *Proceedings of the 1996 USENIX Annual Technical Conference: FREENIX Track*, June 1999.

[Mohan92] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Transactions on Database Systems*, 17(1):94–162, March 1992.

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

[Ng97] Wee Teck Ng and Peter M. Chen. Integrating Reliable Memory in Databases. In *Proceedings of the 1997 International Conference on Very Large Data Bases (VLDB)*, pages 76–85, August 1997.

[Ng99] Wee Teck Ng and Peter M. Chen. The Systematic Improvement of Fault Tolerance in the Rio File Cache. In *Proceedings of the 1999 Symposium on Fault-Tolerant Computing (FTCS)*, June 1999.

[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.

[Papathanasiou98] Athanasios Papathanasiou and Evangelos P. Markatos. Lightweight Transactions on Networks of Workstations. In *Proceedings of the 1998 International Conference on Distributed Computing Systems (ICDCS)*, May 1998.

[Patterson95] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. Informed Prefetching and Caching. In *Proceedings of the 1995 Symposium on Operating Systems Principles*, pages 79–95, December 1995.

[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.

[Ranganathan98] Parthasarathy Ranganathan, Kourosh Gharachorloo, Sarita V. Adve, and Luiz Andre Barroso. Performance of Database Workloads on Shared-Memory Systems with Out-of-Order Processors. In *Proceedings of the 1998 In-*

ternational Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 307–318, October 1998.

[Rashid88] Richard F. Rashid, Jr. Avadis Tevanian, Michael Young, David Golub, Robert Baron, David Black, Jr. William J. Bolosky, and Jonathan Chew. Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. *IEEE Transactions on Computers*, 37(8):896–908, August 1988.

[Rela96] Mario Zenha Rela, Henrique Madeira, and Joao G. Silva. Experimental Evaluation of the Fail-Silent Behavior in Programs with Consistency Checks. In *Proceedings of the 1996 Symposium on Fault-Tolerant Computing (FTCS)*, June 1996.

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

[Rosenblum95] Mendel Rosenblum, Edouard Bugnion, Stephen Alan Herrod, Emmett Witchel, and Anoop Gupta. The Impact of Architectural Trends on Operating System Performance. In *Proceedings of the 1995 Symposium on Operating Systems Principles*, pages 285–298, December 1995.

[Satyanarayanan93] M. Satyanarayanan, Henry H. Mashburn, Puneet Kumar, David C. Steere, and James J. Kistler. Lightweight Recoverable Virtual Memory. In *Proceedings of the 1993 Symposium on Operating System Principles*, pages 146–160, December 1993.

[Scott92] Steven L. Scott, James R. Goodman, and Mark K. Vernon. Performance of the SCI Ring. In *Proceedings of the 1992 International Symposium on Computer Architecture*, May 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 *Proceedings of the 1988 International Symposium on Fault-Tolerant Computing (FTCS)*, pages 102–107, 1988.

[Seltzer96] Margo I. Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith. Dealing With Disaster: Surviving Misbehaving Kernel Extensions. *Operating Systems Design and Implementation (OSDI)*, October 1996.

[Shanley96] Tom Shanley. *Protected Mode Software Architecture*. Addison-Wesley Developer Press, 1996.

[Siewiorek93] Daniel P. Siewiorek, John J. Hudak, Byung-Hoon Suh, and Zary Segal. Development of a Benchmark to Measure System Robustness. In *Proceedings of the 1993 International Symposium on Fault-Tolerant Computing*, pages 88–97, June 1993.

[Siewiorek98] Daniel P. Siewiorek. *Reliable Computer Systems: Design and Evaluation*. A K Peters, 1998.

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

[Silva96] Joao G. Silva, Joao Carreira, Henrique Madeira, Diamantino Costa, and Francisco Moreira. Experimental Assessment of Parallel Systems. In *Proceedings of the 1996 Symposium on Fault-Tolerant Computing (FTCS)*, June 1996.

[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.

[Stonebraker81] Michael Stonebraker. Operating system support for database management. *Communications of the ACM*, 24(7):412–418, July 1981.

[Stonebraker87] M. Stonebraker. The design of the POSTGRES storage system. In *Proceedings of the 1987 International Conference on Very Large Data Bases*, pages 289–300, September 1987.

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

[Sullivan91b] Mark Sullivan and R. Chillarege. Software Defects and Their Impact on System Availability–A Study of Field Failures in Operating Systems. In *Proceedings of the 1991 International Symposium on Fault-Tolerant Computing*, June 1991.

[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.

[Sullivan93] Mark Paul Sullivan. *System Support for Software Fault Tolerance in Highly Available Database Management Systems*. PhD thesis, University of California at Berkeley, January 1993.

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

[Thakur95] Anshuman Thakur, Ravishankar K. Iyer, Luke Young, and Inhwan Lee. Analysis of Failures in the Tandem NonStop-UX Operating System. In *Proceedings of the 1995 International Symposium on Software Reliability Engineering*, pages 40–50, October 1995.

[TPC90] TPC Benchmark B Standard Specification. Technical report, Transaction Processing Performance Council, August 1990.

[Tsai96] Timothy K. Tsai, Ravishankar K. Iyer, and Doug Jewett. An Approach towards Benchmarking of Fault-Tolerant Commercial Systems. In *Proceedings of the 1996 Symposium on Fault-Tolerant Computing (FTCS)*, June 1996.

[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.