Zettabyte Reliability with Flexible End-to-end Data Integrity

Yupu Zhang, Daniel S. Myers, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
Department of Computer Sciences, University of Wisconsin – Madison
{yupu,dsmyers,dusseau,remzi}@cs.wisc.edu

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

I. Introduction

Preserving data integrity is one of the most important responsibilities of modern storage systems; not surprisingly, many techniques have been developed and applied to improve integrity over the years. For example, different checksums are widely applied to many components, including disks [10], system buses [5], and network protocols [6]. Redundancy, especially in the form of RAID [30], is commonly used to provide recovery.

Unfortunately, despite the presence of various protection techniques, data corruption still occurs. Rare events such as dropped writes or misdirected writes leave stale or corrupt data on disk [1], [11], [32], [33]. Bits in memory get flipped due to chip defects [19], [22], [36] or radiation [25], [47]. Software bugs are also a source of data corruption, arising from low-level device drivers [41], system kernels [14], [18], and file systems [44], [45]. Even worse, design flaws are not uncommon and can lead to serious data loss or corruption [21].

While many features that storage systems provide require great care and coordination across the many layers of the system (e.g., performance), integrity checks for data protection generally remain isolated within individual components. For example, hard disks have built-in ECC for each sector [10], but the ECCs are rarely exposed to the upper-level system; TCP uses Internet checksums to protect data payload [6], but only during the transmission. When data is transferred across components, data is not protected and thus may become silently corrupted.

A more comprehensive approach to data protection should embrace the "end to end" philosophy [34]. 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 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.

To explore this flexible approach, we design and implement flexible end-to-end data integrity within Zettabyte File System [12], resulting a new variant which we call Zettabyte-reliable ZFS (Z^{2}FS). It exposes checksums to the application, and passes checksums through the page cache down to the disk system, thus enabling end-to-end verification. It uses two techniques to provide flexible data protection. The first is *checksum chaining*, which is needed to safely convert from one checksum to another when crossing 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^{2}ZFS), 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^{2}FS 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

978-1-4799-0218-7/13/$31.00 © 2013 IEEE

Zettabyte Reliability, one undetected corruption per Zettabyte read, as a reliability goal of storage systems. Guided by the reliability goal, we use the framework throughout the paper 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 in time. 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%.

The contribution of this paper are as follows:

- We propose the concept of flexible end-to-end data integrity, which dynamically trades off the reliability and performance of a storage system by carefully choosing checksums for data in different places in the stack and changing checksums over time.
- We introduce an analytical framework to reason about checksum choices based on the effectiveness of checksum algorithms and corruption rate of devices. The framework provides ways to evaluate and compare the overall reliability of storage systems.
- We develop Z^2FS, which provides end-to-end protection from application to disk with minimal performance overhead. The design, implementation and evaluation of Z^2FS, combined with the reliability analysis using the framework, demonstrate a holistic way to think about the performance-reliability tradeoff in storage systems.

The rest of the paper is organized as follows. In Section II, we introduce the framework for evaluating reliability of storage systems. We then present the design and implementation of E^2ZFS and Z^2FS in Section III and evaluate both systems in Section IV. Finally, we discuss related work in Section V and conclude in Section VI.

II. 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 probability of undetected data corruption.

A. 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 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 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 ($P_{sys-udc}$).

B. 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 paper. We make assumptions (e.g., independence of bit errors) to simplify our models such that we can focus on reasoning the reliability of storage systems within the framework; discussion on more complex and accurate models is beyond the scope of this paper.

1) 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 bit (4KB).

a) 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 [10], [35]. 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 [11], [33]), 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 bitflips 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 bitflips (from exact 1 bitflip to exact b bitflips):

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

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} [33]. 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 $-log_{10}(\text{UBER})$.

b) 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 [25], [47]. Earlier studies show that memory errors can occur at a rate of 10 to 360 errors/year/GB [27], [28], [37] and suspect that most errors are soft errors, which are transient. However, recent studies show that memory errors occur more frequently [19], [22], [36] and are probably dominated by hard errors (actual device defects). If a memory module has ECC or more complex codes such as chipkill [20], then both soft errors and hard errors within the capability of the codes can be detected or corrected. However, corruption caused by software bugs [39] 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 bitflip, 1 FIT/Mbit means there is one bitflip in one billion hours per Mbit. Assuming each bitflip is independent and the same bit can only experience one flip, we model the number of bitflips 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 bitflips 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 [23] and as high as 167,066 [19]. 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 paper, 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 [36]. 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.

2) 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 bitflips. We focus on two types of checksum: xor (64-bit) and Fletcher (256-bit).

Our approach is similar to the one used in a recent study on checksums for embedded control networks [24]. 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 bitflips that is undetectable by the checksum. We use $F(C)$ to represent the fraction of n bitflips that are undetectable by checksum C. Then, the probability of undetectable n bitflips is $P_c(D,n)\times F(C)$, in which $P_c(D,n)$ is the probability of n bitflips on device D. The actual P_{udc} is the sum of the probabilities of undetectable bitflips from n to b (the size of the block is b bits). Since the occurrence of more than n bitflips is highly unlikely, the probability of undetected n bitflips dominates P_{udc} [24]. 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 [24]: $F(xor) = \frac{b-k}{k(b-1)}$. 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 [4]: $F(Fletcher) = 4.16 \times 10^{-20}$. Therefore, $P_{udc}(D, Fletcher) = P_c(D,3) \times (4.16 \times 10^{-20})$.

C. 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 paper, 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.

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

TABLE I: **Sample System Configurations.** *This table shows four configurations of a local file system that we will study throughout the paper.*

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 low enough such that SDC is extremely rare, but at the same time it should not hinder the system's performance. In this paper, 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. 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.

D. 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 I. 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. Therefore, config 1 has the worst components while config 4 has the best. Config 2 is likely to be a consumer-level system with non-ECC memory and a regular disk. Config 3 may be representative of a server with ECC memory.

The timeline of a data block from being generated to being accessed is shown in Figure 1a. 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. Since most file systems flush dirty blocks to disk at regular time intervals (usually 30 seconds), we assume $t_1 - t_0$ to be 30 seconds for all blocks in this paper.

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 2a, 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 III-C.

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 and 3 have a score of 7.4. But when the disk is more reliable, memory corruption starts to dominate and the reliability score increases as the memory reliability index increases. When both reliability indices are high, NCFS with config 4 has the best reliability score of 12.8, still less than the Zettabyte reliability goal (17.5).

III. 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 straightforward end-to-end approach, and Z^{2}FS, which employs the 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.

A. 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 checksums, data recovery with replicas, and consistency with a copy-on-write transactional model [12]. In addition, other mechanisms such as pooled storage, inline deduplication, snapshots, and clones, provide efficient data management.

1) Background: 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 1b, 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 [12].

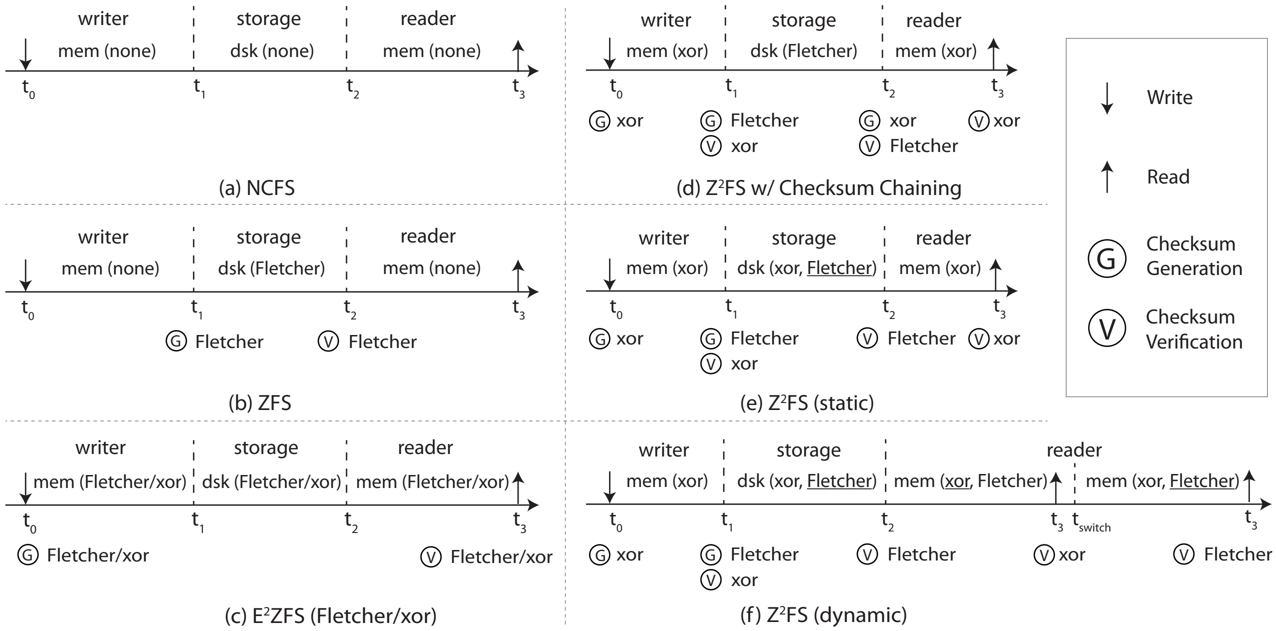


Fig. 1: **Timeline of a Data Block.** *This figure shows timeline of a block from being generated by the writer (t_0) to being read by the reader (t_3) in NCFS, ZFS, E^{2}ZFS, and Z^{2}FS. Each timeline consists of three parts: writer in memory, storage (disk), and reader in memory. The name of the checksum used to protect data during each time period is listed in the parentheses on the right of the device name. When there are two checksums during a time period, the underlined checksum is the primary checksum, as defined in Section III-C.1. Note that in (c), E^{2}ZFS uses the same checksum (either xor or Fletcher) all the way through.*

However, a recent study [46], as well as some anecdotal evidence [3], [8], [9], 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.

2) Reliability Analysis: We apply the framework introduced in Section II 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 2b 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 and config 4 have the highest reliability score of 12.8, but they are still below the reliability goal (17.5). It is interesting to see that config 4 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.

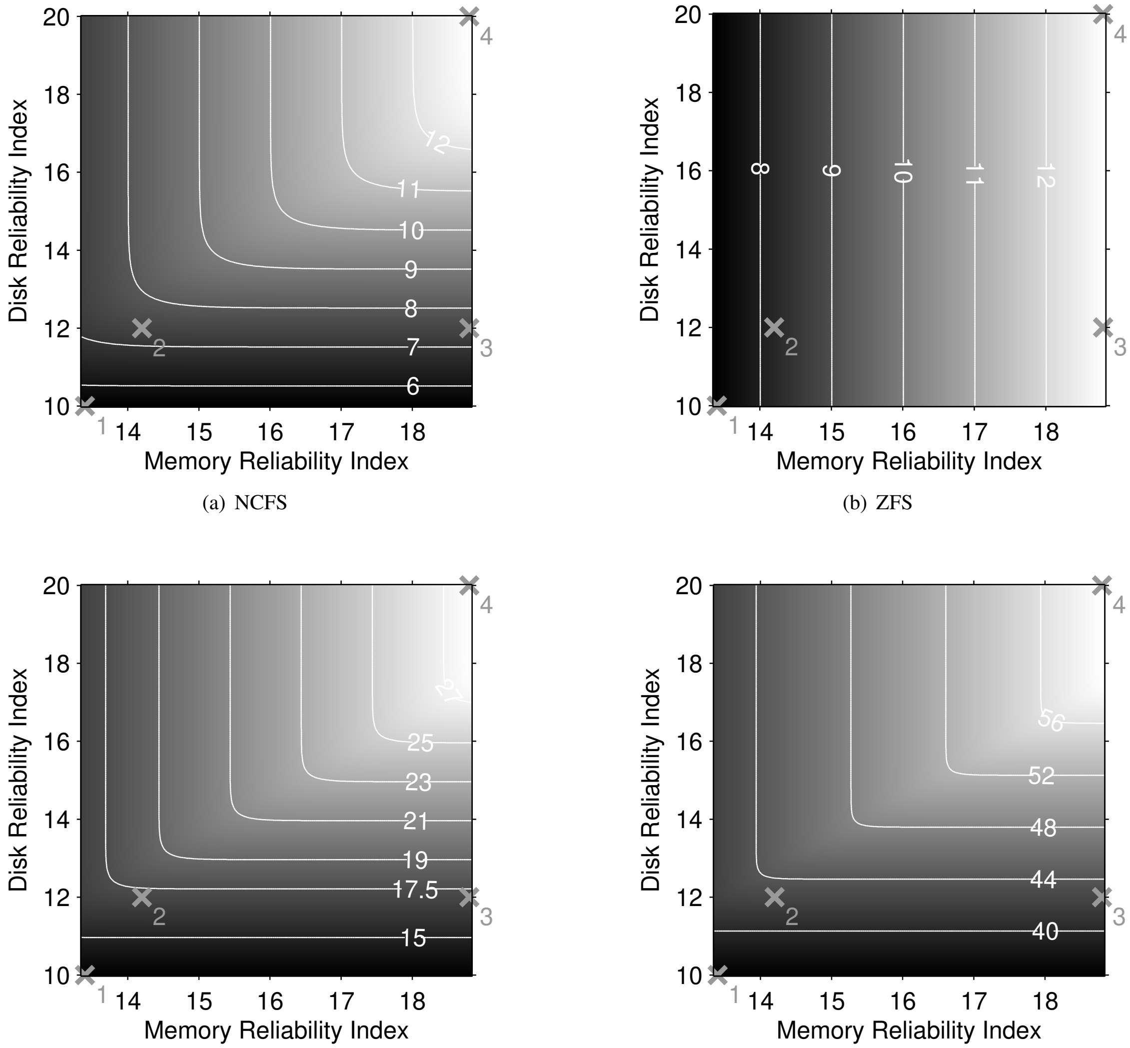
B. *E^{2}ZFS: ZFS with End-to-end Data Integrity*

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.

1) 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



(a) NCFS

(b) ZFS

(c) E^{2}ZFS (xor)

(d) E^{2}ZFS (Fletcher)

Fig. 2: **Reliability Score** ($t_{resident} = 1$). *These figures illustrate contour plots of the reliability score of NCFS, ZFS, E^{2}ZFS (xor), and E^{2}ZFS (Fletcher). Darker color means lower score - worse reliability. On each plot, there are four points marked with a "$\times$" representing the four configurations.*

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 III-D. 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.

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 straightforward end-to-end data integrity.

2) Reliability Analysis: The timeline of a data block from writer to reader is shown in Figure 1c. 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

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

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

period causes a SDC; detected corruption would be caught by the checksum verification performed by the reader. The probability of undetected data corruption is:

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

The reliability scores of E^2ZFS (xor) and E^2ZFS (Fletcher) are shown in Figure 2c and Figure 2d. Overall, E^2ZFS (Fletcher) has the best reliability, with all scores above the reliability goal; config 1 even gets a score of 36.6. E^2ZFS (xor) can meet the goal only when both disk and memory are more reliable. Config 4 has a score of 27.8 while both config 2 and 3 have a score of 17.1. Comparing both figures, when the disk corruption dominates (with an index below 12), E^2ZFS (Fletcher) is much better than E^2ZFS (xor), showing that Fletcher is clearly a better checksum for protecting blocks on disk.

3) Performance Issues: E^2ZFS (xor) is less reliable than E^2ZFS (Fletcher), but it offers better performance, especially when the reader is reading data from memory. Table II 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^2ZFS with Fletcher suffers a throughput drop of 15%. In contrast, E^2ZFS (xor) is able to achieve a throughput just 3% less than ZFS, with the checksum-on-copy optimization [15], 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.

C. Z^2FS: ZFS with Flexible End-to-end Data Integrity

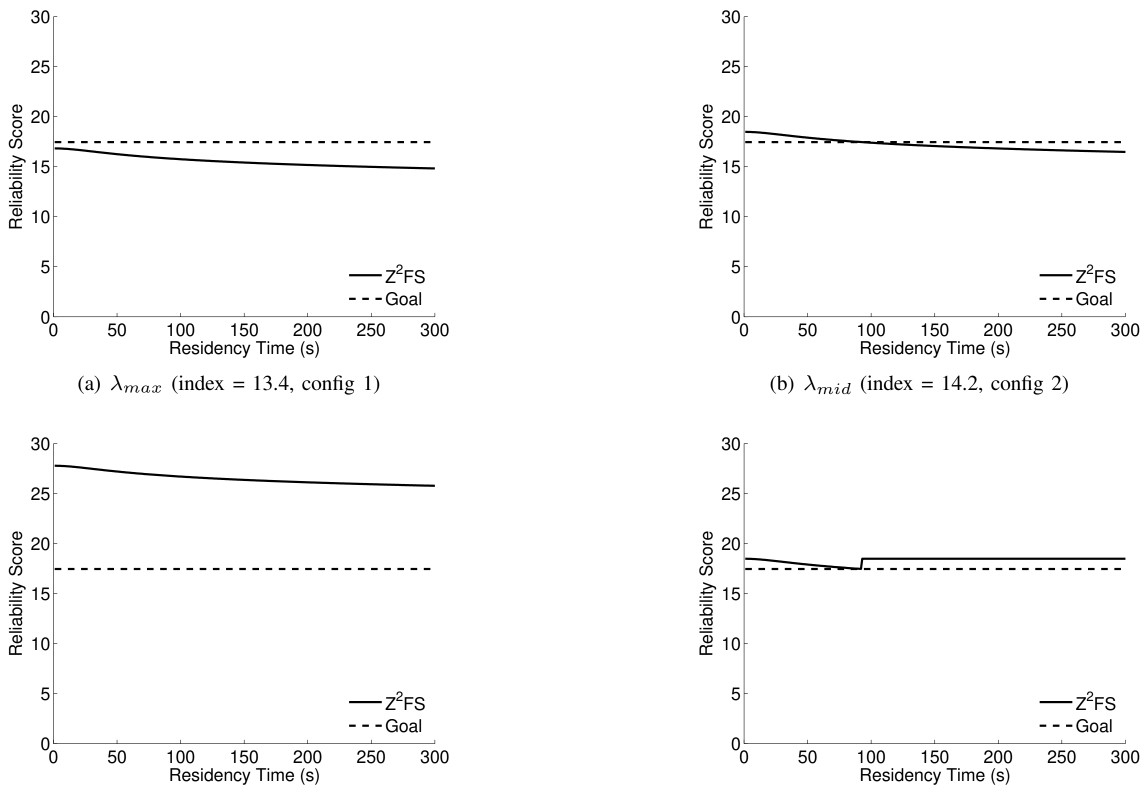
There are two drawbacks with the straight-forward end-to-end approach. Besides the performance problem as shown above, it also suffers from untimely recovery: neither the page cache nor the file system is able to verify the checksum to detect corruption in time. To handle both problems, we build Z^2FS on top of the changes we have made in E^2ZFS 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 III-D.

1) Static Mode with Checksum Chaining: Looking at the reliability score and performance figures of E^2ZFS, a natural question one may ask is: can we combine E^2ZFS (xor) and E^2ZFS (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^2FS, Z^2FS (static), a hybrid of E^2ZFS (xor) and E^2ZFS (Fletcher) that uses xor as the memory checksum and Fletcher as the disk checksum. In static mode, Z^2FS 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.

Z^2FS (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^2FS with checksum chaining is shown in Figure 1d. On the write path, the writer generates an xor checksum at first. When the block is being written to disk, Z^2FS generates a Fletcher checksum using checksum chaining and sends the Fletcher checksum and data to disk. On the read path, Z^2FS 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^2FS has to generate an xor checksum for each data 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^2FS 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^2FS skips the generation of xor checksum on the read path, thus improving the performance. Note that Z^2FS still need to verify the primary checksum (Fletcher) when reading a block from disk.



(a) λ_{max} (index = 13.4, config 1)

(b) λ_{mid} (index = 14.2, config 2)

(c) λ_{min} (index = 18.8, config 3)

(d) λ_{mid} (index = 14.2, config 2)

Fig. 4: **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.*

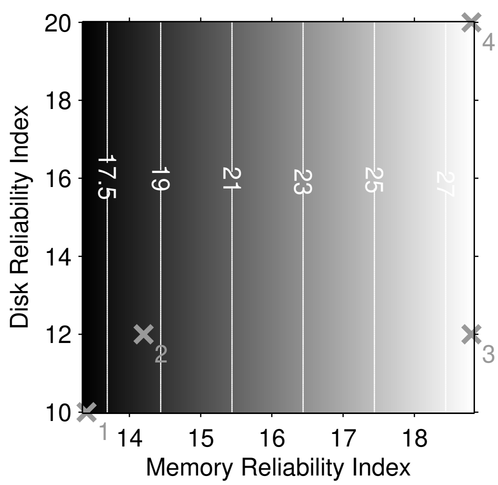


Fig. 3: **Reliability Score ($t_{resident} = 1$) of Z^{2}FS (static).**

2) Reliability Analysis of Static Mode: Figure 1e shows an updated timeline for Z^{2}FS (static) with this optimization. The probability of undetected corruption for Z^{2}FS (static) is:

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

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.

The reliability score of Z^{2}FS (static) at $t_{resident} = 1$ is shown in Figure 3. 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 3 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. Figure 4a, Figure 4b, and Figure 4c illustrate the reliability score of Z^{2}FS (static) versus residency time in all three cases.

In Figure 4c where the memory reliable index is maximum, the reliability 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 4a, 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 4b with a memory reliability 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.

3) 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).

4) Reliability Analysis of Dynamic Mode: Figure 1f 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}.

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

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

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

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

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

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

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

With the Zettabyte reliability goal $P_{goal} = 3.46 \times 10^{-18}$ and λ_{mid}, we have $t_{switch} = 92$. Figure 4d 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.

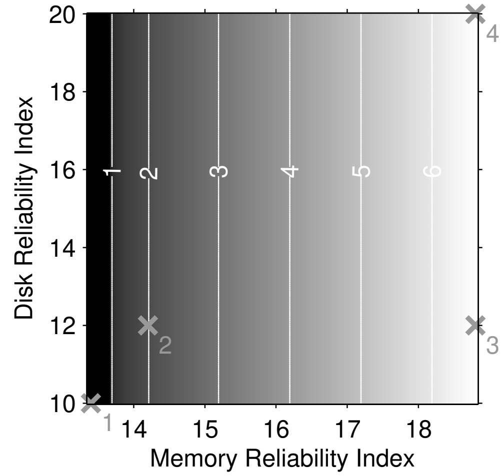


Fig. 5: t_{switch} **of Z^{2}FS (dynamic).** *This figure shows a contour plot of the required switching time to provide Zettabyte reliability in Z^{2}FS (dynamic), with respect to different disk and memory reliability index. The z axis is the base 10 logarithm of t_{switch} in seconds.*

By varying both the disk and memory reliability index, we have Figure 5 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}.

D. Discussion

We now discuss two remaining technical issues: error handling and application integration.

c) 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),

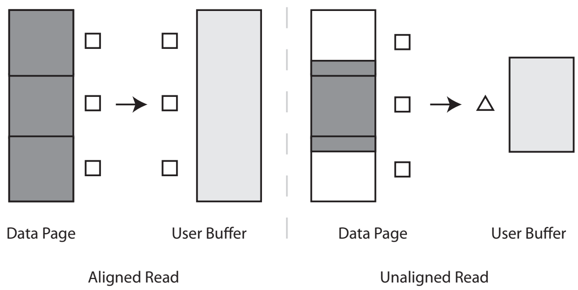


Fig. 6: **Example of Aligned and Unaligned Reads.** *This figure illustrates how Z^2FS handles aligned and unaligned reads. Small squares represent page checksums and small triangles represent user checksums. The dark area represents the requested data.*

	ZFS		**E^2ZFS**		**Z^2FS**	
Timing	**act**	**res**	**act**	**res**	**act**	**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 III: **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.*

E^2ZFS 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 straight-forward end-to-end approach.

As we mentioned in Section III-C.1, to solve the problem, Z^2FS 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^2FS 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^2FS 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^2FS is to use a modified fsync system call. Z^2FS 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.

d) Compatibility with Existing Applications: The compatibility issue mainly comes from the the new interfaces.

First, so far, we have assumed the user buffer is always aligned to page size. In fact, Z^2FS does support generic requests with arbitrary offset and size through checksum chaining. For example, Figure 6 illustrates how Z^2FS handles aligned and generic read requests respectively. In the aligned case, Z^2FS simply returns all three checksums to the application. But when dealing with the unaligned reads, Z^2FS 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: generate the user checksum first and then verify all three page checksums. The same technique is applicable to E^2ZFS, but in common practice, the straight-forward end-to-end approach only supports aligned reads and writes.

Second, the new read/write interfaces may require the application to be aware of the checksums. For applications that really care about data integrity, we believe such changes are necessary. The exposed checksums can be further utilized by applications to protect data at the user level. For other applications that may not want to make changes, both E^2ZFS and Z^2FS provide 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. We should note that currently E^2ZFS and Z^2FS do not support memory-mapped I/O.

IV. Evaluation

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

- How do they handle various data corruption?
- 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.

A. Reliability

The analyses in Section III showed theoretically how Z^2FS can achieve Zettabyte Reliability with different reliability levels of disk and memory. In practice, however, it is difficult to

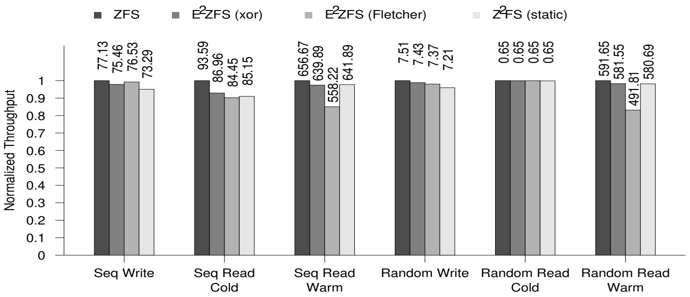


Fig. 7: **Micro Benchmark.** *This graph shows the results of several micro benchmarks on ZFS, E^{2}ZFS, and Z^{2}FS (static). The bars are normalized to the throughput of ZFS. The absolute values in MB/s are shown on top.*

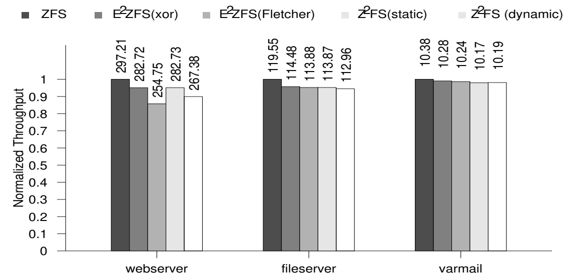


Fig. 8: **Macro Benchmark.** *This figure shows the throughput of our macro benchmarks on ZFS, E^{2}ZFS, Z^{2}FS (static), and Z^{2}FS (dynamic). Each workload runs for 720 seconds. Z^{2}FS (dynamic) has $t_{switch} = 92$ seconds.*

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 1, 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 III 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 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.

B. 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.

e) Micro Benchmark: Figure 7 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 7, 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).

f) Macro Benchmark: We use filebench [40] as our macro benchmark. We choose webserver, fileserver and varmail to evaluate the overall application performance on E^{2}ZFS and Z^{2}FS. Figure 8 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 data in the page cache. Therefore, the throughput is mainly determined by the overhead of checksum calculation. As shown in Figure 8, 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.

C. 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.

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 IV: **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.*

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 V: **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).*

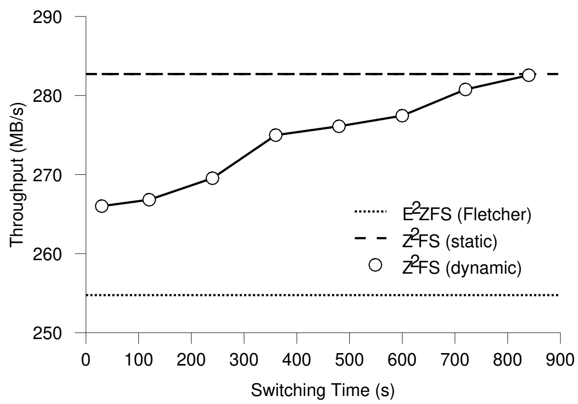


Fig. 9: **Webserver Throughput with Different t_{switch}.** *This figure illustrates the throughput changes of webserver as t_{switch} increases. The dashed line and dotted line represent the throughput of webserver on Z^{2}FS (static) and E^{2}ZFS (Fletcher) respectively. The runtime of the webserver workload is 720 seconds.*

To understand the impact of checksum switching, we run the webserver workload on Z^{2}FS (dynamic) and vary t_{switch}. Figure 9 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).

D. 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 LASR system-call traces [2] 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 IV and the results are shown in Table V. 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.

V. Related Work

A large body of research has been focusing on modeling device-level errors such as memory errors [22] and latent sector errors [29], [35]. There are also many studies on reliability modeling for RAID systems [13], [16], [30]. However, only a few of them cover silent data corruption. Rozier et al. present a fault model for Undetected Disk Errors (UDE) in RAID systems [33]. They build a framework that combines simulation and model to calculate the manifestation rates of undetected data corruption caused by UDEs. Krioukov et al. use model checking to analyze various protection techniques used in current RAID storage systems [21]. They study the interaction between these techniques and find design faults that may lead to data loss or data corruption. In comparison, our work 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.

The protection scheme in the Linux Data Integrity Extension (DIX) [31] and the T10 Protection Information (T10-PI) model [42] (previously known as Data Integrity Field) is very similar to the concept of flexible end-to-end data integrity. 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 [7] 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 [38]. Our work 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 framework also provides a holistic way 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 purely software solution while T10-PI and DIX requires 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^2FS 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^2FS performs checksum switching for data in memory. We believe Z^2FS is the first file system to take data residency time into consideration and provide better protection for data in the page cache. Third, Z^2FS 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^2FS. 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 by Lustre file system [26] in distributed environment and by Oracle's database products [17], [43].

VI. Conclusion

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. It uses different checksum algorithms for different component, and thus can dynamically make tradeoffs between performance and reliability. It also utilizes extra checksum verification below the application to provide in-time detection and recovery. We develop an analytical framework to provide rational behind flexible end-to-end data integrity. 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 an existing file system. Through reliability analysis and various experiments, we show that Z^2FS is able to provide 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.

VII. Acknowledgment

We thank the anonymous reviewers for their tremendous feedback and comments, which have substantially improved the content and presentation of this paper. We thank Christopher Dragga, Tyler Harter, Ao Ma and Thanumalayan Sankaranarayana Pillai for their feedback on the initial draft of this paper. We also thank the other members of the ADSL research group for their insightful comments.

This material is based upon work supported by the National Science Foundation (NSF) under CCF-0811657 and CNS-0834392 as well as generous donations from Google, NetApp and Samsung. 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] Data Integrity. http://indico.cern.ch/getFile.py/access?contribId=3&sessionId=0&resId=1&materialId=paper&confId=13797.
[2] LASR Traces. http://iotta.snia.org/traces/2.
[3] Repeated panics, something gone bad? http://tech.groups.yahoo.com/group/solarisx86/message/38925.
[4] RFC 3385 - Internet Protocol Small Computer System Interface (iSCSI) Cyclic Redundancy Check (CRC)/Checksum Considerations. http://www.ietf.org/rfc/rfc3385.txt.
[5] RFC 3720 - Internet Small Computer Systems Interface (iSCSI). http://www.ietf.org/rfc/rfc3720.txt.
[6] RFC 793 - Transmission Control Protocol. http://www.ietf.org/rfc/rfc793.txt.
[7] RFC1071 - Computing the Internet Checksum. http://www.ietf.org/rfc/rfc1071.txt.
[8] Zfs problem mirror. http://www.mail-archive.com/zfs-discuss@opensolaris.org/msg18079.html.
[9] Zfs problems. http://www.mail-archive.com/zfs-discuss@opensolaris.org/msg04518.html.
[10] L. N. Bairavasundaram, G. R. Goodson, S. Pasupathy, and J. Schindler. An Analysis of Latent Sector Errors in Disk Drives. In *SIGMETRICS '07*, San Diego, CA, June 2007.
[11] 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 '08*, San Jose, CA, February 2008.
[12] J. Bonwick and B. Moore. ZFS: The Last Word in File Systems. http://opensolaris.org/os/community/zfs/docs/zfs_last.pdf.
[13] W. Burkhard and J. Menon. Disk Array Storage System Reliability. In *FTCS-23*, pages 432–441, Toulouse, France, June 1993.
[14] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An Empirical Study of Operating System Errors. In *SOSP '01*, pages 73–88, Banff, Canada, October 2001.
[15] J. Chu and S. Inc. Zero-copy tcp in solaris. In *USENIX ATC'96*, pages 253–264, 1996.
[16] J. Elerath and M. Pecht. Enhanced reliability modeling of raid storage systems. In *DSN'07*, pages 175–184, Edinburgh, UK, June 2007.
[17] EMC. An Integrated End-to-End Data Integrity Solution to Protect Against Silent Data Corruption. www.oracle.com/us/technologies/linux/data-integrity-solution-1852762.pdf.
[18] 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 '01*, pages 57–72, Banff, Canada, October 2001.

[19] A. A. Hwang, I. A. Stefanovici, and B. Schroeder. Cosmic rays don't strike twice: understanding the nature of dram errors and the implications for system design. In *ASPLOS'12*, pages 111–122, London, England, UK, 2012.

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

[21] 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 '08*, pages 127–141, San Jose, CA, February 2008.

[22] X. Li, M. C. Huang, K. Shen, and L. Chu. A realistic evaluation of memory hardware errors and software system susceptibility. In *USENIX ATC'10*, Boston, MA, 2010.

[23] X. Li, K. Shen, M. C. Huang, and L. Chu. A memory soft error measurement on production systems. In *USENIX ATC'07*, 2007.

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

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

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

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

[28] 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.

[29] A. Oprea and A. Juels. A clean-slate look at disk scrubbing. In *FAST'10*, San Jose, CA, 2010.

[30] D. Patterson, G. Gibson, and R. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *SIGMOD '88*, pages 109–116, Chicago, IL, June 1988.

[31] M. K. Petersen. Linux Data Integrity Extensions. In *Linux Symposium*, 2008.

[32] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. IRON File Systems. In *SOSP '05*, pages 206–220, Brighton, UK, October 2005.

[33] E. Rozier, W. Belluomini, V. Deenadhayalan, J. Hafner, K. Rao, and P. Zhou. Evaluating the impact of undetected disk errors in raid systems. In *DSN'09*, Estoril, Lisbon, Portugal, June 2009.

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

[35] B. Schroeder, S. Damouras, and P. Gill. Understanding latent sector errors and how to protect against them. In *FAST'10*, San Jose, CA, 2010.

[36] B. Schroeder, E. Pinheiro, and W.-D. Weber. DRAM errors in the wild: a large-scale field study. In *SIGMETRICS'09*, pages 193–204, 2009.

[37] T. Semiconductor. Soft errors in electronic memory - a white paper. 2004.

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

[39] M. Sullivan and R. Chillarege. Software defects and their impact on system availability-a study of field failures in operating systems. In *FTCS-21*, pages 2–9, June 1991.

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

[41] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the Reliability of Commodity Operating Systems. In *SOSP '03*, Bolton Landing, NY, October 2003.

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

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

[44] J. Yang, C. Sar, and D. Engler. EXPLODE: A Lightweight, General System for Finding Serious Storage System Errors. In *OSDI '06*, Seattle, WA, November 2006.

[45] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using Model Checking to Find Serious File System Errors. In *OSDI '04*, San Francisco, CA, December 2004.

[46] Y. Zhang, A. Rajimwale, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. End-to-end Data Integrity for File Systems: A ZFS Case Study. In *FAST'10*, San Jose, CA, 2010.

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