A Study of Failure Recovery and Logging of High-Performance Parallel File Systems

RUNZHOU HAN, OM RAMESHWAR GATLA, MAI ZHENG, Iowa State University
JINRUI CAO, State University of New York at Plattsburgh
DI ZHANG, DONG DAI, North Carolina University at Charlotte
YONG CHEN, Texas Tech University
JONATHAN COOK, New Mexico State University

Large-scale parallel file systems (PFSes) play an essential role in high performance computing (HPC). However, despite the importance, their reliability is much less studied or understood compared with that of local storage systems or cloud storage systems. Recent failure incidents at real HPC centers have exposed the latent defects in PFS clusters as well as the urgent need for a systematic analysis.

To address the challenge, we perform a study of the failure recovery and logging mechanisms of PFSes in this paper. First, to trigger the failure recovery and logging operations of the target PFS, we introduce a black-box fault injection tool called PFault, which is transparent to PFSes and easy to deploy in practice. PFault emulates the failure state of individual storage nodes in the PFS based on a set of pre-defined fault models, and enables examining the PFS behavior under fault systematically.

Next, we apply PFault to study two widely used PFSes: Lustre and BeeGFS. Our analysis reveals the unique failure recovery and logging patterns of the target PFSes, and identifies multiple cases where the PFSes are imperfect in terms of failure handling. For example, Lustre includes a recovery component called LFSCK to detect and fix PFS-level inconsistencies, but we find that LFSCK itself may hang or trigger kernel panics when scanning a corrupted Lustre. Even after the recovery attempt of LFSCK, the subsequent workloads applied to Lustre may still behave abnormally (e.g., hang or report I/O errors). Similar issues have also been observed in BeeGFS and its recovery component BeeGFS-FSCK. We analyze the root causes of the abnormal symptoms observed in depth, which has led to a new patch set to be merged into the coming Lustre release. In addition, we characterize the extensive logs generated in the experiments in details, and identify the unique patterns and limitations of PFSes in terms of failure logging. We hope this study and the resulting tool and dataset can facilitate follow-up research in the communities and help improve PFSes for reliable high-performance computing.

CCS Concepts: • **Computer systems organization** → **Reliability**; **Secondary storage organization**.

Additional Key Words and Phrases: Parallel File Systems, File System Checkers, Reliability, Failure Handling, Logging, High Performance Computing, Storage Systems

Authors' addresses: Runzhou Han, Om Rameshwar Gatla, Mai Zheng, {hanrz,ogatla,mai}@iastate.edu, Iowa State University, 613 Morrill Rd, Ames, Iowa, 50011; Jinrui Cao, will_cao@nmsu.edu, State University of New York at Plattsburgh, 101 Broad St, Plattsburgh, New York, 12901; Di Zhang, Dong Dai, {dzhang16,dong.dai}@uncc.edu, North Carolina University at Charlotte, 9201 University City Blvd, Charlotte, North Carolina, 28223; Yong Chen, {yong.chen}@ttu.edu, Texas Tech University, 2500 Broadway, Lubbock, Texas, 79409; Jonathan Cook, jcook@cs.nmsu.edu, New Mexico State University, 1780 E University Ave, Las Cruces, New Mexico, 88003.

1553-3077/2022/1-ART1 $15.00
https://doi.org/10.1145/3483447

ACM Reference Format:
Runzhou Han, Om Rameshwar Gatla, Mai Zheng, Jinrui Cao, Di Zhang, Dong Dai, Yong Chen, and Jonathan Cook. 2022. A Study of Failure Recovery and Logging of High-Performance Parallel File Systems. *ACM Trans. Storage* 1, 1, Article 1 (January 2022), 44 pages. https://doi.org/10.1145/3483447

1 INTRODUCTION

Large-scale parallel file systems (PFSes) play an essential role today. A variety of PFSes (e.g., Lustre [1], BeeGFS [2], OrangeFS [3]) have been deployed in high performance computing (HPC) centers around the world to empower large-scale I/O intensive computations. Therefore, the reliability of PFSes is critically important.

However, despite the prime importance, the reliability of PFSes is much less studied or understood compared with that of other storage systems. For example, researchers [4–12] have studied and uncovered reliability issues in different layers of local storage systems (e.g., RAID [5], local file systems [6, 7]) as well as in many distributed cloud systems (e.g., HDFS [13], Cassandra [14], ZooKeeper [15]). However, to the best of our knowledge, there is little equivalent study on PFSes. This raises the concern for PFSes which are built atop of local storage systems and are responsible for managing large datasets at a scale comparable to cloud systems.

In fact, in a recent failure incident at an HPC center (HPCC) in Texas [16], multiple storage clusters managed by the Lustre parallel file system [1] suffered severe data loss after power outages [17]. Although many files have been recovered after months of manual efforts, there are still critical data lost permanently, and the potential damage to the scientific discovery is unmeasurable. Similar events have been reported at other HPC centers [18–20]. Such failure incidents suggest the potential defects in the failure handling of production PFSes as well as the urgent need for a systematic study.

Motivated by the real problem, we perform a study of the failure handling mechanisms of PFSes in this paper. We focus on two perspectives: (1) the recovery of PFS, which is important for ensuring data integrity in PFS under failure events; (2) the logging of PFS, which is important for diagnosing the root causes of PFS anomalies after recovery (e.g., I/O errors or data loss).

The first challenge is how to trigger the failure recovery and logging operations of PFSes in a systematic way. While many methods and tools have been proposed for studying distributed cloud systems [8–12, 21–23], we find that none of them is directly applicable to PFSes, largely due to the unique architecture and complexity of PFSes. Major PFSes are designed to be POSIX-compliant to support abundant HPC workloads and middleware (e.g., MPI-IO [24]) transparently with high performance. To this end, they typically include operating system (OS) kernel modules and hook with the virtual file system (VFS) layer of the OS. For example, Lustre [1] requires installing customized Linux kernel modules on all storage nodes to function properly, and the local file system Ext4 must be patched for Lustre's `ldiskfs` backend [25]. Such close interleaving and strong dependency on the OS kernel makes existing methodologies designed for user-level distributed systems (e.g., HDFS) difficult to use for studying PFSes in practice. For instance, CORDS [21] inject faults to cloud systems via a customized FUSE file system, which is incompatible to major PFSes.

Also, different from many cloud systems [14, 26, 27], PFSes do not maintain redundant copies of data at the PFS level, nor do they use well-understood, consensus-based protocols [23] for recovery. As a result, existing methodologies that rely on the specifications of well-known fault-tolerance protocols (e.g., the gossip protocol [14]) are not applicable to PFSes. See §2, §6 and §7 for further discussion.

To address the challenge, we introduce a fault injection tool called PFault, which follows a black-box principle [28] to achieve high usability for studying PFSes in practice. PFault is based on two key observations: (1) External failure events may vary, but only the on-drive persistent states may affect the PFS recovery after rebooting; therefore, we may boil down the generation of various

external failure events to the emulation of the device state on each storage node. (2) Despite the complexity of PFSes, we can always separate the whole system into a global layer across multiple nodes, and a local system layer on each individual node; moreover, the target PFS (including its kernel components) can be transparently decoupled from the underlying hardware through remote storage protocols (e.g., iSCSI [29], NVMe/Fabric [30]), which have been used in large-scale storage clusters for easy management of storage devices. In other words, by emulating the failure states of individual storage nodes via remote storage protocols, we can minimize the intrusion or porting effort for studying PFSes.

Based on the idea above, we build a prototype of PFault based on iSCSI, which covers three representative fault models (i.e., *whole device failure*, *global inconsistency*, and *network partitioning*, as will be introduced in §3.2.2) to support studying the failure recovery and logging of PFSes systematically. Moreover, to address the potential concern of adding iSCSI to the PFS software stack, we develop a non-iSCSI version of PFault, which can be used to verify the potential impact of iSCSI on the behavior of the target PFS under study.

Next, we apply PFault to study two major production PFSes: Lustre [1] and BeeGFS [2]. We apply the three fault models to different types and subsets of nodes in the PFS cluster to create diverse failure scenarios, and then examine the corresponding recovery and logging operations of the target PFS meticulously. Our study reveals multiple cases where the PFSes are imperfect. For example, Lustre includes a recovery component called LFSCK [31] to detect and fix PFS-level inconsistencies, but we find that LFSCK itself may hang or trigger kernel panics when scanning a post-fault Lustre. Moreover, after running LFSCK, the subsequent workloads applied to Lustre may still behave abnormally (e.g., hang or report I/O errors). Similarly, the recovery component of BeeGFS (i.e., BeeGFS-FSCK) may also fail abruptly when trying to recover the post-fault BeeGFS.

In terms of logging, we find that both Lustre and BeeGFS may generate extensive logs during failure handling. However, different from modern cloud systems which often use common libraries (e.g., Log4J [32]) to generate well-formatted logs, the logging methods and patterns of PFSes are diverse and irregular. For example, Lustre may report seven types of standard Linux error messages (e.g., EIO, EBUSY, EROFS) across different types of storage nodes in the cluster, while BeeGFS may only log two types of standard messages on limited nodes under the same faults. On the other hand, BeeGFS may generate more customized error messages, some of which are equivalent to the standard Linux errors. By characterizing the PFS logs in details based on the log sources, content, fault types, and locations, we identify multiple cases where the log messages are inaccurate or misleading, which suggests new opportunities for log enhancement and log-based analysis.

More importantly, based on the substantial PFS logs, PFS source code, and the feedback from PFS developers, we are able to identify the root causes of a subset of the abnormal symptoms observed in the experiments (e.g., I/O error, reboot). The in-depth root cause analysis has clarified the resource leak problem observed in our preliminary experiments [33], and has led to a new patch set to be merged into the mainline Lustre release [34].

To the best of our knowledge, this work is the first comprehensive study on the failure recovery and logging mechanisms of production PFSes widely used in HPC centers. By developing a practical tool and applying it to systematically analyze multiple versions of representative PFSes in depth, we identify the common limitations as well as the opportunities for further improvements. We hope that this study, including the open-source PFault tool and the extensive collection of PFS failure logs [1], can raise the awareness of potential defects in PFSes, facilitate follow-up research

[1]The latest prototype of PFault and the experimental logs are publicly available at https://git.ece.iastate.edu/data-storage-lab/prototypes/pfault

in the communities, and help improve Lustre, BeeGFS, and HPC storage systems in general for reliable high-performance computing.

The rest of the article is organized as follows. In §2, we discuss the background and motivation; In §3, we introduce the PFault tool; In §4, we describe the study methodology based on PFault; In §5, we present the study results of Lustre and BeeGFS; In §6, we elaborate on the lessons learned and the opportunities for further improvements; §7 discusses related work and §8 concludes the paper. In addition, for interested readers, we characterize the extensive failure logs collected in our experiments in appendix §A.

2 BACKGROUND AND MOTIVATION

2.1 Parallel File Systems

Parallel file systems (PFSes) is a critical building block for high performance computing. They are designed and optimized for the HPC environment, which leads to an architecture different from other distributed storage systems (e.g., GoogleFS [26], HDFS [13]). For example, PFSes are optimized for highly concurrent accesses to the same file, and they heavily rely on hardware-level redundancy (e.g., RAID [35]) instead of distributed file system level replication [26] or erasure coding [27]. We use Lustre [25] and BeeGFS [2], two representative PFSes with different design tradeoffs, as examples to introduce the typical architecture of PFSes in this section.

2.1.1 ***Lustre and LFSCK.*** Lustre dominates the market share of HPC centers [36], and more than half of the top 100 supercomputers use Lustre [37]. A Lustre file system usually includes the following components:

- Management Server (**MGS**) and Management Target (**MGT**) manage and store 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**) manage and store the metadata of Lustre. MDS provides request handling for local MDTs. There can be multiple MDSs/MDTs since Lustre v2.4. Also, MGS/MGT can be co-located with MDS/MDT.
- Object Storage Server (**OSS**) and Object Storage Target (**OST**) manage and store 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** mount Lustre to their local directory and launch applications to access the data in Lustre; the applications are typically executed on login nodes or compute nodes which are separated from the storage nodes of Lustre.

Different from most cloud storage systems (e.g., HDFS [13], HBase [38], Cassandra [14]), the major functionalities of Lustre server components are closely integrated with the Linux kernel to achieve high performance. Moreover, Lustre's `ldiskfs` backend modifies Ext4 and heavily relies on the extended attributes of Ext4 for metadata. Such close interleaving with the OS kernel makes analyzing Lustre challenging.

Traditionally, high performance is the most desired metric of PFSes. However, as more and more critical data are generated by HPC applications, the system scale and complexity keeps increasing. Consequently, ensuring PFS consistency and maintaining data integrity under faults becomes more and more important and challenging. To address the challenge, Lustre introduces a special recovery component called **LFSCK** [31] for checking and repairing Lustre after faults, which has been significantly improved since v2.6. Similar to the regular operations of Lustre, LFSCK also involves substantial functionalities implemented at the OS kernel level.

Typically, a Lustre cluster may include one MGS node, one or two dedicated MDS node(s), and two or more OSS nodes, as shown in the target PFS example in Figure 1a (§3.1). And LFSCK may be invoked on demand to check and repair Lustre after faults. We follow such setting in this study. Note that LFSCK may also be invoked automatically by Lustre under certain conditions. For example, the `oi_scrub` procedure of LFSCK may be triggered to scan all objects on the device when an error is found during object index (OI) lookup operations; also, in case there is an error when accessing the `LAST_ID` file on OST, LFSCK will attempt to repair it based on the existing objects on the OST [25]. We explicitly invoke LFSCK with a tunable delay in this study to ensure that all the LFSCK procedures are executed.

2.1.2 ***BeeGFS and BeeGFS-FSCK***. BeeGFS is one of the leading PFSes that continues to grow and gain significant popularity in the HPC community [39]. Conceptually, a BeeGFS cluster is similar to Lustre in the sense that it mainly consists of a management server (MGS), at least one metadata server (MDS), a number of storage servers (OSS) and several client nodes. BeeGFS also includes kernel modules to achieve high performance and POSIX compliance for clients. In addition, a BeeGFS cluster may optionally include other utility nodes (e.g., an Admon server for monitoring with a graphic interface). For simplicity, we use the same acronym names (i.e., MGS, MDS, OSS) to describe equivalent storage nodes in Lustre and BeeGFS in this study.

Facing the same challenge as Lustre to guarantee PFS-level consistency and data integrity, BeeGFS also has a recovery component called **BeeGFS-FSCK**. Different from Lustre's LFSCK, BeeGFS-FSCK collects the PFS states from available servers in parallel, stores them into a user-level database (i.e., SQLite [40]), and issues SQL queries to check for potential errors in BeeGFS, which is similar to the principle of SQCK [41]. Similar to invoking LFSCK, we explicitly invoke BeeGFS-FSCK in this study to ensure that it is fully executed. For simplicity, we use **FSCK** to refer to both LFSCK and BeeGFS-FSCK in the rest of the paper.

2.2 Limitations of Existing Efforts

PFS Test Suites. Similar to other practical software systems, PFSes typically have built-in test suites. For example, Lustre has a testing framework called "auster" to drive more than two thousand active test cases [1]. Similarly, BeeGFS includes a rich set of default tests as well as third party tests [39]. However, most of the test suites are unit tests or micro-benchmarks for verifying the functionality or performance of the PFS during *normal execution*. There are limited cases designed for exercising error handling code paths, but they typically require modifying the PFS source code (e.g., enabling debugging macros, inserting function hooks), which is intrusive. Moreover, they aim to generate one specific function return error to emulate a single buggy function implementation within the source code, instead of emulating external failure events that the entire PFS cluster may have to handle (e.g., power outages or hardware failures). Therefore, existing PFS test suites are not enough for studying the failure recovery and logging mechanisms of PFSes.

Studies of Other Distributed Systems. Great efforts have been made to understand other distributed systems (e.g., [9, 10, 21–23, 42–48]), especially modern cloud systems (e.g., HDFS [13], Cassandra [14], Yarn [49], ZooKeeper [15]). Different from PFSes, most of the heavily-studied distributed systems [13–15, 49] are designed from scratch to handle failure events gracefully in the cloud environment where component failures are the norm rather than exception [26]. To this end, the cloud systems typically do not embed customized modules or patches in the OS kernel. Instead, they consist of loosely coupled user-level modules with well-specified protocols for fault tolerance (e.g., the leader election in ZooKeeper [15], the gossip protocol in Cassandra [14]). Such clean design features have enabled many grey-box/white-box tools [28] which leverage well-understood internal protocols or specifications to analyze the target systems effectively [9, 10, 22, 23, 48].

Unfortunately, while existing methods are excellent for their original goals, we find that none of them can be directly applied to PFSes in practice. We believe one important reason is that PFSes are traditionally designed and optimized for the HPC environment, where performance is critically important and component failures were expected to be minimal. Such fundamental assumption has led to completely different design tradeoffs throughout the decades, which makes existing cloud-oriented efforts sub-optimal or inapplicable for PFSes. More specifically, there are multiple reasons as follows.

First, as mentioned in §1, major PFSes are typically integrated with the OS kernel to achieve high performance and POSIX-compliance. The strong interleaving and dependency on the local storage stack cannot be handled by existing methods designed for user-level distributed systems without substantial engineering efforts (if possible at all).

Second, PFSes tend to integrate reliability features incrementally with regular functionalities without using well-known fault-tolerance protocols. For example, there is no plugable erasure coding modules (as in HDFS 3.X [50]) or explicit consensus-based protocols [23] involved at the PFS layer. Instead, PFSes heavily rely on local storage systems (e.g., patched local file systems and checkers [51]) to protect PFS metadata against corruption on individual nodes, and leverage the FSCK component to check and repair corruptions at the PFS level. Moreover, most of the functionalities of FSCK may be implemented in customized kernel modules together with regular functionalities [52]. Such monolithic and opaque nature makes existing tools that rely on well-understood distributed protocols or require detailed knowledge of internal specifications of the target system difficult to use for studying PFSes in practice [9, 10, 22].

Third, many cloud systems are Java-based and they leverage common libraries for logging (e.g., Log4J [32]). The strongly typed nature of Java and the unified logging format have enabled sophisticated static analysis on the source code and/or system logs for studying cloud systems [10]. However, PFSes are typically implemented in C/C++ with substantial low-level code in the OS kernel which is challenging for static analysis. Moreover, as we will detail in later sections (§5 and §A), PFSes tend to use diverse logging methods with irregular logging formats, which makes techniques depending on clean logs [10] largely inapplicable.

In summary, we find that existing methods are sub-optimal for studying the failure handling of PFSes due to one or more constraints: they may (1) only handle user-level programs (e.g., Java programs) instead of PFSes containing OS kernel modules and patches; (2) require modifications to the local storage stack (e.g., using FUSE [53]) which are incompatible to major PFSes; (3) rely on language-specific features/tools that are not applicable to major PFSes; (4) rely on common logging libraries (e.g., Log4J [32]) and well-formatted log messages that are not available on major PFSes; (5) rely on detailed specifications of internal protocols of target systems, which are not available for PFSes to the best of our knowledge. See §6 and §7 for further discussion.

2.3 Remote Storage Protocols

Remote storage protocols (e.g., NFS [54], iSCSI [29], Fibre Channel [55], NVMe/Fabric [30]) enable accessing remote storage devices as local devices, either at the file level or the block level. In particular, iSCSI[29] is an IP-based protocol allowing one machine (i.e., the iSCSI `initiator`) to access the remote block storage on another machine (i.e., the iSCSI `target`) through the internet. To everything above the block device driver layer (which is not patched by the PFS) on the initiator machine, iSCSI is completely transparent. In other words, file systems including PFSes can be built on iSCSI devices without any modification. We leverage this property to make PFault practical and efficient for studying real-world PFSes.

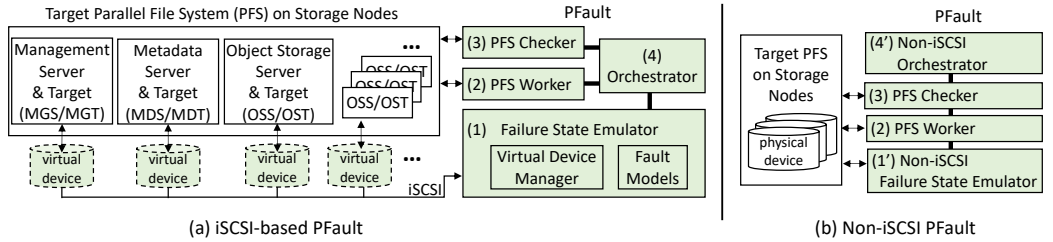

Fig. 1. **Overview of PFault.** *The shaded boxes are the major components of PFault. (a) the iSCSI version enables manipulating PFS images efficiently; (b) the non-iSCSI version enables verifying the potential impact of iSCSI on the target PFS.*

3 HOW TO TRIGGER PFS FAILURE HANDLING AND LOGGING OPERATIONS

In this section, we describe the design and implementation of the PFault tool which enables us to perform a systematic study. As mentioned in §1 and §2, the first challenge we encountered when we initiated the study is that none of existing tools, including the official PFS test suites and the extensive research prototypes (§2.2), can be applied to analyze the failure behaviors of production PFSes like Lustre without substantial modifications (if not impossible). To address the challenge, we design and implement PFault with the following three key goals, which we believe are critically important for studying the failure behaviors of PFSes in practice:

- **Usability**. Applying a tool to study PFSes can take a substantial amount of efforts due to the complexity of the PFS cluster; PFault aims to reduce the burden as much as possible. To this end, PFault makes a key tradeoff to view the target PFS as a black box [28]. It does not require any modification or instrumentation of the PFS code, nor does it require any specification of the recovery protocols of PFS (which is often not documented well).
- **Generality**. By leveraging the iSCSI driver which is transparent to most OS kernel modules, PFault can be applied to study different PFSes with little porting effort, no matter how strong the interleaving is between the distributed layer and the local kernel components of the PFS.
- **Fidelity**. PFault can emulate diverse external failure events (e.g., metadata corruptions, network partitioning) with high fidelity without changing the PFS itself (i.e., non-intrusive).

3.1 Overview

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

There are two versions of PFault: an iSCSI-based version (Figure 1a) and a non-iSCSI version (Figure 1b). The iSCSI version controls the persistent states of the PFS nodes via iSCSI and enables studying the failure recovery and logging mechanisms of the PFS efficiently, while the non-iSCSI version can be used to verify the potential impact of iSCSI on the target PFS.

As shown in Figure 1a, the iSCSI-based PFault includes four major components. (1) The *Failure State Emulator* is responsible for injecting faults to the target PFS. It mounts a set of virtual devices to the storage nodes via iSCSI and forwards all disk I/O commands to the backing files through the iSCSI protocol. Each backing file is one regular image file maintained on the PFault server and configured as the backend device for the iSCSI `target` (§2.3), which represents the persistent state of a corresponding virtual device. Moreover, the *Failure State Emulator* manipulates the backing files and emulates the failure state of each virtual device based on the workloads and a set of predefined fault models. (2) The *PFS Worker* launches workloads to exercise the PFS and generate

I/O operations. (3) The *PFS Checker* invokes the recovery component (i.e., FSCK) of the PFS as well as a set of verifiable workloads to examine the recoverability of the PFS. (4) The *Orchestrator* coordinates the overall workflow and collects the corresponding logs automatically. We discuss the details of the four components in §3.2, §3.3, §3.4, and §3.5, respectively.

Figure 1b shows the non-iSCSI version of PFault, which differs from the iSCSI version in the *Failure State Emulator* and the *Orchestrator* components. We discuss the key difference in §3.6 and summarize the overall workflow in §3.7.

3.2 Failure State Emulator

To study the failure recovery and logging of the PFS, it is necessary to generate faults in a systematic way. Thanks to the great efforts in understanding real-world storage system failures [4, 5, 56–62], we can model a set of representative scenarios at different granularity relatively easily. However, the real challenge is how to build a practical tool to inject the desired faults to the PFS cluster with high usability, generality, and fidelity (i.e., the three important goals described earlier in §3). While various fault injectors have been proposed in the communities [4, 10, 21, 23, 42–46], we find that they are not directly applicable to PFSes due to a number of practical constraints (e.g., cannot handle PFS's kernel modules, require detailed specifications, as explained in §2.2). Based on our key observations on the unique architecture of PFSes, we identify a low-level software layer (i.e., iSCSI) that enables us to implement automatic fault injection on different PFSes with diverse granularity (e.g., file-level metadata corruptions, device- and node-level crashes, and cluster-level network partitioning). More specifically, PFault reduces various failure events to the states of storage devices via *Failure State Emulator*, which mainly includes two sub-components: *Virtual Device Manager* and *Fault Models* (Figure 1a) as follows:

3.2.1 ***Virtual Device Manager (VDM).*** This sub-component manages the states of iSCSI virtual devices to enable efficient failure emulation. The persistent state of the target PFS depends on the I/O operations issued to the devices. To capture all I/O operations in the PFS, the VDM 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 [29]. Thanks to iSCSI, the virtual devices appear to be ordinary local block devices from the PFS perspective. In other words, PFault is transparent to the PFS (including its kernel components) under study.

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

Note that the virtual devices can be mounted to either physical machines or virtual machines (VMs) via iSCSI. In the VM case, the entire framework and the target PFS may be hosted on one single physical machine, which makes studying the PFS with PFault convenient. This design philosophy is similar to ScaleCheck [48] which leverages VMs to enable scalability testing of distributed systems on a single machine.

3.2.2 ***Fault Models.*** This sub-component defines the failure events to be emulated by PFault. For each storage node with a virtual device, PFault manipulates the corresponding backing file and the network daemon based on the pre-defined fault models. The current prototype of PFault includes three representative fault models 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 [4, 5, 56].

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 (§2.3) to disconnect the backing file to the corresponding storage node, which makes the device inaccessible to the PFS immediately. Also, different types of devices (i.e., MGT, MDT, OST) may be disconnected individually or simultaneously to emulate device failures at different scales. By leveraging the remote storage protocol, PFault can emulate different scenarios automatically without any manual effort.

(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 backend (e.g., Ext4-based `ldiskfs` for Lustre) 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 (patched) local file systems, PFSes typically rely on local file systems to maintain the local consistency. For example, the local file system checker (e.g., `e2fsck`[63] for `ldiskfs`) is required to be executed on every storage node before invoking PFS FSCK. In other words, it is perhaps unreasonable to expect PFS FSCK to be able to recover the PFS properly when a local file system is broken. Therefore, in this model we intentionally enforce that every local file system in the PFS cluster must be consistent locally. Note that this is different from existing works which emulate abnormal local file systems (e.g., return errors on local file system operations [21, 23]).

The global inconsistency scenarios may be caused by a variety of reasons. For example, in a data center-wide power outage [17], the local file systems on individual storage nodes may be corrupted to different degrees depending on the PFS I/O operations at the fault time. Similarly, besides power outages, local file systems may also be corrupted due to file system bugs, latent sector errors, etc [4, 56, 64]. The corruptions of the local file system need to be checked and repaired by the corresponding local file system checker. However, the local checker only has the knowledge of local metadata consistency rules (e.g., `ldiskfs` follows Ext4's rules), and it can only manipulates the local state on each node independently. While running the local checker may bring all local file systems back to a locally consistent state, it may (unintentionally) break the global consistency rules of PFS due to its local repair operations (e.g., skipping incomplete journal transactions, recycling a corrupted local `inode`, moving a local file to the "`lost+found`" directory). As a result, the global consistency across PFS nodes may be compromised.

To emulate the fault model effectively and efficiently, PFault uses two complementary approaches as follows:

(1) PFault invokes the debugging tool of the local file system (e.g., `debugfs` [65] for Ext4) to manipulate the local states on selected nodes. The debugging tool allows "trashing" specific metadata structures of the local file system for regression testing. We make use of such feature to randomly corrupt a given percentage of total on-disk files' `inode` fields. After introducing local corruptions, we invoke the checking and repairing utility of the local file system (e.g., `e2fsck` [51]) to repair local inconsistencies and thus bring the local file system back to a (locally) healthy state.

(2) PFault invokes Linux command line utilities (e.g., `rm`) to randomly delete a given percentage of on-disk files entirely on selected nodes. This is to emulate the repairing effect of the local file system where the local checker may move a corrupted local file to the "`lost+found`" directory, making it "missing" from the PFS' perspective. Since the delete operations are regular operations supported by the local file system, the local file system remains consistent. By deleting different local files (e.g., various object files, links) on different types of nodes (i.e., MGS, MDS, OSS), we can easily introduce a wide range of global inconsistencies while maintaining the local consistency.

The two approaches have their tradeoffs. Since the debugging tool can expose accurate type information of the metadata of the local file system, the first approach allows PFault to manipulate the local metadata structures directly and comprehensively. However, introducing corruptions to local metadata directly may cause severe damage beyond the repairing capability of the local file system utility (e.g., `e2fsck`). Consequently, the local image may be "too broken" to be used for further analyzing the global consistency of PFS, and the entire analysis workflow has to be stopped. Such interruption is one major limitation in our preliminary prototype [33] that makes the workflow inefficient. In contrast, the second approach always maintains a usable and consistent local file system state by focusing only on a subset of all possible scenarios, which makes studying the global inconsistency issues of PFS efficient. We use a mix of both approaches in this work.

(c) Network Partitioning (c-Network). This is a typical failure scenario in large-scale networked systems [66], which may be caused by dysfunctional network devices (e.g., switch [67]) or hanging server processes among others [62]. When the failure happens, the cluster splits into more than one "partitions" which cannot communicate with each other.

To emulate the network partitioning 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 [4, 5, 56–62]. By emulating these fault models automatically, PFault enables studying the failure recovery and logging of the target PFS efficiently. Note that in all three cases, PFault introduces the faults from outside of the target PFS (e.g., iSCSI driver below the target PFS's local modules), which ensures the non-intrusiveness to the target PFS. Also, since there are multiple types of storage nodes (e.g., MGS, MDS, OSS) in a typical PFS, a fault 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. In this study, we cover the behaviors of PFSes when faults occurred on each and every type of PFS nodes (§5).

Since PFSes are traditionally optimized for high performance, one might argue that it is perhaps acceptable if the target PFS cannot function normally after experiencing these faults. However, we expect the checking and repairing component of the target PFS (e.g., LFSCK [31] for Lustre and BeeGFS-FSCK [2] for BeeGFS) to be able to detect the potential corruptions in PFS and response properly (e.g., do not hang or crash during checking). Also, we expect the corresponding failure logging component to be able to generate meaningful messages. We believe understanding the effectiveness of such failure handling mechanisms is a fundamental step towards addressing the catastrophe occurred at HPC centers in practice [17].

3.3 PFS Worker

Compared with a fresh file system, an aged file system is more representative of real-world file system usage [68, 69]. Also, an aged file system is more likely to encounter recovery issues under fault 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 §3.2.1.

Besides unmodified data-intensive workloads, another type of useful workloads is customized applications specially designed for examining the recoverability of the PFS. For example, the workload may embed checksums in the data written to the PFS. The checksums can be used by the end user to identify the potential corruptions of files stored in the PFS directly. In this way, the integrity of the user data can be verified without relying on the report of the target PFS (which

might be incorrect). The current prototype of PFault includes examples of both types of workloads, which will be described in details in §4.3.

3.4 PFS Checker

Similar to local file systems, maintaining internal consistency and data integrity is critical for large-scale storage systems including PFSes. Therefore, PFSes typically include a FSCK component (e.g., LFSCK, BeeGFS-FSCK, PVFS2-FSCK) to serve as the last line of defense to recover PFS after faults (§2.1).

The PFS Checker of PFault invokes the default FSCK component of the target PFS to recover the PFS after injecting faults with a tunable delay (i.e., the FSCK delay). Note that if the FSCK component is not designed or implemented properly (which is not uncommon as will be discussed §5), the FSCK itself may hang and thus disturb the automatic workflow of PFault. Therefore, the PFS Checker of PFault includes a tunable time threshold to kill the FSCK procedure in case it becomes non-responsive. Moreover, to verify if the default FSCK can recover PFS properly, the PFS Checker also invokes a set of customized and verifiable checking workloads to access the post-FSCK PFS. This enables examining the PFS' recoverability from the end user's perspective based on the responses of the workloads without relying on FSCK or PFS logs. Examples of such workloads include I/O intensive programs with known checksums for data or known return values for I/O operations. More details will be described in §4.3. Note that the workloads may also become non-responsive because the default FSCK may not be able to fully recover the target PFS. Therefore, PFault also includes a time threshold to kill the non-responsive workloads.

3.5 Orchestrator

To reduce the manual effort as much as possible, the Orchestrator component controls and coordinates the overall workflow of PFault automatically. First, the Orchestrator controls the formatting, installation, deployment of all PFS images via iSCSI to create a valid PFS cluster for study. Next, it coordinates the other three components (i.e., PFS Worker, Failure State Emulator, PFS Checker) to apply workloads, emulate failure events, and perform post-fault checks accordingly as described in §3.3, §3.2, and §3.4. In addition, it collects the extensive logs generated by the target system during the experiment and classifies them based on both time (e.g., pre-fault, post-fault) and space (e.g., logs from MGS, MDS, or OSS) for further investigation.

3.6 Non-iSCSI PFault

By leveraging the remote storage protocol (§2.3), PFault can create a target PFS cluster and perform fault injection testing with little manual effort. While remote storage protocols including iSCSI are transparent to the upper-layer software by design, one might still have concern on the potential impact of iSCSI on the failure behavior of the target PFS. To address the concern, we develop a non-iSCSI version of PFault for verifying the PFS behavior without iSCSI.

As shown in Figure 1b, the target PFS is deployed on the physical devices (instead of virtual devices) of PFS nodes in case of non-iSCSI PFault. The *PFS Worker* and *PFS Checker* are the same as that of the default iSCSI-based version, while the *Failure State Emulator* and the *Orchestrator* are adapted to avoid iSCSI.

Specifically, the emulation methods of the three fault models (§3.2.2) are adapted to different degrees. First, *Network Partitioning* (c-NetWork) can be emulated without any modification because disabling network card(s) is irrelevant to iSCSI. Second, the emulation of *Global Inconsistency* (b-Inconsist) is modified to access the local file system on the physical device of a selected storage node directly, instead of manipulating an iSCSI virtual image file. Third, *Whole Device Failure* (a-DevFail) cannot be emulated conveniently without iSCSI (or introducing other modification to the local

software stack), so we have to leave it as a manual operation. The Orchestrator component is split accordingly to enable inserting manual operations (e.g., unplug a hard drive) between automatic steps (e.g., applying pre-fault and post-fault workloads on PFS). Since the non-iSCSI PFault is designed only for verification purpose, we expect the low-efficient manual part to be acceptable.

3.7 Putting It All Together

In this subsection, we summarize PFault's overall workflow including both iSCSI-based version and non-iSCSI-based version. Since both versions share a number of common steps, we summarize them together in Algorithm 1.

First of all, there are multiple inputs needed to execute the PFault workflow, including the PFault mode '*M*' (i.e., iSCSI or non-iSCSI), a set of PFS cluster configurations '*C*' (e.g., the number of PFS nodes, the hostname and IP address of each node), a set of PFault internal configurations (e.g., fault model '*F*', target node '*N*' to apply the fault, time threshold '*T*' for determining hang). We omit other minor parameters (e.g., delay time for invoking FSCK) for clarity. The outputs of the workflow include a status file (i.e., '*STAT_REC*') recording the target PFS and FSCK's behaviors as well as a set of log files (i.e., '*LOG_REC*') collected at different steps of the workflow. Note that the entire workflow is controlled by the *Orchestrator* (§3.5) which is invisible in Algorithm 1 for simplicity.

More specifically, the workflow includes the following major steps as shown in Algorithm 1:

(1) Cluster Setup (line 3 to 8): If PFault is executed in iSCSI-based mode, we first connect each PFS node to a virtual device via iSCSI. In case of non-iSCSI mode, no special iSCSI setup is needed because we directly use the physical devices on the nodes. Then, PFault formats the PFS devices (either iSCSI devices or physical devices) and mounts the formatted PFS based on the PFS commands and configurations.

(2) Pre-Fault Stage (line 9 to 11): The *PFS Worker* described in §3.3 (i.e., '**PWorker**') applies aging and verifiable workloads to wear the brand-new PFS and to enable verifying post-FSCK PFS behavior later, respectively. Moreover, PFault collects all the logs after applying the workloads, which consists of normal logs generated during the cluster setup and regular I/O operations before fault injection (i.e., '*LOG_REC*.1').

(3) Fault Injection (line 12 to 20): The *Failure State Emulator* described in §3.2 (i.e., '**FSE**') applies a specified fault model *F* to the specified target node(s) *N*. For a-DevFail, in iSCSI-based mode (line 13, 14), PFault automatically disconnects the iSCSI device to emulate a whole device failure; in non-iSCSI mode (line 15, 16), PFault prompts to user and wait for the user to manually remove the physical device. In terms of the other two fault models (i.e., b-Inconsist and c-Network from line 17 to 20), there is no difference between iSCSI mode and non-iSCSI mode since the iSCSI layer is transparent in the two scenarios.

(4) PFS Recovery (line 21 to 25): The *PFS Checker* described in §3.4 (i.e., '**PChecker**') invokes the PFS' FSCK component (after a tunable delay) to recover the PFS (line 21). If the FSCK component hangs for more than a time threshold ('T'), it kills the process to continue the workflow (i.e., '*Kill_Upon_Hang*(*T*)' in line 22). The behavior of FSCK is recorded in *STAT_REC* (line 23). Also, the PFS logs and FSCK logs generated during the recovery are recorded in *LOG_REC* (line 24 and 25).

(5) Post-FSCK Verification (line 26 to 30): Besides running FSCK, *PFS Checker* executes additional post-FSCK workloads to further verify the PFS status after recovery (§3.4). Similar to the previous steps, hanging workloads will be killed after a time threshold. The behavior of the post-FSCK workloads is recorded in *STAT_REC* (line 28), which enables further verifying the PFS status based on the workload responses without relying on PFS FSCK report. The PFS logs generated during the post-FSCK workloads are also recorded for further analysis (line 29).

Algorithm 1: PFault Workflow

Input: PFault modes M which inludes iSCSI mode and non-iSCSI mode; PFS cluster configuration C; fault model F; target node(s) N; time threshold T
Output: *STAT_REC* records PFS and FSCK status and behaviors; *LOG_REC* records logs generated by PFS and FSCK;

Workflow PFault(M, C, F, N, T)
  if *M is iSCSI* **then**
    /* iSCSI mode requires every PFS node to establish one iSCSI virtual device */
    iSCSI_setup();
  else
    /* non-iSCSI mode uses local physical devices directly so no iSCSI setup is needed */
    skip;
  Format_PFS(*C*);
  Mount_PFS(*C*);
  /* PFS Worker applies aging and verifiable workloads */
  PWorker.*aging*;
  PWorker.*verifiable*;
  /* record logs before fault injection (phase 1) */
  *LOG_REC.phase*1 ← *PFS_Log_Dump*(*C*);
  if *F is a-DevFail* **then**
    if *M is iSCSI* **then**
      /* disconnect the iSCSI virtual device automatically in iSCSI mode */
      FSE.*DevFail*(*N*);
    else
      /* in non-iSCSI mode, wait for manual removal of physical device */
      pause;
  if *F is b-Inconsist* **then**
    FSE.*Inconsist*(*N*);
  if *F is c-Network* **then**
    FSE.*Network*(*N*);
  PChecker.*FSCK*;
  Kill_Upon_Hang(*T*);
  /* record FSCK status; record PFS logs and FSCK logs (phase 2) */
  STAT_REC ← **PChecker**.*FSCK.status*;
  *LOG_REC.phase*2 ← *PFS_Log_Dump*(*C*);
  *LOG_REC.phase*2 ← *FSCK_Log_Dump*(*C*);
  /* apply post-FSCK workloads to further verify the PFS status; record workload behaviors; record PFS logs (phase 3) */
  PChecker.*postFSCK.wkld*;
  Kill_Upon_Hang(*T*);
  STAT_REC ← **PChecker**.*postFSCK.status*;
  *LOG_REC.phase*3 ← *PFS_Log_Dump*(*C*);
  return *STAT_REC, LOG_REC*

(6) Finally, the workflow ends by returning *STAT_REC* and *LOG_REC* for in-depth investigation.

Note that while the *Orchestrator* of PFault automates the entire workflow to a great extent, the target PFS may behave extremely badly during the workflow (e.g., crash or reboot as will be discussed in §5). In such cases, the automatic workflow may be interrupted and manual intervention may be needed. We believe it is possible to integrate the PFault prototype with additional virtual machine provisioning (with iSCSI mode) or bare-metal provisioning (with non-iSCSI mode) techniques to reduce the manual intervention further, which we leave as future work.

4 EXPERIMENTAL METHODOLOGY

We build a prototype of PFault (including both iSCSI and non-iSCSI versions) and apply it to study two representative PFSes: Lustre and BeeGFS. In this section, we introduce the experimental platforms (§4.1), the target PFSes (§4.2), and the workloads used by PFault in this study (§4.3). Also, we summarize the experimental efforts in §4.4 and the communications with developers in §4.5. We defer the discussion of detailed study results to the next section (§5).

4.1 Experimental Platforms

As mentioned in §2.1, a typical production PFS cluster may include one MGS node, one or two dedicated MDS node(s), and two or more OSS nodes. We follow such typical setup in our experiments.

Specifically, we first create a seven-node cluster on virtual machines (VMs) hosted on one high-end physical server (Intel Xeon Gold 2.3GHz CPU x2, 256GB DRAM, 960GB SSD, 2TB HDD). In this seven-node main cluster, one node is used for hosting the Failure State Emulator and Orchestrator of PFault, and another node is used as a login/compute node to host PFS Worker and PFS Checker and to launch workloads on behalf of clients. The remaining five nodes are dedicated to the target PFS as storage nodes, which includes one MGS node, one MDS node, and three OSS nodes. On each storage node, there is one iSCSI virtual device mounted to serve as the corresponding target device (i.e., MGT/MDT/OST). This VM-based cluster enables us to deploy PFSes and investigate their behaviors using iSCSI-based PFaultconveniently.

In addition, to ensure reproducibility and to investigate the potential impact of iSCSI on the PFS behaviors, we use another two platforms, including: (a) A twenty-node cluster created on CloudLab [70] where 18 nodes are dedicated to the target PFS with 1 MGS, 1 MDS, and 16 OSS; this cluster is used for verifying that the results observed in the previous private platform are reproducible in the public cloud environment at scale. (b) A four-node cluster consisting of four private physical servers where all physical nodes are used by the target PFS with 1 MGS, 1 MDS, and 2 OSS; the PFault server is co-located with the PFS (i.e., on a PFS node which is not selected for fault injection). This cluster is used for verifying the behaviors of PFSes without iSCSI, i.e., the platform allows us to apply the non-iSCSI PFault for verification conveniently given the easy access to physical devices on different physical servers.

All results presented in §5 and Appendix (§A) are based on experiments using the iSCSI-based PFault on the first seven-node main cluster. Moreover, a subset with unexpected symptoms (e.g., hang, rebooting) has been reproduced and verified on CloudLab (using iSCSI-based PFault) and the four-physical-server cluster (using non-iSCSI PFault). The results are consistent across different platforms and different PFault modes in our experiments. In other words, the impact of iSCSI on the abnormal behaviors observed in our experiments is negligible, which is expected because the iSCSI layer is transparent to the PFS kernel modules. Therefore, we do not differentiate between iSCSI or non-iSCSI modes in the following sections.

4.2 Target PFSes

We have studied three versions of Lustre (v2.8.0, v2.10.0 and v2.10.8) and one version of BeeGFS (v7.1.3) in this work. The latest version of Lustre when we started our study was v2.8.0, which is the first minor version of the 2.8 series (referred to as v2.8 in the rest of the paper). Lustre has evolved to the 2.10 series in the last two years. To reflect the advancement, we apply the same set of experiments on two additional versions: v2.10.0 and v2.10.8. For simplicity, we refer to them together as v2.10.x in the rest of the paper. The experimental results (§5) are consistent across versions unless otherwise specified.

In terms of local OS, we use CentOS 7.2 (Linux kernel v3.10.0-327.3.1.el7 with `ldiskfs` patches) for Lustre v2.8, CentOS 7.6 (kernel v3.10.0-957.1.3.el7 with `ldiskfs` patches) for Lustre v2.10.x, and CentOS 7.5 (kernel v3.10.0-1062.el7.x86_64 with Ext4) for BeeGFS v7.1.3, all of which are the default or recommended setup for the target PFS.

4.3 Workloads

Table 1. **Workloads Used for Studying Lustre and BeeGFS.**

Workload	Description	Purpose
`cp+tar+rm`	copy, compress/decompress, & delete files	age target PFS
`Montage-m101`	an astronomical image mosaic engine	age target PFS
`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-FSCK behavior
`WikiW-async`	write new files asynchronously, read back & verify MD5	analyze post-FSCK behavior
`WikiW-sync`	write new files synchronously, read back & verify MD5	analyze post-FSCK behavior

Table 1 summarizes the workloads included in the current PFault prototype for this study. As shown in the table, `cp+tar+rm` is a set of common file operations (i.e., copying, compressing/decompressing, and deleting files) for aging the PFS under study. `Montage-m101` is a classic HPC application for creating astronomical image mosaics [71], which is used for deriving the target PFS to a representative state. The Wikipedia workloads (i.e., `WikiW-init`, `WikiR`, `WikiW-async`, `WikiW-sync`) use a dataset consisting of archive files of Wikipedia [72]. Each archive file has an official MD5 checksum for self-verifying its integrity. PFault makes use of such property to examine the correctness of PFS states after executing PFS FSCK (e.g., LFSCK for Lustre, BeeGFS-FSCK for BeeGFS).

4.4 Experimental Efforts

The current prototype of PFault is implemented as bash scripts integrating with a set of Linux and PFS utilities (e.g., debugfs [65], LFSCK, BeeGFS-FSCK). The iSCSI-based PFault is built on top of the Linux SCSI Target Framework [29] with additional 1168 Lines of Code (LOC) for the Failure State Emulator, PFS Worker, PFS Checker, and Orchestrator (§3). The non-iSCSI PFault is a variant of the iSCSI-based version, which differs in Failure State Emulator (77 LoC difference) and Orchestrator (106 LOC difference). Note that in both versions of PFault, only about 200 LoC is Lustre/BeeGFS specific (mainly for cluster setup and FSCK invocation).

In total, we have performed around 400 different fault injection experiments covering three fault models (§3.2.2) on five different combinations of PFS nodes (i.e., 1 MGS node, 1 MDS node, 1 OSS node, 3 OSS nodes, 1 MDS + 1 OSS nodes) using the seven-node main cluster. In each experiment, we collect the logs generated by PFS and its FSCK for further analysis. As mentioned in §3.7, the logs are collected at three different phases of each experiment with the help of PFault to enable

thorough analysis: (1) after aging: this phase contains logs of PFS under normal condition (e.g., during cluster setup and the aging workloads); (2) after FSCK: this phase includes logs generated by PFS and FSCK after fault injection; (3) after post-FSCK workloads: this phase contains logs triggered by the post-FSCK workloads, which enables examining the PFS status based on the workload responses without relying on the FSCK report. All experiments are repeated at least three times to ensure that the results are reproducible.

Table 2. **Numbers of Log Files Studied**

Type of Log	Location of Log						**Sum**
	MGS	MDS	OSS#1	OSS#2	OSS#3	Client	
Lustre	135	135	135	135	135	135	810 Lustre + 405 LFSCK
LFSCK	N/A	135	90	90	90	N/A	(1,215 in total)
BeeGFS	135	135	135	135	135	135	810 BeeGFS + 90 BeeGFS-FSCK
BeeGFS-FSCK	N/A	N/A	N/A	N/A	N/A	90	(900 in total)

Table 2 summarizes the subset of log files used for in-depth manual study. In total, we have studied 1,215 log files for Lustre/LFSCK and 900 log files for BeeGFS/BeeGFS-FSCK. Lustre keeps a log buffer on each node of the cluster, so the numbers of log files collected on different node are the same (i.e., "135" in Lustre row). LFSCK has three steps on MDS and two steps on OSSes, and each step generates its own status log, so the number of log files on MDS ("135") is more than that on OSS ("90"). Similar to Lustre, BeeGFS keeps a log file on each node for debugging purpose and the log file is created as soon as a service or client is started. On the other hand, different from LFSCK, BeeGFS-FSCK logs are centralized on two separate files on the client node, which makes the collection relatively easy. A more detailed characterization of the logs is in Appendix §A.

Table 3. **Configurations Used in Experiments.** *Columns 2 to 9 show the main configurations for experiments; columns 10 to 14 show additional configurations for verification; the last column shows that the results are reproducible and consistent ('Y').*

Target PFS	Main Configurations for Experiments								Additional Config. for Verification					Res. Rep. ?
	Local FS	Number of Nodes			stripe count	stripe size	FSCK delay	PFault Mode	No. of OSS	stripe count	stripe size	FSCK delay	PFault Mode	
		MGS	MDS	OSS										
Lustre	ldiskfs	1	1	3	3	64KB	10s	iSCSI	16	16	64KB	30s	iSCSI	Y
									16	8	256KB	15s	iSCSI	Y
									2	2	512KB	10s	non-iSCSI	Y
									2	1	1MB	5s	non-iSCSI	Y
BeeGFS	Ext4	1	1	3	3	64KB	10s	iSCSI	16	8	64KB	30s	iSCSI	Y
									16	8	256KB	15s	iSCSI	Y
									2	2	512KB	10s	non-iSCSI	Y
									2	1	1MB	5s	non-iSCSI	Y

In addition, as mentioned in §4.1, we have further reproduced and verified a subset of experiments with abnormal symptoms (e.g., hang, kernel panics) on a CloudLab cluster (with the iSCSI mode of PFault) and a four-physical-server private cluster (with the non-iSCSI mode of PFault). We find that the abnormal symptoms are reproducible across different platforms and different modes of PFault. We have also tuned another three parameters (i.e., `stripe count`, `stripe size`, FSCK delay) to a wide range of non-default values, and we fine that the results are consistent (i.e., insensitive to the parameters tuned). We summarize all the configurations that have been used in our experiments in Table 3. Since the results are reproducible and consistent, we do not differentiate between different configurations in the following sections. Note that we do not claim our configurations are exhaustive. Given the complexity of PFS, there is almost an infinite number of ways to configure a PFS cluster. But we believe that our platforms and configurations are representative as they cover the default parameters of PFS clusters widely used in practice. More importantly, our setups have helped identify real problems of PFS confirmed by the PFS developers (§4.5). We discuss other hardware-dependent configuration issues further in §6.3.

4.5 Confirmation with Developers

For the abnormal symptoms observed in the experiments, we try our best to analyze the root causes based on the extensive PFS logs, source code, as well as communications with the developers. For example, in our preliminary experiments [33, 73] we observed a resource leak problem where a portion of the internal namespace as well as the storage space on OSTs became unusable by Lustre after running LFSCK. We analyzed the root cause and discussed with the developers, and eventually found that the "leaked" resources may be moved to a hidden "`lost+found`" directory in Lustre by LFSCK when a parameter is set. While the resources are still not usable directly, it is no longer a leak problem. Therefore, we skip the discussion of the issue in this article. On the other hand, our root cause analysis of a kernel panic problem has been confirmed by developers, and a new patch set has been generated based on our study to fix the problem in the coming Lustre release [34]. We discuss the patch set in details in §5.2.2.

5 STUDY RESULTS

In this section, we present the study results on Lustre and BeeGFS. The results are centered around FSCK (i.e., LFSCK and BeeGFS-FSCK) since FSCK is the major recovery component for handling PFS level issues after faults.

First (§5.1), we analyze the target PFS including its FSCK from the end user's perspective (e.g., whether a program can finish normally or not). We present the behavior of the PFS under a variety of conditions enabled by PFault (i.e., different fault models applied on different types of storage nodes), and identify a set of unexpected and abnormal symptoms (e.g., hang, I/O error).

Next (§5.2), we study the failure logs and the root causes of abnormal symptoms. We identify the unique logging methods and patterns of Lustre and BeeGFS. Moreover, based on the information derived from the logs as well as the PFS source code and the feedback from the developers, we pinpoint the root causes of a subset of the abnormal behaviors observed (e.g., reboot).

Third (§5.3), to further understand the recovery procedures of the PFS after faults, we characterize the FSCK-specific logs generated under the diverse conditions. By detailed characterization, we find that FSCK logs may be incomplete or misleading, which suggests opportunities for further improvements (§6).

In addition, we characterize the extensive logs triggered by non-FSCK components of the target PFS in details. For clarity, we summarize the additional results in Appendix (§A).

We would like to clarify that the goal of this study is not to compare Lustre with BeeGFS or to imply which PFS is better. We study Lustre and BeeGFS because (1) both of them are widely used in practice and deserve our efforts, (2) neither of them is perfect in terms of failure handling as far as we know, and (3) they represent different design tradeoffs. So we hope to identify the potential limitations as well as the opportunities for improving both Lustre and BeeGFS. Also, we do not claim that our results are conclusive or complete. Due to the complexity of PFSes, we believe our results only represent a subset of all possible behaviors of Lustre and BeeGFS, and the results may not be translated directly to interpret other PFSes. We will discuss the general lessons learned and the opportunities for further improvements (including extending to other PFSes) in §6.

5.1 Behavior of PFS FSCK and Post-FSCK Workloads

In this subsection, we present the behavior of LFSCK and BeeGFS-FSCK as perceived by the end user when recovering the PFS after faults. As mentioned in §3.4 and §4.3, PFault applies a set of self-verifiable workloads *after* LFSCK/BeeGFS-FSCK (i.e., post-FSCK) to further examine the effectiveness of FSCK. While we do not expect the target PFS to function normally after faults, we

Table 4. **Behavior of LFSCK and Post-LFSCK Workloads.** *The first column shows where the faults are injected. The second column shows the fault models applied. The remaining columns show the responses. "normal": LFSCK appears to finish normally; "reboot": at least one OSS node is forced to reboot; "Invalid": report an "Invalid Argument" error; "I/O err": report an "Input/Output error"; "hang": cannot finish within one hour; "corrupt": checksum mismatch; "✓": complete w/o error reported.*

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**	I/O err	I/O err	I/O err
	a-DevFail (v2.10.x)	**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.x)	**hang**	hang	hang	hang
OSS#1	a-DevFail	**hang**	hang	hang	hang
	a-DevFail (v2.10.x)	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.x)	normal	✓	hang	hang
	b-Inconsist	**reboot**	corrupt	hang	hang
	c-Network	**hang**	hang	hang	hang
MDS + OSS#1	a-DevFail	**Invalid**	hang	hang	hang
	a-DevFail (v2.10.x)	**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.x)	**hang**	hang	hang	hang

expect LFSCK/BeeGFS-FSCK, which is designed to handle the post-fault PFS, to be able to behave properly (e.g., do not hang) and/or identify the underlying corruptions of the PFS correctly.

5.1.1 ***LFSCK.*** Table 4 summarizes the behavior of LFSCK and the behavior of the self-verifiable workloads after running LFSCK. As shown in the first column, we inject faults to five different subsets of Lustre nodes: (1) MGS only, (2) MDS only, (3) one OSS only, (4) all three OSSes, and (5) MDS and one OSS. For each subset, we inject faults based on the three fault models (§3.2). For simplicity, in case of only one OSS is affected, we only show the results on OSS#1; the results on OSS#2 and OSS#3 are similar.

We add the behavior of Lustre/LFSCK v2.10.x when it differs from that of v2.8. As mentioned in §4, we studied two subversions of 2.10.x (i.e., v2.10.0 and v2.10.8). Since the two subversions behave the same in this set of experiments, we combine the results together (i.e., the "v2.10.x" lines).

When faults happen on MGS (the "MGS" row), there is no user-perceivable impact. This is consistent with Lustre's design that MGS is not involved in the regular I/O operations after Lustre is built [25].

When faults happen on other nodes, however, LFSCK may fail unexpectedly. For example, when "a-DevFail" happens on MDS (the "MDS" rows), LFSCK fails with an "Invalid Argument" error ("**Invalid**") and all subsequent workloads encounter errors ("**I/O err**"). Arguably, the workloads' behavior might be acceptable given the fault, but the LFSCK behavior is clearly sub-optimal because it is designed to scan and check a corrupted Lustre gracefully. Such incompleteness is consistent with the observations on local file system checkers [41, 74].

Table 5. **Behavior of BeeGFS-FSCK and Post-FSCK Workloads.** *The first column shows where the faults are injected. The second column shows the fault models applied. The remaining columns show the responses. "normal": BeeGFS-FSCK appears to finish normally; "aborted": terminate with an "Aborted" error; "NoFile": report a "No such file or directory" error; "comm err": report a "communication error"; "I/O err": report an "Input/Output error"; "hang": cannot finish within one hour; "corrupt": checksum mismatch; "✓": complete w/o error reported.*

Node(s) Affected	Fault Models	BeeGFS-FSCK	`WikiR`	`WikiW-async`	`WikiW-sync`
MGS	a-DevFail	normal	✓	✓	✓
	b-Inconsist	normal	✓	✓	✓
	c-Network	**hang**	✓	✓	✓
MDS	a-DevFail	normal	NoFile	I/O err	I/O err
	b-Inconsist	normal	NoFile	I/O err	I/O err
	c-Network	**aborted**	comm err	comm err	comm err
OSS#1	a-DevFail	**aborted**	corrupt	I/O err	I/O err
	b-Inconsist	normal	corrupt	✓	✓
	c-Network	**aborted**	comm err	comm err	comm err
three OSSes	a-DevFail	**aborted**	corrupt	I/O err	I/O err
	b-Inconsist	normal	corrupt	✓	✓
	c-Network	**aborted**	comm err	comm err	comm err
MDS+OSS#1	a-DevFail	**aborted**	NoFile	I/O err	I/O err
	b-Inconsist	normal	NoFile	I/O err	I/O err
	c-Network	**aborted**	hang	comm err	comm err

When "a-DevFail" happens on OSS (the "OSS#1" row), v2.8 and v2.10.x differ a lot. On v2.8, LFSCK and all workloads hang. However, on v2.10.x, 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 `ldiskfs` backend file system, which purges the buffer cache. After remounting, running `WikiR` becomes "hang" (same as v2.8). This suggests that v2.10.x has a more aggressive buffering mechanism compared with v2.8. `WikiW-sync` and `WikiW-aync` can succeed because v2.10.x 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. Compared to the "hang" on v2.8, this is indeed an improvement.

When "a-DevFail" happens on both MDS and OSS (the "MDS+OST#1" row), v2.8 and v2.10.x also behave differently. The behaviors of LFSCK and subsequent workloads in v2.8 (i.e., "**Invalid**" and **hang**") change to "Input/Output error" ("**I/O err**") on v2.10.x, which is an improvement since "I/O err" is closer to the root cause (i.e., a device failure emulated by PFault).

When "b-Inconsist" happens on MDS (the "MDS" row), it is surprising that LFSCK finishes normally without any warning ("normal"). In fact, LFSCK's internal logs also appear to be normal, as we will discuss in §5.3. Such behavior suggests that the set of consistency rules implemented in LFSCK is likely incomplete, similar to the observation on local file system checkers [41, 74].

When "b-Inconsist" happens on OSS (the "OSS#1" row), running LFSCK may crash storage nodes and trigger rebooting abruptly ("**reboot**" in the "LFSCK" column). We will discuss the root cause of the abnormality in details in §5.2.2. Note that `WikiR` reports mismatched checksums ("corrupt") in this case. This is because OSTs store the striped data, "b-Inconsist" on OSTs affects the user data files, which leads to checksum mismatch in the workload.

5.1.2 ***BeeGFS-FSCK.*** Table 5 summarizes the behavior of BeeGFS-FSCK and the behavior of the workloads after running BeeGFS-FSCK. In general, we find that compared with LFSCK, BeeGFS-FSCK's behavior is more unified, i.e., there are fewer types of unexpected symptoms.

Specifically, when faults occur on MGS (the "MGS" row), there is little user-perceivable impact. For example, BeeGFS-FSCK finishes normally under "a-DevFail" and "b-Inconsist" fault models and all workloads finish successfully (i.e., "✓"). This is because MGS is mostly involved when adding/removing nodes to/from the BeeGFS cluster. However, we do observe a difference between BeeGFS-FSCK and LFSCK (Table 4): when applying "c-Network" to MGS, BeeGFS-FSCK may "**hang**" (i.e., no progress within one hour), while LFSCK always finishes normally. On one hand, the hang symptom suggests that BeeGFS-FSCK is more complete because it checks the network connectivity among all nodes including MGS in the cluster. On the other hand, such behavior implies that BeeGFS-FSCK itself cannot handle the case gracefully.

When we apply the fault models to other nodes, there are different responses. For example, when "a-DevFail" or "b-Inconsist" happens on MDS (the "MDS" row), BeeGFS-FSCK appears to complete normally ("normal"). However, BeeGFS-FSCK is unable to fix the inconsistency. As a result, `WikiR` may fail with an "No such file or directory" error ("NoFile") when reading files from the client, and `WikiW-async` and `WikiW-sync` may fail with an "Input/Output error" ("I/O err"). Also, it is interesting to see that BeeGFS-FSCK may response to the device failure on MDS ("a-DevFail") and metadata inconsistency ("b-Inconsist") in the same way (i.e., both appear to be "normal"), which suggests that the current recovery policy is not thorough enough to identify the either of the issues.

When "a-DevFail" or "c-Network"occurs on OSS (the "OSS#1" and "three OSSes" rows), BeeGFS-FSCK often aborts ("**aborted**"). While aborting may be understandable because the data on OSSes become unaccessible under either of the two fault models, the same simple and abrupt response is not helpful for identifying the underlying issue of the PFS cluster, let alone fixing it. Unsurprisingly, after FSCK, `WikiR` may still report checksum mismatch ("corrupt") as the checksum is calculated based on partial file data. `WikiW-async` and `WikiW-sync` may still encounter I/O error ("I/O err") as they cannot access the data on the affected OSS node(s).

5.1.3 ***Summary.*** In summary, the behaviors of Lustre and BeeGFS under the three types of faults are diverse. The symptoms are also dependent on where the faults occur in the PFS cluster. There are multiple cases where FSCK itself may fail unexpectedly (e.g., hang, abort) when attempting to recover the post-fault PFS. In some cases, the FSCK may appear to complete normally without reporting any issue, but the underlying PFS may still be in a corrupted state as exposed by the abnormal response of the subsequent workloads (e.g., I/O error). Table 4 and Table 5 summarizes the incompleteness of FSCK under different fault and node combinations, which may serve as a reference for further refining FSCK capability (See §6).

5.2 Failure Logs and Root Causes

In this subsection, we first characterize the failure logs generated by Lustre and BeeGFS, and then analyze the root causes of a subset of the abnormal symptoms described in §5.1 based on the failure logs, the PFS source code, as well as communications with the developers.

5.2.1 ***Failure Logs of PFS.*** We observe that both Lustre and BeeGFS may generate extensive logs during operations. Based on our investigation, Lustre maintains an internal debug buffer and generates logs on various events (including but not limited to failure handling) on each node of the cluster. Similarly, BeeGFS also maintains log buffers and generates extensive logs. Such substantial logging provides a valuable way to analyze the system behavior. We collect the log messages generated by the target PFS and characterize the ones related to the handling of failure events. In addition to the PFS logs, we find that the FSCK component itself may produce explicit status logs when it is invoked to recover the corrupted target PFS. For clarity, we defer the discussion of FSCK specific logs to the next section (§5.3).

Table 6. **Logging methods observed in Lustre and BeeGFS.**

Sources of Log	Lustre		BeeGFS	
	kernel space	user space	kernel space	user space
C function	printk(), seq_printf()	fprintf()	seq_printf()	printf(), fprintf()
C++ class	–	–	–	std::cerr, Logger, LogContext
Debugging Macro	CDEBUG(), CERROR()	–	–	LOG_DEBUG()

Table 7. **Standard & Equivalent Error Messages Captured in PFS Failure Logs.** *The customized error messages (i.e., h to k rows) are converted to the equivalent standard Linux error messages for clarity.*

ID	Error #	Error Name	Description	Logged by Lustre?	Logged by BeeGFS?
a	2	ENOENT	No such file or directory	Yes	
b	5	EIO	I/O error	Yes	Yes
c	11	EAGAIN	Try again	Yes	
d	16	EBUSY	Device or resource busy	Yes	
e	30	EROFS	Read-only file system	Yes	Yes
f	107	ENOTCONN	Transport endpoint is not connected	Yes	
g	110	ETIMEDOUT	Connection timed out	Yes	
h	CEM-2	ENOENT	No such file or directory		Yes
i	CEM-30	EROFS	Read-only file system		Yes
j	CEM-101	ENETUNREACH	Network is unreachable		Yes
k	CEM-110	ETIMEDOUT	Connection timed out		Yes

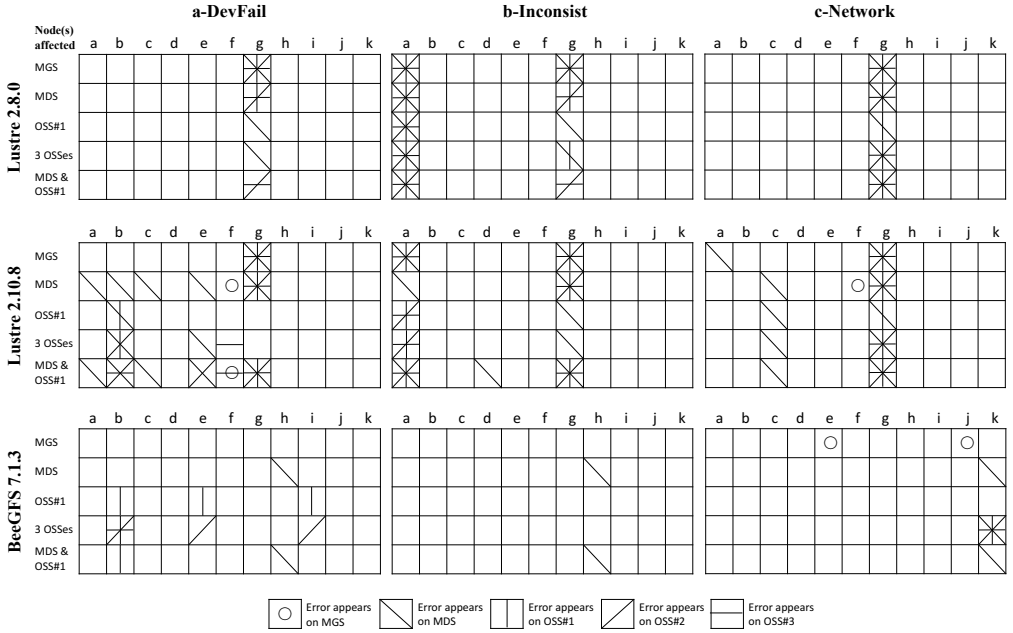

Fig. 2. **Distribution of PFS Error Messages.** *This figure shows the distribution of 11 types of standard error messages (i.e., 'a' to 'k') of Lustre (two major versions) and BeeGFS after applying three fault models (i.e., a-DevFail, b-Inconsist, c-Network). The "Node(s) Affected" column shows where the faults are injected.*

Logging Methods. We first look into how the logs are generated by the target PFS. Unlike modern Java-based cloud storage systems (e.g., HDFS, Cassendra) which commonly use unified and well-formed logging libraries (e.g., Log4J [32] or SLF4J [75]), we find that the logging methods of PFSes

are diverse and irregular. Table 6 summarizes the major methods used for logging in Lustre and BeeGFS. We can see that both Lustre and BeeGFS can generate logs from both kernel space and user space. The two PFSes have a few methods in common (e.g., *fprint*, *seq_printf*), but there are many differences. For example, Lustre uses a set of debugging macros (e.g., CDEBUG, CERROR) for reporting errors with different levels of severity, while BeeGFS uses customized logging classes (e.g., Logger, LogContext) in addition to debugging macro (e.g., LOG_DEBUG) for the same purpose. Moreover, the content and formats of the logs are diverse and irregular. Detailed examples can be found in Table 11, Table 12, and Table 13 of the Appendix (§A). Such diversity and irregularity make analyzing PFSes behaviors based on log patterns (e.g., CrashTuner [10]) challenging. On the other hand, it may also imply new opportunities for learning-based log analysis (See §6).

Patterns of Failure Logs. Given the diverse and irregular logs, we use a combination of three rules to determine if a log message is related to the failure handling activities or not. First, in terms of timing, a failure handling log message must appear after the fault injection. Second, we find that both Lustre and BeeGFS may use standard Linux error numbers or equivalent customized counterparts in their logging methods, so we consider logs with standard Linux error numbers or equivalent customized errors as failure handling logs. In addition, for logs appear after the fault injection but do not contain explicit standard or equivalent errors, we examine failure-related descriptions (e.g., "failed", "commit error", see §A for detailed examples) and double check the corresponding source code to determine their relevance. For clarify, we call the log messages that are related to the failure handling based on the three rules above as *error messages*. Note that the third rule above essentially describes the highly-customized error messages which are neither standard nor equivalent to standard error numbers. For clarify, we discuss those messages in Appendix (§A) and only focus on the standard messages (including the equivalent ones) in the rest of this section.

Table 7 summarizes the major standard and equivalent error messages captured in the two PFSes after fault injection in our experiments, which includes eleven types (i.e., 'a' to 'k') in total. We can see that Lustre mainly uses a set of seven standard Linux error numbers (e.g., '2', '5', '11', '16', '30', '107', '110') while BeeGFS only uses two standard error numbers (i.e., '5' and '30'). On the other hand, BeeGFS uses a few customized error messages which can be mapped to the standard Linux error numbers directly (i.e., row 'h' to 'k'). For clarity, the customized messages have been converted to their standard counterparts in Table 7 (e.g., 'CEM-2' in row 'h' is equivalent to the standard error number '2', both of which mean 'No such file or directory'). The specific examples of customized messages can be found in §A. The difference in the error message logging reflects the different design choices of the two PFSes: although both Lustre and BeeGFS contain Linux kernel modules, Lustre implements much more functionalities in the kernel space compared to BeeGFS. As a result, Lustre captures more standard Linux error numbers and messages directly.

Figure 2 further shows the distribution of the error messages after injecting three types of faults (i.e., a-DevFail, b-Inconsist, c-Network) on two major versions of Lustre and one version of BeeGFS. The "Node(s) Affected" column shows where the faults are injected. Columns 'a' to 'k' represent the eleven types of standard or equivalent error messages described in Table 7. The five different symbols represent the five different PFS nodes where an error message is observed. In case an error message is captured on multiple nodes under the same fault, we use superposition of symbols in the corresponding cell. For example, in Lustre v2.8.0, after PFault injects a-DevFail on MGS, the error message 'g' is captured on MDS, OSS#1, OSS#2 and OSS#3.

Based on Figure 2, we can clearly see that Lustre v2.10.8 generates error messages on more nodes with more standard Linux error numbers compared to Lustre v2.8.0. For example, after injecting a-DevFail to MDS, Lustre v2.10.8 generates error messages with 'a', 'b', 'c', 'e' on MDS, 'f' on MGS, and 'g' on MDS and all OSS nodes. On the other hand, Lustre v2.8.0 only reports 'g'

under the same fault. This implies that Lustre v2.10.8 has enhanced the failure logging significantly compared to v2.8.0. As discussed in the previous sections (e.g., Table 4), most faults are still not handled properly (e.g., v2.10.x may still expose I/O errors to users after FSCK), but we believe that the enhanced logging is one step in the right direction. As will be discussed in §5.2.2, we find that the enhanced logging is valuable in diagnosing the issues in PFSes.

Also, it is interesting to see that 'g' is heavily logged in the two Lustre versions under all three fault models. As mentioned in Table 7, 'g' means connection timed out, which implies that one or more PFS nodes are not reachable. This is expected because under all fault models one or more PFS nodes may crash, hang, or reboot, as described in Table 4. On the other hand, this observation implies that diagnosing the root causes of failures solely based on logs may be challenging because different faults may lead to the same error messages. Therefore, we believe that more fine-grained logging will likely be needed to address the challenge of PFS failure diagnosis.

Compared to Lustre, the distribution of BeeGFS's standard or equivalent error messages looks more sparse in Figure 2. For example, only 'h' is captured under b-Inconsist. This confirms that BeeGFS does not leverage standard Linux error numbers as much as Lustre does in terms of logging. However, this does not necessarily imply that BeeGFS's logging is less effective. In fact, we find that BeeGFS may generate a variety of customized error messages beyond the standard set of Linux error numbers. This reflects the trend of PFS development: Similar to many user-level cloud storage systems (e.g., HDFS), BeeGFS has implemented more functionalities in the user space with more customized error logging compared to the classic Lustre. Please refer to Appendix (§A) for more concrete examples and more detailed characterization of all error messages (including non-standard error messages).

5.2.2 ***Analysis of Error Propagation and Root Cause.*** The extensive logs collected in the experiments provide a valuable vehicle for understanding the behavior of the target PFS. By combining information derived from the experimental logs, the source code, and the feedback from the developers, we are able to identify the error propagation and root causes of a subset of the abnormal behaviors observed in our experiments. In the rest of this subsection, we further discuss why the Lustre checker LFSCK itself may exhibit abnormal behaviors during recovery using three specific examples (i.e., examples of "**I/O err**", "**hang**", "**reboot**" on v2.10.x in Table 4). We illustrate the three simplified cases using Figure 3.

Specifically, Figure 3 shows the critical error propagation path of Lustre and LFSCK under three fault scenarios, i.e., "a-DevFail" on MDS (Figure 3a), "b-Inconsistent" on OSS#1 (Figure 3b), and "c-Network" on MDS (Figure 3c), as defined in §3.2 and §5.1. Each bold black statement represents one internal function of Lustre, which is followed by a short description after it. The internal error codes are highlighted in red in parentheses after the corresponding functions. PFault operations are represented in blue. The red dash boxes highlight the key operations and errors leading to the observable abnormal behaviors. We discuss the three scenarios one by one below.

(1) a-DevFail on MDS ("I/O err"): When "a-DevFail" occurs on MDS, Lustre fails to access the log file immediately ("mgc_process_log" reports error number "-2"), though the error is invisible to the client. LFSCK is able to finish the preparation of its phase 1 normally ("osc_scrub_prepare"). However, the subsequent operations (e.g., "osc_ldiskfs_write_record") from LFSCK require accessing the MDT device, which cannot be accomplished because the MDT is unreachable. These operations generate I/O errors (e.g., "-5") that are eventually propagated to the client by MDS. As a result, the client observes "I/O err" when using LFSCK. Right after the I/O error is reported, we observe error number "-30" (i.e., read-only file system) on MDS. This is because the previous I/O error cannot be handled by Lustre's `ldiskfs` backend. To prevent further corruption, `ldiskfs` sets the file system to read-only [25]. As a result, the subsequent workloads, including LFSCK itself and also fail on

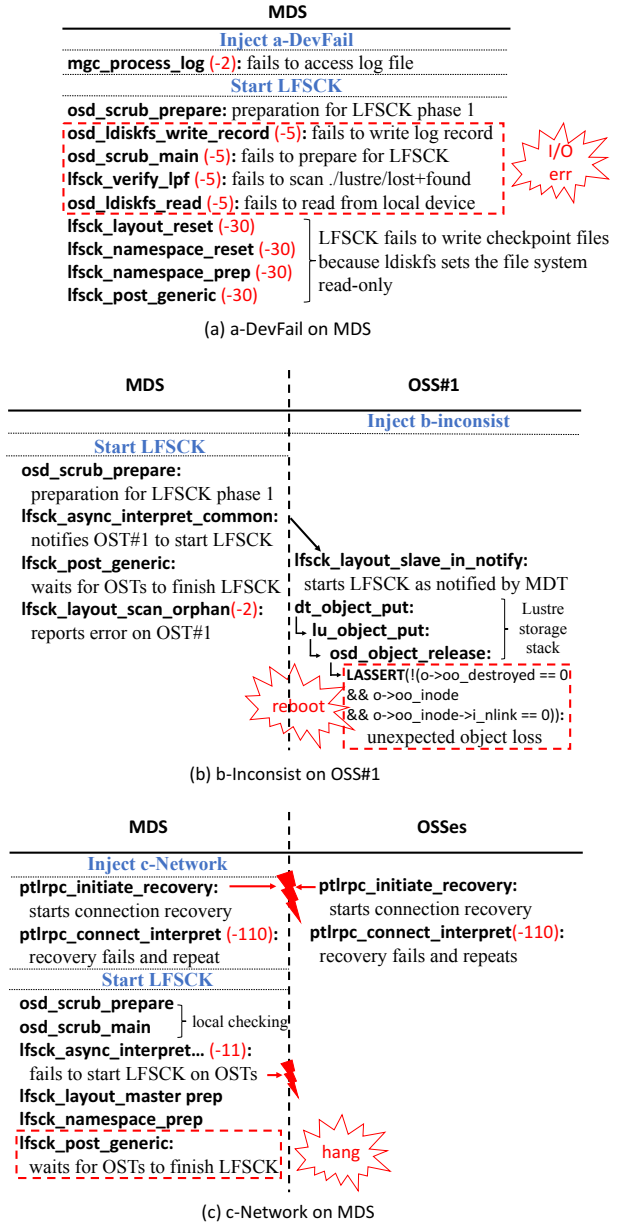

Fig. 3. **Internal Operations of Lustre and LFSCK After Three Types of Faults.** *Each bold black statement represents one Lustre function, which is followed by a short description. Blue lines represent PFault operations. Red dash boxes highlight the key operations leading to the abnormal symptoms observed by the end user.*

write operations. Since LFSCK is supposed to handle corrupted Lustre, we believe that a more elegant design of LFSCK could be verifying the accessibility of the device before issuing the I/O requests, which could avoid throwing out the I/O errors to the user abruptly during LFSCK. We discuss such optimization opportunities further in §6.

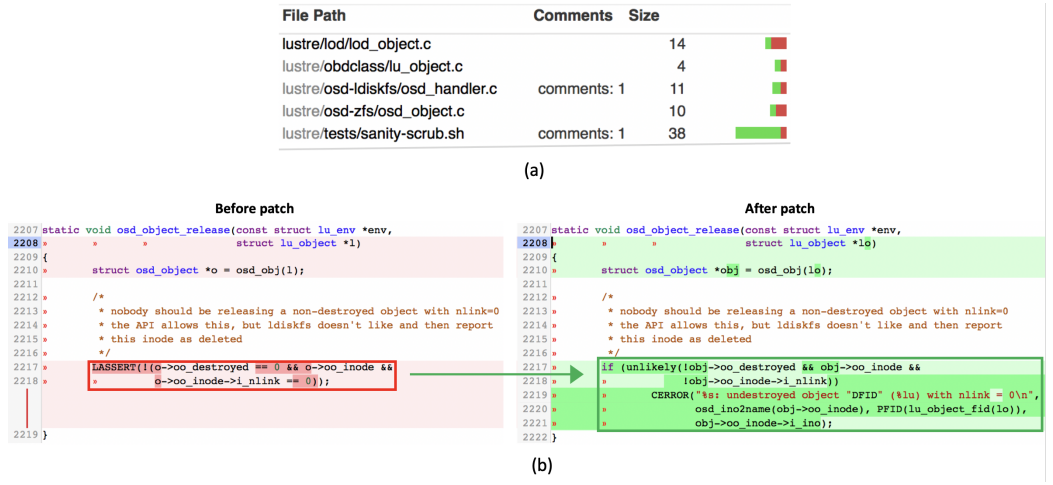

Fig. 4. **A Lustre patch set developed based on this study.** *(a) Five files have been modified in the patch set; the last file (sanity-scrub.sh) includes a new test case generated based on our report; (b) The key modification of the patch set in osd_handler.c.*

(2) b-Inconsist on OSS#1 ("reboot"): When the fault occurs, OSS#1 does not have any abnormal behaviors initially. When LFSCK is invoked on MDS by PFault, the LFSCK main thread on MDS notifies OSS#1 to start a local thread (i.e., the arrow from "lfsck_async_interpret_common" on MDS to "lfsck_layout_slave_in_notify" on OSS#1). The LFSCK thread on OSS#1 then initiates a put operation ("dt_object_put") to remove the object affected by the fault. The put request propagates through the local storage stack of Lustre, and eventually reaches the "OSD" layer ("osd_object_release"), which is the lowest layer of the Lustre abstraction built directly on top of local file system.

The "OSD" layer ("osd_object_release") checks an assertion ("LASSERT") before releasing the object, which requires that the Lustre file's flag "oo_destoryed" and attribute "oo_inode−>i_nlink" cannot be zero simultaneously. This is to ensure that when the Lustre object is not destroyed ('oo_destoryed" == 0), the corresponding local file should exist ("oo_inode−>i_nlink" != 0).

However, the two critical conditions in the assertion depend on Lustre and the local file system operations separately. "oo_destoryed" will be set to 1 by Lustre if Lustre removes the corresponding object, while "oo_inode−>i_nlink" will be set to 0 by local file system when the file is removed. Under the fault model, the local file system checker may remove the corrupted local file without notifying Lustre, leading to inconsistency between the state maintained by the local file system and the state maintained by Lustre. As a result, the assertion fails and triggers a kernel panic, which eventually triggers the "reboot". This subtle interaction between the local file system checker and LFSCK suggests that a holistic co-design methodology is needed to ensure the end-to-end correctness. Note that our analysis of the kernel panic issue has been confirmed by Lustre developers and a new patch set has been generated to fix the problem and other related issues based on our analysis[34]. We elaborate more on the patch set below.

Patch Description: Figure 4 shows the details of the patch set developed to fix the unexpected crash and related issues in Lustre. At the time of this writing, this patch set has involved five files and has been revised and tested for 17 rounds by the developers, which implies the complexity of the code base as well as the thoroughness of the patching procedure. As shown in Figure 4(a),

the five files modified by the patch set include "`lod_object.c`", "`lu_object.c`", "`osd_handler.c`", "`osd_object.c`", and "`sanity-scrub.sh`". Based on our analysis, one key modification is located in "`osd_handler.c`". As shown in Figure 4(b), in function "`osd_object_release`", the patch adds the "`unlikely`" macro to provide hints to the compiler to optimize the branch predication. More importantly, it replaces the diagnostic macro "LASSERT" with the debugging macro "CERROR", because the first assertion always triggers kernel panic when the `ldiskfs` on-disk inode state is different from its in-memory copy, which is an "overly-aggressive" bad behavior as commented by the developers. The modifications to other files include similar refinements to the error handling code paths. The last file (i.e., "`sanity-scrub.sh`") includes a new test case for sanity check derived from our report.

Based on our understanding, replacing the assertion with an error message might be a tentative workaround solution to avoid the immediate crash and reboot. The new test case added to "`sanity-scrub.sh`" is essentially a simplified procedure to emulate a specific scenario under the 'b-Inconsist' fault model in PFault (§3.2.2). By applying the test case in regression testing, the developers have identified other correlated issues in the code base which might require a careful re-design. For example, the developers observed another unexpected crash during the testing and commented as follows: "*I can't align 0x0000000c to any of the 'oo_inode', 'i_sb', 's_id', 'oo_header', or 'i_ino' fields in the parent structs ... It seems that 'oo_header' can be NULL in various places in the code, and the PFID() expansion to* $(fid)->f\_seq$, $(fid)->f\_oid$, $(fid)->f\_ver$ *is triggering on only 'f_ver' and not 'f_seq' (offset 0x0) or 'f_oid' (offset 0x8). We should really get the FID directly from the passed-in lu_object*". While the patch set is still under active revision and additional re-design may be needed due to the complexity of the code base, the fact that PFault has helped trigger the latent issue in production PFS and helped generate a new patch set including a new test case suggests the effectiveness and practical impact of this work.

(3) c-Network on MDS ("hang"): When the fault occurs, MDS can notice the network partition quickly because the remote procedure call (RPC) fails, and the RPC-related functions (e.g., functions with "ptlrpc" in name) may report network errors and repeatedly try to recover the connection with OSS. When LFSCK starts on MDS, its main thread has no trouble in processing the local checking steps (e.g., functions with "osd_scrub" in name return successfully). However, when the main thread tries to notify the OSS to start the LFSCK thread on OSS, the request cannot be delivered to OSS due to the network partition. After finishing the local checking steps on MDS, LFSCK keeps waiting ("lfsck_post_generic") for the OSS's response to proceed with global consistency checking. As a result, the system appears to be hanging from the client's perspective. We believe it would be more elegant for LFSCK to maintain a timer instead of hanging forever. We discuss such optimization opportunities further in §6.

5.3 Logs of LFSCK and BeeGFS-FSCK

In this subsection, we analyze the logs generated by LFSCK and BeeGFS-FSCK when they check and repair the post-fault target PFS to further understand the failure handling of the PFS.

5.3.1 ***LFSCK Logs.*** In addition to the failure logs of Lustre discussed in §5.2.1, we find that LFSCK itself may generate extensive status information in the `/proc` pseudo file system on the MDS and OSS nodes [31] as well as in the debug buffer of Lustre. For clarity, in this section we use *status log* to refer to LFSCK logs maintained in the `/proc` pseudo file system, and *debug buffer log* to refer to LFSCK-triggered events in the debug buffer of Lustre. We characterize these logs in details below.

We find that there are three types of LFSCK status 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

Table 8. **Characterization of LFSCK Status Logs Maintained in /proc.** *The first column shows where the faults are injected. The second column shows the fault models applied. "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.*

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
	a-DevFail (v2.10.8)	–	–	–	comp	comp	comp	comp	comp	comp
	b-Inconsist	comp	**repaired**	comp	comp	comp	comp	comp	comp	comp
	b-Inconsist (v2.10.8)	comp	comp	comp	comp	comp	comp	comp	comp	comp
	c-Network	init	init	init	init	init	init	init	init	init
	c-Network (v2.10.8)	comp	scan-1	scan-1	init	init	init	init	init	init
OSS#1	a-DevFail	scan	scan-1	init	–	–	comp	scan-2	comp	scan-2
	a-DevFail (v2.10.8)	comp	scan-1	comp	–	–	comp	comp2	comp	comp
	b-Inconsist	comp	scan-1	scan-1	comp	comp	comp	scan-2	comp	scan-2
	c-Network	scan	scan-1	init	init	init	comp	scan-2	comp	scan-2
	c-Network (v2.10.8)	comp	scan-1	scan-1	init	init	comp	scan-2	comp	scan-2
three OSSes	a-DevFail	scan	scan-1	init	–	–	–	–	–	–
	a-DevFail (v2.10.8)	scan	scan-1	comp	–	–	–	–	–	–
	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
	c-Network (v2.10.8)	comp	scan-1	scan-1	comp	comp	comp	comp	comp	comp
MDS + OSS#1	a-DevFail	–	–	–	–	–	init	init	init	init
	a-DevFail (v2.10.8)	–	–	–	–	–	comp	comp	comp	comp
	b-Inconsist	comp	**repaired**	scan-1	comp	comp	comp	scan-2	comp	scan-2
	b-Inconsist (v2.10.8)	comp	scan-1	scan-1	init	init	comp	scan-2	comp	scan-2
	c-Network	init	init	init	init	init	init	init	init	init
	c-Network (v2.10.8)	comp	scan-1	scan-1	init	init	comp	comp	comp	comp

the consistency between MDT and OSTs; (3) **namespace log (ns)**: checking the local/global namespace consistency inside/among MDT(s). 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 status logs are generated on MGS.

Table 8 summarizes the logs (i.e., "oi", "lo", "ns") generated on different Lustre nodes after running LFSCK. Similar to Table 4, we add the v2.10.8 logs when it differs from that of v2.8 (i.e., "v2.10.8" lines). As shown in the table, when "b-Inconsist" happens on MDS, LFSCK of v2.8 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 some of the orphan objects on OSSes which do not have corresponding parents (on MDS) correctly. On the other hand, when the same fault model is applied to Lustre v2.10.8 ("b-Inconsist (v2.10.8)"), LFSCK shows "comp" in the "lo" log (instead of "repaired"). This is likely because the randomness in introducing global inconsistencies in PFault (§3.2) leads to a different set of local files being affected on MDS. As a result, we did not observe the orphan object case on v2.10.8.

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., "–").

Table 9. **Characterization of LFSCK-triggered Logs in the Debug Buffer of Lustre v2.10.8 .** *Similar to Table 5, the "Node(s) Affected" column shows the node(s) to which the faults are injected. "–" means no error message is reported, while "x1" , "x2" and "x3" are failure messages corrsponding to the three phases of LFSCK: oi_scrub, lfsck_layout and lfsck_namespace, respectively. The meaning and example of each message type is shown at the bottom part of the table.*

Node(s) Affected	Fault Models	Logs on MDS	Logs on OSS#1	Logs on OSS#2	Logs on OSS#3
MDS	a-DevFail	*x1,x2,x3*	–	–	–
	c-Network	*x2*	–	–	–
OSS#1	a-DevFail	*x2*	*x1,x2*	–	–
	c-Network	*x2*	–	–	–
three OSSes	a-DevFail	*x2*	*x1,x2*	*x1,x2*	*x1,x2*
	c-Network	*x2*	–	–	–
MDS + OSS#1	a-DevFail	*x1,x2,x3*	–	–	–
	b-Inconsist	*x1,x2*	–	–	–
	c-Network	*x2*	–	–	–

Type	Meaning	Message Example
x1	oi_scrub failed	*...osd_scrub_file_store() sdb: fail to store scrub file, expected =... : rc = -5* *...1521:osd_scrub_main() sdb: OI scrub fail to scrub prep: rc = -5*
x2	lfsck_layout failed	*...fail to notify...for lfsck_layout start: rc = -5/-11/-30* *...lfsck_verify_lpf()...scan .lustre/lost+found/ for bad sub-directories: rc = -5* *...lfsck_post_generic()...waiting for assistant to do lfsck_layout post, rc = -30* *...lfsck_layout_store()...fail to store lfsck_layout: rc = -30* *...lfsck_layout_reset()...layout LFSCK reset: rc = -30* *...master engine fail to verify the .lustre/lost+found/... : rc = -5* *...layout LFSCK slave gets the MDT 0 status -11...* *...layout LFSCK hit first non-repaired inconsistency at...*
x3	lfsck_namespace failed	*...lfsck_namespace_prep()...namespace LFSCK prep failed: rc = -30* *...lfsck_namespace_reset()...namespace LFSCK reset: rc = -30*

When LFSCK hangs (i.e., "hang" in Table 4), the logs may keep showing that it is in scanning. We find that internally LFSCK uses a two-phase scanning to check and repair inconsistencies [31], and the "lo" and "ns" logs may further show the two scanning phases (i.e., "scan-1" and "scan-2"). In case the scanning continues for more than one hour without making any visible progress, we kill the LFSCK and show the hanging phases (i.e., "scan-1" or "scan-2") in 8.

Table 9 further summarizes the debug buffer logs triggered by LFSCK. We find that there are three subtypes of LFSCK debug buffer logs (i.e., *x1, x2, x3*), which corresponds to the three phases of LFSCK (i.e., *oi_scrub, lfsck_layout, lfsck_namespace*) respectively. Also, most logs are triggered on MDS (i.e., the "MDS" column), which implies that MDS plays the most important role for LFSCK execution and logging; and most of the triggered error messages are related to *lfsck_layout* (i.e., *x2*), which implies that checking the post-fault Lustre layout across nodes and maintaining data consistency is challenging and complicated. Moreover, there are multiple types of Linux error numbers (e.g., -5, -11, -30) logged, which implies that the *lfsck_layout* procedure involves and depends on a variety of internal operations on local systems. Since LFSCK is designed to check and repair the corrupted PFS cluster, it is particularly interesting to see that LFSCK itself may fail when the local systems are locally correct (i.e., "b-Inconsist" row).

To sum up, we find that in terms of LFSCK status logs, in most cases (other than the two "repaired" cases in Table 8), the logs are simply about LFSCK's execution steps (e.g., "init", "scan-1", "scan", "comp" in Table 8), which provides little information on the potential corruption of the PFS being examined by LFSCK. On the other hand, the corresponding debug buffer log of LFSCK is relatively more informative (Table 9), as it may directly shows the failed operations of LFSCK. To guarantee

Table 10. **Characterization of BeeGFS-FSCK Logs.** *The first column shows where the faults are injected. "conn" means the log shows FSCK is connected to the server/servers; "wait" means the log shows FSCK is waiting for mgmtd; "failed" means the log shows FSCK "connect failed"; "comp" means the output of FSCK is "normal"; "N/A" means the FSCK hangs without generating any output file; "orphaned chunk" means "Checking: Chunk without an inode pointing to it"; "wrong attributes" means "Attributes of file inode are wrong"; "metadata err" means "Communication with metadata node failed"; "fetch err" means "An error occurred while fetching data from servers".*

Node(s) Affected	Fault Models	Status Logs (**.log*)	Checking Logs (**.out*)
MGS	a-DevFail	conn	normal
	b-Inconsist	conn	normal
	c-Network	wait	N/A
MDS	a-DevFail	conn	**orphaned chunk**
	b-Inconsist	conn	**orphaned chunk**
	c-Network	failed	**metadata err**
OSS#1	a-DevFail	conn	**fetch err**
	b-Inconsist	conn	comp
	c-Network	failed	**metadata err**
three OSSes	a-DevFail	conn	**fetch err**
	b-Inconsist	conn	**wrong attributes**
	c-Network	failed	**metadata err**
MDS + OSS#1	a-DevFail	conn	**fetch err**
	b-Inconsist	conn	**orphaned chunk**
	c-Network	failed	**metadata err**

that we do not miss any valuable error messages, 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. In most cases there are no differences, except for minor updates such as the counts of execution and the running time of LFSCK. Therefore, we believe the characterization of LFSCK logs is accurate.

5.3.2 ***BeeGFS-FSCK Logs.*** Unlike LFSCK which generates logs in a distributed manner (i.e., on all MDS and OSS nodes), we find that BeeGFS-FSCK centralizes its logs on the client node. We characterizes BeeGFS-FSCK's logs in Table 10.

Specifically, we find that the BeeGFS-FSCK logs are grouped in two separate files on the client node. The first file stores the status of BeeGFS-FSCK, which is relatively simple and only includes one of three states: "conn", "wait" and "failed" (i.e., the "Status Logs (**.log*)" column). This set of status logs is roughly equivalent to LFSCK's status logs.

The second file stores BeeGFS-FSCK's checking results (the "Checking Logs (**.out*)" column), which are relatively more informative. For example, when "b-Inconsist" happens on MDS (the "MDS" and "MDS+OST#2" rows), BeeGFS-FSCK reports a message "finding a data chunk without an inode pointing to it" ("**orphaned chunk**"), which correctly implies that BeeGFS is in an inconsistent state after the fault. However, based on the logs in Table 5, BeeGFS-FSCK is unable to fix the inconsistency (i.e., `WikiR` fails with an "NoFile" error when reading files from the client, and `WikiW-async` and `WikiW-sync` fail with an "I/O err" in Table 5).

Also, it is interesting to see that BeeGFS-FSCK treats the device failure on MDS ("a-DevFail") and metadata inconsistency ("b-Inconsist") in the same way (i.e., both report "**orphaned chunk**"). This may lead to confusion for pinpointing the root cause because the report is the same for different faults. In other words, more fine-grained checking or logging mechanisms may be needed.

When "a-DevFail" happens on OSS (the "OSS#1" and "three OSSes" rows), BeeGFS-FSCK reports "errors occurred while fetching data from servers" ("**fetch err**"). This is reasonable because the data on OSSes become unaccessible under the fault model.

When "b-Inconsist" occurs on three OSSes, BeeGFS-FSCK may report that the attributes of file inode are wrong ("**wrong attributes**"), which suggests that BeeGFS-FSCK can detect the inconsistency. This behavior is much accurate and useful compared with that of LFSCK under the same scenario.

When "c-Network" happens on MDS (the "MDS" and "MDS+OSS#1" rows), BeeGFS-FSCK reports an error message"communication with metadata node failed" ("**metadata err**"). This is reasonable because MDS is not accessible under the fault model. However, when "c-Network" is applied to OSSes (the "OSS#1" and "three OSSes" rows), BeeGFS-FSCK still reports the same message, which may be misleading as OSS nodes are responsible for storing the user data.

In summary, we find that BeeGFS-FSCK is able to detect a number of subtle inconsistencies in BeeGFS after faults (e.g., "**orphaned chunk**", "**wrong attributes**"). Compared with LFSCK, BeeGFS-FSCK can report relatively more detailed information for diagnosis. However, in some cases the error messages are still sub-optimal, which suggests opportunities for further optimization.

6 LESSONS LEARNED AND FUTURE WORK

We have presented a comprehensive study on Lustre and BeeGFS, which has revealed their unique failure handling and logging patterns and has led to actual enhancements of PFS. Besides the specific contributions, this study has a number of general implications and suggests many opportunities for further improvements. We highlight a number of general lessons learned and discuss a few promising directions in this section.

6.1 Implications on Analyzing the Failure Handling Mechanisms of PFSes

In this study, we focus on the failure handling mechanisms of PFSes, which is mainly inspired by two sources: (1) The real-world failure incidents causing downtime and data loss at HPC centers [17–20]; (2) The abundant research efforts exposing the failure handling issues of local and cloud storage systems [4, 8–12, 21–23, 61, 76, 77]. By looking into the unique architecture of major PFSes, we identify the gap between the requirements of testing PFSes and the state-of-the-art methods as elaborated in §2. In order to bridge the gap, We find that we have to sacrifice many sophisticated designs proposed in the literature (e.g., protocol-aware methods) due to the complexity and the opaque nature of PFSes. Therefore, the current PFault prototype follows a black-box principle [28] to achieve the usability, generality, and fidelity as described in §3. The fact that this work has helped improve the leading PFS suggests that the methodology is effective in filling the void and bridging the gap in practice.

However, the black-box approach is not perfect. In particular, we find that it is fundamentally limited in terms of *diagnosing* the abnormal symptoms observed in PFSes. In this study, we have to manually investigate the substantial logs generated during the experiments and the associated PFS code base and documentation to understand the root causes, which is time consuming and not scalable for complicated large-scale systems like PFSes.

As a tradeoff to the black-box approach, a grey-box or white-box approach [28] may leverage the knowledge of the internal logic of the target program to collect feedback (e.g., code coverage) and/or guide the generation of test inputs, which may improve the test efficiency as well as the diagnosis of target systems. To be effective, such approaches typically require well-documented internal specifications, strong tool support for code analysis or instrumentation (e.g., AspectJ [78] for Java programs), etc., which remains challenging in the context of production PFSes with substantial weakly-typed code (e.g., C) in the kernel space. Therefore, despite the limitation of the black-box

principle, we believe that our work is one fundamental step towards more sophisticated analysis for PFSes in practice, and we hope that the extensive results collected in this work will facilitate follow-up exploration of grey-box/white-box approaches for analyzing PFSes in the communities.

6.2 Integration with Other Tools

The prototype of PFault is designed and implemented in a modular and extendable manner. For example, PFault invokes the local file systems utilities (e.g., `debugfs` [51]) to manipulate the states of individual PFS nodes (§3.2.2), which may be replaced with other tools (e.g., customized fault injectors) to achieve additional functionalities. Also, the modules in PFault may be integrated into other workflows in a standalone manner beyond what is presented in this work (e.g., manipulating the configurations of PFS nodes and collecting regular logs for performance tuning). As one concrete example, we elaborate on the research opportunity of integration with fuzzing tools, which are gaining significant traction recently [7, 79–85], in this subsection.

Fuzzing is a classic technique for generating effective inputs and improving the test coverage [79]. Since 1990s [86], fuzzing has been applied to study a wide range of programs [7, 79–83, 86]. In particular, a number of fuzzing tools (i.e., fuzzers) have been proposed for practical systems including file systems and OS kernels in recent years. For instance, Janus [80] uses two-dimensional fuzzing which mutates both on-disk metadata and system calls to expose bugs in local file systems. Similarly, Hydra [7] analyzes semantic bugs in local file systems through fuzzing. However, these existing fuzzers can only handle a local file system on a single node instead of distributed PFSes.

A few researchers have tried to fuzz networked software systems. For example, Raft consensus protocol has been fuzzed [83] through manipulation on RPC messages in a black-box manner without feedback loop or code coverage measurement. Similarly, AFLNet [87] is a grey-box fuzzer for network protocols used by servers. In this work, the vanilla AFL is expanded by network communication over C Socket APIs [84], which allows the fuzzer to act as a client and enables remote fuzzing. However, the fuzzer can only mutate the sequence of messages sent from client to server, the input space of which is much smaller compared to the distributed storage state needed to fuzz a PFS effectively [85].

Therefore, applying fuzzing to PFSes remains challenging. Multiple innovations are likely needed for the integration, including reducing the size of the initial seed pool, identifying critical components for instrumentation and collecting execution feedback, among others. One potential technique we are exploring is the in-memory API fuzzing on a single function [79], which focuses only a portion of the target program and thus might reduce the complexity. We leave such integration as future work.

6.3 Analyzing Hardware-Dependent Features of PFS Clusters

In this work, we focus on studying the failure recovery and logging mechanisms of PFS from the software perspective (e.g., the FSCK component and the logging methods). As mentioned in §4, to ensure the reproducibility and consistency of our results, we have tried a variety of different configurations with the resources available to us, including virtual and physical servers, private and public platforms, PFS node counts, stripe counts, stripe sizes, iSCSI/non-iSCSI, FSCK delay, etc., which have helped identify and fix real problems confirmed by PFS developers. On the other hand, modern PFS clusters may include additional advanced features that require special hardware support. For example, Lustre may be configured with a failover feature when the MDS nodes are equipped with the remote power control (RPC) mechanism, which requires both hardware support (e.g., IPMI/BMI device for power control) and external power management software support (e.g., PowerMan, Corosync, Pacemaker) [25]. The failover pair shares the same storage device and provides server process failover. Similarly, BeeGFS has an advanced feature called buddy mirroring

with additional failover capability. Such advanced features are designed to improve the failure handling mechanisms of PFSes and to provide additional reliability and/or availability guarantees for PFSes. Based on our understanding, however, these mechanisms might not be able to handle all the failure scenarios considered in this study. For example, the process failover mechanism in Lustre is designed to provide redundancy at the process level while still sharing the physical device; consequently, the a-DevFail fault model may still affect the Lustre cluster. Due to the limitation of our current hardware platform, we leave the study of such hardware-dependent features as future work.

6.4 Improving the Failure Handling Mechanisms of PFSes

We have exposed a number of limitations of PFSes in terms of failure recovery and logging in this study, especially on the FSCK component. We may improve the corresponding mechanisms of Lustre and BeeGFS based on the study results. For example, we find that Lustre logs can often capture the correct fault types (e.g., network connection fails), which implies that it is possible to detect the problem and avoid the abnormal behavior during LFSCK (e.g., "hang"). Similarly, it is possible to eliminate the abrupt "I/O err" by verifying the existence of the device before accessing. Along the same direction, one recent work studies the recovery rules of LFSCK in details and proposes to improve the completeness of LFSCK accordingly [88]. In addition, PFault may be applied to study and improve other important PFSes (e.g., OrangeFS, Ceph). Since PFault is designed with usability and portability in mind, we expect the porting efforts to be minimal.

Also, we find that the extensive logs generated by PFSes including their FSCK components are valuable for understanding the behaviors and diagnosing the root causes. However, as detailed in §5, in many cases the log information may be incomplete or misleading, which suggests opportunities for refining the logging mechanisms. In fact, the patch set created by the developers to fix the crash problem exposed by our study (§5.2.2) is also related to the internal logging macros of Lustre (e.g., CERROR, LASSERT). Given the complex code base of PFSes, manually refactoring the logging code is unlikely to be effective or scalable. Instead, automatic logging support or enhancements (e.g., LogEnhancer [89]) are likely needed to address the challenge, which we leave as future work.

6.5 Challenges and Opportunities for Log-Based Analysis

The extensive experimental logs generated in our study include both normal and abnormal cases, and the PFault tool may be applied to other PFSes to generate additional failure logs. Given the large quantity of the logs, we believe our work provides a valuable vehicle for applying learning-based log analysis to optimize PFSes, which has proved to be promising for failure detection and diagnosis of other large-scale systems (e.g., DeepLog [90]). In fact, `SentiLog`, which leverages PFault and the associated logs, is one recent effort along this direction [91]. On the other hand, we find that PFS logs are much more diverse and irregular than typical cloud systems logs, which makes many existing log-based analysis methodologies (e.g., CrashTuner [10]) largely inapplicable for PFSes. More sophisticated techniques in terms of log parsing and feature extraction [92] are likely needed to handle the PFS logs effectively. We release the prototype of PFault as well as the experimental logs publicly on GitLab to facilitate the follow-up research in the communities [93].

7 RELATED WORK

In this section, we discuss related work that has not been covered sufficiently in the previous sections.

Tools and Studies of Parallel File Systems. Due to the prime importance of PFSes, many analysis tools have been proposed by the HPC community to improve them [94–98]. For example, there is

a variety of tools for instrumentation, profiling, and tracing of I/O activities, such as mpiP [95], LANL-Trace [96], HPCT-IO [99], IOT [97], and TRACE [98] and so on. On the one hand, since these tools are mostly designed for studying and improving the performance of PFSes, they cannot emulate external failure events for studying the failure handling of PFSes as in PFault. One the other hand, we believe that these tools may also help in reliability. For example, Darshan [100, 101] 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. More recently, Sun et.al. [102] proposes to study the crash consistency of PFSes via replaying workload traces, which may benefit from the extensive real logs collected via PFault; also, SentiLog [91] applies sentimental analysis to detect PFS anomalies based on the logs generated by PFault. Therefore, PFault and the existing PFS efforts are complementary.

Tools and Studies of Other Distributed Systems. Many tools have been proposed for analyzing distributed systems (e.g., [11, 12, 21–23, 42–47]), especially for modern Java-based cloud systems (e.g., HDFS [13], Cassandra [14], Yarn [49], ZooKeeper [15]). While they are effective for their original goals, few of them have been or can be directly applied to study PFSes in practice due to one or more constraints. For examples, they may (1) only work for user-level programs, instead of PFSes containing OS kernel modules and patches; (2) require modifications to the local storage stack that are incompatible to major PFSes; (3) rely on Java-specific features/tools that are not applicable to major PFSes; (4) rely on unified and well-formed logging mechanisms (e.g., Log4J [32]) that are not available on major PFSes; (5) rely on detailed specifications of internal protocols of the target system, which are difficult to derive for PFSes due to the complexity and the lack of documentation. We discuss a number of representative works in more details below.

As far as we know, the most relevant work is CORDS [21], where the researchers customize a FUSE file system to analyze eight user-level distributed storage systems and find that none of them can consistently use redundancy to recovery from faults. They inject two types of local corruptions (i.e., zeros or junk on a single file-system block), which is similar to the global inconsistency fault model emulated by PFault. On the other hand, the FUSE-based approach is not applicable to PFSes which often have special requirements on the OS kernel and/or local file system features (i.e., Lustre requires a patched version of Ext4 or ZFS).

MOLLY [103] proposes lineage-driven fault injection (LDFI) for discovering bugs in fault-tolerant protocols of distributed systems. By rewriting protocols in a declarative language (i.e., Dedalus) and leveraging a SAT solver, MOLLY can effectively provide correctness and coverage guarantee for the protocols under test. However, applying LDFI to study PFSes remains challenging. Among others, rewriting production PFS or FSCK in a declarative language is prohibitively expensive in practice. Moreover, PFSes do not maintain redundant replica at the PFS level, nor do they use well-specified protocols for recovery. As a result, it is difficult to derive the execution model or correctness properties of PFS required by LDFI. On the other hand, the high-level idea of leveraging data lineage to connect system outcomes to the data and messages that led to them could potentially help analyze the root causes of the abnormal symptoms observed in our study.

SAMC [9] applies semantic aware model checking to study seven protocols used by Cassandra, Yarn, and ZooKeeper. Different from the black-box approach taken by PFault, SAMC uses a white-box approach to incorporate semantic information (e.g., local message independence) of the target system in its state-space reduction policies. While effective in exposing deep bugs in cloud systems, SAMC depends on detailed specifications of distributed fault-tolerance protocols, which are not applicable to PFS and FSCK. Moreover, it requires modifying target systems using AspectJ [78],

which is not applicable to major PFSes. In contrast, PFault focuses on emulating general external failure events for PFSes via a black-box transparent approach, trading off fine-grained control for usability. We leave the potential integration of model checking with PFSes as future work.

ScaleCheck [48] focuses on testing scalability bugs in distributed systems. It leverages Java language supports (e.g., JVMTI [104] and Reflection [105]) to identify scale-dependent collections, and makes use of multiple novel co-location techniques (e.g., single-process cluster using the isolation support of Java class loader) to make target system single-machine scale-testable. PFault is similar to ScaleCheck in the sense that both aim to make large distributed systems easier to analyze with less physical resource constraints; on the other hand, the Java-specific techniques are unlikely to be directly applicable to study PFSes which are mostly written in type-unsafe languages.

More recently, CrashTuner [10] proposes the concept of "meta-info" to locate fault injection points for detecting crash recovery bugs in distributed systems efficiently and effectively. The target system must be written in Java because the static analysis and instrumentation tools (i.e., WALA [106] and Javasist [107]) are Java-specific and rely on the strong type system of Java. While in theory there may be similar compiler tools for instrumenting PFSes written in C/C++ (e.g., LLVM [108]), implementing the same idea to study PFSes with OS kernel components would require substantial efforts (if possible at all). Moreover, the "meta-info" variables must be derived from well-formed logs (e.g., messages with clear `nodeID` and `taskID` information generated by Log4J [32] or SLF4J [75]), which is not applicable to PFSes because the log messages of PFSes are diverse and irregular as exposed in our study (§5.2.1 and §A). On the other hand, the extensive and complex logs collected in our experiments provide a valuable dataset for exploring potential implicit "meta-info" of PFSes via sophisticated learning-based approaches as discussed in §6.

In addition, many researchers have studied the failures occurred in large-scale production systems [20, 109–112], which provides valuable insights for emulating realistic failure events in PFault to trigger the failure recovery and logging operations of PFSes.

Tools and Studies of Local Storage Systems. Great efforts have been made to study the bugs or failure behaviors of local storage software and/or hardware (e.g., hard disks [56, 59], RAID [5], flash-based SSDs [76, 112, 113], persistent memories [114], local file systems and checkers [6, 7, 41, 77, 80, 115–119]) through a variety of approaches (e.g., fault injection [64, 77], model checking [115], formal methods [120], fuzzing [7, 80]). While the tools are effective for their original design goals, applying them to study large-scale PFSes remains challenging. For example, model checking still faces the state explosion problem despite of various path reduction optimizations [9]. Also, turning a practical system like Lustre into a precise or verifiable model is prohibitively expensive in terms of human efforts. On the other hand, these existing efforts provide valuable insights on the reliability of local storage systems, and they may help in emulating realistic failure states of individual storage nodes in PFault. Moreover, some techniques (e.g., fuzzing) could potentially be integrated with PFault as discussed in §6.

8 CONCLUSIONS

As the scale and complexity of PFSes keeps increasing, maintaining PFS consistency and data integrity becomes more and more challenging. Motivated by the real challenge, we perform a study of the failure recovery and logging mechanisms of PFSes in this paper. We apply the PFault tool to study two widely used PFSes: Lustre and BeeGFS. Through extensive log analysis and root cause diagnosis, our study has revealed the abnormal behaviors of the failure handling mechanisms in PFSes. Most importantly, our study has led to a new patch to address a kernel panic problem in the latest Lustre.

This research study is a critical step on our roadmap toward achieving robust high-performance computing. Given the prime importance of PFSes in HPC systems and data centers, this study also calls for community's collective efforts in examining reliability challenges and coming up with advanced and highly-efficient solutions. We hope this study can inspire more research efforts along this direction. We also believe that such a study, including the open-source PFault and the extensive PFS logs, can have a long-term impact on the design of large-scale file systems, storage systems, and HPC systems.

9 ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers and the TOS editors for their time and insightful feedback. The authors also thank Andreas Dilger and other PFS developers for valuable discussions. This work was supported in part by NSF under grants CCF-1717630/1853714, CCF-1910747, and CNS-1943204. Any opinions, findings, and conclusions expressed in this material are those of the authors and do not necessarily reflect the views of the sponsor.

REFERENCES

[1] Lustre File System. [Online]. Available: http://lustre.org/
[2] BeeGFS File System. [Online]. Available: https://www.beegfs.io/
[3] The OrangeFS Project, 2017. [Online]. Available: http://www.orangefs.org/
[4] M. Zheng, J. Tucek, F. Qin, and M. Lillibridge, "Understanding the robustness of SSDs under power fault," in *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST'13)*, 2013.
[5] A. Ma, F. Douglis, G. Lu, D. Sawyer, S. Chandra, and W. Hsu, "Raidshield: Characterizing, monitoring, and proactively protecting against disk failures," in *13th USENIX Conference on File and Storage Technologies (FAST 15)*. Santa Clara, CA: USENIX Association, Feb. 2015, pp. 241–256. [Online]. Available: https://www.usenix.org/conference/fast15/technical-sessions/presentation/ma
[6] C. Min, S. Kashyap, B. Lee, C. Song, and T. Kim, "Cross-checking semantic correctness: The case of finding file system bugs," in *Proceedings of the 25th Symposium on Operating Systems Principles*, ser. SOSP '15. New York, NY, USA: ACM, 2015, pp. 361–377. [Online]. Available: http://doi.acm.org/10.1145/2815400.2815422
[7] S. Kim, M. Xu, S. Kashyap, J. Yoon, W. Xu, and T. Kim, "Finding semantic bugs in file systems with an extensible fuzzing framework," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 147–161. [Online]. Available: https://doi.org/10.1145/3341301.3359662
[8] T. S. Pillai, V. Chidambaram, R. Alagappan, S. Al-Kiswany, A. C. Arpaci-Dusseau, and R. 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)*, October 2014.
[9] T. Leesatapornwongsa, M. Hao, P. Joshi, J. F. Lukman, and H. S. Gunawi, "Samc: Semantic-aware model checking for fast discovery of deep bugs in cloud systems," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, Oct. 2014, pp. 399–414. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/leesatapornwongsa
[10] J. Lu, C. Liu, L. Li, X. Feng, F. Tan, J. Yang, and L. You, "Crashtuner: Detecting crash-recovery bugs in cloud systems via meta-info analysis," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP '19. New York, NY, USA: ACM, 2019, pp. 114–130. [Online]. Available: http://doi.acm.org/10.1145/3341301.3359645
[11] P. Joshi, H. S. Gunawi, and K. Sen, "PREFAIL: A programmable tool for multiple-failure injection," in *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, 2011, pp. 171–188.
[12] P. Joshi, M. Ganai, G. Balakrishnan, A. Gupta, and N. Papakonstantinou, "Setsudō: perturbation-based testing framework for scalable distributed systems," in *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems*, 2013, pp. 1–14.
[13] Hadoop Distributed File System, 2006-now. [Online]. Available: https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html
[14] Apache Cassandra, 2008-now. [Online]. Available: https://cassandra.apache.org
[15] Apache Zookeeper, Accessed January 2021. [Online]. Available: https://zookeeper.apache.org
[16] High Performance Computing Center, Texas Tech University, 2017. [Online]. Available: http://www.depts.ttu.edu/hpcc/

[17] Power Outage Event at High Performance Computing Center (HPCC) in Texas , 2016. [Online]. Available: https://www.ece.iastate.edu/~mai/docs/failures/2016-hpcc-lustre.pdf
[18] GPFS Failures at Ohio Supercomputer Center (OSC), 2016. [Online]. Available: https://www.ece.iastate.edu/~mai/docs/failures/2016-hpcc-lustre.pdf
[19] Multiple Switch Outages at Ohio Supercomputer Center (OSC), 2016. [Online]. Available: https://www.ece.iastate.edu/~mai/docs/failures/2016-hpcc-lustre.pdf
[20] H. S. Gunawi, R. O. Suminto, R. Sears, C. Golliher, S. Sundararaman, X. Lin, T. Emami, W. Sheng, N. Bidokhti, C. McCaffrey, G. Grider, P. M. Fields, K. Harms, R. B. Ross, A. Jacobson, R. Ricci, K. Webb, P. Alvaro, H. B. Runesha, M. Hao, and H. Li, "Fail-Slow at Scale: Evidence of Hardware Performance Faults in Large Production Systems," in *16th USENIX Conference on File and Storage Technologies (FAST 18)*, 2018.
[21] A. Ganesan, R. Alagappan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Redundancy does not imply fault tolerance: Analysis of distributed storage reactions to single errors and corruptions." in *FAST*, 2017, pp. 149–166.
[22] H. S. Gunawi, T. Do, P. Joshi, P. Alvaro, J. M. Hellerstein, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, K. Sen, and D. Borthakur, "FATE and DESTINI: A Framework for Cloud Recovery Testing," ser. NSDI'11, 2011.
[23] R. Alagappan, A. Ganesan, E. Lee, A. Albarghouthi, V. Chidambaram, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Protocol-aware recovery for consensus-based storage," in *16th {USENIX} Conference on File and Storage Technologies ({FAST} 18)*, 2018, pp. 15–32.
[24] Open MPI, 2004-now. [Online]. Available: https://www.open-mpi.org
[25] Lustre Software Release 2.x: Operations Manual, 2017. [Online]. Available: http://lustre.org/documentation/
[26] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *ACM SIGOPS operating systems review*, vol. 37, no. 5. ACM, 2003, pp. 29–43.
[27] M. Xia, M. Saxena, M. Blaum, and D. A. Pease, "A tale of two erasure codes in HDFS," in *13th USENIX Conference on File and Storage Technologies (FAST 15)*, 2015, pp. 213–226.
[28] G. J. Myers, T. Badgett, T. M. Thomas, and C. Sandler, *The art of software testing*. Wiley, 2004, vol. 2.
[29] L. S. target framework (tgt), 2017. [Online]. Available: http://stgt.sourceforge.net/
[30] N. E. over Fabrics Specification Released, 2017. [Online]. Available: http://www.nvmexpress.org/nvm-express-over-fabrics-specification-released/
[31] LFSCK: an online file system checker for Lustre, 2017. [Online]. Available: https://github.com/Xyratex/lustre-stable/blob/master/Documentation/lfsck.txt
[32] Apache log4j, a logging library for Java, 2001-now. [Online]. Available: http://logging.apache.org/log4j/2.x/
[33] J. Cao, O. R. Gatla, M. Zheng, D. Dai, V. Eswarappa, Y. Mu, and Y. Chen, "PFault: A General Framework for Analyzing the Reliability of High-Performance Parallel File Systems," in *Proceedings of the 2018 International Conference on Supercomputing (ICS)*, 2018.
[34] Lustre Patch: LU-13980 osd: remove osd_object_release LASSERT, 2020. [Online]. Available: https://review.whamcloud.com/#/c/40058/
[35] D. A. Patterson, G. Gibson, and R. H. Katz, *A case for redundant arrays of inexpensive disks (RAID)*. ACM, 1988, vol. 17, no. 3.
[36] HPC User Site Census, 2016. [Online]. Available: http://www.intersect360.com/
[37] Top500 Supercomputers, 2019. [Online]. Available: https://www.top500.org/lists/2016/11/
[38] Apache HBase, 2020. [Online]. Available: https://hbase.apache.org
[39] BeeGFS Documentation v7.2, 2020. [Online]. Available: https://doc.beegfs.io/latest/overview/overview.html
[40] S. documents, 2017.
[41] H. S. Gunawi, A. Rajimwale, A. C. Arpaci-dusseau, and R. H. Arpaci-dusseau, "SQCK: A Declarative File System Checker," in *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.
[42] S. Dawson, F. Jahanian, and T. Mitton, "Orchestra: a probing and fault injection environment for testing protocol implementations," in *Proceedings of IEEE International Computer Performance and Dependability Symposium*, 1996, pp. 56–.
[43] Seungjae Han, K. G. Shin, and H. A. Rosenberg, "Doctor: an integrated software fault injection environment for distributed real-time systems," in *Proceedings of 1995 IEEE International Computer Performance and Dependability Symposium*, 1995, pp. 204–213.
[44] D. T. Stott, B. Floering, D. Burke, Z. Kalbarczpk, and R. K. Iyer, "Nftape: a framework for assessing dependability in distributed systems with lightweight fault injectors," in *Proceedings IEEE International Computer Performance and Dependability Symposium. IPDS 2000*, 2000, pp. 91–100.
[45] J. H. Barton, E. W. Czeck, Z. Z. Segall, and D. P. Siewiorek, "Fault injection experiments using fiat," *IEEE Transactions on Computers*, vol. 39, no. 4, pp. 575–582, 1990.
[46] Jepsen. [Online]. Available: https://github.com/jepsen-io/jepsen

[47] X. Yuan and J. Yang, "Effective concurrency testing for distributed systems," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1141–1156. [Online]. Available: https://doi.org/10.1145/3373376.3378484

[48] C. A. Stuardo, T. Leesatapornwongsa, R. O. Suminto, H. Ke, J. F. Lukman, W.-C. Chuang, S. Lu, and H. S. Gunawi, "Scalecheck: A single-machine approach for discovering scalability bugs in large distributed systems," in *17th USENIX Conference on File and Storage Technologies (FAST 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 359–373. [Online]. Available: https://www.usenix.org/conference/fast19/presentation/stuardo

[49] Apache Hadoop YARN, 2020. [Online]. Available: https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html

[50] Apache Hadoop, 2019. [Online]. Available: https://hadoop.apache.org/docs/stable/

[51] E2fsprogs: Ext2/3/4 Filesystems Utilities, 2017. [Online]. Available: http://e2fsprogs.sourceforge.net

[52] D. Dai, O. R. Gatla, and M. Zheng, "A Performance Study of Lustre File System Checker: Bottlenecks and Potentials," in *2019 35th Symposium on Mass Storage Systems and Technologies (MSST)*, 2019.

[53] FUSE. Linux FUSE (Filesystem in Userspace) interface. [Online]. Available: https://github.com/libfuse/libfuse

[54] R. Sandberg, D. Golgberg, S. Kleiman, D. Walsh, and B. Lyon, "Innovations in internetworking," C. Partridge, Ed. Norwood, MA, USA: Artech House, Inc., 1988, ch. Design and Implementation of the Sun Network Filesystem, pp. 379–390. [Online]. Available: http://dl.acm.org/citation.cfm?id=59309.59338

[55] M. Primmer, "An Introduction to Fibre Channel," *HP Journal*, 1996.

[56] L. N. Bairavasundaram, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, G. R. Goodson, and B. Schroeder, "An analysis of data corruption in the storage stack," *Trans. Storage*, vol. 4, no. 3, pp. 8:1–8:28, Nov. 2008. [Online]. Available: http://doi.acm.org/10.1145/1416944.1416947

[57] L. N. Bairavasundaram, G. R. Goodson, S. Pasupathy, and J. Schindler, "An analysis of latent sector errors in disk drives," in *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '07. New York, NY, USA: ACM, 2007, pp. 289–300. [Online]. Available: http://doi.acm.org/10.1145/1254882.1254917

[58] E. B. Nightingale, J. R. Douceur, and V. Orgovan, "Cycles, cells and platters: An empirical analysis of hardware failures on a million consumer PCs," in *Proceedings of the Sixth Conference on Computer Systems*, ser. EuroSys '11. New York, NY, USA: ACM, 2011, pp. 343–356.

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

[60] S. Subramanian, Y. Zhang, R. Vaidyanathan, H. S. Gunawi, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and J. F. Naughton, "Impact of disk corruption on open-source DBMS," in *ICDE*, 2010, pp. 509–520.

[61] M. Zheng, J. Tucek, D. Huang, F. Qin, M. Lillibridge, E. S. Yang, B. W. Zhao, and S. Singh, "Torturing databases for fun and profit," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, 2014, pp. 449–464. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/zheng_mai

[62] Network Partition, 2017. [Online]. Available: https://www.cs.cornell.edu/courses/cs614/2003sp/papers/DGS85.pdf

[63] e2fsck(8) — Linux manual page, 2017. [Online]. Available: https://man7.org/linux/man-pages/man8/e2fsck.8.html

[64] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "IRON File Systems," in *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP'05)*, Brighton, United Kingdom, October 2005, pp. 206–220.

[65] debugfs, 2017. [Online]. Available: http://man7.org/linux/man-pages/man8/debugfs.8.html

[66] A. Alquraan, H. Takruri, M. Alfatafta, and S. Al-Kiswany, "An analysis of network-partitioning failures in cloud systems," in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'18. USA: USENIX Association, 2018, p. 51–68.

[67] P. Gill, N. Jain, and N. Nagappan, "Understanding network failures in data centers: Measurement, analysis, and implications," *SIGCOMM Comput. Commun. Rev.*, vol. 41, no. 4, p. 350–361, Aug. 2011. [Online]. Available: https://doi.org/10.1145/2043164.2018477

[68] K. A. Smith and M. I. Seltzer, "File system aging—increasing the relevance of file system benchmarks," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 25, no. 1. ACM, 1997, pp. 203–213.

[69] A. Conway, A. Bakshi, Y. Jiao, W. Jannen, Y. Zhan, J. Yuan, M. A. Bender, R. Johnson, B. C. Kuszmaul, D. E. Porter, and M. Farach-Colton, "File systems fated for senescence? nonsense, says science!" in *15th USENIX Conference on File and Storage Technologies (FAST 17)*. Santa Clara, CA: USENIX Association, 2017, pp. 45–58. [Online]. Available: https://www.usenix.org/conference/fast17/technical-sessions/presentation/conway

[70] CloudLab. [Online]. Available: http://cloudlab.us/

[71] Montage: An Astronomical Image Mosaic Engine, 2017. [Online]. Available: http://montage.ipac.caltech.edu/

[72] Wikipedia:Database download, 2017. [Online]. Available: https://en.wikipedia.org/wiki/Wikipedia:Database_download

[73] J. Cao, S. Wang, D. Dai, M. Zheng, and Y. Chen, "A generic framework for testing parallel file systems," in *Proceedings of the 1st Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems (PDSW-DISCS)*, 2016.

[74] J. a. C. M. Carreira, R. Rodrigues, G. Candea, and R. Majumdar, "Scalable testing of file system checkers," in *Proceedings of the 7th ACM European Conference on Computer Systems*, ser. EuroSys '12, 2012, p. 239–252.

[75] Simple logging facade for Java, 2019. [Online]. Available: http://www.slf4j.org

[76] M. Zheng, J. Tucek, F. Qin, M. Lillibridge, B. W. Zhao, and E. S. Yang, "Reliability analysis of ssds under power fault," in *To appear in the ACM Transactions on Computer Systems (TOCS)*, 2016. [Online]. Available: http://dx.doi.org/10.1145/2992782

[77] O. R. Gatla, M. Hameed, M. Zheng, V. Dubeyko, A. Manzanares, F. Blagojević, C. Guyot, and R. Mateescu, "Towards robust file system checkers," in *16th USENIX Conference on File and Storage Technologies (FAST)*. USENIX Association, Feb. 2018.

[78] AspectJ, 2001-now. [Online]. Available: https://www.eclipse.org/aspectj/

[79] V. J. M. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The Art, Science, and Engineering of Fuzzing: A Survey," *IEEE Transactions on Software Engineering*, 2019.

[80] W. Xu, H. Moon, S. Kashyap, P. Tseng, and T. Kim, "Fuzzing file systems via two-dimensional input space exploration," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 818–834.

[81] M. Xu, S. Kashyap, H. Zhao, and T. Kim, "Krace: Data Race Fuzzing for Kernel File Systems," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020.

[82] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin, "Razzer: Finding kernel race bugs through fuzzing," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019.

[83] C. Scott, "Fuzzing Raft for Fun and Publication," 2015. [Online]. Available: https://colin-scott.github.io/blog/2015/10/07/fuzzing-raft-for-fun-and-profit/

[84] socket(2) — Linux manual page, 2020. [Online]. Available: https://man7.org/linux/man-pages/man2/socket.2.html

[85] R. Banabic, G. Candea, and R. Guerraoui, "Automated vulnerability discovery in distributed systems," in *2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W)*, 2011, pp. 188–193.

[86] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Commun. ACM*, vol. 33, no. 12, 1990. [Online]. Available: https://doi.org/10.1145/96267.96279

[87] V. T. Pham, M. Böhme, and A. Roychoudhury, "Aflnet: A greybox fuzzer for network protocols," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, 2020, pp. 460–465.

[88] R. Han, D. Zhang, and M. Zheng, "Fingerprinting the Checker Policies of Parallel File Systems," in *Proceedings of the 5th International Parallel Data Systems Workshop (PDSW) held in conjunction with IEEE/ACM SC20: The International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020.

[89] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage, "Improving software diagnosability via log enhancement," *ACM Transactions on Computer Systems (TOCS)*, 2012.

[90] M. Du, F. Li, G. Zheng, and V. Srikumar, "DeepLog: Anomaly Detection and Diagnosis from System Logs through Deep Learning," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS'17)*, 2017.

[91] D. Zhang, D. Dai, R. Han, and M. Zheng, "SentiLog: Anomaly Detecting on Parallel File Systems via Log-based Sentiment Analysis," in *Proceedings of the 13th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2021.

[92] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP)*, 2009.

[93] GitLab repository for PFault by Data Storage Lab@ISU , 2020. [Online]. Available: https://git.ece.iastate.edu/data-storage-lab/prototypes/pfault

[94] D. Huang, X. Zhang, W. Shi, M. Zheng, S. Jiang, and F. Qin, "Liu: Hiding disk access latency for hpc applications with a new ssd-enabled data layout," in *2013 IEEE 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*, 2013, pp. 111–120.

[95] J. S. Vetter and M. O. McCracken, "Statistical Scalability Analysis of Communication Operations in Distributed Applications," in *ACM SIGPLAN Notices*, vol. 36, no. 7. ACM, 2001, pp. 123–132.

[96] HPC-5 Open Source Software project, LANL-Trace, 2015. [Online]. Available: institutes.lanl.gov/data/tdata/

[97] P. C. Roth, "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, 2007, pp. 50–55.

[98] M. P. Mesnier, M. Wachs, R. R. Simbasivan, J. Lopez, J. Hendricks, G. R. Ganger, and D. R. O'hallaron, "//trace: Parallel trace replay with approximate causal events." USENIX, 2007.

[99] S. Seelam, I. Chung, D.-Y. Hong, H.-F. Wen, H. Yu *et al.*, "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, 2008, pp. 1–8.
[100] Darshan:HPC I/O Characterization Tool, 2017. [Online]. Available: http://www.mcs.anl.gov/research/projects/darshan/
[101] P. Carns, R. Latham, R. Ross, K. Iskra, S. Lang, and K. Riley, "24/7 Characterization of Petascale I/O Workloads," in *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on.* IEEE, 2009, pp. 1–10.
[102] J. Sun, C. Wang, J. Huang, and M. Snir, "Understanding and Finding Crash-Consistency Bugs in Parallel File Systems," in *Proceedings of the 12th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2020.
[103] P. Alvaro, J. Rosen, and J. M. Hellerstein, "Lineage-driven fault injection," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 2015, pp. 331–346.
[104] Java Virtual Machine Tool Interface (JVM TI). [Online]. Available: https://docs.oracle.com/javase/8/docs/technotes/guides/jvmti/
[105] Trail: The Reflection API. [Online]. Available: https://docs.oracle.com/javase/tutorial/reflect/index.html
[106] "WALA Home page," 2015. [Online]. Available: http://wala.sourceforge.net/wiki/index.php/
[107] Java bytecode engineering toolkit, 1999. [Online]. Available: https://www.javassist.org/
[108] The LLVM Compiler Infrastructure, 2020. [Online]. Available: https://llvm.org
[109] E. Xu, M. Zheng, F. Qin, Y. Xu, and J. Wu, "Lessons and actions: What we learned from 10k ssd-related storage system failures," in *2019 USENIX Annual Technical Conference (USENIX ATC 19).* Renton, WA: USENIX Association, Jul. 2019, pp. 961–976. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/xu
[110] E. Xu, M. Zheng, F. Qin, J. Wu, and Y. Xu, "Understanding ssd reliability in large-scale cloud systems," in *2018 IEEE/ACM 3rd International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems (PDSW-DISCS)*, 2018.
[111] H. S. Gunawi, M. Hao, R. O. Suminto, A. Laksono, A. D. Satria, J. Adityatama, and K. J. Eliazar, "Why does the cloud stop computing? lessons from hundreds of service outages," in *Proceedings of the Seventh ACM Symposium on Cloud Computing*, ser. SoCC '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 1–16. [Online]. Available: https://doi.org/10.1145/2987550.2987583
[112] A. M. Bianca Schroeder, Raghav Lagisetty, "Flash reliability in production: The expected and the unexpected," in *14th USENIX Conference on File and Storage Technologies (FAST 16).* Santa Clara, CA: USENIX Association, Feb. 2016, pp. 67–80. [Online]. Available: https://www.usenix.org/conference/fast16/technical-sessions/presentation/schroeder
[113] M. Zheng, J. Tucek, F. Qin, and M. Lillibridge, "Understanding the robustness of ssds under power fault," in *Proceedings of 11th USENIX Conference on File and Storage Technologies (FAST)*, 2013, pp. 271–284.
[114] D. Zhang, O. R. Gatla, W. Xu, and M. Zheng, "A study of persistent memory bugs in the linux kernel," in *Proceedings of the 14th ACM International Conference on Systems and Storage (SYSTOR)*, 2021.
[115] J. Yang, C. Sar, and D. Engler, "EXPLODE: a lightweight, general system for finding serious storage system errors," in *Proceedings of the Seventh Symposium on Operating Systems Design and Implementation (OSDI '06)*, November 2006, pp. 131–146.
[116] L. N. Bairavasundaram, S. Sundararaman, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Tolerating file-system mistakes with envyfs," in *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, ser. USENIX'09. Berkeley, CA, USA: USENIX Association, 2009, pp. 7–7. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855807.1855814
[117] L. Lu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and S. Lu, "A study of linux file system evolution," in *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13).* San Jose, CA: USENIX, 2013, pp. 31–44. [Online]. Available: https://www.usenix.org/conference/fast13/technical-sessions/presentation/lu
[118] O. R. Gatla, M. Zheng, M. Hameed, V. Dubeyko, A. Manzanares, F. Blagojevic, C. Guyot, and R. Mateescu, "Towards robust file system checkers," *ACM Transactions on Storage (TOS)*, vol. 14, no. 4, pp. 1–25, 2018.
[119] O. R. Gatla and M. Zheng, "Understanding the fault resilience of file system checkers," in *Proceedings of the 9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2017.
[120] H. Chen, D. Ziegler, T. Chajed, A. Chlipala, M. F. Kaashoek, and N. Zeldovich, "Using crash hoare logic for certifying the fscq file system," in *Proceedings of the 25th Symposium on Operating Systems Principles*, ser. SOSP '15. New York, NY, USA: ACM, 2015, pp. 18–37. [Online]. Available: http://doi.acm.org/10.1145/2815400.2815402

A APPENDIX: CHARACTERIZATION OF PFS FAILURE LOGS

In this appendix, we characterize the extensive failure logs generated by the target PFS in our experiments. As described in 5.2, we use three rules to identify the PFS logs related to failure handling and we call them as *error messages*. In total, we observe seven, thirteen, and fifteen different types of error messages on Lustre v2.8, Lustre v2.10.8, and BeeGFS v7.1.3 respectively, which we describe in details below.

A.1 Failure Logs of Lustre v2.8

We first analyze the error messages of Lustre v2.8. As shown in Table 11, we observe seven types of error messages (i.e., *y1 to y7*) when faults are injected on different nodes, including *Recovery failed* (*y1-y3*), *Log updating failed* (*y4*), *Lock service failed* (*y5*), and *Failing over* (*y6*,*y7*). If an error message has a Linux error number, the number is usually appended to the end of the message. A minor logging inconsistency we observe is that Lustre debug macros use a variable "rc" to represent Linux error number and print out both "rc" and its value in most cases (e.g., "rc 0/0" in *y5*), while in some cases only the value is shown (e.g., "-110" in *y1*).

It is interesting to see that in v2.8, MGS dose not report any error messages under the three fault models (i.e., empty in the "Logs on MGS" column). This is consistent with Lustre's design that MGS/MGT is mostly used for configuration when building Lustre, instead of the core functionalities.

Table 11. **Characterization of Logs Generated in the Debug Buffer of Lustre v2.8 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 "y7" are seven 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.*

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

Type	Meaning	Message Example
y1	MGS Recovery failed	*...ptlrpc_connect_interpret() recovery of MGS on MGC 192.x.x.x...failed (-110)*
y2	MDS Recovery failed	*...ptlrpc_connect_interpret() recovery of lustre-MDT0000_UUID...failed (-110)*
y3	OSS Recovery failed	*...ptlrpc_connect_interpret() recovery of lustre-OST0001_UUID...failed (-110)*
y4	Log updating failed	*...updating log 2 succeed 1 fail [...lustre-sptlrpc(fail)...*
y5	Lock service failed	*...ldlm_request.c:1317: ldlm_cli_update_pool()...@Zero SLV or Limit found...rc 0/0*
y6	Failing over MDT	*...obd_config.c:652:class_cleanup() Failing over lustre-MDT0000...*
y7	Failing over OST	*...obd_config.c:652:class_cleanup() Failing over lustre-OST0001...*

On the other hand, all three fault models can trigger extensive log messages on MDS and OSS. For example, when "a-DevFail" happens on MDS (the "MDS" row), all OSS nodes can notice the failure, and they try to recover MDT but eventually fail (i.e., *y2*). 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., *y4*) and locking (i.e., *y5*) 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., *y6*, *y7*). Because a complete failover configuration on Lustre requires additional sophisticated software and hardware support [25], 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 [25], 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 §3.2), and does not kill the MDS/OSS sever processes, it is unclear how failing over server processes could handle the device failure.

A.2 Failure Logs of Lustre v2.10.8

Besides Lustre v2.8, we have also studied the logs of Lustre v2.10.8 under the same experiments, and summarized them in Table 12. Note that we have discussed LFSCK-specific debug buffer logs of Lustre v2.10.8 in Table 9 (§5.3), so we skip them in here.

As shown in Table 12, the first seven types of error messages (i.e., *y1*-*y7*) are almost the same as the corresponding messages in Table 11. Message *y4* has a slightly different wording, but it is still related to Lustre's logging service.

On the other hand, we observe more types of error messages on v2.10.8 (i.e., *y8* – *y12* in Table 12) compared to v2.8 (in Table 11). Specifically, *y8* to *y11* (i.e., *Client's request failed*, *Client was evicted*, *Client-server connection failed*, *Client-OST I/O errors*) are client related failures; and *y12* represents failures of accessing metadata on OST.

Besides generating different types of error messages, another key difference between v2.10.8 (Table 12) and v2.8 (Table 11) is that MGS does report some information under faults in v2.10.8 (Table 12). In particular, under the "b-Inconsist" or "c-Network" fault models, MGS can report that the client's request has failed due to time out or network error (i.e., *y8*). This implies that MGS is aware of Lustre's internal traffic failures. Moreover, when both MDS and OSS#1 suffer from "a-DevFail" (the "MDS+OSS#1" row), MGS notifies that the client is evicted by Lustre's locking services (i.e., *y9*). In the meantime, MDS reports that the connection between client and servers fails (i.e., *y10*), and logs from OST#2 and OST#3 shows that they encounter errors when dealing with clients I/O requests (i.e., *y11*). This observation suggests that Lustre v2.10 has a more extensive logging to help understand system failures across nodes.

Also, we observe that *Local metadata unaccessible* (i.e., *y12*) can be triggered when "c-Network" happens on OSSes (the "OSS#1" and "three OSSes" rows), and it can only be collected from OSSes. This type of error message appears when OSS's local metadata becomes inaccessible. Most of their Linux error numbers are "-5", which means an I/O error occurs when Lustre tries to look up OSS's local metadata. Moreover, we find that *y12* often appears together with LSFCK-triggered error messages (Table 9 in §5.3). This is because LFSCK is responsible for checking and repairing the

Table 12. **Characterization of Logs Generated in the Debug Buffer of Lustre v2.10.8 After Faults.** *Similar to Table 11, this table shows detailed Debug Buffer logs from Lustre v2.10.8. the"Node(s) Affected" column shows the node(s) to which the faults are injected. "–" means no error message is reported. "y1" to "y12" are twelve 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.*

Node(s) Affected	Fault Models	Logs on MGS	Logs on MDS	Logs on OSS#1	Logs on OSS#2	Logs on OSS#3
MGS	a-DevFail	–	–	–	–	–
	b-Inconsist	*y8*	*y1,y4*	*y1,y4*	*y1,y4*	*y1,y4*
	c-Network	*y8*	*y1,y4*	*y1*	*y1*	*y1*
MDS	a-DevFail	–	*y2,y4*	*y2,y8*	*y2,y8*	*y2,y8*
	b-Inconsist	–	*y2,y4,y6*	*y2*	*y2*	*y2*
	c-Network	*y8*	*y1,y3,y8*	*y2,y8*	*y2,y8*	*y2,y8*
OSS#1	a-DevFail	–	–	*y12*	–	–
	b-Inconsist	*y8*	*y3,y5,y8*	*y4,y7*	*y4*	*y4*
	c-Network	*y8*	*y3,y8*	*y1,y2,y8*	–	–
three OSSes	a-DevFail	–	–	*y12*	*y12*	*y12*
	b-Inconsist	*y8*	*y3,y5,y8*	*y4,y7*	*y4,y7*	*y4,y7*
	c-Network	*y8*	*y3,y8*	*y1,y2,y8*	*y1,y2,y8*	*y1,y2,y8*
MDS + OSS#1	a-DevFail	*y8,y9*	*y2,y3,y4,y10*	*y2,y8*	*y2,y8,y11*	*y2,y8,y11*
	b-Inconsist	*y8*	*y2,y3,y4,y5,y6,y8,y10*	*y2,y4,y7*	*y2,y4*	*y2,y4*
	c-Network	*y8*	*y1,y3,y8*	*y1,y2,y8*	*y2,y8*	*y2,y8*

Type	Meaning	Message Example
y1	MGS Recovery failed	*...ptlrpc_connect_interpret() recovery of MGS on MGC 192.x.x.x...failed (-110)* *...ptlrpc_fail_import() import MGS@MGC10.x.x.x@tcp_0 for...not replayable...*
y2	MDS Recovery failed	*...ptlrpc_connect_interpret() recovery of lustre-MDT0000_UUID...failed (-110)*
y3	OSS Recovery failed	*...ptlrpc_connect_interpret() recovery of lustre-OST0001_UUID...failed (-110)*
y4	Log updating failed	*...mgc_process_log() MGC.x.x.x@tcp: configuration from log lustre-sptlrpc failed (-2).*
y5	Lock service failed	*...ldlm_request.c:1317: ldlm_cli_update_pool()...@Zero SLV or Limit found...*
y6	Failing over MDT	*...obd_config.c:652:class_cleanup() Failing over lustre-MDT0000...*
y7	Failing over OST	*...obd_config.c:652:class_cleanup() Failing over lustre-OST0001...*
y8	Client's request failed	*...ptlrpc_expire_one_request()...Request sent has timed out for slow reply:...rc 0/-1* *...ptlrpc_expire_one_request()...Request sent has failed due to network error:...rc 0/-1*
y9	Client was evicted	*...ldlm_failed_ast()...MGS: A client on nid...tcp was evicted... : rc -107* *...ldlm_handle_ast_error()...client...returned error from...(...rc -107), evict it ns: ...*
y10	Client-server connection failed	*...ptlrpc_check_status()...: operation ost_connect to node...failed: rc = -5/-16/-30*
y11	Client-OST I/O errors	*...tgt_client_del()...failed to update server data, skip client ... zeroing, rc -5* *...tgt_client_new()...Failed to write client lcd at idx..., rc -5/-30*
y12	Local metadata unaccessible	*...osd_ldiskfs_write_record() sdb: error reading offset... rc = -5* *...osd_ldiskfs_read() sdb: can't read ...@...on ino... : rc = -5* *...osd_idc_find_or_init() can't lookup: rc = -5* *...osd_trans_commit_cb() transaction @... commit error: 2*

metadata. The second phase of LFSCK (*"lfscl_layout"*) needs to access the metadata on OSSes, which will trigger *y12* under the fault models.

In summary, we find the messages in the debug buffer of Lustre (if reported) to be detailed and informative. As shown in the "Message Example" of Table 11 and able 12. 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.

A.3 Failure Logs of BeeGFS v7.1.3

Table 13 summarizes the BeeGFS logs. As shown in the table, the logs can be roughly classified into 15 types ("y1" to "y15"). Each log message usually contains multiple sentences describing the issue in details, including specific IDs of relevant nodes and/or files ("Message Example"). Therefore,

compared to the logs of Lustre (Table 11, Table 12 and Table 10X in §A.1 and §A.2), we find that BeeGFS's logging is more sophisticated.

Table 13. **Characterization of Logs of BeeGFS v7.1.3 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 "y15" are 15 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.*

Node(s) Affected	Fault Models	MGS Logs	MDS Logs	OSS#1 Logs	OSS#2 Logs	OSS#3 logs
MGS	a-DevFail	*y1,y2*	–	–	–	–
	b-Inconsist	–	–	–	–	–
	c-Network	*y1,y2,y3,y4,y5*	*y7,y8*	–	–	–
MDS	a-DevFail	–	*y9,y10,y11*	–	–	–
	b-Inconsist	–	*y9,y10,y11*	–	–	–
	c-Network	*y4*	*y5,y7,y8,y12,y13*	–	–	–
OSS#1	a-DevFail	–	–	*y14*	–	–
	b-Inconsist	–	–	–	–	–
	c-Network	*y3*	*y7,y8,y12,y14,y15*	–	–	–
three OSSes	a-DevFail	–	–	*y14*	*y14*	*y14*
	b-Inconsist	–	–	–	–	–
	c-Network	*y3*	*y7,y8,y12,y14,y15*	*y5,y7,y8,y12,y13*	*y5,y7,y8,y12,y13*	*y5,y7,y8,y12,y13*
MDS + OSS#1	a-DevFail	–	*y9,y10,y11*	*y14*	–	–
	b-Inconsist	–	*y9,y10,y11*	–	–	–
	c-Network	*y4,y6*	*y5,y7,y8,y12,y13*	*y7,y8*	–	–

Type	Meaning	Message Example
y1	Temporary file failure	*...TempFileTk.cpp:29 >> Could not open temporary file. tmpname:...* *...TempFileTk.cpp:65 >> Could not write to temporary file tmpname:...* *...TempFileTk.cpp:44 >> Failed to unlink tmpfile after error...Read-only file system...*
y2	Target state write failed	*...MgmtdTargetStateStore.cpp:431 >> Could not save target states. nodeType...*
y3	OSS auto-offline	*...Auto-offline >> No...received from storage target for...seconds...set...offline...* *...Auto-offline >> No...received from storage target for...seconds...set...probably-offline...*
y4	MDS auto-offline	*...Auto-offline >> No...received from metadata node for...seconds...set...offline...* *...Auto-offline >> No...received from metadata node for...seconds...set...probably-offline...*
y5	Unreachable network	*...StandardSocket::sendto >> Attempted to send message to unreachable network:...*
y6	Download from MGS failed	*...InternodeSyncer.cpp:418 >> Downloading...from management node failed...* *...InternodeSyncer.cpp:784 >> Download from management node failed...* *...Update states and mirror groups >> Downloading...from management node failed...*
y7	Connect failed	*...NodeConn... >> Connect failed...Error: Unable to establish connection...* *...NodeConn... >> Connect failed on all available routes...*
y8	Retrying communication	*...MessagingTk... >> Retrying communication...message type: GetNodes...* *...MessagingTk.cpp:281 >> Retrying communication...message type: CloseChunkFile...*
y9	Entry directory lost	*...Inode... >> Unable to open entries directory:...No such file or directory*
y10	Inode read failed	*...MetaStore... >> Failed to read inodes from hash dirs...*
y11	Directory entry related failures	*...StorageTkEx... >> Unable to open dentries directory...No such file or directory* *...DirEntry... >> Unable to create dentry file:...No such file or directory* *...make meta dir-entry >> Failed to create: name:...entryID:...in path:...* *...DirEntryStore...>> Unable to open dentry directory...No such file or directory*
y12	RPC related failure	*...Messaging (RPC) >> Communication error: Receive timed out from...* *...Messaging (RPC) >> Communication error: Receive timed out from:...* *...Messaging (RPC) >> Unable to connect to:...*
y13	MGS release failed	*...XNodeSync >> Pushing node free space to management node failed.*
y14	Chunk related failure	*...ChunkFetcherSlave.cpp:108 >> readdir failed...Input/output error...* *...ChunkDirStore... >> Unable to create chunk path:...Read-only file system* *...ChunkStore.cpp:661 >> Unable to create path for file...Read-only file system...* *...SessionLocalFile (open) >> Failed to open chunkFile:...* *...Close Helper... >> Problems occurred during release of storage server file handles...* *...Close Helper (close chunk files S) >> Problems occurred during close of chunk files...* *...Stat Helper (refresh chunk files) >> Problems occurred during file attribs refresh...*
y15	Communication with OSS failed	*...Close chunk file work >> Communication with storage target failed...* *...Close Helper... >> Communication with storage target failed:...Communication error* *...Stat chunk file work >> Communication with storage target failed...*

Also, we find that all BeeGFS nodes, including MGS, can report extensive events, which implies all nodes are always active (unlike Lustre's MGS). For example, when "c-Network" happens on MGS, multiple failure events are recorded on MGS (i.e., *y1*,*y2*,*y3*,*y4*,*y5*). When "c-Network" happens on other nodes (e.g., MDS or OSS), MGS can also record failure events accordingly. This implies that MGS is responsible for monitoring the network connection of all other nodes. If the network connection between MGS and any other node is broken, MGS will record that the corresponding node is "Auto-offline".

All three fault models can trigger extensive log messages in BeeGFS. However, in contrast to Lustre, BeeGFS's logs often concentrate on the affected node(s). For example, when "a-DevFail" happens on OSS (the "OSS#1" and "three OSSes" rows), only OSSes themselves generate logs (i.e., *y14*), and the logs are all about data chunks on the affected OSS. No MGS or MDS logs are generated. Similarly, when "a-DevFail" happens on MDS (the "MDS" row) and causes metadata loss (i.e., *y9*,*y10*,*y11*), no OSS logs are reported.

Compared to Lustre, BeeGFS generates less logs under "b-Inconsist" fault model. For example, only MDS has logs about "b-Inconsist" (i.e., *y9*,*y10*,*y11*). Note that the logs are the same as the logs under "a-DevFail". This implies that more fine-grained checking and logging mechanism is needed to differentiate the two different cases.

The "c-Network" fault model leads to the largest amount of logs on BeeGFS. When "c-Network" happens on MGS (the "MGS" row), MGS reports multiple types of logs as discussed previously; moreover, MDS outputs logs about connection failure (i.e., *y7*) and communication retry (i.e., *y8*). Similarly, when "c-Network" happens on MDS or OSS, the affected node may report a variety of logs including network/connection failures (i.e., *y5* and *y7*), RPC related failures (i.e., *y12*), retrying communication (i.e., *y8*), MGS release failed (i.e., *y13*), etc. This diversity suggests that BeeGFS has a relatively comprehensive monitoring mechanism.

In summary, we find that BeeGFS logs are more detailed and comprehensive than Lustre logs. Particularly, the MGS is heavily involved in logging, which is consistent with BeeGFS's design. On the other hand, we find that BeeGFS's logging is still suboptimal. For example, there are few logs about data inconsistencies on OSS nodes, and device failure and metadata inconsistency are logged in the same way, which suggests that there is still much room for improvement in terms of accurate logging.