Cooperative Data Protection

by

Yupu Zhang

A dissertation submitted in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy
(Computer Sciences)

at the
UNIVERSITY OF WISCONSIN-MADISON

2014

Date of final oral examination: 02/10/14

Committee in charge:

Andrea C. Arpaci-Dusseau, Professor, Computer Sciences
Remzi H. Arpaci-Dusseau, Professor, Computer Sciences
Shan Lu, Assistant Professor, Computer Sciences
Michael M. Swift, Associate Professor, Computer Sciences
Peter Z.G. Qian, Associate Professor, Statistics

To my parents

Acknowledgements

First and foremost, I would like to express my deep gratitude to my advisors, Andrea Arpaci-Dusseau and Remzi Arpaci-Dusseau, who guided me through my Ph.D. studies. There is an old saying in China: a teacher for a day is a father for a lifetime. I feel extremely lucky and thankful to have both of them as my Ph.D. "parents".

My initial interest in system research was born when I took Remzi's Advanced Operating Systems class. When I just came here, operating systems was definitely not one of my favorites. However, Remzi's excellent teaching and deep knowledge convinced me that building systems is such a fun and challenging process that I can definitely start a Ph.D. journey in systems. My last concern for research, writing papers, was eased by Andrea's meticulous guidance. After reading a paper draft, she always provided me with a lot of feedback, ranging from grammatical corrections to organizational suggestions, which greatly improved the quality of my work. She taught me crucial skills to convert a complicated system written in C into a nice story with words and figures, which I could never have learned by myself in such a short time. Throughout my Ph.D. studies, not only did they give me numerous pieces of advices on how to be a good researcher, they also showed me how to be a better person. I am extremely thankful for their patience and support during my ups and downs. Without their encouragement, I would never have completed this exceptional Ph.D. journey.

Next, I would like to thank my thesis-committee members, Shan Lu, Peter Qian, and Mike Swift, for their insights and suggestions for my research. I would especially like to thank Mike for his detailed comments and challenging questions during my preliminary exam and defense, which greatly help me in improving and finishing my thesis.

I have benefited greatly from interning at NetApp. I would like to thank the company as well as my mentor, Kiran Srinivasan, and my manager, Shankar Pasupathy, for providing a terrific internship experience.

I am fortunate to have had the opportunity to work with smart and hardwork-

ing colleagues: Chris Dragga, Daniel Myers, Abhishek Rajimwale, Lanyue Lu, Swaminathan Sundararaman, Sriram Subramaniam, Haryadi Gunawi, Thanh Do, and Samer Al-Kiswany. I also have enjoyed interacting with other students: Nitin Agrawal, Ishani Ahuja, Leo Arulraj, Vijay Chidambaram, Tyler Harter, Jun He, Asim Kadav, Ao Ma, Joe Meehean, Sankaralingam Panneerselvam, Deepak Ramamurthi, Mohit Saxena, Laxman Visampalli, Zev Weiss, Suli Yang, Wei Zhang, and Yiying Zhang.

I am lucky to have so many friends at Madison. To name a few: Henry Chung, Guoliang Jin, Ji Liu, Jie Liu, Lanyue Lu, Ao Ma, Linhai Song, Chong Sun, Chong Sun, Wenfei Wu, Wentao Wu, Wei Zhang, and Yiying Zhang. I would like to especially thank my roommates, Guoliang Jin and Jie Liu, for accompanying me during these years. I also would also like to thank Yiying Zhang for being a wonderful and helping officemate. Of course, I am also grateful for the support from other friends who are not at Madison: Shaochen Huang, Qiang Li, Kun Qian, and Yuxiang Zheng.

Finally, I would like to thank my family back in China, especially my parents, for their unconditional love and support. When I am struggling with my research and sometimes with my life, they have always been supportive and encouraging. When I have even the smallest success, they are so happy that they almost want everyone in the world to know about it. Thank you, Baba and Mama, I dedicate this dissertation to you!

Abstract

COOPERATIVE DATA PROTECTION
Yupu Zhang

Storage systems employ various techniques to protect user data from hardware failures and software defects. These techniques, while effective in their own domains, fail to provide comprehensive protection. In this dissertation, we identify the problem of *isolated protection* in both local storage systems and cloud storage services, and propose *cooperative data protection* to address this problem.

In the first half of this dissertation (on local storage systems), we present a study of the effects of disk and memory corruption on ZFS, a modern commercial file system with numerous reliability mechanisms. Through careful and thorough fault injection, we show that ZFS is robust to a wide range of disk faults, but because of its isolated integrity checks that only cover on-disk data, it is less resilient to memory corruption, which can lead to corrupt data being returned to applications or system crashes.

To solve this problem, we introduce flexible end-to-end data integrity, which enables all components along the I/O path (e.g., page cache, file system) to handle checksums cooperatively. Each component is able to alter its protection scheme to meet the performance and reliability demands of the system. We apply this new concept to ZFS and build Zettabyte-Reliable ZFS (Z^2FS). Z^2FS provides dynamical tradeoffs between performance and protection and offers Zettabyte Reliability, which is at most one undetected corruption per Zettabyte of data read. We develop an analytical framework to evaluate reliability; the protection approaches in Z^2FS are built upon the foundations of the framework. For comparison, we implement a straight-forward End-to-End ZFS (E^2ZFS) with the same protection scheme for all components. Through analysis and experiment, we show that Z^2FS is able to achieve better overall performance than E^2ZFS, while still offering Zettabyte Reliability.

In the second half of this dissertation (on cloud storage services), we analyze

how reliable cloud-based synchronization services are in the face of local corruption and crashes. We perform fault injection experiments on several popular synchronization services and local file systems, and find that despite the excellent reliability that the cloud back-end provides, the loose coupling of these services and local file systems makes synchronized data more vulnerable than users might believe. Local corruption may be propagated to the cloud, polluting all copies on other devices, and a crash or untimely shutdown may lead to inconsistency between a local file and its cloud copy. Even without these failures, these services cannot provide causal consistency.

To solve this problem, we present ViewBox, an integrated synchronization service and local file system that provides freedom from data corruption and inconsistency. ViewBox detects these problems using ext4-cksum, a modified version of ext4, and recovers from them using a user-level daemon, cloud helper, to fetch correct data from the cloud. To provide a stable basis for recovery, ViewBox employs the view manager on top of ext4-cksum. The view manager creates and exposes views, consistent in-memory snapshots of the file system, which the synchronization client then uploads. Our experiments show that ViewBox detects and recovers from both corruption and inconsistency, while incurring minimal overhead.

Contents

Chapter 1

Introduction

People are generating tremendous amount of data everyday. By some estimates, there were 2.8 Zettabytes of data created in 2012, and the amount of data is expected to double by 2015 [115]. Not only governments and corporations, but also regular persons have contributed to this data explosion, by storing musics, photos, videos, and even email messages. Regardless of where data is placed, in a personal computer, an enterprise server, or the cloud, the underlying storage systems are responsible for preserving data correctly for a long time.

Unfortunately, storage systems are built upon imperfect hardware and software; hardware errors, crash, and software bugs all can corrupt data. Rare events in hard drives such as dropped writes or misdirected writes leave stale or corrupt data on disk [3, 23, 89, 92]. Bits in memory get flipped due to chip defects [63, 71, 97] or radiation [75, 133]. Untimely crash, if not handled properly, can lead to inconsistent data in the file system [37, 129]. Software bugs are also a source of data corruption, arising from low-level device drivers [111], system kernels [38, 47], and file systems [125, 126]. Even worse, design flaws are not uncommon and can lead to serious data loss or corruption [69].

As storage systems have evolved over the years, designers have developed various mechanisms to handle some of the aforementioned problems. Besides the built-in hardware ECC in hard drives, many modern file systems support high-level checksums to detect corruption [29, 91, 104], and some of them even provide replicas inside the file system to facilitate recovery [29]. Underneath the file system, RAID is widely used to provide redundancy for recovery [86]. Nowadays, backing up data to the cloud is also an appealing solution to preserve data [67]. In case of crash or power loss, file systems usually apply techniques such as journaling [116], soft updates [50], or copy-on-write [62], to provide metadata or data consistency.

However, these protection techniques, while effectively protecting data in their own domains, fail to provide comprehensive data protection for the entire system. As one example, many of the techniques are able to detect and recover from disk corruption, but they cannot protect in-memory data [131]. As another example, cloud storage services usually protect its data using checksums and tend to store multiple copies, but if the local file system exposes corrupt data, corruption may be propagated to the cloud, and thus pollute all the replicas [129].

All these failures occur due to *isolated protection* in storage systems, and we propose *cooperative data protection* to solve these problems. The goals of this dissertation are two-fold: first, to examine the threats to data protection in current storage systems due to isolated protection; second, to develop techniques that enable components in storage systems to work cooperatively to provide comprehensive data protection.

We address the goals of this dissertation in two aspects: local storage systems and cloud storage services. For local storage systems, we first analyze the impact of disk corruption and memory corruption on a modern file system, ZFS, and show that memory corruption is largely ignored and poses great harm to data integrity [131]. Then, we build Z^{2}FS, which embraces a new protection scheme called flexible end-to-end data integrity and provides protection to both in-memory and on-disk data without sacrificing much performance [130]. For cloud storage services, especially cloud-based file synchronization services, we first examine how disk corruption and system crashes could lead to the propagation of bad data across all synchronized devices [129]. Then we develop ViewBox, an integrated file system and synchronization service that provides data integrity, crash consistency, and even causal consistency for both local and cloud data [127]. The following sections elaborate on each of these contributions of the dissertation.

1.1 Cooperative Data Protection in Local Storage

One of the primary challenges faced by storage systems is to protect data despite the presence of imperfect components in the storage stack. In the first part of the dissertation, we focus on data protection in local storage systems. Specifically, we first use ZFS as an example and show that its isolated integrity check does not protect data in memory. Then, we propose and apply flexible end-to-end data integrity to ZFS to achieve cooperative data protection.

1.1.1 Data Protection Analysis of ZFS

File and storage systems have evolved various techniques to handle corruption. Different types of checksums can be used to detect when corruption occurs [25, 29, 104, 109], and redundancy, likely in mirrored or parity-based form [86], can be applied to recover from corruption. While such techniques are not foolproof [69], they clearly have made file systems more robust to disk corruption.

Unfortunately, the effects of *memory corruption* on data integrity have been largely ignored in file system design. Hardware-based memory corruption occurs as both transient *soft errors* and repeatable *hard errors* due to a variety of radiation mechanisms [27, 75, 133], and recent studies have confirmed their presence in modern systems [72, 84, 97]. Software can also cause memory corruption; bugs can lead to "wild writes" into random memory contents [34], thus polluting memory; studies confirm the presence of software-induced memory corruptions in operating systems [2, 5, 14, 124].

To study how robust modern file systems are to disk and memory corruption, we analyze a state-of-the-art file system, ZFS [29], by performing fault injection tests representative of realistic disk and memory corruptions. We choose ZFS for our analysis because it is a modern and mature commercial file system with numerous robustness features, including end-to-end checksums, data replication, and transactional updates; the result, according to the designers, is "provable data integrity" [29].

In our analysis, we find that ZFS is indeed robust to a wide range of disk corruptions, thus partially confirming that many of its design goals have been met. However, we also find that ZFS often fails to maintain data integrity in the face of memory corruption. In many cases, ZFS is either unable to detect the corruption, returns bad data to the user, or simply crashes.

1.1.2 Z^{2}FS: Zettabyte Reliability with Flexible End-to-end Data Integrity

A more comprehensive approach to data protection should embrace the "end to end" philosophy [94]. In this approach, checksums are generated by an application and percolate through the entire storage system. When reading data, the application can check whether the calculated checksum matches the stored checksum, thus improving data integrity.

Unfortunately, the straight-forward end-to-end approach has two drawbacks. The first is *performance*; depending on the cost of checksum calculation, performance can suffer when repeatedly accessing data from the in-memory page cache.

The second is *timeliness*; if a data block is corrupted in memory before being flushed to disk, the corruption can only be detected when it is later read by an application, which is likely too late to recover from the corruption.

To address these issues, we propose a concept called *flexible end-to-end data integrity*. We argue that it is not necessary for all components on the I/O path to use the same checksum. By carefully choosing a different checksum for each component (and perhaps altering said checksum over time), the system can deliver better performance while still maintaining a high level of protection. By enabling all components to handle checksums cooperatively, the system can detect and recover from corruption in time.

To explore this flexible approach, we design and implement flexible end-to-end data integrity within ZFS, resulting in a new variant which we call Zettabyte-reliable ZFS (Z^2FS). Z^2FS exposes checksums to the application, and passes checksums through the page cache down to the disk system, thus enabling end-to-end verification. Z^2FS uses two techniques to provide flexible data protection. The first is *checksum chaining*, which carefully orders the generation of new checksum and the verification of old checksum such that there is no vulnerability window for data when it crosses domains (e.g., when moving from a stronger on-disk checksum to a weaker but more performant in-memory one). The second is *checksum switching*, which enables a component (e.g., memory) to switch the checksum it is using dynamically, thus preserving a high level of reliability for blocks that remain resident for extended periods of time. For comparison, we also develop End-to-End ZFS (E^2ZFS), which embraces the straight-forward end-to-end protection and uses only one type of checksum for both the page cache and disk.

Underlying Z^2FS is an analytical framework that enables us to understand reliability of storage systems against data corruption. The framework takes models of devices and checksums used in a storage system as input, and calculates the probability of undetected data corruption when reading a data block from the system as a reliability metric. We define *Zettabyte Reliability*, one undetected corruption per Zettabyte read, as a reliability goal of storage systems. Guided by the reliability goal, we use the framework to provide rationale behind flexible end-to-end data integrity.

Through fault injection experiments, we show that Z^2FS is able to detect and recover from corruption that occurs to a block in memory before it is flushed to disk in the write path. Using both controlled benchmarks as well as real-world traces, we demonstrate that Z^2FS is able to meet or exceed the performance of E^2ZFS while still providing Zettabyte reliability. Especially for workloads dominated by warm reads, Z^2FS ourperforms E^2ZFS by up to 17%.

1.2 Cooperative Data Protection across Local and Cloud Storage

With the emergence of cloud storage, especially in the form of cloud-based file synchronization services, local file systems are now connected to the cloud, and user data becomes synchronized and replicated on multiple devices. These services are great additions to local file systems and provide better protection for user data, but the loose coupling of these services and the file systems actually puts data in danger in various ways. In the second part of the dissertation, we focus on new challenges to data protection across local and cloud storage. We first conduct an analysis of various file synchronization services and show how they propagate corrupt and inconsistent data to the cloud. Then, we build ViewBox, an integrated synchronization service and file system in which the underlying file system works cooperatively with the file synchronization service to provide comprehensive data protection.

1.2.1 Data Protection Analysis of Cloud Storage Services

File synchronization services occupy a unique design point between distributed file systems, like NFS [95] or Coda [68], and file backup services, like Mozy [8] or Data Domain [132]. Like the former, file synchronization services provide a means for users to access their files on any machine connected to the service. Like the latter, however, file synchronization services propagate local changes asynchronously, and often provide a means to restore previous versions of files. Furthermore, they are only loosely integrated with the file system, allowing them to be portable across a wide range of devices.

While the automatic propagation of files as they are modified is no doubt key to these services' success, the perceived reliability and consistency they provide is also instrumental to their appeal. The Dropbox tour goes as far as to state that "none of your stuff will ever be lost" [44]. Unfortunately, the loose coupling of cloud synchronization services with the underlying file system gives the lie to this claim. While the data stored remotely is generally robust, local client software is unable to distinguish between deliberate modifications and unintentional errors, potentially causing corruption to automatically propagate to all machines associated with a user. Thus, despite the presence of multiple redundant copies, synchronization destroys the user's data.

To understand this "false sense of security", we perform fault injections experiments on several popular cloud-based synchronization services. We first examine how these services can silently propagate data corruption to all synchronized de-

vices, and then show how these services cannot guarantee data consistency with the underlying file system after a crash. Furthermore, we show that a stronger level of inconsistency, causal inconsistency, may occur and thus cause even more harm to both local and cloud data.

1.2.2 ViewBox: Integrating File Systems with Cloud Storage Services

The analysis reveals that the root cause of data protection failures is the loose coupling of synchronization services and local file systems, and they take equal responsibilities for these failures. Therefore, we develop ViewBox, a system that integrates local file system and cloud-based synchronization services to provide better data integrity, crash consistency, and recoverability.

ViewBox synchronizes data between the local machine and the cloud through *views*, in-memory snapshots of the local synchronized folder. ViewBox relies on three primary components to guarantee the correctness of views: ext4-cksum, the view manager, and the cloud helper. Ext4-cksum serves as the local file system, which is able to detect corrupt and inconsistent data through data checksumming. Atop ext4-cksum, we place the view manager, a file system extension that creates views and exposes views to the synchronization client. The view manager provides consistency through *cloud journaling* by creating views at file-system epochs and uploading views to the cloud. To reduce the overhead of maintaining views, the view manager employs *incremental snapshotting* by keeping only deltas (changed data) in memory since the last view. Finally, in case of corruption or crash, ViewBox uses an independent user-space daemon, the cloud helper, to interact with the server-backend and utilize the views on the cloud to restore the system to a correct state.

We build ViewBox with two file synchronization services: Dropbox [44], one of the most popular synchronization services to date, and Seafile [99], an open source synchronization service based on GIT [52]. Through reliability experiments, we demonstrate that ViewBox detects and recovers from local data corruption, thus preventing the corruption's propagation. We also show that upon a crash, ViewBox successfully rolls back the local file system state to a previously uploaded view, restoring it to a causally consistent image. By comparing ViewBox to Dropbox or Seafile running atop unmodified ext4, we find that ViewBox incurs less than 5% overhead across a set of workloads. In some cases, ViewBox even improves the synchronization time by 30%.

1.3 Summary of Contributions / Outline

Below is a summary of the contributions of the dissertation, which also serves as an outline for the rest of the dissertation:

- **Threats to Data Protection:** Chapter 2 provides background on various threats to data protection in existing storage systems: disk corruption, memory corruption, and crashes.
- **Cooperative Data Protection in Local Storage:** In Chapter 3, we present an empirical analysis of the reliability of ZFS in the face of disk and memory corruption. Then, in Chapter 4, we propose the concept of flexible end-to-end data integrity, introduce an analytical framework to provide the rationale behind the concept, and implement Z^2FS, which provides comprehensive data protection (from both disk and memory corruption). The concept, framework, and techniques used in implementing Z^2FS, all together demonstrate a holistic way to think about the performance-reliability tradeoff in storage systems, which is the first major contribution of the dissertation.
- **Cooperative Data Protection across Local and Cloud Storage:** Chapter 5 presents an analysis of data protection failures (focusing on disk corruption and crash) when file synchronization services are running on top of current file systems. Chapter 6 describes our solution to the found problems, ViewBox, an integrated file system and synchronization services that synchronizes data based on file-system views. Both the analysis and the solution serve as the second major contribution of this dissertation.
- **Related Work:** Chapter 7 summarizes previous research efforts on protecting data in storage systems.
- **Conclusion and Future Work:** Chapter 8 concludes this dissertation, first summarizing our work and highlighting the lessons learned, and then discussing various avenues for future work that arise from our research.

Chapter 2

Threats to Data Protection

This chapter provides the motivation for the dissertation by describing various threats to data protection in storage systems. Specifically, we focus on two types of threats, data corruption and data inconsistency. Data corruption occurs mostly due to hardware failures and software bugs, and we describe why it happens, how frequently it occurs, and how systems try to deal with it in Section 2.1. Data inconsistency, on the other hand, usually results from the file system's improper handling during an untimely system crash or reboot. We discuss how file systems provide consistency and why data consistency is not always guaranteed in Section 2.2.

2.1 Data Corruption

We now discuss data corruption in detail. Although it can occur at any place in a storage system, we only focus on corruption on disk and in memory, because both are the major media for long-term data storage and accesses.

2.1.1 Disk Corruption

We define disk corruption as a state when any data accessed from disk does not have the expected contents due to some problem in the storage stack. This is different from latent sector errors, not-ready-condition errors and recovered errors [22] in disk drives, where there is an explicit notification from the drive about the error condition.

Why It Happens

Disk corruption happens due to many reasons originating at different layers of the storage stack. Errors in the magnetic media lead to the problem of "bit-rot" where the magnetic properties of a single bit or few bits are damaged. Spikes in power, erratic arm movements, and scratches in media can also cause corruption in disk blocks [19, 98, 113]. On-disk ECC catches many (but not all) of these corruption.

Errors are also induced due to bugs in complex drive firmware (modern drives contain hundreds of thousands of lines of firmware code [89]). Some reported firmware problems include a misdirected write where the firmware accidentally writes to the wrong location [118] or a lost write (or phantom write) where the disk reports a write as completed when in fact it never reaches the disk [109]. Bus controllers have also been found to incorrectly report disk requests as complete or to corrupt data [55, 117].

Finally, software bugs in operating systems are also potential sources of corruption. Buggy device drivers can issue disk requests with bad parameters or data [38, 47, 111]. Software bugs in the file system itself can cause incorrect data to be written to disk.

How Frequently It happens

Disk corruption are prevalent across a broad range of modern drives. There is much anecdotal evidence of corruption in hard disks [25, 109, 118]. In 2008, in a study of 1.53 million disk drives over 41 months [23], Bairavasundaram et al. show that more than 400,000 blocks had checksum mismatches, 8% of which were discovered during RAID reconstruction, creating the possibility of real data loss. They also found that nearline disks develop checksum mismatches an order of magnitude more often than enterprise class disk drives.

How to Handle It

Systems use a number of techniques to handle disk corruption. We discuss some of the most widely used techniques along with their limitations.

Checksums: Checksums are small pieces of data computed over data blocks with a specific function and are used to verify data integrity. For on-disk data integrity, checksums are stored or updated on disk during write operations and read back to verify the block or sector contents during reads.

Many storage systems have used checksums for on-disk data integrity, such as Tandem NonStop [25] and NetApp Data ONTAP [109]. Similar checksumming techniques have also been used in file systems [29, 91].

However, Krioukov et al. show that checksumming, if not carefully integrated into the storage system, can fail to protect against complex failures such as lost writes and misdirected writes [69]. Further, checksumming does not protect against corruption that happens due to bugs in software, typically in large code bases [38, 125].

Redundancy: Redundancy in on-disk structures also helps to detect and, in some cases, recover from disk corruption. For example, some B-Tree file systems such as ReiserFS [30] store page-level information in each internal page in the B-Tree. Thus, a corrupt pointer that does not connect pages in adjacent levels is caught by checking this page-level information. Similarly, ext2 [32] and ext3 [116] use redundant copies of superblock and group descriptors to recover from corruption.

However, it has been shown that many of these file systems still sometimes fail to detect corruption, leading to greater problems [89]. Further, Gunawi et al. show instances where ext2/ext3 file system checkers fail to use available redundant information for recovery [57].

RAID: Another popular technique is to use a RAID storage system [86] underneath the file system. However, RAID is designed to tolerate the loss of a certain number of disks or blocks (e.g., RAID-5 tolerates one, and RAID-6 two) and it may not be possible with RAID alone to accurately identify the block (in a stripe) that is corrupted. Secondly, some RAID systems have been shown to have flaws where a single block loss leads to data loss or silent corruption [69]. Finally, not all systems incorporate multiple disks, which limits the applicability of RAID.

2.1.2 Memory Corruption

We define memory corruption as the state when the contents accessed from the main memory have one or more bits changed from the expected value (from a previous store to the location). From the software perspective, it may not be possible to distinguish memory corruption from disk corruption on a read of a disk block.

Why It Happens

Errors in the memory chip are one source of memory corruption. Memory errors can be classified as *soft errors* which randomly flip bits in RAM without leaving any permanent damage, and *hard errors* which corrupt bits in a repeatable manner due to physical damage.

Researchers have discovered radiation mechanisms that cause errors in semiconductor devices at terrestrial altitudes. Nearly three decades ago, May and Woods found that if an alpha particle penetrates the die surface, it can cause a random,

single-bit error [75]. Zeigler and Lanford found that cosmic rays can also disrupt electronic circuits [133]. More recent studies and measurements confirm the effect of atmospheric neutrons causing single event upsets (SEU) in memories [83, 84].

Memory corruption can also happen due to software bugs. The use of unsafe languages like C and C++ makes software vulnerable to bugs such as dangling pointers, buffer overflows and heap corruption [28], which can result in seemingly random memory corruption.

How Frequently It Happens

Early studies and measurements on memory errors provided evidence of soft errors. Data collected from a vast storehouse of data at IBM over a 15-year period [84] confirmed the presence of errors in RAM and that the upset rates increase with elevation, indicating atmospheric neutrons as the likely cause.

In 2009, a measurement-based study of memory errors in a large fleet of commodity servers over a period of 2.5 years [97], Schroeder et al. observe DRAM error rates that are orders of magnitude higher than previously reported, with 25,000 to 70,000 FIT per Mbit (1 FIT equals 1 failure in 10^9 device hours). They also find that more than 8% of the DIMMs they examined (from multiple vendors, with varying capacities and technologies) were affected by bit errors each year. Finally, they also provide strong evidence that memory errors are dominated by hard errors, rather than soft errors.

Another study [72] of production systems including 300 machines for a multi-month period found 2 cases of suspected soft errors and 9 cases of hard errors suggesting the commonness of hard memory faults.

Besides hardware errors, software bugs that lead to memory corruption are widely extant. Reports from the Linux Kernel Bugzilla Database [5], USCERT Vulnerabilities Notes Database [14], CERT/CC advisories [2], as well as other anecdotal evidence [34] show cases of memory corruption happening due to software bugs.

How to Handle It

Systems use both hardware and software techniques to handle memory corruption. Below, we discuss the most relevant hardware and software techniques.

ECC: Traditionally, memory systems have employed Error Correction Codes [35] to correct memory errors. Unfortunately, ECC is unable to address all soft-error problems. Studies found that the most commonly-used ECC algorithms called SEC/DED (Single Error Correct/Double Error Detect) can recover from only 94%

of the errors in DRAMs [48]. Further, many consumer systems do not use ECC protection in order to reduce cost [59].

More sophisticated techniques like Chipkill [64] have been proposed to withstand multi-bit failure in DRAMs. However, such techniques are expensive and have been restricted to proprietary server systems, leaving the problem of memory corruption open in commodity systems.

Programming models and tools: Another approach to deal with memory errors is to use recoverable programming models [80] at different levels (firmware, operating system, and applications). However, such techniques require support from hardware to detect memory corruption. Further, a holistic change in software is required to provide recovery solution at various levels.

Much effort has also gone into detecting software bugs that cause memory corruption. Tools such as metal [58] and CSSV [42] apply static analysis to detect memory corruption. Others such as Purify [61] and SafeMem [90] use dynamic monitoring to detect memory corruption at runtime. However, as discussed previously, software-induced memory corruption still remains a problem.

2.2 Data Inconsistency

The problem of data inconsistency usually occurs due to file system failing to provide strong consistency guarantee upon a crash. File systems maintains various metadata structures to organize data. Performing a single file system operation, such as write(), usually involves changes to several metadata structures. For example, appending a block to a file in ext3 requires at least three blocks to be written to disk: a data bitmap block, an inode block, and the data block. In order to correctly apply such an operation to the on-disk file system image, all these blocks must be written to disk as a whole. However, when crash occurs, it is possible that some of the changes do not make to the disk. For instance, if the data block is not written, the file would point to garbage data, resulting in *data inconsistency*. If the data bitmap block is not written, the actual status of the data block (used by the inode) does not match the bitmap (free), which leads to *metadata inconsistency*.

File system developers have been using several techniques to address the consistency problem. One simple approach is to let the inconsistency occur and then use a tool, usually called file system checker (fsck) [76], to fix the inconsistency. This approach can fix metadata inconsistency in most cases, but it cannot, for example, detect the data inconsistency case mentioned above. Therefore, many file systems have built-in mechanism to prevent inconsistency in runtime, and the most common technique is journaling. Journaling, or write-ahead logging, provides con-

sistency by grouping multiple updates into transactions, which are first written to a circular log and then later checkpointed to their fixed location in the file system. Journaling is quite popular, seeing use in ext3 [116], ext4 [73], XFS [110], HFS+ [21], and NTFS [79]. Recording all data and metadata in the log can provide data consistency, but doing so doubles all write traffic in the system. Thus, normally, these file systems only journal metadata, which can lead to inconsistencies in file data upon recovery, even if the file system carefully orders its data and metadata writes (as in ext4's ordered mode, for instance).

Data inconsistency can be avoided entirely using copy-on-write, but it is an infrequently used solution. Copy-on-write never overwrites data or metadata in place; thus, if a crash occurs mid-update, the original state will still exist on disk, providing a consistent point for recovery. Implementing copy-on-write involves substantial complexity, however, and only recent file-systems, like ZFS [29] and btrfs [91], support it for personal use.

2.3 Summary

Modern storage systems are facing great challenges in protecting data. Disk errors, memory bit flips, and software bugs can all corrupt data. The combination of untimely crash and imperfect crash handling of file system may lead to data inconsistency. We have presented some existing mechanisms to deal with these problems, but unfortunately they are still separated techniques and cannot provide comprehensive data protection. In the following chapters we will show why they fail to protect data in local file systems as well as cloud storage services, and explore new cooperative techniques to maintain data integrity and consistency.

Chapter 3

Data Protection Analysis of Local File Systems

Disk corruption is one of the primary sources for unreliability in data storage. As file systems have evolved over the years, designers have focused on this problem and devised techniques to deal with it [29, 86, 104]. Unfortunately, memory corruption has been ignored and poses a growing threat to data integrity. As discussed in Section 2.1.2, recent studies measured increasing memory error rate due to hardware faults, and various bug reports show the occurrence of memory corruption due to software bugs.

The problem of memory corruption is critical for file systems that cache a great deal of data in memory for performance. Almost all modern file systems use a page cache or buffer cache to store copies of on-disk data and metadata in memory. Moreover, frequently-accessed data and important metadata may be cached in memory for long periods of time, making them more susceptible to memory corruption.

In this chapter, we ask: how robust are modern local file systems to disk and memory corruptions? To answer this query, we perform a series of fault injection experiments on ZFS to study how it responds to disk and memory corruptions. Before we go into details about the experiments, we first we provide some background on ZFS in Section 3.1. Then, we present our analysis of data protection in ZFS with disk and memory corruptions in Section 3.2 and Section 3.3, respectively. Finally, Section 3.4 gives an analysis of the probabilities of different failure scenarios in ZFS due to memory errors.

3.1 Background

ZFS is a state-of-the-art file system from Sun (now Oracle) which takes a unified approach to data management. ZFS provides data integrity, transactional consistency, scalability, and a multitude of useful features such as snapshots, copy-on-write clones, and simple administration [29]. In this section, we first present a high-level overview of ZFS, focusing on the reliability mechanisms. Then, we discuss the disk layout of ZFS in detail and illustrate how ZFS organizes metadata and data through a on-disk walkthrough. Finally, we briefly discuss in-memory data structures.

3.1.1 ZFS Overview

ZFS claims to provide provable data integrity by using techniques like checksums, replication, and transactional updates. Further, the use of a pooled storage in ZFS lends it additional RAID-like reliability features. In the words of the designers, ZFS is the "The Last Word in File Systems." We now describe the reliability mechanisms in ZFS.

Checksums for data integrity checking: ZFS maintains data integrity by using checksums for on-disk blocks. The checksums are kept separate from the corresponding blocks by storing them in the parent blocks. ZFS provides for these parental checksums of blocks by using a generic block pointer structure to address all on-disk blocks.

The block pointer structure contains the checksum of the block it references. Before using a block, ZFS calculates its checksum and verifies it against the stored checksum in the block pointer. The checksum hierarchy forms a self-validating Merkle tree [78]. With this mechanism, ZFS is able to detect silent data corruption, such as bit rot, phantom writes, and misdirected reads and writes.

Replication for data recovery: Besides using RAID techniques (described below), ZFS provides for recovery from disk corruption by keeping replicas of certain "important" on-disk blocks. Each block pointer contains pointers to up to three copies of the block being referenced. By default ZFS stores multiple copies for metadata (three copies for pool metadata and two copies for file system metadata) and one copy for data. Upon detecting a corruption due to checksum mismatch, ZFS uses a redundant copy with a correctly-matching checksum.

COW transactions for atomic updates: ZFS maintains data consistency in the event of system crashes by using a copy-on-write transactional update model. ZFS manages all metadata and data as objects. Updates to all objects are grouped together as a transaction group. To commit a transaction group to disk, new copies

are created for all the modified blocks (in a Merkle tree). The root of this tree (the *uberblock*) is updated atomically, thus maintaining an always-consistent disk image. In effect, the copy-on-write transactions along with block checksums (in a Merkle tree) preclude the need for journaling [120], though ZFS occasionally uses a write-ahead log for performance reasons.

Storage pools for additional reliability: ZFS provides additional reliability by enabling RAID-like configuration for devices using a common storage pool for all zfs instances. ZFS presents physical storage to file systems in the form of a storage pool (called *zpool*). A storage pool is made up of *virtual devices* (vdev). A virtual device could be a physical device (e.g., disks) or a logical device (e.g., a mirror that is constructed by two disks). This storage pool can be used to provide additional reliability by using devices as RAID arrays. ZFS provides automatic repairs in mirrored configurations and provides a disk scrubbing facility to detect latent sector errors.

3.1.2 ZFS On-disk Organization

ZFS organizes its metadata and data into a two level architecture, as shown in Figure 3.1. The zfs level contains on-disk structures that are used to represent a zfs instance, such as a file system, a snapshot, or a clone. The zpool level maintains data structures that keep track of all file system instances and their relationship. We now discuss some of these basic on-disk structures and their usage in ZFS.

Basic Structures

Block pointers: A block pointer is the basic structure in ZFS for addressing a block on disk and connecting different structures. It provides a generic mechanism to keep parental checksums and replicas of on-disk blocks. Figure 3.2 shows the block pointer used by ZFS. As shown, the block pointer contains up to three block addresses, called DVAs (*data virtual addresses*), each pointing to a different block having the same contents. These are referred to as *ditto blocks*. The number of DVAs varies depending on the importance of the block. The current policy in ZFS is that there is one DVA for user data, two DVAs for file system metadata, and three DVAs for global metadata across all file system instances in the pool [81]. As discussed earlier, the block pointer also contains a single copy of the checksum of the block being pointed to.

Objects: All blocks on disk are organized in objects. Physically, an object is represented on disk by a structure called `dnode_phys_t` (hereafter referred to as *dnode*). A dnode contains an array of up to three block pointers, each of which points to

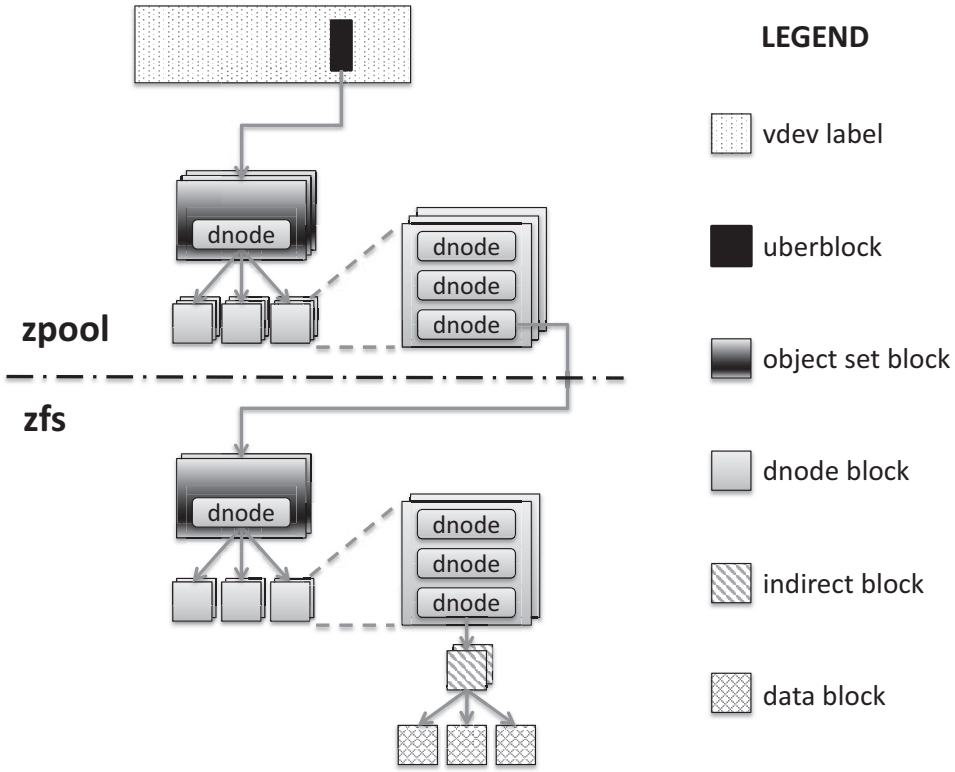


Figure 3.1: **ZFS Two-level Layout** *The figure shows the two-level layout of ZFS on-disk structures.*

either a leaf block (e.g., a data block) or an indirect block (full of block pointers). These blocks pointed to by the dnode form a block tree. A dnode also contains a bonus buffer at the end, which stores an object-specific data structure for different types of objects. For example, a dnode of a file object contains a structure called `znode_phys_t` (*znode*) in the bonus buffer, which stores file attributes such as access time, file mode and size of the file. The dnode then points to a block tree with data blocks at the leaf level, as shown in Figure 3.1.

Object sets: Object sets are used in ZFS to group related objects. An example of a object set is a file system, which contains file objects and directory objects belonging to this file system. An object set is represented by a structure called `objset_phys_t`, which consists of a meta dnode and a ZIL (ZFS Intent Log) header. The meta dnode points to a group of dnode blocks; dnodes representing the objects in this object set are stored in these dnode blocks. The object de-

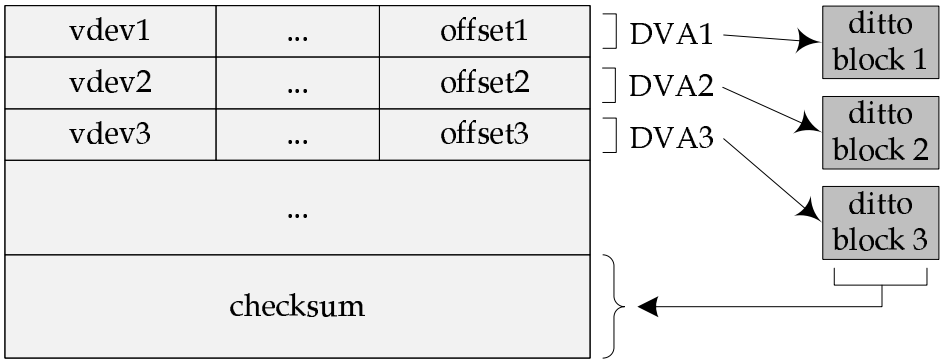


Figure 3.2: **Block pointer** *The figure shows how the block pointer structure points to (up to) three copies of a block (ditto blocks), and keeps a single checksum.*

Level	Object Name	Simplified Explanation
zpool	MOS dnode	A dnode object that contains dnode blocks, which store dnodes representing pool-level objects.
	Object directory	A ZAP object whose blocks contain name-value pairs referencing further objects in the MOS object set.
	Dataset	It represents an object set (e.g., a file system) and tracks its relationships with other object sets (e.g., snapshots and clones).
	Dataset directory	It maintains an active dataset object along with its child datasets. It has a reference to a dataset child map object. It also maintains properties such as quotas for all datasets in this dataset directory.
	Dataset child map	A ZAP object whose blocks hold name-value pairs referencing child dataset directories.
zfs	FS dnode	A dnode object that contains dnode blocks, which store dnodes representing filesystem-level objects.
	Master node	A ZAP object whose blocks contain name-value pairs referencing further objects in this file system.
	File	An object whose blocks contain file data.
	Directory	A ZAP object whose blocks contain name-value pairs referencing files and directories inside this directory.

Table 3.1: **Summary of ZFS objects visited** *The table presents a summary of all ZFS objects visited in the walkthrough, along with a simplified explanation. Note that ZAP stands for ZFS Attribute Processor. A ZAP object is used to store name-value pairs.*

scribed by the meta dnode is called “dnode object”. The ZIL header points to a list of blocks, which holds transaction records for ZFS’s logging mechanism. The `objset_phys_t` structure is stored in an *objset block*.

Datasets: An object set is eventually encapsulated by a zpool-level object called dataset. A dataset could be a file system, a clone, or a snapshot. A dataset contains statistics such as the space consumption of an object set, and tracks its relationship with other related datasets. For example, a file system dataset maintains the inter-dependency between the file system and its snapshots and clones. A dataset is represented by a dnode with a `dsl_dataset_phys_t` structure in the bonus field. The dnode itself does not point to the objset block; it is the `dsl_dataset_phys_t` structure that contains a block pointer referencing the objset block.
Uberblock: As shown in Figure 3.1, all zpool-level objects form another object set and the corresponding objset block is pointed to by a root block pointer in an *uberblock*. An uberblock (similar to a superblock) is used to provide access to the current pool data and verify its integrity. The uberblock is self-checksummed and updated atomically.
Vdev label: Each physical vdev is labeled with a *vdev label* that describes this device and other related virtual devices. Four copies of the label are stored in each physical vdev to provide redundancy and a two-stage update mechanism is used to guarantee that there is always a valid vdev label in the device [108]. Every vdev label contains an array of uberblocks; updating an uberblock involves writing the new uberblock to the next entry in the array (in a round robin fashion) and mark the new entry the active uberblock. Therefore, if a crash occurs during the update, ZFS will always fall back to the previous uberblock, thus guaranteeing consistency.

On-disk Layout

Next, we present more details on ZFS on-disk layout. This overview will help the reader to understand the range of our fault injection experiments presented in later sections. A complete description of ZFS on-disk structures can be found elsewhere [108].

For the purpose of illustration, we demonstrate the steps that ZFS takes to locate a file system and to locate file data in it in a simple storage pool. Figure 3.3 shows the on-disk layout of the simplified pool with a sample file system called “myfs”, along with the sequence of objects and blocks accessed by ZFS. A summary of all visited objects is described in Table 3.1. Note that we skip the details of how in-memory structures are set up and assume that data and metadata are not cached in memory to begin with.

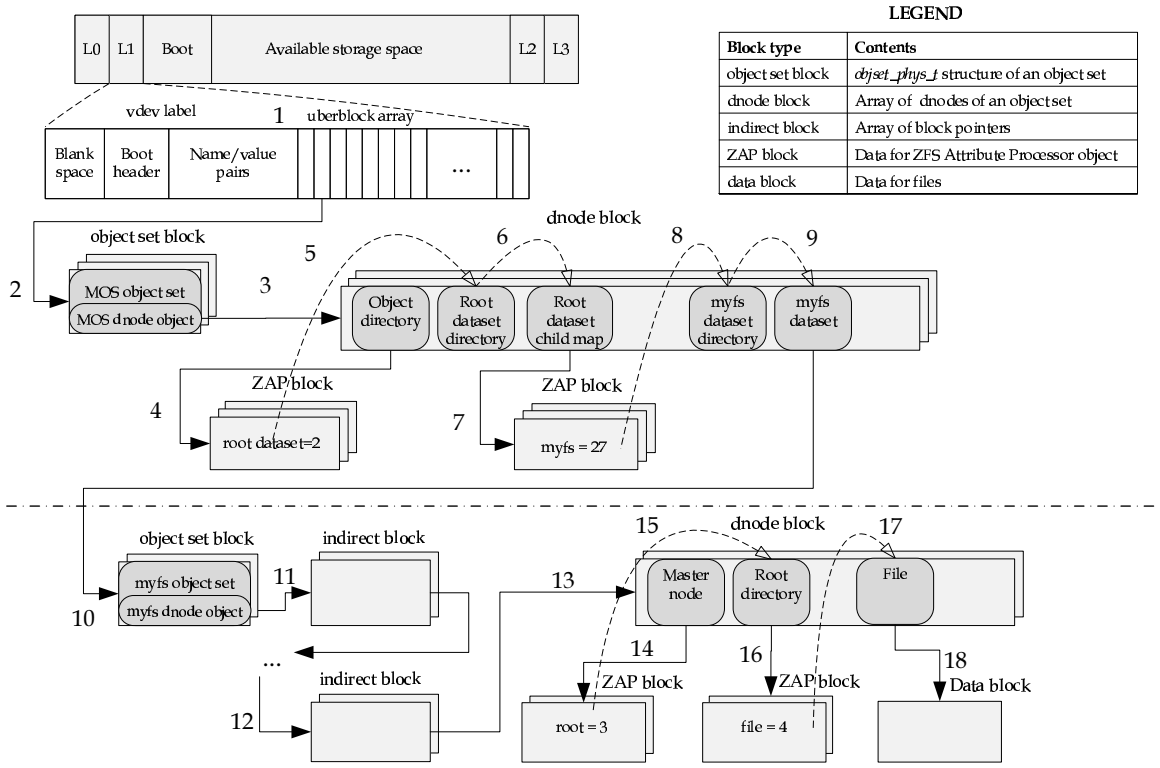


Figure 3.3: **ZFS On-disk Walk** *The figure illustrates a walkthrough of on-disk structures of ZFS to locate a data block in a file system "myfs". Zpool contains a sample file system named "myfs". All data structures are shown by rounded boxes, and blocks are shown by rectangular boxes. Solid arrows point to allocated blocks and dotted arrows represent references to objects inside blocks. The legend at the top shows the types of on-disk blocks and their contents.*

As shown in the figure, four copies of vdev labels are located at fixed locations on the disk (two each at the start and end). The active uberblock is found in any one of the labels (step 1). The uberblock points to a meta object set (MOS) (step 2), which is an object set holding pool-wide information for describing and managing relationships between various file system instances. Since MOS is pool-wide metadata, there are three copies of the block containing it.

A special object in MOS called the object directory is used to keep track of further zpool-level objects (step 3 and 4). The object directory contains references (object numbers) to various other objects in the object set. One of these references is the root dataset directory (step 5). A dataset directory encapsulates a group of related datasets and maintains their common properties, such as quota, block size, checksum algorithm, etc. Every zfs in zpool has a corresponding dataset directory. A dataset directory always has a single "active dataset", which represents the active zfs instance; other datasets are its snapshots, clones, etc. Therefore, the root dataset directory represents the root file system in the pool and it is used to access all child dataset directories.

The root dataset directory points to a dataset child map object (step 6), which contains references to all child dataset directories, including "myfs" (step 7). Finally, the dataset directory of "myfs" is found (step 8) and the active dataset of the directory points to the current "myfs" file system (step 9). The object set pointed to by this dataset contains further file-system specific metadata structures (step 10). Since the objset block is zfs-level metadata, ZFS keeps two copies of the block. The "myfs" object set further points to several layers of indirect blocks which eventually lead to a large array of dnodes describing file system objects (step 11-13). Since all these blocks are also file-system specific metadata, there are two copies of all the indirect blocks as well as the dnode blocks at the leaf level.

There is a special object called master node for each file system. It contains references to the root directory of a file system (step 14). The root directory is then traversed to find further child directories and files in the "myfs" file system (step 15-17). Finally, the file objects contain the block pointers to their corresponding data blocks (step 18).

3.1.3 ZFS In-memory Structures

ZFS in-memory structures can be classified into two categories: those that exist in the page cache and those that are in memory outside of the page cache; for convenience we call the latter *in-heap* structures. Whenever a disk block is accessed, it is loaded into memory. Disk blocks containing data and metadata are cached in the ARC page cache [77], and stay there until evicted. Data blocks are stored only in

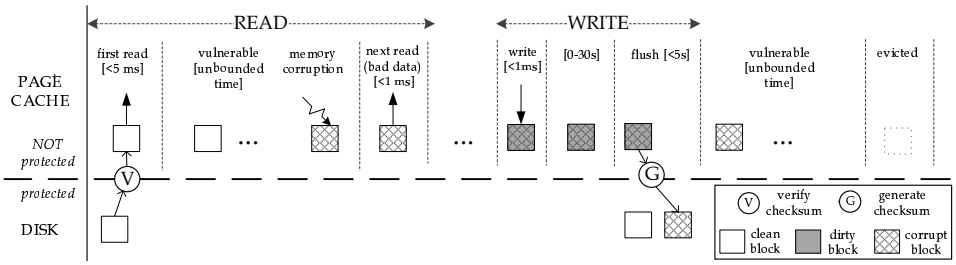


Figure 3.4: **Lifecycle of a block** *This figure illustrates one example of the lifecycle of a block. The left half represents the read timeline and the right half represents the write timeline. The black dotted line is a protection boundary, below which a block is protected by the checksum, otherwise unprotected.*

the page cache, while most metadata structures are stored in both the page cache (as copies of on-disk structures) and the heap. Note that block pointers inside indirect blocks are also metadata, but they only reside in the page cache. Uberblocks and vdev labels, on the other hand, only stay in the heap.

To help the reader understand the vulnerability of ZFS to memory corruptions discussed in later sections, Figure 3.4 illustrates one example of the lifecycle of a block (i.e., how a block is read from and written asynchronously to disk). To simplify the explanation, we consider a pair of blocks in which the target block to be read or written is pointed to by a block pointer contained in the parental block. The target block could be a data block or a metadata block. The parental block could be an indirect block (full of block pointers), a dnode block (array of dnodes, each of which contains block pointers), or an object set block (a dnode is embedded in it). The user of the block could be a user-level application or ZFS itself. Note that only the target block is shown in the figure.

At first, the target block is read from disk to memory. For read, there are two scenarios, as shown in the left half of Figure 3.4. On the first read of a target block, it is read from the disk and immediately verified against the checksum stored in the block pointer in the parental block. Then the target block is returned to the user. On a subsequent read of a block already in the page cache, the read request gets the cached block from the page cache directly, without verifying the checksum.

In both cases, after the read, the target block stays in the page cache until evicted. The block remains in the page cache for an unbounded interval of time depending on many factors such as the workload and the cache replacement policy.

After some time, the block is updated. The write timeline is illustrated in the right half of Figure 3.4. All updates are first done in the page cache and then flushed to disk. Thus before the updates occur, the target block is either in the page cache

already or just loaded to the page cache from disk. After the write, the updated block stays in the page cache for at most 30 seconds and then it is flushed to disk.

During the flush, a new physical block is allocated and a new checksum is generated for the dirty target block. The new disk address and checksum are then written to the block pointer contained in the parental block, thus making it dirty. After the target block is written to the disk, the flush procedure continues to allocate a new block and calculate a new checksum for the parental block, which in turn dirties its subsequent parental block. Following the updates of block pointers along the tree (solid arrows in Figure 3.3), it finally reaches the uberblock which is self-checksummed. After the flush, the target block is kept in the page cache until it is evicted.

3.2 On-disk Data Integrity in ZFS

In this section, we analyze the robustness of ZFS against disk corruptions. Our aim is to find whether ZFS can maintain data integrity under a variety of disk corruption scenarios. Specifically, we wish to find if ZFS can detect and recover from all disk corruptions in data and metadata and how ZFS reacts to multiple block corruptions at the same time. Through experiments, we find that ZFS is able to detect all and recover from most disk corruptions.

3.2.1 Methodology

Now we present the methodology of our reliability analysis of ZFS against disk corruptions. We discuss our fault injection framework first and then present our test procedures and workloads.

Fault Injection Framework

Our experiments are performed on a 64-bit Solaris Express Community Edition (build 108) virtual machine with 2GB memory. We use ZFS pool version 14 and ZFS file system version 3. We run ZFS on top of a single disk for our experiments.

To emulate disk corruptions, we developed a fault injection framework consisting of a pseudo-driver to perform fault injection on disk blocks and an application for controlling the experiments. The pseudo-driver is a standard Solaris layered driver that interposes between the ZFS virtual device and the disk driver beneath. We analyze the behavior of ZFS by looking at return values, checking system logs, and tracing system calls.

Test Procedure and Workloads

In our tests, we wanted to understand the behavior of ZFS to disk corruptions on different types of blocks. We injected faults by flipping bits at random offsets in disk blocks. Since we used the default setting in ZFS for compression (metadata compressed and data uncompressed), our fault injection tests corrupted compressed metadata and uncompressed data blocks on disk. We injected faults on nine different classes of ZFS on-disk blocks and for each class, we corrupted a single copy as well as all copies of blocks.

In our fault injection experiments on pool-wide and file system level metadata, we used "mount" and "remount" operations as our workload. The "mount" workload indicates that the target block is corrupted with the pool exported and "myfs" not mounted, and we subsequently mount it. This workload forces ZFS to use on-disk copies of metadata. The "remount" workload indicates that the target block is corrupted with "myfs" mounted and we subsequently umount and mount it. ZFS uses in-memory copies of metadata in this workload.

For injecting faults in file and directory blocks in a file system, we used two simple operations as workloads: "create file" creates a new file in a directory, and "read file" reads a file's contents.

3.2.2 Results and Observations

The results of our fault injection experiments are shown in Table 3.2. The table reports the results of experiments on pool-wide metadata and file system metadata and data. It also shows the results of corrupting a single copy as well as all copies of blocks. We now explain the results in detail in terms of the observations we made from our fault injection experiments.

Observation 1: *ZFS detects all corruptions due to the use of checksums.* In our fault injection experiments on all metadata and data, we found that bad data was never returned to the user because ZFS was able to detect all corruptions due to the use of checksums in block pointers. The parental checksums are used in ZFS to verify the integrity of all the on-disk blocks accessed. The only exception are uberblocks, which do not have parent block pointers. Corruptions to the uberblock are detected by the use of checksums inside the uberblock itself.

Observation 2: *ZFS gracefully recovers from single metadata block corruptions.* For pool-wide metadata and file system wide metadata, ZFS recovered from disk corruptions by using the ditto blocks. ZFS keeps three ditto blocks for pool-wide metadata and two for file system metadata. Hence, on single-block corruption to metadata, ZFS was successfully able to detect the corruption and use other avail-

Level	Block	Single ditto: mount	Single ditto: remount	Single ditto: create file	Single ditto: read file	All ditto: mount	All ditto: remount	All ditto: create file	All ditto: read file
zpool	vdev label[1]	R	R			E	R		
	uberblock	R	R			E	R		
	MOS object set block	R	R			E	R		
	MOS dnode block	R	R			E	R		
zfs	myfs object set block	R	R			E	R		
	myfs indirect block	R	R			E	R		
	myfs dnode block	R	R			E	R		
	dir ZAP block			R	R			E	E
	file data block				E				E

[1] excluding the uberblocks contained in it.

Table 3.2: **On-disk corruption analysis** *The table shows the results of on-disk experiments. Each cell indicates whether ZFS was able to recover from the corruption (R), whether ZFS reported an error (E), whether ZFS returned bad data to the user (B), or whether the system crashed (C). Blank cells mean that the workload was not exercised for the block.*

able correct copies to recover from it; this is shown by the cells (R) in the "Single ditto" column for all metadata blocks.

Observation 3: *ZFS does not recover from data block corruptions.* For data blocks belonging to files, ZFS was not able to recover from corruptions. ZFS detected the corruption and reported an error on reading the data block. Since ZFS does not keep multiple copies of data blocks by default, this behavior is expected; this is shown by the cells (E) for the file data block.

Observation 4: *In-memory copies of metadata help ZFS to recover from serious multiple block corruptions.* In an active storage pool, ZFS caches metadata in memory for performance. ZFS performs operations on these cached copies of metadata and writes them to disk on transaction group commits. These in-memory copies of metadata, along with periodic transaction commits, help ZFS recover from multiple disk corruptions.

In the "remount" workload that corrupted all copies of uberblock, ZFS recovered from the corruptions because the in-memory copy of the active uberblock remains as long as the pool exists. The in-memory copy is subsequently written to a new disk block in a transaction group commit, making the old corrupted copy

void. Similar results were obtained when corrupting other pool-wide metadata and file system metadata, and ZFS was able to recover from these multiple block corruptions (R).

Observation 5: *ZFS cannot recover from multiple block corruptions affecting all ditto blocks when no in-memory copy exists.* For file system metadata, like directory ZAP blocks, ZFS does not always keep an in-memory copy unless the directory has been accessed. Thus, on corruptions to both ditto blocks, ZFS reported an error. This behavior is shown by the results (E) for directories indicating for the "create file" and "read file" operations. Note that we performed these corruptions without first accessing the directory, so that there were no in-memory copies. Similarly, in the "mount" workload, when the pool was inactive (exported) and thus no in-memory copies existed, ZFS was unable to recover from multiple disk corruptions and responded with errors (E).

Observation 4 and 5 also lead to an interesting conclusion that an active storage pool is likely to tolerate more serious disk corruptions than an inactive one.

In summary, ZFS successfully detects all corruptions and recovers from them as long as one correct copy exists. The in-memory caching and periodic flushing of metadata on transaction commits help ZFS recover from serious disk corruptions affecting all copies of metadata. For user data, ZFS does not keep redundant copies and is unable to recover from corruptions. ZFS, however, detects the corruptions and reports an error to the user.

3.3 In-memory Data Integrity in ZFS

Although ZFS was not specifically designed to tolerate memory corruptions, we still would like to know how ZFS reacts to memory corruptions, i.e., whether ZFS can detect and recover from a single bit flip in data and metadata blocks. In this section, we perform a series of fault injection experiments to study the behavior of ZFS in the presence of memory corruptions. We find that ZFS has no precautions for memory corruptions: bad data blocks are returned to the user or written to disk, file system operations fail, and many times the whole system crashes.

3.3.1 Methodology

Now we discuss the fault injection framework and the test procedure and workloads. The injection framework is similar to the one used for on-disk experiments. The only difference is the pseudo-driver, which in this case, interacts with the ZFS stack by calling internal functions to locate the in-memory structures.

Object	Data Structures	Workload
MOS dnode	dnode_t, dnode_phys_t	zfs create, zfs destroy, zfs rename, zfs list, zfs mount, zfs umount
Object directory	dnode_t, dnode_phys_t, mzap_phys_t, mzap_ent_phys_t	
Dataset	dnode_t, dnode_phys_t, dsl_dataset_phys_t	
Dataset directory	dnode_t, dnode_phys_t, dsl_dir_phys_t	
Dataset child map	dnode_t, dnode_phys_t, mzap_phys_t, mzap_ent_phys_t	
FS dnode	dnode_t, dnode_phys_t	zfs umount, path traversal
Master node	dnode_t, dnode_phys_t, mzap_phys_t, mzap_ent_phys_t	
File	dnode_t, dnode_phys_t, znode_phys_t	open, close, lseek, read, write, access, link, unlink, rename, truncate (chdir, mkdir, rmdir)
Dir	dnode_t, dnode_phys_t, znode_phys_t, mzap_phys_t, mzap_ent_phys_t	

Table 3.3: **Summary of Tested Objects** *The table presents a summary of all ZFS objects corrupted in our in-memory analysis, along with their data structures and the workloads exercised on them.*

We wished to find out the behavior of ZFS in response to corruptions in different in-memory objects. Since all data and metadata in memory are uncompressed, we performed a controlled fault injection in various objects. For metadata, we randomly flipped a bit in each individual field of the structure separately; for data, we randomly corrupted a bit in a data block of a file in memory. We repeated each fault injection test five times. We performed fault injection tests on nine different types of objects at two levels (zfs and zpool) and exercised different set of workloads as listed in Table 3.3. Table 3.4 shows all data structures inside the objects and all the fields we corrupted during the experiments.

For data blocks, we injected bit flips at an appropriate time as described below. For reads, we flipped a random bit in the data block after it was loaded to the page cache; then we issued a subsequent read() on that block to see if ZFS returned the corrupted block. In this case, the read() call fetched the block from the page cache. For writes, we corrupted the block after the write() call finished but before the target block was written to the disk.

Data Structure	Fields
dnode_t	dn_nlevels, dn_bonustype, dn_indblkshift, dn_nblkptr, dn_datablkszsec, dn_maxblkid, dn_compress, dn_bonuslen, dn_checksum, dn_type
dnode_phys_t	dn_nlevels, dn_bonustype, dn_indblkshift, dn_nblkptr, dn_datablkszsec, dn_maxblkid, dn_compress, dn_bonuslen, dn_checksum, dn_type, dn_used, dn_flags,
mzap_phys_t	mz_block_type, mz_salt
mzap_ent_phys_t	mze_value, mze_name
znode_phys_t	zp_mode, zp_size, zp_links, zp_flags, zp_parent
dsl_dir_phys_t	dd_head_dataset_obj, dd_child_dir_zapobj, dd_parent_obj
dsl_dataset_phys_t	ds_dir_obj

Table 3.4: **Summary of Tested Data structures and Fields** *The table lists all fields we corrupted in the in-memory experiments.* `mzap_phys_t` *and* `mzap_ent_phys_t` *are metadata stored in ZAP blocks. The last three structures are object-specific structures stored in the dnode bonus buffer.*

For metadata, in our fault injection experiments, we covered a broad range of metadata structures (totally 16 core objects/structures). To reduce the sample space for experiments to more interesting cases, we made two choices. First, we always injected faults to the in-memory structure after it was accessed by the file system, so that both the in-heap version and page cache version already exist in the memory. Second, among the in-heap structures, we only corrupted the `dnode_t` structure (in-heap version of `dnode_phys_t`). The dnode structure is the most widely used metadata structure in ZFS and every object in ZFS is represented by a dnode. Hence, we anticipate that corrupting the in-heap dnode structure will cover many interesting cases.

3.3.2 Results and Observations

We present the results of our in-memory experiments in Table 3.5. As shown, ZFS fails to catch data block corruptions due to memory errors in both read and write

experiments. Single bit flips in metadata blocks not only lead to returning bad data blocks, but also cause more serious problems like failure of operations and system crashes. Note that Table 3.5 only shows cases with apparent problems. In other cases that are either indicated by a dot (.) in the result cells or not shown at all in Table 3.5, the corresponding operation either did not access the corrupted field or completed successfully with the corrupted field. However, in all cases, ZFS did not correct the corrupted field.

Next we present our observations on ZFS behavior and user-visible results. The first five observations are about ZFS behavior and the last five observations are about user-visible results of memory corruptions.

Observation 1: *ZFS does not use the checksums in the page cache along with the blocks to detect memory corruptions.* Checksums are the first guard for detecting data corruption in ZFS. However, when a block is already in the page cache, ZFS implicitly assumes that it is protected against corruptions. In the case of reads, the checksum is verified only when the block is being read from the disk. Following that, as long as the block stays in the page cache, it is never checked against the checksum, despite the checksum also being in the page cache (in the block pointer contained in its parental block). The result is that ZFS returns bad data to the user on reads.

For writes, the checksum is generated only when the block is being written to disk. Before that, the dirty block stays in the page cache with an outdated checksum in the block pointer pointing to it. If the block is corrupted in the page cache before it is flushed to disk, ZFS calculates a checksum for the bad block and stores the new checksum in the block pointer. Both the block and its parental block containing the block pointer are written to disk. On subsequent reads of the block, it passes the checksum verification and is returned to the user.

Moreover, since the detection mechanisms already fail to detect memory corruptions, recovery mechanisms such as ditto blocks and the mirrored zpool are not triggered to recover from the damage.

The results in Table 3.5 indicate that when a data block was corrupted, the application that issued a read() or write() request was returned bad data (B), as shown in the last row. When metadata blocks were corrupted, ZFS accessed the corrupted data structures and thus behaved wrongly, as shown by other cases in the result table.

		File	Dir	MOS dnode	Dataset directory	Dataset childmap	Dataset
Structure	**Field**	**O R W A U N T**	**O A L U N T M C D**	**c d r l m u**	**c d r l m u**	**c d r**	**c d r l m**
dnode_t	dn_type	. . . . .	. . . . .	C C C C C C	. . . . .	. . .	. . . . .
	dn_indblkshift	. BC . . C . .	. . E E E . E . E	. . . . .	. . . . .	. . .	. . . . .
	dn_nlevels	. . C . . . C	. . C C C . C . C	C C C C C C	. . . . .	C C C	C C C . .
	dn_checksum	. . C . . . C	. . . . .	. . . . .	. . . . .	. . .	. . . . .
	dn_compress	. . C	. . . . .	. . . . .	. . . . .	. . .	. . . . .
	dn_maxblkid	 C	 C	. . . . .	. . . . .	. . .	. . . . .
dnode_phys_t	dn_indblkshift	. . . . C . .	. . . . .	. . . . .	. . . . .	. . .	. . . . .
	dn_nlevels	. BC C . C . .	 C	. . . . .	. . . . .	. C .	. . . . .
	dn_nblkptr	. . . . .	. . . . .	. . . . .	. . . . .	. . .	. C . . .
	dn_bonuslen	. . C	. . . . .	. . . . .	. C . . .	. . .	. C . . .
	dn_maxblkid	. B . . C . C	 C	. . . . .	. C . . .	. C .	. C . . .
znode_phys_t	zp_size	. . . . .	. . . . . E				
	zp_flags	E . . E . E E	E E E E E E E E				
dsl_dir_phys_t	dd_head_dataset_obj				E E E E . .		
	dd_child_dir_zapobj				EC EC EC EC EC C		
dsl_dataset_phys_t	ds_dir_obj						. E E . .
data block		B B					

Table 3.5: **In-memory corruption results** *The table shows our memory corruption results. The operations exercised are O(open), R(read), W(write), A(access), L(link), U(unlink), N(rename), T(truncate), M(mkdir), C(chdir), D(rmdir), c(zfs create), d(zfs destroy), r(zfs rename), l(zfs list), m(zfs mount) and u(zfs umount). Each result cell indicates whether the system crashed (C), whether the operation failed with wrong results or with a misleading message (E), whether a bad data block was returned (B) or whether the operation completed (.). Large blanks mean that the operations are not applicable.*

Observation 2: *The window of vulnerability of blocks in the page cache is unbounded.* As Figure 3.4 shows, after a block is loaded into the page cache by first read, it stays there until evicted. During this interval, if a corruption happens to the block, any subsequent read will get the corrupted block because the checksum is not verified. Therefore, as long as the block is in the page cache (unbounded), it is susceptible to memory corruptions.

Observation 3: *Since checksums are created when blocks are written to disk, any corruption to blocks that are dirty (or will be dirtied) is written to disk permanently on a flush.* As described in Section 3.1, dirty blocks in the page cache are written to disk during a flush. During the flush, any dirty block will further cause updates of all its parental blocks; a new checksum is then calculated for each updated block and all of them are flushed to disk. If a memory corruption happens to any of those blocks before a flush (above the black dotted line before G in Figure 3.4), the corrupted block is written to disk with a new checksum. The checksum is thus valid for the corrupted block, which makes the corruption permanent. Since the window of vulnerability is long (30 seconds), and there are many blocks that will be flushed to disk in each flush, we conjecture that the likelihood of memory corruption leading to permanent on-disk corruptions is high.

We did a block-based fault injection to verify this observation. We injected a single bit flip to a dirty (or to-be-dirtied) block before a flush; as long as the flipped bit in the block was not overwritten by subsequent operations, the corrupted block was written to disk permanently.

Observation 4: *Dirtying blocks due to updating file access time increases the possibility of making corruptions permanent.* By default, access time updates are enabled in ZFS; therefore, a read-only workload will update the access time of any file accessed. Consequently, when the structure containing the access time (znode) goes inactive (or when there is another workload that updates the znode), ZFS writes the block holding the znode to disk and updates and writes all its parental blocks. Therefore, any corruption to these blocks will become permanent after the flush caused by the access time update. Further, as mentioned earlier, the time interval when the corruption could happen is unbounded.

Observation 5: *For most metadata blocks in the page cache, checksums are not valid and thus useless in detecting memory corruptions.* By default, most metadata blocks such as indirect blocks and dnode blocks are compressed on disk. Since the checksums for these blocks are used to prevent disk corruptions, they are only valid for compressed blocks, which are calculated after they are compressed during writes and verified before they are decompressed during reads. When metadata blocks are in the page cache, they are uncompressed. Therefore, the checksums

contained in the corresponding block pointers are useless.

Observation 6: *When metadata is corrupted, operations fail with wrong results, or give misleading error messages (E).* For example, when `zp_flags` in `dnode_phys_t` for a file object was corrupted, open() may return an error code EACCES (permission denied). The case occurred when the 41st bit of `zp_flags` was flipped from 0 to 1, which signifies that the file is quarantined by an anti-virus software. Therefore, open() was incorrectly denied, giving an error code EACCES. The calls access(), rename() and truncate() also failed for the same reason.

Another example of a misleading error message happened when `dd_head_dataset_obj` in `dsl_dir_phys_t` for a dataset directory object was corrupted. In this case, "zfs create" failed to create a new file system under the parent file system represented by the corrupted object. ZFS gave a misleading error message saying that the parent file system did not exist. ZFS gave similar error messages in other cases (E) under "Dataset directory" and "Dataset".

Observation 7: *Many corruptions lead to a system crash (C).* For example, when `dn_nlevels` (the height of the block tree pointed to by the dnode) in `dnode_phys_t` for a file object was corrupted and the file was read, the system crashed due to a NULL pointer dereference. In this case, ZFS used the wrong value of `dn_nlevels` to traverse the block tree of the file object and obtained an invalid block pointer. Therefore, the block size obtained from the block pointer was an arbitrary value, which was then used to index into an array whose size was much less than the value. As a result, the system crashed when a NULL pointer was dereferenced.

Observation 8: *The read() system call may return bad data.* As shown in Table 3.5, for metadata corruptions, there were three cases where read() gave bad data block to the user. In these cases, ZFS simply trusted the value of the corrupted field and used it to traverse the block tree pointed to by the dnode, thus returning bad blocks. For example, when `dn_nlevels` in `dnode_phys_t` for a file object was changed from 3 to 1, ZFS gave an incorrect block to the user on a read request for the first block of the file. The bad block was returned because ZFS assumed that the tree only had one level, and incorrectly returned an indirect block to the user. Such cases where wrong blocks are returned to the user also have the potential for security vulnerabilities.

Observation 9: *There is no recovery for corrupted metadata.* In the cases where no apparent error happened (as indicated by a dot or not shown) and the operation was not meant to update the corrupted field, the corruption remained in the metadata block in the page cache.

In summary, ZFS fails to detect and recover from memory corruptions. Check-

sums in the page cache are not used to protect the integrity of blocks. Therefore, bad data blocks are returned to the user or written to disk. Moreover, corrupted metadata blocks are accessed by ZFS and lead to operation failure and system crashes.

3.4 Probability Analysis of Memory Corruption

In this section, we present a preliminary analysis of the likelihood of different failure scenarios due to memory errors in a system using ZFS. Specifically, given that one random bit in memory is flipped, we compute the probabilities of four scenarios: reading corrupt data (R), writing corrupt data (W), crashing/hanging (C) and running successfully to completion (S). These probabilities help us to understand how severely file system data integrity is affected by memory corruptions and how much effort file system developers should make to add extra protection to maintain data integrity.

3.4.1 Methodology

We apply fault-injection techniques to perform the analysis. Considering one run of a specific workload as a trial, we inject a fixed number number of random bit flips to the memory and record how the system reacts. By doing multiple trials, we measure the number of trials where each scenario occurs, thus estimating the probability of each scenario given that certain number of bits are flipped. Then, we calculate the probability of each scenario given the occurrence of one single bit flip.

We have extended our fault injection framework to conduct the experiments. We replaced the pseudo-driver with a user-level "injector" which injects random bit flips to the physical memory. We used filebench [107] to generate complex workloads. We modified filebench such that it always writes predefined data blocks (e.g., full of 1s) to disk. Therefore, we can check every read operation to verify that the returned data matches the predefined pattern. We can also verify the data written to disk by checking the contents of on-disk files.

We used the framework as follows. For a specific workload, we ran 100 trials. For each trial, we used the injector to generate 16 random bit flips at the same time when the workload has been running for 3 minutes. We then kept the workload running for 5 minutes. Any occurrence of reading corrupt data (R) was reported. When the workload was done, we checked all on-disk files to see if there was any corrupt data written to the disk (W). Since we only verify write operations after each run of a workload, some intermediate corrupt data might have been overwritten and

thus the actual number of occurrence of writing corrupt data could be higher than measured here. We also logged whether the system hung or crashed (C) during each trial, but we did not determine if it was due to corruption of ZFS metadata or other kernel data structures.

It is important to notice that we injected 16 bit flips in each trial because it let us observe a sufficient number of failure trials in 100 trials. However, we apply the following calculation to derive the probabilities of different failure scenarios given that 1 bit is flipped.

3.4.2 Calculation

We use $P_k(X)$ to represent the probability of scenario X given that k random bits are flipped, in which X could be R, W, C or S. Therefore, $P_k(\bar{X}) = 1 - P_k(X)$ is the probability of scenario X not happening given that k bits are flipped. In order to calculate $P_1(X)$, we first measure $P_k(X)$ using the method described above and then derive $P_1(X)$ from $P_k(X)$, as explained below.

- **Measure** $P_k(X)$ Given that k random bit flips are injected in each trial, we denote the total number of trials as N and the number of trials in which scenario X occurs at least once as N_X. Therefore,

$$P_k(X) = \frac{N_X}{N}$$

- **Derive** $P_1(X)$ Assume k bit flips are independent, then we have

$$P_k(\bar{X}) = (P_1(\bar{X}))^k, \text{ when } X = R, W \text{ or } C$$

$$P_k(X) = (P_1(X))^k, \text{ when } X = S$$

  Substituting $P_k(\bar{X}) = 1 - P_k(X)$ into the equations above, we can get,

$$P_1(X) = 1 - (1 - P_k(X))^{\frac{1}{k}}, \text{ when } X = R, W \text{ or } C$$

$$P_1(X) = (P_k(X))^{\frac{1}{k}}, \text{ when } X = S$$

3.4.3 Results

The analysis is performed on the same virtual machine as mentioned in Section 3.2.1. The machine is configured with 2GB memory and a single disk running ZFS. We first ran some controlled micro-benchmarks (e.g., sequential read) to verify that the

Workload	$P_{16}(R)$	$P_{16}(W)$	$P_{16}(C)$	$P_{16}(S)$
varmail	9% [4, 17]	0% [0, 3]	5% [1, 12]	86% [77, 93]
oltp	26% [17, 36]	2% [0, 8]	16% [9, 25]	60% [49, 70]
webserver	11% [5, 19]	20% [12, 30]	19% [11, 29]	61% [50, 71]
fileserver	69% [58, 78]	44% [34, 55]	23% [15, 33]	28% [19, 38]

Workload	$P_1(R)$	$P_1(W)$	$P_1(C)$	$P_1(S)$
varmail	0.6% [0.2, 1.2]	0% [0, 0.2]	0.3% [0.1, 0.8]	99.1% [98.4, 99.5]
oltp	1.9% [1.2, 2.8]	0.1% [0, 0.5]	1.1% [0.6, 1.8]	96.9% [95.7, 97.8]
webserver	0.7% [0.3, 1.3]	1.4% [0.8, 2.2]	1.3% [0.7, 2.1]	97.0% [95.8, 97.9]
fileserver	7.1% [5.4, 9.0]	3.6% [2.5, 4.8]	1.6% [1.0, 2.5]	92.4% [90.2, 94.2]

Table 3.6: $P_{16}(X)$ **and** $P_1(X)$ *The upper table presents percentage values of the probabilities and 95% confidence intervals (in square brackets) of reading corrupt data (R), writing corrupt data (W), crash/hang and everything being fine (S), given that 16 bits are flipped, on a machine of 2GB memory. The lower table gives the derived percentage values given that 1 bit is corrupted. The working set size of each workload is less than 2GB; the average amount of page cache consumed by each workload after the bit flips are injected is 31MB (varmail), 129MB (oltp), 441MB (webserver) and 915MB (fileserver).*

methodology and the calculation is correct (the result is not shown due to limited space). Then, we chose four workloads from filebench: varmail, oltp, webserver and fileserver, all of which were exercised with their default parameters. A detailed description of these workloads can be found elsewhere [107].

Table 3.6 provides the probabilities and confidence intervals given that 16 bits are flipped and the derived values given that 1 bit is flipped. Note that for each workload, the sum of $P_k(R)$, $P_k(W)$, $P_k(C)$ and $P_k(S)$ is not necessary equal to 1, because there are cases where multiple failure scenarios occur in one trial.

From the lower table in Table 3.6, we see that a single bit flip in memory causes a small but non-negligible percentage of runs to experience failure. For all workloads, the probability of reading corrupt data is greater than 0.6% and the probability of crashing or hanging is higher than 0.3%. The probability of writing corrupt data varies widely from 0 to 3.6%. Our results also show that in most cases, when the working set size is less than the memory size, the more page cache the workload consumes, the more likely that a failure would occur if one bit is flipped.

In summary, when a single bit flip occurs, the chances of failure scenarios happening can not be ignored. Therefore, efforts should be made to preserve data integrity in memory and prevent these failures from happening.

3.5 Summary

In this chapter, we analyzed a state-of-the-art file system, ZFS, to study the implications of disk and memory corruptions to data integrity. We used carefully controlled fault injection experiments to simulate realistic disk and memory errors and presented our observations about ZFS behavior and its robustness.

While the reliability mechanisms in ZFS are able to provide reasonable robustness against disk corruptions, memory corruptions still remain a serious problem to data integrity. Our results for memory corruptions indicate cases where bad data is returned to the user, operations silently fail, and the whole system crashes. Our probability analysis shows that one single bit flip has small but non-negligible chances to cause failures such as reading/writing corrupt data and system crashing.

We argue that file systems should be designed with comprehensive data protection. File systems should not only provide protection against disk corruptions, but also aim to protect data from memory corruptions, which may require cooperation from the page cache and even user-level applications.

Chapter 4

Z^{2}FS: Cooperative Data Protection in Local Storage

Many features that storage systems provide require great care and coordination across the many layers of the system (e.g., performance), but integrity checks for data protection generally remain isolated within individual components. For example, as shown in Chapter 3, ZFS uses checksums to protect on-disk block, but fails to extend the checksums to protect in-memory data; hard disks have built-in ECC for each sector [22], but the ECCs are rarely exposed to the upper-level system; TCP uses Internet checksums to protect data payload [11], but only during the transmission. When data is transferred across components, data is not protected and thus may become silently corrupted.

A comprehensive approach is to apply the straight-forward end-to-end data protection [94], where high-level applications generate and verify checksums for their data such that the checksums protect data throughout the entire I/O stack. This approach does provide better data protection, but it suffers the performance and timeliness problems, as discussed in Chapter 1.

To address both problems, we propose a new concept called *flexible end-to-end data integrity*. With this concept, all components on the I/O path are aware of the checksum, and different components can choose different type of checksum, depending on the reliability characteristics (e.g., failure rate) and performance requirements (e.g., throughput) of the component. Then, we develop an analytical framework to provide rationale for the new concept. Specifically, the framework is able to evaluate and compare the reliability of different storage systems, and help to choose proper checksums for different components. Finally, guided by the framework, we build Zettabyte-reliable ZFS (Z^{2}FS) by applying flexible end-to-end data

protection to ZFS. Z^{2}FS is able to provide Zettabyte Reliability while performing comparably to ZFS.

The rest of the chapter is organized as follows. In Section 4.1, we introduce the framework for evaluating reliability of storage systems. We then present the design of Z^{2}FS in Section 4.2 and discuss some implementation issues in Section 4.3. Finally, we evaluate Z^{2}FS in Section 4.4.

4.1 Reliability of Storage Systems with Data Corruption

We now present a framework to analyze the reliability of storage systems with data corruption. The framework uses analytical models for each type of device and checksum in a system to calculate a reliability metric in terms of the probability of undetected data corruption.

4.1.1 Overview

The reliability of a storage system can be evaluated based on how likely corruption would occur. There are two types of corruption: detected and undetected (silent data corruption, SDC). Detected corruption is the case the system is built to detect and may recover from, but SDC is what the system is not prepared for. SDC does more harm in that it would be treated as correct data and may further pollute other good data (e.g., RAID reconstruction with corrupted data). Therefore, we focus on the probability of SDC in a storage system. To quantify how likely a SDC would occur, we use the probability of undetected data corruption (udc) when reading a data block from the system $P_{sys-udc}$ as a reliability metric.

$P_{sys-udc}$ for a storage system depends on various devices, each of which may experience corruptions caused by different factors. Each device may employ different types of hardware protection and the upper-level system or application may add extra protection mechanisms. Therefore, we propose a framework that takes a ground-up approach to derive the system-level reliability metric from underlying devices.

The framework consists of models for devices and checksums. All models are built around the basic storage unit, a data block of b bits. For a raw device D (with its own hardware-level checksum), we are interested in how likely corruption would occur to a block and escape from the detection of the device's checksum ($P_c(D)$). To detect such corruption, high-level (software) checksums are usually applied on top of a raw device (henceafter, we will use "checksum" to indicate the high-level checksum). Each data block has a checksum of k bits. For a checksum

C and device D, we focus on the device-level probability of undetected corruption ($P_{udc}(D, C)$) when the checksum is used to protect a data block on the device.

Devices with different checksums are connected in various ways to form the whole system. A data block can pass through or stay in several devices from the time it is born to the time it is accessed. By considering all possible corruption scenarios during this time period, we calculate the overall probability of undetected data corruption when reading the data block from the system ($P_{sys-udc}$).

4.1.2 Models for Devices and Checksums

To demonstrate how to apply the framework, we present models for devices and checksums that will be used throughout the chapter. We make assumptions (e.g., independence of bit errors) to simplify our models such that we can focus on reasoning about the reliability of storage systems within the framework; discussion on more complex and accurate models is beyond the scope of this chapter.

Device Model

We consider two types of devices, hard disks (dsk) and memory (mem) , and one type of corruption: random bit flip. We assume the block size b is 32768 bits (4KB).

Hard Disks Hard disks are a long-term storage medium for data, and are known to be unreliable. Hard disks can exhibit unusual behaviors because of hardware faults such as latent sector errors [22, 96]. These errors can usually be detected by disk ECC. The less-likely but more harmful silent data corruption may come from hardware bit rot, buggy firmware, or mechanic faults (such as dropped writes and misdirected writes [23, 92]), causing random bit flips and block corruption. These errors are not detectable by disk ECC.

Bit error rate (BER) is often used to characterize the reliability of a hard disk. BER is defined as the number of bit errors divided by the total number of bits transferred and often refers to detected bit error (by disk ECC). For silent corruption, we are more interested in the undetected bit error rate (UBER), which is the rate of errors that have escaped from ECC. Assuming each bit error in a data block is independent and the number of bit errors follows a binomial distribution, the probability of an undetected bit flip is equal to UBER. Assuming there is at most one flip for each bit, the probability of i bit flips in a b-bit block is:

$$P_c(dsk, i) = \binom{b}{i} (\text{UBER})^i (1 - \text{UBER})^{b-i}$$

Therefore, the probability of corruption in a block is the sum of the probabilities of all possible bit flips (from exactly 1 bit flip to exact b bit flips):

$$P_c(dsk) = \sum_{i=1}^{b} \binom{b}{i} (\text{UBER})^i (1 - \text{UBER})^{b-i}$$

While BER is often reported by disk manufactures, ranging from 10^{-14} to 10^{-16}, there is no published data on UBER. Rozier et al. estimated that the rate of undetected disk error caused by far-off track writes and hardware bit corruption is between 10^{-12} and 10^{-13} [92]. Although we do not know the percentage of errors caused by either fault, we conservatively assume that most are bit errors and thus we pick 10^{-12} as the UBER for current disks. In our study, we choose a wider range for UBER, from 10^{-10} to 10^{-20}, to cover more reliability levels. To simplify the presentation, we define the *disk reliability index* as $-log_{10}(\text{UBER})$.

Memory Memory (DRAM) is mainly used to cache data for performance. Bit flips are the main corruption type, probably due to chip faults or external radiation [75, 133]. Earlier studies show that memory errors can occur at a rate of 10 to 360 errors/year/GB [83, 84, 100] and suspect that most errors are soft errors, which are transient. However, recent studies show that memory errors occur more frequently [63, 71, 97] and are probably dominated by hard errors (actual device defects). If a memory module has ECC or more complex codes such as chipkill [64], then both soft errors and hard errors within the capability of the codes can be detected or corrected. However, corruption caused by software bugs [106] are not detectable by these hardware codes.

For memory, the error rate is usually measured as failure in time (FIT) per Mbit. Assuming each failure is a bit flip, 1 FIT/Mbit means there is one bit flip in one billion hours per Mbit. Assuming each bit flip is independent and the same bit can only experience one flip, we model the number of bit flips in a b-bit block during a time period t as a Poisson distribution with a constant failure rate λ errors/second/bit. Therefore, the probability of i bit flips in a b-bit block during time t is:

$$P_c(mem, i, t) = \frac{e^{-b\lambda t}(b\lambda t)^i}{i!}$$

Summing up the probabilities of all possible bit corruptions, we have:

$$P_c(mem, t) = \sum_{i=1}^{b} \frac{e^{-b\lambda t}(b\lambda t)^i}{i!}$$

Previous studies reported FIT/Mbit as low as 0.56 [72] and as high as 167,066 [63]. Converting to errors/second/bit gives the range for λ, from 1.48×10^{-19} (λ_{min}) to 4.42×10^{-14} (λ_{max}). In this chapter, we choose 6.62×10^{-15} (λ_{mid}) as the error rate of non-ECC memory; it is derived from 25,000 FIT/Mbit, which is the lower bound of the DRAM error rate measured in a recent study [97]. We pick λ_{min} as the error rate of ECC memory, because most errors would have been detected by ECC. We use $-log_{10}(\lambda)$ as the *memory reliability index*. The corresponding indices for λ_{min}, λ_{mid}, and λ_{max} are 18.8, 14.2, and 13.4.

Checksum Model

The effectiveness of a checksum is measured by the probability of undetected corruption given an error rate. It is usually difficult, sometimes impossible, to have an accurate model for the probability, because of the complexity of errors and the data-dependency property of some checksums. Therefore, we apply an analytic approach to evaluate checksums for random bit flips.

We focus on two types of checksum: xor (64-bit) and Fletcher (256-bit). Exclusive or checksums (xor) are calculated by XORing each fixed-sized chunk of a data block. For example, a 64-bit xor checksum over a 4KB data block is computed by XORing every 64-bit of data in the block. The xor checksum is very fast to calculate, but it can only detect one bit error. On the other hand, Flecther checksum is more complex, which involves calculating two checksums at a time. For instance, to compute a 256-bit Fletcher checksum from a 4KB block, the block is first divided into an array of 128-bit data chunks ($d_1, d_2, ..., d_{256}$), and two 128-bit checksums (s_1 and s_2) are initialized with 0. Then for every data chunk d_i (i from 1 to 256), s_1 and s_2 are calculated using one's complement addition as follows: $s_1 = (s_1 + d_i) \, mod \, 2^{128}$ and $s_2 = (s_2 + s_1) \, mod \, 2^{128}$. Finally, the two checksums are concatenated to form the Fletcher checksum of the block. Fletcher checksum is slower to compute than xor, but it can detect all 1-bit errors and 2-bit errors in a 4KB block.

Our approach to model both checksums is similar to the one used in a recent study on checksums for embedded control networks [74]. The idea is based on Hamming Distance (HD). A checksum C with HD=n can detect all bit errors up to $n-1$ bits, but there is at least one case of n bit flips that is undetectable by the checksum. We use $F(C)$ to represent the fraction of n bit flips that are undetectable by checksum C. Then, the probability of undetectable n bit flips is $P_c(D, n) \times F(C)$, in which $P_c(D, n)$ is the probability of n bit flips on device D. The actual P_{udc} is the sum of the probabilities of undetectable bit flips from n to b (the size of the block is b bits). Since the occurrence of more than n bit flips is highly unlikely,

Reliability Score	Reliability Goal	$P_{sys-udc}$
8.4	Terabyte	3.73×10^{-9}
11.4	Petabyte	3.64×10^{-12}
14.4	Exabyte	3.55×10^{-15}
17.5	Zettabyte	3.46×10^{-18}

Table 4.1: **Reliability Scores** *This table lists a mapping from reliability scores to different reliablity goals.*

the probability of undetected n bit flips dominates P_{udc} [74]. Therefore, we have the approximation of $P_{udc}(D, C) = P_c(D, n) \times F(C)$.

The value of $P_c(D, n)$ can be easily calculated based on the model of each device, so the key parameter is $F(C)$. Assuming the block size is b bits and the checksum size is k bits, there is an analytical formula for xor [74]: $F(xor) = \frac{b-k}{k(b-1)}$. Since the HD for xor is 2, we have: $P_{udc}(D, xor) = P_c(D, 2) \times \frac{b-k}{k(b-1)}$. But for Fletcher (HD=3), we can only get an approximation [10]: $F(Fletcher) = 4.16 \times 10^{-20}$. Therefore, $P_{udc}(D, Fletcher) = P_c(D, 3) \times (4.16 \times 10^{-20})$.

4.1.3 Calculating $P_{sys-udc}$

Based on previous models, given the configuration of a storage system, we can calculate $P_{sys-udc}$ by summing up the probabilities of every silent corruption scenario during the time from the data being generated to it being read. We define the *reliability score* for a system as $-log_{10}(P_{sys-udc})$; higher scores mean better reliability.

Finding all scenarios that lead to a silent corruption is tricky. In reality, it is possible that multiple devices corrupt the same data when it is transferred through or stored on them. In this chapter, we assume that in each scenario, there is only one corruption from when a data block is born to when it is read from the system. One reason is that data corruption is rare - multiple corruptions to the same data block are unlikely. Another reason is that with this assumption, we do not have to reason about complex interactions of corruption from multiple devices, which may require more advanced modeling techniques.

Determining whether a value of $P_{sys-udc}$ is good enough for a storage system is not easy. Ideally, the best value of $P_{sys-udc}$ is 0, but this is impossible. In reality, $P_{sys-udc}$ is a tradeoff between reliability and performance; it should be

Cfg Num	Cfg Name	Index Mem	Index Dsk	Description
1	low-end	13.4	10	worst mem & dsk
2	consumer	14.2	12	non-ECC mem & regular dsk
3	enterprise	18.8	12	ECC mem & regular dsk
4	server	18.8	20	ECC mem & best dsk

Table 4.2: **Sample System Configurations** *This table shows four configurations of a local file system that we will study throughout the dissertation.*

low enough such that SDC is extremely rare, but at the same time it should not hinder the system's performance. In this chapter, we use *Zettabyte Reliability* as a reliability goal of storage systems. Zettabyte reliability means that there is at most one SDC when reading one Zettabyte data from a storage system. With our models, assuming the block size and the IO size is 4KB, this goal translates to $P_{sys-udc} = P_{goal} = 3.46 \times 10^{-18}$, which in terms of a reliability score is 17.5. Intuitively, we can map other reliability scores to similar reliability metrics, as shown in Table 4.1. Note that the numerical value of the reliability goal may differ depending on the accuracy of the assumptions and models, and it may not be precise; our purpose is to use it as a way to demonstrate how to make proper tradeoffs between performance and protection in a storage system.

4.1.4 Example: NCFS

To illustrate how to apply the framework to evaluate the reliability of a storage system, we use a local file system with no checksum (NCFS) as an example. We focus on four configurations of the system, as listed in Table 4.2. Within the range for each index, we use the minimum value to represent the worst memory or disks which may be faulty or prone to corrupting data. We use the maximum disk index to represent disks that are much more reliable than regular disks.

The timeline of a data block from being generated to being accessed is shown in Figure 4.1. A writer application generates the block at t_0. The block stays in memory until t_1 when it is flushed to disk. The block is then read into memory at t_2 and finally accessed by a reader application at t_3. The residency time of the block in writer's memory and reader's memory is $t_1 - t_0$ and $t_3 - t_2$ respectively. To simplify the model and also because most file systems flush dirty blocks to disk

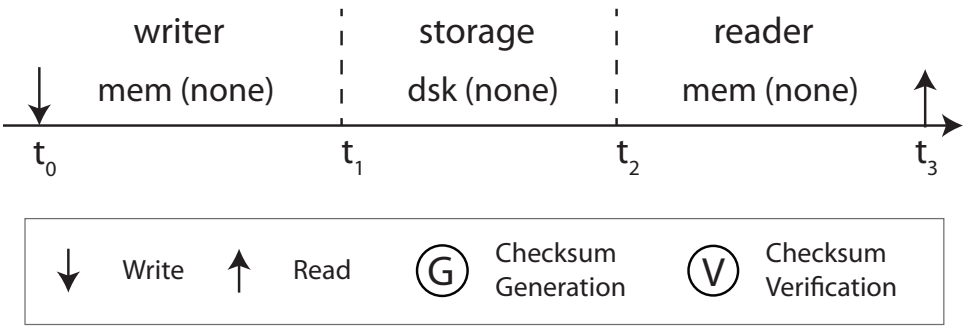


Figure 4.1: **Timeline of a Data Block in NCFS** *This figure shows timeline of a block from being generated by the writer (t_0) to being read by the reader (t_3) in NCFS. The timeline consists of three parts: writer in memory, storage (disk), and reader in memory. The name of the checksum used to protect data during each time period is listed in the parentheses on the right of the device name.*

at regular time intervals (usually 30 seconds), we assume $t_1 - t_0$ to be 30 seconds for all blocks in this chapter.

Based on the "one corruption" assumption, there are three scenarios that will lead to silent data corruption: corruption that occurs in the reader's memory, disk, or the writer's memory. Therefore, $P_{sys-udc}$ for NCFS is approximately the sum of the probabilities of corruption in each device:

$$\begin{aligned} P_{\text{NCFS}-udc} =& P_c(mem, t_{resident}) + P_c(dsk) \\ &+ P_c(mem, 30) \end{aligned}$$

where $t_{resident} = t_3 - t_2$ is the residency time (in seconds) of the block in the reader's memory and 30 is the residency time of it in the writer's memory. $P_{sys-udc}$ is a function of three variables: the reliability indices of memory and disk in the system, and the residency time $t_{resident}$.

The reliability score of NCFS ($t_{resident} = 1$) is shown in Figure 4.2, with the four configurations marked as "$\times$". We choose $t_{resident} = 1$ because it represents a best case (approximately) for reliability and we will discuss the sensitivity of reliability score to $t_{resident}$ in Section 4.2.3.

As one can see from the figure, when either the disk or the memory reliability index is low, corruption on that device dominates the reliability score. For example, when the disk reliability index is 12, the reliability score of the system almost does not change when the memory reliability index varies; both config 2 (consumer) and config 3 (enterprise) have a score of 7.4 (even worse than Terabyte reliability). But when the disk is more reliable, memory corruption starts to dominate and the

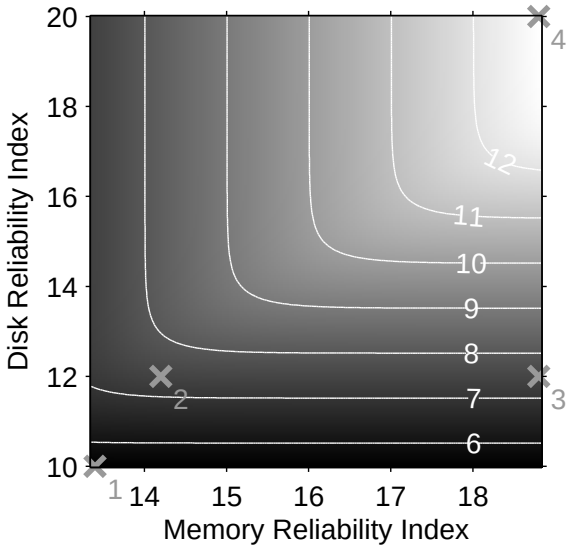


Figure 4.2: **NCFS Reliability Score ($t_{resident} = 1$)** *This figure illustrates a contour plot of the reliability score of NCFS. Darker color means lower score - worse reliability. Four points marked with a "$\times$" represent the four sample configurations: low-end (1), consumer (2), enterprise (3), server (4).*

reliability score increases as the memory reliability index increases. When both reliability indices are high, NCFS with config 4 (server) has the best reliability score of 12.8 (a little better than Petabyte), still less than the Zettabyte reliability goal (17.5).

4.2 From ZFS to Z^{2}FS

To explore end-to-end concepts in a file system, we now present two variants of ZFS: E^{2}ZFS, which takes the straight-forward end-to-end approach, and Z^{2}FS, which employs flexible end-to-end data integrity. Specifically, we show how ZFS, a modern file system with strong protection against disk corruption, can be further hardened with end-to-end data integrity to protect data all the way from application to disk, achieving Zettabyte reliability with better performance.

4.2.1 ZFS: the Original ZFS

ZFS is a state-of-the-art open source file system originally created by Sun Microsystems with many reliability features. ZFS provides data integrity by using

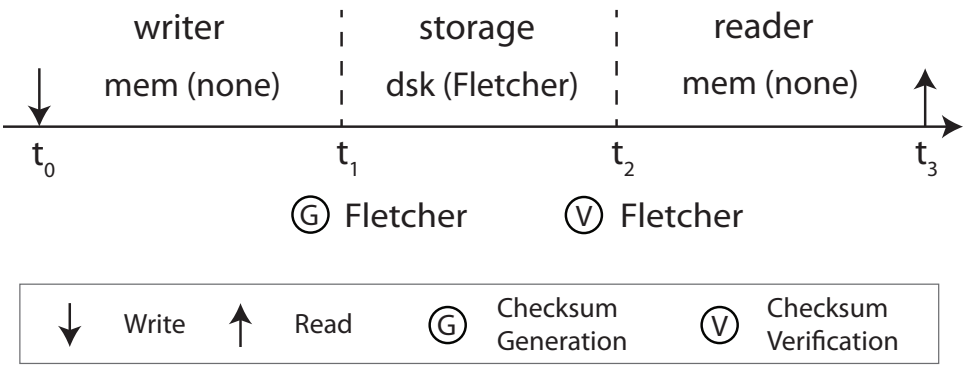


Figure 4.3: **Timeline of a Data Block in ZFS** *This figure shows timeline of a block in ZFS. The name of the checksum used to protect data during each time period is listed in the parentheses on the right of the device name. None means no checksum is used.*

checksums, data recovery with replicas, and consistency with a copy-on-write transactional model [29]. In addition, other mechanisms such as pooled storage, inline deduplication, snapshots, and clones, provide efficient data management.

Problem

One important feature that distinguishes ZFS from most other file systems is that ZFS provides protection from disk corruption by using checksums. ZFS maintains a *disk checksum* (Fletcher, by default) for each disk block and keeps the checksum in a block pointer structure. As shown in Figure 4.3, when ZFS writes a block to disk at t_1, it generates a Fletcher checksum. When ZFS reads the block back, it verifies the checksum and places it in the page cache. In this manner, ZFS is able to detect many kinds of corruption caused by disk faults, such as bit rot, phantom writes, and misdirected reads and writes [29].

However, Chapter 3, as well as some anecdotal evidence [9, 16, 17], shows that ZFS is vulnerable to memory corruption. The checksum in ZFS is only verified and generated at the boundary of memory and disk; once a block is cached in memory, the checksum is never verified again. Applications could read bad data from the page cache without knowing that it is corrupted. Even worse, if a dirty data page is corrupted before the new checksum is generated, the bad data will get to disk permanently with a matching checksum and later reads will not be able to detect the corruption.

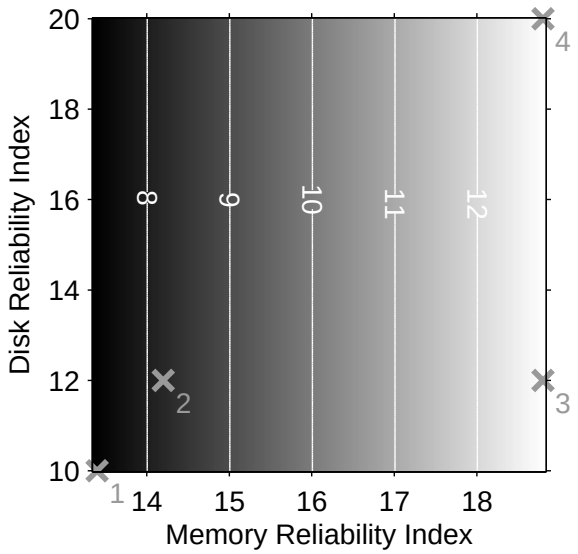


Figure 4.4: **ZFS Reliability Score ($t_{resident} = 1$)** *This figure illustrates a contour plot of the reliability score of ZFS. Darker color means lower score - worse reliability. Four points marked with a "$\times$" represent the four sample configurations: low-end (1), consumer (2), enterprise (3), server (4).*

Reliability Analysis

We apply the framework introduced in Section 4.1 to calculate the reliability score for ZFS. Similar to NCFS, there are three scenarios that cause SDC:

$$\begin{aligned} P_{\text{ZFS}-udc} = & P_c(mem, t_{resident}) \\ & + P_{udc}(dsk, Fletcher) \\ & + P_c(mem, 30) \end{aligned}$$

Because ZFS has on-disk blocks protected by Fletcher, only undetected corruption contributes to $P_{\text{ZFS}-udc}$.

Figure 4.4 depicts the reliability score of ZFS. With Fletcher protecting data on disk, the reliability score is now dominated by memory corruption. However, the reliability score is not improved much, due to the lack of protection of in-memory data. Both config 3 (enterprise) and config (server) 4 have the highest reliability score of 12.8 (above Petabyte reliability), but they are still below the Zettabyte reliability goal (17.5). It is interesting to see that config 4 (server) in ZFS has the same best reliability score as itself in NCFS, which indicates that when both the disk and memory reliability indices are the highest, memory corruption is more

severe than disk corruption. Therefore, we need to protect data in memory.

4.2.2 E^{2}ZFS: ZFS with End-to-end Data Integrity

To improve the reliability of ZFS, data both in memory and on disk must be protected. One way to achieve this is to apply the straight-forward end-to-end concept. In common practice, the writer generates an application-level checksum for the data block and sends both the checksum and data to the file system. Because the page cache and the file system are not aware of the checksum, the writer usually uses a portion of the data block to store the checksum. When the reader reads back the block, it can verify the checksum portion to ensure the integrity of the data portion. The checksum protects the data block all the way from the writer to the reader.

Because ZFS already maintains a checksum for each on-disk block in the block pointer, we do not have to append the application checksum on top of ZFS's checksum. Instead, we can simply store the application checksum in the block pointer, replacing the original disk checksum. Therefore, we only have to expose the checksum to the reader and writer, and make sure the page cache and the file system are oblivious to the checksum.

Implementation

To achieve the straight-forward end-to-end data integrity, we make the following changes to ZFS, transforming it into E^{2}ZFS.

First, we attach checksums to all buffers along the I/O path: user buffer, data page and disk block. Since ZFS already provides *disk checksum* for each disk block, we add *memory checksum* to the user buffer and the data page. It enables the system to pass checksums between the application and disk. Since only one checksum algorithm is used throughout the system, the memory checksum and the disk checksum are the same as the application-generated checksum, assuming the user buffers are always aligned to data pages. We will discuss the alignment issue in Section 4.3. E^{2}ZFS currently supports both xor and Fletcher, but only one can be used at a time.

Second, we enhance the existing read/write system calls with a new argument to transfer checksums between user and kernel space. The new argument is a buffer containing all checksums corresponding to the blocks in the user buffer. On reads, the application receives both data and checksum, and thus is able to verify the integrity of data. On writes, the application must generate a checksum for each data block, and send both the data block and checksum through the new system call.

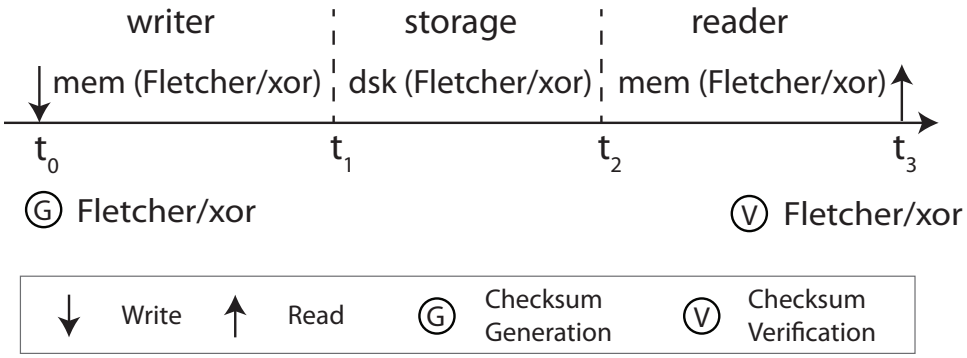


Figure 4.5: **Timeline of a Data Block in E^{2}ZFS** *This figure shows timeline of a block in E^{2}ZFS. E^{2}ZFS uses the same checksum (either xor or Fletcher) all the way through.*

Finally, we modify the checksum handling at the boundary of memory and disk such that the checksum is always passed through this boundary without any extra processing. E^{2}ZFS simply stores both data and checksum on disk and does not generate or verify the checksum. In this way, only the applications (reader and writes) are responsible of verifying and generating the checksums, thus providing the straight-forward end-to-end data integrity.

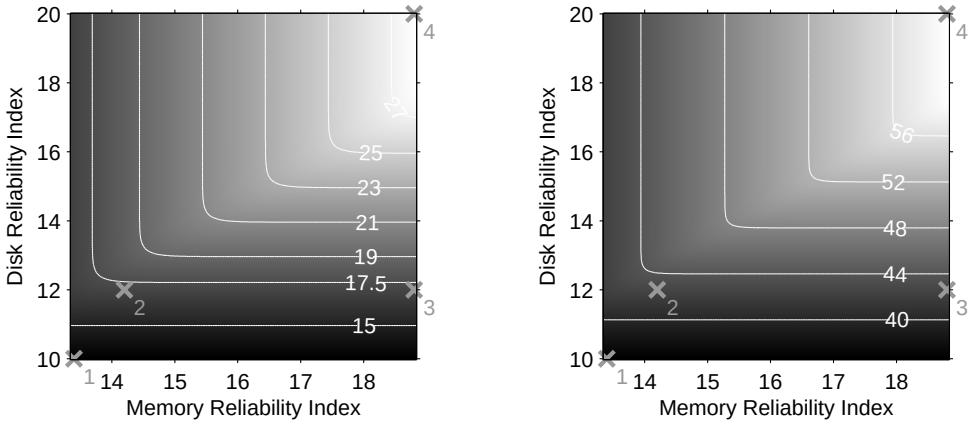
Reliability Analysis

The timeline of a data block from writer to reader is shown in Figure 4.5. E^{2}ZFS uses one type of checksum (xor or Fletcher) all the way through. The writer generates the checksum for the data block at t_0, and passes both the checksum and data block to the file system. Both are then written to disk at t_1 and read back at t_2. The reader receives them at t_3 and verifies the checksum.

In E^{2}ZFS, only undetected corruption during each time period causes a SDC; detected corruption would be caught by the checksum verification performed by the reader. The probability of undetected data corruption is:

$$\begin{aligned} P_{\text{E}^2\text{ZFS}-udc} = & P_{udc}(mem, Fletcher/xor, t_{resident}) \\ & + P_{udc}(dsk, Fletcher/xor) \\ & + P_{udc}(mem, Fletcher/xor, 30) \end{aligned}$$

The reliability scores of E^{2}ZFS (xor) and E^{2}ZFS (Fletcher) are shown in Figure 4.6(a) and Figure 4.6(b). Overall, E^{2}ZFS (Fletcher) has the best reliability, with all scores above the reliability goal. E^{2}ZFS (xor) can meet the goal only when



(a) E^{2}ZFS (xor) (b) E^{2}ZFS (Fletcher)

Figure 4.6: **E^{2}ZFS Reliability Score ($t_{resident} = 1$)** *These figures illustrate contour plots of the reliability score of E^{2}ZFS (xor) and E^{2}ZFS (Fletcher). Four points marked with a "$\times$" represent the four sample configurations: low-end (1), consumer (2), enterprise (3), server (4).*

System	TP (MB/s)	Normalized TP
ZFS	656.67	100%
E^{2}ZFS (Fletcher)	558.22	85%
E^{2}ZFS (xor)	639.89	97%

Table 4.3: **Overhead of Checksum Calculation** *This table shows the throughput of sequentially reading a 1GB file from the page cache in ZFS, E^{2}ZFS (xor), and E^{2}ZFS(Fletcher).*

both disk and memory are more reliable. Config 4 (server) has a score of 27.8 while both config 2 (consumer) and config 3 (enterprise) have a score of 17.1 (just short of Zettabyte reliability). Comparing both figures, when the disk corruption dominates (with an index below 12), E^{2}ZFS (Fletcher) is much better than E^{2}ZFS (xor), showing that Fletcher is clearly a better checksum for protecting blocks on disk.

Performance Issues

E^{2}ZFS (xor) is less reliable than E^{2}ZFS (Fletcher), but it offers better performance, especially when the reader is reading data from memory. Table 4.3 shows the throughput of reading a 1GB file from the page cache. As one can see, ZFS has the best throughput because there is no checksum calculation involved. E^{2}ZFS with Fletcher suffers a throughput drop of 15%. In contrast, E^{2}ZFS (xor) is able to achieve a throughput just 3% less than ZFS, with the checksum-on-copy optimization [39], which calculates the xor checksum while data is copied between kernel space and user space. The checksum-on-copy technique can be applied easily and efficiently due to the simplicity of xor checksum, but may not be a good option for stronger and more complex checksums such as Fletcher.

4.2.3 Z^{2}FS: ZFS with Flexible End-to-end Data Integrity

There are two drawbacks with the straight-forward end-to-end approach. Besides the performance problem as shown above, it also suffers from untimely recovery: neither the page cache nor the file system is able to verify the checksum to detect corruption in time. To handle both problems, we build Z^{2}FS on top of the changes we have made in E^{2}ZFS by further applying the concept of flexible end-to-end data integrity. For the timeliness problem, a simple fix is to add an extra verification when the data is being flushed to disk and when the data is being read from disk. For the performance problem, however, more analysis and techniques are required. We will focus on the performance problem in this section and discuss the timeliness problem in Section 4.3.

In this section, we will introduce two operation modes in Z^{2}FS: static mode, in which checksums are changed only across components (e.g., between memory and disk), and dynamic mode, where checksums are even changed overtime.

Static Mode with Checksum Chaining

Looking at the reliability score and performance figures of E^{2}ZFS, a natural question one may ask is: can we combine E^{2}ZFS (xor) and E^{2}ZFS (Fletcher) to make a system with better performance while still meeting the reliability goal? To answer this question, we introduce the static mode of Z^{2}FS, Z^{2}FS (static), a hybrid of E^{2}ZFS (xor) and E^{2}ZFS (Fletcher) that uses xor as the memory checksum and Fletcher as the disk checksum. In static mode, Z^{2}FS must perform a checksum conversion at the cache-disk boundary. To handle the conversion correctly, we develop a technique called *Checksum Chaining*, which carefully changes the checksum to avoid any vulnerable window.

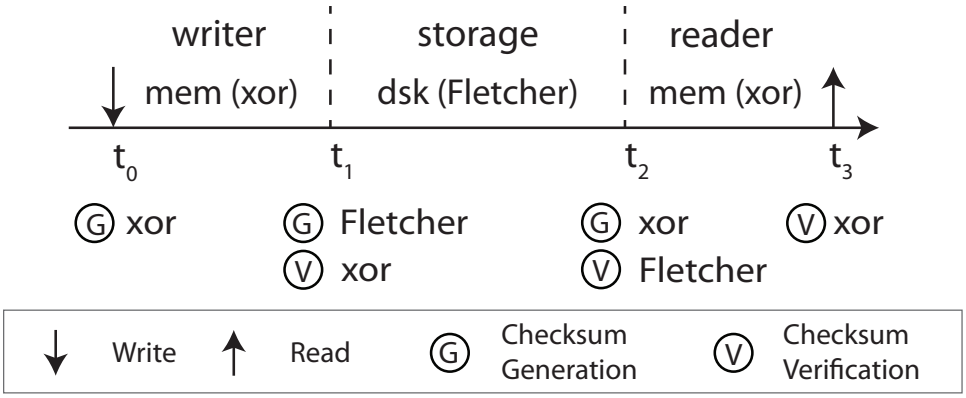


Figure 4.7: **Timeline of a Data Block in Z^{2}FS with Checksum Chaining** *This figure shows timeline of a block in Z^{2}FS with checksum chaining, which is applied at t_1 and t_2.*

Z^{2}FS (static) converts the checksum from xor to Fletcher when writing data to disk. With checksum chaining, it must generate the Fletcher checksum *before* verifying the xor checksum. In this way, the creation of the new Fletcher checksum occurs before the last use (verification) of the old xor checksum; the coverage of the new and old checksums overlaps. It is as if the two checksums are chained to each other. A successful verification of the xor checksum indicates that with high probability, the Fletcher checksum was generated over the correct data and thus Fletcher checksum is correct. If the order of generating Fletcher and verifying xor is reversed, there is a vulnerable window in between. If the data is corrupted in the window, the new Fletcher checksum will be calculated over the corrupted data, resulting in silent corruption, because the checksum actually "matches" the bad data.

The timeline of a data block in Z^{2}FS with checksum chaining is shown in Figure 4.7. On the write path, the writer generates an xor checksum at first. When the block is being written to disk, Z^{2}FS generates a Fletcher checksum using checksum chaining and sends the Fletcher checksum and data to disk. On the read path, Z^{2}FS generates an xor checksum using checksum chaining when reading the data block from disk, and then passes it to the reader along with the data block. The reader finally verifies the xor checksum. As a side effect of checksum chaining, the xor checksum is verified at the cache-disk boundary on the write path and the Fletcher checksum is verified on the read path, which helps to catch any detectable corruption in time.

With checksum chaining, Z^{2}FS has to generate an xor checksum for each data

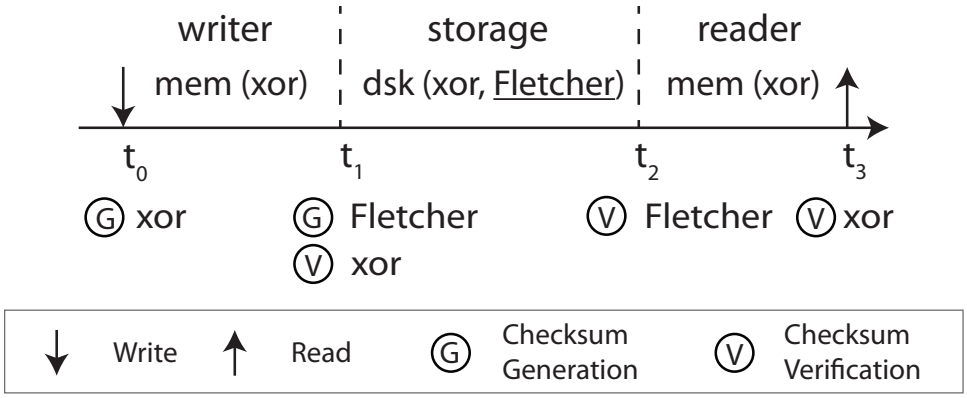


Figure 4.8: **Timeline of a Data Block in Z^{2}FS (static)** *This figure shows timeline of a block in Z^{2}FS (static). When there are two checksums during a time period, the underlined checksum is the primary checksum, as defined in Section 4.2.3.*

block when reading it from disk, which may affect the performance. In fact, the same xor checksum already existed when the data block was first written by the application. Instead of regenerating the xor checksum on every read, Z^{2}FS simply stores both the xor checksum and the Fletcher checksum on disk when writing a data block, and then when reading it, both checksums are available. The Fletcher checksum is called the *primary checksum*, because it is the required disk checksum. By grouping both checksums and storing them on disk, Z^{2}FS skips the generation of xor checksum on the read path, thus improving the performance. Note that Z^{2}FS still need to verify the primary checksum (Fletcher) when reading a block from disk.

Reliability Analysis of Static Mode

Figure 4.8 shows an updated timeline for Z^{2}FS (static) with this optimization. The probability of undetected corruption for Z^{2}FS (static) is:

$$\begin{aligned} P_{\text{Z}^2\text{FS}-udc} = & P_{udc}(mem, xor, t_{resident}) \\ & + P_{udc}(dsk, xor\&Fletcher) \\ & + P_{udc}(mem, xor, 30) \end{aligned}$$

Note that the corruption on disk must be undetectable by both xor and Fletcher. Since the block will be checked against the Fletcher checksum at t_2 and against the xor checksum at t_3, if either checksum catches the corruption, there will not be a silent data corruption.

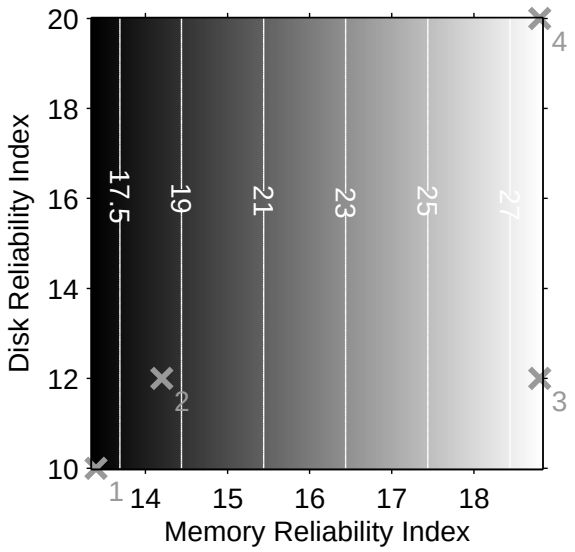
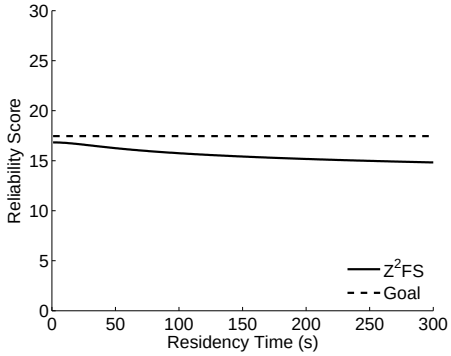


Figure 4.9: **Reliability Score ($t_{resident} = 1$) of Z^{2}FS (static)** *This graph is a contour plot of the reliability score of Z^{2}FS (static). Darker color means lower score - worse reliability. Four points marked with a "$\times$" represent the four sample configurations: low-end (1), consumer (2), enterprise (3), server (4).*

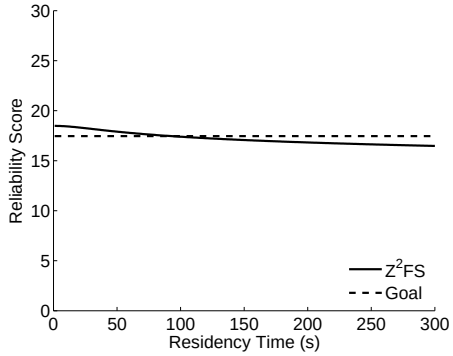
The reliability score of Z^{2}FS (static) at $t_{resident} = 1$ is shown in Figure 4.9. Since on-disk blocks are protected by Fletcher, memory corruption dominates. When memory corruption is severe with an index less than 13.7, the reliability score is below the goal. As the memory reliability index increases, the reliability score increases and becomes above the goal. However, as $t_{resident}$ increases, the reliability score will decrease and at some point it is possible to drop below the goal.

To find out when we should use Z^{2}FS (static), we focus on memory reliability and $t_{resident}$. We take a close look at three cases based on the memory reliability index: 13.4 ($\lambda_{max} = 1.99 \times 10^{-14}$), 14.2 ($\lambda_{mid} = 6.62 \times 10^{-15}$), and 18.8 ($\lambda_{min} = 1.48 \times 10^{-19}$). Since Figure 4.9 shows that memory corruption dominates, the value of the disk reliability index in each case does not affect the reliability score. Therefore, we fix the disk reliability index at 10 for the first case, and at 12 for second and third case; the three cases now correspond to config 1, 2 and 3 (low-end, consumer, and enterprise). Figure 4.10(a), Figure 4.10(b), and Figure 4.10(c) illustrate the reliability score of Z^{2}FS (static) versus residency time in all three cases.

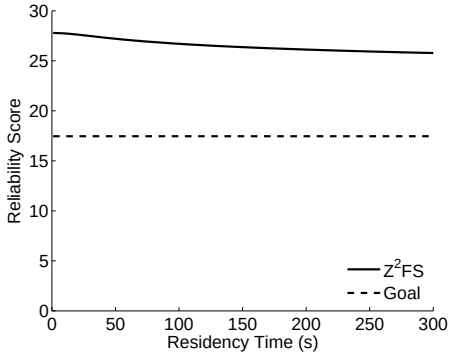
In Figure 4.10(c) where the memory reliable index is maximum, the reliability



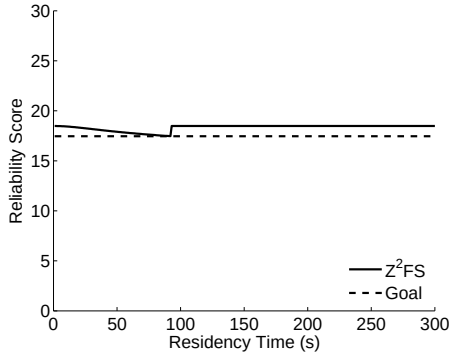
(a) λ_{max} (index = 13.4, config 1, low-end)



(b) λ_{mid} (index = 14.2, config 2, consumer)



(c) λ_{min} (index = 18.8, config 3, enterprise)



(d) λ_{mid} (index = 14.2, config 2, consumer)

Figure 4.10: **Reliability Score vs $t_{resident}$ in Z^{2}FS** *These figures show the relationship between reliability score and residency time in Z^{2}FS. The first three are for the static mode, and the last for the dynamic mode, in which the checksum switching occurs at 92 seconds.*

score is above the goal and they will intersect after about seven weeks (not shown). It indicates that xor is probably strong enough for data in memory; Z^{2}FS (static) fits right into this case.

In contrast, when the index is minimum as shown in Figure 4.10(a), the whole line of Z^{2}FS is below the goal. It shows that xor is not strong enough to protect data in memory. To handle this extreme case, Z^{2}FS (static) skips checksum chaining and uses Fletcher all the way through, but keeps the extra verification at the boundary of memory and disk. In this way, Z^{2}FS (static) can provide the same level of reliability as E^{2}ZFS (Fletcher).

The most interesting case is shown in Figure 4.10(b) with a memory reliability

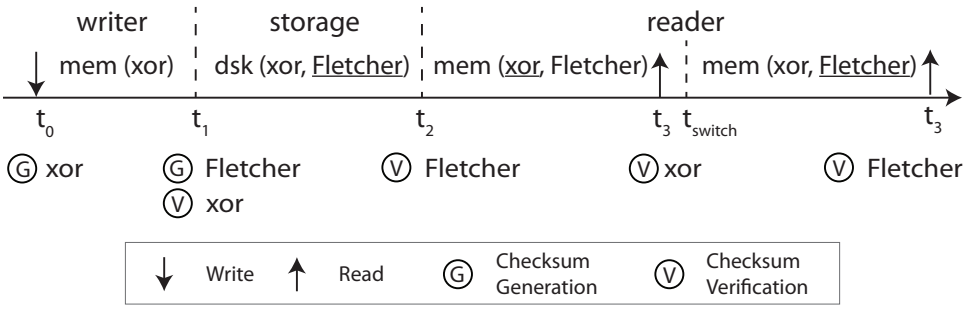


Figure 4.11: **Timeline of a Data Block in Z^{2}FS (dynamic)** *This figure shows timeline of a block in Z^{2}FS (dynamic). The memory checksum is switched from xor to Fletcher at t_switch.*

index of 14.2. When the residency time is less than 92 seconds, Z^{2}FS is able to keep the reliability score above the goal. However, after then the score drops below the goal and slowly converges to E^{2}ZFS (xor). In this case, in order to make sure the reliability score is always above the goal, Z^{2}FS may need to change to a stronger checksum at some point when data stays longer in memory.

Dynamic Mode with Checksum Switching

To prevent the reliability score from dropping below the goal as the residency time increases, we apply a technique called *Checksum Switching* to Z^{2}FS (static). The idea behind checksum switching is simple. On the read path, there are already two checksums on disk: xor and Fletcher. Z^{2}FS can simply read both checksums into memory; for the first t_{switch} seconds, Z^{2}FS uses xor as the *weaker memory checksum* and then switch to Fletcher as the *stronger memory checksum* after t_{switch} seconds. It is just a simple change of checksum and there is no extra overhead. We call this mode Z^{2}FS (dynamic).

Reliability Analysis of Dynamic Mode

Figure 4.11 shows the timeline of a block in Z^{2}FS (dynamic mode). The static mode is essentially a special case of dynamic mode with a extremely large value of t_{switch} such that t_3 is always in between t_2 and t_{switch}.

Calculating $P_{sys-udc}$ Depending on whether t_3 is before or after t_{switch}, we have:

$$\begin{aligned} P_{\text{Z}^2\text{FS}-udc} =& P_{udc}(mem, xor, t_{resident}) \\ &+ P_{udc}(dsk, xor\&Fletcher) \\ &+ P_{udc}(mem, xor, 30) \end{aligned}$$

where $t_3 = t_2 + t_{resident}$ is between t_2 and t_{switch}, and:

$$\begin{aligned} P_{\text{Z}^2\text{FS}-udc} =& P_{udc}(mem, Fletcher, t_{resident}) \\ &+ P_{udc}(dsk, Fletcher) \\ &+ P_{udc}(mem, xor, 30) \end{aligned}$$

where $t_3 = t_2 + t_{resident}$ is greater than t_{switch}.

Determining t_{switch} By replacing $t_{resident}$ in the first formula with t_{switch}, we can solve for t_{switch} from the equation below:

$$P_{\text{Z}^2\text{FS}-udc} = P_{goal}$$

With the Zettabyte reliability goal $P_{goal} = 3.46 \times 10^{-18}$ and λ_{mid}, we have $t_{switch} = 92$. Figure 4.10(d) shows the reliability score of Z^{2}FS in dynamic mode. As we can see from the figure, checksum switching occurs at 92 seconds so that the score afterwards is still above the goal.

By varying both the disk and memory reliability index, we have Figure 4.12 showing the values of t_{switch} that are required to meet the goal of Zettabyte reliability. When the memory reliability index is high ($\lambda = \lambda_{min}$, e.g., config 3 and 4), t_{switch} is about seven weeks; in this case, Z^{2}FS (static) is strong enough, which also offers the best performance. When the memory reliability index is extremely low (e.g., config 1), Z^{2}FS (static) keeps using Fletcher as both disk and memory checksum to provide the best reliability. When the memory reliability index is in between (e.g., config 2), Z^{2}FS (dynamic) strikes a nice balance between reliability and performance by switching the checksum at t_{switch}.

4.3 Discussion

We now discuss three technical issues when implementing Z^{2}FS: checksum chaining, application integration, and error handling.

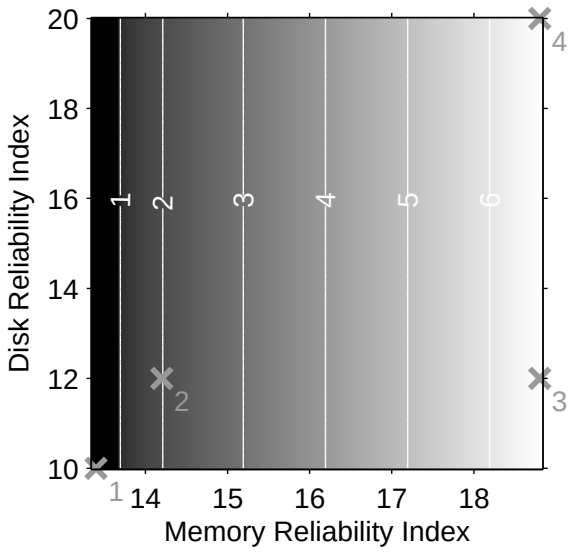


Figure 4.12: t_{switch} **of Z^{2}FS (dynamic)** *This figure shows a contour plot of the required switching time to provide Zettabyte reliability in Z^{2}FS (dynamic), with respect to different disk and memory reliability index. The z axis is the base 10 logarithm of t_{switch} in seconds. Four points marked with a "$\times$" represent the four sample configurations: low-end (1), consumer (2), enterprise (3), server (4).*

Symbol	Description
X	a data object, could be ORG or DST
$X.data$	the data of the object X
$X.cksum$	the checksum of object X
$X.size$	the size of $X.data$
$X.alg$	the checksum algorithm for $X.cksum$
S	size of moved data
$m(X)$	moved data in X
$o(X)$	overwritten data in X
$r(X)$	remaining data in X
$g(cksum, alg, data)$	generate $cksum$ using alg over $data$
$v(cksum, alg, data)$	verify $cksum$ using alg over $data$

Table 4.4: **Model Notation for Checksum Chaining** *The table depicts all notations used to describe the model for checksum chaining.*

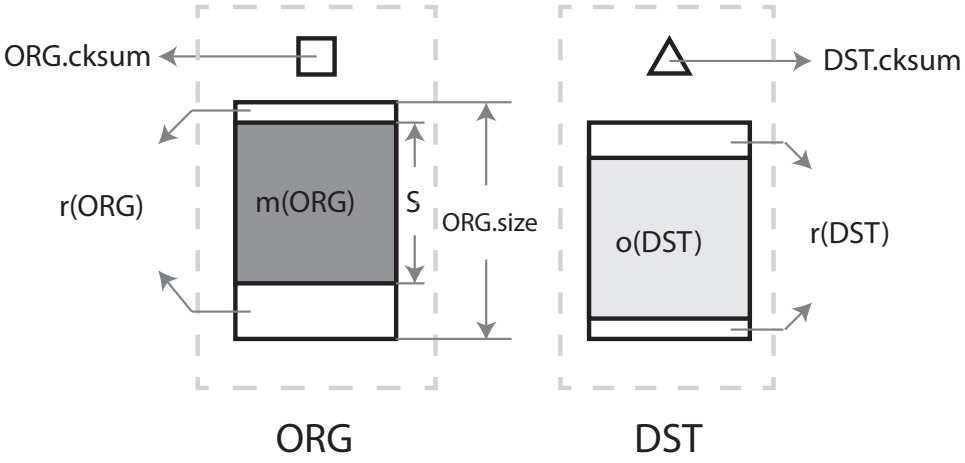


Figure 4.13: **An Example of the Notations** *This figure shows some of the notations in a data movement example. Small squares and triangles represent checksums. Different shapes of checksum symbol indicates the algorithm or the value of the checksum are different. Each big rectangle represents a data object over which a checksum is calculated. Heavy-shaded squares represent the moved data and light-shaded squared represent overwritten data.*

4.3.1 Checksum Chaining

So far, we have assumed the user buffer is always aligned to page size. In fact, checksum chaining does support generic requests with arbitrary offset and size, which is implemented in Z^{2}FS through checksum-ware interfaces. Before we talk about the new interfaces, we first we propose a simple model to characterize all scenarios where checksum chaining could apply when data is moved across buffers.

Notations In the model, data is always protected by a checksum. We use a data object to represent a piece of data and a corresponding checksum. Data in different data objects can be of different sizes and the checksum algorithms can also differ. Therefore, a data object has four properties: $data$, $cksum$, $size$ and alg.

Data movement is defined here as a piece of data moved from the origin data object ORG to destination data object DST. The moved data from ORG is represented by $m(ORG)$, and the overwritten data in DST is represented by $o(DST)$. The moved and overwritten data is of size S. In some cases, S may not be the same as $ORG.size$ or $DST.size$; some portion of data in ORG is not moved and some portion of data in DST is not overwritten. The remaining data is represented by $r(ORG)$ or $r(DST)$. All notations are explained in Figure 4.4 and illustrated in Figure 4.13.

During the data movement, $m(ORG)$ is copied from ORG to DST and the checksum of DST is updated. Checksum chaining is thus defined as follows: assuming D is the data stored in DST after the data movement, the new $DST.cksum$ is calculated over D *before* the integrity of D is verified using $ORG.cksum$ and the old $DST.cksum$. A special case of checksum chaining is when ORG and DST are of the same size, and ORG and DST use the same checksum algorithm. In this case, $ORG.cksum$ is copied to $DST.cksum$ directly when the moved data, without any recalculation. We call this special case checksum forwarding.

Checksum forwarding is straightforward and has no overhead except the copying of the checksum, but it has strict requirements for the alignment and checksum algorithms of the moved data, ORG and DST. In contrast, checksum chaining can be applied in any scenario, but it has the overhead of one or more checksum calculations.

In checksum chaining, the order of new checksum generation and old checksum verification must not be reversed. If $DST.cksum$ is calculated AFTER D is verified, there is a vulnerable window in between. If the data is corrupted in this time window, the new $DST.cksum$ will be calculated using corrupt data. This is a type of silent corruption which is undetectable using the new checksum because the checksum actually "matches" the corrupted data. With the correct order, a successful verification indicates that $DST.cksum$ is generated over the correct data and thus can be trusted. Because the creation of $DST.cksum$ occurs before the last use of $ORG.cksum$ and old $DST.cksum$, the coverage of new and old checksums overlaps; it is as if two checksums are chained to each other.

Five Cases of Checksum Chaining Data movement is not just a simple data copy operation. Transferring a piece of data from its initial origin to its final destination usually involves multiple copies through different layers of the system. The alignment and size of the moved data, as well as the size and checksum algorithm of ORG and DST in all layers are important factors. Depending on the S, and alg and $size$ of both ORG and DST objects, data movement can be classified into the following five cases, as shown in Figure 4.14. For each case, we first give the condition these properties must satisfy and then describe when and how checksum forwarding or chaining is applied in detail.

Case 1: Aligned Data Movement (Same Checksum Algorithms)

$$ORG.alg == DST.alg \text{ and}$$
$$S == ORG.size == DST.size$$

One example of Case 1 is transferring data blocks between the page cache and disk when both components use the same checksum. The size of a data page is

Figure 4.14: **Cases of Checksum Chaining** *This figure shows five typical cases of data movement. In Case 1, 1′ and 2, the moved data is aligned with ORG and DST. In Case 3, 4 and 5, the moved data is not aligned with ORG, DST or both, respectively. The size of moved data could be the same as $DST.size$ as in Case 1, 2 and 3, or different as in Case 4 and 5. The sequence of checksum chaining is shown as G and V operations in each case. The number of these operations is used as an estimate of the overhead.*

usually the same as a disk block, and data is always moved in full between them.

In this case, all data in ORG is copied to DST. Since the checksum algorithms are the same for both objects, one can apply checksum forwarding:

$$\text{(1) } DST.data \leftarrow ORG.data$$
$$\text{(2) } DST.cksum \leftarrow ORG.cksum$$

Before moving forward to Case 2, we introduce Case 1′, a more reliable version of Case 1 with an extra verification, as shown in Figure 4.14. Because checksum forwarding does not detect any corruption, doing such a verification provides an opportunity of early detection and in-time recovery. Otherwise, if the data is already corrupted, it will not be detected until the next time the data is accessed and verification is performed. In fact, this is a tradeoff between reliability and performance. With the overhead of one extra verification, possible corruptions can be detected early and repaired in time.

Note that Case 1 has the lowest overhead, because there is no checksum calculation involved. For Case 1′, as well as the next four cases, one can estimate the overhead by counting the number of checksum operations (generation and verification) needed in each case. Each of these operations are shown in Figure 4.14 as a circled G or V, respectively. To accurately measure the overhead, one needs to consider the size of data as well as the speed of the checksum algorithm.

Case 2: Aligned Data Movement (Different Checksum Algorithms)

$$ORG.alg \neq DST.alg \text{ and}$$
$$S == ORG.size == DST.size$$

In this case, since the checksum algorithms are different, $DST.cksum$ must be calculated using $DST.alg$. Checksum chaining should be applied:

(1) $g(DST.cksum, DST.alg, ORG.data)$
(2) $v(ORG.cksum, ORG.alg, ORG.data)$
(3) $DST.data \leftarrow ORG.data$

Case 3 Unaligned Data Movement (Partial-to-Full)

$$ORG.buf \neq DST.buf \text{ and}$$
$$ORG.size > DST.size \text{ and}$$
$$S == ORG.size$$

A good example of Case 3 is an application reading data from the page cache into a user buffer, with an offset not aligned to the block size (page size). In this example, ORG is a data page and DST is a user buffer. The moved data is just a portion of the full block stored in the page.

In this case, $DST.data$ is overwritten by a partial amount of $ORG.data$. Irrespective of the checksum algorithms used by ORG and DST, checksum chaining must be applied. A correct order is:

(1) $g(DST.cksum, DST.alg, m(DST))$
(2) $v(ORG.cksum, ORG.alg, ORG.data)$
(3) $DST.data \leftarrow m(ORG)$

Note that in (1) the checksum is calculated only over the moved data in ORG, while in (2) the verification is performed using all data in ORG, because $ORG.cksum$ covers all its data and there is no checksum for the moved data. Therefore, for the same S, the overhead of this case is actually higher than Case 2.

All cases introduced so far have one commonality: the original data in DST is overwritten by the new data copied from ORG, so there is no need to verify $DST.cksum$. The next two cases, however, have part of $DST.data$ overwritten by new data. Therefore, an extra verification is needed to make sure the portion of data in DST that is not modified is correct.

Case 4 Unaligned Data Movement (Full-to-Partial)

$ORG.size < DST.size$ and
$S = ORG.size$

Case 4 happens when an application writes data to the file system with an offset not aligned to the block size; the user buffer (ORG) is thus not aligned to the data page (DST), because only part of the data page is overwritten.

In this case, $ORG.data$ overwrites a part of $DST.data$. The net effect is that the new $DST.data$ contains $ORG.data$ and the remaining portion of old $DST.data$ is not overwritten. The new $DST.data$ is represented by $ORG.data + r(DST)$. Therefore, the new $DST.cksum$ must be calculated over $ORG.data + r(DST)$ before the data movement, as if $ORG.data$ were already copied to DST. To make sure both $ORG.data$ and $r(DST)$ are good while $DST.cksum$ is being calculated, they have to be verified. Therefore, the correct order of checksum chaining is:

(1) $g(tmpcksum, DST.alg, ORG.data + r(DST))$
(2) $v(ORG.cksum, ORG.alg, ORG.data)$
(3) $v(DST.cksum, DST.alg, DST.data)$
(4) $DST.cksum \leftarrow tmpcksum$
(5) $o(DST.data) \leftarrow ORG.data$

Unlike the previous cases, Case 4 requires two verifications, one over $ORG.data$ and the other over $DST.data$.

Case 5 Unaligned Data Movement (Partial-to-Partial)

$S \neq ORG.size$ and $S \neq DST.size$

This is the general case of unaligned data movement: part of $ORG.data$ is copied to DST and overwrites part of $DST.data$. The method of implementing checksum chaining is similar to Case 4, with a slight change to step (1) and step (5):

(1) $g(tmpcksum, DST.alg, m(ORG) + r(DST))$
(2) $v(ORG.cksum, ORG.alg, ORG.data)$
(3) $v(DST.cksum, DST.alg, DST.data)$
(4) $DST.cksum \leftarrow tmpcksum$
(5) $o(DST.data) \leftarrow m(ORG.data)$

Although this case does not occur in Z^{2}FS, we include Case 5 for the sake of completeness.

4.3.2 Integration with Existing Applications

First, Z^{2}FS supports generic requests with arbitrary offset and size through checksum-aware interfaces. These interfaces differ from the traditional read/write interfaces

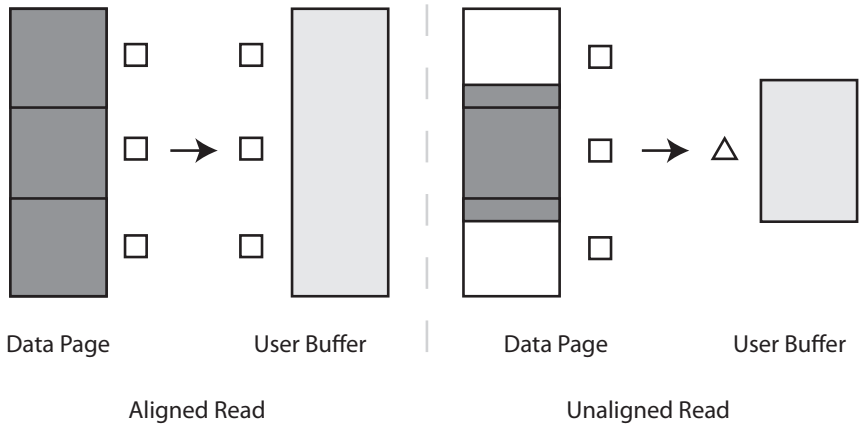


Figure 4.15: **Example of Aligned and Unaligned Reads** *This figure illustrates how Z^{2}FS handles aligned and unaligned reads. Small squares represent page checksums and small triangles represent user checksums. The dark area represents the requested data.*

in that both data and it associated checksums are transferred between the user space and the kernel space. For example, Figure 4.15 illustrates how Z^{2}FS handles aligned and generic read requests respectively. In the aligned case, Z^{2}FS simply returns all three checksums to the application. But when dealing with the unaligned reads, Z^{2}FS calculates a new checksum that covers the requested data and sends it to the application. The order of checksum generation and verification conforms with checksum chaining (see Case 3 and Case 4 above): generate the user checksum first and then verify all three page checksums. Note that the applications must be modified to use the new interfaces. We believe such changes are necessary, because the exposed checksums can be further utilized by applications to protect data at the user level.

Second, Z^{2}FS also provides a compatibility library that preserves the traditional interfaces. The library performs checksum generation and verification on behalf of the application. The tradeoff is that applications do not have access to the checksums, thus losing some data protection at the user level.

4.3.3 Error Handling

Both E^{2}ZFS and Z^{2}FS use checksums to verify data integrity. Whenever a mismatch happens, it is reasonable to think the data is corrupted, not the checksum, because the checksum is usually much smaller than the data it protects and has a lower chance of becoming corrupted. In the unusual case where the checksum is

corrupted, good data would be considered corrupted. This false positive about data corruption does not hurt data integrity; in fact, any checksum mismatch indicates that the data cannot be trusted, either because the data itself is corrupted, or because the checksum cannot prove the data is correct. Therefore, both systems must handle verification failures properly.

In E^{2}ZFS, there is only one verification, which occurs when the reader reads a data block. If the verification fails, the reader will re-read the same block from the file system. If the corruption happens in the page cache (reader's memory), E^{2}ZFS can get the correct data from disk and return it to the reader. However, if the corruption occurs before the block is written to disk on the write path, it is too late to recover from the corruption. This is the timeliness problem of the straightforward end-to-end approach.

As we mentioned in Section 4.2.3, to solve the problem, Z^{2}FS has extra checksum verifications at the boundary of memory and disk. On the write path, the verification is part of the checksum chaining. If it fails, Z^{2}FS aborts the write immediately and inform the application, thus preventing corrupt data going to disk. The application then can re-write the block. On the read path, Z^{2}FS verifies the primary checksum (Fletcher) after getting a data block from disk and will re-read it if the verification fails.

Note that informing the application about the failed write is quite challenging. It is easy for synchronous writes; because the verification occurs before the write system call returns, the application can just check the return value of the system call. However, for asynchronous writes, the verification is performed by the background flushing thread. To properly return the error information to the application, our solution in Z^{2}FS is to use a modified fsync system call. Z^{2}FS creates an error table for each opened file to record which data page fails the verification. Whenever *fsync* is called, it checks the error table of the corresponding file and returns all block numbers found in the table. Because at that time all verifications of dirty pages belonging to the file have already finished, *fsync* can give the most up-to-date error information. Therefore, by calling *fsync* periodically, the application can know the latest status of the blocks it wrote and perform necessary recovery in time.

4.4 Evaluation

We now evaluate and compare E^{2}ZFS and Z^{2}FS along two axes: reliability and performance. Specifically, we want to answer the following questions:

- How do they handle various data corruption?

Timing	ZFS act	ZFS res	E^{2}ZFS act	E^{2}ZFS res	Z^{2}FS act	Z^{2}FS res
$t_0 \sim t_1$	$-$	$\times$	d_3r	e	d_1r	$\surd$
$t_1 \sim t_2$	d_2r	e	d_3r	e	d_2r	e
$t_2 \sim t_3$	$-$	$\times$	d_3r	$\surd$	d_3r	$\surd$

Table 4.5: **Fault Injection Results** *The columns (from left to right) show the time period when the fault was injected (Timing), how the system and the reader reacts (act) and the result of the read request from the reader (res). Under the act column, "d_ir" means the corruption is detected at t_i and a retry is performed. Under the res column, "$\times$" means silent data corruption, "e" means the corruption is detected but can not be recovered (assuming there is only one copy of the data on disk), and "$\surd$" means the reader gets good data.*

- What is the the overall performance of both systems?
- What is the impact of checksum switching on performance?
- What is the performance of both systems on real-world workloads?

We perform all experiments on a machine with a single-core 2.2GHz AMD Opteron processor, 2GB memory, and a 1TB Hitachi Deskstar hard drive. We use Solaris Express Community Edition (build 108), ZFS pool version 14 and ZFS file system version 3.

4.4.1 Reliability

The analyses in Section 4.2 showed theoretically how Z^{2}FS can achieve Zettabyte Reliability with different reliability levels of disk and memory. In practice, however, it is difficult to experimentally measure the reliability of a system, especially since we have no knowledge of the actual failure rate of the disk and memory in use. Therefore, we focus on demonstrating the advantage of flexible end-to-end data integrity in detecting and recovering from corruption, through a series of fault injection experiments.

We inject a single bit flip to a data block during each time period in Figure 4.3, and record how each system reacts and whether the reader can get correct data. We perform the same set of experiments on all three systems, ZFS, E^{2}ZFS, and Z^{2}FS.

Table 4.5 summarizes the fault injection results. For the fault injected before the block goes to disk ($t_0 \sim t_1$), only Z^{2}FS is able to detect it before t_1 and ask the writer to retry, thus preventing corrupt data getting to disk. The reader in E^{2}ZFS

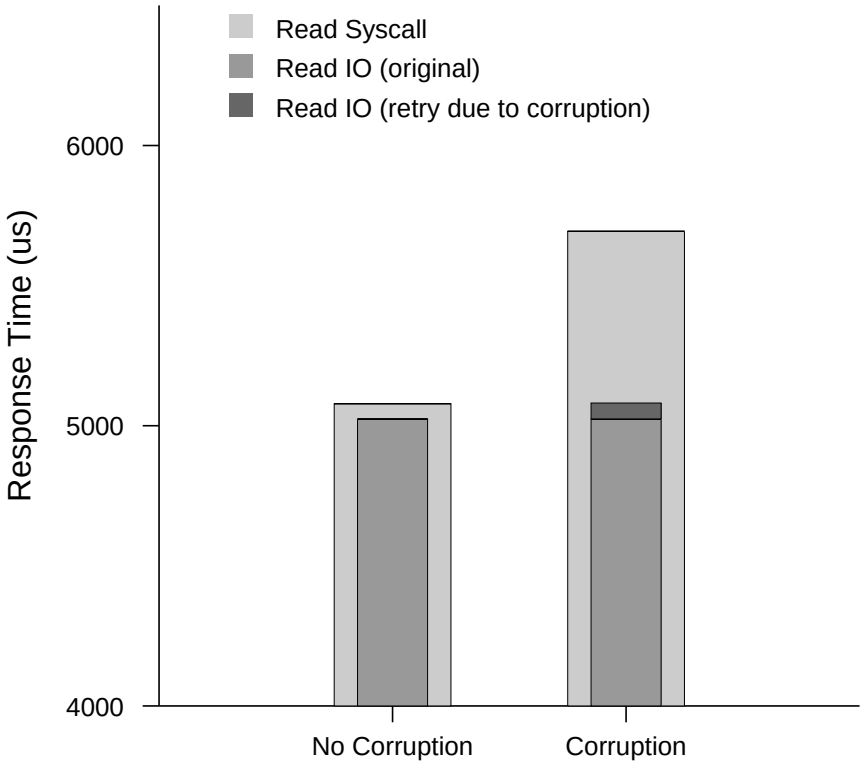


Figure 4.16: **Corruption in the Read Path (Cold)** *This graph shows the time breakdown of a read system call in Z^{2}FS when a block is correct or found corrupted in the page cache. The y-axis is in micro seconds. Since the cache is cold, the block is first read from disk.*

can also detect the fault at t_3, but it is too late to recover the data. When data on disk is corrupted ($t_1 \sim t_2$), neither E^{2}ZFS nor Z^{2}FS is able to recover. For the fault injected after the block leaves disk on the read path ($t_2 \sim t_3$), the reader in both Z^{2}FS and E^{2}ZFS can detect it and re-read the block from disk. Since ZFS only has protection for on-disk blocks, it can only catch corruption that occurs on disk.

To show that Z^{2}FS behaves as expected during the fault injection experiments, we measure the time cost of read and write system calls, as well as the I/O time of each disk read and write. Figure 4.16, 4.17, and 4.18 present the time breakdown of a read or a write system call in three cases: cold read, warm read and write with fsync.

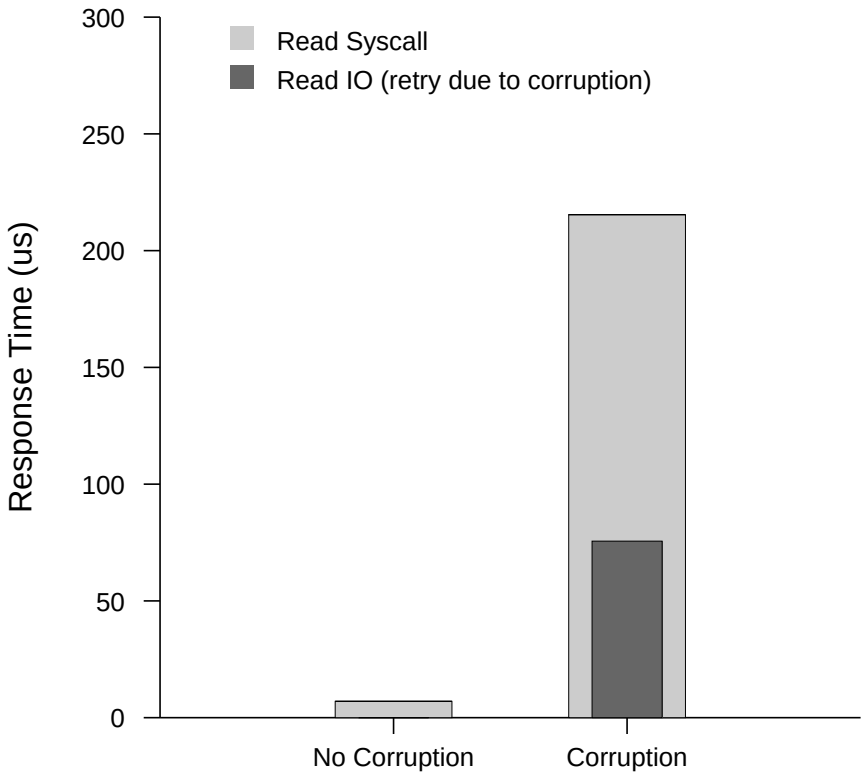


Figure 4.17: **Corruption in the Read Path (Warm)** *This graph shows the time breakdown of a read system call in Z^{2}FS when a block is correct or found corrupted in the page cache. The y-axis is in micro seconds. Since the cache is warm, the block can be returned directly from the page cache.*

Read (cold): In this case, the reader reads a 4KB block from Z^{2}FS and the block is not present in the page cache. We clear the disk cache at the beginning of our experiment so that the first read always gets the block from disk. When no fault is injected, there is only one I/O, which takes about 5000 micro seconds, as shown in Figure 4.16. When a fault is injected while the block is in the page cache, Z^{2}FS is able to detect the corruption and re-read the block from disk. Since the second read I/O hits disk cache, the actual I/O time is small, only about 60 micro seconds.

Read (warm): As shown in Figure 4.17, the result is similar to the previous case, except that there is no huge first-time I/O cost, because the requested block is already cached.

Write with fsync: In this case, the writer writes a 4KB block to Z^{2}FS and calls fsync immediately. When there is no corruption, the write system call returns instantly (the short white bar above the x-axis in Figure 4.18), because the write is asynchronous. The following fsync flushes the data block to disk and logs the write operation in a log block (totally two I/Os). Because both I/Os go to the disk cache, the I/O time is only about 120 micro seconds. Then, the file system issues a cache flush to the disk so that all blocks cached by the disk cache are forced to disk. The wait time for flush to finish is long, which dominates the response time of fsync. When the block is corrupted in the page cache, Z^{2}FS is able to detect the corruption before writing it out to disk. The writer gets an error code from fsync and calls write and fsync again to re-do the write, which are shown as the second set of bars on top of the previous failed fsync. Note that there is only one write I/O (log block) during the failed fsync, because the data block write is aborted.

4.4.2 Overall Performance

We use a series of micro and macro benchmarks to evaluate the performance of E^{2}ZFS and Z^{2}FS. All benchmarks are compiled with the compatibility library.

Micro Benchmark Figure 4.19 shows the results of our micro benchmark experiments. Sequential write/read is writing/reading a 1GB file in 4KB requests. Random write/read is writing/reading 100MB of a 1GB file in 4KB requests. To avoid the effect of checksum switching, Z^{2}FS is in static mode. From Figure 4.19, one can see that under random write and random read (cold), the performance of Z^{2}FS and E^{2}ZFS is close to ZFS. Because both workloads are dominated by disk seeks, the overhead of checksum calculation is small. In the cases where the cache is warm, since no physical I/Os are involved, the calculation of checksums dominates the processing time. E^{2}ZFS (Fletcher) is about 15-17% slower than ZFS, while both E^{2}ZFS (xor) and Z^{2}FS only have a 3% throughput drop. In sequential write and sequential read (cold), the performance of Z^{2}FS is comparable to E^{2}ZFS (Fletcher).

Macro Benchmark We use filebench [107] as our macro benchmark. We choose webserver, fileserver and varmail to evaluate the overall application performance on

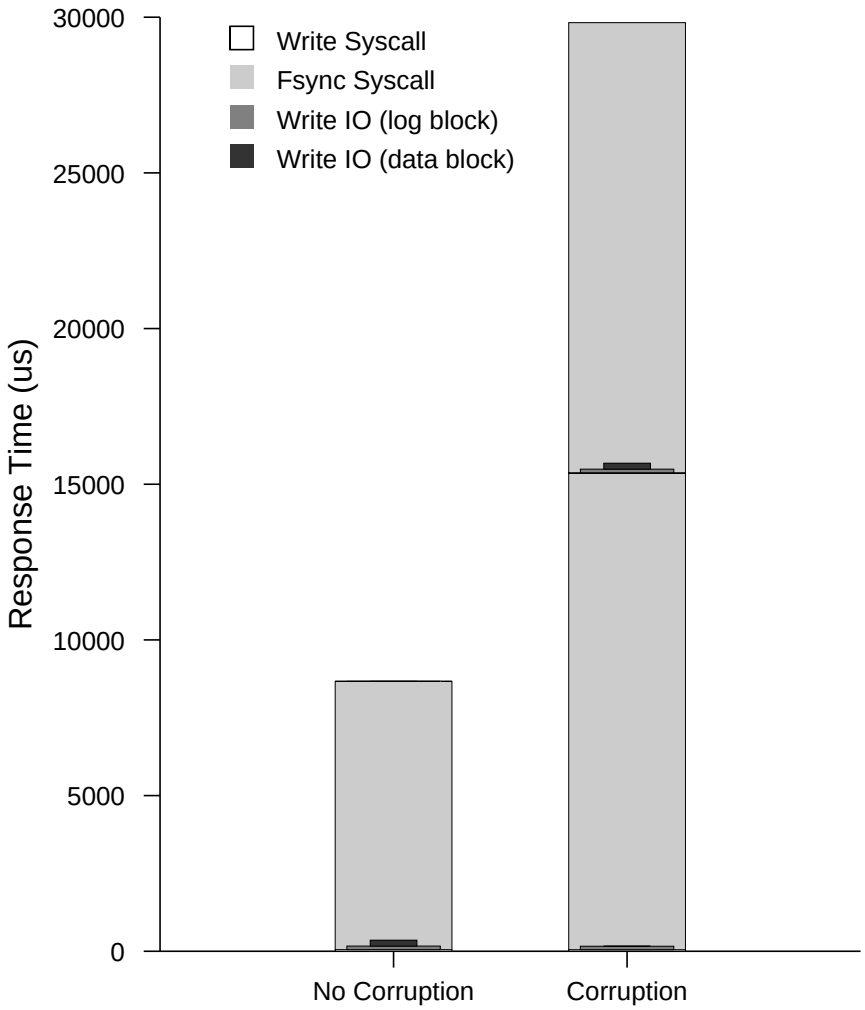


Figure 4.18: **Corruption in the Write Path** *This graph shows the time breakdown of a write system call followed by a fsync in Z^{2}FS when a block is correct or found corrupted in the page cache. The y-axis is in micro seconds.*

E^{2}ZFS and Z^{2}FS. Figure 4.20 depicts the throughput of these workloads.

Webserver is a multi-threaded read-intensive workload. It consists of 100 threads, each of which performs a series of open-read-close operations on multiple files and then appends to a log file. After reaching a steady state, all reads are satisfied by

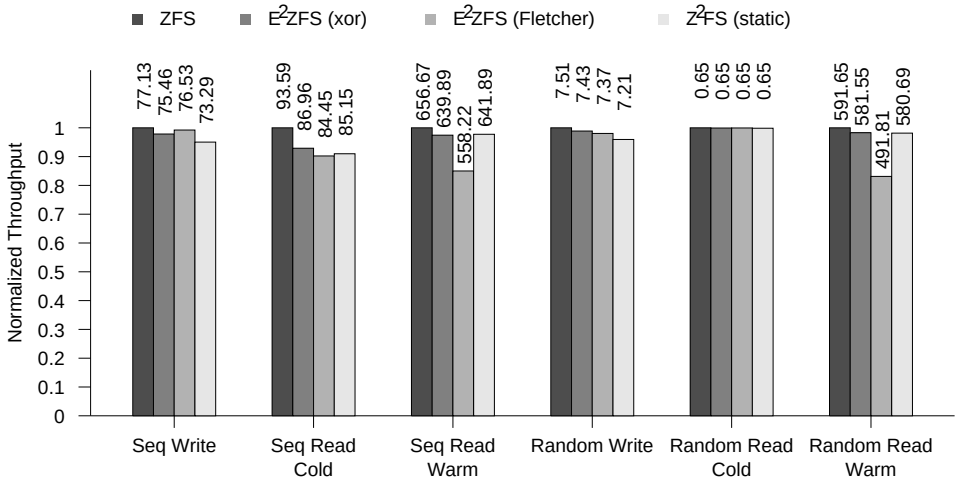


Figure 4.19: **Micro Benchmark** *This graph shows the results of several micro benchmarks on ZFS, E^{2}ZFS, and Z^{2}FS (static). The bars are normalized to the throughput of ZFS. The absolute values in MB/s are shown on top.*

data in the page cache. Therefore, the throughput is mainly determined by the overhead of checksum calculation. As shown in Figure 4.20, E^{2}ZFS (xor) and Z^{2}FS (static) has the closest performance to ZFS, because they always calculate the xor checksum. E^{2}ZFS (Fletcher) is about 15% percent slower than ZFS, which matches our previous micro benchmark result. In Z^{2}FS (dynamic), the memory checksum is changed from xor to Fletcher when a block stays in memory for more than 92 seconds, so the overall throughput is in between Z^{2}FS (static) and E^{2}ZFS (Fletcher).

Fileserver is configured with 50 threads performing creates, deletes, appends, whole-file writes and whole-file reads. It's write-intensive with a 1:2 read/write ratio. In this case, the throughput of Z^{2}FS is comparable to E^{2}ZFS (Fletcher) and E^{2}ZFS (xor).

Varmail emulates a multi-threaded mail server. Each thread performs a set of create-append-sync, read-append-sync, read, and delete operations. It has about half reads and half writes and is dominated by random I/Os. Therefore, the overall throughput of Z^{2}FS and E^{2}ZFS is no different than ZFS.

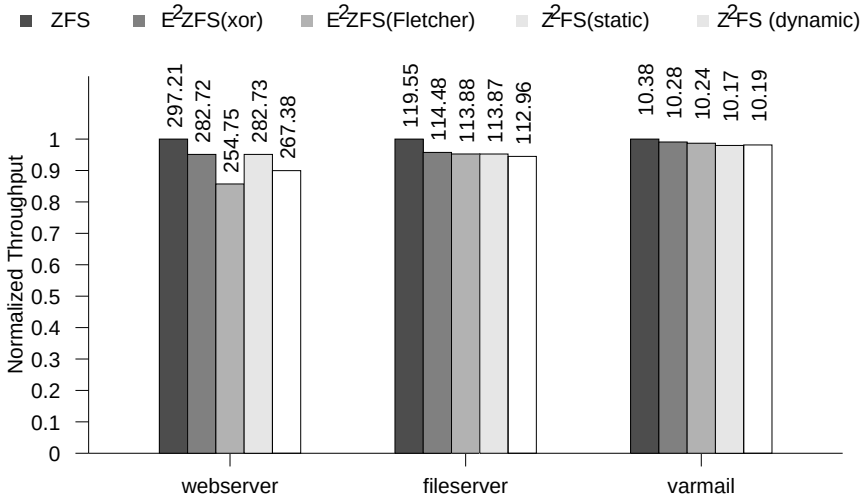


Figure 4.20: **Macro Benchmark** *This figure shows the throughput of our macro benchmarks on ZFS, E^{2}ZFS, Z^{2}FS (static), and Z^{2}FS (dynamic). Each workload runs for 720 seconds. Z^{2}FS (dynamic) has $t_{switch} = 92$ seconds.*

4.4.3 Impact of Checksum Switching

One key parameter in Z^{2}FS is t_{switch}, which is the maximum residency time of a data block in reader's memory before checksum switching occurs. The value of t_{switch} indicates a tradeoff between reliability and performance. Given a reliability goal, longer t_{switch} means worse reliability score (still above the goal), but better performance because the weaker memory checksum can be used for a longer time.

To understand the impact of checksum switching, we run the webserver workload on Z^{2}FS (dynamic) and vary t_{switch}. Figure 4.21 illustrates the relationship between the throughput of the workload and t_{switch}. As t_{switch} increases, the performance of Z^{2}FS (dynamic) gets closer to Z^{2}FS (static), because more and more warm reads are verifying the xor checksum. When t_{switch} is the same as or longer than the runtime, Z^{2}FS (dynamic) matches the performance of Z^{2}FS (static). Even when t_{switch} is short (e.g., 30 seconds), Z^{2}FS (dynamic) still outperfoms E^{2}ZFS (Fletcher).

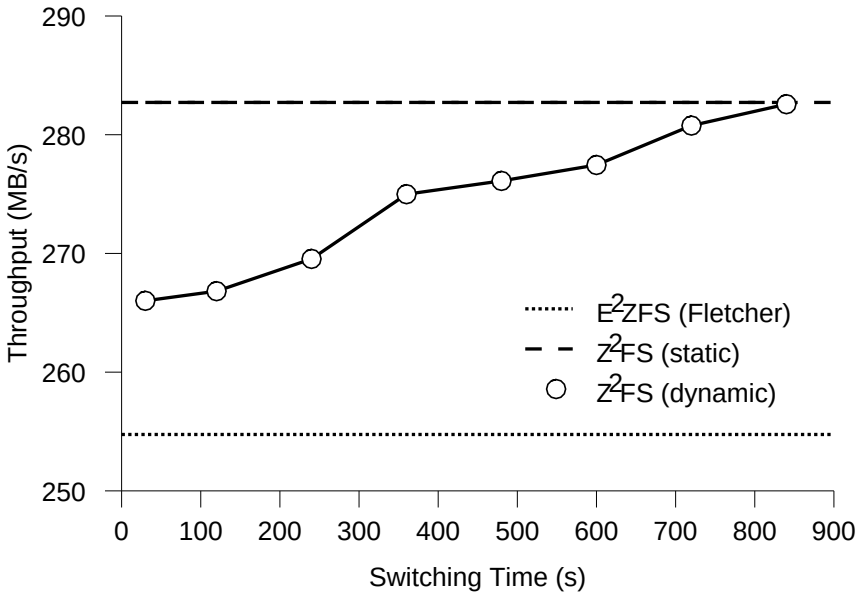


Figure 4.21: **Webserver Throughput with Different** t_{switch} *This figure illustrates the throughput changes of webserver as* t_{switch} *increases. The dashed line and dotted line represent the throughput of webserver on Z*^{2}*FS (static) and E*^{2}*ZFS (Fletcher) respectively. The runtime of the webserver workload is 720 seconds.*

Trace Num	Read Count	Cache Hit Rate	Before t_{switch}	After t_{switch}
1	14343	98.0%	34.5%	65.5%
2	35209	96.9%	58.9%	41.1%
3	61437	98.8%	83.7%	16.3%

Table 4.6: **Trace Characteristics** *Read count is the total number of 4KB-read in each trace. Hit rate is the cache hit rate for data reads. Before/After* t_{switch} *is the percentage of warm reads that access a data block with a residency time less/greater than* $t_{switch} = 92$ *seconds.*

4.4.4 Trace Replay

So far we have shown the performance benefit of Z^{2}FS using artificially generated workloads. Now, we evaluate Z^{2}FS by replaying real-world traces. We use the

Trace Num	Total Read Time (s) E^{2}ZFS (Fletcher)	Z^{2}FS (static)	Z^{2}FS (dynamic)
1	1.00	0.91 (9.0%)	0.95 (5.0%)
2	4.34	3.73 (14.1%)	3.82 (12.0%)
3	6.58	5.46 (17.0%)	5.47 (16.9%)

Table 4.7: **Trace Replay Result** *The table shows the total time spent on read system calls for each trace on each system. The percentage in the parentheses is the speedup of Z^{2}FS with respect to E^{2}ZFS (Fletcher).*

LASR system-call traces [6] collected between 2000 and 2001, which cover thirteen machines used for software development and research projects. The traces are not I/O intensive, but they contain realistic access patterns that are hard to emulate with controlled benchmarks. We build a single-threaded trace replayer to sequentially replay the system calls at the same speed as they were recorded. All unaligned read and write requests are converted into aligned ones such that we can replay the trace on E^{2}ZFS, which only supports aligned requests.

We choose three one-hour long traces from the collection and replay them on E^{2}ZFS (Fletcher), Z^{2}FS (static), and Z^{2}FS (dynamic, t_{switch} = 92). The characteristics of the traces are listed in Table 4.6 and the results are shown in Table 4.7. As one can see from the tables, overall, Z^{2}FS has better performance than E^{2}ZFS (Fletcher). In trace 3, most of the warm reads (83.7%) are accessing data blocks with a residency time less than 92 seconds, and thus there are more calculations of xor checksum than Fletcher on Z^{2}FS (dynamic), which makes its performance closer to Z^{2}FS (static). In contrast, 65.5% of the warms reads in trace 1 are of blocks that have stayed in memory for more than 92 seconds, so the performance of Z^{2}FS (dynamic) is closer to E^{2}ZFS (Fletcher). Therefore, workloads dominated by warm reads can benefit most from Z^{2}FS (dynamic) if most read accesses to a block occur during the first t_{switch} seconds of that block in memory.

4.5 Summary

The straight-forward approach of end-to-end data integrity provides great protection against corruption, but the requirement of using one strong high-level checksum for all components along the I/O path leads to lower application performance

and untimely detection and recovery.

To address these issues, we present a new concept: flexible end-to-end data integrity. A system with flexible end-to-end data integrity uses different checksum algorithms for different component, and thus can dynamically make tradeoffs between performance and reliability. Such a system also utilizes extra checksum verification below the application to provide in-time detection and recovery. In this way, all components in the I/O path provide strong data protection in a cooperative manner; every component is aware of the checksums and performs necessary checksum operations, such as generation, verification, switching or passing, to prevent silent data corruption.

To apply the concept to a system, we first develop an analytical framework to provide rational behind flexible end-to-end data integrity. Then, we build E^2ZFS and Z^2FS, to study both end-to-end concepts and demonstrate how to apply flexible end-to-end data integrity to ZFS. Through reliability analysis and various experiments, we show that Z^2FS is able to provide Zettabyte reliability with comparable or better performance than E^2ZFS. Our analysis framework provides a holistic way to reason about the tradeoff between performance and reliability in storage systems.

Chapter 5

Data Protection Analysis of Cloud Storage Services

Cloud-based file synchronization services, such as Dropbox [44], SkyDrive [122], and Google Drive [53], provide a convenient means both to synchronize data across a user's devices and to back up data in the cloud. While automatic synchronization of files is a key feature of these services, the reliable cloud storage they offer is fundamental to their success. Generally, the cloud backend will checksum and replicate its data to provide integrity [18] and will retain old versions of files to offer recovery from mistakes or inadvertent deletion [44]. The robustness of these data protection features, along with the inherent replication that synchronization provides, can give the user with a strong sense of data safety.

Unfortunately, this is merely a sense, not a reality; the loose coupling of these services and the local file system endangers data even as these services strive to protect it. While the data stored remotely is generally robust, local client software is unable to distinguish between deliberate modifications and unintentional errors, potentially causing corrupt or inconsistent data to automatically propagate to all machines associated with a user. Thus, despite the presence of multiple redundant copies, synchronization destroys the user's data.

In this chapter, we demonstrate these problems through fault injection experiments. We first present some background on file synchronization services in Section 5.1. Then, in Section 5.2 we explore several case studies wherein synchronization services propagate corruption and spread inconsistency. Finally, we analyze how the limitations of file synchronization services and file systems directly cause these problems in Section 5.3.

5.1 Background

In order to understand the causes of the incorrect behavior of file synchronization services, it is necessary to first understand how they operate. File synchronization services are aptly named; they do their best to ensure that their users' files are synchronized across all of their devices, as well as the cloud. While their design space has some variety in it, ranging from Apple's iCloud synchronizing specific application data [20] to Wuala's use of a user-space file system [123], the basic functionality of these services is relatively homogeneous. We find that there are two popular ways of implementing such a service, based on the underlying synchronization protocol. Services such as Dropbox and ownCloud rely on a specific file synchronization protocol, rsync [93] and csync [41] respectively. On the other hand, many open-source synchronization services, including Seafile [99] and sparkleshare [103], are built on top of distributed version control systems such as GIT [52]. Thus, we provide a brief case study of Dropbox and Seafile to cover both types of services; while the details are application-specific, the overall architecture applies to a variety of services.

5.1.1 Dropbox

Dropbox consists of two main components: a client-side daemon and a cloud backend. The daemon monitors changes in the local file system and uploads them to the cloud. The cloud software, in turn, stores these files and then propagates them to the user's other devices. As the cloud component runs remotely, we can only infer its characteristics through interacting with it via the network and through what Dropbox has published about it. As Drago et al. [43] have already examined many of these details elsewhere, we focus primarily on the client in our discussion. While the client is closed source, since it runs locally, we can directly observe its behavior. In the following discussion, we concentrate on two aspects of this behavior: how it manages its internal metadata and its procedures for synchronizing files.

Data Management

The Dropbox client operates as a userspace daemon, requiring no direct operating system support or kernel modules, and observes a single folder, ensuring that its contents are synchronized with the cloud. To track local states, it uses several SQLite databases, most of which are encrypted. These databases store metadata related to the user's files, such as the most recent time each file was modified, as well as hashes of each file used to identify their contents. Dropbox uses this

information to coordinate its synchronization with the cloud.

Dropbox's view of the user's file namespace is much more simple than that of the file system. It identifies files by their full pathnames and does not represent directories in its database. If the user performs a rename of a file, it deletes the file from the cloud and re-uploads the renamed version; similarly, if the user deletes a directory, the client deletes all children of that directory and re-uploads them, identified by their new full pathname.

Dropbox provides a revision history for each file that it tracks, allowing a user to revert a file to any of its previously uploaded states, within certain time limits depending on the level of the user's subscription. While useful, Dropbox's constrained view of the file system limits the extent of this history. In particular, renamed files cannot explicitly be reverted to prior versions before they were renamed. Instead, the user must restore the file of the original name and delete the renamed file.

File Synchronization

Upon booting, the Dropbox client registers with the cloud and checks whether any files have changed or been added remotely. If so, it downloads them into a staging area and renames them into the local directory once complete, so that the user never sees an incomplete update. At the same time, it also scans the local directory to detect whether any modifications have occurred while it was offline, comparing stats such as timestamps and size of each file with the version stored in its databases. If these differ, it infers that the file was changed and runs rsync to upload the changes to the cloud; to save bandwidth, it divides files into chunks and only sends those chunks not already owned by the user. In the event that it detects a conflict between two versions of a file, it performs no resolution; instead, it keeps both versions of the file and renames one to indicate that it is in conflict.

Once running, the Dropbox client continues to actively synchronize its folder. When remote changes occur, the server sends it a notification, causing the client to immediately download the new data in the same manner as the initial upload. To detect local changes, the client employs a notification service, such as Linux's inotify, that informs it of events in the local file system. This information is generally vague—inotify, for instance, reports little more than the file name and the type of event, such as a create, write, or unlink, that occurred–but suffices to allow Dropbox to maintain synchrony. Again, the client uses rsync to upload only the changes in each file and performs deduplication.

5.1.2 Seafile

Similar to Dropbox, Seafile also has a client-side daemon and a server backend. Unlike Dropbox, which interacts with files in the file system directly, Seafile maintains a GIT-like repository (repo) to manage a synchronized folder. A local synchronized folder is called a working tree. Seafile tracks and stores updates of the folder in local and remote repositories. The remote repo on the server holds the master branch, acting as a backend to store all data and version histories. The local repo contains the local branch, representing the current state of the folder. The synchronization is then performed between the master branch and the local branch.

Data Management

Unlike Dropbox, which only records file metadata in a local database, Seafile uses repos to track both data and metadata. A repo is essentially an object store. Files and directories in the folder are all stored as objects in the store, identified by SHA-1 hashes. A file's data is divided into chunks with variable length. A file is represented by a Seafile Object which stores a list of hashes of data chunks. A directory is represented by a SeafDir Object containing a list of directory entries, each of which points to a Seafile Object or a SeafDir Object. The hash of the root directory in the folder is called a commit ID, which uniquely represents a state of the entire folder. Therefore, the history of changes to a folder is recorded as a series of commit IDs. Similarly, the revision history of each file is tracked by a series of hash values of its Seafile objects.

The remote repo maintains the complete version history for synchronized files, including all the previously used but unreferenced data chunks. The client repository, on the other hand, only keeps a short history of changes. Unused data chunks are garbage collected at the beginning of each run of the local Seafile client daemon. At any time, the master branch points to a remote commit ID on the server and the local branch points to the latest local commit ID on the client.

File Synchronization

A Seafile client daemon runs on the client and monitors both the local folder and server for updates. When there are local changes, the client commits the changes to the local branch and then synchronizes the local branch to the server. When there are remote changes, the client first downloads the master branch from the server, then commits local changes, and finally merges the master branch into the local branch. The client performs conflict handling during the merge, in which a conflicting copy from the master branch is renamed and then committed to the

local branch. After the merge, the client uploads the local branch to the server, including all the regular local changes and changes due to conflicts. Finally, the master branch is updated to point to the state just uploaded.

Seafile client detects offline changes in a way similar to Dropbox. After every commit, it records in a local index file various stats of every file in the folder, including modification time and file size. When the client starts, it performs a local scan to find out if there are offline changes. This process involves checking every file in the folder and comparing timestamps against the ones in the index file.

When the client is running, it monitors both the local folder and the server for updates. For local changes, Seafile client relies on inotify, but it only uses inotify as an indicator. It still depends on a scan to find out what files and directories were modified. In comparison, Dropbox makes fully use of inotify to detect local changes. The client detects remote updates by polling the server every 30 seconds. The client checks if the commit ID of the local branch differs from the commit ID of the master branch. If they differ, it means that there are remote changes. Since there is no remote scan, the polling process is fast and efficient.

5.2 Data Protection Failures

We now present three case studies to show different failures caused by the semantic gap between local file systems and synchronization services. The first two of these failures, the propagation of corruption and inconsistency, result from the client's inability to distinguish between legitimate changes and failures of the file system. While these problems can be warded off by using more advanced file systems, the third, causal inconsistency, is a fundamental result of current file-system semantics.

5.2.1 Data Corruption

Data corruption is not uncommon and can result from a variety of causes, ranging from disk faults to operating system bugs [23, 38, 47, 89]. Corruption can be disastrous, and one might hope that the automatic backups that synchronization services provide would offer some protection from it. These backups, however, make them likely to propagate this corruption; as clients cannot detect corruption, they simply spread it to all of a user's copies, potentially leading to irrevocable data loss.

To investigate what might cause disk corruption to propagate to the cloud, we first inject a disk corruption to a block in a file synchronized with the cloud (by flipping bits through the device file of the underlying disk). We then manipulate the file in several different ways, and observe which modifications cause the corruption

FS	Service	Data write	Metadata mtime	Metadata ctime	Metadata atime
ext4 (Linux)	Dropbox	LG	LG	LG	L
	ownCloud	LG	LG	L	L
	Seafile	LG	LG	LG	LG
ZFS (Linux)	Dropbox	L	L	L	L
	ownCloud	L	L	L	L
	Seafile	L	L	L	L
HFS+ (Mac OS X)	Dropbox	LG	LG	L	L
	ownCloud	LG	LG	L	L
	GoogleDrive	LG	LG	L	L
	SugarSync	LG	L	L	L
	Syncplicity	LG	LG	L	L

Table 5.1: **Data Corruption Results** *"L": corruption remains local. "G": corruption is propagated (global).*

to be uploaded. We repeat this experiment for Dropbox, ownCloud, and Seafile atop ext4 (both ordered and data journaling modes) and ZFS [15] in Linux (kernel 3.6.11) and Dropbox, ownCloud, Google Drive, SugarSync, and Syncplicity atop HFS+ in Mac OS X (10.5 Lion).

We execute both data operations and metadata-only operations on the corrupt file. Data operations consist of both appends and in-place updates at varying distances from the corrupt block, updating both the modification and access times; these operations never overwrite the corruption. Metadata operations change only the timestamps of the file. We use *touch -a* to set the access time, *touch -m* to set the modification time, and *chown* and *chmod* to set the attribute-change time.

Table 5.1 displays our results for each combination of file systems and services. Since ZFS is able to detect local corruption, none of the synchronization clients propagate corruption. However, on ext4 and HFS+, all clients propagate corruption to the cloud whenever they detect a change to file data and most do so when the modification time is changed, even if the file is otherwise unmodified. In both cases, clients interpret the corrupted block as a legitimate change and upload it. Seafile uploads the corruption whenever any of the timestamps changes. SugarSync is the only service that does not propagate corruption when the modification time changes, doing so only once it explicitly observes a write to the file or it restarts.

FS	Service	Upload local ver.	Download cloud ver.	OOS
ext4 (ordered)	Dropbox	√	×	√
	ownCloud	√	√	√
	Seafile	N/A	N/A	N/A
ext4 (data)	Dropbox	√	×	×
	ownCloud	√	√	×
	Seafile	√	×	×
ZFS	Dropbox	√	×	×
	ownCloud	√	√	×
	Seafile	√	×	×

Table 5.2: **Crash Consistency Results** *There are three outcomes: uploading the local (possibly inconsistent) version to cloud, downloading the cloud version, and OOS (out-of-sync), in which the local version and the cloud version differ but are not synchronized. "×" means the outcome does not occur and "√" means the outcome occurs. Because in some cases the Seafile client fails to run after the crash, its results are labeled "N/A".*

5.2.2 Crash Inconsistency

The inability of synchronization services to identify legitimate changes also leads them to propagate inconsistent data after the crash recovery. To demonstrate this behavior, we initialize a synchronized file on disk and in the cloud at version v_0. We then write a new version, v_1, and inject a crash which may result in an inconsistent version $v_1\prime$ on disk, with mixed data from v_0 and v_1, but the metadata remains v_0. We observe the client's behavior as the system recovers. We perform this experiment with Dropbox, ownCloud, and Seafile on ZFS and ext4.

Table 5.2 shows our results. Running the synchronization service on top of ext4 with ordered journaling produces erratic and inconsistent behavior for both Dropbox and ownCloud. Dropbox may either upload the local, inconsistent version of the file or simply fail to synchronize it, depending on whether it had noticed and recorded the update in its internal structures before the crash. In addition to these outcomes, ownCloud may also download the version of the file stored in the cloud if it successfully synchronized the file prior to the crash. Seafile arguably exhibits the best behavior. After recovering from the crash, the client refuses to run, as it detects that its internal metadata is corrupted. Manually clearing the client's metadata and resynchronizing the folder allows the client to run again; at this point, it detects a conflict between the local file and the cloud version.

All three services behave correctly on ZFS and ext4 with data journaling. Since the local file system provides strong crash consistency, after crash recovery, the local version of the file is always consistent (either v_0 or v_1). Regardless of the version of the local file, both Dropbox and Seafile always upload the local version to the cloud when it differs from the cloud version. OwnCloud, however, will download the cloud version if the local version is v_0 and the cloud version is v_1. This behavior is correct for crash consistency, but it may violate causal consistency, as we will discuss.

5.2.3 Causal Inconsistency

The previous problems occur primarily because the file system fails to ensure a key property—either data integrity or consistency—and does not expose this failure to the file synchronization client. In contrast, causal inconsistency derives not from a specific failing on the file system's part, but from a direct consequence of traditional file system semantics. Because the client is unable to obtain a unified view of the file system at a single point in time, the client has to upload files as they change in piecemeal fashion, and the order in which it uploads files may not correspond to the order in which they were changed. Thus, file synchronization services can only guarantee eventual consistency: given time, the image stored in the cloud will match the disk image. However, if the client is interrupted—for instance, by a crash, or even a deliberate powerdown—the image stored remotely may not capture the causal ordering between writes in the file system enforced by primitives like POSIX's `sync` and `fsync`, resulting in a state that could not occur during normal operations.

To investigate this problem, we run a simple experiment in which a series of files are written to a synchronization folder in a specified order (enforced by fsync). During multiple runs, we vary the size of each file, as well as the time between file writes, and check if these files are uploaded to the cloud in the correct order. We perform this experiment with Dropbox, ownCloud, and Seafile on ext4 and ZFS, and find that for all setups, there are always cases in which the cloud state does not preserve the causal ordering of file writes.

While causal inconsistency is unlikely to directly cause data loss, it may lead to unexpected application behavior or failure. For instance, suppose the user employs a file synchronization service to store the library of a photo-editing suite that stores photos as both full images and thumbnails, using separate files for each. When the user edits a photo, and thus, the corresponding thumbnail as well, it is entirely possible that the synchronization service will upload the smaller thumbnail file first. If a fatal crash, such as a hard-drive failure, occurs before the client can

finish uploading the photo, then the service will still retain the thumbnail in its cloud storage, along with the original version of the photo, and will propagate this thumbnail to the other devices linked to the account. The user, accessing one of these devices and browsing through their thumbnail gallery to determine whether their data was preserved, is likely to see the new thumbnail and assume that the file was safely backed up before the crash. The resultant mismatch will likely lead to confusion when the user fully reopens the file later.

5.3 Discussion

Our experiments demonstrate genuine problems with file synchronization services; in many cases, they not only fail to prevent corruption and inconsistency, but actively spread them. Responsibility for preventing corruption and inconsistency hardly rests with synchronization services alone; much of the blame can be placed on local file systems, as well. In this section, we analyze the limitations in synchronization services and local file systems and show how they lead to data protection failures.

5.3.1 Where Synchronization Services Fail

Most synchronization services monitor its synchronization folder for changes using a file-system notification service, such as Linux's inotify or Mac OS X's Events API. While these services inform the synchronization clients of both namespace changes and changes to file content, they provide this information at a fairly coarse granularity—per file, for inotify, and per directory for the Events API, for instance. In the event that these services fail, the machine crashes, or the client itself fails or is closed for a time, then the client detects changes in local files by examining their statistics, including size and modification timestamps.

Given this behavior, the causes of synchronization services' inability to handle corruption and inconsistency become apparent. As file-system notification services provide no information on what file contents have changed, the synchronization client must assume that any changes that it detects result from legitimate user action; it has no means of distinguishing unintentional changes, like corruption and inconsistent crash recovery.

Inconsistent crash recovery is further complicated by the client's internal metadata tracking. For example, with Dropbox, if the system crashes during an upload and restores the file to an inconsistent state, the client will recognize that it needs to resume uploading the file, but it cannot detect that the contents are no longer

FS	Corruption	Crash	Causal
ext4 (ordered)	×	×	×
ext4 (data)	×	√	×
ZFS	√	√	×

Table 5.3: **Summary of File System Capabilities** *This table shows the synchronization failures each file system is able to handle correctly. There are three types of failures: Corruption (data corruption), Crash (crash inconsistency), and Causal (causal inconsistency). "√" means the failure does not occur and "×" means the failure may occur.*

consistent. Conversely, if Dropbox had finished uploading and updated its internal timestamps, but the crash recovery reverted the file's metadata to an older version, Dropbox must upload the file, since the differing timestamp could potentially indicate a legitimate change.

5.3.2 Where Local File Systems Fail

File systems frequently fail to take the preventative measures necessary to avoid data protection failures and, in addition, fail to expose adequate interfaces to allow synchronization services to deal with them. As summarized in Table 5.3, neither a traditional file system, ext4, nor a modern file system, ZFS, is able to avoid all failures.

File systems primarily prevent corruption via checksums. When writing a data or metadata item to disk, the file system stores a checksum over the item as well. Then, when it reads that item back in, it reads the checksum and uses that to validate the item's contents. While this technique correctly detects corruption, file system support for it is limited. ZFS and btrfs are some of the few widely available file systems that employ checksums over the whole file system; ext4 uses checksums, but only over metadata [40]. Even with checksums, however, the file system can only detect corruption, requiring other mechanisms to repair it.

Recovering from crashes without exposing inconsistency to the user is a problem that has dogged file systems since their earliest days, and has been addressed with a variety of solutions, such as journaling and copy-on-write. However, as discussed in Chapter 2, the most popular file systems, including ext3, ext4, HFS+, and NTFS, usually only perform metadata journaling, sacrificing data consistency for performance. As a result, the inconsistencies upon a crash cause the erratic behavior observed in Section 5.2.2.

Finally, avoiding causal inconsistency requires access to stable views of the file

system at specific points in time. File-system snapshots, such as those provided by ZFS or Linux's LVM [7], are currently the only means of obtaining such views. However, snapshot support is relatively uncommon, and when implemented, tends not to be designed for the fine granularity at which synchronization services capture changes.

5.4 Summary

As our observations have shown, the sense of safety provided by synchronization services is largely illusory. The limited interface between clients and the file system, as well as the failure of many file systems to implement key features, can lead to corruption and flawed crash recovery polluting all available copies, and causal inconsistency may cause bizarre or unexpected behavior. Thus, naively assuming that these services will provide complete data protection can lead instead to data loss, especially on some of the most commonly-used file systems.

Even for file systems capable of detecting errors and preventing their propagation, such as ZFS and btrfs, the separation of synchronization services and the file system incurs an opportunity cost. Despite the presence of correct copies of data in the cloud, the file system has no means to employ them to facilitate recovery. Tighter integration between the service and the file system can remedy this, allowing the file system to automatically repair damaged files. However, this makes avoiding causal inconsistency even more important, as naive techniques, such as simply restoring the most recent version of each damaged file, are likely to directly cause it.

Chapter 6

ViewBox: Cooperative Data Protection across Local and Cloud Storage

Both cloud-based file synchronization services and file systems go to extensive efforts to preserve user data. However, our analysis in Chapter 5 reveals that both systems fail to protect user data in several scenarios. Because the client has no means of determining whether file changes are intentional or the result of corruption, it may send both to the cloud, ultimately spreading corrupt data to all of a user's devices. Crashes compound this problem; the client may upload inconsistent data to the cloud, download potentially inconsistent files from the cloud, or fail to synchronize changed files. Finally, even in the absence of failure, the client cannot normally preserve causal dependencies between files, since it lacks stable point-in-time images of files as it uploads them. This can lead to an inconsistent cloud image, which may in turn lead to unexpected application behavior.

In this chapter, we present ViewBox, a system in which local file system and cloud-based synchronization services are integrated and work cooperatively to solve the problems above. Instead of synchronizing individual files, ViewBox synchronizes views, in-memory snapshots of the local synchronized folder that provide data integrity, crash consistency, and causal consistency. The local file system exposes views to the synchronization client such that the client only uploads updates from the views. Since the client only updates views in their entirety, ViewBox guarantees the correctness and consistency of the cloud image, which it then uses to correctly recover from local failures. Furthermore, by making the server aware of views, ViewBox can synchronize views across clients and properly handle conflicts

without losing data.

The rest of the chapter is organized as follows. We first present the high-level design of ViewBox in Section 6.1. We then describe the implementation of ViewBox in detail in 6.2. Finally, we evaluate our prototype ViewBox system in Section 6.3.

6.1 Design

To remedy the problems outlined in the previous section, we propose ViewBox, an integrated solution in which the local file system and the synchronization service cooperate to detect and recover from these issues. Instead of a clean-slate design, we structure ViewBox around ext4 (ordered journaling mode), Dropbox, and Seafile, in the hope of solving these problems with as few changes to existing systems as possible.

Ext4 provides a stable, open-source, and widely-used solution on which to base our framework. While both btrfs and ZFS already provide some of the functionality we desire, they lack the broad deployment of ext4. Additionally, as it is a journaling file system, ext4 also bears some resemblance to NTFS and HFS+, the Windows and Mac OS X file systems; thus, many of our solutions may be applicable in these domains as well.

Similarly, we employ Dropbox because of its reputation as one of the most popular, as well as one of the most robust and reliable, synchronization services. Unlike ext4, it is entirely closed source, making it impossible to modify directly. Despite this limitation, we are still able to make significant improvements to the consistency and integrity guarantees that both Dropbox and ext4 provide. However, certain functionalities are unattainable without modifying the synchronization service. Therefore, we take advantage of an open source synchronization service, Seafile, to show the capabilities that a fully integrated file system and synchronization service can provide. Although we only implement ViewBox with Dropbox and Seafile, we believe that the techniques we introduce apply generally to other synchronization services.

In this section, we first outline the fundamental goals driving ViewBox. We then provide a high-level overview of the architecture with which we hope to achieve these goals. Our architecture performs three primary functions: detection, synchronization, and recovery; we discuss each of these in turn.

6.1.1 Goals

In designing ViewBox, we focus on four primary goals, based on both resolving the problems we have identified and on maintaining the features that make users appreciate file-synchronization services in the first place.

Integrity: Most importantly, ViewBox must be able to detect local corruption and prevent its propagation to the rest of the system. Users frequently depend on the synchronization service to back up and preserve their data; thus, the file system should never pass faulty data along to the cloud.

Consistency: When there is a single client, ViewBox should maintain causal consistency between the client's local file system and the cloud and prevent the synchronization service from uploading inconsistent data. Furthermore, if the synchronization service provides the necessary functionality, ViewBox must provide multi-client consistency: file-system states on multiple clients should be synchronized properly with well-defined conflict resolution.

Recoverability: While the previous properties focus on containing faults, containment is most useful if the user can subsequently repair the faults. ViewBox should be able to use the previous versions of the files on the cloud to recover automatically. At the same time, it should maintain causal consistency when necessary, ideally restoring the file system to an image that previously existed.

Performance: Improvements in data protection cannot come at the expense of performance. ViewBox must perform competitively with current solutions even when running on the low-end systems employed by many of the users of file synchronization services. Thus, naive solutions, like synchronous replication [65], are not acceptable.

6.1.2 Fault Detection

The ability to detect faults is essential to prevent them from propagating and, ultimately, to recover from them as well. In particular, we focus on detecting corruption and data inconsistency. While ext4 provides some ability to detect corruption through its metadata checksums, these do not protect the data itself. Thus, to correctly detect all corruption, we add checksums to ext4's data as well, storing them separately so that we may detect misplaced writes [29, 69], as well as bit flips. Once it detects corruption, ViewBox then prevents the file from being uploaded until it can employ its recovery mechanisms.

In addition to allowing detection of corruption resulting from bit-flips or bad disk behavior, checksums also allow the file system to detect the inconsistent crash

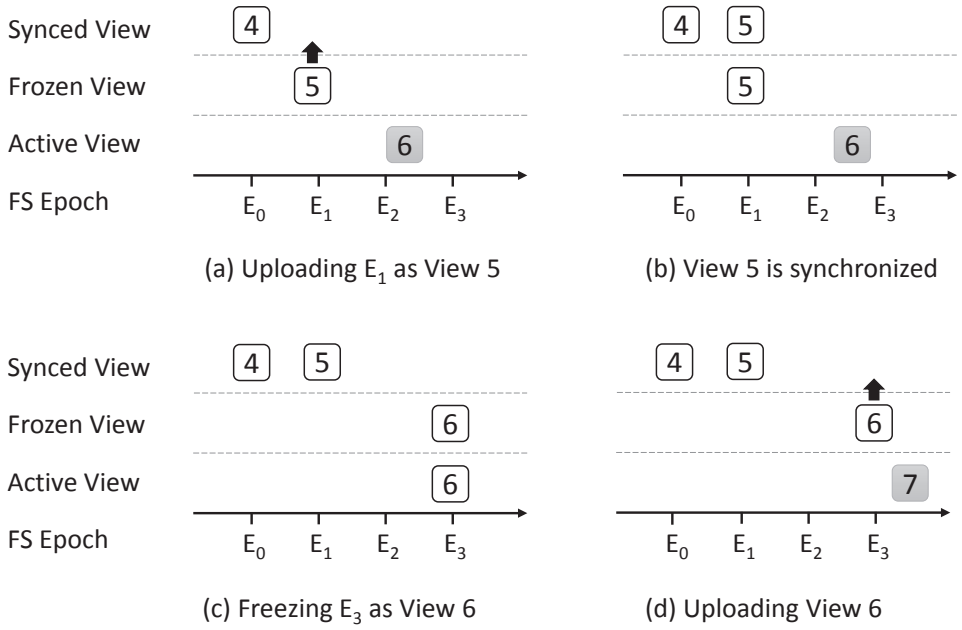


Figure 6.1: **Synchronizing Frozen Views** *This figure shows how view-based synchronization works, focusing on how to upload frozen views to the cloud. The x-axis represents a series of file system epochs. Squares represent various views in the system, with a view number as ID. When an active view is shaded, it means that the view is not at an epoch boundary and cannot be frozen.*

recovery that could result from ext4's journal. Because checksums are updated independently of their corresponding blocks, an inconsistently recovered data block will not match its checksum. As inconsistent recovery is semantically identical to data corruption for our purposes—both comprise unintended changes to the file system—checksums prevent the spread of inconsistent data, as well. However, they only partially address our goal of correctly restoring data, which requires stronger functionality.

6.1.3 View-based Synchronization

Ensuring that recovery proceeds correctly requires us to eliminate causal inconsistency from the synchronization service. Doing so is not a simple task, however. It requires the client to have an isolated view of all data that has changed since the last synchronization; otherwise, user activity could cause the remote image to span several file system images but reflect none of them.

While file-system snapshots provide consistent, static images [62], they are too heavyweight for our purposes. Because the synchronization service stores all file data remotely, there is no reason to persist a snapshot on disk. Instead, we propose a system of in-memory, ephemeral snapshots, or *views*.

View Basics

Views represent the state of the file system at specific points in time, or epochs, associated with quiescent points in the file system. We distinguish between three types of views: active views, frozen views, and synchronized views. The active view represents the current state of the local file system as the user modifies it. Periodically, the file system takes a snapshot of the active view; this becomes the current frozen view. Once a frozen view is uploaded to the cloud, it then becomes a synchronized view, and can be used for restoration. At any time, there is only one active view and one frozen view in the local system, while there are multiple synchronized views on the cloud.

To provide an example of how views work in practice, Figure 6.1 depicts the state of a typical ViewBox system. In the initial state, (a), the system has one synchronized view in the cloud, representing the file system state at epoch 0, and is in the process of uploading the current frozen view, which contains the state at epoch 1. While this occurs, the user can make changes to the active view, which is currently in the middle of epoch 2 and epoch 3.

Once ViewBox has completely uploaded the frozen view to the cloud, it becomes a synchronized view, as shown in (b). ViewBox refrains from creating a new frozen view until the active view arrives at an epoch boundary, such as a journal commit, as shown in (c). At this point, it discards the previous frozen view and creates a new one from the active view, at epoch 3. Finally, as seen in (d), ViewBox begins uploading the new frozen view, beginning the cycle anew.

Because frozen views are created at file-system epochs and the state of frozen views is always static, synchronizing frozen views to the cloud provides both crash consistency and causal consistency, given that there is only one client actively synchronizing with the cloud. We call this *single-client consistency*.

Multi-client Consistency

When multiple clients are synchronized with the cloud, the server must propagate the latest synchronized view from one client to other clients, to make all clients' state synchronized. Critically, the server must propagate views in their entirety; partially uploaded views are inherently inconsistent and thus should not be visible.

However, because synchronized views necessarily lag behind the active views in each file system, the current active file system may have dependencies that would be invalidated by a remote synchronized view. Thus, remote changes must be applied to the active view in a way that preserves local causal consistency.

To achieve this, ViewBox handles remote changes in two phases. In the first phase, ViewBox applies remote changes to the frozen view. If a changed file does not exist in the frozen view, ViewBox adds it directly; otherwise, it adds the file under a new name that indicates a conflict (e.g., "foo.txt" becomes "remote.foo.txt"). In the second phase, ViewBox merges the newly created frozen view with the active view. ViewBox propagates all changes from the new frozen view to the active view, using the same conflict handling procedure. At the same time, it uploads the newly merged frozen view. Once the second phase completes, the active view is fully updated; only after this occurs can it be frozen and uploaded.

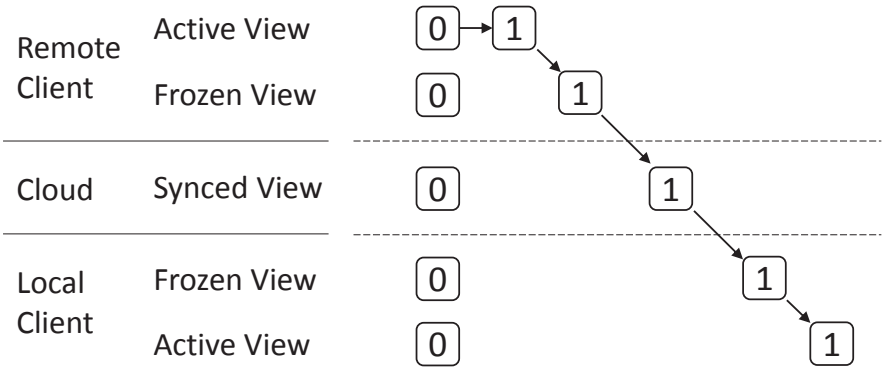
To correctly handle conflicts and ensure no data is lost, we follow the same policy as GIT [54]. This can be summarized by the following three guidelines:

- Preserve any local or remote change; a change could be the addition, modification, or deletion of a file.
- When there is a conflict between a local change and a remote change, always keep the local copy untouched, but rename and save the remote copy.
- Synchronize and propagate both the local copy and the renamed remote copy.

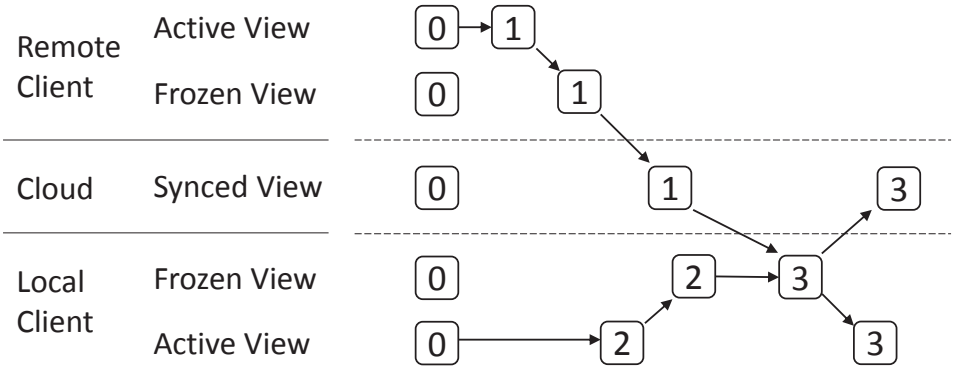
Figure 6.2 illustrates how ViewBox handles remote changes. In case (a), both the remote and local clients are synchronized with the cloud, at view 0. The remote client makes changes to the active view, and subsequently freezes and uploads it to the cloud as view 1. The local client is then informed of view 1, and downloads it. Since there are no local updates, the client directly applies the changes in view 1 to its frozen view and propagates those changes to the active view.

In case (b), both the local client and the remote client perform updates concurrently, so conflicts may exist. Assuming the remote client synchronizes view 1 to the cloud first, the local client will refrain from uploading its frozen view, view 2, and download view 1 first. It then merges the two views, resolving conflicts as described above, to create a new frozen view, view 3. Finally, the local client uploads view 3 while simultaneously propagating the changes in view 3 to the active view.

In the presence of simultaneous updates, as seen in case (b), this synchronization procedure results in a cloud state that reflects a combination of the disk states of all clients, rather than the state of any one client. Eventually, the different client and cloud states will converge, providing *multi-client consistency*. This model is



(a) Directly Applying Remote Updates



(b) Merging and Handling Potential Conflicts

Figure 6.2: **Handling Remote Updates** *This figure demonstrates two different scenarios where remote updates are handled. While case (a) has no conflicts, case (b) may, because it contains concurrent updates.*

weaker than our single-client model; thus, ViewBox may not be able to provide causal consistency for each individual client under all circumstances.

Unlike single-client consistency, multi-client consistency requires the cloud server to be aware of views, not just the client. Thus, ViewBox can only provide multi-client consistency for open source services, like Seafile; providing it for closed-source services, like Dropbox, will require explicit cooperation from the service provider.

6.1.4 Cloud-aided Recovery

With the ability to detect faults and to upload consistent views of the file system state, ViewBox is now capable of performing correct recovery. There are effectively two types of recovery to handle: recovery of corrupt files, and recovery of inconsistent files at the time of a crash.

In the event of corruption, if the file is clean in both the active view and the frozen view, we can simply recover the corrupt block by fetching the copy from the cloud. If the file is dirty, the file may not have been synchronized to the cloud, making direct recovery impossible, as the block fetched from cloud will not match the checksum. If recovering a single block is not possible, the entire file must be rolled back to a previous synchronized version, which may lead to causal inconsistency.

Recovering causally-consistent images of files that were present in the active view at the time of a crash faces the same difficulties as restoring corrupt files in the active view. Restoring each individual file to its most recent synchronized version is not correct, as other files may have been written after the now-corrupted file and, thus, depend on it; to ensure these dependencies are not broken, these files also need to be reverted. Thus, naive restoration can lead to causal inconsistency, even with views.

Instead, we present users with the choice of individually rolling back damaged files, potentially risking causal inconsistency, or reverting to the most recent synchronized view, ensuring correctness but risking data loss. As we anticipate that the detrimental effects of causal inconsistency will be relatively rare, the former option will be usable in many cases to recover, with the latter available in the event of bizarre or unexpected application behavior.

6.2 Implementation

Now that we have provided a broad overview of ViewBox's architecture, we delve more deeply into the specifics of our implementation. As with Section 6.1, we divide our discussion based on the three primary components of our architecture: detection, as implemented with our new *ext4-cksum* file system; view-based synchronization using our *view manager*, a file-system agnostic extension to ext4-cksum; and recovery, using a user-space recovery daemon called *cloud helper*.

6.2.1 Ext4-cksum

Like most file systems that update data in place, ext4 provides minimal facilities for detecting corruption and ensuring data consistency. While it offers experimental

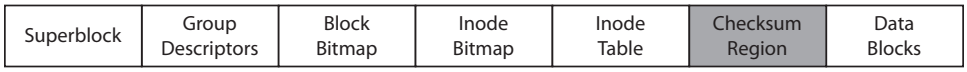


Figure 6.3: **Ext4-cksum Disk Layout** *This graph shows the typical layout of a block group in ext4-cksum. The shaded region, the checksum table, contains data checksums for blocks in the block group.*

metadata checksums, these do not protect data; similarly, its default ordered journaling mode only protects the consistency of metadata, while providing minimal guarantees about data. Thus, it requires changes to meet our requirements for integrity and consistency. We now present ext4-cksum, a variant of ext4 that supports data checksums to protect against data corruption and to detect data inconsistency after a crash without the high cost of data journaling.

Checksum Region

There are several ways in which we could add data checksums to ext4. The simplest way is to store a checksum within its protecting block, which is viable if the disk supports 520-byte sectors [112]. If not, some bytes in the 4KB block will have to be sacrificed to store the checksum, which may cause alignment problems with applications. In addition, because this method stores the data block and the checksum in the same logical write unit, it cannot detect misdirected writes or phantom writes [69]. Alternatively, the file system could inline the checksum for each block with the pointer to it in metadata, as ZFS does. While this method can work well, it can substantially limit the maximum file size, due to the need to store checksums, and it may work awkwardly with ext4's current implementation of extents.

Ext4-cksum stores data checksums in a fixed-sized *checksum region* immediately after the inode table in each block group, as shown in Figure 6.3. All checksums of data blocks in a block group are preallocated in the checksum region. This region acts similarly to a bitmap, except that it stores checksums instead of bits, with each checksum mapping directly to a data block in the group. Since the region starts at a fixed location in a block group, the location of the corresponding checksum can be easily calculated, given the physical (disk) block number of a data block.

The size of the region depends solely on the total number of blocks in a block group and the length of a checksum, both of which are determined and fixed during file system creation. Currently, ext4-cksum uses the built-in crc32c checksum, which is 32-bit long. Therefore, it reserves a 32-bit checksum for every 4KB block, imposing a space overhead of 1/1024; for a regular 128MB block group, the size of the checksum region is 128KB.

Checksum Handling for Reads and Writes

When a data block is read from disk, the corresponding checksum must be verified. Before the file system issues a read of a data block from disk, it gets the corresponding checksum by reading the checksum block. After the file system reads the data block into memory, it verifies the block against the checksum. If the initial verification fails, ext4-cksum will retry. If the retry also fails, ext4-cksum will report an error to the application. Note that in this case, if ext4-cksum is running with the cloud helper daemon, ext4-cksum will try to get the remote copy from cloud and use that for recovery. The read part of a read-modify-write is handled in the same way.

A read of a data block from disk always incurs an additional read for the checksum, but not every checksum read will cause high latency. First, the checksum read can be served from the page cache, because the checksum blocks are considered metadata blocks by ext4-cksum and are kept in the page cache like other metadata structures. Second, even if the checksum read does incur a disk I/O, because the checksum is always in the same block group as the data block, the seek latency will be minimal. Third, to avoid checksum reads as much as possible, ext4-cksum employs a simple prefetching policy: always read 8 checksum blocks (within a block group) at a time. Advanced prefetching heuristics, such as those used for data prefetching, are applicable here.

Ext4-cksum does not update the checksum for a dirty data block until the data block is written back to disk. Before issuing the disk write for the data block, ext4-cksum reads in the checksum block and updates the corresponding checksum. This applies to all data write-backs, caused by a background flush, fsync, or a journal commit.

Since ext4-cksum treats checksum blocks as metadata blocks, with journaling enabled, ext4-cksum logs all dirty checksum blocks in the journal. In ordered journaling mode, this also allows the checksum to detect inconsistent data caused by a crash. In ordered mode, dirty data blocks are flushed to disk before metadata blocks are logged in the journal. If a crash occurs before the transaction commits, data blocks that have been flushed to disk may become inconsistent, because the metadata that points to them still remains unchanged after recovery. As the checksum blocks are metadata, they will not have been updated, causing a mismatch with the inconsistent data block. Therefore, if such a block is later read from disk, ext4-cksum will detect the checksum mismatch.

To ensure consistency between a dirty data block and its checksum, data write-backs triggered by a background flush and fsync can no longer simultaneously occur with a journal commit. In ext4 with ordered journaling, before a transaction

has committed, data write-backs may start and overwrite a data block that was just written by the committing transaction. This behavior, if allowed in ext4-cksum, would cause a mismatch between the already logged checksum block and the newly written data block on disk, thus making the committing transaction inconsistent. To avoid this scenario, ext4-cksum ensures that data write-backs due to a background flush and fsync always occur before or after a journal commit.

6.2.2 View Manager

To provide consistency, ViewBox requires file synchronization services to upload frozen views of the local file system, which it implements through an in-memory file-system extension, the view manager. In this section, we detail the implementation of the view manager, beginning with an overview. Next, we introduce two techniques, cloud journaling and incremental snapshotting, which are key to the consistency and performance provided by the view manager. Then, we describe the synchronization processes that upload a frozen view to cloud. Finally, we briefly discuss how to integrate the synchronization client with the view manager to handle remote changes and conflicts.

View Manager Overview

The view manager is a light-weight kernel module that creates views on top of a local file system. Since it only needs to maintain two local views at any time (one frozen view and one active view), the view manager does not modify the disk layout or data structures of the underlying file system. Instead, it relies on a modified tmpfs to present the frozen view in memory and support all the basic file system operations to files and directories in it. Therefore, a synchronization client now monitors the exposed frozen view (rather than the actual folder in the local file system) and uploads changes from the frozen view to the cloud. All regular file system operations from other applications are still directly handled by ext4-cksum. The view manager uses the active view to track the on-going changes and then reflects them to the frozen view. Note that the current implementation of the view manager is tailored to our ext4-cksum and it is not stackable [128]. We believe that a stackable implementation would make our view manager compatible with more file systems.

Consistency through Cloud Journaling

As we discussed in Section 6.1.3, to preserve consistency, frozen views must be created at file-system epochs. Therefore, the view manager freezes the current

active view at the beginning of a journal commit in ext4-cksum, which serves as a boundary between two file-system epochs. At the beginning of a commit, the current running transaction becomes the committing transaction. When a new running transaction is created, all operations belonging to the old running transaction have completed, and operations belonging to the new running transaction have not started yet. The view manager freezes the active view at this point, ensuring that no in-flight operation spans multiple views. All changes since the last frozen view are preserved in the new frozen view, which is then uploaded to the cloud, becoming the latest synchronized view.

To ext4-cksum, the cloud acts as an external journaling device. Every synchronized view on the cloud matches a consistent state of the local file system at a specific point in time. Although ext4-cksum still runs in ordered journaling mode, when a crash occurs, the file system now has the chance to roll back to a consistent state stored on cloud. We call this approach cloud journaling.

Low-overhead via Incremental Snapshotting

During cloud journaling, the view manager achieves better performance and lower overhead through a technique called incremental snapshotting. The view manager always keeps the frozen view in memory and the frozen view only contains the data that changed from the previous view. The active view is thus responsible for tracking all the files and directories that have changed since it last was frozen. When the view manager creates a new frozen view, it marks all changed files copy-on-write (COW), which preserves the data at that point. The new frozen view is then constructed by applying the changes associated with the active view to the previous frozen view.

The view manager uses several in-memory and on-cloud structures to support this incremental snapshotting approach. First, the view manager maintains an *inode mapping table* to connect files and directories in the frozen view to their corresponding ones in the active view. The view manager represents the namespace of a frozen view by creating *frozen inodes* for files and directories in tmpfs (their counterparts in the active view are thus called *active inodes*), but no data is usually stored under frozen inodes (unless the data is copied over from the active view due to copy-on-write). When a file in the frozen view is read, the view manager finds the active inode and fetches data blocks from it. The inode mapping table thus serves as a translator between a frozen inode and its active inode.

Second, the view manager tracks namespace changes in the active view by using an *operation log*, which records all successful namespace operations (e.g., create, mkdir, unlink, rmdir, and rename) in the active view. The log records the

type of an operation and all operands, in the form of active inode numbers. For example, for a file create, the inode numbers of the parent dir and the created file inode are logged. When the active view is frozen, the log is replayed onto the previous frozen view to bring it up-to-date, reflecting the new state.

Third, the view manager uses a *dirty table* to track what files and directories are modified in the active view. Once the active view becomes frozen, all these files are marked copy-on-write. Then, by generating inotify events based on the operation log and the dirty table, the view manager is able to make the synchronization client check and upload these local changes to the cloud. After the synchronization is finished, the view becomes a synchronized view on the cloud.

Finally, the view manager keeps *view metadata* on the server for every synchronized view, which is used to identify what files and directories are contained in a synchronized view. For services such as Seafile, which internally keeps the modification history of a folder as a series of snapshots [99], the view manager is able to use its snapshot ID (called commit ID by Seafile) as the view metadata. For services like Dropbox, which only provides file-level versioning, the view manager creates a view metadata file for every synchronized view, consisting of a list of pathnames and revision numbers of files in that view. The information is obtained by querying the Dropbox server. The view manager stores these metadata files in a hidden folder on the cloud, so the correctness of these files is not affected by disk corruption or crashes.

Synchronizing Views to the Cloud

Now, we describe how the view manager synchronizes views to the server.

Initial Synchronization: Assuming there are no crash and no remote changes, when a local file system is mounted and the synchronization client starts, the client scans the synchronization folder and uploads any offline changes to the server. With ViewBox, to ensure that the synchronization client captures a view of the initial on-disk state of the synchronized folder, the view manager freezes the initial state of ext4-cksum before the client starts. The client then scans the frozen view and synchronizes any offline changes to the cloud, in the same way as the unmodified synchronization client. We call this process *initial synchronization*. Note that during the initial synchronization, ext4-cksum is not accessible to applications other than the client, as if it were not mounted.

The view manager creates the initial frozen view by cloning the whole namespace from ext4-cksum (the active view). It creates the same directories in the frozen view directly, and clones files from the active view by allocating sparse files in their

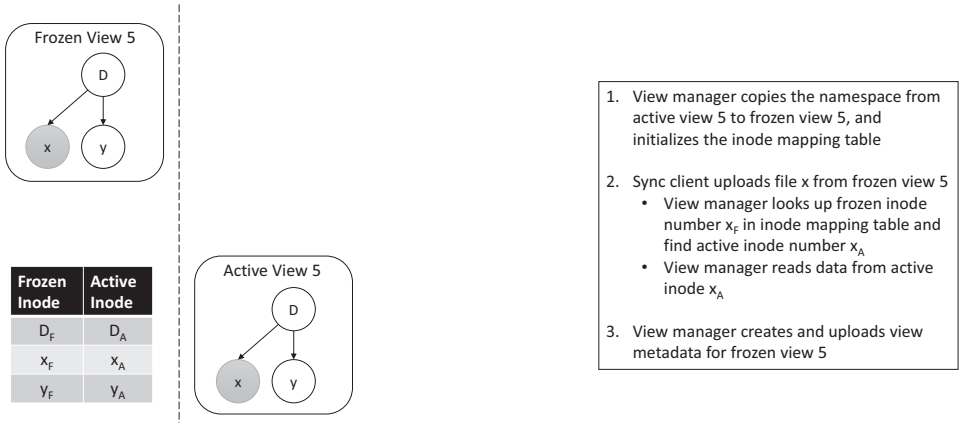


Figure 6.4: **Initial Synchronization** *This figure shows how the view manager initializes and uploads a frozen view upon file system mount. In the frozen view and active view, D is a directory containing two files x and y. File x is shaded because it was modified while the synchronization client is offline. The table represents the inode mapping table, in which N_A is the active inode number of N and N_F is the frozen inode number of N.*

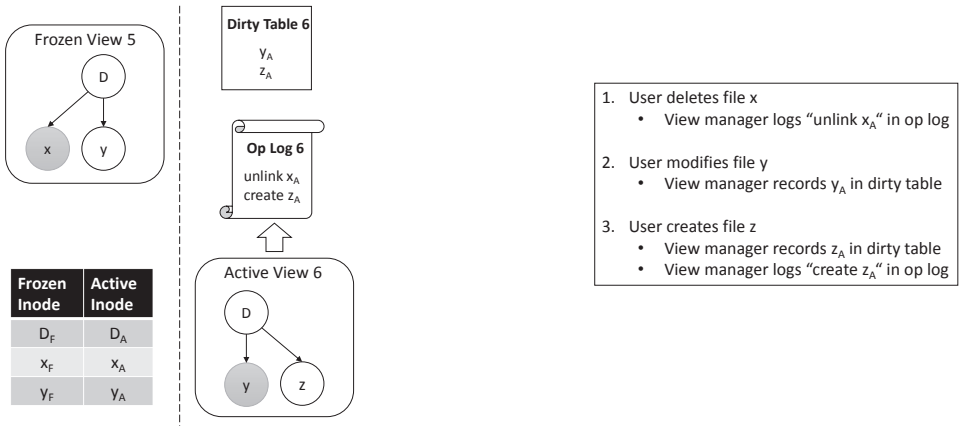


Figure 6.5: **Tracking Changes in an Active View** *This figure illustrates how the view manager tracks changes in active view 6 using the dirty table and op log. File y is shaded in active view 6 because it was modified.*

corresponding directories in the frozen view. These frozen files have the same inode attributes (such as mtime and size) as their active versions, but do not contain any data. The inode mapping table is initialized during this process.

Then, the synchronization client starts to scan the frozen view, in order to detect offline changes. The client reads all new and modified files from the frozen view and uploads them to the server. Because the frozen view does not contain any data for the file, the view manager handles data reads by looking up the inode mapping table, finding the active inode, and reading blocks from the active view. After the client finishes uploading the view, the view manager creates and stores view metadata of the view on the server.

Figure 6.4 shows an example of how the view manager performs initial synchronization. We will use the same example to illustrate how the view manager works in the following discussion.

Regular Synchronization: Once the initial synchronization finishes, the active view becomes visible to applications and starts to carry out operations. The view manager uses an operation log and a dirty table to record namespace changes and file changes in the active view, as shown in Figure 6.5. At some point, the active view is frozen and a new active view is immediately created. While the frozen view is being synchronized to the cloud, the new active view continues to serve requests from applications. We call this process *regular synchronization*. Once the frozen view is synchronized, the view manager starts the same process again.

Freezing an active view: The view manager freezes the current active view at the beginning of the upcoming transaction commit in ext4-cksum. When the active view is frozen, the op-log and dirty table are attached to the frozen view and become *frozen op-log* and *frozen dirty table*. At the same time, a new active view is created on top of the ext4-cksum, with an empty *active op-log* and *active dirty table*. Figure 6.6 shows how the view manager freezes the previous active view 6 and creates a new active view 7.

Establishing a frozen view: In ViewBox, a frozen view does not have to be persistent. Instead, it only needs to be present when it is being synchronized to the cloud. Therefore, the view manager takes a light-weight in-memory snapshot approach. The key is to break the state of the snapshot into three parts: namespace, inode attributes and file data.

The view manager relies on the op-log to quickly bring the namespace up-to-date. When the active view becomes frozen, the namespace in the frozen view is

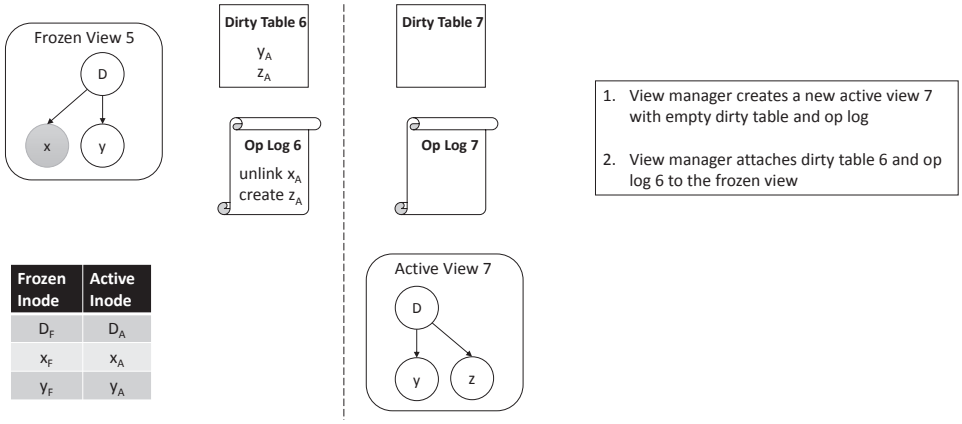


Figure 6.6: **Freezing an Active View** *This figure shows how the view manager freezes active view 6 and creates active view 7.*

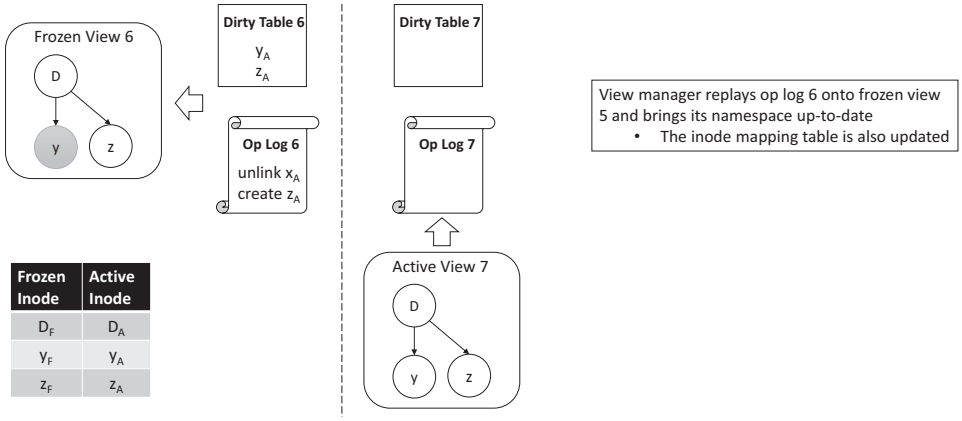


Figure 6.7: **Establishing a Frozen View** *This figure shows how the view manager updates the namespace of the frozen view to reflect the state of active view 6.*

stale; it still reflects the state of previously synchronized frozen view, so does the inode mapping table. Since the frozen op-log recorded all namespace operations that took place between when the synchronized view was frozen and when the current frozen view was frozen, the view manager keeps the namespace up-to-date

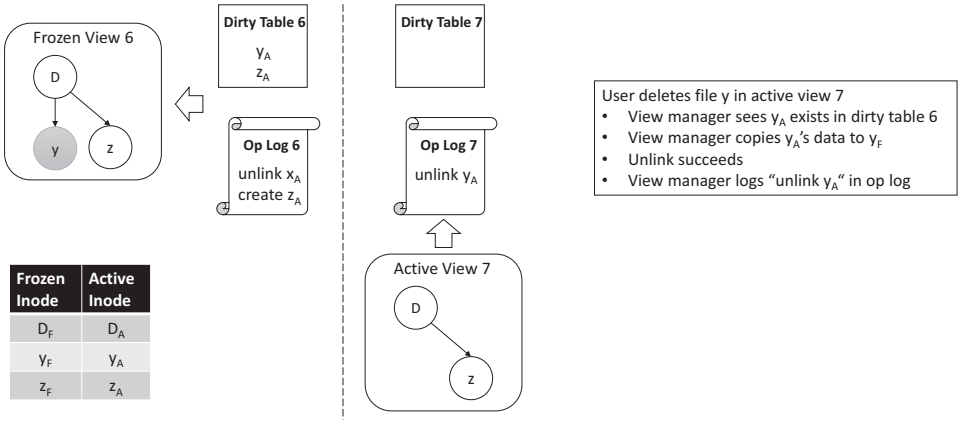


Figure 6.8: **Example of Copy-on-write** *This figure shows an example of copy-on-write that is trigger by deleting a frozen file in the active view.*

by replaying these logged operations, as shown in Figure 6.7. The inode mapping table is also updated during the replay. By storing concrete directory structures in the frozen view, any namespace operation that takes place in the current active view will not affect the namespace of the frozen view. Therefore, there is no need to perform copy-on-write (COW) on metadata related to namespace, which avoids the complication and overhead of copying these metadata structures.

However, preserving inode attributes and file data still need COW. After the replay is finished, all files in the frozen view have their frozen inodes allocated. However, these inodes are only placeholders; they still have the previous attributes and there are no data blocks allocated. For inode attributes, when a frozen inode is to be accessed or when the corresponding active inode is to be changed, the inode attributes of the active inode would be copied to the frozen inode. Therefore, operations such as utime, chown, and chmod in the active view will trigger COW for inode attributes. For file data, the frozen view does not keep a copy of a data block, unless the data block is to be modified or deleted in the active view, so operations including write, truncate, unlink, and rename (in which a file is overwritten by the renamed file) will cause COW for affected data blocks. Figure 6.8 illustrates a COW example in which the file y is removed in active view 7. If a data block remains unchanged in the active view, when the block is read from the frozen view, the view manager will directly fetch that data block from the active inode in ext4-cksum, either from disk or from the page cache. This saves an unnecessary page

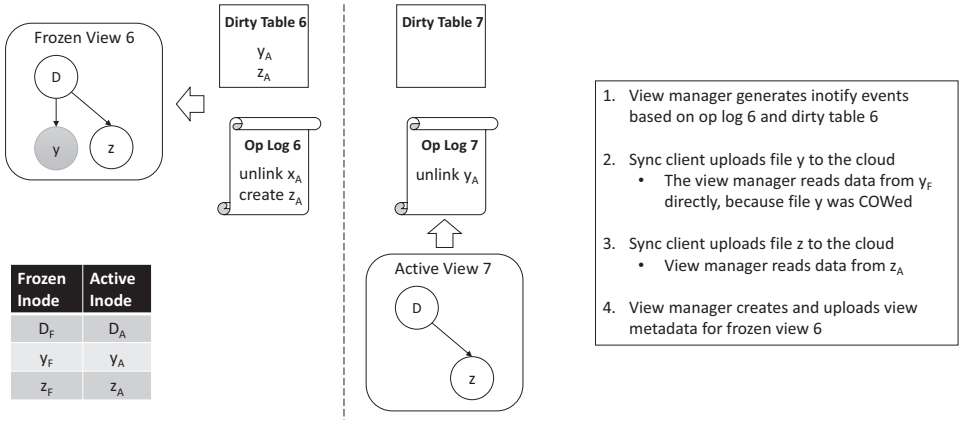


Figure 6.9: **Uploading a Frozen View** *This figure illustrates how the view manager uploads frozen view 6 to the cloud.*

copying from the active inode to the frozen inode.

Moreover, not every file in the frozen view needs to be COWed. Since synchronization services only upload changed files to the server, the view manager only has to COW changed files. As discussed before, the dirty table recorded files that were changed when the frozen view was active. Once the view is frozen, all recorded inodes are marked COW. Therefore, before any file-changing operation takes place in the active view, the view manager checks if the file's inode exists in the frozen dirty table. If it exists, the inode attributes and any data block that will be affected by the operation but have not been COWed will be copied to the frozen view. Otherwise, the operation will be carried out without any COW overhead.

Uploading a frozen view: In a regular file system without views, the synchronization client relies on the inotify mechanism to monitor file and directory changes in real-time, and uploads those changes accordingly. In ViewBox, however, the client monitors the frozen view exposed by the view manager, in which most changes (other than the replayed namespace operations) do not take place in real-time. Therefore, the view manager recreates and replays inotify events to drive the synchronization client upload changes in the frozen view. After the client finishes uploading the frozen view, the view manager creates necessary view metadata. Finally, the view manager destroys the frozen op-log and the frozen dirty table, cleans up COWed data pages in the frozen view, and prepares to freeze the current active

view. Figure 6.9 shows the steps the view manager takes to upload frozen view 6.

Handling Remote Changes

All the techniques we have introduced so far focus on how to provide single-client consistency and do not require modifications to the synchronization client or the server. They work well with proprietary synchronization services such as Dropbox. However, when there are multiple clients running ViewBox and performing updates at the same time, the synchronization service itself must be view-aware. To handle remote updates correctly, we modify the Seafile client to perform the two-phase synchronization described in Section 6.1.3. We choose Seafile to implement multi-client consistency, because both its client and server are open-source. More importantly, its data model and synchronization algorithm are similar to GIT, which fits our view-based synchronization well.

6.2.3 Cloud Helper

When corruption or a crash occurs, ViewBox performs recovery using backup data on the cloud. Recovery is performed through a user-level daemon, cloud helper. The daemon is implemented in Python, which acts as a bridge between the local file system and the cloud. It interacts with the local file system using ioctl calls and communicates with the cloud through the service's web API.

For data corruption, when ext4-cksum detects a checksum mismatch, it sends a block recovery request to the cloud helper. The request includes the pathname of the corrupted file, the offset of the block inside the file, and the block size. The cloud helper then fetches the requested block from the server and returns the block to ext4-cksum. Ext4-cksum re-verifies the integrity of the block against the data checksum in the file system and returns the block to the application. If the verification still fails, it is possibly because the block has not been synchronized or because the block is fetched from a different file in the synchronized view on the server with the same pathname as the corrupted file.

When a crash occurs, the cloud helper performs a scan of the ext4-cksum file system to find potentially inconsistent files. If the user chooses to only roll back those inconsistent files, the cloud helper will download them from the latest synchronized view. If the user chooses to roll back the whole file system, the cloud helper will identify the latest synchronized view on the server, and download files and construct directories in the view. The former approach is able to keep most of the latest data but may cause causal inconsistency. The latter guarantees causal

Service ViewBox w/	Data write	Metadata mtime	ctime	atime
Dropbox	DR	DR	DR	DR
Seafile	DR	DR	DR	DR

Table 6.1: **Data Corruption Results of ViewBox** *In all cases, the local corruption is detected (D) and recovered (R).*

Service ViewBox w/	Upload local ver.	Download cloud ver.	Out-of-sync (no sync)
Dropbox	$\times$	$\surd$	$\times$
Seafile	$\times$	$\surd$	$\times$

Table 6.2: **Crash Consistency Results of ViewBox** *The local version of the file is inconsistent, and is rolled back to the previous version on the cloud.*

consistency, but at the cost of losing updates that took place during the frozen view and the active view when the crash occurred.

6.3 Evaluation

We now present the evaluation results of our ViewBox prototype. We first show that our system is able to recover from data corruption and crashes correctly and provide causal consistency. Then, we evaluate the underlying ext4-cksum and view manager components separately, without synchronization services. Finally we study the overall synchronization performance of ViewBox with Dropbox and Seafile.

We implemented ViewBox in the Linux 3.6.11 kernel, with Dropbox client 1.6.0, and Seafile client and server 1.8.0. All experiments are performed on machines with a 3.3GHz Intel Quad Core CPU, 16GB memory, and a 1TB Hitachi Deskstar hard drive. For all experiments, we reserve 512MB of memory for the view manager. We run every experiment 10 times and report the average result.

6.3.1 Cloud Helper

We first perform the same set of fault injection experiments as in Section 2. The corruption and crash test results are shown in Table 6.1 and Table 6.2. Because the

Workload	ext4 (MB/s)	ext4-cksum (MB/s)	Slowdown
Seq. write	103.69	99.07	4.46%
Seq. read	112.91	108.58	3.83%
Rand. write	0.70	0.69	1.42%
Rand. read	5.82	5.74	1.37%

Table 6.3: **Microbenchmarks on ext4-cksum** *This figure compares the throughput of several micro benchmarks on ext4 and ext4-cksum. Sequential write/read are writing/reading a 1GB file in 4KB requests. Random write/read are writing/reading 128MB of a 1GB file in 4KB requests. For sequential read workload, ext4-cksum prefetches 8 checksum blocks for every disk read of a checksum block.*

local state is initially synchronized with the cloud, the cloud helper is able to fetch the redundant copy from cloud and recover from corruption and crashes. We also confirm that ViewBox is able to preserve causal consistency.

6.3.2 Ext4-cksum

We now evaluate the performance of standalone ext4-cksum, focusing on the overhead caused by data checksumming. Table 6.3 shows the throughput of several microbenchmarks on ext4 and ext4-cksum. From the table, one can see that the performance overhead is quite minimal. Note that checksum prefeteching is important for sequential reads; if it is disabled, the slowdown of the workload increases to 15%.

We perform a series of macrobenchmarks using Filebench on both ext4 and ext4-cksum with checksum prefetching enabled. The results are shown in Table 6.4. For the fileserver workload, the overhead of ext4-cksum is quite high, because there are 50 threads reading and writing concurrently and the negative effect of the extra seek for checksum blocks accumulates. The webserver workload, on the other hand, experiences little overhead, because it is dominated by warm reads.

It is surprising to notice that ext4-cksum greatly outperforms ext4 in varmail. This is actually a side effect of the ordering of data write-backs and journal commit, as discussed in Section 6.2.1. Note that because ext4 and ext4-cksum are not mounted with "journal_async_commit", the commit record is written to disk with a cache flush and the FUA (force unit access) flag, which ensures that when the commit record reaches disk, all previous dirty data (including metadata logged in the

Workload	ext4 (MB/s)	ext4-cksum (MB/s)	Slowdown
Fileserver	79.58	66.28	16.71%
Varmail	2.90	3.96	-36.55%
Webserver	150.28	150.12	0.11%

Table 6.4: **Macrobenchmarks on ext4-cksum** *This table shows the throughtput of fileserver, varmail, and webserver workloads on ext4 and ext4-cksum. Fileserver is configured with 50 threads performing creates, deletes, appends, whole-file writes, and whole-file reads. Varmail emulates a multi-threaded mail server. Each thread performs a set of create-append-sync, read-append-sync, read, and delete operations. It has about half reads and half writes and is dominated by random I/Os. Webserver is a multi-threaded read-intensive workload.*

journal) has already been forced to disk. When running varmail in ext4, data blocks written by fsyncs from other threads during the journal commit are also flushed to disk at the same time, which causes high latency. In contrast, since ext4-cksum does not allow data write-back from fsync to run simultaneously with the journal commit, the amount of data flushed is much smaller, which improves the overall throughput of the workload.

6.3.3 View Manager

We now study the performance of various file system operations in an active view when a frozen view exists. The view manager runs on top of ext4-cksum.

We first evaluate the performance of various operations that do not cause copy-on-write (COW) in an active view. These operations are create, unlink, mkdir, rmdir, rename, utime, chmod, chown, truncate and stat. We run a workload that involves creating 1000 8KB files across 100 directories and exercising these operations on those files and directories. We prevent the active view from being frozen so that all these operations do not incur a COW. We see a small overhead (mostly less than 5% except utime, which is around 10%) across all operations, as compared to their performance in the original ext4., This overhead is mainly caused by operation logging and other bookkeeping performed by the view manager.

Next, we show the normalized response time of operations that do trigger copy-on-write in Table 6.5. These operations are performed on a 10MB file after the file is created and marked COW in the frozen view. All operations cause all 10MB of file data to be copied from the active view to the frozen view. The copying overhead

Operation	Normalized Response Time	
	Before COW	After COW
unlink (cold)	484.49	1.07
unlink (warm)	6.43	0.97
truncate (cold)	561.18	1.02
truncate (warm)	5.98	0.93
rename (cold)	469.02	1.10
rename (warm)	6.84	1.02
overwrite (cold)	1.56	1.10
overwrite (warm)	1.07	0.97

Table 6.5: **Copy-on-write Operations in the View Manager** *This table shows the normalized response time (against ext4) of various operations on a frozen file (10MB) that trigger copy-on-write of data blocks. "Before COW"/"After COW" indicates the operation is performed before/after affected data blocks are COWed.*

is listed under the "Before COW" column, which indicates that these operations occur before the affected data blocks are COWed. When the cache is warm, which is the common case, the data copying does not involve any disk I/O but still incurs up to 7x overhead. To evaluate the worst case performance (when the cache is cold), we deliberately force the system to drop all caches before we perform these operations. As one can see from the table, all data blocks are read from disk, thus causing much higher overhead. Note that cold cache cases are rare and may only occur during memory pressure. We further measure the performance of the same set of operations on a file that has already been fully COWed. As shown under the "After COW" column, the overhead is negligible, because no data copying is performed.

6.3.4 ViewBox with Dropbox and Seafile

We assess the overall performance of ViewBox using three workloads: openssh (building openssh from its source code), iphoto_edit (editing photos in iPhoto, about 5GB data), and iphoto_view (browsing photos in iPhoto, about 1GB data). The latter two workloads are from the iBench trace suite [60] and are replayed using Magritte [119]. We believe that these workloads are representative of ones people run with synchronization services.

The results of running all three workloads on ViewBox with Dropbox and Seafile are shown in Table 6.6 and Table 6.7. In all cases, the runtime of the work-

	ext4 + Dropbox		ViewBox with Dropbox	
Workload	Runtime	Sync Time	Runtime	Sync Time
openssh	36.4	49.0	36.0	64.0
iphoto_edit	577.4	2115.4	563.0	2667.3
iphoto_view	149.2	170.8	153.4	591.0

Table 6.6: **Performance of ViewBox with Dropbox** *This table compares the runtime and sync time (in seconds) of various workloads running on top of the unmodified ext4 and ViewBox using Dropbox. Runtime is the time it takes to finish the workload and sync time is the time it takes to finish synchronizing.*

	ext4 + Seafile		ViewBox with Seafile	
Workload	Runtime	Sync Time	Runtime	Sync Time
openssh	36.0	44.8	36.0	56.8
iphoto_edit	566.6	857.6	554.0	598.8
iphoto_view	150.0	166.6	156.4	175.4

Table 6.7: **Performance of ViewBox with Seafile** *This table compares the runtime and sync time (in seconds) of various workloads running on top of the unmodified ext4 and ViewBox using Seafile. Runtime is the time it takes to finish the workload and sync time is the time it takes to finish synchronizing.*

load in ViewBox is at most 5% slower and sometimes even faster than that of the unmodified ext4 setup, which shows that view-based synchronization does not have a negative impact on the foreground workload. We also find that the memory overhead of ViewBox (the amount of memory consumed by the view manager to store frozen views) is minimal, at most 20MB across all three workloads.

We expect the synchronization time of ViewBox to be longer because ViewBox does not start synchronizing until the current file system state is frozen, which may cause delays. The results of openssh confirm our expectations. However, for iphoto_view and iphoto_edit, the synchronization time on ViewBox with Dropbox is much greater than that on ext4. This is due to Dropbox's lack of proper interface support for views, as described in Section 6.2.2. Because both workloads use a file system image with around 1200 directories, to create the view metadata for each view, ViewBox has to query the Dropbox server numerous times, causing substantial overhead. In contrast, ViewBox can avoid this overhead with Seafile because it has direct access to Seafile's internal metadata. Thus, the synchronization time of

iphoto_view in ViewBox with Seafile is near that in ext4.

Note that the iphoto_edit workload actually has a much shorter synchronization time on ViewBox with Seafile than on ext4. Because the photo editing workload involves many writes, Seafile delays uploading when it detects files being constantly modified. After the workload finishes, many files have yet to be uploaded. Since frozen views prevent interference, ViewBox can finish synchronizing about 30% faster.

6.4 Summary

Despite their near-ubiquity, file synchronization services ultimately fail at one of their primary goals: protecting user data. Not only do they fail to prevent corruption and inconsistency, they actively spread it in certain cases. The fault lies equally with local file systems, however, as they often fail to provide the necessary capabilities that would allow synchronization services to catch these errors. To remedy this, we propose ViewBox, an integrated system that allows the local file system and the synchronization client to work together to prevent and repair errors.

Rather than synchronizing individual files, as current file synchronization services do, ViewBox centers around views, in-memory file-system snapshots which have their integrity guaranteed through on-disk checksums. Since views provide consistent images of the file system, they provide a stable platform for recovery that minimizes the risk of restoring a causally inconsistent state. As they remain in-memory, they incur minimal overhead.

We implement ViewBox to support both Dropbox and Seafile clients, and find that it prevents the failures that we observe with unmodified local file systems and synchronization services. Equally importantly, it performs competitively with unmodified systems. This suggests that the cost of correctness needs not be high; it merely requires adequate interfaces and cooperation.

Chapter 7

Related Work

This chapter discusses various research efforts and real systems that are related to this dissertation. We first discuss literature on analyzing system reliability using fault injection and modeling techniques. Then, we summarize research on improving data integrity and consistency in storage systems.

7.1 Fault Injection

Software-implemented fault injection techniques have been widely used to analyze the robustness of systems [26, 33, 56, 66, 101, 114]. For example, FINE used fault injection to emulate hardware and software faults in the operating system [66]; Gu et al. [56] injected faults to instruction streams of Linux kernel function to characterize Linux kernel behavior.

More recent works have applied type-aware fault injection to analyze failure behaviors of different file systems to disk corruptions. Prabhakaran et al. injected partial disk failures to various file systems to understand the behavior of these systems in the presence of disk errors and randomly-corrupted disk blocks [89]. Bairavasundaram et al. developed and applied type-aware pointer corruption to NTFS and ext3 to study how both systems handle pointer corruption in their metadata structures [24]. Our analysis of on-disk data integrity in ZFS and data corruption with synchronization services is similar to these studies.

Furthermore, fault injection has also been used to analyze effects of memory corruption on systems. FIAT [26] used fault injection to study the effects of memory corruption in a distributed environment. Krishnan et al. applied a memory corruption framework to analyze the effects of metadata corruption on NFS [70]. Our study on in-memory data integrity is related to these studies in their goal of

finding effects of memory corruption.

However, our work on ZFS is the first comprehensive reliability analysis of local file system that covers carefully controlled experiments to analyze both on-disk and in-memory data integrity. Specifically, for our study of memory corruptions, we separately analyze ZFS behavior for faults in page cache metadata and data and for metadata structures in the heap. To the best of our knowledge, this is the first such comprehensive study of end-to-end file system data integrity.

Similarly, our analysis of cloud-based synchronization services is the first study on the reliability of these services. We study the impact of disk corruption and system crash to synchronization services and reveal the surprising fact that multiple copies do not always make data safe.

7.2 Reliability Modeling

A large body of research has been focusing on modeling device-level errors such as memory errors and latent sector errors. Li et al. performed a series of measurement of soft errors on real production systems, and developed models for error rates and error patterns [71, 72]. Schroeder et al. conducted a detailed static analysis of latent sector errors and provided parameters for models derived from the analysis [96]. Based on the models, they proposed and evaluated several new protection schemes against latent sector errors.

There are many studies on reliability modeling for RAID systems [31, 45, 86], but only a few of them cover silent data corruption. Rozier et al. presented a fault model for Undetected Disk Errors (UDE) in RAID systems [92]. They built a framework that combines simulation and model to calculate the manifestation rates of undetected data corruption caused by UDEs. Krioukov et al. used model checking to analyze various protection techniques used in current RAID storage systems [69]. They study the interaction between these techniques and find design faults that may lead to data loss or data corruption. In comparison, our reliability framework focuses on bit errors from various devices (not just disk or RAID). We use analytical models to evaluate the reliability of different devices and different checksums in terms of the probability of undetected corruption. Our framework calculates a system-level metric that can be used to compare the reliability of different storage systems.

7.3 Techniques for Data Integrity

Using checksums to detect data corruption is common. File systems, such as PFS [104], GoogleFS [51], IRON file system [89], btrfs [91] and ZFS [29], use checksums to protect on-disk blocks. Many database systems, such as Berkeley DB [85] and SQL Server [1], support page-level checks to make sure data is not corrupted on disk. In networking, the Internet checksum [12], used by most Internet protocols, is designed to detect transmission errors. The integrity check specified in RPCSEC_GSS [13] protects RPC messages during transmission. All these checks are applied in a single subsystem/protocol, while flexible end-to-end data integrity focuses on cross-component data protection. In addition, our ext4-cksum is similar to these systems in using checksums, but to our knowledge, it is the first work to add data checksumming to ext4. In Z^2FS, we take advantage of existing checks as well as our newly added checks to provide wider coverage of data protection.

Many of the systems above, such as GoogleFS, IRON file system, and ZFS, rely on locally stored redundant copies for automatic recovery, which may or may not be available. In contrast, ViewBox is the first work of which we are aware that employs the cloud for recovery.

The concept of flexible end-to-end data integrity is similar to the protection scheme in the Linux Data Integrity Extension (DIX) [87] and the T10 Protection Information (T10-PI) model [112] (previously known as Data Integrity Field). DIX provides end-to-end protection from the application to the I/O controller, while T10-PI covers the data path between the I/O controller and the disk. Within this framework, checksums are passed from the application all the way to the disk, and can be verified by the disk drive, as well as the components inbetween. Although T10-PI requires CRC as the checksum, DIX is able to use the Internet checksum [12] to achieve better performance and relies on the I/O controller to convert the Internet checksum to CRC. The behavior of each components in the I/O path is well modeled by the data integrity architecture from SNIA [102]. Our flexible end-to-end concept differs from their scheme in that they focus on *defining* the behavior of each node while our work helps to *reason* about the rational behind certain behaviors, such as what checksum should be used by which component, and when and where the system should change checksum. Our reliability framework also provides a holistic way to think about the tradeoffs between performance and protection.

In terms of implementation, Z^2FS offers similar protection as DIX, but it is different from DIX in several aspects. First, Z^2FS is a software solution while T10-PI and DIX require support from hardware vendors. The hard drives and the controller must support 520-byte sector because the checksum is stored in the extra

8-byte area for each sector. Z^{2}FS uses space maintained by the file system to store checksums so that it is able to provide similar protection as DIX without special hardware. It can also be easily extended to support T10-PI. Second, in addition to checksum chaining (conversion) at the disk-memory boundary Z^{2}FS performs checksum switching for data in memory. We believe Z^{2}FS is the first file system to take data residency time into consideration and provide better protection for data in the page cache. Third, Z^{2}FS is a full-featured local file system that exposes checksum to applications through new and generic APIs so that any application can be modified to take advantage of the data protection offered by Z^{2}FS. In comparison, DIX is currently a block layer extension in Linux. To our best knowledge, there is no local file system support or user-level APIs available; DIX is now only used in Lustre file system [82] and Oracle's database products [46, 121].

7.4 Techniques for Data Consistency

A variety of research work, such as IRON file system [89] and OptFS [36], explores the use of checksums for purposes beyond simply detecting corruption. IRON ext3 introduces transactional checksums, which allow the journal to issue all writes, including the commit block, concurrently; the checksum detects any failures that may occur. OptFS extends transactional checksum to cover dirty data blocks that are flushed during journal commit, so that the system is able to detect inconsistent data upon a crash. Ext4-cksum is mostly related to OptFS in that ext4-cksum also relies on checksums to detect inconsistent data, but OptFS requires data block to be checksummed whenever the block is updated in the page cache, which may lead to high response time for write system calls (due to checksum calculation). In contrast, ext4-cksum only generates checksums when data blocks are written back, which usually occurs in the background and does not incur much overhead.

Similarly, a number of works have explored means of providing greater crash consistency than ordered and metadata journaling provide. Data journaling mode in ext3 and ext4 provides full crash consistency, but its high overhead makes it unappealing. OptFS [36] is able to achieve data consistency and deliver high performance through an optimistic protocol, but it does so at the cost of durability while still relying on data journaling to handle overwrite cases. In contrast, ViewBox avoids overhead by allowing the local file system to work in ordered mode, while providing consistency through the views it synchronizes to the cloud; it then can restore the latest view after a crash to provide full consistency. Like OptFS, this sacrifices durability, since the most recent view on the cloud will always lag behind the active file system. However, this approach is optional, and, in the normal case,

ordered mode recovery can still be used.

ViewBox's snapshotting component, the view manager, bears some resemblance to ext3cow [88] and Next3 [49], but these similarities are mostly superficial. Like both of these systems, the view manager performs copy-on-write once per snapshot. However, unlike these systems, the view manager does not persist its snapshots on disk, relying instead on the cloud back-end to store uploaded views. Additionally, while we implement the view manager as an extension to ext4, it requires no modification to on-disk data structures and could easily be applied to any other Linux file system. Finally, while ext3cow's focus on file history resembles Dropbox's file revision history interface, ViewBox shifts from this interface to focus on complete images, as this is the only way to guarantee causal consistency when restoring previous file versions.

Chapter 8

Conclusion and Future Work

One of the major responsibilities of storage systems is to store data correctly and protect it from being damaged. Existing systems and many research projects have employed various techniques to fulfill this responsibility, but most of the techniques only focus on protecting data in a specific component in the storage stack, while failing to provide comprehensive protection – corrupt data or inconsistent data still goes undetected and is exposed to users or applications.

In this dissertation, we identified this problem of isolated protection in both local and cloud storage systems, and proposed several cooperative data protection techniques to address the problem. For local storage systems, we first analyzed the impact of disk and memory corruption to ZFS and found that ZFS fails to protect in-memory data (Chapter 3). Then, we proposed the concept of flexible end-to-end data integrity and built Z^2FS by applying the concept to ZFS, which provides end-to-end protection with improved performance (Chapter 4). For cloud storage services, we started by studying how synchronization clients propagate corrupt data and inconsistent data to the cloud due to the loose coupling of local file systems and synchronization services (Chapter 5). We then built ViewBox, a system in which local file systems and synchronization services work cooperatively to provide data integrity, consistency, and recoverability (Chapter 6).

In this chapter, we first summarize our analysis and solutions in Section 8.1, then list a set of lessons learned over the course of this work in Section 8.2, and finally discuss directions for future research in Section 8.3.

8.1 Summary

This dissertation is mainly divided in two parts: cooperative data protection in local storage, and cooperative data protection across local and cloud storage. Each part further consists of a problem analysis and a solution. We now summarize each part in turn.

8.1.1 Cooperative Data Protection in Local Storage

In the first part of the dissertation, we focused on the impact of disk corruption and memory corruption in local storage systems and we chose ZFS, a modern and mature file system, as our study subject. First, we evaluated how robust ZFS is against disk and memory corruption. We injected corruption to data blocks and metadata structures both on disk and in memory. We found that ZFS is able to detect and recover from most injected disk corruption, due to the usage of checksums for on-disk blocks and file-system level replication for important metadata structures. However, because the protection is only limited to disk blocks, ZFS fails to protect in-memory data and metadata, which leads to bad data blocks being silently returned to the user or written to disk, file system operation failures, and whole system crashes. Our findings indicated that end-to-end data protection is needed to protect data from both memory and disk corruption.

Then, we explored techniques to provide end-to-end data protection. A straightforward way to achieve this is to apply the traditional end-to-end concept, in which applications generate and verify checksum (usually a strong one) for their data. However, this approach suffers from slow performance for workloads that repeatedly access data from the page cache due to the overhead of calculating checksums. Moreover, when the corruption occurs in the write path, it fails to detect the corruption in time, and thus it is not able to recover from it.

To address both problems, we proposed a new concept called flexible end-to-end data integrity, which enables all components in the storage system to be aware of checksums and changes checksums scheme across components (sometimes even over time) to achieve a balance between performance and reliability. We developed an analytical model to reason about which checksums to be used on which component, and then built Z^2FS to demonstrate how to apply flexible end-to-end data integrity to an existing file system, ZFS. As a comparison, we also built E^2ZFS with straight-forward end-to-end data integrity. Through analysis and fault injection experiments, we showed that Z^2FS is able to provide Zettabyte reliability (at most one undetected corruption per Zettabyte data read), and can detect and recover from corruption in the write path. Through performance experiments, we

showed that Z^2FS performs comparably to the original ZFS in various micro and macro benchmarks and outperforms E^2ZFS by up to 17% in workloads dominated by warm reads.

8.1.2 Cooperative Data Protection across Local and Cloud Storage

The second part of the dissertation focused on the impact of disk corruption and untimely crashes in local file systems and cloud storage services (cloud-based file synchronization services). We first performed fault injection experiments on several popular synchronization services and studied how well they protect data. Through disk corruption experiments, we found that in many cases, all the services we examined propagate local corruption to the cloud and thus corrupt copies on other devices. Through crash tests, we found that the synchronization clients behave inconsistently; sometimes they upload inconsistent files to the cloud, sometimes they download stale versions of files from the server, and sometimes they refuse to synchronize despite the fact that the local copy is different than the cloud copy. Further, we showed that these services cannot provide causal consistency because the clients are not able to obtain an unified and consistent view of the local file system. Our analysis revealed that the root cause of these problems is the loose coupling of synchronization services and local file systems.

Next, we designed, implemented, and evaluated a new system called ViewBox, in which the synchronization service works cooperatively with the local file system to provide data integrity and consistency. The key idea behind ViewBox is views, in-memory snapshots of the synchronizing folder. Instead of uploading files, ViewBox synchronizes views between the devices and cloud. To guarantee the correctness of views, ViewBox relies on three components: ext4-cksum, the view manager, and the cloud helper. Ext4-cksum adds data checksuming to ext4 and serves as the local file system in ViewBox. The added checksum is able to detect both corruption and inconsistency. The view manager is an extension to ext4-cksum which creates views at file system epochs and exposes views to the synchronization client; the consistency of views is thus guaranteed. The cloud helper is a user-level daemon that uses views on the cloud to perform recovery when corruption or inconsistency is detected. We built ViewBox around two synchronization services, Dropbox and Seafile. Through fault injection experiments, we showed that ViewBox is able to detect and recover from corruption and crash, and therefore prevent bad data from being propagated. Compared to Dropbox and Seafile running on top of unmodified ext4, we showed that ViewBox incurs less than 5% overhead in many workloads, and in some cases reduces the synchronization time by 30%.

8.2 Lessons Learned

In this section, we present a list of general lessons we learned while working on this dissertation.

- **Reliability does not come for free.** First, data protection techniques usually hurt the performance of the system. In E^{2}ZFS, we moved the checksum generation and verification up to the application level to achieve end-to-end data integrity, which caused about a 15% slowdown compared to the original ZFS in some workloads. In ViewBox, we added checksums to ext4 and we find that the overall throughout of ext4-cksum is worse than the original ext4. The former case is due to the CPU overhead of calculating the checksum, and the latter is because of extra I/Os and seeks to read and write checksum blocks.

  Second, optimization helps to reduce the overhead. In Z^{2}FS, we chose xor as memory checksum and we applied the checksum-on-copy optimization [39] to make it extremely faster (3% overhead compared to the original ZFS) than the naive implementation (7% overhead). In ViewBox, we implemented prefetching of checksum blocks for sequential read workloads such that the throughput slowdown (compared to original ext4) is improved from 15% to 4%.

  Finally, fast storage device needs fast checksum calculation. Current systems perform well with strong checksums because the checksum calculation usually occurs with a (traditional) disk I/O, which is already costly. As fast devices (such as SSDs) are becoming popular and widely deployed, storage systems cannot hide the computational cost of checksum behind I/O time anymore, so we believe that either we have to find a checksum that is strong enough to protect data and fast enough to not cause noticeable slowdown, or we should take other approaches to reduce the calculation overhead (e.g., through specialized instructions or additional chips).

- **One size (checksum) does not fit all.** With the straight-forward end-to-end protection scheme, usually one checksum is used all the way from application to disk. This simplifies the implementation of a system, but strips the flexibility away; reliability can be achieved by using a stronger checksum, but the performance hurts. Our flexible end-to-end data protection proposes to use different checksum for different components, depending on the their reliability and/or performance characteristics, such that the reliability and

performance of the whole system can be tweaked to satisfy certain requirement. We believe that such flexibility should be provided by future storage systems, especially software-defined storage systems.

- **Multiple copies do not always make data safe.** File synchronization services automatically upload local data to the cloud, and propagate it to other synchronized devices. These services give the users a perception that there are multiple copies of their data and their data must be safe. However, our analysis showed that this is merely a false sense of "security". When the local file system or the synchronization client cannot distinguish legitimate changes (actual updates) from "unauthorized" changes (corrupt or inconsistent data), bad data may be uploaded to the server and thus pollute all copies – failing to guarantee the correctness of data renders all the replicas useless. We believe that the replication itself does not necessarily improve data reliability; the ability to verify the integrity of data is the foundation replication should rely on.

8.3 Future Work

In this section, we outline various directions for future work.

8.3.1 Characteristic Study of Data Corruption

Our analytical framework described in Section 4.1 models data corruption as independent bit flips in a fixed-sized data block, which simplifies the model but unfortunately fails to represent the reality. Bairavasundaram et al. found that corruption (checksum mismatches) that occur in the same disk is not independent and has spatial and temporal locality [23]. Schroeder et al. found that memory errors also have strong time and space correlations [63]. Therefore, in order to better understand and model data corruption, we believe that a study of data corruption characteristic will be an interesting future direction.

The focus of the study would be to find out the pattern of corruption and how likely each pattern occurs. If the corruption is caused by dropped writes, the corrupt data would be the same as the previous data at the same location but may look completely different than the correct data. If the corruption is caused by bit-rots, it is highly likely that the corruption is just several bit flips. In this case, it would also be interesting to know the distribution of the number of bit flips.

In addition to helping to improve the modeling of data corruption, the study would be beneficial in several other ways. First, categorizing data corruption events

may provide some hints on why data corruption occurs and which component should be blamed for it. For example, if most corruption events are random bit flips, it is possible that the disk drive is defected and should be replaced. Second, understanding corruption pattern would help with the invention of special checksums. As mentioned above, fast devices need fast checksums to avoid the performance slowdown. If the corruption pattern of such a device is known, one may be able to apply a checksum that is specially designed to handle that corruption pattern and performs much faster than a generic and strong checksum.

8.3.2 Application-level Data Protection

This dissertation has focused on data protection provided by file systems and file synchronization services, but it does not address another important piece: applications. Since applications are the ones that generate data and process data, it is critical to make sure applications protect data and handle corruption correctly. It is well known that corporate applications, such as database systems and mail servers, already use checksums to protect data from corruption [1, 4, 85, 105]. Therefore, studying the robustness of home-user applications, such as document editors or photo managers, will be an interesting future avenue and the first step would be a thorough analysis of how data corruption affects application behavior.

We can inject various faults (such as corruption, read error and write error) when an application reads/writes data from/to the file system, and see how the application reacts. We may classify application behaviors into three categories: detection, recovery and functionality. In terms of detection, applications may ignore the failure or corruption, detect and inform the user, or detect and hide from the user. In terms of recovery, applications may perform no recovery, retry, repair, or wait for user instruction. By functionality, we mean after the error handling (detection and recovery) whether the applications work as usual, abort abnormally, or perform incorrect actions.

There are two challenges in this fault injection analysis. First, to effectively inject faults, we have to understand various file formats. Different applications work with different file formats. Each file format is like a file system and has its own organization of metadata and data. For example, a MP3 file contains a stream of MP3 frames, each of which consists of a header (metadata) and an audio data block. In contrast, a DOC file is actually a mini FAT file system, which contains text files, images files and other metadata structures that make up the document. Therefore, it is important to study how metadata and data in each file format is organized and what are the meanings of the metadata structures. Second, automation of the fault injection experiments may be difficult. Unlike traditional UNIX programs, most

home-user applications are all GUI-based and they interact with users extensively. The involvement of human users may hinder the efficiency of fault injection experiments. To solve this problem, we can use advanced scripting languages, such as AppleScript, to control GUIs.

Once we have the results from the fault injection analysis, we will be able to explore techniques to improve the robustness of applications in the face of data corruption.

8.3.3 Cooperative Data Protection in Networked Storage Systems

We have explored techniques for cooperative data protection in local storage systems and cloud storage systems. We believe that another important environment to look into is networked storage systems.

Network File System (NFS) is a popular network file system protocol, originally developed by Sun, which allows users to access files across a network. NFS relies on a security protocol called RPCSEC_GSS [13] to provide data integrity, in which RPC messages containing NFS requests and responses are checksummed (*NFS checksum*). However, the protocol also suffers from the problem of isolated protection; the checksum is only used during network transmission and there is no end-to-end protection between the client-side application and the server disk.

One approach to achieve cooperative data protection is to apply the concept of flexible end-to-end data integrity to NFS. First, Z^2FS can be directly used here as the local file system on the client and the server. Second, models for network data corruption, TCP/IP checksum, and NFS checksums used by RPCSEC_GSS (e.g., DES) are needed to evaluate how reliable the whole system is with the addition of the network part, and to choose a proper NFS checksum to meet the performance and reliability requirement. Finally, checksum chaining must be applied at the boundary of page cache and the NFS layer to connect the client's or the server's memory checksum and the NFS checksum.

8.4 Closing Words

In this dissertation, we have identified the problem of isolated protection in existing storage systems, and proposed various techniques to achieve cooperative data protection. As the amount of generated data explodes, the use of low-cost hardware increases, and the complexity of storage systems grows, existing and future storage systems will face more and more challenges to data protection. By demonstrating the power of cooperation, we hope that this dissertation can help researchers, de-

signers, and developers to rethink data protection and build reliable storage systems with cooperative data protection.

Bibliography

[1] Buffer Management - SQL Server 2008 R2. http://msdn.microsoft.com/en-us/library/aa337525.aspx.

[2] CERT/CC Advisories. http://www.cert.org/advisories/.

[3] Data Integrity. http://indico.cern.ch/getFile.py/access?contribId=3&sessionId=0&resId=1&materialId=paper&confId=13797.

[4] Eseutil /K Checksum Mode. http://technet.microsoft.com/en-us/library/bb123632%28EXCHG.65%29.aspx.

[5] Kernel Bug Tracker. http://bugzilla.kernel.org/.

[6] LASR Traces. http://iotta.snia.org/traces/2.

[7] lvcreate(8) - linux man page.

[8] Mozy. https://www.mozy.com.

[9] Repeated panics, something gone bad? http://tech.groups.yahoo.com/group/solarisx86/message/38925.

[10] RFC 3385 - Internet Protocol Small Computer System Interface (iSCSI) Cyclic Redundancy Check (CRC)/Checksum Considerations. http://www.ietf.org/rfc/rfc3385.txt.

[11] RFC 793 - Transmission Control Protocol. http://www.ietf.org/rfc/rfc793.txt.

[12] RFC1071 - Computing the Internet Checksum. http://www.ietf.org/rfc/rfc1071.txt.

[13] RFC2203 - RPCSEC_GSS Protocol Specification. http://www.ietf.org/rfc/rfc2203.txt.

[14] US-CERT Vulnerabilities Notes Database. http://www.kb.cert.org/vuls/.

[15] ZFS on Linux. http://zfsonlinux.org.

[16] Zfs problem mirror. http://www.mail-archive.com/zfs-discuss@opensolaris.org/msg18079.html.

[17] Zfs problems. http://www.mail-archive.com/zfs-discuss@opensolaris.org/msg04518.html.

[18] Amazon. Amazon Simple Storage Service (Amazon S3). http://aws.amazon.com/s3/.

[19] Dave Anderson, Jim Dykes, and Erik Riedel. More Than an Interface: SCSI vs. ATA. In *Proceedings of the 2nd USENIX Symposium on File and Storage Technologies (FAST '03)*, San Francisco, California, April 2003.

[20] Apple. icloud. `http://www.icloud.com/`.

[21] Apple. Technical Note TN1150. `http://developer.apple.com/technotes/tn/tn1150.html`, March 2004.

[22] Lakshmi N. Bairavasundaram, Garth R. Goodson, Shankar Pasupathy, and Jiri Schindler. An Analysis of Latent Sector Errors in Disk Drives. In *Proceedings of the 2007 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '07)*, San Diego, California, June 2007.

[23] Lakshmi N. Bairavasundaram, Garth R. Goodson, Bianca Schroeder, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. An Analysis of Data Corruption in the Storage Stack. In *Proceedings of the 6th USENIX Symposium on File and Storage Technologies (FAST '08)*, pages 223–238, San Jose, California, February 2008.

[24] Lakshmi N. Bairavasundaram, Meenali Rungta, Nitin Agrawal, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Michael M. Swift. Analyzing the Effects of Disk Pointer Corruption. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '08)*, Anchorage, Alaska, June 2008.

[25] Wendy Bartlett and Lisa Spainhower. Commercial Fault Tolerance: A Tale of Two Systems. *IEEE Transactions on Dependable and Secure Computing*, 1(1):87–96, January 2004.

[26] J.H. Barton, E.W. Czeck, Z.Z. Segall, and D.P. Siewiorek. Fault Injection Experiments Using FIAT. *IEEE Transactions on Computers*, 39(4):1105–1118, April 1990.

[27] Robert Baumann. Soft errors in advanced computer systems. *IEEE Design & Test of Computers*, 22(3):258–266, 2005.

[28] Emery D. Berger and Benjamin G. Zorn. Diehard: probabilistic memory safety for unsafe languages. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI '06)*, Ottawa, Canada, June 2006.

[29] Jeff Bonwick and Bill Moore. ZFS: The Last Word in File Systems. `http://opensolaris.org/os/community/zfs/docs/zfs_last.pdf`, 2007.

[30] Florian Buchholz. The structure of the Reiser file system. `http://homes.cerias.purdue.edu/~florian/reiser/reiserfs.php`.

[31] W. Burkhard and Jai Menon. Disk Array Storage System Reliability. In *Proceedings of the 23rd International Symposium on Fault-Tolerant Computing (FTCS-23)*, pages 432–441, Toulouse, France, June 1993.

[32] Remy Card, Theodore Ts'o, and Stephen Tweedie. Design and Implementation of the Second Extended Filesystem. In *First Dutch International Symposium on Linux*, Amsterdam, Netherlands, December 1994.

[33] 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*, 1998.

[34] John Chapin, Mendel Rosenblum, Scott Devine, Tirthankar Lahiri, Dan Teodosiu, and Anoop Gupta. Hive: Fault Containment for Shared-Memory Multiprocessors. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, Copper Mountain Resort, Colorado, December 1995.

[35] C. L. Chen. Error-correcting codes for semiconductor memories. *SIGARCH Comput. Archit. News*, 12(3):245–247, 1984.

[36] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Optimistic Crash Consistency. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, Farmington, PA, November 2013.

[37] Vijay Chidambaram, Tushar Sharma, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Consistency Without Ordering. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST '12)*, San Jose, California, February 2012.

[38] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An Empirical Study of Operating System Errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 73–88, Banff, Canada, October 2001.

[39] Hsiao-keng Jerry Chu. Zero-copy tcp in solaris. In *Proceedings of the 1996 USENIX Annual Technical Conference*, San Diego, CA, 1996.

[40] Jonathan Corbet. Improving ext4: bigalloc, inline data, and metadata checksums. `http://lwn.net/Articles/469805/`, November 2011.

[41] csync. csync. `http://www.csync.org/`.

[42] Nurit Dor, Michael Rodeh, and Mooly Sagiv. CSSV: towards a realistic tool for statically detecting all buffer overflows in C. In *Proceedings of the 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '03)*, San Diego, California, June 2003.

[43] Idilio Drago, Marco Mellia, Maurizio M. Munafò, Anna Sperotto, Ramin Sadre, and Aiko Pras. Inside Dropbox: Understanding Personal Cloud Storage Services. In *Proceedings of the 2012 ACM conference on Internet measurement conference (IMC '12)*, Boston, MA, November 2012.

[44] Dropbox. The dropbox tour. `https://www.dropbox.com/tour`.

[45] Jon G. Elerath and Michael Pecht. Enhanced reliability modeling of raid storage systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '07)*, Edinburgh, UK, June 2007.

[46] EMC. An Integrated End-to-End Data Integrity Solution to Protect Against Silent Data Corruption. `http://www.oracle.com/us/technologies/linux/data-integrity-solution-1852762.pdf`.

[47] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 57–72, Banff, Canada, October 2001.

[48] A. Eto, M. Hidaka, Y. Okuyama, K. Kimura, and M. Hosono. Impact of neutron flux on soft errors in mos memories. In *International Electron Devices Meeting 1998 (IEDM '98)*, 1998.

[49] Amir G. Next3 snapshots design. Technical report, CTERA Networks, Ltd., July 2011.

[50] Gregory R. Ganger and Yale N. Patt. Metadata Update Performance in File Systems. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation (OSDI '94)*, pages 49–60, Monterey, California, November 1994.

[51] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 29–43, Bolton Landing, New York, October 2003.

[52] GIT. Git. `http://git-scm.com`.

[53] Google. Google drive. `http://www.google.com/drive/about.html`.

[54] David Greaves, Junio Hamano, et al. git-read-tree(1): - linux man page. `http://linux.die.net/man/1/git-read-tree`.

[55] Roedy Green. EIDE Controller Flaws Version 24. `http://mindprod.com/jgloss/eideflaw.html`.

[56] Weining Gu, Z. Kalbarczyk, Ravishankar K. Iyer, and Zhenyu Yang. Characterization of Linux Kernel Behavior Under Errors. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '03)*, pages 459–468, San Francisco, California, June 2003.

[57] Haryadi S. Gunawi, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. SQCK: A Declarative File System Checker. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI '08)*, San Diego, California, December 2008.

[58] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson Engler. A system and lang-uage for building system-specific, static analyses. In *Proceedings of the 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '03)*, San Diego, California, June 2003.

[59] James Hamilton. Successfully Challenging the Server Tax. `http://perspectives.mvdirona.com/2009/09/03/SuccessfullyChallengingTheServerTax.aspx`.

[60] Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. A File is Not a File: Understanding the I/O Behavior of Apple Desktop Applications. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '11)*, pages 71–83, Cascais, Portugal.

[61] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the USENIX Winter Technical Conference (USENIX Winter '92)*, San Francisco, CA, 1992.

[62] Dave Hitz, James Lau, and Michael Malcolm. File System Design for an NFS File Server Appliance. In *Proceedings of the USENIX Winter Technical Conference (USENIX Winter '94)*, San Francisco, California, January 1994.

[63] Andy A. Hwang, Ioan A. Stefanovici, and Bianca Schroeder. Cosmic rays don't strike twice: understanding the nature of dram errors and the implications for system design. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*, London, UK, March 2012.

[64] Dell T. J. A white paper on the benefits of chipkill- correct ecc for pc server main memory. *IBM Microelectronics Division*, 1997.

[65] Minwen Ji, Alistair C Veitch, and John Wilkes. Seneca: remote mirroring done write. In *Proceedings of the USENIX Annual Technical Conference (USENIX '03)*, San Antonio, Texas, June 2003.

[66] Wei-lun Kao, Ravishankar K. Iyer, and Dong Tang. FINE: A Fault Injection and Monitoring Environment for Tracing the UNIX System Behavior Under Faults. In *IEEE Transactions on Software Engineering*, pages 1105–1118, 1993.

[67] Osama Khan, Randal Burns, James Plank, William Pierce, and Cheng Huang. Rethinking erasure codes for cloud file systems: minimizing i/o for recovery and degraded reads. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST '12)*, San Jose, California, February 2012.

[68] J.J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Transactions on Computer Systems*, 10(1), February 1992.

[69] Andrew Krioukov, Lakshmi N. Bairavasundaram, Garth R. Goodson, Kiran Srinivasan, Randy Thelen, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Parity Lost and Parity Regained. In *Proceedings of the 6th USENIX Symposium on File and Storage Technologies (FAST '08)*, pages 127–141, San Jose, California, February 2008.

[70] Swetha Krishnan, Giridhar Ravipati, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Barton P. Miller. The Effects of Metadata Corruption on NFS. In *Proceedings of the 3rd International Workshop on Storage Security and Survivability (StorageSS'07)*, Alexandria, Virginia, October 2007.

[71] Xin Li, Michael C. Huang, Kai Shen, and Lingkun Chu. A realistic evaluation of memory hardware errors and software system susceptibility. In *Proceedings of the USENIX Annual Technical Conference (USENIX '10)*, Boston, Massachusetts, June 2010.

[72] Xin Li, Kai Shen, Michael C. Huang, and Lingkun Chu. A memory soft error measurement on production systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX '07)*, Santa Clara, CA, June 2007.

[73] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, Laurent Vivier, and Bull S.A.S. The New Ext4 Filesystem: Current Status and Future Plans. In *Ottawa Linux Symposium (OLS '07)*, Ottawa, Canada, July 2007.

[74] Theresa C. Maxino and Philip J. Koopman. The effectiveness of checksums for embedded control networks. *IEEE Trans. Dependable Secur. Comput.*, 6(1):59–72, January 2009.

[75] T. C. May and M. H. Woods. Alpha-particle-induced soft errors in dynamic memories. *IEEE Trans. on Electron Dev*, 26(1), 1979.

[76] Marshall Kirk McKusick, Willian N. Joy, Samuel J. Leffler, and Robert S. Fabry. Fsck - The UNIX File System Check Program. Unix System Manager's Manual - 4.3 BSD Virtual VAX-11 Version, April 1986.

[77] Nimrod Megiddo and Dharmendra Modha. Arc: A self-tuning, low overhead replacement cache. In *Proceedings of the 2nd USENIX Symposium on File and Storage Technologies (FAST '03)*, San Francisco, California, April 2003.

[78] Ralph C. Merkle. A digital signature based on a conventional encryption function. In *A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology (CRYPTO '87)*, 1987.

[79] Microsoft. How ntfs works. `http://technet.microsoft.com/en-us/library/cc781134(v=ws.10).aspx`, March 2003.

[80] Dejan Milojicic, Alan Messer, James Shau, Guangrui Fu, and Alberto Munoz. Increasing relevance of memory hardware errors: a case for recoverable programming models. In *Proceedings of the 9th Workshop on ACM SIGOPS European Workshop*, 2000.

[81] Bill Moore. Ditto Blocks - The Amazing Tape Repellent. `http://blogs.sun.com/bill/entry/ditto_blocks_the_amazing_tape`.

[82] Nathan Rutman. Improvements in Lustre Data Integrity. `http://legacy.xyratex.com/pdfs/lustre/Improvements_in_Lustre_Data_Integrity.pdf`.

[83] Eugene Normand. Single event upset at ground level. *Nuclear Science, IEEE Transactions on*, 43(6):2742–2750, 1996.

[84] T. J. O'Gorman, J. M. Ross, A. H. Taber, J. F. Ziegler, H. P. Muhlfeld, C. J. Montrose, H. W. Curtis, and J. L. Walsh. Field testing for cosmic ray soft errors in semiconductor memories. *IBM Journal of Research and Development*, 40(1):41–50, 1996.

[85] Michael A. Olson, Keith Bostic, and Margo Seltzer. Berkeley db. In *Proceedings of the USENIX Annual Technical Conference (USENIX '99)*, Monterey, California, June 1999.

[86] David Patterson, Garth Gibson, and Randy Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the 1988 ACM SIGMOD Conference on the Management of Data (SIGMOD '88)*, pages 109–116, Chicago, Illinois, June 1988.

[87] Martin K. Petersen. Linux Data Integrity Extensions. In *Linux Symposium*, 2008.

[88] Zachary Peterson and Randal Burns. Ext3cow: a time-shifting file system for regulatory compliance. *Trans. Storage*, 1(2):190–212, 2005.

[89] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. IRON File Systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 206–220, Brighton, United Kingdom, October 2005.

[90] Feng Qin, Shan Lu, and Yuanyuan Zhou. Safemem: Exploiting ecc-memory for detecting memory leaks and memory corruption during production runs. In *In Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA 05')*, 2005.

[91] Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: The Linux B-Tree Filesystem. *ACM Transactions on Storage (TOS)*, 9(3):9:1–9:32, August 2013.

[92] E. Rozier, W. Belluomini, V. Deenadhayalan, J. Hafner, K.K. Rao, and P. Zhou. Evaluating the impact of undetected disk errors in raid systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '09)*, Lisbon, Portugal, June 2009.

[93] rsync. rsync. `http://www.samba.org/rsync/`.

[94] Jerome H. Saltzer, David P. Reed, and David D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

[95] Russel Sandberg. The Design and Implementation of the Sun Network File System. In *Proceedings of the 1985 USENIX Summer Technical Conference*, pages 119–130, Berkeley, CA, June 1985.

[96] Bianca Schroeder, Sotirios Damouras, and Phillipa Gill. Understanding latent sector errors and how to protect against them. In *Proceedings of the 8th USENIX Symposium on File and Storage Technologies (FAST '10)*, San Jose, California, February 2010.

[97] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. DRAM errors in the wild: a large-scale field study. In *Proceedings of the 2009 Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS/Performance '09)*, Seattle, Washington, June 2009.

[98] Thomas J.E. Schwarz, Qin Xin, Ethan L. Miller, Darrell D.E. Long, Andy Hospodor, and Spencer Ng. Disk Scrubbing in Large Archival Storage Systems. In *Proceedings of the 12th Annual Meeting of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, Volendam, Netherlands, October 2004.

[99] Seafile. Seafile. `http://seafile.com/en/home/`.

[100] Tezzaron Semiconductor. Soft errors in electronic memory - a white paper. 2004.

[101] D.P. Siewiorek, J.J. Hudak, B.H. Suh, and Z.Z. Segal. Development of a Benchmark to Measure System Robustness. In *Proceedings of the 23rd International Symposium on Fault-Tolerant Computing (FTCS-23)*, Toulouse, France, June 1993.

[102] SNIA Technical Proposal. Architectural Model for Data Integrity. `http://snia.org/sites/default/files/Data_Integrity_Architectural_Model_v1.0.pdf`.

[103] sparkleshare. Sparkleshare. `http://sparkleshare.org`.

[104] Christopher A. Stein, John H. Howard, and Margo I. Seltzer. Unifying File System Protection. In *Proceedings of the USENIX Annual Technical Conference (USENIX '01)*, pages 79–90, Boston, Massachusetts, June 2001.

[105] Sriram Subramanian, Yupu Zhang, Rajiv Vaidyanathan, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Jeffrey F. Naughton. Impact of Disk Corruption on Open-Source DBMS. In *Proceedings of the 26th International Conference on Data Engineering (ICDE '10)*, Long Beach, California, March 2010.

[106] M. Sullivan and R. Chillarege. Software defects and their impact on system availability-a study of field failures in operating systems. In *Proceedings of the 21st International Symposium on Fault-Tolerant Computing (FTCS-21)*, Montreal, Canada, June 1991.

[107] Sun Microsystems. Solaris Internals: FileBench. `http://www.solarisinternals.com/wiki/index.php/FileBench`.

[108] Sun Microsystems. ZFS On-Disk Specification. `http://www.opensolaris.org/os/community/zfs/docs/ondiskformat0822.pdf`.

[109] Rajesh Sundaram. The Private Lives of Disk Drives. `http://partners.netapp.com/go/techontap/matl/sample/0206tot_resiliency.html`.

[110] Adan Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the XFS File System. In *Proceedings of the USENIX Annual Technical Conference (USENIX '96)*, San Diego, California, January 1996.

[111] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the Reliability of Commodity Operating Systems. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing, New York, October 2003.

[112] T10 Technical Committee. SCSI Block Commands - 3. `http://www.t10.org/members/w_sbc3.htm`.

[113] The Data Clinic. Hard Disk Failure. `http://www.dataclinic.co.uk/hard-disk-failures.htm`.

[114] T. K. Tsai and R. K. Iyer. Measuring Fault Tolerance with the FTAPE Fault Injection Tool. In *The 8th International Conference On Modeling Techniques and Tools for Computer Performance Evaluation*, pages 26–40, September 1995.

[115] Patrick Tucker. Has big data made anonymity impossible? http://www.technologyreview.com/news/514351/has-big-data-made-anonymity-impossible/.

[116] Stephen C. Tweedie. Journaling the Linux ext2fs File System. In *The Fourth Annual Linux Expo*, Durham, North Carolina, May 1998.

[117] John Wehman and Peter den Haan. The Enhanced IDE/Fast-ATA FAQ. http://thef-nym.sci.kun.nl/cgi-pieterh/atazip/atafq.html.

[118] Glenn Weinberg. The Solaris Dynamic File System. http://members.visi.net/~thedave/sun/DynFS.pdf.

[119] Zev Weiss, Tyler Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. ROOT: Replaying Multithreaded Traces with Resource-Oriented Ordering. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, Farmington, PA, November 2013.

[120] Andre Wenas. ZFS FAQ. http://blogs.sun.com/awenas/entry/zfs_faq.

[121] Wim Coekaerts. ASMLib. https://blogs.oracle.com/wim/entry/asmlib.

[122] Microsoft Windows. Skydrive. http://windows.microsoft.com/en-us/skydrive/download.

[123] Wuala. Wuala. http://www.wuala.com/.

[124] Yichen Xie, Andy Chou, and Dawson Engler. Archer: using symbolic, path-sensitive analysis to detect memory access errors. In *Proceedings of the 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '03)*, Helsinki, Finland, September 2003.

[125] Junfeng Yang, Can Sar, and Dawson Engler. EXPLODE: A Lightweight, General System for Finding Serious Storage System Errors. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, Seattle, Washington, November 2006.

[126] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using Model Checking to Find Serious File System Errors. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, San Francisco, California, December 2004.

[127] Yupu Zhang, Chris Dragga, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. ViewBox: Integrating Local File Systems with Cloud Storage Services. In *Proceedings of the 12th USENIX Symposium on File and Storage Technologies (FAST '14)*, Santa Clara, California, February 2014.

[128] Erez Zadok, Ion Badulescu, and Alex Shender. Extending File Systems Using Stackable Templates. In *Proceedings of the USENIX Annual Technical Conference (USENIX '99)*, Monterey, California, June 1999.

[129] Yupu Zhang, Chris Dragga, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. *-Box: Towards Reliability and Consistency in Dropbox-like File Synchronization Services. In *Proceedings of the 5th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '13)*, San Jose, California, June 2013.

[130] Yupu Zhang, Daniel S. Myers, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Zettabyte Reliability with Flexible End-to-end Data Integrity. In *Proceedings of the 29th IEEE Conference on Massive Data Storage (MSST '13)*, Long Beach, CA, May 2013.

[131] Yupu Zhang, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. End-to-end Data Integrity for File Systems: A ZFS Case Study. In *Proceedings of the 8th USENIX Symposium on File and Storage Technologies (FAST '10)*, San Jose, California, February 2010.

[132] Benjamin Zhu, Kai Li, and Hugo Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proceedings of the 6th USENIX Symposium on File and Storage Technologies (FAST '08)*, San Jose, California, February 2008.

[133] J. F. Ziegler and W. A. Lanford. Effect of Cosmic Rays on Computer Memories. *Science*, 206(4420):776–788, 1979.