# End-to-end Data Integrity for File Systems: A ZFS Case Study

Yupu Zhang, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
*Computer Sciences Department, University of Wisconsin-Madison*

## Abstract

*We present a study of the effects of disk and memory corruption on file system data integrity. Our analysis focuses on Sun's ZFS, a modern commercial offering with numerous reliability mechanisms. Through careful and thorough fault injection, we show that ZFS is robust to a wide range of disk faults. We further demonstrate that ZFS is less resilient to memory corruption, which can lead to corrupt data being returned to applications or system crashes. Our analysis reveals the importance of considering both memory and disk in the construction of truly robust file and storage systems.*

## 1 Introduction

One of the primary challenges faced by modern file systems is the preservation of data integrity despite the presence of imperfect components in the storage stack. Disk media, firmware, controllers, and the buses and networks that connect them all can corrupt data [4, 52, 54, 58]; higher-level storage software is thus responsible for both detecting and recovering from the broad range of corruptions that can (and do [7]) occur.

File and storage systems have evolved various techniques to handle corruption. Different types of checksums can be used to detect when corruption occurs [9, 14, 49, 52], and redundancy, likely in mirrored or parity-based form [43], can be applied to recover from it. While such techniques are not foolproof [32], they clearly have made file systems more robust to disk corruptions.

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 [11, 35, 62], and recent studies have confirmed their presence in modern systems [34, 41, 46]. Software can also cause memory corruption; bugs can lead to "wild writes" into random memory contents [18], thus polluting memory; studies confirm the presence of software-induced memory corruptions in operating systems [1, 2, 3, 60].

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 corruptions.

In this paper, we ask: how robust are modern file systems to disk and memory corruptions? To answer this query, we analyze a state-of-the-art file system, Sun Microsystem's ZFS, by performing fault injection tests representative of realistic disk and memory corruptions. We choose ZFS for our analysis because it is a modern and important 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" [14].

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. We further find that many of these cases could be avoided with simple techniques.

The contributions of this paper are:

- To our knowledge, the first study to empirically analyze the reliability of ZFS.
- To our knowledge, the first study to analyze local file system reliability techniques in the face of memory corruption.
- A novel holistic approach to analyzing both disk and memory corruptions using carefully-controlled fault-injection techniques.
- A simple framework to measure the likelihood of different memory corruption failure scenarios.
- Results that demonstrate the importance of both memory and disk in end-to-end data protection.

The rest of this paper is organized as follows. In Section 2, we motivate our work by discussing the problem of disk and memory corruption. In Section 3, we provide some background on the reliability features of ZFS. Section 4 and Section 5 present our analysis of data integrity in ZFS with disk and memory corruptions. Section 6 gives an preliminary analysis of the probabilities of different failure scenarios in ZFS due to memory errors. In Section 7, we present initial results of the data integrity analysis in ext2 with memory corruptions. Section 8 discusses related work and Section 9 concludes our work.

## 2 Motivation

This section provides the motivation for our study by describing how potent the problem of disk and memory corruptions is to file system data integrity. Here, we discuss why such corruptions happen, how frequently they occur, and how systems try to deal with them. We discuss disk and memory corruptions separately.

### 2.1 Disk corruptions

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 (discussed in [6]) in disk drives, where there is an explicit notification from the drive about the error condition.

#### 2.1.1 Why they happen

Disk corruptions happen 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 corruptions in disk blocks [4, 47, 54]. On-disk ECC catches many (but not all) of these corruptions.

Errors are also induced due to bugs in complex drive firmware (modern drives contain hundreds of thousands of lines of firmware code [44]). Some reported firmware problems include a misdirected write where the firmware accidentally writes to the wrong location [58] or a lost write (or phantom write) where the disk reports a write as completed when in fact it never reaches the disk [52]. Bus controllers have also been found to incorrectly report disk requests as complete or to corrupt data [24, 57].

Finally, software bugs in operating systems are also potential sources of corruption. Buggy device drivers can issue disk requests with bad parameters or data [20, 22, 53]. Software bugs in the file system itself can cause incorrect data to be written to disk.

#### 2.1.2 How frequently they happen

Disk corruptions are prevalent across a broad range of modern drives. In a recent study of 1.53 million disk drives over 41 months [7], 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. In addition, there is much anecdotal evidence of corruption in storage stacks [9, 52, 58].

#### 2.1.3 How to handle them

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

**Checksums**: Checksums are block hashes computed with a collision-resistant hash 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 [9] and NetApp Data ONTAP [52]. Similar checksumming techniques have also been used in file systems [14, 42].

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 [32]. Further, checksumming does not protect against corruptions that happen due to bugs in software, typically in large code bases [20, 61].

**Redundancy**: Redundancy in on-disk structures also helps to detect and, in some cases, recover from disk corruptions. For example, some B-Tree file systems such as ReiserFS [15] 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 [16] and ext3 [56] use redundant copies of superblock and group descriptors to recover from corruptions.

However, it has been shown that many of these file systems still sometimes fail to detect corruptions, leading to greater problems [44]. Further, Gunawi et al. show instances where ext2/ext3 file system checkers fail to use available redundant information for recovery [26].

**RAID storage**: Another popular technique is to use a RAID storage system [43] 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 [32]. Finally, not all systems incorporate multiple disks, which limits the applicability of RAID.

### 2.2 Memory corruptions

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.

#### 2.2.1 Why they happen

Errors in the memory chip are one source of memory corruptions. 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 [35]. Zeigler and Lanford found that cosmic rays can also disrupt electronic circuits [62]. More recent studies and measurements confirm the effect of atmospheric neutrons causing single event upsets (SEU) in memories [40, 41].

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 [12], which can result in seemingly random memory corruptions.

#### 2.2.2 How frequently they happen

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 [41] confirmed the presence of errors in RAM and that the upset rates increase with elevation, indicating atmospheric neutrons as the likely cause.

In a recent measurement-based study of memory errors in a large fleet of commodity servers over a period of 2.5 years [46], 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 [34] 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 [2], USCERT Vulnerabilities Notes Database [3], CERT/CC advisories [1], as well as other anecdotal evidence [18] show cases of memory corruption happening due to software bugs.

#### 2.2.3 How to handle them

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

**ECC**: Traditionally, memory systems have employed Error Correction Codes [19] 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 [23]. Further, many commodity systems simply do not use ECC protection in order to reduce cost [28].

More sophisticated techniques like Chipkill[30] 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 corruptions open in commodity systems.

**Programming models and tools**: Another approach to deal with memory errors is to use recoverable programming models [38] at different levels (firmware, operating system, and applications). However, such techniques require support from hardware to detect memory corruptions. Further, a holistic change in software is required to provide recovery solution at various levels.

Much effort has also gone into detecting software bugs which cause memory corruptions. Tools such as metal [27] and CSSV [21] apply static analysis to detect memory corruptions. Others such as Purify [29] and SafeMem [45] use dynamic monitoring to detect memory corruptions at runtime. However, as discussed in Section 2.2.2, software-induced memory corruptions still remain a problem.

### 2.3 Summary

In modern systems corruption occurs both within the storage system and in memory. Many commercial systems apply sophisticated techniques to detect and recover from disk-level corruptions; beyond ECC, little is done to protect against memory-level problems. Therefore, the protection of critical user data against memory corruptions is largely left to software.

## 3 ZFS reliability features

ZFS is a state-of-the-art file system from Sun which takes a unified approach to data management. It provides data integrity, transactional consistency, scalability, and a multitude of useful features such as snapshots, copy-on-write clones, and simple administration [14].

In terms of reliability, 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 [37]. 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 (*ditto blocks*) of the block being referenced. By default ZFS stores multiple copies for 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 [59], 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 file system 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. Further, ZFS also introduces a new data replication model, RAID-Z, a novel solution similar to RAID-5 but using a variable stripe width to eliminate the write-hole issue in RAID-5 [13]. Finally, ZFS provides automatic repairs in mirrored configurations and provides a disk scrubbing facility to detect latent sector errors.

# 4 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.

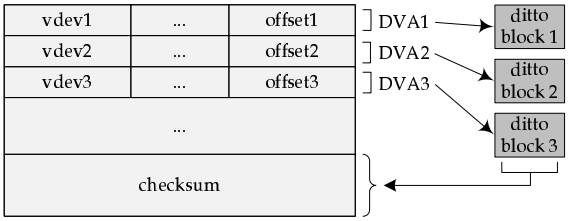


Figure 1: **Block pointer.** *The figure shows how the block pointer structure points to (up to) three copies of a block (ditto blocks), and keeps a single checksum.*

We find that ZFS is able to detect all and recover from most disk corruptions. We present our analysis, including methodology and results in later sections. First, we present a brief background about the on-disk organization in ZFS, focusing on how data integrity is maintained.

## 4.1 ZFS on-disk organization

All on-disk data and metadata in ZFS are treated as objects, where an object is a collection of blocks. Objects are further grouped into object sets. Other structures such as uberblocks are also used to organize data on disk. We now discuss these basic on-disk structures and their usage in ZFS.

### 4.1.1 Basic structures

**Block pointers**: A block pointer is the basic structure in ZFS for addressing a block on disk. It provides a generic mechanism to keep parental checksums and replicas of on-disk blocks. Figure 1 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 [39]. 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 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

| 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 its 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 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 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.*

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.

**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 described 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.

**Other structures**: ZFS uses other structures to organize on-disk data. 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 [51]. An *uberblock* (similar to a superblock) inside the vdev label is used to provide access to the pool data and verify its integrity. The uberblock is self-checksummed and updated atomically.

#### 4.1.2 On-disk layout

In this section, we present some details about 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 [51].

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 2 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 simple explanation of all visited objects is described in Table 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.

**Find pool metadata** (steps 1-2): As the starting point, ZFS locates the active uberblock in the vdev label of the device. ZFS then uses the uberblock to locate and verify the integrity of pool-wide metadata contained in an object set called Meta Object Set (MOS). There are three copies of the object set block representing the MOS.

**Find a file system** (steps 3-10): To locate a file system, ZFS accesses a series of objects in MOS, all of which have three ditto blocks. Once the dataset representing "myfs" is found, it is used to access file system wide metadata contained in an object set. The integrity of file system metadata is checked using the block pointer in the dataset, which points to the object set block. All file system metadata blocks have two ditto copies.

**Find a file and a data block** (steps 11-18): To locate a file, ZFS then uses the directory objects in the "myfs" object set. Finally, by following the block pointers in the dnode of the file object, ZFS finds the required data block. The integrity of every traversed block is confirmed by verifying the checksum in its block pointers.

The legend in Figure 2 shows a summary of all the on-disk block types visited during the traversal. Our fault injection tests for analyzing robustness of ZFS against disk corruptions (discussed in the next subsection) inject bit errors in the on-disk blocks shown in Figure 2.

### 4.2 Methodology of analysis

In this section, we discuss 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.

#### 4.2.1 Fault injection framework

Our experiments are performed on a 64-bit Solaris Express Community Edition (build 108) virtual machine with 2GB non-ECC memory. We use ZFS pool version 14 and ZFS filesystem 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

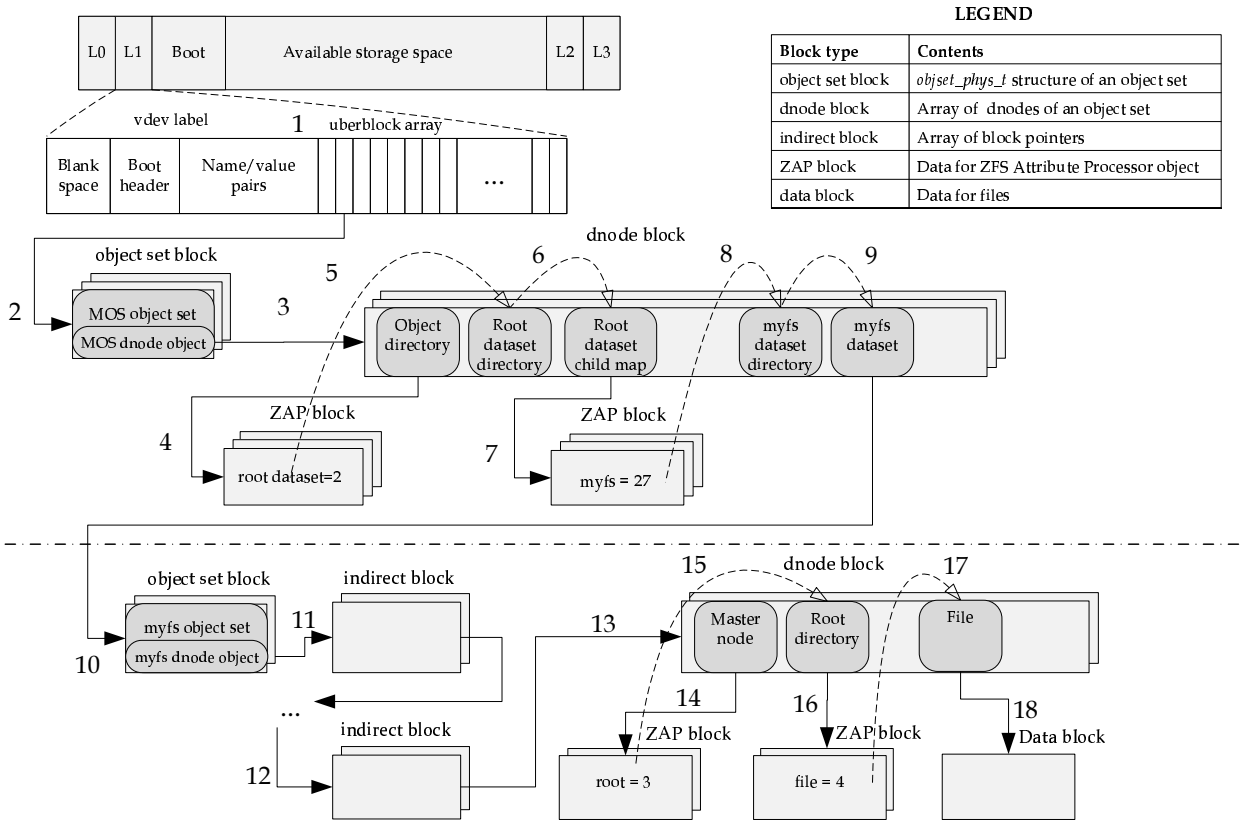


Figure 2: **ZFS on-disk structures.** *The figure shows the on-disk structures of ZFS including the pool-wide metadata and file system metadata. In the example above, the zpool contains a sample file system named "myfs". All ZFS on-disk data structures are shown by rounded boxes, and on-disk blocks are 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.*

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.

#### 4.2.2 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.

### 4.3 Results and observations

The results of our fault injection experiments are shown in Table 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 ver-

| 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 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.*

ify 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 available 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.

# 5 In-memory data integrity in ZFS

In the last section we showed the robustness of ZFS to disk corruptions. 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. Our fault injection experiments indicate 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.

This section is organized as follows. First, we briefly describe ZFS in-memory structures. Then, we discuss the test methodology and workloads we used to conduct the analysis. Finally, we present the experimental results and our observations.

## 5.1 ZFS in-memory structures

In order to better understand the in-memory experiments, we present some background information on ZFS in-memory structures.

### 5.1.1 In-memory structures

ZFS in-memory structures can be classified into two categories: those that exist in the page cache and those that

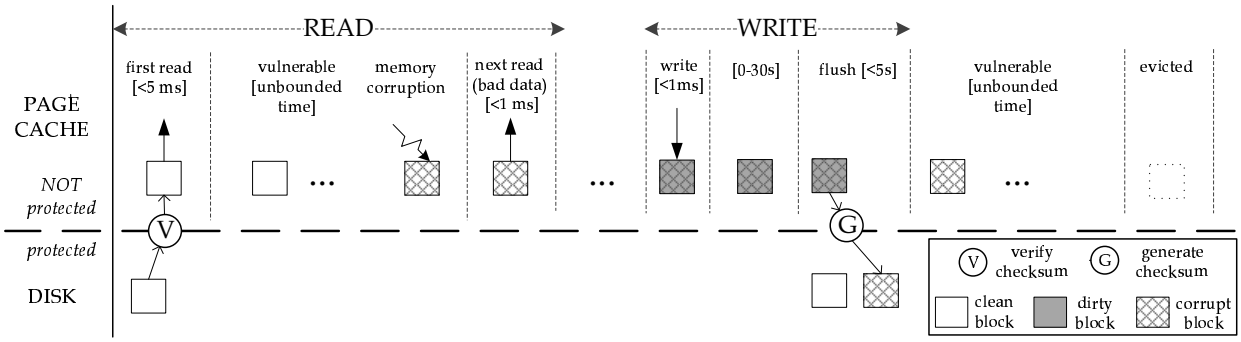


Figure 3: **Lifecycle of a block.** *This figure illustrates one example of the lifecycle of a block. The left half represents the read timeline and the right half represents the write timeline. The black dotted line is a protection boundary, below which a block is protected by the checksum, otherwise unprotected.*

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 [36], and stay there until evicted. Data blocks are stored only in 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.

### 5.1.2 Lifecycle of a block

To help the reader understand the vulnerability of ZFS to memory corruptions discussed in later sections, Figure 3 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. On first read of a target block not in the page cache, 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. 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 2), 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.

## 5.2 Methodology of analysis

In this section, 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.

### 5.2.1 Test procedure and workloads

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

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

| 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 4: **Summary of data structures and fields corrupted.** *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.*

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. Table 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.

For metadata, in our fault injection experiments, we covered a broad range of metadata structures. However, 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.

## 5.3 Results and observations

We present the results of our in-memory experiments in Table 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 5 is a subset of the results showing only 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 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

| | | 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 5: **In-memory corruption results.** *The table shows a subset of 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.*

such as ditto blocks and the mirrored zpool are not triggered to recover from the damage.

The results in Table 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.

**Observation 2:** *The window of vulnerability of blocks in the page cache is unbounded.* As Figure 3 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 5.1.2, 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), 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.

We now discuss our observations about user-visible results of memory corruptions.

**Observation 6:** *When metadata is corrupted, operations fail with wrong results, or give misleading error messages (E).* As shown in Table 5, when `zp_flags` in `dnode_phys_t` for a file object was corrupted, in one case open() returned an error code EACCES (permission denied). This case occurred when the 41$^{st}$ 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; there is one case where "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".

A case where wrong results are returned occurred when `dd_child_dir_zapobj` was corrupted. This field refers to a dataset child map object containing references to child file systems. On corrupting this field, "zfs list", which should list all file systems in the pool, did not list the child file systems of the corrupted dataset directory.

**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 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 many corruptions. Checksums 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.

# 6 Probability of bit-flip induced failures

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 complete (S). These probabilities help us to understand how severely filesystem data integrity is affected by memory corruptions and how much effort filesystem developers should make to add extra protection to maintain data integrity.

## 6.1 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. Therefore, 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 [50] 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.

## 6.2 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,
$$P_1(X) = 1 - (1 - P_k(X))^{\frac{1}{k}}, \text{ when } X = R, W \text{ or } C$$
$$P_1(X) = (P_k(X))^{\frac{1}{k}}, \text{ when } X = S$$

## 6.3 Results

The analysis is performed on the same virtual machine as mentioned in Section 4.2.1. The machine is configured with 2GB non-ECC memory and a single disk running ZFS. We first ran some controlled micro-benchmarks (e.g., sequential read) to verify that the 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 [50].

| **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 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).*

Table 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 6, we see that a single bit flip in memory causes a small but non-negligible percentage of runs to experience an 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.

# 7 Beyond ZFS

In addition to ZFS, we have applied the same fault injection framework used in Section 5 to a simpler filesystem, ext2. Our initial results indicate that ext2 is also vulnerable to memory corruptions. For example, corrupt data can be returned to the user or written to disk. When certain fields of a VFS inode are corrupted, operations on that inode fail or the whole system crashes. If the inode is dirty, the corrupted fields of the VFS inode are propagated to the inode in the page cache and are then written to disk, making the corruptions permanent. Moreover, if the superblock in the page cache is corrupted and flushed

to disk, it might result in an unmountable filesystem.

In summary, so far we have studied two extremes: ZFS, a complex filesystem with many techniques to maintain on-disk data integrity, and ext2, a simpler filesystem with few mechanisms to provide extra reliability. Both are vulnerable to memory corruptions. It seems that regardless of the complexity of the file system and the amount of machinery used to protect against disk corruptions, memory corruptions are still a problem.

## 8 Related work

Software-implemented fault injection techniques have been widely used to analyze the robustness of systems [10, 17, 25, 31, 48, 55]. For example, FINE used fault injection to emulate hardware and software faults in the operating system [31]; Weining et al. [25] injected faults to instruction streams of Linux kernel function to characterize Linux kernel behavior.

More recent works [5, 8, 44] have applied type-aware fault injection to analyze failure behaviors of different file systems to disk corruptions. Our analysis of on-disk data integrity in ZFS is similar to these studies.

Further, fault injection has also been used to analyze effects of memory corruptions on systems. FIAT [10] used fault injection to study the effects of memory corruptions in a distributed environment. Krishnan et al. applied a memory corruption framework to analyze the effects of metadata corruption on NFS [33]. Our study on in-memory data integrity is related to these studies in their goal of finding effects of memory corruptions.

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.

## 9 Summary and discussion

In this paper, 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 end-to-end data integrity as a goal. File systems should not only provide protection against disk corruptions, but also aim to protect data from memory corruptions. Although dealing with memory corruptions is hard, we conclude by discussing some techniques that file systems can use to increase protection against memory corruptions.

**Block-level checksums in the page cache**: File systems could protect the vulnerable data and metadata blocks in the page cache by using checksums. For example, ZFS could use the checksums inside block pointers in the page cache, update them on block updates, and verify the checksums on reads. However, this does incur an overhead in computation as well as some complexity in implementation; these are always the tradeoffs one has to make for reliability.

**Metadata checksums in the heap**: Even with block-level checksums in the page cache, there are still copies of metadata structures in the heap that are vulnerable to memory corruptions. To provide end-to-end data integrity, data-structure checksums may be useful in protecting in-heap metadata structures.

**Programming for error detection**: Many serious effects of memory corruptions can be mitigated by using simple programming practices. One technique is to use existing redundancy in data structures for simple consistency checks. For instance, the case described in Observation 8 (Section 5.3) could be detected by comparing the expected level calculated from the `dn_levels` field of `dnode_phys_t` with the actual level stored inside the first block pointer. Another simple technique is to include magic numbers in metadata structures for sanity checking. For example, some "crash" cases happened due to bad block pointers obtained during the block tree traversal (Observation 7 in Section 5.3). Using a magic number in block pointers could help detect such cases and prevent unexpected behavior.

## Acknowledgment

We thank the anonymous reviewers and Craig Soules (our shepherd) for their tremendous feedback and comments, which have substantially improved the content and presentation of this paper. We thank Asim Kadav for his initial work on ZFS on-disk fault injection. We also thank the members of the ADSL research group for their insightful comments.

This material is based upon work supported by the National Science Foundation under the following grants: CCF-0621487, CNS-0509474, CNS-0834392, CCF-0811697, CCF-0811697, CCF-0937959, as well as by generous donations from NetApp, Inc, Sun Microsystems, and Google.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or other institutions.

## References


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

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

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

[4] D. Anderson, J. Dykes, and E. Riedel. More Than an Interface: SCSI vs. ATA. In *FAST*, 2003.

[5] L. N. Bairavasundaram, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Dependability Analysis of Virtual Memory Systems. In *DSN*, 2006.

[6] L. N. Bairavasundaram, G. R. Goodson, S. Pasupathy, and J. Schindler. An Analysis of Latent Sector Errors in Disk Drives. In *SIGMETRICS*, 2007.

[7] L. N. Bairavasundaram, G. R. Goodson, B. Schroeder, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. An Analysis of Data Corruption in the Storage Stack. In *FAST*, 2008.

[8] L. N. Bairavasundaram, M. Rungta, N. Agrawal, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. M. Swift. Analyzing the Effects of Disk-Pointer Corruption. In *DSN*, 2008.

[9] W. Bartlett and L. Spainhower. Commercial Fault Tolerance: A Tale of Two Systems. *IEEE Trans. on Dependable and Secure Computing*, 1(1), 2004.

[10] J. Barton, E. Czeck, Z. Segall, and D. Siewiorek. Fault Injection Experiments Using FIAT. *IEEE Trans. on Comp.*, 39(4), 1990.

[11] R. Baumann. Soft errors in advanced computer systems. *IEEE Des. Test*, 22(3):258–266, 2005.

[12] E. D. Berger and B. G. Zorn. Diehard: probabilistic memory safety for unsafe languages. In *PLDI*, 2006.

[13] J. Bonwick. RAID-Z. http://blogs.sun.com/bonwick/entry/raid_z.

[14] J. Bonwick and B. Moore. ZFS: The Last Word in File Systems. http://opensolaris.org/os/community/zfs/docs/zfs_last.pdf.

[15] F. Buchholz. The structure of the Reiser file system. http://homes.cerias.purdue.edu/∼florian/reiser/reiserfs.php.

[16] R. Card, T. Ts'o, and S. Tweedie. Design and Implementation of the Second Extended Filesystem. In Proceedings of the First Dutch International Symposium on Linux, 1994.

[17] J. Carreira, H. Madeira, and J. G. Silva. Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers. *IEEE Trans. on Software Engg.*, 1998.

[18] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta. Hive: Fault Containment for Shared-Memory Multiprocessors. In *SOSP*, 1995.

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

[20] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An Empirical Study of Operating System Errors. In *SOSP*, 2001.

[21] N. Dor, M. Rodeh, and M. Sagiv. CSSV: towards a realistic tool for statically detecting all buffer overflows in C. In *PLDI*, 2003.

[22] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *SOSP*, 2001.

[23] A. Eto, M. Hidaka, Y. Okuyama, K. Kimura, and M. Hosono. Impact of neutron flux on soft errors in mos memories. In *IEDM*, 1998.

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

[25] W. Gu, Z. Kalbarczyk, R. K. Iyer, and Z. Yang. Characterization of Linux Kernel Behavior Under Errors. In *DSN*, 2003.

[26] H. S. Gunawi, A. Rajimwale, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. SQCK: A Declarative File System Checker. In *OSDI*, 2008.

[27] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *PLDI*, 2002.

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

[29] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *USENIX Winter*, 1992.

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

[31] W. Kao, R. K. Iyer, and D. Tang. FINE: A Fault Injection and Monitoring Environment for Tracing the UNIX System Behavior Under Faults. In *IEEE Trans. on Software Engg.*, 1993.

[32] A. Krioukov, L. N. Bairavasundaram, G. R. Goodson, K. Srinivasan, R. Thelen, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Parity Lost and Parity Regained. In *FAST*, 2008.

[33] S. Krishnan, G. Ravipati, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and B. P. Miller. The Effects of Metadata Corruption on NFS. In *StorageSS*, 2007.

[34] X. Li, K. Shen, M. C. Huang, and L. Chu. A memory soft error measurement on production systems. In *USENIX*, 2007.

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

[36] N. Megiddo and D. Modha. Arc: A self-tuning, low overhead replacement cache. In *FAST*, 2003.

[37] R. C. Merkle. A digital signature based on a conventional encryption function. In *CRYPTO*, 1987.

[38] D. Milojicic, A. Messer, J. Shau, G. Fu, and A. Munoz. Increasing relevance of memory hardware errors: a case for recoverable programming models. In *ACM SIGOPS European Workshop*, 2000.

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

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

[41] 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 J. Res. Dev.*, 40(1):41–50, 1996.

[42] Oracle Corporation. Btrfs: A Checksumming Copy on Write Filesystem. http://oss.oracle.com/projects/btrfs/.

[43] D. Patterson, G. Gibson, and R. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *SIGMOD*, 1988.

[44] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. IRON File Systems. In *SOSP*, 2005.

[45] F. Qin, S. Lu, and Y. Zhou. Safemem: Exploiting ecc-memory for detecting memory leaks and memory corruption during production runs. In *HPCA*, 2005.

[46] B. Schroeder, E. Pinheiro, and W.-D. Weber. DRAM errors in the wild: a large-scale field study. In *SIGMETRICS*, 2009.

[47] T. J. Schwarz, Q. Xin, E. L. Miller, D. D. Long, A. Hospodor, and S. Ng. Disk Scrubbing in Large Archival Storage Systems. In *MASCOTS*, 2004.

[48] D. Siewiorek, J. Hudak, B. Suh, and Z. Segal. Development of a Benchmark to Measure System Robustness. In *FTCS-23*, 1993.

[49] C. A. Stein, J. H. Howard, and M. I. Seltzer. Unifying File System Protection. In *USENIX*, 2001.

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

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

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

[53] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the Reliability of Commodity Operating Systems. In *SOSP*, 2003.

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

[55] 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*, 1995.

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

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

[58] G. Weinberg. The Solaris Dynamic File System. http://members.visi.net/∼thedave/sun/DynFS.pdf.

[59] A. Wenas. ZFS FAQ. http://blogs.sun.com/awenas/entry/zfs_faq.

[60] Y. Xie, A. Chou, and D. Engler. Archer: using symbolic, path-sensitive analysis to detect memory access errors. In *FSE*, 2003.

[61] J. Yang, C. Sar, and D. Engler. EXPLODE: A Lightweight, General System for Finding Serious Storage System Errors. In *OSDI*, 2006.

[62] J. F. Ziegler and W. A. Lanford. Effect of cosmic rays on computer memories. *Science*, 206(4420):776–788, 1979.