PFault: A General Framework for Analyzing the Reliability of High-Performance Parallel File Systems

Jinrui Cao, Om Rameshwar Gatla, Mai Zheng
New Mexico State University
{will_cao,omram,zheng}@nmsu.edu

Dong Dai, Vidya Eswarappa, Yan Mu, Yong Chen
Texas Tech University
{dong.dai,vidya.eswarappa,yan.mu,yong.chen}@ttu.edu

ABSTRACT

High-performance parallel file systems (PFSes) are of prime importance today. However, despite the importance, their reliability is much less studied compared with that of local storage systems, largely due to the lack of an effective analysis methodology.

In this paper, we introduce PFault, a general framework for analyzing the failure handling of PFSes. PFault automatically emulates the failure state of each storage device in the target PFS based on a set of well-defined fault models, and enables analyzing the recoverability of the PFS under faults systematically.

To demonstrate the practicality, we apply PFault to study Lustre, one of the most widely used PFSes. Our analysis reveals a number of cases where Lustre's checking and repairing utility LFSCK fails with unexpected symptoms (e.g., I/O error, hang, reboot). Moreover, with the help of PFault, we are able to identify a resource leak problem where a portion of Lustre's internal namespace and storage space become unusable even after running LFSCK. On the other hand, we also verify that the latest Lustre has made noticeable improvement in terms of failure handling comparing to a previous version. We hope our study and framework can help improve PFSes for reliable high-performance computing.

CCS CONCEPTS

• **Computer systems organization → Reliability**;

KEYWORDS

Parallel file systems, reliability, high performance computing

ACM Reference Format:
Jinrui Cao, Om Rameshwar Gatla, Mai Zheng and Dong Dai, Vidya Eswarappa, Yan Mu, Yong Chen. 2018. PFault: A General Framework for Analyzing the Reliability of High-Performance Parallel File Systems. In *ICS '18: 2018 International Conference on Supercomputing, June 12–15, 2018, Beijing, China.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3205289.3205302

1 INTRODUCTION

Storage systems must handle failures (e.g., power outages, server crashes, device errors) gracefully, which is difficult to achieve in practice. For example, recent studies have uncovered reliability vulnerabilities at almost every layer of the local storage stack (e.g., devices [54], RAID [25], local file systems [27]). This raises the concern for PFSes (e.g., Lustre [23], OrangeFS [46]) building on top of the local storage stack, which are responsible for managing much larger volumes of data in many high-performance data centers.

In fact, in a recent incident at the High Performance Computing Center (HPCC) in Texas [16], the storage clusters managed by Lustre file systems suffered severe data loss after power outages [19]. Although many files have been recovered after months of manual efforts, there is still critical data lost permanently, and the potential damage to the scientific discovery is unmeasurable. Similar failure events have been reported in other recent studies on large-scale production systems [14]. In an effort to identify the potential vulnerabilities in Lustre and its repair utility LFSCK, and to improve the reliability of PFSes in general, we introduce PFault, a general framework for analyzing the failure handling of PFSes, in this paper.

There are two major challenges. The first one is how to generate faults at scale in a systematic and controllable way. Manually unplugging the power cord for thousands of times is simply impractical. The second challenge is the complexity. Even a local file system may consist of many tens of thousands of lines of kernel-space code. Adding a global layer across many already-complicated local systems makes analyzing the whole system behavior difficult. Moreover, PFSes may have strong dependency on local systems. For example, Ext4 is patched for Lustre's ldiskfs backend [24], while it is unusable for Ceph OSD Daemons [44].

To address the challenges, we make two key observations. First, failure events may vary, but only the on-drive persistent states may affect the recovery after rebooting. Therefore, we boil down the generation of various failure events to the emulation of the device state on each node.

Second, despite the complexity of PFSes, we can always separate the whole system into a global PFS layer across multiple nodes, and a local system layer on each individual node. Moreover, the local system layer can be largely decoupled from the rest of the system through remote storage protocols (e.g., iSCSI [45], NVMe/Fabric [32]), which have already been used in large-scale storage clusters for easy management of storage devices. In other words, by emulating the failure state of each individual node via remote storage protocols, we enable analyzing different PFSes with little disturbance and porting effort.

Based on the ideas above, we build a prototype of PFault on top of a customized iSCSI driver. PFault first creates a virtual device on each storage node of the target PFS via iSCSI, and then

ICS '18, June 12–15, 2018, Beijing, China
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5783-8/18/06…$15.00
https://doi.org/10.1145/3205289.3205302

manipulates the backing stores to emulate the failure states on virtual devices under certain workloads and fault models. We explore a wide set of failure events that need to be handled properly in practice, and summarize them into three fault models: *whole device failure*, *global inconsistency*, and *network partitioning*. After emulating failure states, PFault invokes the checking and repairing utility of the PFS as well as a set of customized workloads to examine the recoverability of the target PFS.

To demonstrate the practicality, we apply PFault to analyze Lustre [23], one of the most widely used PFSes in high-performance computing facilities. We find that LFSCK, the checking and repairing utility of Lustre, may behave abnormally under different faults (e.g., hang, fail to start, or trigger the rebooting of storage nodes).

Moreover, with the help of PFault, we are able to find that a portion of Lustre's internal namespace as well as the storage space may become unusable after faults, which we refer to as a *resource leak problem*. The leaked resource cannot be detected or reclaimed by LFSCK. To address the problem, we build a simple tool called *LeakCK*, which can detect leaked internal data files of Lustre based on their reachability from the client.

In addition, we find that Lustre and LFSCK generate extensive logs for diagnosing the system behavior under faults. While many logs are accurate and informative, some others are not. Such status implies that detecting and handling failures properly in large-scale PFSes remains a challenging task. By releasing the prototype [1], we hope PFault can help improve Lustre as well as other PFSes for reliable high-performance computing.

The rest of the paper is organized as follows. In §2, we introduce the background. In §3, we describe the design and implementation of PFault. We present our analysis of Lustre in §4. Finally, we discuss related work (§5) and conclude (§6).

2 BACKGROUND

2.1 Parallel File Systems

Parallel file systems (PFSes) are designed for high-performance computing (HPC) in a well controlled environment, which leads to an architecture different from other distributed storage systems (e.g., HDFS [15]). For example, PFSes are optimized for highly concurrent accesses to the same file, and they heavily rely on hardware like RAID [33] to protect data against failure events. Although traditionally high performance is the most desired metric of PFSes, high reliability is also becoming increasingly important as more and more critical data are kept in various PFSes today.

The goal of PFault is to help improve the failure handling of PFSes by systematically emulating faults and facilitating post-fault analysis. To this end, we use Lustre[24], one of the most widely used PFSes, as an example to apply PFault in this work. Lustre dominates the market share of HPC centers [18], and more than half of the top 100 supercomputers use Lustre [47]. Similar to other PFSes, a Lustre file system usually includes the following components:

- **Management Server (MGS) and Management Target (MGT)** manages and stores the configuration information of Lustre. Multiple Lustre file systems in one cluster can share the MGS and MGT.
- **Metadata Server (MDS) and Metadata Target (MDT)** manages and stores the metadata of Lustre. MDS provides network request handling for one or more local MDTs. There can be multiple MDSs and MDTs since Lustre v2.4. Also, MGS/MGT can be co-located with MDS/MDT.
- **Object Storage Server (OSS) and Object Storage Target (OST)** manages and stores the actual user data. OSS provides the file I/O service and the network request handling for one or more local OSTs. User data are stored as one or more objects, and each object is stored on a separate OST.
- **Clients** launch applications to access the data in Lustre, usually from login nodes or compute nodes.

Although Lustre is optimized for the HPC environment, maintaining consistency and data integrity under faults is becoming more and more desirable as the scale and complexity increases. To this end, Lustre includes a failover feature, which allows MDS/OSS to continue execution on a standby node after crashes. Moreover, Lustre introduces an online utility called LFSCK [21] for checking and repairing itself after faults, which has been significantly improved since v2.6.

A typical deployment of Lustre may include one MGS node, one or two dedicated MDS node(s), and two or more OSS nodes, as shown in Figure 1. We follow such setting in our experiments (§4).

2.2 Remote Storage Protocols

Remote storage protocols (e.g., NFS [38], iSCSI [45], Fibre Channel [36], NVMe/Fabric [32]) enable accessing remote storage devices as local devices, either at the file level or the block level. In particular, iSCSI[45] is an IP-based protocol allowing one machine (the initiator) to access the remote block storage through the internet. 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.

Compared to other protocols, iSCSI does not require special hardware (unlike Fibre Channel), works at the block level (unlike NFS), and is mature (unlike NVMe/Fabric). Therefore, in this work we use iSCSI to decouple the major components of PFault from the target system, which minimizes the disturbance and enables testing different parallel file systems with little porting effort.

3 DESIGN AND IMPLEMENTATION OF PFAULT

3.1 Overview

Figure 1 shows the overview of PFault and its connection with a target PFS under analysis. We use Lustre as an example of PFS, which includes three types of storage nodes (i.e., MGS/MGT, MDS/MDT, OSS/OST) as described in Section 2.1.

There are four major components in PFault: (1) the *Virtual Device Manager* mounts a set of virtual devices to the storage nodes via iSCSI, and forward all disk I/O commands to the backing files; (2) the *Failure State Emulator* manipulates the backing files and emulates the failure state of each virtual device based on the workloads and fault models; (3) the *PFS Worker* launches workloads to exercise the PFS and generate I/O operations; (4) the *PFS Checker*

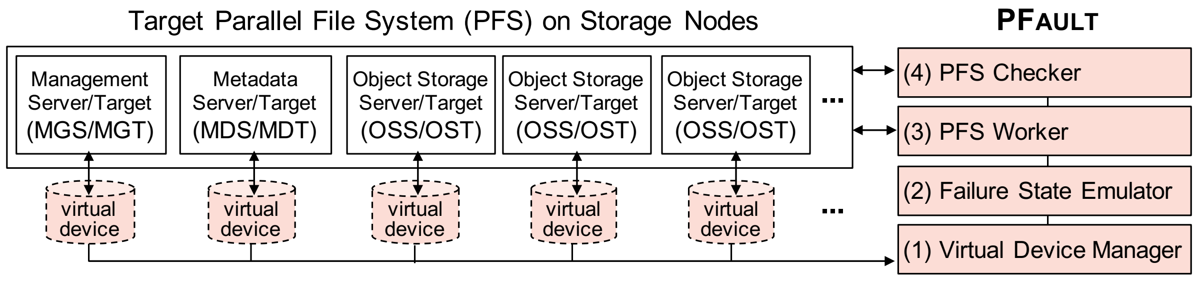
Target Parallel File System (PFS) on Storage Nodes
PFAULT
Management Server/Target (MGS/MGT)
Metadata Server/Target (MDS/MDT)
Object Storage Server/Target (OSS/OST)
Object Storage Server/Target (OSS/OST)
Object Storage Server/Target (OSS/OST)
...
virtual device
virtual device
virtual device
virtual device
virtual device
...
(4) PFS Checker
(3) PFS Worker
(2) Failure State Emulator
(1) Virtual Device Manager

Figure 1: Overview of PFault. *The shaded boxes are the major components of PFault: (1) Virtual Device Manager mounts virtual devices to the storage nodes of the PFS via iSCSI; (2) Failure State Emulator emulates failure states under certain workloads and fault models; (3) PFS Worker exercises the PFS and generates I/O operations; (4) PFS Checker examines the recoverability of the PFS.*

invokes the checking and repairing utility of the PFS and a set of verifiable workloads to examine the recoverability of the PFS.

Note that although Figure 1 uses Lustre as an example, the framework can be applied to other PFSes with little porting effort by installing the corresponding software on the virtual devices.

3.2 Virtual Device Manager

The state of the target PFS depends on the I/O operations. To capture all I/O operations in the PFS, the Virtual Device Manger creates and maintains a set of backing files, each of which is corresponding to one storage device used in the storage nodes. The backing files are mounted to the storage nodes as virtual devices via the iSCSI protocol [45]. Thanks to iSCSI, the virtual devices are just ordinary local disks from the PFS' perspective. In other words, PFault is transparent to the PFS under analysis.

All I/O operations in the PFS are eventually translated into low-level disk I/O commands, which are transferred to the remote Virtual Device Manager via iSCSI. The Virtual Device Manager updates the content of the backing files according to the commands received and satisfies the I/O requests accordingly.

3.3 Failure State Emulator

To analyze the failure handling of the PFS, it is necessary to generate faults in a systematic and controllable way. To this end, PFault reduces various failure events to the states of virtual devices via the Failure State Emulator. Specifically, for each storage node with a virtual device, PFault manipulates the corresponding backing file and the network daemon based on a set of fault models. The current prototype of PFault includes three representative fault models defined as follows:

(a) Whole Device Failure (a-DevFail). This is the case when a storage device becomes inaccessible to the PFS entirely, which can be caused by a number of reasons including RAID controller failures, firmware bugs, accumulation of sector errors, etc [2, 25, 54].

Since PFault is designed to decouple the PFS from the virtual devices via iSCSI, we can simply log out the virtual devices to emulate this fault model. More specifically, PFault uses the *logout* command in the iSCSI protocol (§2.2) to disconnect the backing file to the corresponding storage node, which makes the device inaccessible to the PFS immediately.

(b) Global Inconsistency (b-Inconsist). In this case, all storage devices are still accessible to the PFS, i.e., the I/O requests from the PFS can be satisfied normally. Also, the local file system (e.g., Ext4) on each storage node is consistent. However, the global state of the PFS, which consists of all local states, is inconsistent from the PFS' perspective.

Because PFSes are built on top of local file systems, it seems unreasonable to expect a PFS to behave properly when a local file system is broken. Therefore, in this model we intentionally enforce that each local file system must be consistent. Such scenario (i.e., local file systems are consistent, but the global state is incorrect) may also be caused by a variety of reasons. For example, in a data center-wide power outage [19], the local file systems on individual storage nodes may be corrupted to different degrees depending on the local I/O operations at the fault time. Therefore, it is necessary to check and repair the local file systems first [21]. Since the local file system checker (e.g., e2fsck [50] for Ext4) only has the knowledge of its own metadata consistency rules, it can only fix the local inconsistencies on a single node, and thus expose a potentially inconsistent global state to the PFS. Besides power outages, local file systems may be corrupted for other reasons such as file system bugs, latent sector errors, etc [2, 35, 54]. While running a local file system checker may bring the local file system back to a consistent state, it may (unintentionally) break the global consistency rules due to its local repair operations (e.g., skipping incomplete local journal transactions, moving a local file to the "lost+found" directory). As a result, the global consistency across nodes may be compromised.

To emulate this fault model, PFault first invokes the debug tool of the local file system (e.g., debugfs [10] for Ext4 and xfs_db for XFS) to manipulate the local states on selected nodes. These debug tools allow "trashing" specific metadata structures of the local file system for regression testing. We make use of such feature to corrupt the local file systems. After introducing local corruptions, we invoke the checking and repairing utility of the local file system to repair any local inconsistencies.

Moreover, after repairing the local file system, we compare it with the original local file system (before corruption). This is to verify if the repaired local state is different from the original state. If it is different, then we have successfully generated a different but consistent local file system. PFault remounts this new local file system to the PFS to create an inconsistent global state.

(c) Network Partitioning (c-Network). Network partitioning is a typical failure in large-scale networked systems, which may be caused by dysfunctional network devices (e.g., switch) or hanging server processes among others [29]. When the failure happens, the cluster splits into more than one "partitions" which cannot communicate with each other.

To emulate this effect, PFault disables the network card(s) used by the PFS on selected nodes through the network daemon, which effectively isolates the selected nodes to the rest of the system.

Summary & Expectation. The three fault models defined above represent a wide range of real-world failure scenarios [2, 3, 25, 29, 30, 39, 43, 53, 54]. By emulating these models automatically, PFault enables analyzing the failure handling of the target PFS thoroughly. Since there are usually multiple types of storage nodes in a PFS, a failure may affect the PFS in different ways depending on the types of nodes affected. Therefore, PFault allows specifying which types of nodes to apply the fault models through a configuration file.

Note that in all three models, we intentionally avoid introducing inconsistencies at the local file system level. However, it is straightforward to extend PFault to emulate more aggressive scenarios with inconsistent local file systems We leave emulating other fault models as future work.

Also, since many PFSes are optimized for high performance, it is perhaps acceptable if the target PFS cannot function normally when experiencing these faults. However, we expect the failure handling utility of the PFS (e.g., LFSCK for Lustre) to be able to detect the potential issues and response properly.

3.4 PFS Worker

Comparing with a fresh file system, an aged file system is more representative [8, 42], and is more likely to encounter issues under faults due to the more complicated internal state. Therefore, the PFS Worker invokes data-intensive workloads (e.g., unmodified HPC applications) to age the target PFS and generate a representative state before injecting faults. Internally, the PFS distributes the I/O operations to storage nodes, which are further transferred to the Virtual Device Manager as described in §3.2.

Besides unmodified data-intensive workloads, another type of useful workloads is customized applications specially designed for examining the failure handling capability of the PFS. For example, the workload may embed checksums in the data written to the PFS, which can help identify the potential corruption of user files from the end user's perspective without relying on the reports of the PFS. We include examples of both types of workloads in §4.

3.5 PFS Checker

Similar to local file systems, maintaining consistency and data integrity is critical for large-scale file systems. Therefore, most multi-node file systems include a checking and repairing utility to serve as the last line of defense against global inconsistencies (e.g., LFSCK [21], Ceph FSCK [44]).

The PFS Checker component invokes the checking and repairing utility of the target PFS to recover the PFS after injecting faults. In addition, the PFS Checker also invokes customized checking workloads to access the recovered PFS, and examines the PFS' recoverability from the end user's perspective based on the responses of the workloads. We also include examples of customized checking workloads in §4.

4 CASE STUDY: LUSTRE FILE SYSTEM

4.1 Experimental Methodology

We have built a prototype of PFault and applied it to study Lustre. Two platforms were used for the experiments: (1) a seven-node cluster created on virtual machines hosted on our private servers; (2) a twenty-node cluster created on CloudLab [7], where the experimental profile is publicly available [49]. All results were collected using the first platform for convenience, and a subset with unexpected symptoms (e.g., hang, rebooting, resource leak) has been repeated and verified using the CloudLab profile.

In the seven-node cluster, one node was used for hosting the Virtual Device Manager and Failure State Emulator of PFault, and another node was used as a login/compute node to host PFS Worker and PFS Checker and launch workloads on behalf of clients. The remaining five nodes were dedicated to the PFS as storage nodes.

On the five storage nodes, we created a Lustre file system with one MGS node, one MDS node, and three OSS nodes. On each node, there was one virtual device mounted to serve as the corresponding target device (i.e., MGT/MDT/OST).

First, we designed a set of workloads to analyze Lustre's failure handling from the end user's perspective (e.g., whether a program finishes normally or not). We find that the behavior of LFSCK varies a lot under different faults (§4.2).

Next, we discovered a resource leak problem where a portion of the internal namespace as well as the storage space on OSTs becomes unusable by Lustre even after running LFSCK. We discuss the problem as well as our tool for detecting the leak in §4.3.

Finally, we collected the extensive logs generated by Lustre and LFSCK during the experiments. We find that some log messages are accurate and informative, but some others are not. We analyze the logs in details in §4.4 and §4.5.

We have analyzed two versions of Lustre in our experiments. The latest version when we started our study was v2.8. All results and analysis in §4 are based on v2.8, unless stated otherwise. Besides, we have repeated the same experiments on the latest v2.10. We describe both versions in §4.2 and §4.3. However, we have not finished analyzing the logs of v2.10 at the time of writing due to the extensive volume. Therefore, the log analysis in §4.4 and §4.5 only represents the logging mechanism of v2.8. Overall, we find that v2.10 has made noticeable improvement over v2.8 in terms of failure handling.

4.2 Behavior of LFSCK

In this section, we present the behavior of LFSCK as perceived by the end user when checking and repairing Lustre after faults. We defer the detailed analysis of Lustre's and LFSCK's internal debug logs to §4.4 and §4.5. Also, we apply a set of workloads *after* LFSCK to further verify the effectiveness of LFSCK. While we do not expect Lustre to function normally after the faults, we expect LFSCK to be able to detect the potential issues.

4.2.1 ***Workloads.*** Table 1 summarizes the workloads used in the experiment. `create+write+delete` is a set of common file

Workload	Description	Purpose
`create+write+delete`	create, write, & delete files	age Lustre
`Montage-m101`	an astronomical image mosaic engine	age Lustre
`WikiW-init`	write an initial set of Wikipedia files (w/ known MD5)	generate verifiable data
`WikiR`	read the initial Wikipedia files & verify MD5	analyze post-LFSCK behavior
`WikiW-async`	write new files asynchronously, read back & verify MD5	analyze post-LFSCK behavior
`WikiW-sync`	write new files synchronously, read back & verify MD5	analyze post-LFSCK behavior

Table 1: Workloads Used in PFault for Analyzing Lustre and LFSCK. *create+write+delete and Montage-m101 are used for aging Lustre; WikiW-init writes a verifiable set of Wikipedia files for fault injection; the other three workloads (i.e., WikiR, WikiW-async, WikiW-sync) are used for analyzing the behavior of Lustre after LFSCK from the end user's perspective.*

Node(s) Affected	Fault Models	LFSCK	`WikiR`	`WikiW-async`	`WikiW-sync`
MGS	a-DevFail	normal	✓	✓	✓
	b-Inconsist	normal	✓	✓	✓
	c-Network	normal	✓	✓	✓
MDS	a-DevFail	**Invalid**	hang	hang	hang
	a-DevFail (v2.10)	**I/O err**	I/O err	I/O err	I/O err
	b-Inconsist	normal	✓	✓	✓
	c-Network	**I/O err**	hang	hang	hang
	c-Network (v2.10)	**hang**	hang	hang	hang
OSS#2	a-DevFail	**hang**	hang	hang	hang
	a-DevFail (v2.10)	normal	✓	✓	✓
	b-Inconsist	**reboot**	corrupt	hang	hang
	c-Network	**hang**	hang	hang	hang
three OSSes	a-DevFail	**hang**	hang	hang	hang
	a-DevFail (v2.10)	normal	✓	hang	hang
	b-Inconsist	**reboot**	corrupt	hang	hang
	c-Network	**hang**	hang	hang	hang
MDS + OSS#2	a-DevFail	**Invalid**	hang	hang	hang
	a-DevFail (v2.10)	**I/O err**	I/O err	I/O err	I/O err
	b-Inconsist	**reboot**	corrupt	hang	hang
	c-Network	**I/O err**	hang	hang	hang
	c-Network (v2.10)	**hang**	hang	hang	hang

Table 2: Response of LFSCK and post-LFSCK Workloads. *The first column shows where the faults are injected. The second column shows the fault models applied (v2.10 means applying to Lustre v2.10). "normal": LFSCK finishes normally; "reboot": at least one OSS node is forced to reboot; "Invalid": an "Invalid Argument" error; "I/O err": an "Input/Output error"; "hang": cannot finish within one hour; " ✓ ": complete w/o error; "corrupt": checksum mismatch. The bold font highlights the unexpected response of LFSCK.*

operations (i.e., creating files, writing files, and deleting files) for aging the Lustre under testing. `Montage-m101` is a classic HPC application for creating astronomical image mosaics [28], which is also used for bringing Lustre to a representative state. The runtime of the aging workloads is tunable through a script, and we age Lustre for about one hour in all experiments.

The Wikipedia workloads (i.e., `WikiW-init`, `WikiR`, `WikiW-async`, and `WikiW-sync`) use a dataset consisting of the archive files of Wikipedia [51]. Each archive file has an official MD5 checksum for verifying its integrity.

To emulate failure states, `WikiW-init` writes a known initial set of archive files to Lustre. In the meantime, PFault generates failure states under `WikiW-init` based on the fault models (§3.3). By using MD5 checksums, we can verify the integrity of the initial files without relying on the reports of Lustre or LFSCK.

After emulating failure states, we run LFSCK to check and repair Lustre as it is designed to be an online tool. If LFSCK cannot finish within one hour (i.e., "hang"), we kill the LFSCK process. And after LFSCK, we run `WikiR`, `WikiW-sync`, and `WikiW-aync` one by one in sequence. `WikiR` reads the initial set of archive files, calculates the MD5 checksums, and verifies if the checksums match the official values. This workload checks the behavior of read operations after LFSCK. `WikiW-async` writes new archive files to Lustre asynchronously, reads the files back, and verifies the checksums. This is to check if asynchronous write operations can be served properly after LFSCK. Similarly, `WikiW-sync` checks the behavior of synchronous writes.

4.2.2 ***Observations.*** Table 2 summarizes the response of LFSCK and the workloads after LFSCK. As shown in the first column, we inject faults to five different sets of Lustre nodes: (1) MGS only, (2) MDS only, (3) OSS #2 only, (4) all three OSSes, and (5) MDS and OSS#2. For each set, we inject faults based on the three fault models (§3.3). We add the behavior of Lustre/LFSCK v2.10 (i.e., "v2.10" lines) when it differs from that of v2.8.

When faults happen on MGS (the "MGS" row), there is no user-perceivable impact. LFSCK finishes normally ("normal") and all workloads complete successfully ("✓"). This is consistent with Lustre's design that MGS is not involved in the regular I/O operations after Lustre is built [24].

When faults happen on other nodes, however, LFSCK may fail unexpectedly. For example, when "a-DevFail" happens on MDS (the "MDS" and "MDS+OST#2" rows), LFSCK fails with an "Invalid Argument" error ("Invalid") and all workloads cannot make progress ("hang") on v2.8. All these behaviors change to "'Input/Output error" ("I/O err") on v2.10, which is an improvement since "I/O err" is closer to the root cause (i.e., a whole device failure).

When "a-DevFail" happens on OSS (the "OSS#2" row), v2.8 and v2.10 differ a lot. On v2.8, LFSCK and all workloads hang. However, on v2.10, LFSCK finishes normally, and all workloads succeed (i.e., "✓"). `WikiR` can succeed because it reads the initial files buffered in the memory. We verify this by unmounting and remounting the file system, which purges the buffer cache. After remounting, running `WikiR` becomes "hang" (same as v2.8). This suggests that v2.10 has a more aggressive buffering mechanism compared with v2.8. `WikiW-sync` and `WikiW-aync` can succeed because v2.10 skips the missing OST and uses the remaining two OSTs for storing striped data. We verify this by analyzing the internal data files on OSTs. Comparing to the "hang" on v2.8, this is indeed an improvement.

When "b-Inconsist" happens on MDS (the "MDS" row), it is surprising that LFSCK finishes normally without any warning ("normal"). LFSCK's internal logs also show normal as we will see in §4.5. However, by examining the internal files of Lustre, we find that there is a resource leak problem under this scenario, which we will dicuss in 4.3.

When "b-Inconsist" happens on OSS (the "OSS#2" row), running LFSCK may trigger the rebooting of storage nodes unexpectedly ("reboot"). Also, `WikiR` reports mismatched checksums ("corrupt"). This is because OSTs store the striped data, "b-Inconsist" on OSTs affects the internal data files, which is not detectable by LFSCK.

In summary, we observe several unexpected behavior of LFSCK (e.g., "Invalid", "hang", "reboot"). We also verify that v2.10 has made noticeable improvement on failure handling compared to v2.8. We analyze the behavior in more details through the logs generated by Lustre and LFSCK in §4.4 and §4.5.

4.3 The Resource Leak Problem

4.3.1 ***Problem Description.*** In this section, we focus on a special case observed in our experiments where a portion of the internal namespace as well as the storage space on OSTs becomes unusable by Lustre even after running LFSCK, which we refer to as the resource leak problem. We observed the problem on both Lustre v2.8 and v2.10.

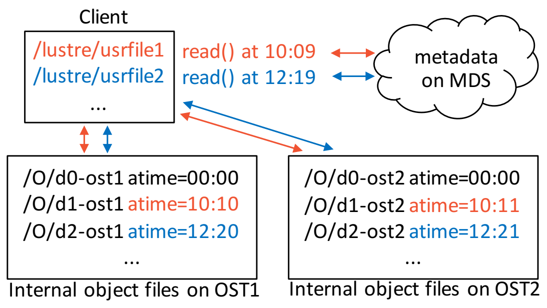

Figure 2: The Basic Idea of LeakCK. *Touching user files retrieves the mapping information from MDS and updates the access time (atime) of the corresponding internal object files; LeakCK identifies the unreachable internal object files based on the obsolete atime.*

# of OSTs	Stripe Size (KB)	Leaked Data (MB)	Detected ?
1	64	100	Yes
1	128	200	Yes
2	128	200	Yes
2	256	400	Yes
3	256	400	Yes
3	512	800	Yes

Table 3: Effectiveness of LeakCK. *LeakCK correctly detects the resource leak ("Yes") in six cases with different number of OSTs, stripe sizes, and amount of leaked data.*

The problem can be triggered by the following steps: (1) write one 300MB user file to Lustre synchronously (`FileOne300MB`); based on the striping rule, internally Lustre generates a 100MB object file on each of the three OSTs (`d1-ost1`, `d1-ost2`, `d1-ost3`); (2) corrupt the local file system of MDT; run e2fsck to fix local inconsistencies; (3) run LFSCK to check and fix global inconsistencies; (4) write another 150MB user file synchronously (`FileTwo150MB`), read it back and verify its checksum.

Essentially, the steps above emulate the *global inconsistency* fault model described in 3.3, and belong to the "b-Inconsist" case under `WikiW-sync` on MDS in Table 2, where LFSCK finishes normally ("normal") and `WikiW-sync` completes successfully ("✓").

After step (4), we observe that `FileTwo150MB` is written successfully based on the checksum, while `FileOne300MB` becomes invisible by the client. We examine the repaired local file system on MDT, and find that the metadata belonging to `FileOne300MB` has been cleaned by e2fsck for fixing the local corruption. In other words, the status of MDT after e2fsck is equivalent to the status without writing `FileOne300MB`. Since OSTs still hold the striped object files of `FileOne300MB`, there is a global inconsistency between MDT and OSTs from Lustre's perspective. Therefore, we expect LFSCK to be able to identify or fix such inconsistency.

However, by examining the internal files on the OSTs through the local file systems, we find that LFSCK did not detect or recycle `d1-ost1`, `d1-ost2`, `d1-ost3`, and Lustre cannot reuse the names and space belonging to these object files. Instead, new object files are created for `FileTwo150M` alongside `d1-ost1`, `d1-ost2`, `d1-ost3`.

We keep writing user files to Lustre until it reports "no space left". Then, we examine the internal object files of Lustre through the local file systems again, and find that `d1-ost1`, `d1-ost2`, `d1-ost3` still exist. Since neither `FileOne300MB` nor `d1-ost1`, `d1-ost2`, `d1-ost3` can be retrieved through Lustre or LFSCK, the internal names as well as the storage space occupied by these object files become the leaked resource.

In summary, we observe that the usable internal namespace and storage space of Lustre may shrink after faults, and LFSCK is unable to detect or fix the problem.

4.3.2 ***A Proposed Patch: LeakCK****.* To address the resource leak problem, we design and implement a simple tool called LeakCK, which is based on the observation that all "live" internal object files should be reachable from client.

Figure 2 shows the basic idea. LeakCK first reads a user file ("/lustre/myfile1") and records the timestamp of the read (e.g., "10:09"). Internally, the read operation retrieves the file mapping information from MDS. Based on the mapping, the corresponding internal object files on OSTs ("/O/d1-ost1" and "/O/d1-ost2") are accessed. Consequently, the access time ("atime") of the object files are updated (e.g., "10:10" and "10:11"). The LeakCK subcomponents running on OSSes collect and record the access time through the local file systems. Based on the access time, LeakCK identifies the object files belonging to the client-visible user file. In this way, LeakCK detects all reachable object files, reports those unreachable as the potential leak, and optionally move them to a "lost+found" directory.

One requirement of LeakCK is the synchronization of time across nodes, which can be easily achieved by running a Network Time Protocol (NTP) daemon (e.g., `ntpd` [31]). To tolerate the potential latency, LeakCK always waits for a time period after checking the internal object files corresponding to one user file. In this way, the access time of object files belonging to different user files can be separated accurately.

4.3.3 ***Effectiveness****.* To evaluate the effectiveness of LeakCK, we create six resource leak scenarios using the four steps described in §4.3.1. As shown in Table 3, the six cases vary in terms of the number of OSTs involved, the stripe size, and the total amount of leaked data. The stripe size and the number of OSTs for striping are set via the `lfs setstripe` command. The leaked data are generated using the `WikiW-sync` workload (Table 1) followed by the "b-Inconsist" fault on MDS. In all cases, LeakCK correctly reports the locations and sizes of the leaked internal data files.

4.3.4 ***Limitations****.* Besides the striped internal object files on OSTs, there may be other internal metadata files (e.g., object index files) associated with obsolete user file, which may become leaked as well. The current prototype of LeakCK only detects leaked internal object files on OSTs, which are usually much larger than the metadata files.

To detect all leaked resources, a detector would need the full knowledge of the internal metadata structures and layout of Lustre, which is out of the scope of this work. However, LeakCK's approach may be applicable to other PFSes as it treats the PFS as a black box to a great extent.

4.4 Logs of Lustre

Internally, Lustre maintains a debug buffer and generates extensive logs on various events, which provides a valuable way to analyze the system behavior. After emulating the faults, we collect the messages generated in the debug buffer of Lustre and summarize them in Table 4.

As shown in Table 4, we observe eight types of error messages (i.e., *y1 to y8*) when faults are injected on different nodes, including *Disconnection* (*y1*), *Recovery failed* (*y2-y4*), *Log updating failed* (*y5*), *Lock service failed* (*y6*), and *Failing over* (*y7*,*y8*).

It is interesting to see that MGS only reports one type of messages(i.e., only *y1* in the "Logs on MGS" column), while MDS and OSS nodes generate many types of messages. This is consistent with Lustre's design that MGS/MGT is mostly used for configuration when building Lustre, instead of the core functionalities.

All three fault models can trigger extensive log messages. For example, whenever "a-DevFail" happens on MDS (the "MDS" row), all nodes can notice the disconnection (i.e., *y1*). Moreover, all OSSes try to recover MDT but eventually fail (i.e., *y3*). This is because the OST handler on each OSS node keeps monitoring the connection with MDT (via `mdt_health_check`), and automatically tries to reconnect until timeout.

Also, "b-Inconsist" may generate various types of logs, depending on different inconsistencies caused by different local corruptions. When "b-Inconsist" happens on MDS (the "MDS" row), many services such as logging (i.e., *y5*) and locking (i.e., *y6*) may be affected. This is consistent with Lustre's design that MDS/MDT is critical for all regular operations.

Besides, when "a-DevFail" or "b-Inconsist" happens on MDS or OSS, it may trigger the failover of the affected node (i.e., *y7*, *y8*). Because a complete failover configuration on Lustre requires additional sophisticated software and hardware support [24], we cannot evaluate the effectiveness of the failover feature further using our current platform, and we leave it as future work.

However, we notice a potential mismatch between the documentation and the failover logs observed. Based on the documentation [24], the failover functionality of Lustre is designed for MDS/OSS sever processes instead of MDT/OST devices. For example, two MDS nodes configured as a failover pair must share the *same* MDT device, and when one MDS sever fails the remaining MDS can begin serving the unserved MDT. Because "a-DevFail" affects only the device (i.e., it emulates a whole device failure as discussed in §3.3), and does not kill the MDS/OSS sever processes, it is unclear how failing over server processes could handle the device failure.

In summary, we find the messages in the debug buffer (if reported) to be detailed and informative. As shown in the "Message Example" of Table 4, the messages usually include specific file names, line numbers, and function calls involved, which are valuable for understanding and diagnosing the system behavior. On the other hand, some log messages may not directly reflect the root cause of failures, which may imply that a more precise mechanism for detecting faults is needed.

Node(s) Affected	Fault Models	Logs on MGS	Logs on MDS	Logs on OSS#1	Logs on OSS#2	Logs on OSS#3
	a-DevFail	*y1*	*y2*	*y2*	*y2*	*y2*
MGS	b-Inconsist	*y1*	*y2,y5*	*y2,y5*	*y2,y5*	*y2,y5*
	c-Network	–	*y2*	*y2*	*y2*	*y2*
	a-DevFail	*y1*	*y1,y7*	*y1,y3*	*y1,y3*	*y1,y3*
MDS	b-Inconsist	*y1*	*y1,y5,y6,y7*	*y1,y3,y5,y6*	*y1,y3,y5,y6*	*y1,y3,y5,y6*
	c-Network	–	*y2,y4*	*y3*	*y3*	*y3*
	a-DevFail	*y1*	*y1,y4*	–	*y1,y8*	–
OSS#2	b-Inconsist	*y1*	*y1,y4,y5,y6*	*y5*	*y1,y5,y8*	*y5*
	c-Network	–	*y4*	–	*y2,y3*	–
three	a-DevFail	*y1*	*y1,y4*	*y1,y8*	*y1,y8*	*y1,y8*
OSSes	b-Inconsist	*y1*	*y1,y4,y5,y6*	*y1,y5,y8*	*y1,y3,y5*	*y1,y5,y8*
	c-Network	–	*y4*	*y2,y3*	*y2,y3*	*y2,y3*
MDS	a-DevFail	*y1*	*y1,y7*	*y1,y3*	*y1,y8*	*y1,y3*
+	b-Inconsist	*y1*	*y1,y5,y6,y7*	*y1,y3,y5,y6*	*y1,y5,y6,y8*	*y1,y3,y5,y6*
OSS#2	c-Network	–	*y2,y4*	*y3*	*y2,y3*	*y3*

Type	Meaning	Message Example
y1	Disconnection	*...genops.c:1244:class_disconnect() disconnect: cookie 0x923a4db81e68...*
y2	MGS Recovery failed	*...ptlrpc_connect_interpret() recovery of MGS on MGC 192.x.x.x...failed...*
y3	MDS Recovery failed	*...ptlrpc_connect_interpret() recovery of lustre-MDT0000_UUID...failed...*
y4	OSS Recovery failed	*...ptlrpc_connect_interpret() recovery of lustre-OST0001_UUID...failed...*
y5	Log updating failed	*...updating log 2 succeed 1 fail [...lustre − sptlrpc(fail)...*
y6	Lock service failed	*...ldlm_request.c:1317: ldlm_cli_update_pool()...@Zero SLV or Limit found...*
y7	Failing over MDT	*...obd_config.c:652:class_cleanup() Failing over lustre-MDT0000...*
y8	Failing over OST	*...obd_config.c:652:class_cleanup() Failing over lustre-OST0001...*

Table 4: Logs Generated in the Debug Buffer of Lustre After Faults. *The"Node(s) Affected" column shows the node(s) to which the faults are injected. "–" means no error message is reported. "y1" to "y8" are eight types of messages reported in the logs. The meaning of each type is shown at the bottom part of the table. The "Message Example" column shows a snippet of each type of messages adapted from the logs.*

4.5 Logs of LFSCK

Besides Lustre's internal logs, LFSCK also generates extensive status logs in the `/proc` pseudo file system on the MDS and OSS nodes [21]. We analyze these logs in this section.

There are three types of LFSCK logs, each of which corresponds to one major component of LFSCK: (1) **oi_scrub log (oi)**: linearly scanning all objects on the local device and verifying object indexes; (2) **layout log (lo)**: checking the regular striped file layout and verifying the consistency between MDT and OSTs; (3) **namespace log (ns)**: checking the local/global namespace consistency inside/among MDT(s). In addition, the debug buffer of Lustre may also record the activities of LFSCK (via `lctl set_param printk= +lfsck`), which we refer to as **debug buffer log**. On the MDS node, all types of logs are available. On OSS nodes, the namespace log is not available as it is irrelevant to OSTs. None of the LFSCK logs are generated on MGS.

Table 5 summarizes the logs (i.e., "oi", "lo", "ns") generated on different nodes after running LFSCK. We find that the debug buffer log is always consistent with the other logs, so we omit it here.

As shown in the table, when "b-Inconsist" happens on MDS, LFSCK may report that three orphans have been repaired (i.e., "repaired") in the "lo" log. This is because the corruption and repair of the local file system on MDS may lead to inconsistency between the MDS and the three OSSes. Based on the log, LFSCK is able to identify and repair the orphan objects on OSSes which do not have corresponding parents (on MDS) correctly.

When "a-DevFail" happens on MDS or OSS node(s), all LFSCK logs on the affected node(s) disappear from the `/proc` file system, and thus are unavailable (i.e., "–").

When LFSCK hangs (i.e., "hang" in Table 2), the logs may keep showing that it is in scanning. Internally, LFSCK uses a two-phase scanning to check and repair inconsistencies thoroughly [21], and the "lo" and "ns" logs may further show the scanning phases (i.e., "scan-1" and "scan-2"). In these scanning cases, we kill LFSCK after waiting for one hour without observing any progress.

In most cases (other than the two "repaired" cases), the logs are simply about LFSCK's progress (e.g., "init", "scan-1", "scan", "comp"). The corresponding debug buffer log (not shown) is relatively more informative. For example, it may show "layout lfsck slave queries master" on the OSS node, which describes the internal operations of LFSCK. Nevertheless, we find that these logs are still mostly about its execution status, instead of the potential issues of Lustre.

In summary, we find that LFSCK's logs is less informative for diagnosing problems comparing to Lustre's internal logs (§4.4). To guarantee that we do not miss any valuable error reports, we run LFSCK before injecting the faults to generate a set of logs under the normal condition. Then, we compare the logs of the two runs of LFSCK (i.e., with and without faults), and examine the difference.

Node(s) Affected	Fault Models	Logs on MDS			Logs on OSS#1		Logs on OSS#2		Logs on OSS#3	
		oi	lo	ns	oi	lo	oi	lo	oi	lo
MGS	a-DevFail	comp	comp	comp	comp	comp	comp	comp	comp	comp
	b-Inconsist	comp	comp	comp	comp	comp	comp	comp	comp	comp
	c-Network	comp	comp	comp	comp	comp	comp	comp	comp	comp
MDS	a-DevFail	–	–	–	init	init	init	init	init	init
	b-Inconsist	comp	**repaired**	comp	comp	comp	comp	comp	comp	comp
	c-Network	init	init	init	init	init	init	init	init	init
OSS#2	a-DevFail	scan	scan-1	init	comp	scan-2	–	–	comp	scan-2
	b-Inconsist	comp	scan-1	scan-1	comp	scan-2	comp	comp	comp	scan-2
	c-Network	scan	scan-1	init	comp	scan-2	init	init	comp	scan-2
three OSSes	a-DevFail	scan	scan-1	init	–	–	–	–	–	–
	b-Inconsist	comp	scan-1	scan-1	comp	comp	comp	comp	comp	comp
	c-Network	scan	scan-1	init	init	init	init	init	init	init
MDS + OSS#2	a-DevFail	–	–	–	init	init	–	–	init	init
	b-Inconsist	comp	**repaired**	scan-1	comp	scan-2	comp	comp	comp	scan-2
	c-Network	init	init	init	init	init	init	init	init	init

Table 5: Logs of LFSCK on Different Nodes. *The first column shows where the faults are injected. "oi", "lo", "ns" represent "oi_scrub log", "layout log", "namespace log", respectively. "comp" means the log shows LFSCK "completed"; "init" means the log shows the "init" state (no execution of LFSCK); "repaired" means the log shows "repaired three orphans"; "scan" means the log keeps showing "scanning" without making visible progress for an hour; "scan-1" means "scanning phase 1"; "scan-2" means "scanning phase 2"; "–" means the log is not available. We omit the debug buffer log of LFSCK as it is always consistent with the "oi", "lo", "ns" logs.*

In many cases there are no differences, except for minor updates such as the counts of execution and the running time of LFSCK.

5 RELATED WORK

Reliability Analysis of Storage Systems. Existing methods for testing the reliability of storage systems mainly include model checking [20, 52], formal methods [6], and automatic testing [12, 13, 27, 34, 53]. While these techniques are effective for testing local storage systems, applying them to large-scale PFSes remain challenging. For example, model checking still faces the state explosion problem despite of various path reduction optimizations [20]. Also, turning a practical system like Lustre into a controllable model is prohibitively difficult if possible. Similarly, formally verifying the behavior of a large-scale PFS like Lustre is almost infeasible in practice due to the complexity. 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, or only work for single-node systems. Since PFSes usually span many nodes and have special requirements or dependency on local systems, it is challenging to apply these existing frameworks to PFSes. An early version of PFault uses iSCSI to decouple the PFS from the analysis framework, but it can only collect I/O commands without injecting faults for reliability analysis [4].

One recent study [11] analyzes eight popular distributed storage systems and finds that none of them can consistently use redundancy to recovery from file-system faults. They build a fault injection framework called Cords, which consists of a FUSE file system and a set of workloads and behavior-inference scripts. Since Lustre has special requirements on local file systems (i.e., requiring a patched Ext4 or ZFS), FUSE-based Cords cannot be directly applied to study Lustre. Also, Cords only inject two types of corruptions (i.e., zeros or junk) to a single file-system block, while PFault uses three fault models to cover a wide range of real-world failure scenarios.

Besides, many other studies have examined the bugs or failure behaviors of local storage software and/or hardware (e.g., hard disks [2], RAID [25], flash-based SSDs [40, 54, 55], local file systems [22]). Generally, these studies provide valuable insights on the reliability of local storage systems, and they may help in emulating realistic failure states of storage nodes in PFault.

Performance Analysis of Parallel File Systems. Due to the prime importance of PFSes, many analysis tools have been proposed by the HPC community to improve them. For example, there is a variety of tools for instrumentation, profiling, and tracing of IO activities, such as mpiP [48], LANL-Trace [17], HPCT-IO [41], IOT [37], and TRACE [26] and so on. While these tools are mostly designed for improving the performance of PFSes, they may also help in reliability. For example, Darshan [5, 9] is able to capture the I/O characteristics of various HPC applications, including access patterns, frequencies, and duration time. Since all I/O requests are served by the backend PFS, these captured I/O characterization may be used by PFault to further reason the behavior of the PFS and identify the potential root causes of abnormalities observed.

Generally, these performance analysis tools are complimentary to PFault which focuses on emulating different fault models and analyzing the failure handling of PFSes.

6 CONCLUSIONS AND FUTURE WORK

High-performance PFSes are scaling to more and more nodes and are responsible for managing larger and larger volumes of scientific data. As the scale and complexity keeps increasing, maintaining consistency and data integrity under faults becomes more and more challenging.

To address the challenge. we have introduced PFault, a general framework for analyzing the failure handling of PFSes. PFault automatically captures I/O commands on all storage nodes of a target PFS, emulates realistic failure states based on well-defined fault models, and enables examining the recoverability of the PFS systematically. Moreover, we have applied PFault to study the widely used Lustre file system. Our analysis reveals a number of unexpected behavior of the recovery utility of Lustre (i.e., LFSCK), including a resource leak problem.

PFault is designed to be deployable in practice. For instance, the PFS Worker and Checker can be run through compute nodes, and the target PFS (and other HPC software stack) can be hooked to the Failure State Emulator (via iSCSI) without any modification. It is expected that our study and tools (i.e., PFault and LeakCK) will help improve Lustre as well as other PFSes for reliable high-performance computing.

This research study is a critical step on our roadmap toward achieving robust high-performance computing. In the future, we would like to explore the automation of diagnosing and understanding the root causes of the failure symptoms during the recovery. Also, we would like to explore additional fault models to capture more realistic failure scenarios.

Given the prime importance of PFSes in HPC systems and data centers, this reseach also calls for community's collective efforts in examining reliability challenges and coming up with advanced and highly-efficient solutions. We hope this work can inspire more research efforts along this direction. We also believe that such a study, associated methodologies, and insights drawn from our observations, can have a long-term impact on the design of large-scale file systems, storage systems, and HPC systems.

7 ACKNOWLEDGMENTS

We are thankful to the anonymous reviewers for their valuable feedback. This research is supported in part by the National Science Foundation under grants CNS-1338078, IIP-1362134, CCF-1409946, CNS-1566554, CCF-1717630, and CCF-1718336.

REFERENCES

[1] 2018. Prototype of PFAULT and LeakCK . https://www.cs.nmsu.edu/~mzheng/lab/lab.html. (2018).
[2] Lakshmi N. Bairavasundaram, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Garth R. Goodson, and Bianca Schroeder. 2008. An Analysis of Data Corruption in the Storage Stack. *Trans. Storage* 4, 3, Article 8 (Nov. 2008), 28 pages. https://doi.org/10.1145/1416944.1416947
[3] Lakshmi N. Bairavasundaram, Garth R. Goodson, Shankar Pasupathy, and Jiri Schindler. 2007. An Analysis of Latent Sector Errors in Disk Drives. In *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '07)*. ACM, New York, NY, USA, 289–300. https://doi.org/10.1145/1254882.1254917
[4] Jinrui Cao, Simeng Wang, Dong Dai, Mai Zheng, and Yong Chen. 2016. A Generic Framework for Testing Parallel File Systems. In *Proceedings of the 1st Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems (PDSW-DISCS '16)*. IEEE Press, Piscataway, NJ, USA, 49–54. https://doi.org/10.1109/PDSW-DISCS.2016.12
[5] Philip Carns, Robert Latham, Robert Ross, Kamil Iskra, Samuel Lang, and Katherine Riley. 2009. 24/7 Characterization of Petascale I/O Workloads. In *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*. IEEE, 1–10.
[6] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2015. Using Crash Hoare Logic for Certifying the FSCQ File System. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. ACM, New York, NY, USA, 18–37. https://doi.org/10.1145/2815400.2815402
[7] CloudLab. 2017. http://cloudlab.us/
[8] Alex Conway, Ainesh Bakshi, Yizheng Jiao, William Jannen, Yang Zhan, Jun Yuan, Michael A. Bender, Rob Johnson, Bradley C. Kuszmaul, Donald E. Porter, and Martin Farach-Colton. 2017. File Systems Fated for Senescence? Nonsense, Says Science!. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*. USENIX Association, Santa Clara, CA, 45–58. https://www.usenix.org/conference/fast17/technical-sessions/presentation/conway
[9] Darshan:HPC I/O Characterization Tool. 2017. http://www.mcs.anl.gov/research/projects/darshan/.
[10] debugfs. 2017. http://man7.org/linux/man-pages/man8/debugfs.8.html.
[11] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2017. Redundancy Does Not Imply Fault Tolerance: Analysis of Distributed Storage Reactions to Single Errors and Corruptions.. In *FAST*. 149–166.
[12] Om Rameshwar Gatla, Muhammad Hameed, Mai Zheng, Viacheslav Dubeyko, Adam Manzanares, Filip Blagojević, Cyril Guyot, and Robert Mateescu. 2018. Towards Robust File System Checkers. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*. USENIX Association, Oakland, CA, 105–122. https://www.usenix.org/conference/fast18/presentation/gatla
[13] Om Rameshwar Gatla and Mai Zheng. 2017. Understanding the Fault Resilience of File System Checkers. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*. USENIX Association, Santa Clara, CA. https://www.usenix.org/conference/hotstorage17/program/presentation/gatla
[14] Haryadi S. Gunawi, Riza O. Suminto, Russell Sears, Casey Golliher, Swaminathan Sundararaman, Xing Lin, Tim Emami, Weiguang Sheng, Nematollah Bidokhti, Caitie McCaffrey, Gary Grider, Parks M. Fields, Kevin Harms, Robert B. Ross, Andree Jacobson, Robert Ricci, Kirk Webb, Peter Alvaro, H. Birali Runesha, Mingzhe Hao, and Huaicheng Li. 2018. Fail-Slow at Scale: Evidence of Hardware Performance Faults in Large Production Systems. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*. USENIX Association, Oakland, CA, 1–14. https://www.usenix.org/conference/fast18/presentation/gunawi
[15] HDFS. 2017. https://hadoop.apache.org.
[16] High Performance Computing Center, Texas Tech University. 2017. http://www.depts.ttu.edu/hpcc/
[17] HPC-5 Open Source Software project, LANL-Trace. 2015. institutes.lanl.gov/data/tdata/.
[18] HPC User Site Census. 2016. http://www.intersect360.com/.
[19] HPCC Power Outage Event Announced at 8:50:17 AM Central Standard Time on Monday January 11 2016 (URL omitted for double-blind reviewing). 2016.
[20] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F. Lukman, and Haryadi S. Gunawi. 2014. 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)*. USENIX Association, Broomfield, CO, 399–414. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/leesatapornwongsa
[21] LFSCK: an online file system checker for Lustre. 2017. https://github.com/Xyratex/lustre-stable/blob/master/Documentation/lfsck.txt
[22] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. 2013. A Study of Linux File System Evolution. In *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*. USENIX, San Jose, CA, 31–44. https://www.usenix.org/conference/fast13/technical-sessions/presentation/lu
[23] Lustre File System. 2017. http://lustre.org/
[24] Lustre Software Release 2.x: Operations Manual. 2017. http://lustre.org/documentation/
[25] Ao Ma, Fred Douglis, Guanlin Lu, Darren Sawyer, Surendar Chandra, and Windsor Hsu. 2015. RAIDShield: Characterizing, Monitoring, and Proactively Protecting Against Disk Failures. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*. USENIX Association, Santa Clara, CA, 241–256. https://www.usenix.org/conference/fast15/technical-sessions/presentation/ma
[26] Michael P Mesnier, Matthew Wachs, Raja R Simbasivan, Julio Lopez, James Hendricks, Gregory R Ganger, and David R O'hallaron. 2007. //Trace: Parallel Trace Replay with Approximate Causal Events. USENIX.
[27] Changwoo Min, Sanidhya Kashyap, Byoungyoung Lee, Chengyu Song, and Taesoo Kim. 2015. Cross-checking Semantic Correctness: The Case of Finding File System Bugs. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. ACM, New York, NY, USA, 361–377. https://doi.org/10.1145/2815400.2815422
[28] Montage: An Astronomical Image Mosaic Engine. 2017. http://montage.ipac.caltech.edu/. (2017).
[29] Network Partition. 2017. https://www.cs.cornell.edu/courses/cs614/2003sp/papers/DGS85.pdf
[30] Edmund B. Nightingale, John R. Douceur, and Vince Orgovan. 2011. Cycles, Cells and Platters: An Empirical Analysis of Hardware Failures on a Million Consumer PCs. In *Proceedings of the Sixth Conference on Computer Systems (EuroSys '11)*.

ACM, New York, NY, USA, 343–356.
[31] NTPD. 2017. http://doc.ntp.org/4.1.0/ntpd.htm
[32] NVM Express over Fabrics Specification Released. 2017. http://www.nvmexpress.org/nvm-express-over-fabrics-specification-released/
[33] David A Patterson, Garth Gibson, and Randy H Katz. 1988. *A case for redundant arrays of inexpensive disks (RAID).* Vol. 17. ACM.
[34] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2014. All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14).*
[35] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2005. IRON File Systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP'05).* Brighton, United Kingdom, 206–220.
[36] Meryem Primmer. 1996. An Introduction to Fibre Channel. *HP Journal* (1996).
[37] Philip C Roth. 2007. Characterizing the I/O behavior of scientific applications on the Cray XT. In *Proceedings of the 2nd international workshop on Petascale data storage: held in conjunction with Supercomputing'07.* ACM, 50–55.
[38] R. Sandberg, D. Golgberg, S. Kleiman, D. Walsh, and B. Lyon. 1988. Innovations in Internetworking. Artech House, Inc., Norwood, MA, USA, Chapter Design and Implementation of the Sun Network Filesystem, 379–390. http://dl.acm.org/citation.cfm?id=59309.59338
[39] Bianca Schroeder and Garth A. Gibson. 2007. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you?. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST'07).*
[40] Bianca Schroeder, Raghav Lagisetty, and Arif Merchant. 2016. Flash Reliability in Production: The Expected and the Unexpected. In *14th USENIX Conference on File and Storage Technologies (FAST 16).* USENIX Association, Santa Clara, CA, 67–80. http://usenix.org/conference/fast16/technical-sessions/presentation/schroeder
[41] Seetharami Seelam, I Chung, Ding-Yong Hong, Hui-Fang Wen, Hao Yu, et al. 2008. Early Experiences in Application Level I/O Tracing on Blue Gene Systems. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on.* IEEE, 1–8.
[42] Keith A Smith and Margo I Seltzer. 1997. File system agingâĂŤincreasing the relevance of file system benchmarks. In *ACM SIGMETRICS Performance Evaluation Review,* Vol. 25. ACM, 203–213.
[43] Sriram Subramanian, Yupu Zhang, Rajiv Vaidyanathan, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Jeffrey F. Naughton. 2010. Impact of disk corruption on open-source DBMS. In *ICDE.* 509–520.
[44] Ceph File System. 2017. http://docs.ceph.com/docs/master/
[45] Linux SCSI target framework (tgt). 2017. http://stgt.sourceforge.net/.
[46] The OrangeFS Project. 2017. http://www.orangefs.org/
[47] Top500 Machines. 2017. https://www.top500.org/lists/2016/11/
[48] Jeffrey S Vetter and Michael O McCracken. 2001. Statistical Scalability Analysis of Communication Operations in Distributed Applications. In *ACM SIGPLAN Notices,* Vol. 36. ACM, 123–132.
[49] Website of the experimental environment, logs, PFAULT and LeakCK (URL omitted for double-blind reviewing). 2017.
[50] ext4 wiki. 2017. https://ext4.wiki.kernel.org/.
[51] Wikipedia:Database download. 2017. https://en.wikipedia.org/wiki/Wikipedia:Database_download
[52] Junfeng Yang, Can Sar, and Dawson Engler. 2006. 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).* 131–146.
[53] Mai Zheng, Joseph Tucek, Dachuan Huang, Feng Qin, Mark Lillibridge, Elizabeth S. Yang, Bill W Zhao, and Shashank Singh. 2014. Torturing Databases for Fun and Profit. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14).* USENIX Association, Broomfield, CO, 449–464. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/zheng_mai
[54] Mai Zheng, Joseph Tucek, Feng Qin, and Mark Lillibridge. 2013. Understanding the Robustness of SSDs under Power Fault. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST'13).*
[55] Mai Zheng, Joseph Tucek, Feng Qin, Mark Lillibridge, Bill W Zhao, and Elizabeth S. Yang. 2016. Reliability Analysis of SSDs under Power Fault. In *To appear in the ACM Transactions on Computer Systems (TOCS).* http://dx.doi.org/10.1145/2992782