Understanding and Finding Crash-Consistency Bugs in Parallel File Systems

Jinghan Sun, Chen Wang, Jian Huang, and Marc Snir
University of Illinois at Urbana-Champaign

Abstract

Parallel file systems (PFSes) and parallel I/O libraries have been the backbone of high-performance computing (HPC) infrastructures for decades. However, their crash consistency bugs have not been extensively studied, and the corresponding bug-finding or testing tools are lacking. In this paper, we first conduct a thorough bug study on the popular PFSes, such as BeeGFS and OrangeFS, with a cross-stack approach that covers HPC I/O library, PFS, and interactions with local file systems. The study results drive our design of a scalable testing framework, named PFSCHECK. PFSCHECK is easy to use with low performance overhead, as it can automatically generate test cases for triggering potential crash-consistency bugs, and trace essential file operations with low overhead. PFSCHECK is scalable for supporting large-scale HPC clusters, as it can exploit the parallelism to facilitate the verification of persistent storage states.

1 Introduction

Parallel file Systems are the standard I/O systems for large-scale supercomputers. They aim at maximizing storage capacity as well as I/O bandwidth for files shared by parallel programs. They scale to tens of thousands of disks accessed by hundreds of thousands of processes.

To avoid data loss, various fault tolerance techniques have been proposed for PFSes, including checkpointing [12,15,17] and journaling [3, 14]. However, crash consistency bugs are still happening and causing severe damages. For instance, a severe data loss took place at Texas Tech HPCC after two power outages in 2016, resulting in metadata inconsistencies in its Lustre file system [1]; the most recent crash in Lustre in the Stampede Supercomputer suspended its service for six days. As supercomputing time is expensive, both long recovery time and loss of data are expensive.

To understand crash consistency bugs, researchers have conducted intensive studies on commodity file systems [2,7,13,16]. Pillai et al. [16] investigated the characteristics of crash vulnerabilities in Linux file systems, such as ext2, ext4, and btrfs. To alleviate these bugs, recent studies have applied verification techniques to develop bug-free file systems [6,21]. Prior researchers also exploited model checking [4, 26] and fuzzing [11,25] techniques to pinpoint crash-consistency bugs with defined specifications. However, most of these prior studies focused on the regular file systems. Few of them can be directly applied to PFSes, due to the unique architecture of PFS, its workload characteristics, and the increased complexity of HPC I/O stack. Specifically, files can be shared by all processes in a parallel job, and I/O is performed through a variety of parallel I/O libraries which may have their own crash consistency implementation. A proper attribution of application-level bugs to one layer or another in the I/O stack depends on the "contract" between each layer.

To further understand crash consistency bugs in parallel I/O, we conduct a study with popular PFSes, including BeeGFS and OrangeFS. We manually create seven typical I/O workloads with the parallel I/O library HDF5 [5], and investigate crash-consistency bugs across the I/O stack. Our study results demonstrate that workloads on PFSes suffer from much (2.6-3.8$\times$) more from crash consistency bugs than local file system (see Table 2). We also develop a study methodology to distinguish bugs in the PFS from bugs in the parallel I/O library, using a formal definition of the crash consistency contract. We find that the number of crash-consistency bugs in parallel I/O libraries is comparable to that in PFSes.

To identify crash-consistency bugs in various PFSes and parallel I/O libraries, and facilitate their design and implementation, we propose to develop a scalable and generic testing framework, named PFSCHECK, with four major goals: (1) PFSCHECK should be easy to use, with as much automation of testing as possible; (2) PFSCHECK should be lightweight, with minimal performance overhead; (3) PFSCHECK should be scalable, with the increasing complexity of PFS configurations; (4) PFSCHECK should be accurate, which can identify the exact locations of crash-consistency bugs.

To achieve these aforementioned goals, we develop PFSCHECK with four major components: (1) a PFS-specific automated test-case generator, which will automatically generate a limited number of effective test cases for crash consistency testing. The generator follows POSIX APIs and hide the underlying I/O library details from upper-level developers; (2) a lightweight logging mechanism, which will trace the essential file operations on both client and storage server side with low overhead; (3) a fast crash state exploration mechanism, which will efficiently prune the large number of crash states brought by parallel workloads and multiple servers; (4) a bug classifier, which will properly attribute the reported crash-consistency bugs to PFS or I/O library with the given crash consistency model. We wish to develop PFSCHECK into an open-source testing platform that can facilitate the

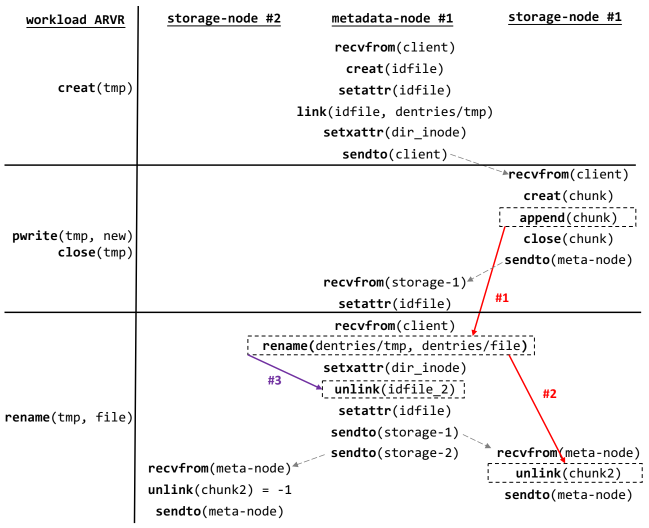

Figure 1: Crash-Vulnerability Examples in BeeGFS.

testing of developed and developing parallel file systems and parallel I/O libraries.

2 Background and Motivation

2.1 Crash Consistency in in HPC I/O

PFSes, such as BeeGFS [10], OrangeFS [23], Lustre [19], and GPFS [18], have been the main I/O infrastructure for supercomputers. PFSes were designed for different hardware configurations, and different workloads than distributed file systems (DFSes) like GFS [9] and HDFS [20, 22]. Specifically, the I/O stack of PFSes has three different properties. First, while the "one file per process" use mode is common, PFSes are optimized for shared access to a single file by all processes in a large-scale parallel computing setting. A large file may be stripped across all storage servers. This brings challenges to enforce the ordering of filesystem operations across large-scale server machines. Second, the metadata and data blocks are managed separately. Their complicated coordination causes crash-consistency bugs, even if each server has employed transactional mechanisms to enforce local atomic storage operations. Third, PFSes have a deep software stack, which often includes specific parallel I/O libraries. For example, an application may use HDF5 [5] that invokes MPI-IO [8] to execute POSIX I/O operations on the PFS.

The post-crash behaviors of an I/O system depend on what has been persisted before failures. Typically, PFSes rely on local file systems to hold data on each storage server, and the I/O library relies on the PFS. Thus, we need to conduct data recovery from persisted data structures layer by layer: the local file system first, followed by the PFS, and then the parallel I/O library. Crash consistency specifies what conditions should be satisfied by each one of these layers – what is the contract it obeys. The upper layer will perform its own recovery assuming that the layer below satisfied its contract. However, POSIX APIs used by file systems do not specify this contract [4], nor do PFSes clarify what they guarantee after crash recovery.

In this paper, we assume that the proper contract, which we call *prefix crash consistency*, is that the state of the file system after recovery is that obtained by executing correctly a prefix of the sequence of the I/O calls before the crash and none of the operations following this prefix. This prefix includes all `fsync` operations preceding the crash, the last operation in this prefix may be only partially completed, if it is not atomic. A similar definition can be provided for other I/O systems – one needs to define which operations are atomic and which are "persist" operations. If the I/O interface supports concurrent calls, then the order between I/O calls is the partial causality order, rather than a cut through a sequence.

Table 1: Test suites used in our study.

Test suite	Initializer	Workload (s)	Checking Items
ARVR	file with old content	file renamed from a tmp file with new content	atomicity of replacement
WAL	file with old content	logging before updating file, then remove log	update atomicity; otherwise check log content
HDF5	two groups and two datasets	create; delete; rename; resize dimension; h5part write dataset	atomicity; readability of existing datasets and groups;

2.2 Motivating Examples

We present a crash-vulnerability scenario of PFS in Figure 1. The application attempts to replace the content of a target file by creating, modifying, and renaming a temporary file in BeeGFS (see the detailed setting in § 3.1). The file operations are issued by BeeGFS client, and forwarded to the corresponding server machines. In this motivating scenario, we reveal three crash inconsistency states. Two of them are caused by cross-node persistence reordering, one is caused by the intra-node persistence reordering.

(1) When the `append` to the file on storage node #1 is persisted to the disk after the `rename` of directory entry on the metadata node, a crash happens right before the `append`. In this case, BeeGFS will suffer from data loss. This introduces crash vulnerability to the application, since its assumed atomicity is broken. (2) Similar inconsistency will happen if reordering happens between `rename` on the metadata node and `unlink` of the file on the storage node. Both of these two inconsistencies cannot be resolved by `beegfs-fsck`. (3) The third crash vulnerability is caused by the intra-node persistence reordering on the metadata node between `rename` and `unlink`. Two directory operations cannot be reordered on ext4, but this may occur on btrfs. Remounting BeeGFS upon such a crash will result in an inconsistency state – the original file cannot be opened. Fortunately, this inconsistency can be handled by `beegfs-fsck`.

Table 2: Distribution of crash vulnerabilities in the file system BeeGFS, OrangeFS, and ext4.

File system	ARVR	WAL	H5-create	H5-delete	H5-resize	H5-rename	H5-write	Total
BeeGFS	2	2	0	2	2	5	2	15
OrangeFS	1	2	0	9	1	6	0	19
ext4	0	0	0	1	1	3	0	5

3 PFS Crash-Vulnerability Characterization

To further understand the crash consistency in parallel I/O systems, we run real parallel I/O test cases and investigate the causes and consequences of identified crash vulnerabilities.

3.1 Study Methodology

We run I/O workloads on two different PFSes: BeeGFS and OrangeFS (formerly known as PVFS2). Both of the PFSes are configured with one metadata server and two storage servers. Each storage server runs ext4 locally with journaling enabled. The file striping size of storage servers is 64KB. BeeGFS stores its metadata with extended attributes by default. As for OrangeFS, we use its default Berkeley DB to store metadata.

We show the test suites used in our study in Table 1. They include the atomic-replace-via-rename (ARVR), write-ahead-logging (WAL) [4] scenarios that have been used in prior studies for testing file systems, and several workloads that use HDF5 library [5]. For each workload, we make every effort to recover it after the crash. For those tests with ARVR and WAL, we run the file system checker (`fsck`) of each PFS. For HDF5, we perform recovery with the official tool `h5clear` in addition to `fsck`. We report the crash vulnerabilities that cannot be fixed by existing recovery mechanisms.

3.2 Study Results

Crash vulnerability distribution. As both BeeGFS and OrangeFS are POSIX-compliant, we compare them with the local file system ext4 under the same setting of test cases. We show the results in Table 2. As we expected, the number of crash vulnerabilities in parallel file systems is more (2.6–3.8$\times$) than that in local file systems. These vulnerabilities can be easily triggered by a simple file operation, such as `delete`, `resize`, and `rename`. As most of the PFSes have the similar system architecture, and crash consistency guarantees, we observe that the number of crash vulnerabilities in BeeGFS is on a par with that of OrangeFS.

Crash vulnerability consequence. Similar to the consequences of crash vulnerabilities reported in commodity file systems [13], the crash vulnerabilities in PFSes will cause severe consequences as well, including data loss and inability to open/read files. Note that all the reported cases of crash inconsistency cannot be resolved with the existing recovery tools, such as `fsck` tool provided by PFSes and `h5clear`. This highly motivates us to develop new testing or bug-finding tools for parallel file systems, which will be discussed in § 4.

Crash vulnerability causes. To further investigate the root causes of these reported crash vulnerabilities in PFSes, we log the file system operations in both client and back-end metadata/storage servers when running each test case, and check the consistency of storage states after replaying them in different orders.

We categorize the crash states of PFSes into three types: (1) atomicity-related issues (AR) due to the incomplete execution of an PFS atomic operation, but persistence order obeys the happens-before order; (2) intra-node reordering (INR), in which the storage states are caused by the reordered storage operations in the local file system; (3) cross-node reordering (CNR), in which the storage states are caused by the reordered storage operations across multiple storage servers.

When we check the crash states with the traced storage operations, we begin with the checking of AR, then INR and CNR. We only report unique crash states in the study. The redundant crash states, for instance, an INR on a inconsistent AR state, are not reported.

When testing the stack of PFS+HDF5, we distinguish the reordering of the storage operations that satisfy the prefix crash consistency requirement for the PFS from those that do not. If the bug appears only for the reordering of the later category, we attribute it to the PFS; if it appears for the reordering in the first category, we attribute it to HDF5.

We report the detailed study results in Table 3. We observe that (1) the number of crash vulnerabilities caused by the CNR is larger than that caused by INR. This is due to the difficulty of executing operations across multiple servers in proper order. (2) AR vulnerabilities are a large portion (58.8%) of the crashes, this is because most of PFSes today do not provide transactional guarantees for user-level file operations. (3) The crash vulnerabilities can be due to both PFS (55.8%) and HDF5 (44.1%). HDF5 would introduce a large number of crash vulnerabilities, even if the underlying PFS fully enforced crash consistency.

To identify crash vulnerabilities, prior works have developed model checking, system verification, and testing techniques in commodity file systems. However, they cannot be directly applied to the PFS stack due to the unique design and implementation of PFSes, motivating us to exploit an alternative approach to address this challenge.

Table 3: Causes and consequences of crash vulnerabilities in PFSes. We show the vulnerability distribution among different type of root causes in the 4-6th column (Left:OrangeFS/Right:BeeGFS), the vulnerability location in 7-8th column (Left:OrangeFS/Right:BeeGFS), and the consequence in 9th column.

Workloads	# of vulnerabilities		Crash State			Root Cause		Consequences
	OrangeFS	BeeGFS	INR	CNR	AR	PFS	HDF5	
ARVR	1	2	-	1 / 2	-	1 / 2	-	data loss
WAL	2	2	1 / 0	1 / 2	-	2 / 2	-	removed or uncreated log
H5-create	0	0	-	-	-	-	-	n/a
H5-delete	9	2	-	2 / 1	7 / 1	8 / 0	1 / 2	OrangeFS unavailable; dataset unreadable
H5-resize	1	2	-	0 / 1	1 / 1	-	1 / 2	unable to read resized dataset
H5-rename	6	5	-	1 / 1	5 / 4	1 / 1	5 / 4	atomicity violation; link info error
H5-write	-	2	-	- / 1	- / 1	- / 2	-	unable to access data group
Total	19	15	1 / 0	5 / 8	13 / 7	12 / 7	7 / 8	

4 PFSCheck Design

In this section, we present PFSCheck, which aims to efficiently test PFSes for identifying the crash vulnerabilities in their design and implementation. PFSCheck will automatically generate parallel I/O test cases, check crash consistency of storage states after running each test, and pinpoint the root cause in the storage stack.

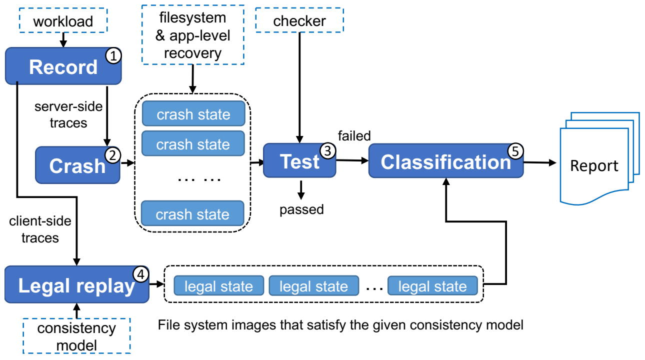

Figure 2: The system architecture of PFSCheck.

4.1 PFSCheck Overview

We demonstrate the system architecture of PFSCheck in Figure 2. PFSCheck takes five steps to check crash vulnerability for each benchmark. In the first phase (①), PFSCheck will trace both I/O and network operations when running each test case. The trace includes both server-side and client-side operations. PFSCheck will perform correlation analysis to associate each client request to the corresponding storage operations on PFS storage servers. In the second phase (②), PFSCheck will generate all possible crash states, based on the recorded execution. We will apply pruning algorithm to avoid redundancy of crash states in the search space. In the third phase (③), PFSCheck will recover the system from crash states, and then test its consistency with multiple checkers (e.g., `fsck`, `h5check`). To further identify the root cause of the reported crash vulnerability, PFSCheck will perform *legal replay* in the fourth phase (④), it will produce legal crash states where PFS obeys given crash consistency models. PFSCheck will utilize these legal crash states to classify whether the reported crash vulnerability happens in parallel I/O library or not (⑤). We will further discuss each of these phases as follows, respectively.

4.2 Generating Test Cases Automatically

Generating test cases in a manual manner will not be sufficient to systematically reveal crash consistency bugs in PFSes. In PFSCheck, we develop an automatic test-case generation mechanism by minimizing the efforts required from developers and testers. To achieve this, we take two essential steps to generate test cases. First, we use the popular POSIX APIs to hide the diversity of parallel I/O libraries. Given a specific parallel I/O library in our test setting, PFSCheck will transfer the POSIX-based I/O programs to the lower-level parallel I/O programs. This will not only ease the programmability of generating new test cases, but also enable the code reuse for different PFSes and parallel I/O libraries. Second, PFSCheck will adjust the parameters (e.g., access granularity) used in the test cases based on the PFS configurations, such that we can explore corner cases in our testing.

4.3 Recording I/O Operations

After we generate essential test cases, PFSCheck will trace the I/O operations to explore crash states in the follow-up analysis. It will trace at least two types of I/O operations for PFS services: storage operations that modified the storage state, and the network operations for the communications between metadata and data servers. This is because they will determine the happens-before order of these I/O operations. The order we observe on the server side will reflect one possible interleaving of client-side parallel I/O operations. In order to explore all possible interleavings, PFSCheck uses the *Recorder* tracing tool [24] to trace all relevant client-side

I/O operations (including HDF5, MPI-IO and POSIX), and then build a mapping between each client-side operation and server-side operations.

4.4 Emulating Crash States

As discussed in § 4.3, PFSCheck will record the I/O operations involved in each test case. However, different persistence ordering of those I/O operations may expose different crash vulnerabilities across the PFS stack. In order to cover all potential crash vulnerabilities, a straightforward way is to use the brute-force approach to explore all possible crash states of these I/O operations. However, this will produce a large exploration space, and is time consuming. To address this challenge, we will determine which I/O operations can be reordered by studying the history of executed test cases. Specifically, PFSCheck will replay the collected I/O operation trace, and reorder these I/O operations during the replay. PFSCheck will maintain a list to track the reordered cases that have been executed, such that we can avoid redundant emulations. Additionally, in order to avoid frequent PFS restart, we also propose to provide an incremental crash state reconstruction mechanism to reduce the number of restart operations. In the reconstruction, PFSCheck will also be careful with server-side cache coherence to its disk state, it will invalidate the cache when necessary.

4.5 Verifying Storage States

After we finish the emulation of crash states, PFSCheck will check their data consistency. It performs necessary recovery for each crash state by running multiple recovery tools (e.g., `fsck`, `h5clear`) after remounting the PFS. PFSCheck allows developers to use library-specific data recovery tools to test the crash consistency for different parallel I/O libraries. If these tools fail to recover the PFS or access the user data, we assume that crash vulnerabilities happen.

4.6 Pinpointing the Root Cause

As discussed, crash vulnerabilities could happen across the entire PFS stack. Their root causes may lie in parallel I/O libraries, or the PFS, or the local file system running on each storage server. To help PFSCheck pinpoint the root cause of these crash vulnerabilities, we propose the *legal replay* technique, in which it produces legal storage states where PFS exactly follows the prefix crash consistency model. For each legal storage state, PFS will also satisfy its contract with the upper-level parallel I/O library. In this case, if a vulnerable crash state matches with any of those legal states, PFSCheck believes that the vulnerability is located in the parallel I/O library. This is because the legal storage state has indicated that there is no crash inconsistency happening in the PFS. If a vulnerable crash state does not match with any of those legal states, PFSCheck believes that the vulnerability is located in the PFS. Therefore, legal replay performs as a classifier in PFSCheck to pinpoint the location of the crash vulnerabilities. With all these proposed components, PFSCheck is able to detect PFS crash vulnerabilities in an automatic, efficient, and accurate way.

5 Discussion and Future Work

The work described in this paper suggests many future directions to strengthen the the development of PFSCheck.

PFSCheck Implementation. We plan to fully implement its key components with increased automation. We will enable PFSCheck to support more parallel I/O libraries, such as NetCDF and MPI-IO, as well as more parallel file systems like Lustre and GPFS. Our ultimate goal is to develop PFSCheck into a general testing framework, such that the community can use it to conduct crash-vulnerability tests for the existing and newly developed PFSes. We also plan to open source our PFSCheck framework, such that developers can extend it to support other I/O libraries and file systems.

PFSCheck Evaluation. We plan to evaluate the efficiency of PFSCheck at different aspects. They include (1) the performance that indicates how much time it will take to finish each test case; (2) the accuracy that indicates whether PFSCheck can identify the root cause of reported crash vulnerabilities; (3) the completeness that indicates whether PFSCheck can identify all the potential crash vulnerabilities; (4) the scalability that indicates whether PFSCheck can scale, as we increase the number of storage servers or update the configuration of PFS and I/O libraries.

Extend PFSCheck to Other Consistency Models. In this paper, we mainly focus on the crash consistency, we would like to extend PFSCheck to support other crash consistency models enabled in parallel file systems. Accordingly, we will update those algorithms used in our current crash state emulation and storage state verification.

6 Conclusion

In this paper, we explored crash consistency issues in parallel file systems. We conducted a characterization study of the crash vulnerabilities in popular parallel file systems. As expected, PFSes and I/O libraries suffer from more crash consistency bugs than regular file systems, due to the scale and complexity of the I/O stack. To this end, we propose to develop a generic testing framework for identifying these crash vulnerabilities. We outlined the design of PFSCheck for the systematic testing of parallel I/O systems and the proper attribution of issues to different layers of the I/O stack.

Acknowledgment

We thank the anonymous reviewers for their helpful comments and feedback. This work was partially supported by NSF grant CCF-1763540, CNS-1850317, and CCF-1919044.

References

[1] HPCC power outage event at Texas Tech. http://www.ece.iastate.edu/~mai/docs/failures/2016-hpcc-lustre.pdf, 2016.

[2] Ramnatthan Alagappan, Aishwarya Ganesan, Yuvraj Patel, Thanumalayan Sankaranarayana Pillai, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Correlated crash vulnerabilities. In *Proccedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*, Savannah, GA, 2016.

[3] Remzi H. Arpaci-Dusseau. Operating systems: Three easy pieces. 42(1), 2017.

[4] James Bornholt, Antoine Kaufmann, Jialin Li, Arvind Krishnamurthy, Emina Torlak, and Xi Wang. Specifying and checking file system crash-consistency models. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'16)*, Atlanta, GA, April 2016.

[5] Dhruba Borthakur. HDFS architecture guide. *Hadoop Apache Project*, 53(1-13), 2008.

[6] Haogang Chen, Tej Chajed, Alex Konradi, Stephanie Wang, Atalay İleri, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying a high-performance crash-safe file system using a tree specification. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)*, 2017.

[7] 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 (SOSP'15)*, Monterey, CA, 2015.

[8] Peter Corbett, Dror Feitelson, Sam Fineberg, Yarsun Hsu, Bill Nitzberg, Jean-Pierre Prost, Marc Snir, Bernard Traversat, and Parkson Wong. Overview of the MPI-IO parallel I/O interface. In *Proceedings of the Workshop on Input/Output in Parallel and Distributed Systems (IPPS'95)*, 1995.

[9] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, Bolton Landing, NY, 2003.

[10] Jan Heichler. An introduction to BeeGFS, 2014.

[11] Seulbae Kim, Meng Xu, Sanidhya Kashyap, Jungyeon Yoon, Wen Xu, and Taesoo Kim. Finding Semantic Bugs in File Systems with an Extensible Fuzzing Framework. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP'19)*, Ontario, Canada, October 2019.

[12] Jharrod LaFon, Satyajayant Misra, and Jon Bringhurst. On distributed file tree walk of parallel file systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'12)*, 2012.

[13] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. Finding crash-consistency bugs with bounded black-box crash testing. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*, pages 33–50, 2018.

[14] Sarp Oral, Feiyi Wang, David Dillow, Galen Shipman, Ross Miller, and Oleg Drokin. Efficient object storage journaling in a distributed parallel file system. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST'10)*, 2010.

[15] X. Ouyang, R. Rajachandrasekar, X. Besseron, H. Wang, J. Huang, and D. K. Panda. CRFS: A Lightweight User-Level Filesystem for Generic Checkpoint/Restart. In *Proceedings of the 2011 International Conference on Parallel Processing (ICPP'11)*, Taipei, Taiwan, 2011.

[16] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 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)*, Broomfield, CO, 2014.

[17] Kai Ren, Qing Zheng, Swapnil Patil, and Garth Gibson. IndexFS: Scaling file system metadata performance with stateless caching and bulk insertion. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'14)*, 2014.

[18] Frank B Schmuck and Roger L Haskin. GPFS: A shared-disk file system for large computing clusters. In *FAST*, volume 2, 2002.

[19] Philip Schwan et al. Lustre: Building a file system for 1000-node clusters. In *Proceedings of the 2003 Linux symposium*, volume 2003, pages 380–386, 2003.

[20] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop distributed file system. In *Proceedings of the IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST'10)*, 2010.

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

[22] Sayantan Sur, Hao Wang, Jian Huang, Xiangyong Ouyang, and Dhahaleswar K. Panda. Can high-performance interconnects benefit hadoop distributed file system. In *Proceedings of the Workshop on Micro Architectural Support for Virtualization, Data Center Computing, and Clouds (MASVDC'10)*, Atlanta, GA, 2010.

[23] OrangeFS Team. The OrangeFS project, 2015.

[24] Chen Wang, Jinghan Sun, Marc Snir, Kathryn Mohror, and Elsa Gonsiorowski. Recorder 2.0: Efficient parallel I/O tracing and analysis. In *The IEEE International Workshop on High-Performance Storage*, 2020.

[25] Wen Xu, Hyungon Moon, Sanidhya Kashyap, Po-Ning Tseng, and Taesoo Kim. Fuzzing File Systems via Two-Dimensional Input Space Exploration. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland'19)*, San Francisco, CA, May 2019.

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