High Performance Metadata Integrity Protection in the WAFL Copy-on-Write File System

Harendra Kumar; Yuvraj Patel, *University of Wisconsin—Madison;*
Ram Kesavan and Sumith Makam, *NetApp*

https://www.usenix.org/conference/fast17/technical-sessions/presentation/kumar

This paper is included in the Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST '17).

February 27–March 2, 2017 • Santa Clara, CA, USA

ISBN 978-1-931971-36-2

Open access to the Proceedings of the 15th USENIX Conference on File and Storage Technologies is sponsored by USENIX.

High-Performance Metadata Integrity Protection in the WAFL Copy-on-Write File System

Harendra Kumar*
Independent Researcher
harendra.kumar@gmail.com

Yuvraj Patel*
University of Wisconsin–Madison
yuvraj@cs.wisc.edu

Ram Kesavan & Sumith Makam
NetApp
{ram.kesavan, makam.sumith}@gmail.com

Abstract

We introduce a low-cost incremental checksum technique that protects metadata blocks against in-memory scribbles, and a lightweight digest-based transaction auditing mechanism that enforces file system consistency invariants. Compared with previous work, our techniques reduce performance overhead by an order of magnitude. They also help distinguish scribbles from logic bugs. We also present a mechanism to pinpoint the cause of scribbles on production systems. Our techniques have been productized in the NetApp® WAFL® (Write Anywhere File Layout) file system with negligible performance overhead, greatly reducing corruption-related incidents over the past five years, based on millions of runtime hours.

1 Introduction

Storage systems comprise unreliable hardware components such as disks [6, 51, 7], disk shelves, storage interconnect fabric, RAM [52], CPU [45, 56] and data transport buses. This hardware is driven by a software stack or by a dedicated storage operating system that is built around a file system such as ext3 [60], ext4 [42], ZFS [13], btrfs [49], or WAFL [32, 24]. The software is built from heterogeneous components that might also be unreliable because of inherent bugs that could affect other parts of the software ecosystem. Hardware failures and bugs in software can both corrupt data [5, 58, 61]. The file system must provide mechanisms to detect, avoid, and recover from such corruptions [6].

In general, data can be corrupted before it is written to persistent media, while it is residing on the media, or in the read path. Data at rest is protected from media failures by using detection techniques such as checksums [8, 13, 57] and scrubbing [53], and by using recovery techniques such as redundancy [47]. File system crash consistency is provided by techniques such as journaling [30, 59, 40, 50, 12], shadow paging [32, 13, 49], or soft updates [28]. However, memory scribbles that are caused by software bugs [3, 1, 2, 15, 62] or by hardware failures [9, 44, 46, 52, 39, 65], or logic bugs that are in the file system code can still introduce metadata inconsistencies in the write path.

The transaction auditing mechanism Recon [27] is a promising method to improve write integrity. However, with Recon the original version of metadata blocks must be cached and later compared with the modified copies for the audit. In copy-on-write file systems such as WAFL, this requirement can lead to heavy performance regression, especially under metadata-heavy workloads. A highly optimized WAFL implementation that used Recon-like auditing resulted in an unacceptable 30% throughput regression for critical workloads. Furthermore, Recon-like auditing cannot distinguish corruptions that are caused by memory scribbles from those that are caused by logic bugs. For an industrial-scale deployment, the ability to distinguish between the two causes is crucial for fast resolution of corruption bugs.

In this paper, we introduce two novel techniques that in combination can detect arbitrary inconsistencies more efficiently than Recon can. They also provide the crucial ability to distinguish between scribbles and logic bugs. First, we introduce a single rolling checksum through the lifetime of each metadata block – whether in-memory or on persistent media – to protect it against random scribbles. Next, we introduce a digest-based transaction auditing system to prevent logic bugs. Unlike Recon, digest-based auditing does not require caching the original versions of metadata blocks, and is therefore realized with negligible performance overhead. By having separate solutions for scribbles and logic bugs, we can prioritize implementation of auditing invariants based on the return on investment, which is invaluable for a large and complex file system implementation such as WAFL. In

*Research performed while working at NetApp

addition, the file system is doubly protected from scribbles by both techniques. Finally, we also introduce a low-overhead page protection technique that pin-points the root cause of software-caused scribbles, thereby providing quick resolution to corruption bugs.

These solutions have been field-tested across almost a quarter million NetApp ONTAP® customer systems over the past five years, and the data shows that we have attained among the highest levels of data integrity. The overall performance penalty that these data integrity mechanisms incur is almost unnoticeable. They have led to a more than three times reduction in inconsistencies due to software caused corruptions. To the best of our knowledge, we have not encountered a single corruption bug, across our entire customer deployment, that managed to get past the mechanisms that we propose in this paper. Reported corruptions have affected only metadata that were not protected by these mechanisms in the corresponding releases. In addition, by detecting problems early in the development cycle, the solutions have significantly improved engineering productivity.

The primary contributions of this paper are: (1) end-to-end metadata checksums, (2) lightweight digest-based transaction auditing, and (3) fine-grained page protection to pin-point the scribbling code-path.

2 Motivation

NetApp is a storage and data management company that offers software, systems, and services to manage and store data, including its flagship ONTAP operating system [34]. ONTAP implements a proprietary file system called WAFL (Write Anywhere File Layout) [32]. Numerous ONTAP systems are deployed across the world, and various hardware and software bugs have been incurred by them over the past two decades. Some of these incidents have resulted in latent file system inconsistencies that were discovered much later.

When these inconsistencies are discovered, the file system is marked inconsistent and a file system check is initiated. WAFL provides both online [35] and offline [33] file system consistency checks. Offline checking involves significant downtime, which is proportional to the dataset size. Online checking causes less downtime but affects system performance until it has finished. On many occasions, the checks end up discovering data loss, identifying damaged files, or even suggesting recovery from backups.

Although it is extremely rare for a given WAFL file system to end up inconsistent, the sheer number of customer systems that collectively log millions of runtime hours every day make it a more likely occurrence for NetApp's technical support staff. Our main goal was to fortify WAFL against inconsistencies to reduce both disruption for our customers and support costs for NetApp.

2.1 Write Path Metadata Corruptions

Two types of problems can cause file system recovery runs: (1) inconsistencies due to in-memory corruptions of metadata in the write path, and (2) inconsistencies due to the loss of metadata because of media failures that are beyond the redundancy threshold of the underlying RAID mechanism. This paper addresses problem 1. Note that persistent block checksums cannot protect the metadata from in-memory corruptions because the checksum computation occurs after the block is scribbled [64].

Metadata corruption in the write path can result from three primary causes: (1) logic bugs in metadata computation and updates, (2) scribbles of metadata blocks that are used as input for new metadata computation, and (3) scribbles of metadata blocks before they are written to persistent storage.

2.2 Scribbles Versus Logic Bugs

Let us first distinguish between two types of corruption bugs: scribbles and logic bugs.

A scribble (Heisenbug) [29] overwrites an arbitrary data element. It usually occurs randomly and is equally likely to corrupt any data or metadata, producing unpredictable results. Because of its unpredictable nature, a scribble is difficult to reproduce with systematic testing. It is difficult to diagnose because the observed symptoms are far removed from the original location and time of its occurrence. Mistaking a scribble for a logic bug can result in a futile bug chase, and wastage of engineering resources. Often, scribbles are the cause of known but unfixed bugs that adversely affect product quality.

Most hardware scribbles go undetected. ECC memory [16] can detect failures but with some limitation [25], and it might not always be used because of its cost [31, 36]. Software-induced scribbles can be detected using the processor's page protection mechanism, but again with significant costs and limitations.

In contrast, a logic bug (Bohrbug) [29] is inherent to the metadata computation and update logic. A computation with a logic bug generates incorrect outcome but stores it to its own memory location. Therefore, a logic bug is more predictable and limited in its impact. The observed symptoms are predictably confined to a particular behavioral aspect, which makes it easier to diagnose.

2.3 Causes of Corruptions

Scribbles can be caused by software bugs such as buffer overflow, incorrect pointer computation, and dangling pointers in a shared address space [11]. In WAFL, the file system buffer cache memory can be allocated away for other uses and then be recycled back, which makes file system buffers, including metadata buffers, likely victims of software scribbles.

Scribbles are also caused by hardware failures [55] such as memory errors, DMA errors, or CPU register bit flips [10] that remain undetected. Scribble bugs are not that rare [52], and memory scribbles due to hardware failures are expected to be more common in the future [14, 55]. It has been shown that scribbles can be induced by external attacks in a controlled way on a shared infrastructure. [38, 48].

A scribble of an intermediate result or a key data structure can also produce a second order corruption. In our experience, this class of bugs is extremely rare.

Logic bugs are typically found in insufficiently tested code; mature, field-tested code is less likely to have them.

2.4 WAFL File System Overview

Next, we briefly introduce WAFL before we evaluate an existing auditing solution for copy-on-write (COW) file systems. WAFL is a UNIX-style file system with a collection of inodes that represent its files [32]. The file system is written out as a tree of blocks that are rooted at a superblock. Every file system object in WAFL, including metadata, is a file. WAFL is a COW file system, in which every modified block is written to a new location on disk. Only the superblock is ever written in place.

As buffers and inodes are modified (or dirtied) by client operations, they are written out in batches for performance and consistency. Every mutable client operation is also recorded to a log in nonvolatile memory (NVRAM) before it is acknowledged; the operations in the log are replayed to recover data if a crash occurs. WAFL collects the resultant dirty buffers and inodes from hundreds of thousands of logged operations, and uses a checkpoint mechanism called a *consistency point* (CP) to flush them to persistent media as one large transaction. Each CP is an atomic transaction that succeeds only if all of its state is successfully written to persistent storage. Updates to in-memory data structures are isolated and targeted for a specific CP so that each CP represents a consistent and complete state of the file system. When the entire set of new blocks that belong to a CP is persisted, a new file system superblock is atomically written in place that references this new file system tree [32, 24].

2.5 Recon for COW File Systems

Recon is a transaction auditing mechanism that verifies all file system consistency invariants before a transaction is committed. Recon examines physical blocks below the file system, and infers the types of metadata blocks when they are read or written. This allows parsing and interpretation of the blocks, similar to semantically smart disks [54]. However, because of increased metadata overhead per transaction, metadata caching and comparison in Recon's design becomes unsustainable for COW file systems such as WAFL.

In WAFL, each write to a user data block causes the file system to read in the corresponding parent indirect block, to free the old block pointer to this user data block, and to allocate a new block pointer. This process recurses up the tree of blocks that constitute the file system image, all the way up to the super block. Thus, writing to a single user data block might require reading and writing more than one metadata block. Even though WAFL amortizes this overhead by batching numerous operations in a single transaction, the overhead for a Recon-like audit is still significantly high.

Let us analyze the cost of verifying the block accounting metadata in WAFL. When a block pointer is inserted into or deleted from an indirect block, WAFL sets or clears the corresponding bit in its bitmaps. A Recon-like audit compares the persistent versions of indirect blocks with the modified versions that are being committed to generate a list of block pointers that are allocated or freed in the current transaction. This list is then tallied with the corresponding changes in bitmap bits, which are obtained by comparing the persistent and modified versions of the corresponding bitmap blocks. To enable this comparison, a copy of the persistent version of each metadata block is cached in memory before the block is modified.

The cost of a Recon-like audit primarily includes: (1) making a copy of the indirect block before modification, (2) making copies of the bitmap blocks to record the allocated and freed blocks, (3) comparing the unmodified and modified versions of the indirect blocks and the bitmap blocks, and (4) verifying that the changes in the bitmap blocks are consistent with the changes in the indirect blocks. This process involves significant CPU cost and memory bandwidth, and if the metadata blocks are to be read from persistent media, it also involves significant I/O cost.

The memory requirement of a Recon-like audit scales up with the number of modified metadata blocks that are in the transaction. Caching the pristine versions of modified metadata blocks becomes impractical. Many of those blocks might be evicted from the cache and therefore must be read in from persistent storage just in time for

the audit, and the additional I/O directly affects the cost of auditing. Moreover, the increased memory pressure can force us to trigger transactions early, which results in smaller batch sizes and increased metadata overhead, in turn reducing the overall write throughput.

The audit of a consistency invariant cannot begin until the corresponding metadata are finalized. Self-referential metadata, such as bitmaps, are not finalized until the very end of a CP [37], and only then can they be audited. This serialization reduces the I/O efficiency of the CP, which in turn negatively affects the achievable write throughput of the file system.

2.6 Drawbacks of Recon

Performance: The designs presented in this paper were conceptualized before the publication of Recon, and coincidentally, we first tried the same metadata caching approach. A highly optimized prototype of Recon-style auditing for bitmap and indirect block cross consistency checks showed a performance degradation of approximately 30% on our internal database/OLTP benchmark; the benchmark is described in Section 7. The authors of the original Recon paper observed a similar performance penalty in their research prototype implementation for btrfs [22].

Diagnostics: Although transaction auditing detects the presence of an inconsistency, it cannot distinguish between a scribble and a logic bug. When the inconsistency is due to a scribble, the audit cannot identify the affected metadata block; it can only point to the set of blocks that were used to verify the invariant that failed its check.

Implementation challenge: Because scribbles can affect any metadata without bias, a foolproof transaction auditing needs consistency invariants that cover the entire metadata. Building such an exhaustive set of invariants is quite expensive in terms of both system performance and engineering resources, and it has a low return on investment for mature, field-tested code with no logic bugs.

3 Approach

3.1 Goals

For ONTAP, performance and field diagnostics are both as important as data integrity.

Performance: A 30% throughput regression to protect against a relatively rare though disruptive event is considered too high a price by the business. To be competitive, we need a solution with excellent metadata integrity protection that costs an order of magnitude less in terms of performance.

Diagnostics: Corruptions in general and scribbles in particular, are the hardest to diagnose. For faster diagnosis, we need the ability to distinguish scribbles from logic bugs. As explained in section 2.2, they are different in nature and require different techniques to determine the root cause. The diagnostic capability is even more important during product development when the likelihood of scribbles is higher and many person-hours are wasted on corruption bugs. Worse, the product sometimes ships with known, but unreproducible bugs that remain undiagnosed.

3.2 Solution Overview

In-memory metadata checksums: To distinguish scribbles from logic bugs and to identify the scribbled block, we use a general checksum protection for in-memory metadata blocks. Before a block is written to persistent storage, a checksum verification determines whether the block was scribbled.

Metadata page protection: Even though checksum verification prevents scribbles from being persisted, it cannot implicate the culprit code. We use processor assisted page granular protection to catch the culprit code. Because this approach has a higher performance tax, we recommend enabling it on a customer's system only when a metadata block checksum failure is reproducible.

Transaction auditing: Consistency invariants are verified by comparing digests of changes that occurred in a transaction. It is very efficient to create and compare digests. Changes to a metadata block are added to a digest as and when it is modified. Thus, we do not need cached copies of the original blocks.

By having a separate solution for scribble detection, we have the flexibility to prioritize the implementation of auditing invariants that yield higher return on investment. We can start with code-paths that are prone to logic bugs, such as code-paths with higher complexity or code-paths that correspond to newer, untested features.

4 Scribble Protection

In this section, we describe a high-performance and multiprocessor-capable incremental checksum scheme to protect in-memory metadata against scribbles. We also present some limitations, after which we present the page protection mechanism that overcomes the most important limitation.

The goal is to detect any unauthorized change to metadata blocks that are written out in a transaction. Because we are trying to protect against scribblers that share the same address space, we cannot use address space iso-

lation as a protection technique. The proposed page-granular protection in Section 4.7 is quite expensive, and therefore is used only in diagnostic mode. Furthermore, none of these techniques can detect a corruption that is caused by hardware problems.

Checksumming is a well-known data integrity protection mechanism. Block checksums are widely used to protect the integrity of the blocks of the file system that reside on persistent storage. Z^{2}FS [63] even proposes using a single checksum as well as a checksum handover scheme to protect the data while in transit across heterogeneous software components.

However, none of the known schemes protect the data while it resides in memory. In this section, we describe a scheme that can be used effectively to protect in-memory metadata on a large scale, and with negligible performance overhead.

4.1 End-to-End Checksum

We use a single rolling or incremental checksum [4, 41] to protect each metadata block through its entire life cycle, whether it is in memory or on persistent storage. When a block is read in from persistent storage, the stored checksum of the block is also read and then is verified by the file system. While in memory, this verified checksum is used to protect the block and is updated incrementally by every legal update of the block.

When a block is written out to persistent storage as part of a transaction, a fresh checksum is computed by the file system. This freshly computed checksum is used to verify the incrementally updated in-memory checksum, and if the verification fails, we know that the block has been scribbled. That way, the block remains protected whether it is in memory or on persistent storage.

Note that the file system already pays for the cost of checksum computation during each block read and write. The only additional work that is needed is the incremental update of that checksum whenever the block is modified in memory.

4.2 Checksum Initialization

A metadata block that is written out by a transaction is either newly created or is an existing block that was read from persistent storage earlier but was modified during the transaction. We initialize the in-memory checksum for a block as soon as it comes into existence in memory. If newly created, the checksum is initialized to a derivable constant value based on the zero-state of the corresponding metadata. If read in from storage, the checksum is initialized to the stored and verified checksum.

4.3 Incremental Checksum Updates

Legal modification of a metadata block in memory is typically accomplished by invoking well-defined APIs in the file system code. We add a hook in each of these APIs so that the corresponding block checksum is kept up to date on modifications. Before a block is modified, the new checksum is computed incrementally by using the old checksum, the original data that is being overwritten, the position of the original data in the block, and the new data. In the next few paragraphs, we show precisely how to incrementally recompute an Adler [23] checksum when a single byte in a block is modified.

In the following equations, "div" and "mod" represent operators that return a quotient and a remainder, respectively, in an integer division. Assuming that a data block D is composed of n bytes $D_1, D_2, ..., D_n$, the *Adler32* checksum of D is computed as follows:

$$A = (1 + D_1 + D_2 + ... + D_n) \bmod 65521$$
$$B = (n \times D_1 + (n-1) \times D_2 + ... + D_n + n) \bmod 65521$$
$$Adler32(D) = A + B \times 65536$$

If the original checksum of block D is C, and we replace byte D_i in the block with a new byte D'_i, then the new checksum *Adler32Incr* can be computed as follows:

$$A = C \bmod 65536$$
$$B = C \text{ div } 65536$$
$$\Delta D_i = D'_i - D_i$$
$$A' = (A + \Delta D_i) \bmod 65521$$
$$B' = (B + (n + 1 - i) \times \Delta D_i) \bmod 65521$$
$$Adler32Incr(C, D_i, D'_i, i) = A' + B' \times 65536$$

Similarly, we can also recompute the new checksum incrementally when multiple contiguous bytes are modified in the block.

If any portion of a block is scribbled, its incremental checksum becomes inconsistent with respect to its contents, and it remains so even after any number of subsequent legal updates to the block. Thus, when the file system eventually recomputes the full checksum (before it writes the block to persistent storage) by using the current contents of the block, the full checksum does not match the incremental checksum.

The incremental checksum update is optimal because it requires additional memory accesses only to read the original contents, to read the old checksum, and to update the checksum. In most cases, the file system code reads the metadata before overwriting it anyway; therefore, the original contents are likely to be in the processor cache. The CPU cost of computing the new checksum is proportional to the amount of data that is modified. In terms of the memory overhead, this approach requires an additional 4 bytes per data block to maintain the rolling checksum.

If preferred, a stronger 64-bit checksum can be used. Because they are amenable to this form of incremental computation, Adler [23], Fletcher [26], or any other position-dependent checksum can be used. For better performance, a chunk size larger than a byte can be used in the checksum computation. For WAFL, we used a modified version and a highly optimized implementation of the simple incremental checksum computation shown previously.

4.4 Lockless Multiprocessor Updates

WAFL is designed to run simultaneously on several processors [21]. Thus, it is quite common for a metadata block to be modified concurrently by multiple processors. However, to minimize cache-line thrashing and expensive lock contention, the WAFL multiprocessor programming model [21] avoids using spinlocks as much as possible.

To avoid lock contention, an incremental checksum for a block can be split across processors and be updated in a lockless manner. Each processor computes and accumulates the delta checksums for its own updates to the block in a per-processor checksum field. The per-processor deltas are then combined to derive the final incremental checksum.

Two independent checksum fragments, C_1 and C_2, can be combined as follows:

$A = (C_1 \bmod 65536 + C_2 \bmod 65536) \bmod 65521$
$B = (C_1 \operatorname{div} 65536 + C_2 \operatorname{div} 65536) \bmod 65521$
$combine(C_1, C_2) = A + B \times 65536$

Two processors that modify bytes D_i and D_j simultaneously can maintain their respective per-processor checksum fragments, C_1 and C_2. Each fragment is updated independently using *Adler32Incr* from the previous section. Before verification, the fragments are combined with the original checksum of the block, C, to arrive at the final checksum C':

$C_1 = C_2 = 0$
$C'_1 = Adler32Incr(C_1, D_i, D'_i, i)$
$C'_2 = Adler32Incr(C_2, D_j, D'_j, j)$
$C' = combine(C, combine(C'_1, C'_2))$

However, each per-processor checksum fragment requires additional memory (4 bytes) per block. This extra memory is quite a worthwhile trade-off because it saves us the cost of acquiring and contending on spinlocks.

4.5 Checksum Verification

When a block is written out to persistent storage as part of a transaction, we recompute the checksum on the entire block and compare it with the incremental checksum. If the two checksums do not match, we have detected a scribble on the block.

This verification can be performed either before or after the write I/O for the block is issued to storage. In the former case, checksum computation can be optimized by combining it with RAID parity computation [17]. However, this approach opens a window for undetectable scribbles after the checksum computation but before the write I/O is completed. Verification of the checksum after the completion of the write I/O closes that window. In any case, corruptions that are injected by the data transfer fabric while it services the write I/O cannot be detected until the block is read again.

Upon detection of a scribble, we abort the ongoing transaction commit, preventing the corruption from being persisted. To protect the ONTAP node from any other potential corruptions from the same bug, we reboot the node instead of aborting an individual transaction. Because ONTAP is configured in high-availability pairs, the partner node takes ownership of the rebooted node's file systems, and those file systems are all still consistent because they are defined by their most recently completed transaction. The partner node then replays the user operations that are recorded in the NVRAM log, and the prior and consistent metadata is read from storage, is modified, and then is committed as part of a brand-new transaction.

4.6 Assumptions and Limitations

Incremental checksumming is robust against bugs in the checksum update code. If we miss adding the incremental checksum hook to any of the legal APIs that modify metadata, then any transaction that includes a call to the API results in a checksum verification failure, thereby forcing us to fix the bug.

This mechanism helps distinguish scribbles from logic bugs, but it does not implicate the culprit code-path that tampered with the memory of the corrupted block.

The checksum may not be strong enough to detect all kinds of corruptions. Adler and Fletcher checksums are known to be stronger against larger errors than against smaller ones [43]. They work quite well for storage media failures because the corruption size is usually bigger; however, some in-memory corruptions can be smaller, and therefore the chances that they go undetected are somewhat higher. In the unlikely event that a scribble remains undetected by the checksum, it will certainly be detected by the transaction auditing (described in the next section). However, such a corruption might, unfortunately, be attributed to a logic bug.

If a bug causes an incorrect block pointer to be supplied as an argument to a legal metadata update API, the API

will modify the wrong block and its corresponding incremental checksum. This bug will not be detected by checksum verification, but will be detected by the transaction auditing mechanism. We have not come across this sort of bug in our experience with WAFL. One potential solution is to use different signatures to categorize the callers of the APIs into different groups; for example, each group could rotate its incremental checksum by a specific number of bits. That way an updater from an incorrect group would result in an incorrect checksum.

The incremental checksum technique can also be used to protect other important data structures that participate in any periodic transactional episode.

4.7 Diagnostics Using Page Protection

A checksum verification failure indicates that the block has been scribbled, but it does not implicate the culprit code-path because the scribble might have occurred long ago. Regular address space-based protection cannot help if the culprit code-path shares the same address space; this is true of much of the kernel code in ONTAP.

If a scribble is reproducible, we provide an option to enable metadata page protection to directly implicate the code-path. To provide good performance even under heavy and frequent modification of metadata, we use a combination of page-level protection and the Write-Protect Enable (WP) bit in the x86-64 processors [18].

To protect metadata blocks from scribbles we keep the individual metadata pages read-only by default. One way to allow safe updates would be to mark the corresponding page read-write just before a legal updater modifies the block. When the requested modification has finished, the page can be marked read-only again. Thus, an illegal modification finds the page read-only, generates a page fault, and yields a stack trace that points to the scribbler.

However, this scheme does not perform well if metadata is modified frequently. The frequent switching of page permissions (read-only to read/write to read-only) not only causes a flood of TLB flushes, but also creates a storm of inter-processor TLB invalidation interrupts. Using this scheme to protect all indirect blocks and bitmap blocks degraded performance by approximately 70%, in our experiments, thereby rendering the scheme unusable even in debug mode.

To reduce the performance impact, we keep the pages read-only all the time. To enable legal writes, we disable protection globally (by flipping the WP bit) before modification and we re-enable it after modification. With this approach, the performance degradation comes down to about 20%, which is acceptable for a diagnostic mode.

When write protection is disabled on a CPU core, it can write to any address. This ability implies two risks: (1) the metadata modification code itself might scribble, or (2) interrupt handlers that are serviced by the processor during that window might scribble otherwise read-only memory. However, these risks are close to zero because these code blocks (the metadata update APIs and interrupt handlers) are typically small pieces of code and are extremely well tested. Furthermore, we perform address range checks on the target address before we disable protection.

As Section 7.3 shows, this feature has proved to be invaluable for product development and has been used in the field as well. Moreover, as explained in Section 6, it has been an invaluable tool for quickly identifying code-paths that required our incremental checksum hooks

5 Transaction Auditing

In this section, we introduce a digest-based transaction auditing technique, explain what invariants it checks by using an example, breakdown its performance, and analyze its limitations. The audit verifies that the changes to the file system state being committed in a given transaction are self-consistent. There are two categories of consistency invariants: local and distributed:

Local consistency invariant: A local consistency invariant is confined to a given metadata block. For example, all block pointers in an indirect block must be within the file system block number range. Such an invariant is inexpensive to check because it does not require loading any other blocks.

Distributed consistency invariant: A distributed invariant defines a consistency relationship across several blocks from different metadata. For example, when a block pointer is cleared from an indirect block, the corresponding bit in the bitmap block must be cleared. A distributed consistency invariant is expensive to check because it requires identifying the changes to several blocks from different metadata.

In contrast to Recon, we intercept modifications to metadata at the file system layer. During a transaction, changes are accumulated to create digests that are used later to verify consistency invariants. This design obviates the need to cache the unmodified metadata blocks. Unlike Recon, the cost of recording the changes is proportional to the actual changes rather than to the number of modified blocks.

5.1 Digest-Based Auditing

Digest-based audits help verify distributed consistency invariants inexpensively. The key idea is to create digests

of changes and to cross-verify the digests rather than individual changes. This strategy drastically reduces the amount of work that is required in verification and still provides strong enough guarantees to be useful in practice.

To illustrate the mechanism, let us use cross consistency between indirect blocks and bitmap blocks as an example. We intercept indirect and bitmap block modification operations by using the aforementioned hook in the modification APIs, and we create a digest of those modifications.

Let us say that an indirect block contains 64-bit block numbers $B_1, B_2, ..., B_n$. Suppose block numbers B_i, B_j, and B_k were replaced in a transaction by newly allocated block numbers N_i, N_j, and N_k. A hook in the API that modifies the indirect block updates checksums of the original block numbers in a *free digest* and the new block numbers in an *allocated digest*; each digest is maintained per transaction across all updates to the metadata of the file system:

$\Sigma IndFree = B_i + B_j + B_k + ...$
$\Sigma IndAlloc = N_i + N_j + N_k + ...$

Similarly, a hook in the bitmap block modification API maintains per transaction digests of all block numbers whose corresponding bits flipped from 0 to 1 (new allocations) and vice-versa (frees):

$\Sigma BitmapFree = B_i + B_j + B_k + ...$
$\Sigma BitmapAlloc = N_i + N_j + N_k + ...$

At the end of the transaction, the audit verifies that the two sets of digests agree with each other:

$\Sigma IndFree == \Sigma BitmapFree$
$\Sigma IndAlloc == \Sigma BitmapAlloc$

Similarly, several other distributed consistency invariants can be inexpensively verified by using the digest scheme; Section 5.4 describes them. Audit digest verification failure is handled in the same way as checksum verification failure, as explained in Section 4.5.

5.2 Audit Performance

A digest-based audit performs much better than a Recon-like audit because digests are inexpensive to compute and metadata blocks need not be cached and compared. Specifically: (1) the data that we are adding to the digest is readily available in the processor's cache because it has just been accessed; (2) digest update involves just one addition operation and one memory access for each metadata update; (3) there are no expensive I/O operations due to buffer cache misses; and (4) the digest is created incrementally with each metadata update operation which eliminates the need for an exclusive phase in which we identify and verify all changes individually.

Note that digest update is performed together with the block's incremental checksum update by using the same hook. Thus, we efficiently use one memory access to do three things: (1) modify the indirect block, (2) update the audit digest, and (3) update the incremental checksum. The final verification is an inexpensive comparison of a few bytes.

A digest-based audit of bitmap and indirect block changes drastically reduced the overall cost of transaction auditing from 30% throughput regression to less than 2% on our database/OLTP benchmark.

5.3 Strengths and Weaknesses

The audit is provably robust with respect to bugs in the digest update code. If we miss adding the hook to update the digest in any of the legal APIs that are used to modify metadata, the corresponding digest verification fails.

A digest is typically a simple sum without any position-related information because we compare sets and not sequences. In theory, logic bugs can result in just the "right" pattern of incorrect updates that a digest-based verification cannot detect. In the previous example, it is possible for the two summations to match even if the actual updates were incorrect. Over the past five years of this feature's existence, such a case has never been hit in internal development or in the field. File systems have been corrupted only because the audit infrastructure of the corresponding ONTAP release did not include the invariant for a particular consistency property.

5.4 List of Distributed Invariants

In addition to the invariant explained in Section 5.1, we check many other distributed invariants of the WAFL file system as part of the audit. Most of them are inexpensive and are enabled by default in production systems. A few of them are somewhat expensive and might be disabled by default in production on some specific low-end configurations with insufficient CPU horsepower.

Table 1 shows a subset of the distributed invariants that we have implemented. We do not present other invariants that are very specific to the persistent layout of the WAFL file system, and that require more background to explain. Note that all invariants were not implemented in one go; rather, they were implemented in phases across several releases based on the return on investment.

	Description of Equation
1	Each inode tracks a count of all blocks to which it points. The file system also maintains a total count of all the allocated blocks. Their deltas much match.
2	The bitmap uses a bit to track the allocated state of each block in the file system [37]. The file system also maintains a total count of all allocated blocks. The delta of the latter must equal the delta of the number of bits that flipped to 1 (i.e., allocated) minus the number of bits that flipped to 0 (i.e., free).
3	The inode metadata tracks the allocation status of each inode. The file system maintains a total count of all allocated inodes. The delta of the latter must equal the number of inodes that changed state from free to allocated minus the number of inodes that changed state from allocated to free.
4	The inode metadata tracks deleted inodes that are moved to a hidden namespace awaiting block reclamation [37]. The current file system maintains a total count of these hidden inodes. The delta of the latter must equal the number of inodes that were deleted (i.e., moved into the hidden namespace) minus the number of inodes that were removed from the hidden namespace after all their blocks had been reclaimed.
5	The *refcount* file maintains an integer count to track all extra references to each block; WAFL uses this file to support de-duplication. The file system maintains the physical space that is saved by de-duplication as a count of blocks. Their deltas much match.
6	Each inode tracks a count of physical blocks that are saved by compression. The file system maintains the physical space that is saved by compression as a count of blocks. Their deltas must match.

Table 1: Some important audit equations implemented in WAFL

6 Implementation

Intercepting modifications: All three features – incremental checksum, page granular protection, and transaction auditing – must intercept all modifications to a protected metadata block. We inserted a unified hook into every legal API that is used to update metadata. The API supplies all the requisite parameters to the hook to update the incremental checksum, to update the corresponding digests, and to toggle memory protection.

Almost all modifications to metadata blocks go through well-known WAFL APIs, so it was easy to insert our hooks inside those APIs. However, given more than two decades' worth of code growth and churn, there were a few hidden and somewhat obscure places in the code that updated metadata blocks directly; we used the page protection feature to find them. All metadata pages are read-only by default, and the hook is needed to toggle that mode. When the feature was turned on, any update of metadata from an obscure code-path immediately generated a page fault with a useful stack trace, which enabled us to modularize the code-path and insert the hook.

Special optimizations: Two common modifications to the metadata of a file system are: (1) the update of block pointers in an indirect block, and (2) flipping of bits in the bitmaps to indicate the allocated or freed status of blocks. We wrote custom, optimized checksum computation routines for those cases, i.e., the modification of a fixed-length block pointer (64 bits in the case of WAFL) and the modification of a single bitmap bit. For other updates, we created a generalized incremental checksum computation routine that is based on the offset (in the block) and on the length of the update so that it can handle variable length modification. An example use of such generalized computation is file deletion processing, which requires bulk updates of metadata blocks.

Complexity: Intercepting each modification to a metadata block was a bit intrusive, but it made the code more modular and fostered better development practices. On one hand, our implementation is more complex compared with Recon because we intercept each modification to a metadata block. But on the other hand it is simpler because we do not have to implement a cache and therefore avoid the many problems associated with caching.

7 Evaluation

In this section, we evaluate the performance and the benefits of the various mechanisms that we presented earlier.

We used an in-house workload generator that emulates random reads and writes to model the query and update operations of a database/OLTP application. It was built to be very similar to the Storage Performance Council Benchmark-1 (SPC-1) [20]. The Standard Performance Evaluation Corporation home-directory style benchmark (SPEC SFS) [19] was also used, but those results are not presented here because our protection mechanisms showed negligible overhead. The heavy random overwrites that the database/OLTP benchmark produces put a much higher stress on our mechanisms because more metadata is modified per transaction.

7.1 Incremental Checksum Performance

As explained earlier in Section 4.3 this mechanism uses the per-block persistent checksum that is already enforced and paid for by the WAFL I/O stack. The only additional work required is to compute the checksum incrementally when the metadata is modified in memory. The raw CPU cycles for computing and updating the checksum are negligible and likely are partially absorbed by idle cycles between processor pipeline stages. The memory overhead is 4 bytes per-processor (for lockless concurrent updates) for every 4KB metadata block, which is less than 0.1% of the total buffer cache memory. Furthermore, checksum computation and update does not directly affect the latency of a user operation because almost all metadata updates occur asynchronously after the user operation has been completed and acknowledged.

Figure 1(a) and Figure 1(b), respectively, show this cost on: (a) a midrange system with 12 Intel® Westmere cores, 96GB DRAM and 4GB NVRAM; and (b) a low-end system with 4 Intel Wolfdale cores, 20GB DRAM, and 2GB NVRAM. Sufficient numbers of SAS hard disks were attached to both systems to eliminate any storage bottleneck. Experiments were run by using our database/OLTP benchmark with incremental checksums turned on and turned off, and the observed latency was plotted against the achieved IOPS throughput with an increasing input IOPS load. We plotted only the load points with latencies that were less than or equal to 30ms, which is the maximum that the SPC-1 benchmark allows.

We see an increase in latency in the range of -0.9ms to 1.7ms and zero to 0.8ms on the midrange and low-end systems, respectively; these ranges translate to -3.5% to 10.5% and zero to 17%, respectively. Note that at very low latencies, a small increase in latency translates into a large percentage of change even though the absolute change is of little practical consequence. If we look at the achieved throughput at any given latency on the midrange system, we see from zero to a maximum of 1% regression. On the low-end system, the throughput regression varies from 5% (at 5ms) to zero (at 24ms).

File deletion is another workload that generates heavy metadata updates, and thereby stresses the incremental checksum mechanism. To process files that are pending deletion [37], WAFL must asynchronously walk several indirect blocks, clear block pointers in them, and update the relevant metadata in large quantities. Unfortunately, SPEC SFS generates an insufficient file deletion load. Similarly, the *SCSI_UNMAP* operation also causes heavy metadata updates, but is not generated in sufficient numbers by benchmarks such as SPC-1. Hence, we fashioned a custom benchmark that creates numerous very large files (a few terabytes' worth of space),

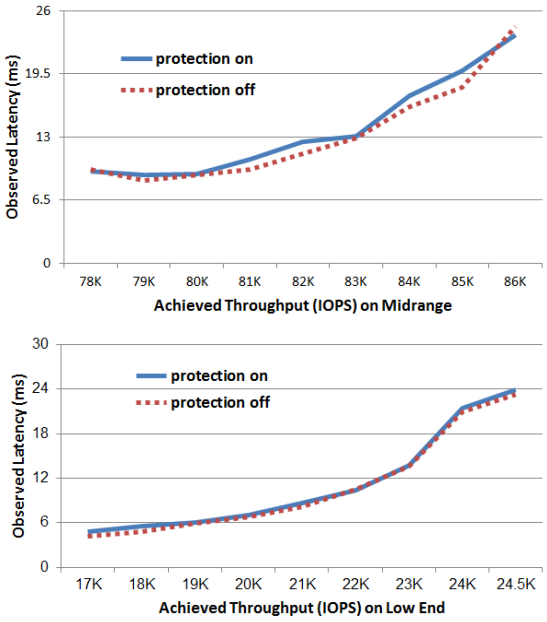

Figure 1: Latency versus throughput with and without incremental checksum protection with a database/OLTP workload on: (a) a midrange system with 12 cores, 96GB DRAM, and 4GB NVRAM; and (b) a low-end system with 4 cores, 20GB DRAM, and 2GB NVRAM.

then deletes them all and measures the delete rate that the system achieves. Although the achievable delete rate was pretty much the same (and there was no actual client latency to measure), we computed the overhead of incremental checksum protection as a function of the number of blocks freed. To compute this overhead, we added up cycles that were spent by the functions that update these pointers and the associated metadata. The benchmark was run on the previously mentioned midrange and low-end systems. On the midrange system, the WAFL CPU cost for freeing each block was computed to be around 2.16μs and 2.33μs with incremental checksums turned off and on, respectively. The corresponding numbers on the low-end system were 2.26μs and 2.43μs, respectively. This represents a 7.8% and 7.5% overhead, respectively. To put things in perspective, the CPU cycles that were spent across all WAFL code-paths during that interval were 3μs to 4μs per block freed. Therefore, this overhead is 2.5% as a fraction of the total WAFL CPU cycles.

7.2 Full Protection Performance

In this section, we measure the performance overhead of both auditing and incremental checksum working together. As explained earlier in Section 5.2 the CPU cost of maintaining audit digests is negligible, and the cost is minimized further by combining it with incremental

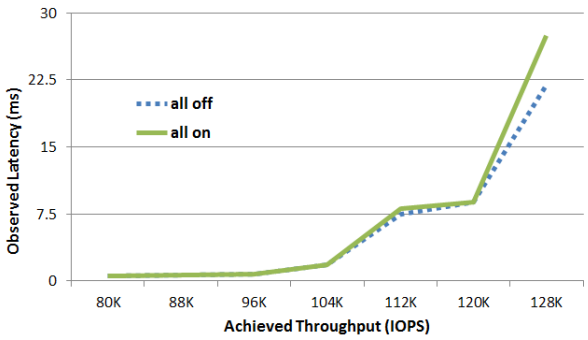

Figure 2: Latency versus throughput with and without all protection (audit and incremental checksum) with a database/OLTP workload on a high-end all-flash system with 20 cores, 128GB DRAM, and 8GB NVRAM.

checksum updates. The memory overhead is tiny as it requires just a few bytes of digest per auditing invariant.

Figure 2 studies the effect of protection mechanisms with the database/OLTP benchmark on a high-end all-flash system with 20 Intel Ivy Bridge cores, and 128GB of DRAM, and several shelves' worth of solid-state drives. Experiments were run with all protection turned off and on. The protection-on case includes incremental checksum and auditing with the 20+ audit equations that we have implemented till date. The observed latency was plotted against the achieved IOPS throughput with an increasing input IOPS load. We see that at up to 120,000 IOPS, there is absolutely no impact on latencies. At high loads of 128,000 IOPS and above, we see a rapid latency increase of about 5.5ms, or about 25%. Note that most customers choose to run their high-performance database/OLTP-style loads either at or below the 2ms latency mark, where we see absolutely zero impact on latencies.

We should point out that compared to the midrange system used in Figure 1 the system used in Figure 2 is a high-end system with CPU cycles to spare, next-generation Intel chipsets, larger processor caches, more DRAM, and solid-state drives.

7.3 Scribble Diagnostics

Page protection is turned off by default in the shipped product; it can be turned on when required for diagnostics. If a corruption bug is repeatedly hit on a system, the other two protection mechanisms prevent it from being written to persistent storage. However, the repeated file system restarts can be disruptive for the customer. Because the rogue code-path or thread has already run to completion by the time the scribble is detected, it typically takes about one person-month of senior developer time to find the root cause of an average memory scribble bug. However, if the customer is willing to incur a 20% performance penalty by turning on page protection, the root cause presents itself in the resultant core dump that the page fault generates.

In the past five years, and over millions of hours of total run-time across hundreds of thousands of deployed ONTAP systems, page protection has been needed only once. A customer system hit an incremental checksum protection panic every few hours, and even though the root cause had been narrowed down to some suspect code-paths, it had not been found. The customer turned on page protection, and the resultant core file helped find the bug, which was a buffer overflow in a rarely hit code-path.

During product development, page protection is turned on by default. It has proved to be invaluable not only in identifying corruption bugs early, but also in reducing bugs of unknown origin that might have resurfaced later in the field. Section 7.5 provides relevant statistics.

7.4 Ability to Detect Bugs

WAFL has a built-in command-line tool that injects corruptions into in-memory data structures. Scripts invoke this command with arguments that specify the file system ID, inode number, indirect block level, offset, and length, together with the corruption pattern. The tool loads the data structure into memory (if it isn't already present) and injects the corruption. Several test plans were built so that injected corruptions would be caught by the protection mechanisms before the superblock of the subsequent transaction commit was written out. In some cases, the corruption pattern flipped specific bits in the block bitmap such that the resultant checksum remained unchanged and it could slip through the incremental checksum protection. However, those cases were always caught by the audit.

7.5 Benefits

These mechanisms have proved invaluable in protecting customer systems from becoming corrupted. First, we looked at corruption bugs that hit customer systems over a four-year period before the protection mechanisms were in place. For each bug, we tracked the time from when the development team started looking at it to when the root cause was discovered, and we looked at the number and the expertise of the people involved in fixing the bugs. On average, it took about one person-month of very senior developer time to find the root cause of each bug.

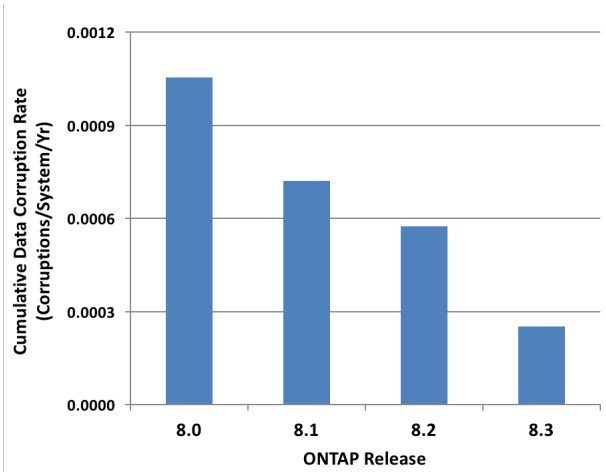

Figure 3: Number of times that inconsistencies made it to persistent storage on customer systems, normalized to total system run-time-hours across four different releases. Repair and recovery procedures were required. Because this data adds up different systems hitting the same bug, the number of actual bugs will be lower.

Since our protection mechanisms have been in place, most of the corruption bugs in WAFL, ONTAP, and hardware drivers have been caught well before customer deployment. Over a five-year period, a total of 75 memory scribble bugs have been found by page protection when testing with debug kernels, 32 memory scribble bugs were found in non-debug kernels by the incremental checksum mechanism, and 23 logic bugs were found by the auditing mechanism. We cannot overstate the value of these results in terms of organizational productivity.

Of course, as is true with any large system, some corruption bugs do manage to escape in-house testing. The first auditing equations shipped with NetApp Data ONTAP® 8.1 in September 2011; incremental checksum, page protection, and more auditing equations shipped with Data ONTAP 8.2; and even more equations shipped with Data ONTAP 8.3. In the past five years, the incremental checksum mechanism has protected customer systems from 8 unique memory scribble bugs 33 times, and the auditing mechanism has protected against 9 unique logic bugs 50 times. In total, that's 83 times that a customer system was saved from running expensive file system recovery procedures and from potential data loss.

Figure 3 shows the number of times that inconsistencies have made it to the persistent file system on customer systems, normalized to total system run-time-hours across the past four releases over a one-year period. The normalization by run-time-hours was necessary because, during that one-year period, the system hours that were recorded on Data ONTAP 8.1 and 8.2 were much higher than for the other two releases. Most systems had upgraded from the older 8.0 release, and not many systems had upgraded to 8.3 yet. Therefore, the raw data was biased toward the 8.1 and 8.2 releases. We see release-to-release improvements of 34%, 21%, and 44%, respectively. The total improvement amounts to a more than 3 times reduction in the rate of occurrence of inconsistencies. Some auxiliary WAFL metadata is not yet covered by the protection mechanisms. Therefore, we sometimes get benign inconsistencies in the persistent file system (not real data corruptions but inconsistencies nevertheless). We expect to see further reduction in inconsistencies once we fix these gaps.

8 Conclusion

We introduced two techniques, incremental checksum and digest-based auditing, that prevent in-memory scribbles and logic bugs from corrupting persistent file system metadata. We disproved the commonly held belief that strong data integrity requires a high performance penalty; we achieved integrity with a negligible performance tax. We distinguished scribbles from logic bugs, and also provided diagnostic capabilities to pinpoint the culprit for software scribbles.

These techniques have greatly improved data integrity in WAFL, resulting in an unprecedented reduction in recovery runs. By catching corruptions early in the development cycle, these techniques have enabled our engineers to innovate rapidly without risking data integrity.

We believe that end-to-end incremental checksumming can be applied to user data blocks, thereby providing round-trip application-level protection at a low cost. This technique can be especially useful in protecting applications that are hosted on third-party infrastructure, in which the reliability of hardware cannot be established or guaranteed. Moreover, continuous checksum protection can harden applications against induced corruption attacks on shared cloud infrastructure. Databases and file systems that are hosted on fabric-attached or cloud storage are good examples of potential benefiters of such end-to-end protection.

9 Acknowledgments

We thank the many WAFL developers who have contributed to the work presented in this paper: Ananthan Subramanian, Santosh Venugopal, Tijin George, Ganga Kondapalli, Mihir Gorecha, Varada Kari, Vishnu Vardhan, and Santhosh Paul. We also thank Remzi H. Arpaci-Dusseau, our reviewers, and our shepherd, Jiri Schindler, for their invaluable feedback.

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] Computation of the internet checksum via incremental update. https://tools.ietf.org/html/rfc1624, 1994.

[5] Dave Anderson, Jim Dykes, and Erik Riedel. More Than an Interface: SCSI vs. ATA. In *FAST*, 2003.

[6] Lakshmi N. Bairavasundaram, Garth R. Goodson, Shankar Pasupathy, and Jiri Schindler. An Analysis of Latent Sector Errors in Disk Drives. In *SIGMETRICS*, 2007.

[7] Lakshmi N. Bairavasundaram, Garth R. Goodson, Bianca Schroeder, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. An Analysis of Data Corruption in the Storage Stack. In *FAST*, 2008.

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

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

[10] M. P. Baze and S. P. Buchner. Attenuation of single event induced pulses in cmos combinational logic. *IEEE Transactions on Nuclear Science*, 44(6):2217–2223, Dec 1997.

[11] Emery D. Berger and Benjamin G. Zorn. Diehard: Probabilistic memory safety for unsafe languages. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 158–168, New York, NY, USA, 2006. ACM.

[12] Steve Best. JFS Overview. www.ibm.com/developerworks/library/l-jfs.html, 2000.

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

[14] Wayne Burleson, Onur Mutlu, and Mohit Tiwari. Invited - who is the major threat to tomorrow's security?: You, the hardware designer. In *Proceedings of the 53rd Annual Design Automation Conference*, DAC '16, pages 145:1–145:5, New York, NY, USA, 2016. ACM.

[15] John Chapin, Mendel Rosenblum, Scott Devine, Tirthankar Lahiri, Dan Teodosiu, and Anoop Gupta. Hive: Fault Containment for Shared-Memory Multiprocessors. In *SOSP*, 1995.

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

[17] Peter Corbett, Bob English, Atul Goel, Tomislav Grcanac Steven Kleiman, James Leong, and Sunitha Sankar. Row-diagonal parity for double disk failure correction. In *Proceedings of Conference on File and Storage Technologies (FAST)*, 2004.

[18] Intel Corporation. Intel 64 and IA-32 Architectures Software Developers Manual Volume 3A: System Programming Guide, Part 1. https://software.intel.com/sites/default/files/managed/7c/f1/253668-sdm-vol-3a.pdf, December 2016.

[19] Standard Performance Evaluation Corporation. Spec sfs 2014. https://www.spec.org/sfs2014/.

[20] Storage Performance Council. Storage performance council-1 benchmark. www.storageperformance.org/results/#spc1_overview.

[21] Matthew Curtis-Maury, Vinay Devadas, Vania Fang, and Aditya Kulkarni. To waffinity and beyond: A scalable architecture for incremental parallelization of file system code. In *Proceeding of Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[22] Fryer Daniel. Personal Communication. Date - 09/07/2016.

[23] P. Deutsch and J.-L. Gailly. Zlib compressed data format specification version 3.3. https://www.ietf.org/rfc/rfc1950.txt, 1996.

[24] John K. Edwards, Daniel Ellard, Craig Everhart, Robert Fair, Eric Hamilton, Andy Kahn, Arkady Kanevsky, James Lentini, Ashish Prakash, Keith A. Smith, and Edward Zayas. FlexVol: flexible, efficient file volume virtualization in WAFL. In *Proceedings of the 2008 USENIX Annual Technical Conference*, pages 129–142, Jun 2008.

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

[26] J. Fletcher. An arithmetic checksum for serial transmissions. *IEEE Transactions on Communications*, 1982.

[27] Daniel Fryer, Kuei Sun, Rahat Mahmood, TingHao Cheng, Shaun Benjamin, Ashvin Goel, and Angela Demke Brown. Recon: Verifying file system consistency at runtime. In *FAST '12*, San Jose, CA, February 2012.

[28] Gregory R. Ganger and Yale N. Patt. Metadata Up-

date Performance in File Systems. In *OSDI '94*, 1994.

[29] Jim Gray. Why do computers stop and what can be done about it? http://www.hpl.hp.com/techreports/tandem/TR-85.7.pdf, 1985.

[30] Robert Hagmann. Reimplementing the Cedar File System Using Logging and Group Commit. In *SOSP '87*, 1987.

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

[32] Dave Hitz, James Lau, and Michael Malcolm. File system design for an NFS file server appliance. In *Proceedings of USENIX Winter 1994 Technical Conference*, pages 235–246, Jan 1994.

[33] NetApp Inc. Netapp quick notes - wafl check. http://netapp-notes.blogspot.com/2008/03/wafl-check.html, 2008.

[34] NetApp Inc. Data ONTAP 8. http://www.netapp.com/us/products/platform-os/ontap/, 2010.

[35] NetApp Inc. Overview of wafliron. https://kb.netapp.com/support/index?page=content\&id=3011877, 2016.

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

[37] Ram Kesavan, Rohit Singh, Travis Grusecki, and Yuvraj Patel. Algorithms and data structures for efficient free space reclamation in WAFL. In *Proceedings of Conference on File and Storage Technologies (FAST)*, 2017.

[38] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 361–372, Piscataway, NJ, USA, 2014. IEEE Press.

[39] Xin Li, Kai Shen, Michael C. Huang, and Lingkun Chu. A memory soft error measurement on production systems. In *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, ATC'07, pages 21:1–21:6, Berkeley, CA, USA, 2007. USENIX Association.

[40] Joshua MacDonald, Hans Reiser, and Alex Zarochentcev. Reiser4 Transaction Design Document. https://reiser4.wiki.kernel.org/index.php/Txn-doc, 2002.

[41] T. Mallory and A. Kullberg. Incremental updating of the internet checksum. https://tools.ietf.org/html/rfc1141, 1990.

[42] Avantika Mathur, Mingming Cao, and Andreas Dilger. ext4: the next generation of the ext3 file system. *;login: - The Usenix Magazine*, Vol. 31, June 2006.

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

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

[45] Thomas R. Nicely. Pentium fdiv bug. http://www.trnicely.net/pentbug/bugmail1.html.

[46] 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.

[47] David Patterson, Garth Gibson, and Randy Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *SIGMOD*, 1988.

[48] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. Flip feng shui: Hammering a needle in the software stack. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 1–18, Austin, TX, August 2016. USENIX Association.

[49] Ohad Rodeh, Josef Bacik, and Chris Mason. Btrfs: The linux b-tree filesystem. *Trans. Storage*, 9(3):9:1–9:32, August 2013.

[50] Mendel Rosenblum and John Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1), 1992.

[51] Bianca Schroeder and Garth Gibson. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you? In *FAST*, 2007.

[52] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. DRAM errors in the wild: A Large-Scale Field Study. In *Proceedings of the 2009 Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS/Performance '09)*, Seattle, Washington, June 2007.

[53] Thomas J. E. Schwarz, Qin Xin, Ethan L. Miller, Darrell D. E. Long, Andy Hospodor, and Spencer Ng. Disk scrubbing in large archival storage systems. In *Proceedings of the The IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, MAS-

COTS '04, pages 409–418, Washington, DC, USA, 2004. IEEE Computer Society.

[54] Muthian Sivathanu, Vijayan Prabhakaran, Florentina I. Popovici, Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Semantically-smart disk systems. In *Proceedings of the 2Nd USENIX Conference on File and Storage Technologies*, FAST '03, pages 73–88, Berkeley, CA, USA, 2003. USENIX Association.

[55] Vilas Sridharan, Nathan DeBardeleben, Sean Blanchard, Kurt B. Ferreira, Jon Stearley, John Shalf, and Sudhanva Gurumurthi. Memory errors in modern systems: The good, the bad, and the ugly. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 297–310, New York, NY, USA, 2015. ACM.

[56] Inquirer Staff. Amd opteron bug can cause incorrect results. `http://www.theinquirer.net`.

[57] Christopher A. Stein, John H. Howard, and Margo I. Seltzer. Unifying file system protection. In *Proceedings of the General Track: 2001 USENIX Annual Technical Conference*, pages 79–90, Berkeley, CA, USA, 2001. USENIX Association.

[58] The Data Clinic. Hard Disk Failure. `http://www.dataclinic.co.uk/hard-disk-failures.htm`, 2004.

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

[60] Stephen C. Tweedie. EXT3, Journaling File System. `olstrans.sourceforge.net/release/OLS2000-ext3/OLS2000-ext3.html`, 2000.

[61] Glenn Weinberg. The Solaris Dynamic File System. `http://members.visi.net/$\sim$thedave/sun/DynFS.pdf`, 2004.

[62] Yichen Xie, Andy Chou, and Dawson Engler. Archer: Using symbolic, path-sensitive analysis to detect memory access errors. In *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-11, pages 327–336, New York, NY, USA, 2003. ACM.

[63] Yupu Zhang, Daniel Myers, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Zettabyte Reliability with Flexible End-to-end Data Integrity. In *MSST'13*, Long Beach, CA, May 2013.

[64] Yupu Zhang, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. End-to-end data integrity for file systems: A zfs case study. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, FAST'10, Berkeley, CA, USA, 2010. USENIX Association.

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