A Generic Framework for Testing Parallel File Systems

Jinrui Cao†, Simeng Wang†, Dong Dai‡, Mai Zheng†, and Yong Chen‡

†Computer Science Department, New Mexico State University, USA, {will_cao, xzqxnet, zheng}@nmsu.edu
‡Computer Science Department, Texas Tech University, USA, {dong.dai, yong.chen}@ttu.edu

***Abstract*—Large-scale parallel file systems are of prime importance today. However, despite of the importance, their failure-recovery capability is much less studied compared with local storage systems. Recent studies on local storage systems have exposed various vulnerabilities that could lead to data loss under failure events, which raise the concern for parallel file systems built on top of them.**

This paper proposes a generic framework for testing the failure handling of large-scale parallel file systems. The framework captures *all* disk I/O commands on *all* storage nodes of the target system to emulate realistic failure states, and checks if the target system can recover to a consistent state without incurring data loss. We have built a prototype for the Lustre file system. Our preliminary results show that the framework is able to uncover the internal I/O behavior of Lustre under different workloads and failure conditions, which provides a solid foundation for further analyzing the failure recovery of parallel file systems.

I. Introduction

Storage systems must handle failure events (e.g., device failures, crashes, power outages) gracefully, which is extremely difficult to achieve in practice. A simple block device may contain many thousands of lines of concurrent firmware [17], and a local file system (e.g., ext4, XFS) usually consists of many tens of thousands of lines of kernel-space code [11]. Given such complexity, it is perhaps no surprise that many recent studies have uncovered vulnerabilities at almost every layer of the local storage stack (e.g., devices [36], [37], RAID [20], drivers [27], block layer [37], local file systems [35], [24], [22]), which may lead to data loss under various failure events. This raises the concern for large-scale parallel file systems (e.g., Lustre [5], Ceph [2]) building on top of the local storage stack. In fact, in a recent power outage accident happened at Texas Tech University HPCC (High Performance Computing Center) [8], the Lustre file system suffered severe data loss after experiencing two consecutive power outages. Although many of the data are recovered eventually after several weeks of manual efforts, there are still critical data lost permanently, and the potential damage to the scientific results are unmeasurable. As of today, it is still unclear where the bug is exactly, let alone how to fix it.

One challenge for testing large-scale parallel file systems is the complexity: adding a complex layer across many already-complicated local storage systems makes analyzing the whole system behavior difficult. Besides the complexity, another challenge is how to generate the failure events (e.g., crashes or power outages) in a systematic and controllable way. Manually unplugging the power cord is simply impractical.

In this paper, we propose a generic framework for testing the failure handling of large-scale parallel file systems. The framework creates virtual devices to capture *all* I/O commands in the target system, emulates realistic failure states on the virtual devices, and checks if the system can gracefully recover to a consistent state without data loss. More specifically, the framework is based on the following observations and design choices:

First, failure events may vary, but only the on-drive persistent states would affect the recovery after rebooting. Thus, we boil down the emulation of various failure events to the states of individual devices. We create a virtual device on each storage node and manipulate the I/O commands to emulate the failure states under workloads.

Second, different parallel file systems have different internal designs as well as different dependency on local file systems (e.g., ext3/4 is the classic backend for Lustre/ldiskfs, while it is not usable for Ceph OSD Daemons [2]). However, despite of such discrepancy, we can always separate the target system into a device layer and a host-side software layer. In other words, by generating the failure states on virtual devices, we enable testing different parallel file systems with little porting effort.

Third, large-scale parallel file systems are inherently complicated and difficult to configure. Thus, the framework must minimize the disturbance to the local storage stack to avoid unexpected issues. To this end, we make use of a remote storage protocol to decouple the framework from the storage nodes in the target system, which makes the deployment simple in practice.

To demonstrate the feasibility, we have built a prototype for the Lustre file system. The prototype maintains a virtual device for every storage node in Lustre, and captures all disk I/O commands to emulate realistic failure states. Our preliminary results show that the framework is able to uncover the internal I/O behavior of all Lustre nodes under different workloads and different failure conditions, which provides a solid foundation for further analyzing and improving the failure handling of large-scale parallel file systems.

II. Background

A. Lustre File System

Lustre is one of the most popular parallel file systems used in modern high performance computing (HPC) platforms. Over half of the top 100 fastest supercomputers in the world are using it; and it has dominated the market share of parallel file systems deployed in HPC clusters [7]. Hence, its failure handling is becoming more and more critical for both HPC community and scientists. This trend can also be easily seen from its recent releases (from 2.6), which significantly improve the LFSCK (i.e., Lustre File System Checker [3]) functionality to enhance the failure handling.

The overall architecture of Lustre mainly includes the following components:

- Management Server (MGS) and Management Target (MGT), which manage store the configuration information for all Lustre file systems in a cluster. MGS can be co-located with MDS.
- Metadata Server (MDS) and Metadata Target (MDT), which store metadata in the Lustre file system. MDS provides network request handling for one or more local MDTs. There can be multiple MDSs and MDTs since Lustre 2.4.
- Object Storage Server (OSS) and Object Storage Target (OST), which store the actual data. The OSS provides the file I/O service and the network request handling for one or more local OSTs. User file data is stored in one or more objects, and each object is stored on a separate OST in a Lustre file system.
- Lustre Clients, which mount the Lustre file system for running scientific applications. They usually run in login nodes and compute nodes.

A typical deployment of Lustre file system usually includes one dedicated MGS server, one or two dedicated MDS and MDT server(s), two or more OSSs and OSTs, and a large number of clients running on compute nodes. In this paper, we build a Lustre file system with the typical setting for experiments (Section IV).

B. Remote Storage Protocols

Remote storage protocols (e.g., NFS [28], Fibre Channel [26], iSCSI [4], NVMe/Fabric [1]) enable accessing remote storage devices as local devices, either at the file level or at the block level. In particular, iSCSI[4] is an IP-based protocol allowing one machine (the initiator) to access remote block storage through the internet, which makes it easy to deploy on modern computer systems. To everything above the block driver on the initiator, iSCSI is completely transparent. In other words, file systems can be built on iSCSI devices without any modification. In this work, we make use of the iSCSI protocol to decouple our framework from the target system, which enables testing different parallel file systems transparently.

III. Design and Implementation

A. Overview

Figure 1 shows the overview of our framework. There are four major components: (1) the *Virtual Device Manager* creates device files and mounts them to the Lustre nodes as virtual devices via iSCSI, which enables capturing all disk I/O commands in Lustre; (2) the *Failure State Emulator* emulates the device states under certain workloads and failure events; (3) the *Data-Intensive Workloads* exercises Lustre and generates I/O operations; (4) the *Post-Failure Checker* examines the post-failure behavior of Lustre and checks its consistency and data integrity. Note that although we use Lustre as an example, the framework can be applied to other parallel file systems by simply installing different software packages on the virtual devices.

B. Virtual Device Manager

The persistent state of a parallel file system depends on the I/O operations in the system. To capture all I/O operations in Lustre, the Virtual Device Manger creates and maintains one device file for each storage device used in Lustre. The device files are mounted to the Lustre nodes as virtual devices via iSCSI. From Lustre's perspective, the virtual devices are just ordinary disks. In other words, the framework is transparent to the parallel file system.

All I/O operations in Lustre are eventually translated into disk I/O commands, which are transferred to the Virtual Device Manager. The Virtual Device Manager updates the content of the backend device files according to the commands received. Moreover, it also logs all commands in a command history log. For each command, the log includes the Lustre node ID (e.g., MDT or OST), the command details (e.g., type, offset, length, timestamp), and the actual data transferred. This fine-grained command-level information uncovers the internal behavior of Lustre. Moreover, the command history is used by the Failure State Emulator (Section III-C) to emulate various failure states.

C. Failure State Emulator

To study the failure handling of the target system, it is necessary to generate the failure events in a systematic and controllable way. The Failure State Emulator achieves this by manipulating the virtual devices and I/O commands, and emulates the failure state of each individual device. Specifically, we emulate the following four failure modes, which represents a wide range of real-world failure scenarios [9], [10], [23], [29], [31], [20], [21], [35], [36]:

(1) Whole device failure. This is the case when a device becomes invisible to the Lustre, which can be caused by various reasons including controller failure, firmware bugs, accumulation of sector errors, etc [36], [37], [20], [21]. Thanks to the clear design of decoupling the framework from Lustre via virtual devices, we can simply log out virtual devices to emulate the failure. More specifically, we use the *logout* command in the iSCSI protocol (Section II-B) to disconnect

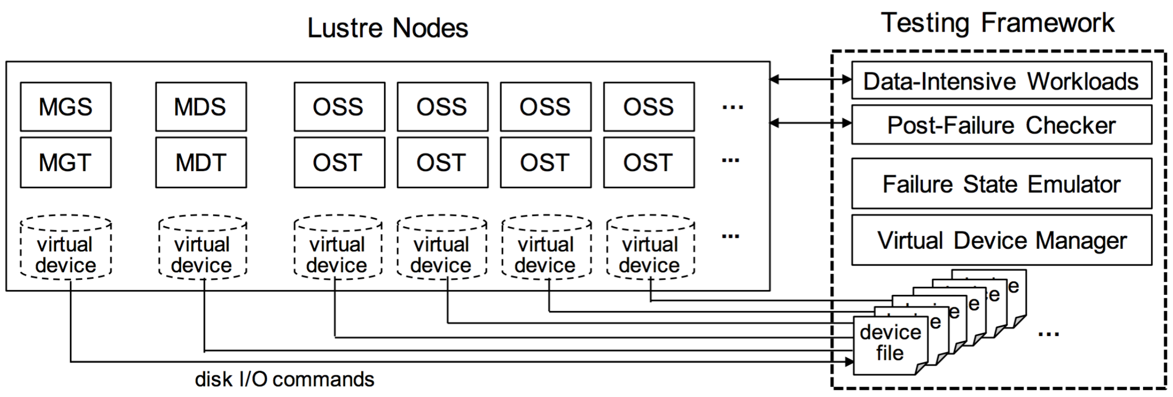

Fig. 1: Framework overview. There are four major components: (1) *Virtual Device Manager* mounts virtual devices to the Lustre nodes via iSCSI; (2) *Failure State Emulator* emulates the device states under certain workloads and failure events; (3) *Data-Intensive Workloads* exercises Lustre and generates I/O operations; (4) *Post-Failure Checker* examines the post-failure behavior of Lustre and checks its consistency and data integrity.

the backing device files to the Lustre nodes, which makes the devices invisible to Lustre immediately.

(2) Clean termination of writes. This emulates a simplified power outage event: all writes before the outage are visible on the device, while all writes after the event are lost. Although simple, this failure mode represents a conservative lower-bound of the failure impact (i.e., easiest to recover from), and has been used to uncover surprising reliability vulnerabilities in matured enterprise storage systems [35]. Since the Virtual Device Manager (Section III-B) has logged all commands during the workload, we can replay partial commands in the command history log to a virtual device to emulate the termination of the write stream.

(3) Reordering of writes. This is a more aggressive failure mode which commits writes to the device in an order different from the issuing order of the I/O requests, which may be caused by the buffering and parallelism on local storage stack under crashes or power outages [34], [36], [24]. Because all commands are recorded in the command history log, we can reorder the commands in the log and replay the (reordered) command sequence to generate the failure device state. Moreover, because the synchronization commands (e.g., SCSI command code 0x35) are also part of the command history, we can honor the write barriers when reordering.

(4) Corruption of device blocks. This is a severe failure mode which may be caused by various reasons including hardware failures or firmware bugs [10], [36]. We emulate the effect by injecting random bits to the device files.

Note that these four failure modes are not exclusive, i.e., we can combine multiple modes (e.g., reordering and corruption) on one virtual device simultaneously. Moreover, we can emulate different failure modes on different devices, which creates a more challenging state for Lustre to recover from.

Workload	Description
Montage/m101	astronomical image mosaic engine
cp	copy a file into Lustre
tar	decompress a file on Lustre
rm	delete a file from Lustre

TABLE I: Workloads ran on Lustre.

D. *Data-Intensive Workloads*

Data-intensive workloads are used to exercise the Lustre file system and generate I/O operations, which is necessary to age the system and bring it to a state that may be difficult to recover from if failure events happen. The workloads can be either unmodified HPC applications, or customized programs for triggering corner cases based on the domain knowledge of the target system. The I/O operations during the workload are transferred to the Virtual Device Manager as described in Section III-B.

E. *Post-Failure Checker*

The Post-Failure Checker examines the post-failure behavior of the system and checks if it can recover to a consistent state without data loss. For checking Lustre, we make use of LFSCK [3]. Note that other file systems also come with default checkers (e.g., CephFS fsck [14]). Besides, we can also embed self-checking information (e.g., checksum) in the data written to the devices to verify the data integrity [36], [37], or use simple data manipulation tools to examine the behavior (See Section IV-B).

IV. Preliminary Results

We have built a prototype of the framework for the Lustre file system. A cluster of seven virtual machines (VMs) were created for the experiments. All VMs were installed with CentOS 7. We built a Lustre file system (version 2.8) on five VMs, including one MGS/MGT node, one MDS/MDT node, and three OSS/OST nodes. Every node was equipped with one virtual device. The sixth VM was used for hosting the

Lustre Nodes	*build Lustre*	*cp*	*tar*	*rm*	*Montage/m101*											
					s1	*s2*	*s3*	*s4*	*s5*	*s6*	*s7*	*s8*	*s9*	*s10*	*s11*	*s12*
MGS/MGT	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
MDS/MDT	5	0.1	5	0.2	6	0.4	6	0.5	6	0.6	6	0.7	6	1	6	1
OSS/OST #1	8	0	14	0	14	28	14	66	14	66	18	66	18	94	56	94
OSS/OST #2	8	15	14	15	14	43	14	81	14	81	19	81	19	109	19	110
OSS/OST #3	8	0	16	0	16	24	16	24	17	24	21	24	21	49	58	49

TABLE II: Numbers of bytes (MB) written to different Lustre nodes during the building of Lustre and under different workloads. *Montage/m101* is split into twelve steps (i.e., *s1* to *s12*) to show the fine-grained write pattern.

Virtual Device Manager and the Failure State Emulator, while the last VM was used as a client for launching workloads and LFSCK. The Virtual Device Manager was built on top of the Linux SCSI target framework [4].

We first ran a set of workloads to uncover the internal I/O pattern of Lustre during normal operations (i.e., without failures). Table I summarizes the workloads. As shown in the table, we ran the *m101* from Montage [6] as the primary data-intensive workload to generate I/O operations on Lustre. Also, we ran a set of commonly used Linux data manipulating tools (*cp*, *tar*, *rm*) as micro benchmarks on Lustre. As of this writing, the framework has been able to log the command history of *all writes* on *all nodes*, including the commands incurred by building Lustre. We discuss the write pattern of these workloads logged by the framework in Section IV-A.

In addition, as an initial step to examine the failure handling of Lustre, we emulated the whole device failure on the MDS/MDT node, and ran LFSCK and a set of simple metadata update operations to check the post-failure behavior of Lustre. We discuss our observations in Section IV-B.

In the current prototype, we only log the history of write commands, which are essential for determining the failure states of devices. It is straightforward to extend to log all types of commands if more complete I/O information is desired.

A. Internal Pattern of Writes on Lustre Nodes without Failure

We ran the workloads in Table I to uncover the internal pattern of writes on all Lustre nodes at the disk command level. Table II summarizes the bytes written (in MB) to each node under the workloads as well as during the building of Lustre. We can see that MGT was written only during the building of Lustre (i.e., 3 MB), and none of the workloads changed the state of MGT later. This verifies that MGT only stores the configuration information of the file system. On the other hand, MDT was updated during *cp*, *tar*, *rm*, and every step of *Montage/m101*. This is because the workloads created, extended, and/or deleted files, which incurred updates to the metadata (e.g., creating/removing inodes) stored on MDT.

Figure 2 further shows the accumulated number of bytes written to different nodes during the workloads. We can see that the metadata updates (i.e., the MDS/MDT line) were orders of magnitude smaller than the user data written to OSSs/OSTs. Moreover, while the user data increased rapidly on OSSs/OSTs, the increase of metadata was very slow (i.e., the MDS/MDT line is almost flat). This observation confirms that Lustre is efficient in terms of metadata management.

B. Post-Failure Behavior

To study the behavior of Lustre under failures, we emulated a whole device failure (Section III-C) by logging out the virtual device on the MDS/MDT node. We then ran a set of metadata update operations as well as LFSCK to examine the post-failure behavior.

Table III summarizes our observations. Interestingly, despite of missing the MDT device, some operations still "completed". For example, the *lfs setstripe* command, which sets the striping pattern of a Lustre file, finished without errors. Similarly, we used *dd* to create a new file and wrote 800MB data to it. The operation "completed" successfully, and we did observe writes on one OST device (we used the default *stripe_count* 1). Since these operations must update the metadata, and there was no actual device on MDS/MDT, we suspected that Lustre buffered the updates in memory instead of committing them to the MDT device synchronously. We verified this by issuing *dd* command with the *sync* option, which enforces synchronous commit on every write. The command reported I/O errors immediately. In other words, Lustre was unaware of the missing of the MDT device until a synchronous commit to it.

Moreover, we ran LFSCK on the post-failure Lustre. Surprisingly, LFSCK completed without reporting the missing device. This behavior implies that in some scenarios (e.g., the entire MDT file-system structure is cached in memory), LFSCK may only check the in-memory state of Lustre without actually verifying the persistent storage. In the case of sudden device failure where the in-memory representation of the file system information is still available temporarily, LFSCK cannot detect the device failure immediately.

Our initial experiments on the post-failure behavior of Lustre have uncovered a potential vulnerability: the 800MB data written via the first *dd* command was committed successfully from the user's perspective, while the corresponding metadata updates was not durable at all (because there was no MDT device). Moreover, the LFSCK failed to detect the missing device immediately. In the case of the entire system failure (e.g., the accident happened at Texas Tech HPCC [8]), the data may be permanently lost.

V. Related Work

Existing methods for testing storage systems mainly include model checking [34], [18], formal methods [12], and automatic testing [36], [35], [24], [22]. While these techniques are effective for testing local storage systems, applying them to large-scale systems remain challenging. For example, model

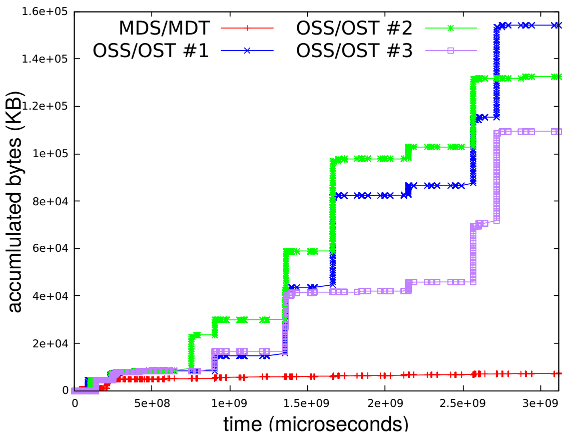

Fig. 2: Accumulated number of bytes (KB) written to different nodes during the workloads.

Operation	Description	Error?
lfs setstripe	set striping pattern	No
dd-nosync	create & extend a Lustre file	No
dd-sync	create & extend a Lustre file	Yes
LFSCK	check & repair Lustre	No

TABLE III: Operations ran on Lustre after a whole device failure on the MDS/MDT node. *dd-nosync* means using *dd* to create and extend a Lustre file with default buffering; *dd-sync* means enforcing synchronous writes on the *dd* command. The last column shows whether the operation reported error or not.

checking still faces the state explosion problem despite of various path reduction optimizations [33], [18], and turning the target system into a controllable model is a non-trivial task. Similarly, formally verifying the behavior of a large-scale system like Lustre is almost impossible in practice. As for automatic testing, the cases are more complicated due to the diverse testing methodologies. But generally, most of the testing frameworks are closely tied to the OS kernel, which makes them difficult to integrate with large-scale parallel file systems.

Besides, many studies have examined the bugs or failure behaviors of storage software and/or hardware (e.g., hard disks [13], [10], [9], [23], [29], [31], RAID [21], [20], flash-based SSDs [15], [16], [32], [36], [30], local file systems [25], [34], [19], databases and key-value stores [35], [24]). Generally, these studies provide valuable information for emulating realistic failure modes in our framework.

VI. Conclusions and Future Work

We have proposed and prototyped a generic framework for testing the failure handling of large-scale parallel file systems. The framework captures disk I/O commands on all storage nodes via virtual devices, emulates realistic failure states on the virtual devices, and checks the post-failure behavior of the target system. Our preliminary results on Lustre file system have uncovered its internal behaviors towards various workloads under both normal and failure conditions, which provides a solid foundation for further studying the failure handling of large-scale parallel file systems.

Although effective, the current prototype can only emulate one single device failure. Also, the operations we used to examine the post-failure behaviors of Lustre are relatively simple, which may not expose some potentially complicated defects. In the future, we would like to implement more failure modes (e.g., clean termination of writes, reordering or writes, corruption) and design more effective post-failure checking operations. Also, we plan to apply the failure modes to multiple nodes in the parallel file system to emulate large-scale accidence, which is more realistic in data centers. Moreover, we will explore the failure handling of other parallel file systems (e.g., Ceph, PVFS) besides Lustre. Eventually, based on the defects exposed by our testing framework, we would like to explore novel mechanisms to enhance the resilience of large-scale parallel file systems.

VII. Acknowledgments

The authors would like to thank the anonymous reviewers for their invaluable feedback. This research is partially supported by the NSF grant #CNS-1566554.

References

[1] NVM Express® over Fabrics Specification Released. http://www.nvmexpress.org/nvm-express-over-fabrics-specification-released/.
[2] Ceph File System. http://docs.ceph.com/docs/master/.
[3] LFSCK: an online file system checker for Lustre. https://github.com/Xyratex/lustre-stable/blob/master/Documentation/lfsck.txt.
[4] Linux SCSI target framework (tgt). http://stgt.sourceforge.net/.
[5] Lustre File System. http://lustre.org/.
[6] Montage: An Astronomical Image Mosaic Engine. http://montage.ipac.caltech.edu/.
[7] 2015 HPC User Site Census. http://www.intersect360.com/, 2016.
[8] Texas Tech University HPCC Power Outage Event. Email Announcement by HPCC at 8:50:17 AM Central Standard Time on Monday January 11, 2016.
[9] Bairavasundaram, L. N., Arpaci-Dusseau, A. C., Arpaci-Dusseau, R. H., Goodson, G. R., and Schroeder, B. An analysis of data corruption in the storage stack. *Trans. Storage 4*, 3 (Nov. 2008), 8:1–8:28.
[10] Bairavasundaram, L. N., Goodson, G. R., Pasupathy, S., and Schindler, J. 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* (New York, NY, USA, 2007), SIGMETRICS '07, ACM, pp. 289–300.
[11] Bairavasundaram, L. N., Sundararaman, S., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Tolerating file-system mistakes with envyfs. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2009), USENIX'09, USENIX Association, pp. 7–7.
[12] Chen, H., Ziegler, D., Chajed, T., Chlipala, A., Kaashoek, M. F., and Zeldovich, N. Using crash hoare logic for certifying the fscq file system. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 18–37.
[13] Chen, P. M., Lee, E. K., Gibson, G. A., Katz, R. H., and Patterson, D. A. RAID: high-performance, reliable secondary storage. *ACM Comput. Surv. 26*, 2 (June 1994), 145–185.
[14] Farnum, G. CephFS fsck: distributed file system checking. http://events.linuxfoundation.org/sites/events/files/slides/Vault%20CephFS%20fsck.pdf.
[15] Gabrys, R., Yaakobi, E., Grupp, L. M., Swanson, S., and Dolecek, L. Tackling intracell variability in TLC flash through tensor product codes. In *ISIT'12* (2012), pp. 1000–1004.

[16] GRUPP, L. M., CAULFIELD, A. M., COBURN, J., SWANSON, S., YAAKOBI, E., SIEGEL, P. H., AND WOLF, J. K. Characterizing flash memory: anomalies, observations, and applications. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture* (New York, NY, USA, 2009), MICRO 42, ACM, pp. 24–33.

[17] GUNAWI, H. S. *Towards Reliable Storage Systems*. PhD thesis, 2009.

[18] LEESATAPORNWONGSA, T., HAO, M., JOSHI, P., LUKMAN, J. F., AND GUNAWI, H. S. Samc: Semantic-aware model checking for fast discovery of deep bugs in cloud systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Broomfield, CO, Oct. 2014), USENIX Association, pp. 399–414.

[19] LU, L., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., AND LU, S. A study of linux file system evolution. In *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)* (San Jose, CA, 2013), USENIX, pp. 31–44.

[20] MA, A., DOUGLIS, F., LU, G., SAWYER, D., CHANDRA, S., AND HSU, W. Raidshield: Characterizing, monitoring, and proactively protecting against disk failures. In *13th USENIX Conference on File and Storage Technologies (FAST 15)* (Santa Clara, CA, Feb. 2015), USENIX Association, pp. 241–256.

[21] MA, A., TRAYLOR, R., DOUGLIS, F., CHAMNESS, M., LU, G., SAWYER, D., CHANDRA, S., AND HSU, W. Raidshield: Characterizing, monitoring, and proactively protecting against disk failures. *Trans. Storage 11*, 4 (Nov. 2015), 17:1–17:28.

[22] MIN, C., KASHYAP, S., LEE, B., SONG, C., AND KIM, T. Cross-checking semantic correctness: The case of finding file system bugs. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 361–377.

[23] NIGHTINGALE, E. B., DOUCEUR, J. R., AND ORGOVAN, V. Cycles, cells and platters: An empirical analysis of hardware failures on a million consumer PCs. In *Proceedings of the Sixth Conference on Computer Systems* (New York, NY, USA, 2011), EuroSys '11, ACM, pp. 343–356.

[24] PILLAI, T. S., CHIDAMBARAM, V., ALAGAPPAN, R., AL-KISWANY, S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. 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)* (October 2014).

[25] PRABHAKARAN, V., BAIRAVASUNDARAM, L. N., AGRAWAL, N., GUNAWI, H. S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. IRON File Systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP'05)* (Brighton, United Kingdom, October 2005), pp. 206–220.

[26] PRIMMER, M. An Introduction to Fibre Channel. *HP Journal* (1996).

[27] RENZELMANN, M. J., KADAV, A., AND SWIFT, M. M. Symdrive: Testing drivers without devices. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2012), OSDI'12, USENIX Association, pp. 279–292.

[28] SANDBERG, R., GOLGBERG, D., KLEIMAN, S., WALSH, D., AND LYON, B. Innovations in internetworking. Artech House, Inc., Norwood, MA, USA, 1988, ch. Design and Implementation of the Sun Network Filesystem, pp. 379–390.

[29] SCHROEDER, B., AND GIBSON, G. A. 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).

[30] SCHROEDER, B., LAGISETTY, R., AND MERCHANT, A. Flash reliability in production: The expected and the unexpected. In *14th USENIX Conference on File and Storage Technologies (FAST 16)* (Santa Clara, CA, Feb. 2016), USENIX Association, pp. 67–80.

[31] SUBRAMANIAN, S., ZHANG, Y., VAIDYANATHAN, R., GUNAWI, H. S., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., AND NAUGHTON, J. F. Impact of disk corruption on open-source DBMS. In *ICDE* (2010), pp. 509–520.

[32] TSENG, H.-W., GRUPP, L. M., AND SWANSON, S. Understanding the impact of power loss on flash memory. In *Proceedings of the 48th Design Automation Conference (DAC'11)* (2011).

[33] YANG, J., CHEN, T., WU, M., XU, Z., LIU, X., LIN, H., YANG, M., LONG, F., ZHANG, L., AND ZHOU, L. MODIST: Transparent model checking of unmodified distributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2009), NSDI'09, pp. 213–228.

[34] YANG, J., SAR, C., AND ENGLER, D. 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)* (November 2006), pp. 131–146.

[35] ZHENG, M., TUCEK, J., HUANG, D., QIN, F., LILLIBRIDGE, M., YANG, E. S., ZHAO, B. W., AND SINGH, S. Torturing databases for fun and profit. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Broomfield, CO, 2014), USENIX Association, pp. 449–464.

[36] ZHENG, M., TUCEK, J., QIN, F., AND LILLIBRIDGE, M. Understanding the robustness of SSDs under power fault. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST'13)* (2013).

[37] ZHENG, M., TUCEK, J., QIN, F., LILLIBRIDGE, M., ZHAO, B. W., AND YANG, E. S. Reliability analysis of ssds under power fault. In *To appear in the ACM Transactions on Computer Systems (TOCS)*.