# Evaluating File System Reliability on Solid State Drives

Shehbaz Jaffer, Stathis Maneas, Andy Hwang, and Bianca Schroeder, *University of Toronto*

https://www.usenix.org/conference/atc19/presentation/jaffer

This paper is included in the Proceedings of the
2019 USENIX Annual Technical Conference.

July 10–12, 2019 • Renton, WA, USA

ISBN 978-1-939133-03-8

# Evaluating File System Reliability on Solid State Drives

Shehbaz Jaffer[*]
*University of Toronto*

Stathis Maneas[*]
*University of Toronto*

Andy Hwang
*University of Toronto*

Bianca Schroeder
*University of Toronto*

## Abstract

As solid state drives (SSDs) are increasingly replacing hard disk drives, the reliability of storage systems depends on the failure modes of SSDs and the ability of the file system layered on top to handle these failure modes. While the classical paper on IRON File Systems provides a thorough study of the failure policies of three file systems common at the time, we argue that 13 years later it is time to revisit file system reliability with SSDs and their reliability characteristics in mind, based on modern file systems that incorporate journaling, copy-on-write and log-structured approaches, and are optimized for flash. This paper presents a detailed study, spanning ext4, Btrfs and F2FS, and covering a number of different SSD error modes. We develop our own fault injection framework and explore over a thousand error cases. Our results indicate that 16% of these cases result in a file system that cannot be mounted or even repaired by its system checker. We also identify the key file system metadata structures that can cause such failures and finally, we recommend some design guidelines for file systems that are deployed on top of SSDs.

## 1 Introduction

Solid state drives (SSDs) are increasingly replacing hard disk drives as a form of secondary storage medium. With their growing adoption, storage reliability now depends on the reliability of these new devices as well as the ability of the file system above them to handle errors these devices might generate (including for example device errors when reading or writing a block, or silently corrupted data). While the classical paper by Prabhakaran et *al.* [45] (published in 2005) studied in great detail the robustness of three file systems that were common at the time in the face of hard disk drive (HDD) errors, we argue that there are multiple reasons why it is time to revisit this work.

The first reason is that failure characteristics of SSDs differ significantly from those of HDDs. For example, recent field studies [39, 43, 48] show that, while their replacement rates (due to suspected hardware problems) are often by an order of magnitude lower than those of HDDs, the occurrence of partial drive failures that lead to errors when reading or writing a block or corrupted data can be an order of magnitude higher. Other work argues that the Flash Translation Layer (FTL) of SSDs might be more prone to bugs compared to HDD firmware, due to their high complexity and less maturity, and demonstrate this to be the case when drives are faced with power faults [53]. This makes it even more important than before that file systems can detect and deal with device faults effectively.

Second, file systems have evolved significantly since [45] was published 13 years ago; the ext family of file systems has undergone major changes from the ext3 version considered in [45] to the current ext4 [38]. New players with advanced file-system features have arrived. Most notably Btrfs [46], a copy-on-write file system which is more suitable for SSDs with no in-place writes, has garnered wide adoption. The design of Btrfs is particularly interesting as it has fewer total writes than ext4's journaling mechanism. Further, there are new file systems that have been designed specifically for flash, such as F2FS [33], which follow a log-structured approach to optimize performance on flash.

The goal of this paper is to characterize the resilience of modern file systems running on flash-based SSDs in the face of SSD faults, along with the effectiveness of their recovery mechanisms when taking SSD failure characteristics into account. We focus on three different file systems: Btrfs, ext4, and F2FS. ext4 is an obvious choice, as it is the most commonly used Linux file system. Btrfs and F2FS include features particularly attractive with respect to flash, with F2FS being tailored for flash. Moreover, these three file systems cover three different points in the design spectrum, ranging from journaling to copy-on-write to log-structured approaches.

The main contribution of this paper is a detailed study, spanning three very different file systems and their ability to detect and recover from SSD faults, based on error injection targeting all key data structures. We observe huge differences across file systems and describe the vulnerabilities of each in detail.

[*]These authors contributed equally to this work.

Over the course of this work we experiment with more than one thousand fault scenarios and observe that around 16% of them result in severe failure cases (kernel panic, unmountable file system). We make a number of observations and file several bug reports, some of which have already resulted in patches. For our experiments, we developed an error injection module on top of the Linux device mapper framework.

The remainder of this paper is organized as follows: Section 2 provides a taxonomy of SSD faults and a description of the experimental setup we use to emulate these faults and test the reaction of the three file systems. Section 3 presents the results from our fault emulation experiments. Section 4 covers related work and finally, in Section 5, we summarize our observations and insights.

## 2 File System Error Injection

Our goal is to emulate different types of SSD failures and check the ability of different file systems to detect and recover from them, based on which part of the file system was affected. We limit our analysis to a local file system running on top of a single drive. Note that although multi-drive redundancy mechanisms like RAID exist, they are not general substitutes for file system reliability mechanisms. First, RAID is not applicable to all scenarios, such as single drives on personal computers. Second, errors or data corruption can originate from higher levels in the storage stack, which RAID can neither detect nor recover.

Furthermore, our work only considers *partial* drive failures, where only part of a drive's operation is affected, rather than *fail-stop* failures, where the drive as a whole becomes permanently inaccessible. The reason lies in the numerous studies published over the last few years, using either lab experiments or field data, which have identified many different SSD internal error mechanisms that can result in partial failures, including mechanisms that originate both from the flash level [10, 12, 13, 16–19, 21, 23, 26, 27, 29–31, 34, 35, 40, 41, 47, 49, 50] and from bugs in the FTL code, e.g. when it is not hardened to handle power faults correctly [52, 53].

Moreover, a field study based on Google's data centers observes that partial failures are significantly more common for SSDs than for HDDs [48].

This section describes different SSD error modes and how they manifest at the file system level, and also our experimental setup, including the error injection framework and how we target different parts of a file system.

### 2.1 SSD Errors in the Field and their Manifestation

This section provides an overview over the various mechanisms that can lead to partial failures and how they manifest at the file system level (all summarized in Table 1).

*Uncorrectable Bit Corruption:* Previous work [10, 12, 13, 16–19, 21, 26, 27, 29, 34, 35, 41] describes a large number of error mechanisms that originate at the flash level and can result in *bit corruption*, including retention errors, read and program disturb errors, errors due to flash cell wear-out and failing blocks. Virtually all modern SSDs incorporate error correcting codes to detect and correct such bit corruption. However, recent field studies indicate that *uncorrectable bit corruption*, where more bits are corrupted than the error correcting code (ECC) can handle, occurs at a significant rate in the field. For example, a study based on Google field data observes 2-6 out of 1000 drive days with uncorrectable bit errors [48]. Uncorrectable bit corruption manifests as a read I/O error returned by the drive when an application tries to access the affected data ("Read I/O errors" in Table 1).

*Silent Bit Corruption:* This is a more insidious form of bit corruption, where the drive itself is not aware of the corruption and returns corrupted data to the application ("Corruption" in Table 1). While there have been field studies on the prevalence of silent data corruption for HDD based systems [9], there is to date no field data on silent bit corruption for SSD based systems. However, work based on lab experiments shows that 3 out of 15 drive models under test experience silent data corruption in the case of power faults [53]. Note that there are other mechanisms that can lead to silent data corruption, including mechanisms that originate at higher levels in the storage stack, above the SSD device level.

*FTL Metadata Corruption:* A special case arises when silent bit corruption affects FTL metadata. Among other things, the FTL maintains a mapping of logical to physical (L2P) blocks as part of its metadata [8]; metadata corruption could lead to "Read I/O errors" or "Write I/O errors", when the application attempts to read or write a page that does not have an entry in the L2P mapping due to corruption. Corruption of the L2P mapping could also result in wrong or erased data being returned on a read, manifesting as "Corruption" to the file system. Note that this is also a silent corruption - i.e. neither the device nor the FTL is aware of these corruptions.

*Misdirected Writes:* This refers to the situation where during an SSD-internal write operation, the correct data is being written to flash, but at the wrong location. This might be due to a bug in the FTL code or triggered by a power fault, as explained in [53]. At the file system level this might manifest as a "Corruption", where a subsequent read returns wrong data, or a "Read I/O error". This form of corruption is silent; the device does not detect and propagate errors to the storage stack above until invalid data or metadata is accessed again.

*Shorn Writes:* A shorn write is a write that is issued by the file system, but only partially done by the device. In [53], the authors observe such scenarios surprisingly frequently during power faults, even for enterprise class drives, while issuing properly synchronized I/O and cache flush commands to the device. A shorn write is similar to a "torn write", where only part of a multi-sector update is written to the disk, but it applies to sector(s) which should have been fully persisted due to the use of a cache flush operation. One possible expla-

| **SSD/Flash Errors** | (a) | (b) | (c) | (d) | (e) |
|---|---|---|---|---|---|
| Uncorrectable Bit Corruption | ✓ | | | | |
| Silent Bit Corruption | | | ✓ | | |
| FTL Metadata Corruption | ✓ | ✓ | ✓ | | |
| Misdirected Writes | ✓ | | ✓ | | |
| Shorn Writes | | | | ✓ | |
| Dropped Write | ✓ | ✓ | ✓ | | ✓ |
| Incomplete Program Operation | | | ✓ | ✓ | |
| Incomplete Erase Operation | | | ✓ | | |

Table 1: *Different types of flash errors and their manifestation in the file system. (a) Read I/O error (b) Write I/O error (c) Corruption (d) Shorn Write (e) Lost Write.*

nation is the mismatch of write granularities between layers. The default block size for file systems is larger (e.g. 4KB for ext4/F2FS, and 16KB for Btrfs) than the physical device (e.g. 512B). A block issued from the file system is mapped to multiple physical blocks inside the device. As a result, during a power fault, only some of the mappings are updated while others remain unchanged. Even if physical block sizes match that of the file system, another possible explanation is because SSDs include on-board cache memory for buffering writes, shorn writes may also be caused by alignment and timing bugs in the drive's cache management [53]. Moreover, recent SSD architectures use pre-buffering and striping across independent parallel units, which do not guarantee atomicity between them for an atomic write operation [11]. The increase in parallelism may further expose more shorn writes.

At the file system level, a shorn write is not detected until its manifestation during a later read operation, where the file system sees a 4KB block, part of which contains data from the most recent update to the block, while the remaining part contains either old or zeroed out data (if the block was recently erased). While this could be viewed as a special form of silent bit corruption, we consider this as a separate category in terms of how it manifests at the file system level (called "Shorn Write" corresponding to column (d) in Table 1) as this form of corruption creates a particular pattern (each sequence of 512 bytes within a 4KB block is either completely corrupted or completely correct), compared to the more random corruption event referred to by column (c).

In [53], the authors observe shorn writes manifesting in two patterns, where only the first 3/8th or the first 7/8th of a block gets written and the rest is not. Similarly in our experiments, we keep only the first 3/8th of a 4KB block. We assume the block has been successfully erased, so the rest of the block remains zeroed out. Our module can be configured to test other shorn write sizes and patterns as well.

*Dropped writes:* The authors in [53] observe cases where an SSD internal write operation gets dropped even after an explicit cache flush (e.g. in the case of a power fault when the update was in the SSD's cache, but not persisted to flash). If the dropped write relates to FTL metadata, in particular to the L2P mapping, this could manifest as a "Read I/O error", "Write I/O error" or "Corruption" on a subsequent read or write of the data. If the dropped write relates to a file system write, the result is the same as if the file system had never issued the corresponding write. We create a separate category for this manifestation which we refer to as "Lost Write" (column (e) in Table 1).

*Incomplete Program operation:* This refers to the situation where a flash program operation does not fully complete (without the FTL noticing), so only part of a flash page gets written. Such scenarios were observed, for example, under power faults [53]. At the file system level, this manifests as a "Corruption" during a subsequent read of the data.

*Incomplete Erase operation:* This refers to the situation where a flash erase operation does not completely erase a flash erase block (without the FTL detecting and correcting this problem). Incomplete erase operations have been observed under power faults [53]. They could also occur when flash erase blocks wear-out and the FTL does not handle a failed erase operation properly. Subsequent program operations to the affected erase block can result in incorrectly written data and consequently "Corruption", when this data is later read by the file system.

## 2.2 Comparison with HDD faults

We note that there are also HDD-specific faults that would manifest in a similar way at the file system level. However, the mechanisms that cause faults within each media are different and can for example affect the frequency of observed errors. One such case are uncorrectable read errors which have been observed at a much higher frequency in production systems using SSDs than HDDs [48] (a trend that will likely only get worse with QLC). There are faults though whose manifestation does actually differ from HDDs to SSDs, due to inherent differences in their overall design and operation. For instance, a part affected by a shorn write may contain previously written data in the case of an HDD block, but would contain zeroed out data if that area within the SSD has been correctly erased. In addition, the large degree of parallelism inside SSDs makes correctness under power faults significantly more challenging than for HDDs (for example, ensuring atomic writes across parallel units). Finally, file systems might modify their behavior and apply different fault recovery mechanisms for SSDs and HDDs; for example, Btrfs turns off metadata duplication by default when deployed on top of an SSD.

## 2.3 Device Mapper Tool for Error Emulation

The key observation from the previous section is that all SSD faults we consider manifest in one of five ways, corresponding to the five columns (a) to (e) in Table 1. This section describes a device mapper tool we created to emulate all five scenarios.

In order to to emulate SSD error modes and observe each individual file system's response, we need to intercept the block I/O requests between the file system and the block device. We leverage the Linux *device mapper* framework to create a virtual block device that intercepts requests between the file system and the underlying physical device. This allows

| Programs |
|---|
| mount, umount, open, creat, access, stat, lstat, chmod, chown, utime, rename, read, write, truncate, readlink, symlink, unlink, chdir, rmdir, mkdir, getdirentries, chroot |

Table 2: *The programs used in our study. Each one stresses a single system call and is invoked several times under different file system images to increase coverage.*

us to operate on block I/O requests and simulate faults as if they originate from a physical device, and also observe the file system's reaction without modifying its source code. In this way, we can perform tracing, parse file system metadata, and alter block contents online, for both read and write requests, while the file system is mounted. For this study, we use the Linux kernel version 4.17.

Our module can intercept read and write requests for selected blocks as they pass through the block layer and return an error code to the file system, emulating categories (a) "Read Error" and (b) "Write Error" in Table 1. Possible parameters include the request's type (read/write), block number, and data structure type. In the case of multiple accesses to the same block, one particular access can be targeted. We also support corruption of specific data structures, fields and bytes within blocks, allowing us to emulate category (c) "Corruption". The module can selectively shear multiple sectors of a block before sending it to the file system or writing it on disk, emulating category (d) "Shorn Write". Our module can further drop one or more blocks while writing the blocks corresponding to a file system operation, emulating the last category (e) "Lost Write". The module's API is generic across file systems and can be expanded to different file systems. Our module can be found at [6].

## 2.4 Test Programs

We perform injection experiments while executing test programs chosen to exercise different parts of the POSIX API, similar to the "singlets" used by Prabhakaran et *al.* [45]. Each individual program focuses on one system call, such as `mkdir` or `write`. Table 2 lists all the test programs that we used in our study. For each test program, we populate the disk with different files and directory structures to increase code coverage. For example, we generate small files that are stored inline within an inode, as well as large files that use indirect blocks. All our programs pedantically follow POSIX semantics; they call `fsync(2)` and `close(2)`, and check the return values to ensure that data and metadata has successfully persisted to the underlying storage device.

## 2.5 Targeted Error Injection

Our goal is to understand the effect of block I/O errors and corruption in detail depending on which part of a file system is affected. That means our error injection testbed requires the ability to target specific data structures and specific fields within a data structure for error injection, rather than randomly injecting errors. We therefore need to identify for each program which data structures are involved and how the parts of the data structure map to the sequence of block accesses generated by the program.

| ext4 | |
|---|---|
| **Data Structure** | **Approach** |
| *super block, group descriptor, inode blocks, block bmap, inode bmap* | `dumpe2fs` |
| *dir_entry* | `debugfs`, get block inode, stat on inode number, check file type |
| *extent* | `debugfs`, check for extent of a file or directory path |
| *data* | `debugfs`, get block inode, stat on inode number, check file type |
| *journal* | `debugfs`, check if parent inode number is 8 |
| **Btrfs** | |
| **Data Structure** | **Approach** |
| *fstree, roottree, csumtree, extentTree, chunkTree, uuidTree, devTree, logTree* | device mapper module check btrfs node header fields at runtime |
| `DIR_ITEM` `DIR_INDEX` `INODE_REF` `INODE_DATA` `EXTENT_DATA` | `btrfs-debug-tree` |
| **F2FS** | |
| *superblock, checkpoint, SIT, NAT, inode, d/ind node, dir. block, data* | device mapper module |

Table 3: *The approach to type blocks collected using either* `blktrace` *or our own device mapper module.*

Understanding the relationship between the sequence of block accesses and the data structures within each file system required a significant amount of work and we had to rely on a combination of approaches. First, we initialize the file system to a clean state with representative data. We then run a specific test program (Table 2) on the file system image, capturing traces from `blktrace` and the kernel to learn the program's actual accessed blocks. Reading the file system source code also enables us to put logic inside our module to interpret blocks as requests pass through it. Lastly, we use offline tools such as `dumpe2fs`, `btrfs-inspect`, and `dump.f2fs` to inspect changes to disk contents. Through these multiple techniques, we can identify block types and specific data structures within the blocks. Table 3 summarizes our approach to identify different data structures in each of the file systems.

After identifying all the relevant data structures for each program, we re-initialize the disk image and repeat test program execution for error injection experiments. We use the same tools, along with our module, to inject errors to specific targets. A single block I/O error or data corruption is injected into a block or data structure during each execution. This allows us to achieve better isolation and characterization of the file system's reaction to the injected error.

Our error injection experiments allow us to measure both immediate and longer-term effects of device faults. We can observe immediate effects on program execution for some cases, such as user space errors or kernel panics (e.g. from write I/O errors). At the end of each test program execution, we unmount the file system and perform several offline tests to verify the consistency of the disk image, regardless of whether the corruption was silent or not (e.g. persisting lost/shorn writes): we invoke the file system's integrity checker (*fsck*), check if the file system is mountable, and check whether

| Symbol | Level | Description |
|---|---|---|
| ○ | $\mathbf{D_{Zero}}$ | No detection. |
| – | $\mathbf{D_{ErrorCode}}$ | Check the error code returned from the lower levels. |
| \ | $\mathbf{D_{Sanity}}$ | Check for invalid values within the contents of a block. |
| / | $\mathbf{D_{Redundancy}}$ | Checksums, replicas, or any other form of redundancy. |
| \| | $\mathbf{D_{Fsck}}$ | Detect error using the system checker. |
| ○ | $\mathbf{R_{Zero}}$ | No attempt to recover. |
| / | $\mathbf{R_{Retry}}$ | Retry the operation first before returning an error. |
| \| | $\mathbf{R_{Propagate}}$ | Error code propagated to the user space. |
| \ | $\mathbf{R_{Previous}}$ | File system resumes operation from the state exactly before the operation occurred. |
| – | $\mathbf{R_{Stop}}$ | The operation is terminated (either gracefully or abruptly); the file system may be mounted as read-only. |
| 🟥 | $\mathbf{R_{Fsck\_Fail}}$ | Recovery failed, the file system cannot be mounted. |
| 🟧 | $\mathbf{R_{Fsck\_Partial}}$ | The file system is mountable, but it has experienced data loss in addition to operation failure. |
| 🟨 | $\mathbf{R_{Fsck\_Orig}}$ | Current operation fails, file system restored to pre-operation state. |
| 🟩 | $\mathbf{R_{Fsck\_Full}}$ | The file system is fully repaired and its state is the same with the one generated by the execution where the operation succeeded without any errors. |

Table 4: *The levels of our detection and recovery taxonomy.*

the program's operations have been successfully persisted by comparing the resultant disk image against the expected one. We also explore longer-term effects of faults where the test programs access data that were previously persisted with errors (read I/O, reading corrupted or shorn write data).

In this study, we use *btrfs-progs v4.4*, *e2fsprogs v1.42.13*, and *f2fs-tools v1.10.0* for our error injection experiments.

### 2.6 Detection and Recovery Taxonomy

We report the *detection* and *recovery* policies of all three file systems with respect to the data structures involved. We characterize each file system's reaction via all observable interfaces: system call return values, changes to the disk image, log messages, and any side-effects to the system (such as kernel panics). We classify the file system's detection and recovery based on a taxonomy that was inspired by previous work [45], but with some new extensions: unlike [45], we also experiment with file system integrity checkers and their ability to detect and recover from errors that the file system might not be able to deal with and as such, we add a few additional categories within the taxonomy that pertain to file system checkers. Also, we create a separate category for the case where the file system is left in its previous consistent state prior to the execution of the program ($R_{Previous}$). In particular, if the program involves updates on the system's metadata, none of it is reflected to the file system. Table 4 presents our taxonomy in detail.

A file system can detect the injected errors online by checking the return value of the block I/O request ($D_{ErrorCode}$), inspecting the incoming data and performing some sanity checks ($D_{Sanity}$), or using redundancies, e.g. in the form of checksums ($D_{Redundancy}$). A successful detection should alert the user via system call return values or log messages.

To recover from errors, the file system can take several actions. The most basic action is simply passing along the error code from the block layer ($R_{Propagate}$). The file system can also decide to terminate the execution of the system call, either gracefully via transaction abort, or abruptly such as crashing the kernel ($R_{Stop}$). Lastly, the file system can perform retries ($R_{Retry}$) in case the error is transient, or use its redundancy data structures to recover the data.

It is important to note that for block I/O errors, the actual data stored in the block is not passed to the disk or the file system. Hence, no sanity check can be performed and $D_{Sanity}$ is not applicable. Similarly, for silent data corruption experiments, our module does not return an error code, so $D_{ErrorCode}$ is not relevant.

We also run each file system's *fsck* utility and report on its ability to detect and recover file systems errors offline, as it may employ different detection and recovery strategies than the online file system. The different categories for *fsck* recovery are shown in Table 4.

## 3 Results

Tables 5 and 6 provide a high-level summary of the results from our error injection experiments following the detection and recovery taxonomy from Table 4. Our results are organized into six columns corresponding to the fault modes we emulate. The six tables in each column represent the fault detection and recovery results for each file system under a particular fault. The columns (a-w) in each table correspond to the programs listed in Table 2, which specify the operation during which the fault mode was encountered, and rows correspond to the file system specific data structure, that was affected by the fault.

Note that the columns in Tables 5 and 6 have a one-to-one correspondence to the fault modes described in Section 2 (Table 1), with the exception of *shorn writes*. After a shorn write is injected during test program execution and persisted to the flash device, we examine two scenarios where the persisted partial data is accessed again: during fsck invocation (*Shorn Write + Fsck* column) and test program execution (*Shorn Write + Program Read* column).

### 3.1 Btrfs

We observe in Table 5 that Btrfs is the only file system that consistently detects all I/O errors as well as corruption events, including those affecting data (rather than only metadata). It achieves this through the extensive use of checksums.

However, we find that Btrfs is much less successful in recovering from any issues than the other two file systems. It is the only file system where four of the six error modes can lead to a kernel crash or panic and subsequently a file system that cannot be mounted even after running `btrfsck`. It also has the largest number of scenarios that result in an unmountable file system after `btrfsck` (even if not preceded by a kernel crash). Furthermore, we find that node level checksums, although good for detecting block corruption, they remove an entire node even if a single byte becomes corrupted. As a result, large chunks of data are removed, causing data loss.

Before we describe the results in more detail below, we provide a brief summary of Btrfs data structures. The Btrfs

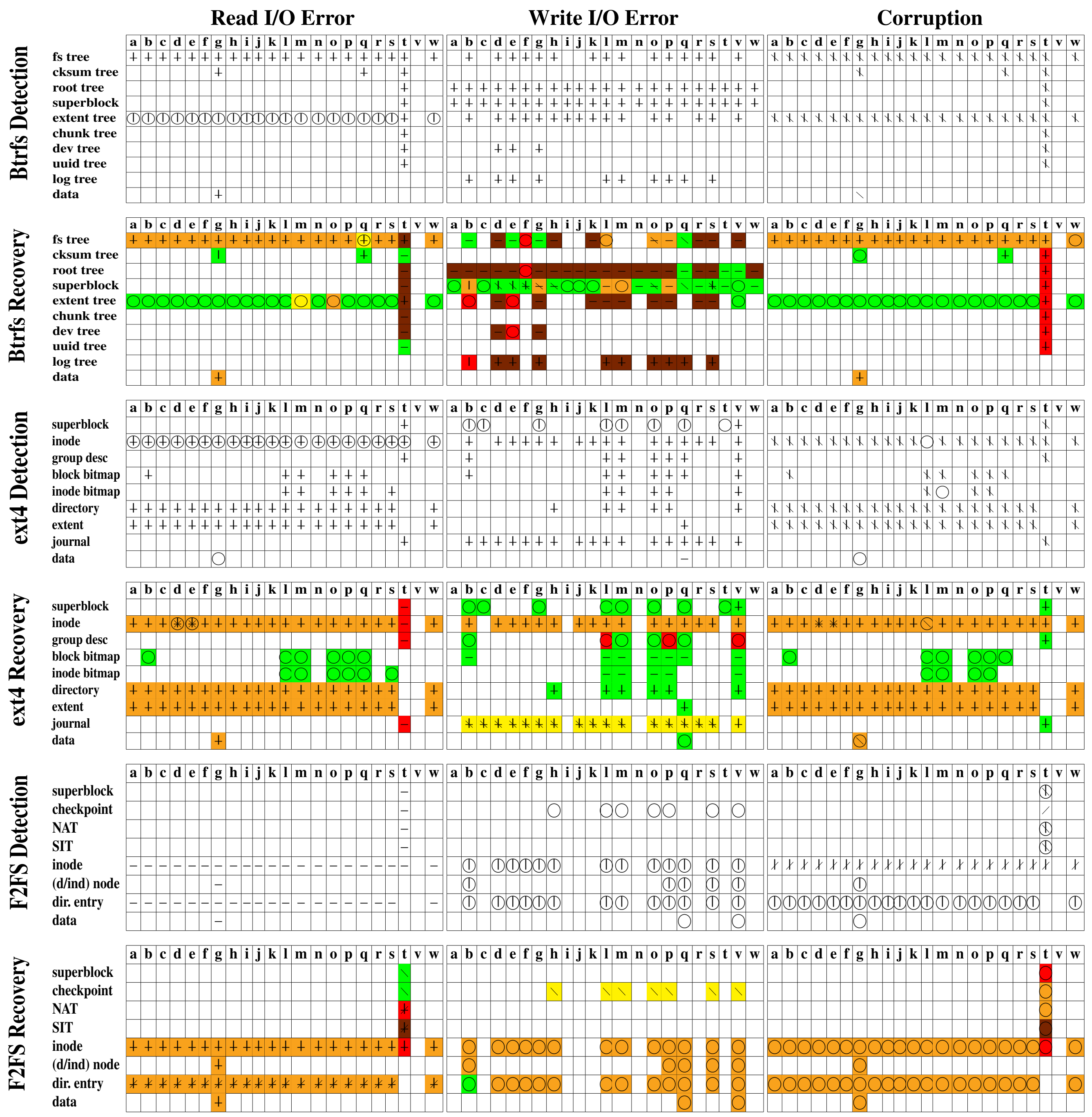
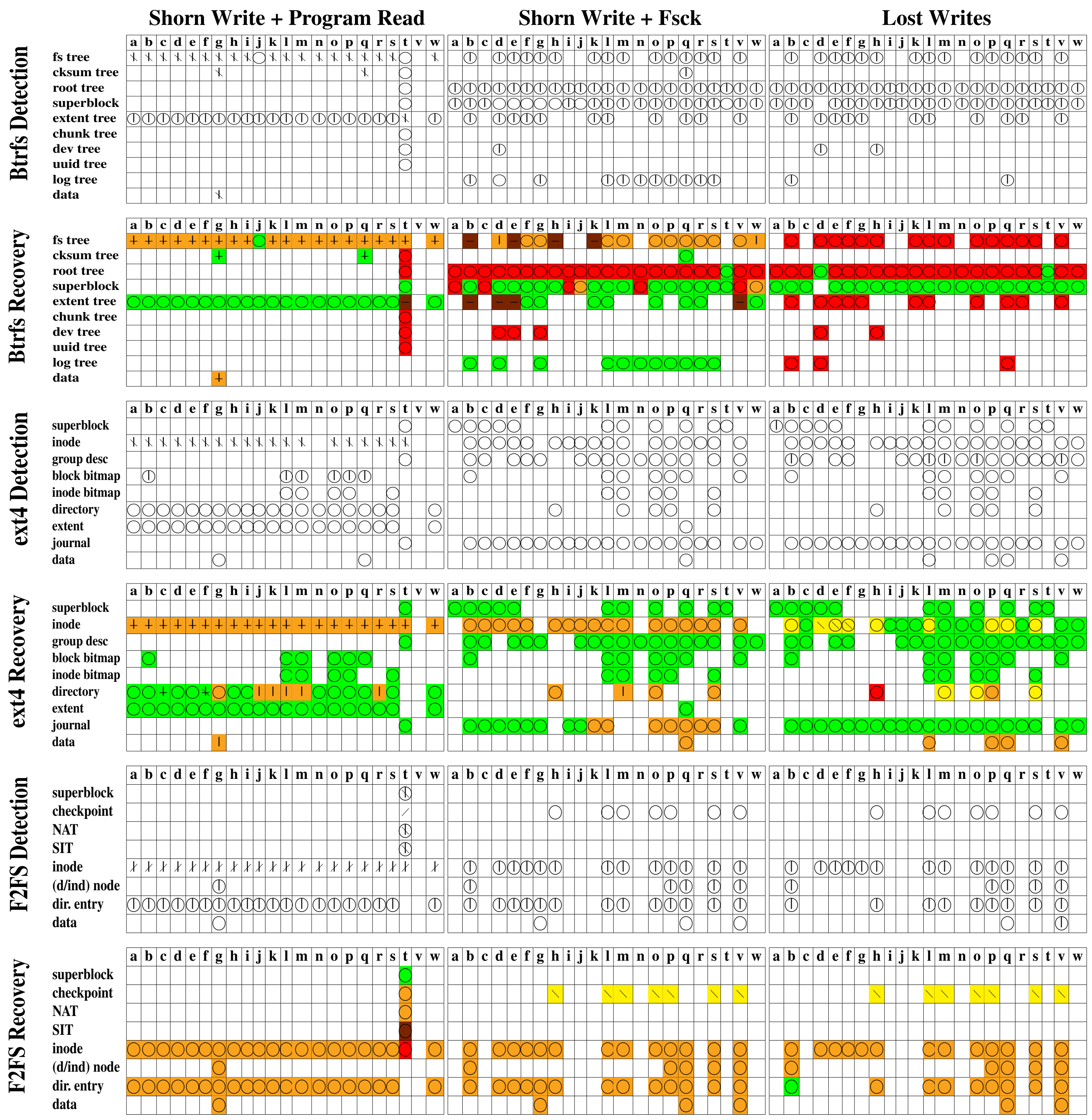

Read I/O Error
Write I/O Error
Corruption
Btrfs Detection
Btrfs Recovery
ext4 Detection
ext4 Recovery
F2FS Detection
F2FS Recovery
a b c d e f g h i j k l m n o p q r s t v w
fs tree
cksum tree
root tree
superblock
extent tree
chunk tree
dev tree
uuid tree
log tree
data
inode
group desc
block bitmap
inode bitmap
directory
extent
journal
checkpoint
NAT
SIT
(d/ind) node
dir. entry

Table 5: *The results of our analysis on the detection and recovery policies of Btrfs, ext4, and F2FS for different read, write, and corruption experiments. The programs that were used are:* ***a:*** *access* ***b:*** *truncate* ***c:*** *open* ***d:*** *chmod* ***e:*** *chown* ***f:*** *utimes* ***g:*** *read* ***h:*** *rename* ***i:*** *stat* ***j:*** *lstat* ***k:*** *readlink* ***l:*** *symlink* ***m:*** *unlink* ***n:*** *chdir* ***o:*** *rmdir* ***p:*** *mkdir* ***q:*** *write* ***r:*** *getdirentries* ***s:*** *creat* ***t:*** *mount* ***v:*** *umount* ***w:*** *chroot. An empty box indicates that the block type is not applicable to the program in execution. Superimposed symbols indicate that multiple mechanisms were used.*

| | | | | |
|---|---|---|---|---|
| ◯ $D_{Zero}$ | – $D_{ErrorCode}$ | \ $D_{Sanity}$ | / $D_{Redundancy}$ | \| $D_{Fsck}$ |
| ◯ $R_{Zero}$ | / $R_{Retry}$ | \| $R_{Propagate}$ | \ $R_{Previous}$ | – $R_{Stop}$ |
| 🟩 $R_{Fsck\_Full}$ | 🟨 $R_{Fsck\_Orig}$ | 🟧 $R_{Fsck\_Partial}$ | 🟥 $R_{Fsck\_Fail}$ | 🟫 $Crash/Panic+R_{Fsck\_fail}$ |



Shorn Write + Program Read
Shorn Write + Fsck
Lost Writes
Btrfs Detection
Btrfs Recovery
ext4 Detection
ext4 Recovery
F2FS Detection
F2FS Recovery
a b c d e f g h i j k l m n o p q r s t v w
fs tree
cksum tree
root tree
superblock
extent tree
chunk tree
dev tree
uuid tree
log tree
data
inode
group desc
block bitmap
inode bitmap
directory
extent
journal
checkpoint
NAT
SIT
(d/ind) node
dir. entry

Table 6: *The results of our analysis on the detection and recovery policies of Btrfs, ext4, and F2FS for different shorn write + program read, shorn write + fsck, and lost write experiments. The programs that were used are:* ***a:*** *access* ***b:*** *truncate* ***c:*** *open* ***d:*** *chmod* ***e:*** *chown* ***f:*** *utimes* ***g:*** *read* ***h:*** *rename* ***i:*** *stat* ***j:*** *lstat* ***k:*** *readlink* ***l:*** *symlink* ***m:*** *unlink* ***n:*** *chdir* ***o:*** *rmdir* ***p:*** *mkdir* ***q:*** *write* ***r:*** *getdirentries* ***s:*** *creat* ***t:*** *mount* ***v:*** *umount* ***w:*** *chroot. An empty box indicates that the block type is not applicable to the program in execution. Superimposed symbols indicate that multiple mechanisms were used.*

| | | | | |
|---|---|---|---|---|
| ◯ $D_{Zero}$ | – $D_{ErrorCode}$ | \ $D_{Sanity}$ | / $D_{Redundancy}$ | ǀ $D_{Fsck}$ |
| ◯ $R_{Zero}$ | / $R_{Retry}$ | ǀ $R_{Propagate}$ | \ $R_{Previous}$ | – $R_{Stop}$ |
| 🟩 $R_{Fsck\_Full}$ | 🟨 $R_{Fsck\_Orig}$ | 🟧 $R_{Fsck\_Partial}$ | 🟥 $R_{Fsck\_Fail}$ | 🟫 *Crash/Panic*+$R_{Fsck\_fail}$ |

file system arranges data in the form of a forest of trees, each serving a specific purpose (e.g. file system tree (*fstree*) stores file system metadata, checksum tree (*csumtree*) stores file/directory checksums). Btrfs maintains checksums for all metadata within tree nodes. Checksum for data is computed and stored separately in a checksum tree. A *root tree* stores the location of the root of all other trees in the file system. Since Btrfs is a copy-on-write file system, all changes made on a tree node are first written to a different location on disk. The location of the new tree nodes are then propagated across the internal nodes up to the root of the file system trees. Finally, the *root tree* needs to be updated with the location of the other changed file system trees.

### 3.1.1 Read errors

All errors get detected ($D_{\text{ErrorCode}}$) and registered in the operating system's message *log*, and the current operation is terminated ($R_{\text{Stop}}$). `btrfsck` is able to run, detect and correct the file system in most cases, with two exceptions. When the *fstree* structure is affected, `btrfsck` removes blocks that are not readable and returns an I/O error. Another exception is when a read I/O error is encountered while accessing key tree structures during a `mount` procedure: `mount` fails and `btrfsck` is unable to repair the file system.

### 3.1.2 Corruption

**Corruption of any B-tree node.** Checksums inside each tree node enable reliable detection of corruption; however, Btrfs employs a different recovery protocol based on the type of the underlying device. When Btrfs is deployed on top of a hard disk, it provides recovery from metadata corruption using metadata replication. Specifically, reading a corrupted block leads to `btrfs-scrub` being invoked, which replaces the corrupted primary metadata block with its replica. Note that `btrfs-scrub` does not have to scan the entire file system; only the replica is read to restore the corrupted block. However, in case the underlying device is an SSD, Btrfs turns off metadata replication by default for two reasons [7]. First, an SSD can remap a primary block and its replica internally to a single physical location, thus deduplicating them. Second, SSD controllers may put data written together in a short time span into the same physical storage unit (i.e. cell, erase block, etc.), which is also a unit of SSD failures. Therefore, `btrfs-scrub` is never invoked in the case of SSDs, as there is no metadata duplication. This design choice causes a single bit flip of *fstree* to wipe out files and entire directories within a *B-Tree* node. If a corrupted tree node is encountered while `mount` reads in all metadata trees into memory, the consequences are even more severe: the operation fails and the disk is left in an inconsistent and irreparable state, even after running `btrfsck`.

**Directory corruption.** We observe that when a node corruption affects a directory, the corruption could actually be recovered, but Btrfs fails to do so. For performance reasons, Btrfs maintains two independent data structures for a directory (`DIR_ITEM` and `DIR_INDEX`). If one of these two becomes corrupted, the other data structure is not used to restore the directory. This is surprising considering that the existing redundancy could easily be leveraged for increased reliability.

### 3.1.3 Write errors

**Superblock & Write I/O errors:** Btrfs has multiple copies of its superblock, but interestingly, the recovery policy upon a write error is not identical for all copies. The superblocks are located at fixed locations on the disk and are updated at every write operation. The replicas are kept consistent, which differs from ext4's behavior. We observe that a write I/O error while updating the primary superblock is considered severe; the operation is aborted and the file system remounts as *read-only*. On the other hand, write I/O errors for the secondary copies of the superblock are detected, but the overall operation completes successfully, and the secondary copy is not updated. While this allows the file system to continue writing, this is a violation of the implicit consistency guarantee between all superblocks, which may lead to problems in the future, as the system operates with a reduced number of superblock copies.

**Tree Node & Write I/O errors:** A write I/O error on a tree node is registered in the operating system's message *log*, but due to the file system's asynchronous nature, errors cannot be directly propagated back to the user application that wrote the data. In almost all cases, the file system is forced to mount as *read-only* ($R_{\text{Stop}}$). A subsequent `unmount` and `btrfsck` run in repair mode makes the device unreadable for *extentTree*, *logTree*, *rootTree* and the root node of the *fstree*.

### 3.1.4 Shorn Write + Program Read

We observe that the behavior of Btrfs during a read of a shorn block is similar to the one we observed earlier for corruption. The only exception is the superblock, as its size is smaller than the 3/8th of the block and it does not get affected.

### 3.1.5 Shorn Write + Fsck

Shorn writes on *root tree* cause the file system to become unmountable and unrecoverable even after a `btrfsck` operation. We also find kernel panics during shorn writes as described in Section 3.1.7.

### 3.1.6 Lost Writes

Errors get detected only during `btrfsck`. They do not get detected or propagated to user space during normal operation. `btrfsck` is unable to recover the file system, which is rendered unmountable due to corruption. The only recoverable case is a lost write to the superblock; for the remaining data structures, the file system eventually becomes unmountable.

### 3.1.7 Bugs found/reported.

We submitted 2 bug reports for Btrfs. The first bug report is related to the corruption of a `DIR_INDEX` key. The file system was able to detect the corruption but deadlocks while listing the directory entries. This bug was fixed in a later version [1]. The second bug is related to read I/O errors specifically on the root directory, which can cause a kernel panic for certain programs. We encountered 2 additional bugs during a shorn write that result in a kernel panic, both having the same root cause. The first case involves a shorn write to the root of the *fstree*, while the second case involves a shorn write to the root of the *extent tree*. In both cases, there is a mismatch in the leaf node size, which forces Btrfs to print the entire tree in the operating system's message *log*. While printing the leaf block, another kernel panic occurs where the size of a Btrfs *item* does not match the Btrfs *header*. Rebooting the kernel and running `btrfsck` fails to recover the file system.

## 3.2 ext4

ext4 is the default file system for many widely used Linux distributions and Android devices. It maintains a *journal* where all metadata updates are sequentially written before the main file system is updated. First, data corresponding to a file system operation is written to the in-place location of the file system. Next, a transaction log is written on the *journal*. Once the transaction log is written on the *journal*, the transaction is said to be *committed*. When the *journal* is full or sufficient time has elapsed, a *checkpoint* operation takes place that writes the in-memory metadata buffers to the in-place metadata location on the disk. On the event of a crash before the transaction is committed, the file system transaction is discarded. If the commit has taken place successfully on the *journal* but the transaction has not been checkpointed, the file system replays the *journal* during remount, where all metadata updates that were committed on the *journal* are recovered from the journal and written to the main file system.

ext4 is able to recover from an impressively large range of fault scenarios. Unlike Btrfs, it makes little use of checksums unless *metadata_csum* feature is enabled explicitly during file system creation. Further, it deploys a very rich set of sanity checks when reading data structures such as directories, inodes and extents[1], which helps it deal with corruptions.

It is also the only one of the three file systems that is able to recover lost writes of multiple data structures, due to its in-place nature of writes and a robust file system checker. However, there are a few exceptions where the corresponding issue remains uncorrectable (see the red cells in Table 6 associated with ext4's recovery).

Furthermore, we observe instances of data loss caused by shorn and lost writes involving write programs, such as `create` and `rmdir`. For shorn writes, ext4 may incur silent errors, and not notify the user about the errors.

Before describing some specific issues below, we point out that our ext4 results are very different from those reported for ext3 in [45], where a large number of corruption events and several read and write I/O errors were not detected or handled properly. Clearly, in the 13 years that have passed since then, ext developers have made improvements in reliability a priority, potentially motivated by the findings in [45].

**I/O errors, corruption and shorn writes of Inodes:** The most common scenario leading to data loss (but still a consistent file system) is a fault, in particular read I/O error, corruption or shorn write, that affects an inode, which results in the data of all files having their inode structure stored inside the affected inode block becoming inaccessible.

**Read I/O errors, corruption and shorn writes of Directory blocks:** A shorn write involving a directory block is detected by the file system and eventually, the corresponding files and directories are removed. Empty files are completely removed by *e2fsck* by default, while non-empty files are placed into the *lost+found* directory. However, the parent-child relationship is lost, which we encode as $R_{Fsck\_Partial}$.

**Write I/O errors and group descriptors:** There is only one scenario where `e2fsck` does not achieve at least partial success: When `e2fsck` is invoked after a write error on a group descriptor, it tries to rebuild the group descriptor and write it to the same on-disk location. However, as it is the same on-disk location that generated the initial error, `e2fsck` encounters the same write error and keeps restarting, running into an infinite loop for 3 cases (see $R_{Fsck\_Fail}$).

**Read I/O error during mount:** ext4 fails to complete the `mount` operation if a read I/O error occurs while reading a critical metadata structure. In this case, the file system cannot be mounted even after invoking `e2fsck`. We observe similar behavior for the other two file systems as well.

## 3.3 F2FS

F2FS is a log-structured file system designed for devices that use an FTL. Data and metadata are separately written into 6 active *logs* (default configuration), which are grouped by data/metadata, files/directories, and other heuristics. This *multi-head logging* allows similar blocks to be placed together and increases the number of sequential write operations.

F2FS divides its logical address space into the *metadata region* and the *main area*. The metadata region is stored at a fixed location and includes the Checkpoint (CP), the Segment Information Table (SIT), and the Node Address Table (NAT). The checkpoint stores information about the state of the file system and is used to recover from system crashes. SIT maintains information on the *segments* (the unit at which F2FS allocates storage blocks) in the main area, while NAT contains the block addresses of the file system's *nodes*, which comprise of file/directory inodes, direct, and indirect nodes. F2FS uses a two-location approach for these three structures. In particular, one of the two copies is "active" and used to initialize the file system's state during `mount`, while the other is a shadow copy that gets updated during the file system's execution. Finally, each copy of the checkpoint points to its

[1]We report failure results for both directory and file extents together. Since our pre-workload generation creates a number of files and directories in the root directory, at least 1 extent block corresponding to the root directory gets accessed by all programs.

corresponding copy of SIT and NAT.

F2FS's behavior when encountering read/write errors or corruption differs significantly from that of ext4 and Btrfs. While read failures are detected and appropriately propagated in nearly all scenarios, we observe that F2FS consistently fails to detect and report any write errors, independently of the operation that encounters them and the affected data structure. Furthermore, our results indicate that F2FS is not able to deal with lost and shorn writes effectively and eventually suffers from data loss. In some cases, a run of the file system's checker (called `fsck.f2fs`) can bring the system to a consistent state, but in other cases, the consequences are severe. We describe some of these cases in more detail below.

### 3.3.1 Read errors

**Checkpoint / NAT / SIT blocks & read errors.** During its `mount` operation, if F2FS encounters a read I/O error while trying to fetch any of the checkpoint, NAT, or SIT blocks, then it mounts as *read-only*. Additionally, F2FS cannot be mounted if the inode associated with the root directory cannot be accessed. In general, `fsck.f2fs` cannot help the file system recover from the error since it terminates with an assertion error every time it cannot read a block from the disk.

### 3.3.2 Write errors & Lost Writes

We observe that F2FS does not detect write errors (as injected by our framework), leading to different issues, such as corruption, reading garbage values, and potentially data loss. As a result, during our experiments, newly created or previously existing entries have been completely discarded from the system, applications have received garbage values, and finally, the file system has experienced data loss due to an incomplete recovery protocol (i.e. when `fsck.f2fs` is invoked). Considering that F2FS does not detect write I/O errors, lost writes end up having the same effect, since the difference between the two is that a lost write is silent (i.e. no error is returned).

### 3.3.3 Corruption

Data corruption is reliably detected only for inodes and checkpoints, which are the only data structures protected by checksums, but even for those two data structures, recovery can be incomplete, resulting in the loss of files (and the data stored in them). The corruption of other data structures can lead to even more severe consequences. For example, the corruption of the superblock can go undetected and lead to an unmountable file system, even if the second copy of the superblock is intact. We have filed two bug reports related to the issues we have identified and one has already resulted in a fix. Below we describe some of the issues around corruption in more detail.

**Inode block corruption.** Inodes are one of only two F2FS data structures that are protected by checksums, yet their corruption can still create severe problems. One such scenario arises when the information stored in the *footer* section of an inode block is corrupted. In this case, `fsck.f2fs` will discard the entry without even attempting to create an entry in the *lost_found* directory, resulting in data loss.

Another scenario is when an inode associated with a directory is corrupted. Then all the regular files stored inside that directory and its sub-directories are recursively marked as *unreachable* by `fsck.f2fs` and are eventually moved to the *lost_found* directory (provided that their inode is valid). However, we observe that `fsck.f2fs` does not attempt to recreate the structure of sub-directories. It simply creates an entry in the *lost_found* directory for regular files in the sub-directory tree, not sub-directories. As a result, if there are different files with the same name (stored in different paths of the original hierarchy), then only one is maintained at the end of the recovery procedure.

**Checkpoint corruption.** Checkpoints are the other data structure, besides inodes, that is protected by checksums. We observe that issues only arise if both copies of a checkpoint become corrupted, in which case the file system cannot be mounted. Otherwise, the uncorrupted copy will be used during the system's `mount` operation.

**Superblock corruption.** While there are two copies of the superblock, the detection of corruption to the superblock relies completely on a set of sanity checks performed on (most) of its fields, rather than checksums or comparison of the two copies. If sanity checks identify an invalid value, then the backup copy is used for recovery. However, our results show that the sanity checks are not capable of detecting all invalid values and thus, depending on the corrupted field, the reliability of the file system can suffer.

One particularly dangerous situation is a corruption of the *offset* field, which is used to calculate a checkpoint's starting address inside the corresponding *segment*, as it causes the file system to boot from an invalid checkpoint location during a `mount` operation and to eventually hang during its `unmount` operation. We filed a bug report which has resulted in a new patch that fixes this problem during the operation of `fsck.f2fs`; specifically, the patch uses the (checksum-protected) checkpoint of the system to restore the correct value. Future releases of F2FS will likely include a patch that enables checksums for the superblock.

Another problem with superblock corruption, albeit less severe, arises when the field containing the counter of supported file extensions, which F2FS uses to identify *cold data*, is corrupted. The corruption goes undetected and as a result, the corresponding file extensions are not treated as expected. This might lead to file system performance problems, but should not affect reliability or consistency.

**SIT corruption.** SIT blocks are not protected against corruption through any form of redundancy. We find cases where the corruption of these blocks severely compromises the consistency of the file system. For instance, we were able to corrupt a SIT block's bitmap (which keeps track of the allocated blocks inside a *segment*) in such a way that the file system hit a bug during its `mount` operation and eventually, became unmountable.

**NAT corruption.** This data structure is not protected against corruption and we observe several problems this can create. First, the node ID of an entry can be corrupted and thus, point to another entry inside the file system, to an invalid entry or a non-existing one. Second, the block address of an entry can be corrupted and thus, point to another entry in the system or an invalid location. In both cases, the original entry is eventually marked as *unreachable* by `fsck.f2fs`, since the reference to it is no longer available inside the NAT copy, and placed in the *lost_found* directory. As already mentioned, files with identical names overwrite each other and eventually only one is stored inside the *lost_found* directory.

**Direct/Indirect Node corruption.** These blocks are used to access the data of large files and also, the entries of large directories (with multiple entries). Direct nodes contain entries that point to on-disk blocks, while indirect nodes contain entries that point to direct nodes. Neither single nor double indirect nodes are protected against corruption. We observe that corruption of these nodes is not detected by the file system. Even when an invocation of `fsck.f2fs` detects the corruption problems can arise. For example, we find a case where after the invocation of `fsck.f2fs` the system kept reporting the wrong (corrupted) size for a file. As a result, when we tried to create a copy of the file, we received a different content.

**Directory entry corruption.** Directory entries are stored and organized into blocks. Currently, there is no mechanism to detect corruption of such a block and we observe numerous problems this can create. For example, when the field in a directory entry that contains the length of the entry's name is corrupted the file system returns garbage information when we try to get the file's status. Moreover, the field containing the node ID of the corresponding inode can be corrupted and as a result point to any node currently stored in the system. Finally, an entry can "disappear" by storing a zero value into the corresponding index inside the directory's bitmap.

In the last two cases, any affected entry is eventually marked as *unreachable* by `fsck.f2fs`, since their parent directory does no longer point to it. As already mentioned, files with identical names overwrite each other and eventually only one is stored inside the *lost_found* directory.

#### 3.3.4 Shorn Write + Program Read

The results when a program reads a block previously affected by a shorn write are similar to those for corruption, since shorn writes can be viewed as a special type of corruption. The only exception is the superblock, as it is a small data structure that happened not to be affected by our experiments.

#### 3.3.5 Shorn Write + Fsck

**Directory entries and shorn writes.** Blocks that contain directory entries are not protected against corruption. Therefore, a shorn write goes undetected and can cause several problems. First, valid entries of the system "disappear" after invoking `fsck.f2fs`, including the special entries that point to the current directory and its parent. Second, in some cases, we additionally observed that after re-mounting the file system, an attempt to list the contents of a directory resulted in an infinite loop. In both cases, the affected entries were eventually marked as *unreachable* by `fsck.f2fs` and were dumped into the *lost_found* directory. As we have already mentioned, files with identical names in different parts of the directory tree conflict with each other and eventually, only one makes it to the *lost_found* directory. In some cases, `fsck.f2fs` is not capable of detecting the entire damage a shorn write has caused; we ran into a case where after remounting the file system, all the entries inside a directory ended up having the same name, eventually becoming completely inaccessible.

#### 3.3.6 Bugs found/reported

We have filed two bug reports related to the issues we have identified around handling corrupted data and one has already resulted in a fix [2, 4]. Moreover, we have reported F2FS's failure to handle write I/O errors [3].

## 4 Related Work

Our work is closest in spirit to the pioneering work by Prabhakaran et *al.* [45], however our focus is very different. While [45] was focused on HDDs, we are specifically interested in SSD-based systems and as such consider file systems with features that are attractive for usage with SSDs, including log-structured and copy-on-write systems. None of the file systems in our study existed at the time [45] was written and they mark a significant departure in terms of design principles compared to the systems in [45]. Also, since we are focused on SSDs, we specifically consider reliability issues that arise from the use of SSDs. Additionally, we provide some extensions to the work in [45], such as exploring whether *fsck* is able to detect and recover from those issues that the file systems cannot handle during their online operation.

Gunawi et *al.* [28] make use of static analysis and explore how file systems and storage device drivers propagate error codes. Their results indicate that write errors are neglected more often than read errors. Our study confirms that write errors are still not handled properly in some file systems, especially when lost and shorn writes are considered. In [51], the authors conduct a performance evaluation on the transaction processing system between ext2 and NILFS2. In [42], the authors explore how existing file systems developed for different operating systems behave with respect to features such as crash resilience and performance. Nonetheless, the provided experimental results only present the performance of read and write operations. Recently, two new studies presented their reliability analysis of file systems in a context other than the local file system. Ganesan et *al.* [25] present their analysis on how modern *distributed storage systems* behave in the presence of file-system faults. Cao et *al.* [20] present their study on the reliability of *high-performance parallel systems*.

In contrast, in our work, we focus on local file systems and explore the effect of SSD related errors.

Finally, different techniques involving hardware or modifications inside the FTL have been proposed to mitigate existing errors inside SSDs [14, 15, 22, 35–37, 44].

## 5 Implications

• ext4 has significantly improved over ext3 in both detection and recovery from data corruption and I/O injection errors. Our extensive test suite generates only minor errors or data losses in the file system, in stark contrast with [45], where ext3 was reported to silently discard write errors.
• On the other hand, Btrfs, which is a production grade file system with advanced features like snapshot and cloning, has good failure detection mechanisms, but is unable to recover from errors that affect its key data structures, partially due to disabling metadata replication when deployed on SSDs.
• F2FS has the weakest detection against the various errors our framework emulates. We observe that F2FS consistently fails to detect and report any write errors, regardless of the operation that encounters them and the affected data structure. It also does not detect many corruption scenarios. The result can be as severe as data loss or even an unmountable file system. We have filed 3 bug reports; 1 has already been fixed and the other 2 are currently under development.
• File systems do not always make use of the existing redundancy. For example, Btrfs maintains two independent data structures for each directory entry for enhanced performance, but upon failure of one, does not use the other for recovery.
• We notice potentially fatal omissions in error detection and recovery for all file systems except for ext4. This is concerning since technology trends, such as continually growing SSD drive capacities and increasing densities as QLC drives which are coming on the market, all seem to point towards increasing rather than decreasing SSD error rates in the future. In particular for flash-focused file systems, such as F2FS, where for a long time focus has been on performance optimization, an emphasis on reliability is needed if they want to be a serious contender for ext4.
• File systems should make every effort to verify the correctness of metadata through sanity checks, especially when the metadata is not protected by a mechanism, such as checksums. The most mature file system in our study, ext4, does a significantly more thorough job at sanity checks than for example F2FS, which has room for improvement. There have also been recent efforts towards this direction in the context of a popular enterprise file system [32].
• Checksums can be a double-edged sword. While they help increase error detection, coarse granularity checksums can lead to severe data loss. For instance, manipulation of even 1 byte of the checksummed file system unit leads to discard of the entire file system unit in the case for Btrfs. Ideally having a directory or a file system level checksum that discards only 1 entity instead of all co-located files/directories should be implemented. A step in this direction is File-Level Integrity proposed for Android [5, 24]. The tradeoff of adding fine-grained checksums is the space and performance overhead, since a checksum protects a single inode instead of a block of inodes. Finally, note that checksums can only help with detecting corruption, but not with recovery (ideally a file system can both detect corruption and recover from it). These points have to be considered together when implementing checksums inside the file system.
• One might wonder whether added redundancy as described in the Iron file system paper [45] might resolve many of the issues we observe. We hypothesize that for flash-based systems, redundancy can be less effective in those (less likely) cases where both the primary and replica blocks land in the same fault domain (same erase block or same flash chip), after being written together within a short time interval. Even though modern flash devices keep multiple erase blocks open and stripe incoming writes among them for throughput, this does not preclude the scenario where both the primary and replica blocks land in the same fault domain.
• Maybe not surprisingly, a few key data structures (e.g. the journal's *superblock* in ext4, the root directory *inode* in ext4 and F2FS, the root node of *fstree* in Btrfs) are responsible for the most severe failures, usually when affected by a silent fault (e.g. silent corruption or silently dropped write).It might be worthwhile to perform a series of sanity checks for such key data structures before persisting them to the SSD e.g. during an `unmount` operation.

## 6 Limitations and Future Work

Some of the fault types we explore in our study are based on SSD models that are several years old by now, whose internal behavior could have changed since then. However, we observe that some issues are inherent to flash and therefore likely to persist in new generations of drives, such as retention and disturb errors, which will manifest as read errors at the file system level. The manifestation of other faults, e.g. those related to firmware bugs or changes in page and block size, might vary for future drive models. Our tool is configurable and can be extended to test new error patterns.

File systems must remain consistent in the face of different types of faults. As part of future work, we plan to extend our device mapper module to emulate additional fault modes, such as timeouts. Additionally, our work can be expanded to include additional file systems, such as XFS, NTFS and ZFS. Finally, another extension to our work could be exploring how file systems respond to timing anomalies as those described in [26], where I/Os related to some blocks can become slower, or the whole drive is slow.

## Acknowledgements

We thank our USENIX ATC '19 reviewers and our shepherd, Theodore Ts'o, for their detailed feedback and valuable suggestions.

## References

[1] Btrfs Bug Report. https://bugzilla.kernel.org/show_bug.cgi?id=198457.

[2] F2FS Bug Report. https://bugzilla.kernel.org/show_bug.cgi?id=200635.

[3] F2FS Bug Report - Write I/O Errors. https://bugzilla.kernel.org/show_bug.cgi?id=200871.

[4] F2FS Patch File. https://sourceforge.net/p/linux-f2fs/mailman/message/36402198/.

[5] fs-verity: File System-Level Integrity Protection. https://www.spinics.net/lists/linux-fsdevel/msg121182.html. [Online; accessed 06-Jan-2019].

[6] Github code repository. https://github.com/uoftsystems/dm-inject.

[7] Btrfs mkfs man page. https://btrfs.wiki.kernel.org/index.php/Manpage/mkfs.btrfs, 2019. [Online; accessed 06-Jan-2019].

[8] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D Davis, Mark S Manasse, and Rina Panigrahy. Design tradeoffs for SSD performance. In *USENIX Annual Technical Conference (ATC '08)*, volume 57, 2008.

[9] Lakshmi N Bairavasundaram, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, Garth R Goodson, and Bianca Schroeder. An Analysis of Data Corruption in the Storage Stack. *ACM Transactions on Storage (TOS)*, 4(3):8, 2008.

[10] Hanmant P Belgal, Nick Righos, Ivan Kalastirsky, Jeff J Peterson, Robert Shiner, and Neal Mielke. A new reliability model for post-cycling charge retention of flash memories. In *Proceedings of the 40th Annual International Reliability Physics Symposium*, pages 7–20. IEEE, 2002.

[11] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. LightNVM: The Linux Open-Channel SSD Subsystem. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST '17)*, pages 359–374, Santa Clara, CA, 2017. USENIX Association.

[12] Simona Boboila and Peter Desnoyers. Write Endurance in Flash Drives: Measurements and Analysis. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST '10)*, pages 115–128. USENIX Association, 2010.

[13] Adam Brand, Ken Wu, Sam Pan, and David Chin. Novel read disturb failure mechanism induced by FLASH cycling. In *Proceedings of the 31st Annual International Reliability Physics Symposium*, pages 127–132. IEEE, 1993.

[14] Yu Cai, Saugata Ghose, Erich F Haratsch, Yixin Luo, and Onur Mutlu. Error characterization, mitigation, and recovery in flash-memory-based solid-state drives. *Proceedings of the IEEE*, 105(9):1666–1704, 2017.

[15] Yu Cai, Saugata Ghose, Yixin Luo, Ken Mai, Onur Mutlu, and Erich F Haratsch. Vulnerabilities in MLC NAND flash memory programming: experimental analysis, exploits, and mitigation techniques. In *23rd International Symposium on High-Performance Computer Architecture (HPCA)*, pages 49–60. IEEE, 2017.

[16] Yu Cai, Erich F Haratsch, Onur Mutlu, and Ken Mai. Error patterns in MLC NAND flash memory: Measurement, Characterization, and Analysis. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 521–526. EDA Consortium, 2012.

[17] Yu Cai, Yixin Luo, Erich F Haratsch, Ken Mai, and Onur Mutlu. Data retention in MLC NAND flash memory: Characterization, optimization, and recovery. In *21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 551–563. IEEE, 2015.

[18] Yu Cai, Onur Mutlu, Erich F Haratsch, and Ken Mai. Program interference in MLC NAND flash memory: Characterization, modeling, and mitigation. In *31st International Conference on Computer Design (ICCD)*, pages 123–130. IEEE, 2013.

[19] Yu Cai, Gulay Yalcin, Onur Mutlu, Erich F Haratsch, Adrian Cristal, Osman S Unsal, and Ken Mai. Flash correct-and-refresh: Retention-aware error management for increased flash memory lifetime. In *30th International Conference on Computer Design (ICCD)*, pages 94–101. IEEE, 2012.

[20] Jinrui Cao, Om Rameshwar Gatla, Mai Zheng, Dong Dai, Vidya Eswarappa, Yan Mu, and Yong Chen. PFault: A General Framework for Analyzing the Reliability of High-Performance Parallel File Systems. In *Proceedings of the 2018 International Conference on Supercomputing*, pages 1–11. ACM, 2018.

[21] Paolo Cappelletti, Roberto Bez, Daniele Cantarelli, and Lorenzo Fratin. Failure mechanisms of Flash cell in program/erase cycling. In *Proceedings of the IEEE International Electron Devices Meeting*, pages 291–294. IEEE, 1994.

[22] Feng Chen, David A. Koufaty, and Xiaodong Zhang. Understanding Intrinsic Characteristics and System Implications of Flash Memory Based Solid State Drives. In *Proceedings of the 2009 ACM SIGMETRICS International Conference on Measurement and Modeling of

*Computer Systems (SIGMETRICS '09)*, pages 181–192, 2009.

[23] Robin Degraeve, F Schuler, Ben Kaczer, Martino Lorenzini, Dirk Wellekens, Paul Hendrickx, Michiel van Duuren, GJM Dormans, Jan Van Houdt, L Haspeslagh, et al. Analytical percolation model for predicting anomalous charge loss in flash memories. *IEEE Transactions on Electron Devices*, 51(9):1392–1400, 2004.

[24] Jake Edge. File-level Integrity. https://lwn.net/Articles/752614/, 2018. [Online; accessed 06-Jan-2019].

[25] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Redundancy Does Not Imply Fault Tolerance: Analysis of Distributed Storage Reactions to Single Errors and Corruptions. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST '17)*, pages 149–166, Santa Clara, CA, 2017. USENIX Association.

[26] L. M. Grupp, A. M. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P. H. Siegel, and J. K. Wolf. Characterizing Flash Memory: Anomalies, Observations, and Applications. In *42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 24–33, Dec 2009.

[27] Laura M Grupp, John D Davis, and Steven Swanson. The bleak future of NAND flash memory. In *Proceedings of the 10th USENIX conference on File and Storage Technologies (FAST '12)*. USENIX Association, 2012.

[28] Haryadi S. Gunawi, Cindy Rubio-González, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dussea, and Ben Liblit. EIO: Error Handling is Occasionally Correct. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST '08)*, pages 14:1–14:16, San Jose, CA, 2008.

[29] S Hur, J Lee, M Park, J Choi, K Park, K Kim, and K Kim. Effective program inhibition beyond 90nm NAND flash memories. *Proc. NVSM*, pages 44–45, 2004.

[30] Seok Jin Joo, Hea Jong Yang, Keum Hwan Noh, Hee Gee Lee, Won Sik Woo, Joo Yeop Lee, Min Kyu Lee, Won Yol Choi, Kyoung Pil Hwang, Hyoung Seok Kim, et al. Abnormal disturbance mechanism of sub-100 nm NAND flash memory. *Japanese Journal of Applied Physics*, 45(8R):6210, 2006.

[31] Myoungsoo Jung and Mahmut Kandemir. Revisiting Widely Held SSD Expectations and Rethinking System-level Implications. In *Proceedings of the 2013 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '13)*, pages 203–216, 2013.

[32] Harendra Kumar, Yuvraj Patel, Ram Kesavan, and Sumith Makam. High Performance Metadata Integrity Protection in the WAFL Copy-on-Write File System. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST '17)*, pages 197–212, Santa Clara, CA, 2017. USENIX Association.

[33] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. F2FS: A New File System for Flash Storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST '15)*, pages 273–286, Santa Clara, CA, 2015. USENIX Association.

[34] Jae-Duk Lee, Chi-Kyung Lee, Myung-Won Lee, Han-Soo Kim, Kyu-Charn Park, and Won-Seong Lee. A new programming disturbance phenomenon in NAND flash memory by source/drain hot-electrons generated by GIDL current. In *Non-Volatile Semiconductor Memory Workshop, 2006. IEEE NVSMW 2006. 21st*, pages 31–33. IEEE, 2006.

[35] Ren-Shuo Liu, Chia-Lin Yang, and Wei Wu. Optimizing NAND flash-based SSDs via retention relaxation. In *Proceedings of the 10th USENIX conference on File and Storage Technologies (FAST '12)*, page 11, San Jose, CA, 2012. USENIX Association.

[36] Yixin Luo, Saugata Ghose, Yu Cai, Erich F Haratsch, and Onur Mutlu. HeatWatch: Improving 3D NAND Flash Memory Device Reliability by Exploiting Self-Recovery and Temperature Awareness. In *24th International Symposium on High Performance Computer Architecture (HPCA)*, pages 504–517. IEEE, 2018.

[37] Yixin Luo, Saugata Ghose, Yu Cai, Erich F. Haratsch, and Onur Mutlu. Improving 3D NAND Flash Memory Lifetime by Tolerating Early Retention Loss and Process Variation. *Proceedings of the 2018 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '18)*, 2(3):37:1–37:48, December 2018.

[38] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. The New ext4 Filesystem: Current Status and Future Plans. In *Proceedings of the Linux symposium*, volume 2, pages 21–33, 2007.

[39] Justin Meza, Qiang Wu, Sanjev Kumar, and Onur Mutlu. A Large-Scale Study of Flash Memory Failures in the Field. In *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '15)*, pages 177–190, 2015.

[40] Neal Mielke, Hanmant P Belgal, Albert Fazio, Qingru Meng, and Nick Righos. Recovery Effects in the Distributed Cycling of Flash Memories. In *Proceedings of the 44th Annual International Reliability Physics Symposium*, pages 29–35. IEEE, 2006.

[41] Neal Mielke, Todd Marquart, Ning Wu, Jeff Kessenich, Hanmant Belgal, Eric Schares, Falgun Trivedi, Evan Goodness, and Leland R Nevill. Bit error rate in NAND flash memories. In *Proceedings of the 46th Annual International Reliability Physics Symposium*, pages 9–19. IEEE, 2008.

[42] Keshava Munegowda, GT Raju, and Veera Manikandan Raju. Evaluation of file systems for solid state drives. In *Proceedings of the Second International Conference on Emerging Research in Computing, Information, Communication and Applications*, pages 342–348, 2014.

[43] Iyswarya Narayanan, Di Wang, Myeongjae Jeon, Bikash Sharma, Laura Caulfield, Anand Sivasubramaniam, Ben Cutler, Jie Liu, Badriddine Khessib, and Kushagra Vaid. SSD Failures in Datacenters: What? When? And Why? In *Proceedings of the 9th ACM International on Systems and Storage Conference (SYSTOR '16)*, pages 7:1–7:11, 2016.

[44] Nikolaos Papandreou, Thomas Parnell, Haralampos Pozidis, Thomas Mittelholzer, Evangelos Eleftheriou, Charles Camp, Thomas Griffin, Gary Tressler, and Andrew Walls. Using Adaptive Read Voltage Thresholds to Enhance the Reliability of MLC NAND Flash Memory Systems. In *Proceedings of the 24th Edition of the Great Lakes Symposium on VLSI (GLSVLSI '14)*, pages 151–156, 2014.

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

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

[47] Marco AA Sanvido, Frank R Chu, Anand Kulkarni, and Robert Selinger. NAND flash memory and its role in storage architectures. *Proceedings of the IEEE*, 96(11):1864–1874, 2008.

[48] Bianca Schroeder, Raghav Lagisetty, and Arif Merchant. Flash Reliability in Production: The Expected and the Unexpected. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST '16)*, pages 67–80, Santa Clara, CA, 2016. USENIX Association.

[49] Kang-Deog Suh, Byung-Hoon Suh, Young-Ho Lim, Jin-Ki Kim, Young-Joon Choi, Yong-Nam Koh, Sung-Soo Lee, Suk-Chon Kwon, Byung-Soon Choi, Jin-Sun Yum, et al. A 3.3 V 32 Mb NAND flash memory with incremental step pulse programming scheme. *IEEE Journal of Solid-State Circuits*, 30(11):1149–1156, 1995.

[50] Hung-Wei Tseng, Laura Grupp, and Steven Swanson. Understanding the Impact of Power Loss on Flash Memory. In *Proceedings of the 48th Design Automation Conference (DAC '11)*, pages 35–40, San Diego, CA, 2011.

[51] Yongkun Wang, Kazuo Goda, Miyuki Nakano, and Masaru Kitsuregawa. Early experience and evaluation of file systems on SSD with database applications. In *5th International Conference on Networking, Architecture, and Storage (NAS)*, pages 467–476. IEEE, 2010.

[52] Mai Zheng, Joseph Tucek, Feng Qin, and Mark Lillibridge. Understanding the Robustness of SSDs Under Power Fault. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST '13)*, pages 271–284, San Jose, CA, 2013. USENIX Association.

[53] Mai Zheng, Joseph Tucek, Feng Qin, Mark Lillibridge, Bill W. Zhao, and Elizabeth S. Yang. Reliability Analysis of SSDs Under Power Fault. *ACM Transactions on Storage (TOS)*, 34(4):1–28, November 2016.