

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 LFSCCK to detect and fix PFS-level inconsistencies, but we find that LFSCCK itself may hang or trigger kernel panics when scanning a corrupted Lustre. Even after the recovery attempt of LFSCCK, 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-FSCCK. We analyze the root causes of the abnormal symptoms observed in depth, which has led to a new patch to fix a kernel panic problem in the latest Lustre. 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 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, reliability, high performance computing

ACM Reference Format:

Runzhou Han, Om Rameshwar Gatla, Mai Zheng, Jinrui Cao, Di Zhang, Dong Dai, Yong Chen, and Jonathan Cook. 2021. A Study of Failure Recovery and Logging of High-Performance Parallel File Systems. 1, 1 (July 2021), 38 pages. <https://doi.org/10.1145/nnnnnnnn.nnnnnnnn>

¹An early version was published in ACM International Conference on Supercomputing (ICS'18); feedback welcome!

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.

Technical Report: ISU-DSL-TR2020b

© 2021 Association for Computing Machinery.

XXXX-XXXX/2021/7-ART \$15.00

<https://doi.org/10.1145/nnnnnnnn.nnnnnnnn>

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 numbers (e.g., EIO, EBUSY, EROFS) across different types of storage nodes in the cluster, while BeeGFS may only log two types of Linux error numbers on limited nodes under the same faults. 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.

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 to be merged to the latest Lustre [34].

To the best of our knowledge, this work is the first comprehensive study on the failure recovery and logging mechanisms of production PFSes. By detailed comparison and analysis on multiple versions of Lustre and BeeGFS, 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 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 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.

¹The prototype of PFAULT and the experimental logs are publicly available at <https://git.ece.iastate.edu/data-storage-lab/prototypes/pfault>

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.

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 LFSCCK, 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]. For simplicity, we use **FSCK** to refer to both LFSCCK 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 pluggable 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 (the initiator) to access the remote block storage 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.

3 THE PFAULT TOOL

In this section, we describe the design and implementation of the PFAULT tool. While there are many desired objectives, PFAULT aims to meet three key goals as follows:

- **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 views 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.** 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.
- **Fidelity.** PFAULT can emulate diverse external failure events with high fidelity without changing the PFS itself (i.e., non-intrusive).

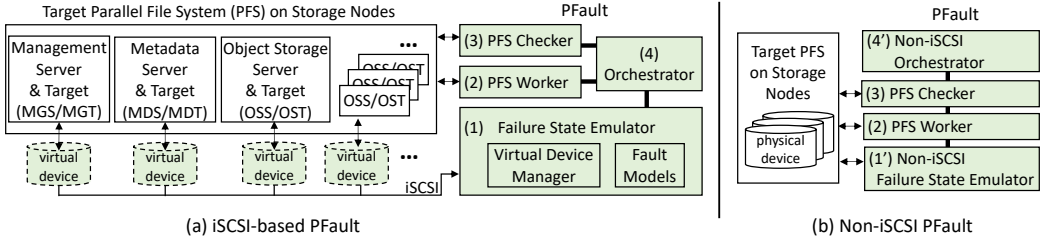


Fig. 1. **Overview of PFAULT.** The shaded boxes are the major components of PFAULT. (a) the iSCSI version enables manipulating PFS images efficiently; (b) the non-iSCSI version enables verifying the potential impact of iSCSI on the target PFS.

3.1 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. Moreover, it 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.

3.2 Failure State Emulator

To study the failure recovery and logging of the PFS, it is necessary to generate faults in a systematic and controllable way. To this end, PFAULT reduces various failure events to the states of storage devices via *Failure State Emulator*, which mainly includes two sub-components: *Virtual Device Manager* and *Fault Models* (Figure 1a).

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*[57] 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, 58]. 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 manipulate 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` [59] 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 that beyond the repairing capability of the local file system. 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 [60], which may be caused by dysfunctional network devices (e.g., switch [61]) 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, 62–67]. 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 [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, which can help identify the potential corruption of user files from the end user's perspective without relying on the logging component of the PFS. 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., LFSD, 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. 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.

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.

4 STUDY 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

Three platforms are used in the study: (1) a seven-node cluster created on virtual machines hosted on one high-end physical server (Intel Xeon Gold 2.3GHz CPU x2, 256GB DRAM, 960GB SSD, 2TB HDD); (2) a twenty-node cluster created on CloudLab [70]; (3) a four-node cluster consisting of physical servers. All results are collected using the iSCSI-based PFAULT on the first seven-node platform for convenience, and a subset with unexpected symptoms (e.g., hang, rebooting) has been reproduced and verified on CloudLab (using iSCSI-based PFAULT) and the physical cluster (using non-iSCSI PFAULT). The results are consistent across platforms in our experiments.

In the seven-node 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.

On the five storage nodes, we create the target PFS with one MGS node, one MDS node, and three OSS nodes. On each node, there is one iSCSI virtual device mounted to serve as the corresponding target device (i.e., MGT/MDT/OST).

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.

4.3 Workloads

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,

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

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., LFCK 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 [59], LFCK, 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). In each experiment, we collect the logs generated by PFS and its FSCK for further analysis. The logs are collected at three different phases of each experiment with the help of PFAULT: (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 summarizes the subset of log files used for in-depth manual study. In total, we have studied 1,215 log files for Lustre/LFCK 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). LFCK 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 LFCK, 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.

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

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 (1,215 in total)
LFSCK	N/A	135	90	90	90	N/A	
BeeGFS	135	135	135	135	135	135	810 BeeGFS + 90 BeeGFS-FSCK (900 in total)
BeeGFS-FSCK	N/A	N/A	N/A	N/A	N/A	90	

example, in our preliminary experiments [33] 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 (§5.2.2) has been confirmed by developers, and a new patch has been generated based on our study to fix the problem in the latest Lustre [34].

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).

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 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 3 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

Table 3. **Behavior of LFSCCK and Post-LFSCCK 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”: LFSCCK 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	LFSCCK	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

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/LFSCCK 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, LFSCCK may fail unexpectedly. For example, when “a-DevFail” happens on MDS (the “MDS” rows), LFSCCK 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 LFSCCK 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, 73].

When “a-DevFail” happens on OSS (the “OSS#1” row), v2.8 and v2.10.x differ a lot. On v2.8, LFSCCK and all workloads hang. However, on v2.10.x, LFSCCK 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-async can

Table 4. **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

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 LFSC 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 LFSC finishes normally without any warning (“normal”). In fact, LFSC’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 LFSC is likely incomplete, similar to the observation on local file system checkers [41, 73].

When “b-Inconsist” happens on OSS (the “OSS#1” row), running LFSC may crash storage nodes and trigger rebooting abruptly (“reboot” in the “LFSC” 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 4 summarizes the behavior of BeeGFS-FSCK and the behavior of the workloads after running BeeGFS-FSCK. In general, we find that compared with LFSC, 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 LFSC (Table 3): when applying “c-Network” to MGS, BeeGFS-FSCK may “hang” (i.e., no progress within one hour), while LFSC 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 inaccessible 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 3 and Table 4 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).

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, Cassandra) which commonly use unified and well-formed logging libraries (e.g., Log4J [32] or SLF4J [74]), we find that the logging methods of PFSes are diverse and irregular. Table 5 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

Table 5. **Logging methods observed in Lustre and BeeGFS.**

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

Table 6. **Standard Linux Error Numbers Captured in PFS Failure Logs.**

Error #	Error Name	Description	Logged by Lustre?	Logged by BeeGFS?
-2	ENOENT	No such file or directory	Yes	
-5	EIO	I/O error	Yes	Yes
-11	EAGAIN	Try again	Yes	
-16	EBUSY	Device or resource busy	Yes	
-30	EROFS	Read-only file system	Yes	Yes
-107	ENOTCONN	Transport endpoint is not connected	Yes	
-110	ETIMEDOUT	Connection timed out	Yes	

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 10, Table 11, and Table 12 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 use standard Linux error numbers in their logging methods, so we consider logs with Linux error numbers as failure handling logs. In addition, for logs appear after the fault injection but do not contain explicit error numbers, 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 rules above as *error messages* in the rest of this section.

Table 6 summarizes all types of Linux error numbers captured in the two PFSes after fault injection in our experiments. We can see that Lustre uses a wide set of standard error numbers (e.g., -2, -5, -11, -16, -110) while BeeGFS only captures two numbers (i.e., -5 and -30). This is mainly due to the different design choices: 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 directly.

Figure 2 further shows the distribution of error messages containing Linux error numbers (i.e., -2, -5, -11, -16, -30, -107, -110) 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. 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, an error message containing “-110” is captured on MDS, OSS#1, OSS#2 and OSS#3.

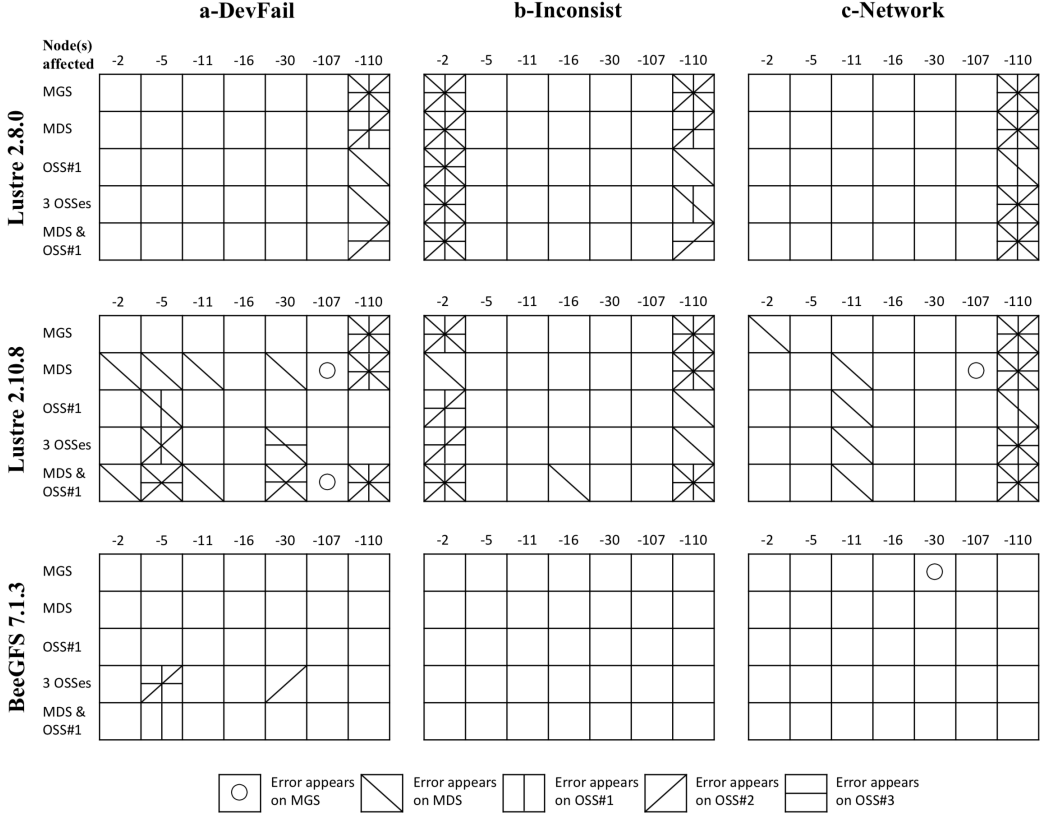


Fig. 2. Distribution of PFS Error Messages. This figure shows the distribution of error messages of Lustre (two major versions) and BeeGFS after applying three fault models (i.e., a-DevFail, b-Inconsist, c-Network). The “Node(s) Affected” column shows where the faults are injected.

Based on Figure 2, we can clearly see that Lustre v2.10.8 generates error messages on more nodes with more 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 “-2”, “-5”, “-11”, “-30” on MDS, “-107” on MGS, and “-110” on MDS and all OSS nodes. On the other hand, Lustre v2.8.0 only reports “-110” under the same fault. This implies that Lustre v2.10.8 has enhanced the failure logging significantly compared to v2.8.0.

Also, it is interesting to see that “-110” is logged in the two Lustre versions under all three fault models. “-110” means connection timed out (Table 6), 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 3. 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.

Compared to Lustre, BeeGFS’s error message distribution looks much more sparse in Figure 2. For example, only “-5” and “-30” are captured under a-DevFail. This confirms that BeeGFS does not leverage Linux error numbers as much as Lustre 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 customized messages which are equivalent to the standard Linux error numbers. For example,

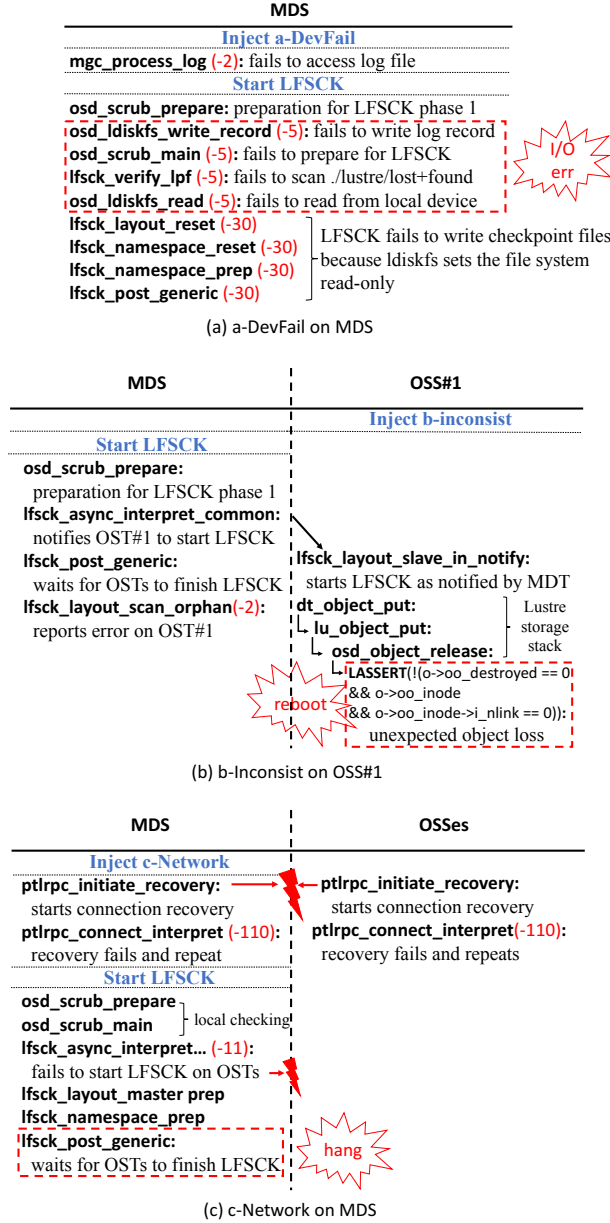


Fig. 3. **Internal Operations of Lustre and LFSCk After Three Types of Faults.** Each bold black statement represents one Lustre function, which is followed by a short description. Blue lines represent PFault operations. Red dash boxes highlight the key operations leading to the abnormal symptoms observed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

we observe that in many BeeGFS error messages there are descriptions such as “No such file or directory” which is equivalent to Linux error number “-2”. Please refer to Appendix (§A) for more concrete examples and more detailed characterization of the error messages.

5.2.2 Root Cause Analysis. 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 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 3). We illustrate the three simplified cases using Figure 3.

Specifically, Figure 3 shows the critical internal operations of Lustre and LFSCk under three 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 Lustre function, which is followed by a short description below it. PFault operations are represented in blue. The red dash boxes highlight the key operations 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 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.

(2) b-Inconsistent 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 “`lfsc_k_async_interpret_common`” on MDS to “`lfsc_k_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 has been generated to fix the problem based on our analysis[34].

(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 LFSCCK 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 LFSCCK thread on OSS, the request cannot be delivered to OSS due to the network partition. After finishing the local checking steps on MDS, LFSCCK keeps waiting (“lfscck_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 LFSCCK to maintain a timer instead of hanging forever. We discuss such optimization opportunities further in §6.

5.3 Logs of LFSCCK and BeeGFS-FSCCK

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

5.3.1 LFSCCK Logs. In addition to the failure logs of Lustre discussed in §5.2.1, we find that LFSCCK 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 LFSCCK logs maintained in the /proc pseudo file system, and *debug buffer log* to refer to LFSCCK-triggered events in the debug buffer of Lustre. We characterize these logs in details below.

We find that there are three types of LFSCCK status logs, each of which corresponds to one major component of LFSCCK: (1) **oi_scrub log (oi)**: linearly scanning all objects on the local device and verifying object indexes; (2) **layout log (lo)**: checking the regular striped file layout and verifying the consistency between MDT and OSTs; (3) **namespace log (ns)**: checking the local/global namespace consistency inside/among MDT(s). 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 LFSCCK status logs are generated on MGS.

Table 7 summarizes the logs (i.e., “oi”, “lo”, “ns”) generated on different Lustre nodes after running LFSCCK. Similar to Table 3, 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, LFSCCK 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, LFSCCK 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)”), LFSCCK 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 LFSCCK logs on the affected node(s) disappear from the /proc file system, and thus are unavailable (i.e., “-”).

When LFSCCK hangs (i.e., “hang” in Table 3), the logs may keep showing that it is in scanning. We find that internally LFSCCK 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 LFSCCK and show the hanging phases (i.e., “scan-1” or “scan-2”) in 7.

Table 7. **Characterization of LFSCCK 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 LFSCCK “completed”; “init” means the log shows the “init” state (no execution of LFSCCK); “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
OSS#1	c-Network (v2.10.8)	comp	scan-1	scan-1	init	init	init	init	init	init
	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
three OSSes	c-Network (v2.10.8)	comp	scan-1	scan-1	init	init	comp	scan-2	comp	scan-2
	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
MDS + OSS#1	c-Network (v2.10.8)	comp	scan-1	scan-1	comp	comp	comp	comp	comp	comp
	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

Table 8 further summarizes the debug buffer logs triggered by LFSCCK. We find that there are three subtypes of LFSCCK debug buffer logs (i.e., $x1$, $x2$, $x3$), which corresponds to the three phases of LFSCCK (i.e., *oi_scrub*, *lfscck_layout*, *lfscck_namespace*) respectively. Also, most logs are triggered on MDS (i.e., the “MDS” column), which implies that MDS plays the most important role for LFSCCK execution and logging; and most of the triggered error messages are related to *lfscck_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 *lfscck_layout* procedure involves and depends on a variety of internal operations on local systems. Since LFSCCK is designed to check and repair the corrupted PFS cluster, it is particularly interesting to see that LFSCCK itself may fail when the local systems are locally correct (i.e., “b-Inconsist” row).

To sum up, we find that in terms of LFSCCK status logs, in most cases (other than the two “repaired” cases in Table 7), the logs are simply about LFSCCK’s execution steps (e.g., “init”, “scan-1”, “scan”, “comp” in Table 7), which provides little information on the potential corruption of the PFS being examined by LFSCCK. On the other hand, the corresponding debug buffer log of LFSCCK is relatively more informative (Table 8), as it may directly shows the failed operations of LFSCCK. To guarantee that we do not miss any valuable error messages, we run LFSCCK before injecting the faults to

Table 8. **Characterization of LFSCCK-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 corresponding to the three phases of LFSCCK: *oi_scrub*, *lfscck_layout* and *lfscck_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 ...fail to notify...for lfscck_layout start: rc = -5/-11/-30			
x2	lfscck_layout failed	...lfscck_verify_lpf()...scan .lustre/lost+found/ for bad sub-directories: rc = -5 ...lfscck_post_generic()...waiting for assistant to do lfscck_layout post, rc = -30 ...lfscck_layout_store()...fail to store lfscck_layout: rc = -30 ...lfscck_layout_reset()...layout LFSCCK reset: rc = -30 ...master engine fail to verify the .lustre/lost+found/... : rc = -5 ...layout LFSCCK slave gets the MDT 0 status -11... ...layout LFSCCK hit first non-repaired inconsistency at...			
x3	lfscck_namespace failed	...lfscck_namespace_prep()...namespace LFSCCK prep failed: rc = -30 ...lfscck_namespace_reset()...namespace LFSCCK reset: rc = -30			

generate a set of logs under the normal condition. Then, we compare the logs of the two runs of LFSCCK (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 LFSCCK. Therefore, we believe the characterization of LFSCCK logs is accurate.

5.3.2 BeeGFS-FSCK Logs. Unlike LFSCCK 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 characterize BeeGFS-FSCK’s logs in Table 9.

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 LFSCCK’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 4, 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 4).

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

Table 9. **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 mgmt; “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

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 inaccessible 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 DISCUSSIONS AND FUTURE WORK

The work described in this paper suggests many opportunities for further improvements. We discuss a few example directions in this section.

6.1 Black-Box vs. Grey-Box/White-Box

The current PFAULT prototype follows a black-box principle [28] for usability and portability, which does not require the expert knowledge of the internals of PFSes. This design decision is largely based on the unique architecture and complexity of PFSes as described in §1 and §2. In contrast, 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 input, which may improve the test efficiency. However, such approach typically requires strong tool support for code analysis or instrumentation (e.g., AspectJ [75] for Java programs), which remains challenging for the PFS code base with substantial weakly typed code (e.g., C) in the kernel. Therefore, we leave the exploration of grey-box/white-box approaches for studying PFSes as future work.

We believe that the black-box PFAULT as well as the extensive study demonstrated in this work is a fundamental step towards more sophisticated analysis for PFSes. To the best of our knowledge, none of existing fault injection tools can achieve PFAULT's functionality for studying PFSes without substantial modification efforts (if possible at all). And this work is the first comprehensive study on the failure recovery and logging mechanisms of production PFSes. One the other hand, we believe that it is possible to integrate other techniques with PFAULT to improve the effectiveness further. We elaborate on one of such techniques in the next subsection.

6.2 Integration with Fuzzing

Fuzzing is a classic technique for generating effective inputs and improving the test coverage [76]. Since 1990s [77], fuzzing has been applied to study a wide range of programs [7, 76–78]. In particular, a number of fuzzing tools (i.e., fuzzers) have been proposed for practical systems including Linux file systems in recent years. For instance, Janus [78] 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 [79] through manipulation on RPC messages in a black-box manner without feedback loop or code coverage measurement. Similarly, AFLNET [80] 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 [81], 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 [82].

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 [76], 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 FSCCK component and the logging methods). 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 features are expected to provide additional reliability and/or availability guarantees for PFSes. However, they may not be able to handle all the faults emulated in this study. For example, the process failover mechanism in Lustre is unlikely to handle the a-DevFail fault model due to the sharing of device. Due to the limitation of our current hardware platform, we leave the study of such hardware-dependent features as future work.

6.4 Improving 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 LFSCCK (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 LFSCCK in details and proposes to improve the completeness of LFSCCK accordingly [83].

In addition, we may apply PFAULT to study and improve other important PFSes (e.g., OrangeFS, Ceph). Since PFAULT is designed with generality and portability in mind, we expect the porting efforts to be minimal.

6.5 Logging Support and Learning-based Log Analysis

In this study, we do not modify or instrument the target PFS. As a result, the current root cause analysis has to rely on the default logging mechanisms of the PFS, which is often insufficient as exposed in our study. In other words, we believe there are research opportunities to enhance the logging support of PFS based on this study. Moreover, the extensive experimental logs generated in our study include both normal and abnormal cases. We believe this is a valuable dataset for applying machine learning based approaches to optimize PFSes, which has been shown to be promising for failure detection and diagnosis of other large-scale systems (e.g., DeepLog [84]). On the other hand, since PFS logs are more diverse and irregular than typical cloud systems logs, more sophisticated algorithms are likely needed to handle them properly. We release the prototype of PFAULT as well as the experimental logs publicly on GitLab to facilitate the follow-up research in the communities [85].

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. For example, there is a variety of tools for instrumentation, profiling, and tracing of I/O activities, such as mpiIP [86], LANL-Trace [87], HPCT-IO [88], IOT [89], and TRACE [90] 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. On the other hand, we believe that these tools may also help in reliability. For example, Darshan [91, 92] 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, another work [93] proposes to study the crash consistency of PFSes via replaying workload traces, which may benefit from the extensive real logs collected via PFAULT. Therefore, PFAULT and the existing PFS works 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 [94] 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 [75], 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 [95] and Reflection [96]) 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 [97] and Javassist [98]) 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 [99]), 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 [74]), 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, 100–102], 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, 65], RAID [5], flash-based SSDs [102, 103], local file systems and checkers [6, 7, 41, 78, 104–107]) through a variety of approaches (e.g., fault injection [58, 107], model checking [104], formal methods [108], fuzzing [7, 78]). 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 editors for their time and insightful feedback. The authors also thank Andreas Dilger and other PFS developers for valuable discussion. 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. Douglass, 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-Kiswani, 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, "Samec: 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] LFSCCK: 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*, ser. ICS'18, 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)*. IEEE, 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] e2fsck(8) — Linux manual page, 2017. [Online]. Available: <https://man7.org/linux/man-pages/man8/e2fsck.8.html>
- [58] 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.
- [59] debugfs, 2017. [Online]. Available: <http://man7.org/linux/man-pages/man8/debugfs.8.html>
- [60] 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.
- [61] 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>
- [62] Network Partition, 2017. [Online]. Available: <https://www.cs.cornell.edu/courses/cs614/2003sp/papers/DGS85.pdf>
- [63] 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>
- [64] 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.
- [65] 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.
- [66] 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.
- [67] 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
- [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. Kuzmaul, 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. 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.
- [74] Simple logging facade for Java, 2019. [Online]. Available: <http://www.slf4j.org>
- [75] AspectJ, 2001–now. [Online]. Available: <https://www.eclipse.org/aspectj/>
- [76] 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.
- [77] 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>

- [78] 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.
- [79] 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/>
- [80] 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.
- [81] socket(2) — Linux manual page, 2020. [Online]. Available: <https://man7.org/linux/man-pages/man2/socket.2.html>
- [82] 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.
- [83] 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.
- [84] 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.
- [85] GitLab repository for PFault by Data Storage Lab@ISU , 2020. [Online]. Available: <https://git.ece.iastate.edu/data-storage-lab/prototypes/pfault>
- [86] 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.
- [87] HPC-5 Open Source Software project, LANL-Trace, 2015. [Online]. Available: institutes.lanl.gov/data/tdata/
- [88] 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.
- [89] 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.
- [90] 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.
- [91] Darshan:HPC I/O Characterization Tool, 2017. [Online]. Available: <http://www.mcs.anl.gov/research/projects/darshan/>
- [92] 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.
- [93] J. Sun, C. Wang, J. Huang, and M. Snir, "Understanding and Finding Crash-Consistency Bugs in Parallel File Systems," in *HotStorage20*, 2020.
- [94] 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.
- [95] Java Virtual Machine Tool Interface (JVM TI). [Online]. Available: <https://docs.oracle.com/javase/8/docs/technotes/guides/jvmti/>
- [96] Trail: The Reflection API. [Online]. Available: <https://docs.oracle.com/javase/tutorial/reflect/index.html>
- [97] "WALA Home page," 2015. [Online]. Available: <http://wala.sourceforge.net/wiki/index.php/>
- [98] Java bytecode engineering toolkit, 1999. [Online]. Available: <https://www.javassist.org/>
- [99] The LLVM Compiler Infrastructure, 2020. [Online]. Available: <https://llvm.org>
- [100] 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>
- [101] 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>
- [102] 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>
- [103] 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>
- [104] 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.

- [105] 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>
- [106] 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>
- [107] O. R. Gatla, M. Hameed, M. Zheng, V. Dubeyko, A. Manzanaraes, 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.
- [108] 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 10, 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 does 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.

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,

Table 10. 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...			

and they try to recover MDT but eventually fail (i.e., y_2). 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., y_4) and locking (i.e., y_5) 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., y_6 , y_7). 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 11. Note that we have discussed LFCK-specific debug buffer logs of Lustre v2.10.8 in Table 8 (§5.3), so we skip them in here.

As shown in Table 11, the first seven types of error messages (i.e., y_1 - y_7) are almost the same as the corresponding messages in Table 10. Message y_4 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., y_8 – y_{12} in Table 11) compared to v2.8 (in Table 10). Specifically, y_8 to y_{11} (i.e., *Client’s request failed*, *Client was evicted*, *Client-server connection failed*, *Client-OST I/O errors*) are client related failures; and y_{12} represents failures of accessing metadata on OST.

Besides generating different types of error messages, another key difference between v2.10.8 (Table 11) and v2.8 (Table 10) is that MGS does report some information under faults in v2.10.8 (Table 11). 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., y_8). 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., y_9). In the meantime, MDS reports that the connection between client and servers fails (i.e., y_{10}), and logs from OST#2 and OST#3 shows that they encounter errors when dealing with clients I/O requests (i.e., y_{11}). 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 inaccessible* (i.e., y_{12}) 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 y_{12} often appears together with LFCK-triggered error messages (Table 8 in §5.3). This is because LFCK is responsible for checking and repairing the metadata. The second phase of LFCK (*“lfscl_layout”*) needs to access the metadata on OSSes, which will trigger y_{12} under the fault models.

Table 11. Characterization of Logs Generated in the Debug Buffer of Lustre v2.10.8 After Faults. Similar to Table 10, 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 inaccessible	...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				

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 10 and able 11. 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 12 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 10, Table 11 and Table 10X in §A.1 and §A.2), we find that BeeGFS’s logging is more sophisticated.

Table 12. 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		...MgmtTargetStateStore.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. ...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:...			
y14	Chunk related failure		...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., y_{14}), 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., y_9, y_{10}, y_{11}), 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., y_9, y_{10}, y_{11}). 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., y_7) and communication retry (i.e., y_8). Similarly, when “c-Network” happens on MDS or OSS, the affected node may report a variety of logs including network/connection failures (i.e., y_5 and y_7), RPC related failures (i.e., y_{12}), retrying communication (i.e., y_8), MGS release failed (i.e., y_{13}), 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.