Understanding the Fault Resilience of File System Checkers

Om Rameshwar Gatla Mai Zheng
Computer Science Department, New Mexico State University

Abstract

File system checkers serve as the last line of defense to recover a corrupted file system back to a consistent state. Therefore, their reliability is critically important. Motivated by real accidents, in this paper we study the behavior of file system checkers under faults. We systematically inject emulated faults to interrupt the checkers and examine the impact on the file system images. In doing so, we answer two important questions: Does running the checker after an interrupted-check successfully return the file system to a correct state? If not, what goes wrong? Our results show that there are vulnerabilities in popular file system checkers which could lead to unrecoverable data loss under faults.

1 Introduction

File systems play an essential role in managing today's data. From desktops to warehouse-scale machines [14], a wide variety of local file systems (e.g., Ext4 [5], XFS [8], Btrfs [1]) and large-scale file systems (e.g., GFS [29], Ceph [2], Lustre [6]) are deployed to store and manage much of the world's precious data. Therefore, the integrity of file systems is critically important.

Unfortunately, despite of various protection techniques, file systems may still become corrupted for many reasons including power outages, server crashes, latent sector errors, software or firmware bugs, etc [12, 13, 25, 37, 39]. Therefore, most file systems come with a checker to serve as the last line of defense [4, 9, 39, 40]. The checker usually scans the on-disk layout of the corresponding file system, resolves inconsistencies based on pre-defined policies, and recovers the corrupted system back to a healthy state.

Due to the prime importance of file system checkers, abundant work has been done to improve them. For example, Gunawi *et al.* [31] finds that the checker of Ext2 may create inconsistent or even insecure repairs; they then proposes a more elegant checker (i.e., `SQCK`) based on a declarative query language. Similarly, `SWIFT` [21] uses a mix of symbolic and concrete execution to test popular file system checkers, and has exposed bugs in all checkers tested. And more recently, `ffsck` [39] reduces the checking time of Ext3-like file systems by changing the disk layout and the indirect block structure of Ext3 and co-designing the checking policies.

One common assumption of the aforementioned work is that the checker can finish normally *without interruption*. In other words, they mostly focus on the behavior of file system checkers during normal executions.

Complementary to the existing efforts, in this work we study the behavior of file system checkers *with interruption*. This is motivated by a real-world accident happened at the High Performance Computing Center (HPCC) in Texas [10, 20], where the Lustre file system suffered severe data loss after experiencing two consecutive power outages: the first one triggered the Lustre file system checker (i.e., `LFSCK`) after restarting the cluster, while the second one interrupted `LFSCK` and led to the final downtime and corruption. Since the Lustre deployed is built on top of a variant of Ext4 (i.e., `ldiskfs`), and `LFSCK` relies on the consistency of the local file system on every node, the overall checking and recovery procedure is complicated (e.g., requires several days [10]). Moreover, with the trend of increasing the capacity of storage devices and scaling to more and more nodes, checking and repairing file systems will likely become more time-consuming and thus more vulnerable to faults.

As one step towards building fault-resilient file system checkers, we perform a comprehensive study on the behavior of existing local file system checkers under faults. Specifically, we explore two important questions: Does running the checker after an interrupted-check successfully return the file system to a correct state? If not, what goes wrong?

To answer the questions, we build a fault-injection framework based on an iSCSI driver [55]. We use a set of test images, which are either provided by the developers of file systems or generated by ourselves, as the input

to trigger the target file system checker. During the execution of the checker, we record the I/O commands at the driver level, and replay partial I/O blocks to emulate the effect of an interrupted checker on the file system image. Moreover, after generating the interrupted state, we re-run the checker again without any fault injection. This is to verify that whether the corruption introduced by the interrupted checker can be recovered or not.

We apply the testing methodology to study `e2fsck` [4] and `xfs_repair` [9], the default checkers of the popular Ext-series file systems and XFS file system, respectively. Our experimental results show that there are vulnerabilities in these representative file system checkers which could lead to additional severe corruption under faults (e.g., the file system becomes "unmountable"). Moreover, many corruptions cannot be repaired by re-executing the default checker again, which implies that the data loss incurred might be permanent.

The rest of the paper is organized as follows. We first introduce the background of file system checkers (§2). Next, we describe our methodology of injecting faults and testing the resilience (§3). Then, we discuss our studies on `e2fsck` (§4) and `xfs_repair` (§5). Finally, we discuss related work (§6) and conclude (§7).

2 Background of File System Checkers

File system checkers are designed to check and repair inconsistencies in file systems. Depending on the structures of file systems, the checkers may perform different number of checking passes and exam different consistency rules. We use `e2fsck` as a concrete example to illustrate the complexity as well as the potential vulnerabilities of the checkers in this section.

The first action of `e2fsck` is to replay the journal (in case of Ext3/Ext4) and then restart itself. Next, `e2fsck` executes the following five passes in order:

Pass 1: Scan the entire 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`), blocks claimed by two inodes (`block_dup_map`), etc. Besides, it performs four sub-passes to generate a list of duplicate blocks and their owners, checks the integrity of the extent trees, etc.

Pass 2: Check directory structure. Based on the bitmaps collected, Pass 2 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, etc.

Pass 3: Check connectivity. This pass ensures that all directories are connected properly in the file system tree. To this end, `e2fsck` first checks whether if a root directory exists. If not, then a new root directory is created and is marked as "done". Next, `e2fsck` iterates over all directory inodes and attempts to traverse up the file system tree, until it reaches a directory marked as "done". If no "done" directory is reachable, 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), `e2fsck` also offers to reconnect the directory to the "lost+found" folder to break the loop.

Pass 4: Check reference counts. Pass 4 iterates over all the inodes to validate the inode link counts. Also, it checks the connectivity of the extended attribute (EA) blocks reconnect them if necessary. In addition, Pass 4 loads a set of readahead bitmaps for the next pass.

Pass 5: Recalculate checksums and flush updates. In the final pass `e2fsck` checks the repaired in-memory data structures against on-disk data structures and flush 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 re-calculated before flushing.

Based on the analysis above, we can see that the checking procedure of `e2fsck` is complicate and may be vulnerable to faults. For example, there are cross-pass dependency (e.g., the bitmaps are used and updated by multiple passes) and some early fixes are not committed until the last pass. We describe our methodology to actually expose the vulnerabilities in the next section.

3 Methodology

3.1 Test Images

Since file system checkers are designed to repair corrupted file systems, it is necessary to have a set of corrupted file system images to trigger the checkers. We call this set of file system images as *test images*.

We consider two ways of generating test images. First, the developers of file systems may provide test images for regression testing of their checkers. For example, `e2fsprogs` [4] includes a set of 175 test images for testing `e2fsck`. These images usually cover the most representative corruption scenarios as envisioned by the developers. In this case, we directly use the test images provided by the developers for triggering the target file system checker.

Second, in case the default test images are not available, we inject random bits to corrupt important on-disk structures of the file system, similar to the previous approaches for introducing corruptions [31, 54]. While this approach requires in-depth understanding of the file system layout, it allows great flexibility in terms of creating

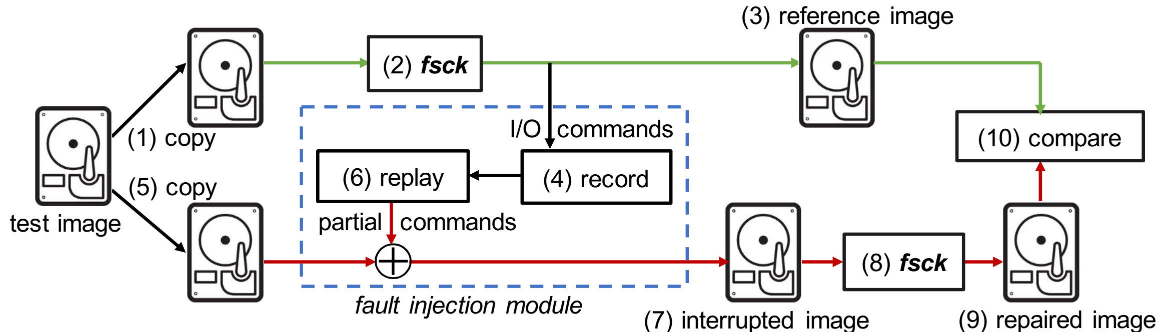

Figure 1: Workflow of testing the fault resilience of a file system checker (i.e., `fsck`). There are 10 steps: (1) make a copy of the test image which contains a corrupted file system; (2) run `fsck` on the copy of test image; (3) store the image generated in (2) as the reference image; (4) record the SCSI 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` on the copy of test image; (7) store the image generated in (6) as the interrupted image; (8) run `fsck` on the interrupted image; (9) store the image generated in (8) as the repaired image; (10) compare the repaired image with the reference image to identify mismatches.

corruption scenarios and could potentially cover corner cases that are not envisioned by the developers.

3.2 Fault Injection Module

Besides generating test images to trigger the target checker, another challenge of evaluating the fault resilience is how to generate faults in a systematic and controllable way. Unplugging the power cord repeatedly is simply impractical.

To this end, we emulate the effect of faults using software. We adopt the "clean power fault" model [55]: there is a minimal atomic unit of write operations, i.e., the size of data written to media is always an integer block multiple (e.g., 512B or 4KB); a fault can occur at any point during the execution of the checker; once a fault happens, all blocks committed before the fault are durable without corruption, and all blocks after the fault have no effect on the media. This simple model can serve as a conservative lower bound of the failure impact. In other words, file system checkers must be able to handle this simple fault model gracefully before addressing more aggressive fault models (e.g., arbitrary reordering of blocks).

To emulate the fault model, we adapt an iSCSI driver [55] to record the I/O commands generated during the execution of the target file system checker in a command history log. Then, we replay a prefix of the command history log (i.e., partial commands) to a copy of the initial test image, which effectively generate the effect of an interrupted checker on the test image.

For each command history log, 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 target checker.

3.3 Workflow

Putting it together, Figure 1 summarizes the overall workflow of testing the fault resilience of file system checkers. As shown in the figure, there are 10 steps: (1) we make a copy of the test image which contains a corrupted file system; (2) the target file system checker (i.e., `fsck`) is executed to check and repair the original corruption on the copy of the test image; (3) after running `fsck`, the resulting image is stored as the reference image; (4) during the checking and repairing procedure of `fsck`, we simultaneously record the I/O commands generated by `fsck`; (5) we make another copy of the original test image; (6) we replay 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 (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 is first performed via the `diff` command. If a mismatch is reported, we further verify the cause of the mismatch manually.

4 Case Study I: `e2fsck`

4.1 Overall Results

We use 175 test images from `e2fsprogs v1.43.1` [4] as inputs to trigger `e2fsck`. The file system block size on all these images is 1KB. To emulate faults on storage systems with different atomic units, we inject faults in two granularities: 512B or 4KB. In other words, we interrupt `e2fsck` after every 512B or 4KB of an I/O trans-

Fault Injection Granularities	Total # of test images	Total # of repaired images
512 B	175	25,062
4 KB	175	3,195

Table 1: Number of test images and repaired images generated under two fault injection granularities.

Fault Injection Granularities	# of images reporting corruptions	
	test images	repaired images
512 B	34	240
4 KB	17	37

Table 2: Number of test images and repaired images reporting corruptions under two fault injection granularities.

fer command. Since the file system block is 1KB, we do not break file system blocks when injecting faults in 4KB granularity.

As described in §3.3, for each fault injected (i.e., each interruption) we run `e2fsck` again and generate one repaired image. From one test image we usually generate multiple repaired images because the repairing procedure usually requires updating multiple file system blocks. For example, to fix the test image "`f_noroot`" (image with no root directory) from `e2fsprogs`, `e2fsck` needs to update four 1KB blocks, which generates eight interrupted images (and consequently repaired images) when faults are injected after each 512B. Table 1 summarizes the total number of repaired images generated under the two granularities. We can see that the 512B case leads to more repaired images because the repairing procedure is interrupted more frequently.

By comparing with the reference image (Figure 1), we can verify if a repaired image contains unrecoverable corruptions or not. If at least one repaired image contains such corruption, we mark the corresponding test image as reporting corruption too. Table 2 shows the number of test images and the number of repaired images reporting corruptions. We can see that with the 512B granularity, more images have corruptions. This is because the updates are broken into smaller pieces, and thus it is more challenging to maintain consistency when interrupted.

To further understand the severeness of the corruptions, we examine the symptoms in details and classify them into four types, including: (1) *unmountable*; (2) *file content corruption*; (3) *misplacement of files* (e.g., a file is either in the"lost+found" folder or completely missing) (4) *others* (e.g., showing "???" after an `ls` command). Table 3 shows the classification. We can see that *file content corruption* and *misplacement of files* are most common. Also, even with a fault injection granularity bigger than the file system block (i.e., 4KB > 1KB), we still observe three cases where the whole file system volume becomes *unmountable*.

Corruption Types	512 B	4 KB
unmountable	41	3
file content corruption	107	10
misplacement of files	82	23
others	10	1
Total	240	37

Table 3: Classification of corruptions observed on repaired images under two fault injection granularities.

Sync Methods	# of images reporting corruptions	
	test images	repaired images
sync every write	45	223
sync every pass	45	243

Table 4: Number of test images and repaired images reporting corruptions after enforcing synchronous updates in two ways.

4.2 Enforcing Synchronous Writes

For performance reason, the existing `e2fsck` buffers most updates in memory and flush them to the disk synchronously only at the end of the last pass. This extensive buffering may cause inconsistencies under faults because there is no ordering or atomicity guarantees when flushing the large amount of buffered updates.

To examine the impact of this design tradeoff, we modify the source code of `e2fsck` to enforce frequent synchronous writes in two ways. First, we change all updates to the disk image to synchronous writes by adding the `O_SYNC` flag. Second, we insert an `fsync` call at the end of each of the five passes of `e2fsck`. We inject faults in 4KB granularity and see if the more frequent synchronization would reduce the window of vulnerability.

As shown in Table 4, in the first scenario (i.e., sync every write), 45 test images and 223 repaired images have corruptions. The numbers in the second scenario (i.e., sync at the end of each pass) are similar. In addition, Table 5 further classifies the corruptions observed on the repaired images into the four types described in §4.1.

From this experiment, we can see that enforcing synchronous updates may avoid some corruptions (e.g., 0 in "others" in Table 5). However, most corruptions cannot be solved in this way. Moreover, by frequently enforcing synchronization, more intermediate states become visible on disk. As a result, we may observe even more unrecoverable corruptions (i.e., the numbers in Table 4 are generally larger than those in Table 2), which suggests that the resilience of `e2fsck` cannot be enhanced simply by enforcing synchronous writes.

4.3 Undo Logging

One way to handle unexpected interruptions is write ahead logging [42]. In fact, the developers of `e2fsck` have envisioned the potential needs and have added built-in support for an undo log, although this feature has sel-

Corruption Types	sync every write	sync every pass
unmountable	203	211
file content corruption	13	11
misplacement of files	7	20
others	0	0
Total	223	242

Table 5: Number of repaired images reporting each type of corruptions after enforcing synchronous updates in two ways.

dom been used due to the degraded performance [3].

To exam the effectiveness of the undo logging, we create a separate block device in our framework to store the undo log, and inject faults to interrupt both the checking and the logging. Surprisingly, we observe a similar amount of corruptions. Further analysis on the source code as well as the system call traces reveals 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 write ahead logging mechanism. This current status suggests that enhancing the checker with transactional support is non-trivial. More effective and efficient mechanisms are needed.

5 Case Study II: `xfs_repair`

We have also studied the resilience of `xfs_repair`, the checker of XFS file system. To generate test images, we make use of the `blocktrash` command of `xfs_db` [9] to flip random bits on the file system images. In the experiment, we generated 3 test images by flipping 10, 20 and 30 bits, respectively. From the 3 test images, the framework generated 124 repaired images in total under the 4KB fault injection granularity, among which 16 images have exhibit the *misplacement of files* corruption.

6 Related Work

Reliability of file system checkers. Gunawi *et al.* [31] inject faults to corrupt carefully selected blocks of a Ext2 file system, and find that the Ext2 checker may create inconsistent or even insecure repairs; they then proposes a more elegant checker (i.e., `SQCK`) based on a declarative query language. Carreira *et al.* [21] propose a tool called `SWIFT`, which uses a mix of symbolic and concrete execution and includes a corruption model for testing file system checkers. They tested the checkers of ext2, ext3, ext4, ReiserFS, and Minix and found bugs in all of them. Ma *et al.* [39] change the disk layout and the indirect block structure of Ext3 (i.e., `rext3`) and co-design the checking policies (i.e., `ffsck`), which enables scanning and repairing the file system much more efficiently. By speeding up the checking procedure, they effectively narrow the window of vulnerability for the checker.

Generally, these studies consider the behavior of file system checkers during normal executions (i.e., no interruption). Complimentary to these efforts, we study checkers' behavior under faults.

Reliability of file systems. Great efforts have been put towards improving the reliability of file systems [16, 23, 26, 27, 32, 37, 41, 44, 48, 52, 54]. For example, EXPLODE [52] uses model checking to find errors in storage software, especially in file systems. Chidambaram *et al.* [26] introduce a backpointer-based NoFS to provide crash consistency. Overall, these research help understand and improve the reliability of file systems, which may reduce the need for file system checkers. However, despite of these efforts, file system checkers remain a necessary component for most file systems.

Reliability of storage devices. In terms of storage devices, research efforts are also abundant [12, 13, 24, 34, 45, 46]. For example, Schroeder *et al.* [46] analyze the disk replacement data of seven production systems over five years. Bairavasundaram *et al.* [12, 13] 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 flash-based solid state drives (SSDs) [11, 15, 17, 18, 19, 22, 28, 30, 33, 35, 36, 38, 43, 47, 49, 50, 51, 53, 56, 57]. These studies provide valuable insights for understanding file system corruptions caused by hardware, and are helpful for generating realistic test images to trigger the checkers in our study.

7 Conclusions and Future Work

We have studied the behavior of file system checkers under emulated faults. Our results show that running the checker after an interrupted-check may not return the file system to a correct state, and the image may even become unmountable. In the future, we would like to apply the methodology to study the reliability of other critical but insufficiently tested procedures (e.g., system upgrades). We hope our study will raise the awareness of reliability vulnerabilities in storage systems in general, and facilitate building truly fault-resilient systems.

8 Acknowledgements

We thank the anonymous reviewers for their insightful comments and suggestions. We also thank the Linux kernel practitioners, especially Theodore Ts'o and Ric Wheeler, for the invaluable discussion at Linux FAST Summit. This work was supported in part by the National Science Foundation (NSF) under grant CNS-1566554. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of NSF.

References

[1] Btrfs File System. https://btrfs.wiki.kernel.org/index.php/Main_Page.

[2] Ceph File System. http://docs.ceph.com/docs/master/.

[3] Discussion with Theodore Ts'o at Linux FAST Summit'17. https://www.usenix.org/conference/linuxfastsummit17.

[4] E2fsprogs: Ext2/3/4 Filesystem Utilities. http://e2fsprogs.sourceforge.net/.

[5] Ext4 File System. https://ext4.wiki.kernel.org/index.php/Main_Page.

[6] Lustre File System. http://opensfs.org/lustre/.

[7] Samba TDB (Trivial DataBase). https://tdb.samba.org/.

[8] XFS File System. http://xfs.org/index.php/Main_Page.

[9] XFS File System Utilities. https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Storage_Administration_Guide/xfsothers.html.

[10] High Performance Computing Center (HPCC) Power Outage Event. Email Annoucement by HPCC, Monday, January 11, 2016 at 8:50:17 AM CST. https://www.cs.nmsu.edu/~mzheng/docs/failures/2016-hpcc-outage.pdf, 2016.

[11] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. Design Tradeoffs for SSD Performance, 2008.

[12] Lakshmi N. Bairavasundaram, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Garth R. Goodson, and Bianca Schroeder. An analysis of data corruption in the storage stack. *Trans. Storage*, 4(3):8:1–8:28, November 2008.

[13] Lakshmi N. Bairavasundaram, Garth R. Goodson, Shankar Pasupathy, and Jiri Schindler. An analysis of latent sector errors in disk drives. In *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '07, pages 289–300, New York, NY, USA, 2007. ACM.

[14] Luiz Andre Barroso and Urs Hoelzle. *The Datacenter As a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool Publishers, 1st edition, 2009.

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

[16] James Bornholt, Antoine Kaufmann, Jialin Li, Arvind Krishnamurthy, Emina Torlak, and Xi Wang. Specifying and checking file system crash-consistency models. *ACM SIGPLAN Notices*, 51(4):83–98, 2016.

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

[18] Yu Cai, Erich F. Haratsch, Onur Mutlu, and Ken Mai. Error Patterns in MLC NAND Flash Memory: Measurement, Characterization, and Analysis. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '12, pages 521–526, San Jose, CA, USA, 2012. EDA Consortium.

[19] Yu Cai, Gulay Yalcin, Onur Mutlu, Erich F Haratsch, Osman Unsal, Adrian Cristal, and Ken Mai. Neighbor-cell assisted error correction for MLC NAND flash memories. In *ACM SIGMETRICS Performance Evaluation Review*, volume 42, pages 491–504. ACM, 2014.

[20] Jinrui Cao, Simeng Wang, Dong Dai, Mai Zheng, and Yong Chen. 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, pages 49–54, Piscataway, NJ, USA, 2016. IEEE Press.

[21] João Carlos Menezes Carreira, Rodrigo Rodrigues, George Candea, and Rupak Majumdar. Scalable testing of file system checkers. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 239–252, New York, NY, USA, 2012. ACM.

[22] Feng Chen, David A. Koufaty, and Xiaodong Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *ACM SIGMETRICS*, 2009.

[23] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M Frans Kaashoek, and Nickolai Zeldovich. Using crash hoare logic for certifying the

fscq file system. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 18–37. ACM, 2015.

[24] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. RAID: high-performance, reliable secondary storage. *ACM Comput. Surv.*, 26(2):145–185, June 1994.

[25] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Optimistic Crash Consistency. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, Farmington, PA, November 2013.

[26] Vijay Chidambaram, Tushar Sharma, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Consistency Without Ordering. In *Proceedings of the 10th Conference on File and Storage Technologies (FAST'12)*, San Jose, California, February 2012.

[27] Daniel Fryer, Kuei Sun, Rahat Mahmood, TingHao Cheng, Shaun Benjamin, Ashvin Goel, and Angela Demke Brown. Recon: Verifying File System Consistency at Runtime. In *Proceedings of the 10th Conference on File and Storage Technologies (FAST'12)*, San Jose, California, February 2012.

[28] Ryan Gabrys, Eitan Yaakobi, Laura M. Grupp, Steven Swanson, and Lara Dolecek. Tackling intracell variability in TLC flash through tensor product codes. In *ISIT'12*, pages 1000–1004, 2012.

[29] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[30] Laura M. Grupp, Adrian M. Caulfield, Joel Coburn, Steven Swanson, Eitan Yaakobi, Paul H. Siegel, and Jack K. Wolf. Characterizing flash memory: anomalies, observations, and applications. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 24–33, New York, NY, USA, 2009. ACM.

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

[32] Haryadi S Gunawi, Cindy Rubio-González, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, and Ben Liblit. Eio: Error handling is occasionally correct. In *FAST*, volume 8, pages 1–16, 2008.

[33] Xavier Jimenez, David Novo, and Paolo Ienne. Wear unleveling: improving nand flash lifetime by balancing page endurance. In *FAST*, pages 47–59, 2014.

[34] Andrew Krioukov, Lakshmi N Bairavasundaram, Garth R Goodson, Kiran Srinivasan, Randy Thelen, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Parity lost and parity regained. In *FAST*, volume 8, pages 1–15, 2008.

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

[36] Jiangpeng Li, Kai Zhao, Xuebin Zhang, Jun Ma, Ming Zhao, and Tong Zhang. How much can data compressibility help to improve nand flash memory lifetime? In *FAST*, pages 227–240, 2015.

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

[38] Youyou Lu, Jiwu Shu, Weimin Zheng, et al. Extending the lifetime of flash-based storage through reducing write amplification from file systems. In *FAST*, volume 13, 2013.

[39] Ao Ma, Chris Dragga, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. ffsck: The fast file system checker. In *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 1–15, San Jose, CA, 2013. USENIX.

[40] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for unix. *ACM Trans. Comput. Syst.*, 2(3):181–197, August 1984.

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

[42] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, March 1992.

[43] T. Ong, A. Frazio, N. Mielke, S. Pan, N. Righos, G. Atwood, and S. Lai. Erratic Erase In ETOX/sup TM/ Flash Memory Array. In *Symposium on VLSI Technology*, VLSI'93, 1993.

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

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

[46] Bianca Schroeder and Garth A. Gibson. 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)*, 2007.

[47] Bianca Schroeder, Raghav Lagisetty, and Arif Merchant. Flash reliability in production: The expected and the unexpected. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 67–80, Santa Clara, CA, February 2016. USENIX Association.

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

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

[50] Simeng Wang, Jinrui Cao, Danny V Murillo, Yiliang Shi, and Mai Zheng. Emulating Realistic Flash Device Errors with High Fidelity. In *IEEE International Conference on Networking, Architecture and Storage (NAS'16)*. IEEE, 2016.

[51] Gala Yadgar, Eitan Yaakobi, and Assaf Schuster. Write once, get 50% free: Saving ssd erase costs using wom codes. In *FAST*, pages 257–271, 2015.

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

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

[54] Yupu Zhang, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. End-to-end Data Integrity for File Systems: A ZFS Case Study. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, FAST'10, pages 3–3, Berkeley, CA, USA, 2010.

[55] Mai Zheng, Joseph Tucek, Dachuan Huang, Feng Qin, Mark Lillibridge, Elizabeth S. Yang, Bill W Zhao, and Shashank Singh. Torturing Databases for Fun and Profit. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 449–464, Broomfield, CO, 2014.

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

[57] Mai Zheng, Joseph Tucek, Feng Qin, Mark Lillibridge, Bill W Zhao, and Elizabeth S. Yang. Reliability Analysis of SSDs under Power Fault. In *ACM Transactions on Computer Systems (TOCS)*, 2016.