Towards Robust File System Checkers

OM RAMESHWAR GATLA and MAI ZHENG, Iowa State University
MUHAMMAD HAMEED, New Mexico State University
VIACHESLAV DUBEYKO, ADAM MANZANARES, FILIP BLAGOJEVIC, CYRIL GUYOT, and ROBERT MATEESCU, Western Digital Research

File systems may become corrupted for many reasons despite various protection techniques. Therefore, most file systems come with a checker to recover the file system to a consistent state. However, existing checkers are commonly assumed to be able to complete the repair without interruption, which may not be true in practice. In this work, we demonstrate via fault injection experiments that checkers of widely used file systems (EXT4, XFS, BtrFS, and F2FS) may leave the file system in an uncorrectable state if the repair procedure is interrupted unexpectedly. To address the problem, we first fix the ordering issue in the undo logging of `e2fsck` and then build a general logging library (i.e., `rfsck-lib`) for strengthening checkers. To demonstrate the practicality, we integrate `rfsck-lib` with existing checkers and create two new checkers: `rfsck-ext`, a robust checker for Ext-family file systems, and `rfsck-xfs`, a robust checker for XFS file systems, both of which require only tens of lines of modification to the original versions. Both `rfsck-ext` and `rfsck-xfs` are resilient to faults in our experiments. Also, both checkers incur reasonable performance overhead (i.e., up to 12%) compared to the original unreliable versions. Moreover, `rfsck-ext` outperforms the patched `e2fsck` by up to nine times while achieving the same level of robustness.

CCS Concepts: • **General and reference** → **Reliability**; • **Computer systems organization** → **Reliability**; *Secondary storage organization*; • **Software and its engineering** → **File systems management**; **Software testing and debugging**;

Additional Key Words and Phrases: Fault tolerance, data corruption, file-system faults

ACM Reference format:
Om Rameshwar Gatla, Mai Zheng, Muhammad Hameed, Viacheslav Dubeyko, Adam Manzanares, Filip Blagojevic, Cyril Guyot, and Robert Mateescu. 2018. Towards Robust File System Checkers. *ACM Trans. Storage* 14, 4, Article 35 (December 2018), 25 pages.
https://doi.org/10.1145/3281031

1 INTRODUCTION

Achieving data integrity is critical for computer systems ranging from a single desktop to large-scale distributed storage clusters [23]. In order to make sense of the ever-increasing amount of data stored, it is common to use local (e.g., Ext4 [5], XFS [76], F2FS [55]) and multi-node file systems (e.g., HDFS [72], Ceph [80], Lustre [10]) to organize the data on top of storage devices. Although

Authors' addresses: O. R. Gatla and M. Zheng, 2520 Osborn Drive, Ames, IA 50011; emails: {ogatla, mai}@iastate.edu; M. Hameed, 1290 Frenger Mall, Las Cruces, NM 88003; email: mkhameed@nmsu.edu; V. Dubeyko, A. Manzanares, F. Blagojevic, C. Guyot, and R. Mateescu, 591 SanDisk Drive, Milpitas, CA 95035; emails: {vyacheslav.dubeyko, adam.manzanares, filip.blagojevic, cyril.guyot, robert.mateescu}@wdc.com.

1553-3077/2018/12-ART35 $15.00
https://doi.org/10.1145/3281031

file systems are designed to maintain data integrity [40, 42, 51, 66, 78, 81], situations arise when the file system metadata needs to be checked for integrity. Such situations may be caused by power outages, server crashes, latent sector errors, software bugs, and the like [21, 22, 35, 57, 60].

File system checkers, such as `e2fsck` for Ext-family file systems [4], serve as the last line of defense to recover a corrupted file system back to a healthy state [60]. They contain intimate knowledge of file system metadata structures and are commonly assumed to be able to complete the repair *without interruption*.

Unfortunately, the same issues that lead to file system inconsistencies (e.g., power outages or crashes) can also occur during file system repair. One real-world example happened at the High Performance Computing Center in Texas [19]. In this accident, multiple Lustre file systems suffered severe data loss after power outages: The first outage triggered the Lustre checker (`lfsck` [7]) after the cluster was restarted, while another outage interrupted `lfsck` and led to downtime and data loss. Because Lustre is built on top of a variant of Ext4 (`ldiskfs` [10]), and `lfsck` relies on `e2fsck` to fix local inconsistencies on each node, the checking and repairing is complicated (e.g., requiring several days [19]). As of today, it is still unclear which step of `lfsck`/`e2fsck` caused the uncorrectable corruptions. With the trend toward increasing storage capacity and scaling to more and more nodes, checking and repairing file systems will likely become more time-consuming and thus more vulnerable to faults. Such accidents and observation motivate us to remove the assumption that file system checkers can always finish normally without interruption.

Previous research has demonstrated that file system checkers themselves are error-prone [31, 48]. File system-specific approaches have also been developed that use higher level languages to elegantly describe file system repair tasks [48]. In addition, efforts have also been made to speed up the repair procedure, which leads to a smaller window of potential data loss due to an interruption [60]. Although these efforts improve file system checkers, they do not address the fundamental issue of improving the resilience of checkers in the face of *unexpected interruptions*.

In this work, we first demonstrate that the checkers of widely used file systems (i.e., `e2fsck` [4], `xfs_repair` [16], `btrfs-fsck`, and `f2fs-fsck`) may leave the file system in an uncorrectable state if the repair procedure is unexpectedly interrupted. We collect corrupted file system images from file system developers and additionally generate test images to trigger the repair procedure. Moreover, we develop `rfsck-test`, an automatic fault injection tool, to systematically inject faults during the repair, and thus manifest the vulnerabilities. In addition, we have also analyzed the root cause of the uncorrectable corruptions caused by interrupting `e2fsck` using log information from the fault injection tool.

To address the problem exposed in our study, we analyze the undo logging feature of `e2fsck` in depth and identify an ordering issue which jeopardizes its effectiveness. We fix the issue and create a patched version called `e2fsck-patch` which is truly resilient to faults.

However, we find that `e2fsck-patch` is inherently suboptimal as it requires extensive sync operations. To address the limitation and to improve the checkers of other file systems, we design and implement `rfsck-lib`, a general logging library with a simple interface. Based on the similarities among checkers, `rfsck-lib` decouples logging from repairing and provides an interface to log the repairing writes in fine granularity.

To demonstrate the practicality, we integrate `rfsck-lib` with existing checkers and create two new checkers: `rfsck-ext`, a robust checker for Ext-family file systems, which adds 50 lines of code (LoC) to `e2fsck`; and `rfsck-xfs`, a robust checker for XFS file system, which adds 15 LoC to `xfs_repair`.[1] Both `rfsck-ext` and `rfsck-xfs` are resilient to faults in our experiments. Also, both checkers incur reasonable performance overhead (i.e., up to 12%) compared to the original

[1]The prototypes of `rfsck-test`, `e2fsck-patch`, `rfsck-lib`, `rfsck-ext`, and `rfsck-xfs` are publicly available [12].

unreliable versions. Moreover, rfsck-ext outperforms e2fsck-patch by up to nine times while achieving the same level of fault resilience.

This article is an extension of our previous work, Gatla et al. 2018 [43]. In this article, we extend our study to include two additional file system checkers: btrfs-fsck for BtrFS and f2fs-fsck for F2FS file systems. We have added two new methods to generate test images. The first method is flexible in terms of corrupting large area, while the second method is precise for corrupting the metadata regions of the file system. In addition, we have provided a detailed analysis of the root causes for the uncorrectable corruptions using the log information from the fault injection tool. To better understand the functionality of btrfs-fsck and f2fs-fsck, we provide some background information and workflow of these checkers. Other minor changes include a detailed functionality of e2fsck and xfs_repair such as the repairs done by the subpasses, and the like.

The rest of the article is organized as follows. First, we introduce the background of the four file system checkers (Section 2). Next, we describe rfsck-test, study the resilience of the checkers, and provide an analysis of the corruptions (Section 3). We analyze the ordering issue of the undo logging of e2fsck in Section 4. Then, we introduce rfsck-lib and integrate it with existing checkers (Section 5). We evaluate rfsck-ext and rfsck-xfs in Section 6 and discuss several issues in Section 7. Finally, we discuss related work (Section 8) and conclude (Section 9).

2 BACKGROUND

Most file systems employ checkers to check and repair inconsistencies. The checkers are usually file system-specific, and they examine different consistency rules depending on the metadata structures. We use a few representative checkers as concrete examples to illustrate the complexity as well as the potential vulnerabilities of checkers in this section.

2.1 Workflow of e2fsck

e2fsck is the checker of the widely used Ext-family file systems. It first replays the journal (in case of Ext3 and Ext4) and then restarts itself. Next, e2fsck runs the following five passes in order:

Pass-1: Scan the file system and check inodes. In this pass e2fsck iterates over all inodes and checks each inode one by one (e.g., validating the mode, size, and block count). Meanwhile, it stores the scanned information in a set of bitmaps, including inodes in use (inode_used_map), inodes with bad fields (inode_bad_map), inodes in bad blocks (inode_bb_map), blocks in use (block_found_map), and blocks claimed by two inodes (block_dup_map) to generate a list of duplicate blocks and their owners, check the integrity of extent trees, and more.

In addition, e2fsck performs four subpasses: **Pass 1B** scans the data blocks of all inodes again and generates a list of duplicate blocks and their owners; **Pass 1C** traverses the file system tree to determine the parent directories of the inodes reported in Pass 1B; **Pass 1D** is a reconciliation pass, that is, for each inode with duplicate blocks, e2fsck prompts the user to choose whether to clone or delete the file; for each inode with duplicate blocks, **Pass 1D** prompts the user for further actions (e.g., whether to clone or delete the file); **Pass 1E** further checks the integrity of the extent trees.

Pass-2: Check directory structure. Based on the bitmap information, e2fsck iterates through all directory inodes and checks a set of rules for each directory. For example, the first directory entry should be ".", the length of each entry (rec_len) should be within a range, and the inode number in the directory entry should refer to an in-use inode. In some cases, If Pass 2 is unable to add checksum to the directory leaf nodes (due to space requirements), then these directories are rebuilt in Pass 3A. To reduce the disk seek time, the directory entries are processed in the sorted order of block numbers.

Pass-3: Check directory connectivity. In this pass e2fsck ensures the connectivity of all directory inodes. e2fsck first checks if a root directory is available; if not, a new root directory is created and is marked "done." Then it traverses the directory tree until it reaches a directory marked as"done." If no such directory is reached, then the current directory inode is marked as disconnected, and e2fsck offers to reconnect it to the "lost+found" folder. During the traversal, if e2fsck sees a directory twice (i.e., there is a loop) then it offers to reconnect this directory to the "lost+found" folder.

In addition, Pass 3 has a subpass called "Pass 3A," which performs directory optimization (such as indexing directory inodes and removing duplicate entries) by rebuilding the tree directories.

Pass-4: Check reference counts. e2fsck iterates over all inodes to validate the inode link counts. Also, it checks the connectivity of the extended attribute blocks and reconnects them if necessary.

Pass-5: Recalculate checksums and flush updates. Finally, e2fsck checks the repaired in-memory data structures against on-disk data structures and flushes necessary updates to the disk. to ensure consistency. Also, if the in-memory data structures are mapped dirty due to the fixes in the previous passes, the corresponding checksums are recalculated before flushing.

2.2 Workflow of xfs_repair

xfs_repair is the checker of the popular XFS file system.[2] Similar to e2fsck, xfs_repair fixes inconsistencies in seven passes (or phases), including:

Pass-1: Superblock verification. In this phase, xfs_repair performs superblock consistency checks such as file system geometry, total free blocks and free inode count, and any contradiction against information stored on secondary superblocks. xfs_repair uses the information in primary superblock to locate the secondary superblocks. If this information is corrupt, then xfs_repair scans the filesystem to locate the secondary superblocks.

Phase 2: Replay log, validate free and inode maps, and validate root inode. In this phase, xfs_repair initially checks whether the filesystem has an internal or external log. If a log, exists then the updates from the log are applied to the filesystem. Next, the checker validates entries in free map and inode allocation maps. The checker also verifies whether the root inode exists. If it exists, then the checker verifies whether the blocks mapped to the root inode are in use or not. If the root inode does not exist, then xfs_repair allocates the default inode for the root directory.

Phase 3: Process Inodes in each allocation group. Inconsistencies related to inodes such as bad magic number, blocks claimed by inodes, and the like are checked and fixed in this phase. xfs_repair uses multiple threads in this phase to process inodes in each allocation group. If any orphan inodes are found, then xfs_repair stores the information in memory and later links these inodes to the lost+found folder in Phase 6.

Phase 4: Check for duplicate block allocations. In this phase inconsistency related to duplicate blocks and extent allocations is checked. This is done by performing a 2-pass check per inode. In the first pass, xfs_repair checks if an inode conflicts with known duplicate extents. In the second pass the block bitmaps are set for all the blocks claimed by the inode.

Phase 5: Rebuild allocation group data structure and superblock. Based on the information available until this phase, xfs_repair rebuilds the allocation group header information and b-tree

[2]There is another utility called xfs_check [16] which checks consistency without repairing; we do not evaluate it in this work as it is impossible for the read-only utility to introduce additional corruption.

structure. At the end of this phase, the new superblock information, such as free block and inode count, are set in memory.

Phase 6: Check inode connectivity. In this phase `xfs_repair` checks for directory inode connectivity by traversing the incore inode tree structure for each allocation group. This is done by following the pointer information in "." and ".." entries in each directory inode. If any directory is disconnected, then it is linked to the lost+found folder.

Phase 7: Verify and correct link counts. In this pass, the link count of each directory inode is verified and corrected if necessary. Since `xfs_repair` uses multiple threads to fix inconsistencies, it stores all the updates in an in-memory cache and flushes all the updates at the end of the program.

Unlike `e2fsck` which is single-threaded, `xfs_repair` employs multi-threading in passes 2, 3, 6, and 7 to improve performance. Nevertheless, we can see that both checkers are complicated and may be vulnerable to faults. For example, later passes may depend on previous passes, and there is no atomicity guarantee for related updates. We describe our method for systematically exposing these vulnerabilities in Section 3.

2.3 Other File System Checkers

Apart from `e2fsck` and `xfs_repair`, we have also studied two other file system checkers: `btrfs-fsck` for BtrFS and `f2fs-fsck` for F2FS file systems.

The `btrfs-check` is the checker tool for BtrFS, but it does not completely repair the BtrFS file system. Most Linux practitioners suggest users run other tools, such as `btrfs-scrub` and `btrfs-rescue`, for recovery. `btrfs-scrub` can only be used to verify the checksums of all data and metadata blocks. If the tool finds a corrupted block, it can repair it only if a correct copy is available. `btrfs-rescue` is similar to `btrfs-scrub`, but it has three modes of operation: `super-recover`, which recovers corrupted superblock from good copies; `chunk-recover`, which recovers the chunk tree by scanning the entire desk; and `zero-log`, which clears filesystem log tree. We can observe that each tool fixes a specific type of corruption; therefore, in this work, we run all these tools in one script and consider it as `btrfs-fsck`.

The `f2fs-fsck` runs similarly to other checkers: It first scans the file system and then checks the metadata structures, such as journal blocks, Node Address Table (NAT) entries, hard links to files, and more. If the checker is unable to fix the file system, then it creates a `lost+found` folder in the current working directory and moves the files to this folder.

2.4 The Logging Support of Checkers

Some file system developers have envisioned the potential need of reverting changes done to the file system. For example, the "undo io manager" has been added to the utilities of Ext-family file systems since 2007 [4, 17]. It can save the content of the location being overwritten to an undo log before committing the overwrite.

However, due to degraded performance as well as log format issues [3, 18], the undo feature has not been integrated into `e2fsck` until recently. Starting from v1.42.12, `e2fsck` includes a "-z" option to allow the user to specify the path of the log file and enable logging [4]. When enabled, `e2fsck` maintains an undo log during the checking and repairing and writes an undo block to the log before updating any block of the image. If `e2fsck` fails unexpectedly, the undo log can be replayed via `e2undo` [4] to revert the undesired changes.

Given the undo logging, one might expect that an interrupted `e2fsck` will not cause any issues. As we will see in the next section, however, this is not true.

3 ARE THE EXISTING CHECKERS RESILIENT TO FAULTS?

In this section, we first describe our method for analyzing the fault resilience of file system checkers (Sections 3.1–3.3), then present our findings on `e2fsck` (Section 3.4) and `xfs_repair` (Section 3.5), and finally provide an analysis on why corruption persists (Section 3.8).

3.1 Generating Corrupted Test Images

File system checkers are designed to repair corrupted file systems, so the first step of testing checkers is to generate a set of corrupted file system images to trigger the target checker. We call this set of images *test images*. To generate test images, we use the following four methods:

Method 1: Some file system developers may provide test images to perform regression testing of their checkers, which usually cover the most representative corruption scenarios as envisioned by the developers [4]. We collect such default test images to trigger the target checker if they are available.

Method 2: In this approach, we create test images ourselves using the debug tools provided by the file system developers (e.g., `debugfs` [4] and `xfs_db` [16]). These tools allow "trashing" of specific metadata structures with random bits, which may cover corruption scenarios beyond the default test images.

Method 3: In this method, we generate the test images by corrupting a large area at a given offset. This method is feasible when we do not have knowledge of the layout of the file system. For example, the layout of `btrfs` [1] and `f2fs` [55] file systems is not well documented, and there are no debug tools available to modify the metadata. In such cases, corrupting a big chunk of data at a random offset may corrupt some metadata components.

Method 4: This method is an extension of the third. In this method, we identify the offsets of the metadata structures and corrupt them. To achieve this, we used the `mkfs` tool [11] of each file system and the recorder feature in our testing framework. We first run the `mkfs` tool of the target file system on a raw test image and record the I/Os written using the recorder in the testing framework. The recorder maintains a log file which shows the offset and size of each write command. Since `mkfs` tool writes only the metadata components to the test image, these offsets can be effectively used to corrupt the test images. The testing framework is discussed in detail in the following sections.

In all these cases, the test images are generated as regular files instead of real physical disks, which makes the testing more efficient.

3.2 Interrupting Checkers

Generating corrupted test images solves only one part of the problem. Another challenge in evaluating fault resilience is determining how to interrupt checkers in a systematic and controllable way. To this end, we emulate the effect of faults using software.

To make the emulation precise and reasonable, we follow the "clean power fault" model [86], which assumes that there is a minimal atomic unit of write operations (e.g., 512B or 4KB). Under this model, the size of data written to the on-disk file system is always an integer multiple of the minimal atomic block. A fault can occur at any point during the repair procedure of the checker; once a fault happens, all atomic blocks committed before the fault are durable without corruption, and all blocks after the fault have no effect on the media.

Apparently, this is an idealized model under power outages or system crashes. More severe damage (e.g., reordering or corruption of committed blocks) may happen in practice [67, 79, 83, 87, 88]. However, such a clear model can serve as a conservative lower bound of the failure impact.

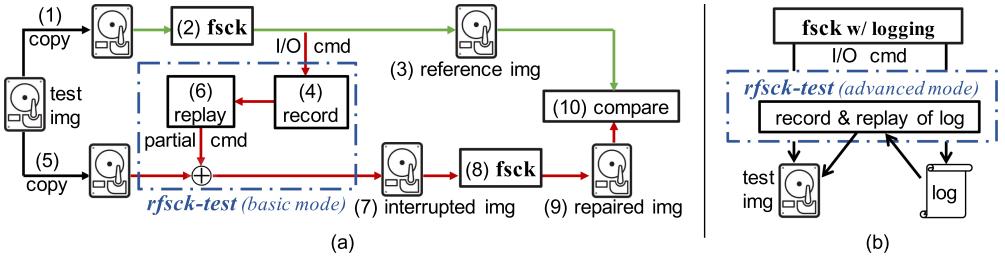

Fig. 1. **(a) Testing the fault resilience of a file system checker (fsck) without logging support.** There are ten steps: (1) make a copy of the test image which contains a corrupted file system; (2) run fsck on the test image copy; (3) store the image generated in step 2 as the reference image; (4) record the I/O commands generated during the fsck; (5) make another copy of the test image; (6) replay partial commands to emulate the effect of an interrupted fsck; (7) store the image generated in step 6 as the interrupted image; (8) run fsck on the interrupted image; (9) store the image generated in step 8 as the repaired image; (10) compare the repaired image with the reference image to identify mismatches. **(b) Testing fsck with logging support.** The workflow is similar except that rfsck-test interrupts the I/O commands sent to both the test image and the log, and the log is replayed between steps 7 and 8.

In other words, file system checkers must be able to handle this fault model gracefully before addressing more aggressive fault models.

Based on the fault model, we build a fault injection tool called `rfsck-test` using a customized driver [9], which has two modes of operation:

Basic mode: This is used for testing a checker *without* logging support. In this mode, the target checker writes to the test image and generates I/O commands through the customized driver. `rfsck-test` records the I/O commands generated during the execution of the checker in a command history file and replays a prefix of the command history (i.e., partial commands) to a copy of the initial test image, which effectively generates the effect of an interrupted checker on the test image. For each command history, we exhaustively replay all possible prefixes and thus generate a set of interrupted images which correspond to injecting faults at different points during the execution of the checker.

Advanced mode: This is used for testing a checker *with* logging support. In this mode, the target checker writes to the test image as well as its log file. `rfsck-test` records the commands sent to both the image and the log in the command history. During the replay, `rfsck-test` selects a prefix of the command history and replays the partial commands either to a copy of the initial test image or to a copy of the initial log, depending on the original destination of the commands. In this way, `rfsck-test` generates the effect of an interrupted checker on both the test image and the log. Moreover, `rfsck-test` replays the log to the test image, which is necessary for the logging to take effect.

3.3 Summary of Testing Framework

Putting it all together, we summarize our framework for testing the fault resilience of checkers with and without logging support as follows:

Testing checkers *without* logging support: As shown in Figure 1(a), there are ten steps: (1) we make a copy of the test image which contains a corrupted file system; (2) the target checker (i.e., `fsck`) is executed to check and repair the original corruption on the copy of the test image; (3) after `fsck` finishes normally in the previous step, the resulting image is stored as the

reference image[3]; (4) during the checking and repairing of `fsck`, the fault injection tool `rfsck-test` operates in the basic mode, which records the I/O commands generated by `fsck` in a command history file; (5) we make another copy of the original test image; (6) `rfsck-test` replays partial commands recorded in Step 4 to the new copy of the test image, which emulates the effect of an interrupted `fsck`; (7) the image generated in Step 6 is stored as the *interrupted image*; (8) `fsck` is executed again on the interrupted image to fix any repairable issues; (9) the image generated in Step 8 is stored as the *repaired image*; (10) finally, we compare the file system on the repaired image with that on the reference image to identify any mismatches.

The comparison in Step 10 is first performed via the `diff` command. If a mismatch is reported, we further verify it manually. Note that in Step 8 we have run `fsck` without interruption, so a mismatch implies that there is some corruption which cannot be recovered by `fsck`.

Testing checkers *with* logging support: The workflow of testing a checker with logging support is similar. As shown in Figure 1(b), `rfsck-test` operates in the advanced mode, which records the I/O commands sent to both the test image and the log and emulates the effect of interruption on both places. Also, between Steps 7 and 8, the (interrupted) log is replayed to the test image to make the logging take effect. The other steps are the same.

3.4 Case Study I: `e2fsck`

In this section, we apply the testing framework to study `e2fsck`. As discussed in Section 2.4, `e2fsck` has recently added undo logging support. For clarity, we name the original version without undo logging as `e2fsck` and the version with undo logging `e2fsck-undo`.

To trigger the checker, we collect 175 Ext4 test images from `e2fsprogs` v1.43.1 [4] as inputs. The sizes of these images range from 8MB to 128MB, and the file system block size is 1KB. To emulate faults on storage systems with different atomic units, we inject faults at two granularities: 512B and 4KB. In other words, we interrupt `e2fsck`/ `e2fsck-undo` after every 512B or 4KB of an I/O transfer command. Since the file system block is 1KB, we do not break file system blocks when injecting faults at the 4KB granularity.

First, we study `e2fsck` using the method in Figure 1(a). As described in Section 3.3, for each fault injected (i.e., each interruption), we run `e2fsck` again and generate one repaired image. Because the repair procedure usually requires updating multiple file system blocks, it can often be interrupted at multiple points depending on the fault injection granularity. Therefore, we usually generate multiple repaired images from one test image.

For example, to fix the test image "`f_dup`" (block claimed by two inodes), `e2fsck` needs to update 16KB in total. At the fault injection granularity of 512B, we generate 32 interrupted images (and consequently 32 repaired images). The last fault is injected after all 16KB blocks, which leads to a repaired image equivalent to the reference image without interruption. Similarly, at the 4KB granularity, we generate 4 repaired images.

For every test image, we generate a number of repaired images and compare each of them with the corresponding reference image. If the comparison reports a mismatch, it implies that the repaired image contains uncorrectable corruption. We count the number of repaired images reporting such corruption. Moreover, if at least one repaired image contains uncorrectable corruption, we mark the test image as reporting corruption, too.

Table 1 summarizes the counts of images in testing `e2fsck` at the two fault injection granularities. The total number of repaired images generated from the 175 Ext4 test images is shown in the third column. We can see that at the 512B granularity there are more repaired images (25,062)

[3]It is possible that a checker may not be able to fully repair a corrupted file system even without interruption [31, 48]. So we simply use the result of an uninterrupted repair as a criterion in this work.

Table 1. **Counts of Images in Testing e2fsck at Two Fault Injection Granularities.** This Table Shows the Number of Repaired Images (3rd Column) Generated from the 175 Ext4 Test Images when Injecting Faults at 512B/4KB Granularities; the Last Two Columns Show the Number of Test Images and Repaired Images Reporting Corruption, Respectively

Fault injection granularity	# of Ext4 test images	# of repaired images generated	# of images reporting corruption	
			test images	repaired images
512 B	175	25,062	34	240
4 KB	175	3,192	17	37

Table 2. **Classification of Corruption.** This Table Shows the Number of Test Images and Repaired Images Reporting Different Corruptions at Two Fault Injection Granularities

Corruption Type	test images		repaired images	
	512 B	4 KB	512 B	4 KB
cannot mount	20	1	41	3
data corruption	9	5	107	10
misplacement	9	11	82	23
others	1	1	10	1

Table 3. **Comparison of e2fsck and e2fsck-undo.** This Table Compares the Number of Test Images Reporting Corruption Under e2fsck and e2fsck-undo

Fault injection granularity	# of images reporting corruption	
	e2fsck	e2fsck-undo
512 B	34	34
4 KB	17	15

because the repairing procedure is interrupted more frequently, while at the 4KB granularity only 3,192 repaired images are generated. Also, more test images report corruption at the 512B granularity (34 > 17). This is because the repair commands are broken into smaller pieces, and thus it is more challenging to maintain consistency when interrupted.

Table 2 further classifies the corruption into four types and shows the number of test images and repaired images reporting each type. Among the four types, *data corruption* (i.e., a file's content is corrupted) and *misplacement* (i.e., a file is either in the"lost+found" folder or completely missing) are the common ones. The most severe corruption is *cannot mount* (i.e., the whole file system volume becomes unmountable). Such corruption has been observed at both fault injection granularities. In other words, interrupting `e2fsck` may lead to an unmountable image, even when a fault cannot break the superblock because the 4KB fault granularity is larger than the 1KB superblock.

Next, to see if undo logging can avoid corruption, we use the method in Figure 1(b) to study `e2fsck-undo`. We focus on the test images which report corruption when testing `e2fsck` (i.e., the 34 and 17 test images in Table 1).

Table 3 compares the number of test images reporting corruption under `e2fsck` and `e2fsck-undo`. Surprisingly, we observe a similar amount of corruption. For example, all 34 images

Table 4. **Counts of Images in Testing xfs_repair at Two Fault Injection Granularities.** This Table Shows the Number of Repaired Images (3rd Column) Generated from the XFS Test Images when Injecting Faults at 512B/4KB Granularities; the Last Two Columns Show the Number of Test Images and Repaired Images Reporting Corruption, Respectively

Fault injection granularity	# of XFS test images	# of repaired images generated	# of images reporting corruption	
			test images	repaired images
512 B	3	1,127	2	443
4 KB	17	1,409	12	737

Table 5. **Counts of Images in Testing btrfs-fsck.** This Table Shows the Number of Repaired Images (3rd Column) Generated from the BtrFS Test Images when Injecting Faults at 4KB Granularity; the Last Two Columns Show the Number of Test Images and Repaired Images Reporting Corruption, Respectively

Fault injection granularity	# of BtrFS test images	# of repaired images generated	# of images reporting corruption	
			test images	repaired images
4 KB	11	234	11	157

which report corruption when testing `e2fsck` at the 512B granularity still report corruption under `e2fsck-undo`. We defer the analysis of the root cause to Section 4.

3.5 Case Study II: `xfs_repair`

We have also applied the testing framework to study `xfs_repair`. Since `xfs_repair` does not support logging, only the method in Figure 1(a) is used.

To generate test images, we create 20 clean XFS images first. Each image is 100MB, and the file system block size is 1KB (same as the Ext4 test images). We use the `blocktrash` command of `xfs_db` [16] to flip 2 random bits on the metadata area of each image. In this way, we generate 20 corrupted XFS test images in total.

Table 4 summarizes the total number of repaired images generated from the XFS test images at two fault injection granularities. We use 3 test images to inject faults at the 512B granularity and 17 images for the 4KB granularity. Similar to the Ext4 case, the smaller granularity (i.e., 512B) leads to more repaired images (i.e., 3 test images lead to 1,127 repaired images). The table also shows the number of test images and repaired images reporting corruption. We can see that there are uncorrectable corruptions under both granularities, as in the Ext4 case.

3.6 Case Study III: `btrfs-fsck`

In addition to `xfs_repair`, we also applied the testing framework to study `btrfs-fsck` and `f2fs-fsck`. In these experiments, we first create clean images of size 128MB for both file systems and fill them using `fs_mark` tool [6]. We then corrupt these test images using Methods 3 and 4, as discussed in Section 3.1. While running the framework, we save all the interrupted images and rerun `btrfs-fsck` to generate the repaired images. Finally, we compare the data on the repaired image with the data in the clean reference image.

Table 5 summarizes the experiment results of BtrFS test images for 4KB fault injection granularity. The repaired images in this case were mountable, and all the user data was consistent. But running the `btrfs-check` reports inconsistency in file system's metadata, and this inconsistency cannot be fixed by any other tools and is therefore permanent.

Table 6. **Counts of Images in Testing f2fs-fsck.** This Table Shows the Number of Repaired Images (3rd Column) Generated from the F2FS Test Images when Injecting Faults at 4KB Granularity; the Last Two Columns Show the Number of Test Images and Repaired Images Reporting Corruption, Respectively

Fault injection granularity	# of F2FS test images	# of repaired images generated	# of images reporting corruption	
			test images	repaired images
4 KB	22	125	3	10

3.7 Case Study IV: `f2fs-fsck`

The `f2fs-fsck` prompts the user to provide an input to perform repairs on the test image, and there are no command line options available to automate this process. Therefore, we have modified the testing framework to manually run the checker and record the I/Os in the background. We then replay the I/Os, save each interrupted image, and then rerun the checker to generate the repaired images.

In our tests we observed that, in most cases ,`f2fs-fsck` is unable to fix the image but it moves all the files to the `lost+found` folder (Section 2.3). The repaired image remains in the corrupted state and cannot be fixed by the checker. But, in some cases, we observe that although the checker prompts the user to move the files to the `lost+found` folder, no such folder is created and the repaired image remains corrupt. We report such images as corrupted. Table 6 shows the fault injection test results on F2FS images. Among the 22 test images, we have corrupted 12 test images using the fourth method. The remaining 10 images were corrupted using the third method, and 3 among these 10 test images have reported corruption.

3.8 Analysis of Uncorrectable Corruption

In this section, we provide an analysis of why these four corruptions are uncorrectable while running `e2fsck`. The fault injection tool `rfsck-test` stores a log that contains the offset and size of each write that is recorded. As mentioned in Section 3.3, we generate one interruption for every partial replay of predetermined number of updates. If the repaired image is corrupted, then we would know what metadata structures were updated and what structures were not updated (or partially updated). For each of the four corruptions, the analysis is as follows:

Cannot Mount: This corruption occurs when the updates to superblock are interrupted. Most of the EXT4 test images in our experiments contain only one group descriptor, and there is no secondary copy of the superblock available. Therefore, another run of the checker cannot verify if the primary superblock is corrupt (or inconsistent), and hence the checker cannot fix this corruption.

Data corruption: This corruption is predominantly observed in test images that have inodes with duplicate block allocations. In our experiments, we observed that `e2fsck` updates the inode and block bitmaps first and then the inode table. If the checker was interrupted in such a way that the inode and block bitmaps are updated while the inode table is still not updated, then this corruption stays uncorrectable.

Missing files: In Section 2.1, we mentioned that `e2fsck` moves any orphaned inodes to the `lost+found` folder. An interruption to this procedure can cause some files to reside in the original directory and others in `lost+found`. In some scenarios, we have also seen that some files were not present on the file system. By analyzing the updates to the image, we observed a contiguous update to metadata structures such as group descriptor, inode and block bitmaps, and the inode table. Any interruption to this contiguous update results in this type of uncorrectable corruption. The

```
1.  /*open undo log*/
2.  undo_open(...){
3.      open(...); /*no O_SYNC*/
4.  }
5.  ...
6.  /*fix 1st inconsistency*/
7.  undo_write_blk64(...){
8.      /*write to undo log asynchronously*/
9.      undo_write_tdb(...){
10.         ...
11.         pwrite(...); /*no fsync()*/
12.     }
13.     /*write to fs image asynchronously*/
14.     io_channel_write_blk64(...){...}
15. }
16. /*fix 2nd, 3rd, ... inconsistencies*/
17. ...
18.
19. /*sync buffered writes to fs image*/
20. ext2fs_flush(...){...}
21. /*close undo log*/
22. undo_close(...){...}
```

Fig. 2. **Workflow of the undo logging in e2fsck-undo.** The writes to the log (Lines 9–12) and the writes to the file system image (Line 14) are asynchronous, and there is no ordering guarantee between the writes.

initial corruption scenarios that report this corruption include bad or no root directory, orphaned indirect inodes, and the like.

Others: In this case, we observed corruption in uncorrectable inodes. The procedure in which these corruptions appear is smilar to the data corruption scenario (i.e., the inode and block bitmaps are updated but the inode table is still not updated due to the interruption of the checker).

4 WHY DOES THE EXISTING UNDO LOGGING NOT WORK?

The study in Section 3 shows that even the checkers of some of the most popular file systems are not resilient to faults. This is consistent with other studies on the catastrophic failures of real-world systems [47, 50], which find that the recovery procedures themselves are often imperfect and that sometimes "the cure is worse than the disease" [50].

One way to handle the faults and provide crash consistency is Write-Ahead Logging (WAL) [64], which has been widely used in databases [14] and journaling file systems [78] for transactional recovery. While it is perhaps not surprising that file system checkers without crash consistency support (e.g., `e2fsck` and `xfs_repair`) may introduce additional corruptions upon interruption, it is counterintuitive that `e2fsck-undo`, which has undo logging support, still cannot prevent cascading damage.

To understand the root cause, we analyzed the source code of `e2fsck-undo` as well as the runtime traces (e.g., system calls and I/O commands) and found that there is no ordering guarantee between the writes to the undo log and the writes to the image being fixed, which essentially invalidates the WAL mechanism.

To better illustrate the issue, Figure 2 shows a simplified workflow of undo logging in `e2fsck-undo`. At the beginning of checking (Lines 2–4), the undo log file is opened without the `O_SYNC` flag. To fix an inconsistency, `e2fsck-undo` first gets the original content of the block being repaired (not shown) and then writes it as an undo block to the log *asynchronously* (Lines 9–12). After the write to the log, it updates the file system image asynchronously (Line 14). The

same pattern (i.e., locate the block that needs to be repaired, copy the old content to the log, and update the file system image) is repeated to fix every inconsistency. At the end, `e2fsck-undo` flushes all buffered writes of the image to the persistent storage (Line 20) and closes the undo log (Line 22).

While the extensive asynchronous writes (Lines 6–17) are good for performance, they are problematic from the WAL's perspective. All asynchronous writes are buffered in memory. Since the dirty pages may be flushed by kernel threads due to memory pressure or timer expiry, or by the internal flushing routine of the host file system, there is no strict ordering guarantee among the buffered writes. In other words, for every single fix, the writes to the log and the writes to the file system image may reach the persistent storage in an arbitrary order. Consequently, when `e2fsck-undo` is interrupted, the file system image may have been modified without the appropriate undo blocks recorded. Because the WAL mechanism works only if a log block reaches persistent storage *before* the updated data block it describes, the lack of ordering guarantee between the writes to the log and the writes to the image invalidates the WAL mechanism. As a result, the existing undo logging does not work as expected.

5 ROBUST FILE SYSTEM CHECKERS

In this section, we describe our method to address the problem exposed in Sections 3 and 4.

First, we fix the ordering issue of `e2fsck-undo` by enforcing necessary sync operations. For clarity, we name the version with our patch `e2fsck-patch`.

Next, we observe that although `e2fsck-patch` may provide the desired robustness, it inherently requires extensive synchronized I/O, which may hurt performance severely. To address this limitation, and to provide a generic solution to the checkers of other file systems, we design and implement `rfsck-lib`, a general logging library with a simple interface. Different from `e2fsck-patch` which interleaves the writes to the log (i.e., *log writes*) and the writes to the image being repaired (i.e., *repair writes*), `rfsck-lib` makes use of the similarities among checkers to decouple logging from the repairing of the file system and provides fine-grained control of logging.

To demonstrate its practicality, we use `rfsck-lib` to strengthen existing checkers and create two new checkers: `rfsck-ext`, a robust checker for Ext-series file systems, and `rfsck-xfs`, a robust checker for XFS file systems, both of which require only a few lines of modification to the original versions.

5.1 Goals

While there are many desired objectives, `rfsck-lib` is designed to meet the following three key goals:

Robustness: Unlike existing checkers that may introduce uncorrectable corruptions when interrupted, we expect checkers integrated with `rfsck-lib` to be resilient to faults. We believe such robustness should be of prime concern for file system practitioners in addition to the heavily studied performance issue [60].

Performance: Guaranteeing robustness may come at the cost of performance because it almost inevitably requires additional operations. However, the performance overhead should be reduced to minimum, which is particularly important for production environments.

Compatibility: We expect `rfsck-lib` to be compatible to existing file systems and checkers. For example, no change to the existing on-disk layouts or repair rules is needed. While such compatibility may sacrifice some flexibility and optimization opportunities, it directly enables improving the robustness of many widely used systems in practice.

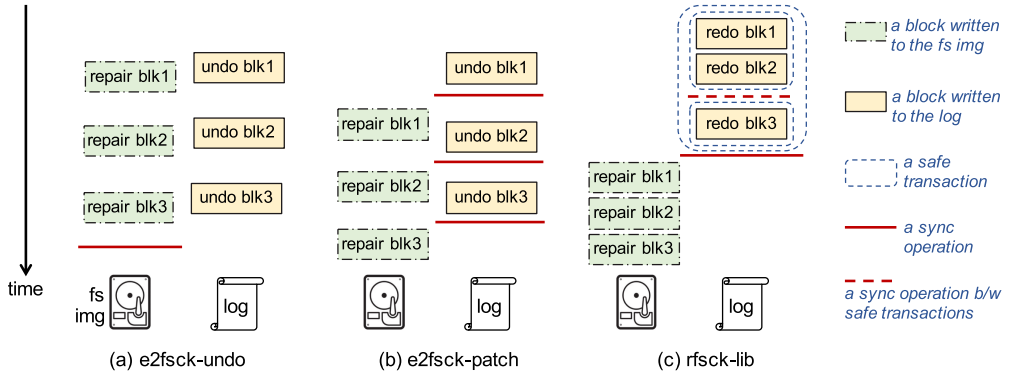

Fig. 3. **Comparison of different logging schemes.** This figure compares different logging schemes using a sequence of blocks written to the file system image (i.e., "fs img") and the log: (a) e2fsck-undo is the logging scheme of e2fsck, which does not have the necessary ordering guarantee between the writes to the log and the writes to the file system image; (b) e2fsck-patch guarantees the correct ordering between each undo block (e.g., "undo blk1") and the corresponding repair block (e.g., "repair blk1") by enforcing a sync operation (i.e., the red line) after each write of an undo block; (c) rfsck-lib uses redo logging to eliminate the frequent sync required in e2fsck-patch and only syncs after a safe transaction which includes a set of blocks constituting a consistent update.

5.2 e2fsck-patch: Fixing the Ordering Issue in e2fsck-undo

As discussed in Section 4, `e2fsck-undo` does not guarantee the necessary ordering between log writes and repair writes. Figure 3(a) illustrates the scenario using a sequence of writes. In this example, three blocks are written to the file system image (i.e., "fs img") to repair inconsistencies (i.e., "repair blk1" to "repair blk3"). Meanwhile, three blocks are written to the undo log (i.e., "undo blk1" to "undo blk3") to save the original content of the blocks being overwritten for the purpose of undoing changes in case the repair fails. Because all blocks are written asynchronously, the repair blocks may reach persistent storage *before* the corresponding undo blocks, which essentially invalidates the undo logging scheme. Although there is a sync operation at the end to the file system image (i.e., the red solid line), it cannot prevent the previous buffered blocks from reaching the persistent storage out of the desired order.

A naive way to solve the issue is to use a synchronous write for each block. However, this is overkill. As long as an undo block (e.g., "undo blk1") becomes persistent , it is unnecessary for the corresponding repair block (e.g., "repair blk1") to be written synchronously. Therefore, we only enforce synchronized I/O for the undo log file.

Specifically, we add the `O_SYNC` flag when opening the undo log file, which is equivalent to adding an `fsync` call after each write to the log [8]. As shown in Figure 3(b), the simple patch guarantees that a repair block is always written *after* the corresponding undo block becomes persistent. On the other hand, all repair blocks are still written asynchronously. In this way, `e2fsck-patch` fixes `e2fsck-undo` with minimum modification.

5.3 rfsck-lib: A General Library for Strengthening File System Checkers

While the logging scheme of `e2fsck-patch` may improve fault resilience, it has two limitations. First, the log writes and the repair writes are interleaved. Consequently, it requires extensive synchronized I/O to maintain the correct ordering (e.g., three sync operations are required in Figure 3(b)), which may incur severe performance overhead. Second, as part of `e2fsck`, the logging

feature is closely tied to Ext-family file systems, and thus it cannot benefit other file system checkers directly. We address these limitations by building a general logging library called `rfsck-lib`.

Similarities Among File System Checkers: Different file systems usually vary a lot in terms of on-disk layouts and consistency rules. However, there are similarities among different checkers, which makes designing a general and efficient solution possible.

First of all, as a user-level utility, file system checkers always repair corrupted images through a limited number of system calls, which are irrelevant to file systems' internal structures and consistency rules. Moreover, based on our survey of popular file system checkers (e.g., `e2fsck`, `xfs_repair`, `fsck.f2fs`), we find that they always use write system calls (e.g., `pwrite` and its variants) instead of other memory-based system calls (e.g., `mmap`, `msync`). Therefore, only a few writes may cause potential cascading corruptions under faults. In other words, by focusing on the writes, we may improve different checkers.

Second, there is natural locality in checkers. Similar to the cylinder groups of FFS [62], many modern file systems have a layout consisting of relatively independent areas with an identical structure (e.g., block groups of Ext4 [5], allocation groups of XFS [76], and cubes of IceFS [58]). Among others, such common design enables co-locating related files to mitigate file system aging [37, 74] while isolating unrelated files. From the checker's perspective, most consistency rules within each area may be checked locally without referencing other areas. Also, each type of metadata usually has its unique structure and consistency rules (e.g., the `rec_len` of each directory entry in an Ext4 inode should be within a range). These local consistency rules may be checked independently without cross-checking other metadata.

Due to locality, checkers usually consist of relatively self-contained components. For example, `e2fsck` includes five passes for checking different sets of consistency rules (Section 2.1). Similarly, `xfs_repair` includes seven passes, and it forks multiple threads to check multiple allocation groups separately (Section 2.2). Such locality exists even without changing the file system layout or reordering the checking of consistency rules [60]. Therefore, it is possible to split an existing checker into several pieces and isolate their impact under faults.

Based on these observations, we describe `rfsck-lib`'s design in the following subsections.

Basic Redo Logging: A corrupted file system image is repaired by a checker through a set of repair writes. If the checker finishes without interruption, the set of writes turns the image back to a consistent state. On the other hand, if the checker is interrupted, only a subset of writes changes the image, and the resulting state may become uncorrectable. Therefore, the key to preventing uncorrectable states is to maintain the atomicity of the checker's writes. To this end, `rfsck-lib` redirects the checker's writes to a log first and then repairs the image based on the log. Essentially, it implements a redo logging scheme [64].

As shown in Figure 3(c), all repair writes are issued to the redo log first (i.e., "redo blk1" to "redo blk3"). After the write of the last redo block (i.e., "redo blk3"), a sync operation (i.e., the red solid line) is issued to make the redo blocks persistent. After the sync operation returns, the image is repaired (i.e., "repair blk1" to "repair blk3") based on the redo log. Compared with `e2fsck-patch` in Figure 3(b), the log writes and the repair writes are separated, and the required number of sync operations is reduced from three to one. Such improvement in terms of sync overhead can be more significant if more blocks on the image need to be repaired.

Fine-Grained Logging with Safe Transactions: While the basic redo logging scheme reduces the ordering constraint to minimum, there is one limitation: If a fault happens before the final sync operation finishes, all checking and repairing efforts may be lost. In some complicated cases where the checker may take hours to finish [60], the waste is undesirable. On the other hand, a checker

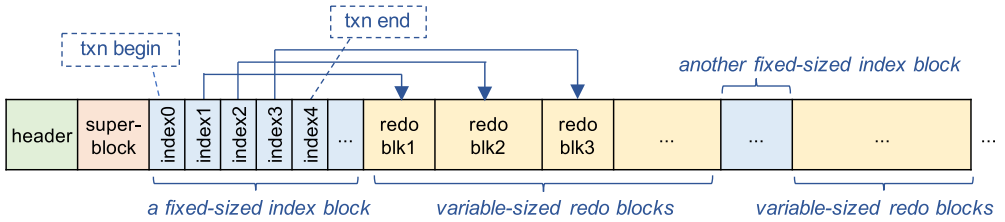

Fig. 4. **The log format of rfsck-lib.** The log includes a header, a superblock, fixed-sized index blocks, and variable-sized redo blocks. Each index block includes a fixed number of indexes. Each index can either describe the beginning/end of a transaction (i.e., "txn begin"/"txn end"), or describe one variable-sized redo block. "index0" to "index4" describe one safe transaction with three redo blocks (i.e.,"redo blk1" to "redo blk3") in this example.

may be split into relatively independent pieces due to locality. Therefore, `rfsck-lib` extends the basic redo logging with safe transactions.

A *safe transaction* is a set of repair writes which will not lead to uncorrectable inconsistencies if they are written to the file system image atomically. In the simplest case, the whole checker (i.e., the complete set of all repair writes) is one safe transaction. At a finer granularity, each pass of the checker (or the check of each allocation group) may be considered as one safe transaction. While a later pass may depend on the result of a previous pass, the previous pass is executed without any dependency on the later passes. Therefore, by guaranteeing the atomicity of each pass as well as the ordering among pass-based safe transactions, the repair writes may be committed in several batches without introducing uncorrectable inconsistencies. In the extreme case, the checking and repairing of each individual consistency rule may be considered as one safe transaction.

Figure 3(c) illustrates the safe transactions. In the simplest case, all three redo blocks (i.e., "redo blk1" to "redo blk3") constitute one safe transaction, and only one sync operation (i.e., the red solid line) is needed, as in the basic redo logging. At a finer granularity, the first two redo blocks (i.e., "redo blk1" and "redo blk2") may constitute one safe transaction (e.g., updating an inode and the corresponding bitmap), and the third block itself (i.e., "redo blk3") may be another safe transaction (e.g., updating another inode). Another sync operation (i.e., the red dash line) is issued between the two transactions to guarantee the correct ordering. If a crash happens between the two sync operations, the first safe transaction (i.e., "redo blk1" and "redo blk2") is still valid. In this case, instead of recalculating the rules and regenerating the blocks, the checker can directly replay the valid transaction from the log after restart.

In summary, a checker may be logged as one or more safe transactions. Compared to the basic redo logging, such fine-grained control avoids losing all recovery efforts before the fault. On the other hand, maintaining the atomicity as well as the ordering requires additional sync operations. So there is a tradeoff between transaction granularity and transaction overhead. Since different systems may have different preferences, `rfsck-lib` simply provides an interface to define safe transactions, without restricting the number of transactions.

Log Format: To support the redo logging with safe transactions, `rfsck-lib` uses a special log format extended from `e2fsck-undo`. As shown in Figure 4, the log includes a header, a superblock, fixed-sized index blocks, and variable-sized redo blocks.

The header starts with a magic number to distinguish the log from other files. In addition, it includes the offsets of the superblock and the first index block, the total number of index blocks, a flag showing whether the log has been replayed, and a checksum of the header itself.

The superblock is copied from the file system image to be repaired, which is used to match the log with the image to avoid replaying an irrelevant log to the image.

Table 7. The Structure of an Index

Field	Description
`uint32_t cksum`	checksum of the redo block
`uint32_t size`	size of the redo block
`uint64_t fs_lba`	LBA in the file system image

The index block includes a fixed number of indexes. Each index can describe the beginning of a transaction (i.e., "txn begin"), the end of a transaction (i.e., "txn end"), or one variable-sized redo block. Therefore, a group of indexes can describe one safe transaction together. For example, in Figure 4, five indexes (i.e., "index0" to "index4") describe one safe transaction with three redo blocks (i.e.,"redo blk1" to "redo blk3").

As shown in Table 7, an index has 16 bytes consisting of three fields. To describe one redo block, the first field (i.e., `uint32_t cksum`) stores a checksum of the redo block, the second field (i.e., `uint32_t size`) stores its size, and the third field (i.e., `uint64_t fs_lba`) stores its logical block address (LBA) in the file system image.

To describe "txn begin" or "txn end," the first field of the index is repurposed to store a transaction ID instead of a checksum, which marks the boundary of indexes belonging to the same transaction. The second field (`size`) is set to zero. Since a valid redo block must have a non-zero size, `rfsck-lib` can differentiate "txn begin" or "txn end" indexes from those describing redo blocks even if a transaction ID happens to collide with a checksum. In addition, the "txn begin" index uses the third field to denote whether the transaction has been replayed or not, and the "txn end" index uses the third field to store a checksum of all indexes in the transaction.

For each write of the checker, `rfsck-lib` creates an index in the index block and then appends the content of the write to the area after the index block as a redo block. Since the writes may have different sizes, the redo blocks may vary in size as well. However, since all other metadata blocks (i.e., header, superblock, index blocks) have known fixed sizes, the offset of a redo block in the log can be identified by accumulating the sizes of all previous blocks. In other words, there is no need to maintain the offsets of redo blocks in the log.

When an index block becomes full, another index block is allocated after the previous redo blocks (which are described by the previous index block). In this way, `rfsck-lib` can support various numbers of writes and transactions.

Interface: To enable easy integration with existing checkers, `rfsck-lib` provides a simple interface. As shown in Table 8, there are seven function calls in total. The first function (`rfsck_get_sb`) is a wrapper for invoking a file system-specific procedure to get the superblock, which is written to the second part of the log (Figure 4). Since all checkers need to read the superblock anyway, `rfsck_get_sb` can wrap around the existing procedure.

`rfsck_open` is used to create a log file at a given path at the beginning of the checker procedure. Internally, `rfsck-lib` initializes the metadata blocks of the log.

`rfsck_txn_begin` is used to denote the beginning of a safe transaction, which creates a "txn begin" index in the log. Similarly, `rfsck_txn_end` denotes the end of a transaction, which generates a "txn end" index and syncs all updates to the log. All writes between `rfsck_txn_begin` and `rfsck_txn_end` are replaced with `rfsck_write`, which creates a redo block and the corresponding index in the log.

`rfsck_replay` is used to replay logged transactions to the file system image. In addition, similar to the `e2undo` utility [4], the replay functionality is also implemented as an independent utility called `rfsck-redo`, which can replay an existing (potentially incomplete) log to a file system image

Table 8. **The Interface of rfsck-lib**. rfsck_get_sb Is a Wrapper Function for Invoking File System-specific Procedure to Get the Superblock, While the Others Are File-system Agnostic

Function	Description
`rfsck_get_sb`	get the superblock
`rfsck_open`	create a redo log
`rfsck_txn_begin`	begin a safe transaction
`rfsck_write`	write a redo block
`rfsck_txn_end`	end of a safe transaction
`rfsck_replay`	replay the redo log
`rfsck_close`	close the redo log

(e.g., after the checker is interrupted). `rfsck-redo` first verifies if the log belongs to the image (based on the superblock). If yes, `rfsck-redo` further verifies the integrity of the log based on metadata and then replays valid transactions. Note that the additional verifications are only needed when the log is replayed by `rfsck-redo`. The `rfsck_replay` function can skip these verifications as it is invoked directly after the logging by the (uninterrupted) checker.

Finally, `rfsck_close` is used at the end of the checker to release all resources used by `rfsck-lib` and exist.

Limitations: The current prototype of `rfsck-lib` is not thread-safe. Therefore, if a checker is multi-threaded (e.g., `xfs_repair`), using `rfsck-lib` may require additional attention to avoid race conditions on logging. However, as we will demonstrate in Sections 5.4 and 6, `rfsck-lib` can still be applied to strengthen `xfs_repair`.

In addition, `rfsck-lib` only provides an interface, which requires manual modification of the source code. Since the modification is simple, we expect the manual effort to be acceptable. Also, it is possible to use compiler infrastructures [13, 15] to automate the code instrumentation, which we leave as future work.

5.4 Integration with Existing Checkers

Strengthening an existing checker using `rfsck-lib` is straightforward given the simple interface. To demonstrate its practicality, we first integrate `rfsck-lib` with `e2fsck` and create a robust checker for Ext-family file systems (i.e., `rfsck-ext`).

There are potential writes to the file system image in each pass of `e2fsck` (including the first scanning pass), so we create a safe transaction for each pass. Moreover, within Pass-1 and Pass-2 (Section 2.1), there are a few places where `e2fsck` explicitly flushes the writes to the image and restarts scanning from the beginning (via `goto` statement). In other words, the restarted scanning (and subsequent passes) requires the previous writes to be visible on the image. In this case, we insert additional `rfsck_txn_end` and `rfsck_replay` before the `goto` statement to guarantee that previous writes are visible on the image for rescanning. We add an "-R" option to allow the user to specify the log path via command line. In total, we add 50 lines of code to `e2fsck`.

Similarly, we also integrate `rfsck-lib` with `xfs_repair`, and create a robust checker for XFS file system (i.e., `rfsck-xfs`). As mentioned in Section 2.2, one feature of `xfs_repair` is multi-threading: It forks multiple threads to repair multiple allocation groups in parallel. The threads update in-memory structures concurrently, and the main thread writes all updates to the image at the end. Although it is possible to encapsulate each repair thread into one safe transaction,

doing so requires additional concurrency control. To minimize the modification, we simply treat the whole repair procedure as one transaction. Since all writes are issued by the main thread, there is no race condition for `rfsck-lib`. We also add an "-R" command line option. In total, we add 15 lines of code to `xfs_repair`.

6 EVALUATION

In this section, we evaluate `rfsck-ext` and `rfsck-xfs` in terms of robustness (Section 6.1) and performance (Section 6.2). Our experiments were conducted on a machine with an Intel Xeon 3.00GHz CPU, 8GB main memory, and two WD5000AAKS hard disks. The operating system is Ubuntu 16.04 LTS with kernel v4.4. To evaluate the robustness, we used the test images reporting corruption under `e2fsck-undo` (Section 3.4) and `xfs_repair` (Section 3.5). To evaluate performance, we created another set of images with practical sizes and measured the execution time of `e2fsck`, `e2fsck-undo`, `e2fsck-patch`, `rfsck-ext`, `xfs_repair`, and `rfsck-xfs`. For each checker, we report the average time of three runs.

In general, we demonstrate that `rfsck-ext` and `rfsck-xfs` can survive fault injection experiments. Also, both checkers incur reasonable performance overhead (i.e., up to 12%) compared to the original unreliable versions. Moreover, `rfsck-ext` outperforms `e2fsck-patch` by up to 9 times while achieving the same level of robustness.

6.1 Robustness

As discussed in Section 3, when injecting faults at the 4KB granularity, 17 Ext4 test images report corruptions under `e2fsck` (Table 1) and 12 XFS test images report corruptions under `xfs_repair` (Table 4). We use these test images to trigger `rfsck-ext` and `rfsck-xfs`, respectively. Since both checkers have logging support, we use the method in Figure 1(b) to evaluate them.

For `rfsck-ext`, all 17 test images report no corruptions. Similarly, for `rfsck-xfs`, all 12 test images report no corruptions. This result verifies that `rfsck-lib` can help improve the fault resilience of existing checkers.

6.2 Performance

The test images used in Section 3 are created as regular files, and they are small in size (i.e., 8MB to 128MB). Therefore, they are unsuitable for evaluating the execution time of checkers. So we create another set of Ext4 and XFS test images with practical sizes (i.e., 100G, 200GB, 500GB) on real hard disks. We first fill up the entire file system by running `fs_mark` [6] five times. Each time, `fs_mark` fills up 20% of the capacity by creating directories and files of a certain size. The file size is a random value between 4KB to 1MB, which is relatively small in order to maximize the number of inodes used. After filling up the entire file system, we inject 2 random bit corruptions to the metadata using either `debugfs` [2] (for Ext4) or `blocktrash` [16] (for XFS). We measure the execution time of checkers on corrupted images and verify that the repair results of `rfsck-ext` and `rfsck-xfs` are the same as that of the original checkers.

Figure 5(a) compares the execution time of `e2fsck`, `e2fsck-undo`, `e2fsck-patch`, and `rfsck-ext` on different images. For each size of image, the bars represent the execution time in seconds (y-axis). Also, the number above each bar shows the normalized execution time (relative to `e2fsck`). We can see that `rfsck-ext` incurs up to 12% overhead, while `e2fsck-patch` may incur more than 8 times the overhead due to extensive sync operations.

Also, we can see that, as the size of file system increases, the overhead of `rfsck-ext` decreases. This is because the execution time of `rfsck-ext` is largely dominated by the scanning in Pass-1 (Section 2.1), which is proportional to the file system size, similar to `e2fsck` [60].

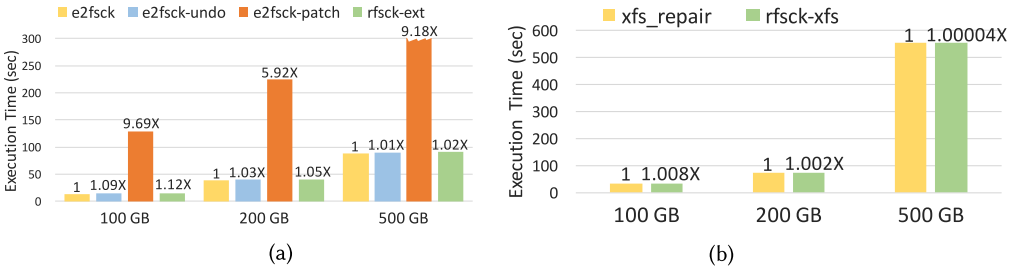

Fig. 5. **Performance comparison of various checkers.** (a) Compares the execution time of e2fsck, e2fsck-undo, e2fsck-patch, and rfsck-ext. The y-axis shows the execution time in seconds. The x-axis shows file system sizes. Similarly, (b) compares the execution time of xfs_repair and rfsck-xfs. The number above each bar indicates the normalized time relative to e2fsck and xfs_repair, respectively.

Similarly, Figure 5(b) compares the execution time of `xfs_repair` and `rfsck-xfs`. We can see that `rfsck-xfs` incurs up to 0.8% overhead, and the overhead also decreases as the file system size increases.

Note that our aging method is relatively simple compared to other aging techniques [37, 74]. Also, the 2-random bit corruption may not necessarily lead to extensive repair operations by checkers. Therefore, the execution time measured here may not reflect the complexity of checking and repairing real-world file systems (which may take hours [38, 39, 60, 75]). We leave generating more representative file systems as future work.

7 DISCUSSION

Co-designing file systems and checkers. Recent work has demonstrated the benefits of co-designing file systems and checkers. For example, by co-designing `rext3` and `ffsck`, `ffsck` may be 10 times faster than `e2fsck` [60]. In contrast, `rfsck-lib` is designed to be file system agnostic, which makes it directly applicable to existing systems. We believe checkers may be improved further in terms of both reliability and performance by co-designing, and we leave it as future work.

Other reliability techniques. There are other techniques which may mitigate the impact of an inconsistent file system image or the loss of an entire image (e.g., replication [45]). However, maintaining the consistency of local file systems and improving checkers is still important for many reasons. For example, a consistent local file system is the building block of large-scale file systems, and the local checker may be the foundation of higher level recovery procedures (e.g., `lfsck` [7]). Therefore, our work is orthogonal to these other efforts.

Robustness. We evaluate the robustness of checkers based on fault injection experiments in this work. The test images we use are limited and may not cover all corruption scenarios or trigger all code paths of the checkers. There are other techniques (e.g., symbolic execution and formal verification) which might provide more coverage, and we leave it as future work.

8 RELATED WORK

Reliability of file system checkers. Gunawi et al. [48] find that the Ext2 checker may create inconsistent or even insecure repairs; they then propose a more elegant checker (i.e., `SQCK`) based on a declarative query language. Carreira et al. [31] propose a tool (i.e., `SWIFT`) to test checkers using a mix of symbolic and concrete execution; they tested five popular checkers and found bugs in all of them. Ma et al. [60] change the structure of Ext3 and co-design the checker, which enables

faster checking and thus narrows the window of vulnerability. Generally, these studies consider the behavior of checkers during normal executions (i.e., no interruption). Complimentarily, we study checkers under faults. An early version of our work [44] studies two file system checkers without improving their robustness, while in this version we provide a practical solution to enhance the checkers.

Reliability of file systems. Great efforts have been made toward improving the reliability of file systems [25, 29, 30, 33, 36, 40, 49, 57, 61, 63, 68, 73, 83, 85]. For example, Prabhakaran et al. [68] analyze the failure policies of four file systems and propose the IRON file system, which implements a family of novel recovery techniques. Fryer et al. [40] transform global consistency rules to local consistency invariants and enable fast runtime checking. CrashMonkey [61] provides a framework to automatically test the crash consistency of file systems. Overall, these research efforts help understand and improve the reliability of file systems, which may reduce the need for checkers. However, despite these efforts, checkers remain a necessary component for most file systems.

Reliability of storage devices. In terms of storage devices, research efforts are also abundant [21, 22, 34, 53, 69, 70]. For example, Bairavasundaram et al. [21, 22] analyze the data corruption and latent sector errors in production systems containing a total of 1.53 million HDDs. Besides HDDs, more recent work has been focused on flash memory and Solid State Drives (SSDs) [20, 24, 26–28, 32, 41, 46, 52, 54, 56, 59, 65, 71, 77, 79, 82, 84, 87, 88]. These studies provide valuable insights for understanding file system corruptions caused by hardware.

9 CONCLUSION

We studied the behavior of various file system checkers under faults. We find that running the checker after an interrupted repair may not return the file system to a valid state. To address the issue, we built a general logging library which can help strengthen existing checkers with little modification. We hope our work will raise the awareness of reliability vulnerabilities in storage systems, and facilitate building truly fault-resilient systems.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful feedback, and the Linux practitioners including Theodore Ts'o, Christoph Hellwig, and Ric Wheeler, for the invaluable discussion at the Linux FAST Summit. This work was supported in part by National Science Foundation (NSF) under grants CNS-1566554 and CCF-1717630. Any opinions, findings, and conclusions expressed in this material are those of the authors and do not necessarily reflect the views of NSF.

REFERENCES

[1] Btrfs File System. n.d. https://btrfs.wiki.kernel.org/index.php/Main_Page.
[2] debugfs. n.d. http://man7.org/linux/man-pages/man8/debugfs.8.html.
[3] Discussion with Theodore Ts'o at Linux FAST Summit'17. n.d. https://www.usenix.org/conference/linuxfastsummit17.
[4] E2fsprogs: Ext2/3/4 Filesystem Utilities. n.d. http://e2fsprogs.sourceforge.net/.
[5] Ext4 File System. n.d. https://ext4.wiki.kernel.org/index.php/Main_Page.
[6] fs_mark: Benchmark file creation. n.d. https://github.com/josefbacik/fs_mark.
[7] LFSCK: an online file system checker for Lustre. n.d. https://github.com/Xyratex/lustre-stable/blob/master/Documentation/lfsck.txt.
[8] Linux Programmer's Manual: O_SYNC flag for open. n.d. http://man7.org/linux/man-pages/man2/open.2.html.
[9] Linux SCSI target framework (tgt). n.d. http://stgt.sourceforge.net/.
[10] Lustre File System. n.d. http://opensfs.org/lustre/.
[11] mkfs. n.d. https://linux.die.net/man/8/mkfs.

[12] Prototypes of rfsck-test, e2fsck-patch, refsck-lib, refsck-ext, rfsck-xfs. n.d. https://www.cs.nmsu.edu/ mzheng/lab/lab.html.
[13] ROSE Compiler Infrastructure. n.d. http://rosecompiler.org/.
[14] SQLite documents. n.d. http://www.sqlite.org/docs.html.
[15] The LLVM Compiler Infrastructure. n.d. https://llvm.org/.
[16] XFS File System Utilities. n.d. https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Storage_Administration_Guide/xfsothers.html.
[17] [PATCH 1/3] e2fsprogs: Add undo I/O manager. 2007. http://lists.openwall.net/linux-ext4/2007/07/25/2.
[18] [PATCH 16/31] e2undo: ditch tdb file, write everything to a flat file. 2015. http://lists.openwall.net/linux-ext4/2015/01/08/1.
[19] High Performance Computing Center (HPCC) Power Outage Event. Email Announcement by HPCC, Monday, January 11, 2016 at 8:50:17 AM CST. 2016. https://www.cs.nmsu.edu/ mzheng/docs/failures/2016-hpcc-outage.pdf.
[20] Nitin Agarwal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. 2008. Design Tradeoffs for SSD Performance. In *Proceedings of the 2008 USENIX Annual Technical Conference (USENIX ATC'08)*, Vol 57.
[21] Lakshmi N. Bairavasundaram, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Garth R. Goodson, and Bianca Schroeder. 2008. An analysis of data corruption in the storage stack. *ACM Transactions on Storage* 4, 3 (Nov. 2008), 8:1–8:28.
[22] Lakshmi N. Bairavasundaram, Garth R. Goodson, Shankar Pasupathy, and Jiri Schindler. 2007. An analysis of latent sector errors in disk drives. In *Proceedings of the 2007 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'07)*. ACM, 289–300.
[23] Luiz Andre Barroso and Urs Hoelzle. 2009. *The Datacenter As a Computer: An Introduction to the Design of Warehouse-Scale Machines* (1st ed.). Morgan and Claypool Publishers.
[24] Hanmant P. Belgal, Nick Righos, Ivan Kalastirsky, Jeff J. Peterson, Robert Shiner, and Neal Mielke. 2002. A new reliability model for post-cycling charge retention of flash memories. In *Proceedings of the 40th Annual Reliability Physics Symposium*. IEEE, 7–20.
[25] James Bornholt, Antoine Kaufmann, Jialin Li, Arvind Krishnamurthy, Emina Torlak, and Xi Wang. 2016. Specifying and checking file system crash-consistency models. *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'16)* 51, 4 (2016), 83–98.
[26] Adam Brand, Ken Wu, Sam Pan, and David Chin. 1993. Novel read disturb failure mechanism induced by FLASH cycling. In *Proceedings of the 31st Annual Reliability Physics Symposium*. IEEE, 127–132.
[27] Yu Cai, Erich F. Haratsch, Onur Mutlu, and Ken Mai. 2012. Error patterns in MLC NAND flash memory: Measurement, characterization, and analysis. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'12)*. EDA Consortium, Dresden, 521–526.
[28] Yu Cai, Gulay Yalcin, Onur Mutlu, Erich F. Haratsch, Osman Unsal, Adrian Cristal, and Ken Mai. 2014. Neighbor-cell assisted error correction for MLC NAND flash memories. In *ACM SIGMETRICS Performance Evaluation Review*, Vol. 42. ACM, 491–504.
[29] Jinrui Cao, Om Rameshwar Gatla, Mai Zheng, Dong Dai, Vidya Eswarappa, Yan Mu, and Yong Chen. 2018. PFault: A general framework for analyzing the reliability of high-performance parallel file systems. In *Proceedings of the 32nd ACM International Conference on Supercomputing (ICS'18)*. 1–11.
[30] Jinrui Cao, Simeng Wang, Dong Dai, Mai Zheng, and Yong Chen. 2016. A generic framework for testing parallel file systems. In *Proceedings of the 1st Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems (PDSW-DISCS'16)*. 49–54.
[31] João Carlos Menezes Carreira, Rodrigo Rodrigues, George Candea, and Rupak Majumdar. 2012. Scalable testing of file system checkers. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys'12)*. ACM, 239–252.
[32] Feng Chen, David A. Koufaty, and Xiaodong Zhang. 2009. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *Proceedings of the ACM Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'09)*.
[33] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2015. Using crash hoare logic for certifying the FSCQ file system. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)*. ACM, 18–37.
[34] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. 1994. RAID: High-performance, reliable secondary storage. *Computer Surveys* 26, 2 (June 1994), 145–185.
[35] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2013. Optimistic crash consistency. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*. Farmington, PA.

[36] Vijay Chidambaram, Tushar Sharma, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2012. Consistency without ordering. In *Proceedings of the 10th Conference on File and Storage Technologies (FAST'12)*.

[37] Alex Conway, Ainesh Bakshi, Yizheng Jiao, William Jannen, Yang Zhan, Jun Yuan, Michael A. Bender, Rob Johnson, Bradley C. Kuszmaul, Donald E. Porter, and Martin Farach-Colton. 2017. File systems fated for senescence? Nonsense, says science! In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST'17)*. 45–58.

[38] GParted Forum. 2009. e2fsck is taking forever. http://gparted-forum.surf4.info/viewtopic.php?id=13613.

[39] JaguarPC Forum. 2006. How long does it take FSCK to run?! http://forums.jaguarpc.com/hosting-talk-chit-chat/14217-how-long-does-take-fsck-run.html.

[40] Daniel Fryer, Kuei Sun, Rahat Mahmood, TingHao Cheng, Shaun Benjamin, Ashvin Goel, and Angela Demke Brown. 2012. Recon: Verifying file system consistency at runtime. In *Proceedings of the 10th Conference on File and Storage Technologies (FAST'12)*.

[41] Ryan Gabrys, Eitan Yaakobi, Laura M. Grupp, Steven Swanson, and Lara Dolecek. 2012. Tackling intracell variability in TLC flash through tensor product codes. In *Proceedings of IEEE International Symposium of Information Theory*. 1000–1004.

[42] Gregory R. Ganger, Marshall Kirk McKusick, Craig A. N. Soules, and Yale N. Patt. 2000. Soft updates: A solution to the metadata update problem in file systems. *ACM Transactions on Computer Systems (TOCS'00)* 18, 2 (2000), 127–153.

[43] Om Rameshwar Gatla, Muhammad Hameed, Mai Zheng, Viacheslav Dubeyko, Adam Manzanares, Filip Blagojević, Cyril Guyot, and Robert Mateescu. 2018. Towards robust file system checkers. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST'18)*. USENIX Association, Oakland, CA, 105–122.

[44] Om Rameshwar Gatla and Mai Zheng. 2017. Understanding the fault resilience of file system checkers. In *Proceedings of the 9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'17)*. USENIX Association, Santa Clara, CA.

[45] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The google file system. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles (SOSP'03)*. 29–43.

[46] Laura M. Grupp, Adrian M. Caulfield, Joel Coburn, Steven Swanson, Eitan Yaakobi, Paul H. Siegel, and Jack K. Wolf. 2009. Characterizing flash memory: Anomalies, observations, and applications. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'09)*. 24–33.

[47] Haryadi S. Gunawi, Mingzhe Hao, Riza O. Suminto, Agung Laksono, Anang D. Satria, Jeffry Adityatama, and Kurnia J. Eliazar. 2016. Why does the cloud stop computing? Lessons from hundreds of service outages. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC'16)*. 1–16.

[48] Haryadi S. Gunawi, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2008. SQCK: A declarative file system checker. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. 131–146.

[49] Haryadi S. Gunawi, Cindy Rubio-González, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Ben Liblit. 2008. EIO: Error handling is occasionally correct. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST'08)*, Vol. 8. 1–16.

[50] Zhenyu Guo, Sean McDirmid, Mao Yang, Li Zhuang, Pu Zhang, Yingwei Luo, Tom Bergan, Madan Musuvathi, Zheng Zhang, and Lidong Zhou. 2013. Failure recovery: When the cure is worse than the disease. In *Proceedings of the 14th Workshop on Hot Topics in Operating Systems (HotOS'13)*.

[51] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. 2012. Erasure coding in windows azure storage. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference (USENIX ATC'12)*. 15–26.

[52] Xavier Jimenez, David Novo, and Paolo Ienne. 2014. Wear unleveling: Improving NAND flash lifetime by balancing page endurance. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST'14)*. 47–59.

[53] Andrew Krioukov, Lakshmi N. Bairavasundaram, Garth R. Goodson, Kiran Srinivasan, Randy Thelen, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2008. Parity lost and parity regained. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'08)*, Vol. 8. 1–15.

[54] H. Kurata, K. Otsuga, A. Kotabe, S. Kajiyama, T. Osabe, Y. Sasago, S. Narumi, K. Tokami, S. Kamohara, and O. Tsuchiya. 2006. The impact of random telegraph signals on the scaling of multilevel Flash memories. In *Proceedings of the 2006 Symposium on VLSI Circuits*. IEEE, 112–113.

[55] Changman Lee, Dongho Sim, Joo-Young Hwang, and Sangyeun Cho. 2015. F2FS: A new file system for flash storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*. 273–286.

[56] Jiangpeng Li, Kai Zhao, Xuebin Zhang, Jun Ma, Ming Zhao, and Tong Zhang. 2015. How much can data compressibility help to improve NAND flash memory lifetime? In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*. 227–240.

[57] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. 2013. A study of Linux file system evolution. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST'13)*. 31–44.

[58] Lanyue Lu, Yupu Zhang, Thanh Do, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2014. Physical disentanglement in a container-based file system. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*. 81–96.

[59] Youyou Lu, Jiwu Shu, Weimin Zheng, et al. 2013. Extending the lifetime of flash-based storage through reducing write amplification from file systems. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST'13)*, Vol. 13.

[60] Ao Ma, Chris Dragga, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2013. ffsck: The fast file system checker. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST'13)*. 1–15.

[61] Ashlie Martinez and Vijay Chidambaram. 2017. CrashMonkey: A framework to automatically test file-system crash consistency. In *Proceedings of the 9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'17)*.

[62] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. 1984. A fast file system for UNIX. *Proceedings of the ACM Transactions on Computer Systems (TOCS'84)* 2, 3 (Aug. 1984), 181–197.

[63] Changwoo Min, Sanidhya Kashyap, Byoungyoung Lee, Chengyu Song, and Taesoo Kim. 2015. Cross-checking semantic correctness: The case of finding file system bugs. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)*. ACM, 361–377.

[64] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. 1992. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems (TODS'92)* (1992).

[65] T. Ong, A. Frazio, N. Mielke, S. Pan, N. Righos, G. Atwood, and S. Lai. 1993. Erratic erase in ETOX/sup TM/ flash memory array. In *Proceedings of the Symposium on VLSI Technology (VLSI'93)*.

[66] Lluis Pamies-Juarez, Filip Blagojević, Robert Mateescu, Cyril Gyuot, Eyal En Gad, and Zvonimir Bandić. 2016. Opening the chrysalis: On the real repair performance of MSR codes. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST'16)*. 81–94.

[67] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2014. All file systems are not created equal: On the complexity of crafting crash-consistent applications. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*.

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

[69] Abhishek Rajimwale, Vijay Chidambaram, Deepak Ramamurthi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2011. Coerced cache eviction and discreet mode journaling: Dealing with misbehaving disks. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN'11)*. IEEE, 518–529.

[70] Bianca Schroeder and Garth A. Gibson. 2007. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you? In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST'07)*.

[71] Bianca Schroeder, Raghav Lagisetty, and Arif Merchant. 2016. Flash reliability in production: The expected and the unexpected. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST'16)*. 67–80.

[72] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The hadoop distributed file system. In *Proceedings of the IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST'10)*. IEEE, 1–10.

[73] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. 2016. Push-button verification of file systems via crash refinement. In *Proceedings of 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*.

[74] Keith A. Smith and Margo I. Seltzer. 1997. File system aging: Increasing the relevance of file system benchmarks. In *Proceedings of the 1997 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'97)*. 203–213.

[75] V. Svanberg. 2009. Fsck takes too long on multiply-claimed blocks. http://old.nabble.com/Fsck-takes-too-long-on-multiply-claimed-blocks-td21972943.html.

[76] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. 1996. Scalability in the XFS file system. In *Proceedings of the 1996 USENIX Annual Technical Conference (USENIX ATC'96)*, Vol. 15.

[77] Huang-Wei Tseng, Laura M. Grupp, and Steven Swanson. 2011. Understanding the impact of power loss on flash memory. In *Proceedings of the 48th Design Automation Conference (DAC'11)*.

[78] Stephen C. Tweedie. 1998. Journaling the linux ext2fs filesystem. In *Proceedings of the 4th Annual Linux Expo*.

[79] Simeng Wang, Jinrui Cao, Danny V. Murillo, Yiliang Shi, and Mai Zheng. 2016. Emulating realistic flash device errors with high fidelity. In *Proceedings of the IEEE International Conference on Networking, Architecture and Storage (NAS'16)*. IEEE.

[80] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. 2006. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI'06)*. 307–320.

[81] Mingyuan Xia, Mohit Saxena, Mario Blaum, and David A. Pease. 2015. A tale of two erasure codes in HDFS. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*. 213–226.

[82] Gala Yadgar, Eitan Yaakobi, and Assaf Schuster. 2015. Write once, get 50% free: Saving SSD erase costs using WOM codes. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*. 257–271.

[83] Junfeng Yang, Can Sar, and Dawson Engler. 2006. EXPLODE: A lightweight, general system for finding serious storage system errors. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI'06)*. 131–146.

[84] Yiying Zhang, Leo Prasath Arulraj, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2012. De-indirection for flash-based SSDs with nameless writes. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12)*.

[85] Yupu Zhang, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2010. 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)*. 29–42.

[86] Mai Zheng, Joseph Tucek, Dachuan Huang, Feng Qin, Mark Lillibridge, Elizabeth S. Yang, Bill W. Zhao, and Shashank Singh. 2014. Torturing databases for fun and profit. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*. 449–464.

[87] Mai Zheng, Joseph Tucek, Feng Qin, and Mark Lillibridge. 2013. Understanding the robustness of SSDs under power fault. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST'13)*.

[88] Mai Zheng, Joseph Tucek, Feng Qin, Mark Lillibridge, Bill W. Zhao, and Elizabeth S. Yang. 2016. Reliability analysis of SSDs under power fault. In *Proceedings of the ACM Transactions on Computer Systems (TOCS'16)*. http://dx.doi.org/10.1145/2992782

Received September 2018; accepted September 2018