Fingerprinting the Checker Policies of Parallel File Systems

Runzhou Han
Iowa State University
hanrz@iastate.edu

Duo Zhang
Iowa State University
duozhang@iastate.edu

Mai Zheng
Iowa State University
mai@iastate.edu

***Abstract*—Parallel file systems (PFSes) play an essential role in high performance computing. To ensure the integrity, many PFSes are designed with a checker component, which serves as the last line of defense to bring a corrupted PFS back to a healthy state. Motivated by real-world incidents of PFS corruptions, we perform a fine-grained study on the capability of PFS checkers in this paper. We apply type-aware fault injection to specific PFS structures, and examine the detection and repair policies of PFS checkers meticulously via a well-defined taxonomy. The study results on two representative PFS checkers show that they are able to handle a wide range of corruptions on important data structures. On the other hand, neither of them is perfect: there are multiple cases where the checkers may behave sub-optimally, leading to kernel panics, wrong repairs, etc. Our work has led to a new patch on Lustre. We hope to develop our methodology into a generic framework for analyzing the checkers of diverse PFSes, and enable more elegant designs of PFS checkers for reliable high-performance computing.**

I. Introduction

High-performance parallel file systems (PFSes) play an essential role today. A variety of PFSes (e.g., Lustre [23], BeeGFS [29], OrangeFS [43], Ceph [42]) have been deployed widely in national labs [6] and data centers [2] to empower large-scale I/O intensive computing. Therefore, the integrity of PFSes is critically important.

To ensure the integrity of PFSes and avoid system downtime or data loss, various fault tolerance techniques have been proposed (e.g., RAID [9], checkpointing [32], [38], [40], failover [7], journaling [15], [37]). Unfortunately, despite the great efforts, PFSes may still become corrupted in practice for many reasons including software bugs, hardware faults, power outages, etc. [11]. For example, in an incident at the High Performance Computing Center (HPCC) in Texas, the storage clusters managed by Lustre were severely corrupted after power outages [12]. Similar cases have been reported in other recent studies [22].

To handle the potential corruption, many PFSes are designed with a *checker* component (e.g., LFSCK [33] for Lustre, BeeGFS-FSCK for BeeGFS). Similar to the checkers of local file systems (e.g., `e2fsck` for Ext2/3/4 [4], `xfs_check` for XFS [10]), the PFS checker usually scans the on-disk layout of the corresponding PFS, detects and repairs inconsistencies based on predefined policies, and serves as the last line of defense to bring the corrupted system back to a healthy state. For simplicity, we call the underlying detection and repair policies of PFS checkers as *checker policy* in this paper.

Nevertheless, designing a correct checker is notoriously difficult due to the complexity of the file system and the diversity of corruption scenarios [20], [25]–[28], [35]. For example, Gunawi *et al.* [28] find that the checker of Ext2 may generate inconsistent or even insecure repairs. Similarly, Carreira *et al.* [20] test the checkers of five popular local file systems and find bugs in all of them, including cases leading to data loss.

The deficiency exposed in local file system checkers raises the concern for PFS checkers, because PFS checkers depend on local file system states and need to examine a much larger scale of states across local file systems. In fact, in the HPCC incident mentioned above, the Lustre checker LFSCK failed to repair the corrupted Lustre for unknown reasons.

Unfortunately, to the best of our knowledge, there is little equivalent thorough study of PFS checkers, largely due to the lack of effective methodologies. One recent framework PFault [18] applies automatic fault injection to test the failure handling of Lustre. While it is effective for its original design goal, it is ineffective for studying PFS checkers due to its coarse-grained fault models and emulation methodology. For example, the whole device failure emulated by PFault can easily affect the entire system including the PFS and the local file system. As a result, it is difficult (if possible at all) to understand the root causes of the symptoms observed, let alone pinpointing the potential deficiency of the PFS checker. Similarly, another recent work [41] proposes to study the crash consistency of PFSes via replaying workload traces, which shares the same limitation as PFault since it crashes the entire system stack.

In this paper, we propose a fine-grained methodology to fingerprint the checker policy, which is one fundamental step to identify the potential deficiency and improve the design of PFS checkers. One key observation is that the PFS metadata is tightly interleaved with the metadata of local file systems on storage devices. For example, Lustre's ldiskfs backend is a variant of Ext4, and Lustre leverages the extended attributes of Ext4 inodes to store various metadata. Similarly, BeeGFS also makes use of the extended attributes of local inodes. In other words, the PFS metadata and the local file system metadata are closely correlated. Therefore, to analyze the PFS checker with high fidelity, we need to decouple such close correlation and identify the contract between PFS and the local file system.

Based on the key observation, we conduct a study on the

checker policy of two important and different PFSes: Lustre and BeeGFS. We leverage gray-box knowledge [14] of PFS metadata, apply *type-aware* fault injection to specific PFS structures while maintaining the integrity of local file systems, and examine the policies of the target PFS checker meticulously. Moreover, inspired by the classic IRON taxonomy for local file systems [39], we characterize the checker policy of Lustre and BeeGFS into multiple levels, which precisely captures the detection and repair capabilities.

The detailed characterization helps us identify opportunities for improving the state-of-the-art PFS checkers. Overall, we find that both LFSCK and BeeGFS-FSCK can detect and repair a variety of corruptions on important PFS data structures. However, there are still multiple cases where the checkers may behave sub-optimally, generating kernel panics, wrong repairs, invalid options, etc. The kernel panic case has been confirmed by Lustre developers and has led to a new patch. Encouraged by the study results, we are building an automatic framework for fingerprinting the checker policy of PFSes in depth. We hope to develop the methodology into an open-source framework to facilitate analyzing the checker policy of diverse PFSes and enable more elegant designs of PFS checkers for reliable high-performance computing.

The rest of the paper is organized as follows: §II introduces the background with motivating examples; §III describes our fingerprinting methodology; §IV presents experimental results on Lustre and BeeGFS; §V discusses future work.

II. Background & Motivation

A. Parallel File Systems & Their Checkers

Parallel file systems (PFSes) are the critical I/O infrastructure for high-performance computing (HPC). They are optimized for highly concurrent accesses to files at scale. We use Lustre [34], one of the most popular PFSes [30], to illustrate the typical architecture as follows:

- **MGS/MGT**: Management Server/Target manages and store the configuration information of Lustre.
- **MDS/MDT**: Metadata Server/Target manages and stores the metadata of Lustre; MDS provides network request handling for one or more local MDTs.
- **OSS/OST**: Object Storage Server/Target manages and stores the actual user data; OSS provides the file I/O service and handles network requests for local OSTs; user data are stored as one or more objects on OSTs.
- **Clients** launch applications to access the data in Lustre, usually from login nodes or compute nodes.

As the system scale and complexity keeps increasing, maintaining PFS consistency and data integrity is becoming more and more important and challenging. Therefore, Lustre introduces an online component called LFSCK [33] for detecting and repairing corruptions, which has been significantly enhanced since v2.6. Similarly, other popular PFSes also include a checker component as the last line of defense to handle corruptions (e.g., BeeGFS-FSCK, PVFS2-FSCK).

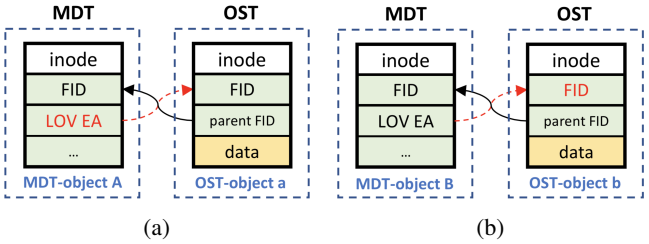

Fig. 1: **Two corruption cases on Lustre**. (a) Corrupted `LOV EA` on MDT, which can be detected and repaired by LFSCK; (b) Corrupted `FID` on OST, which *cannot* be repaired, although redundant information is available.

B. Examples of PFS Checker Policy

In this section, we demonstrate the checker policy of LFSCK using two concrete examples. Fig. 1 shows simplified Lustre data structures on MDT and OST, where white/green/yellow boxes represent inode, inode extended attribute, and data, respectively. In Fig. 1(a), OST-object **a** recognizes MDT-object **A** as its parent, because **a**'s "parent FID" maps to **A**'s "FID" (a global ID for each Lustre file). Normally, **A**'s "LOV EA" should also map to **a**'s "FID". In case "LOV EA" becomes corrupted, we find that LFSCK is able to detect and repair "LOV EA" via redundancy [46].

Similarly, in Fig. 1(b), MDT-object **B**'s "LOV EA" maps to OST-object **b**'s "FID". However, in case **b**'s "FID" becomes corrupted, we find that LFSCK may fail to repair it, although there is redundant information available. Such incompleteness may leave the PFS in a corrupted state, eventually causing downtime or even data loss. We propose a fine-grained approach to address the challenge in the next section.

III. Checker Policy Fingerprinting

In this section, we introduce our method to study the checker policies of PFSes. First, in order to trigger the detection and repairing operations of the target PFS checker, we apply type-aware fault injection to PFS images and generate precise and diverse corruptions (§III-A). Second, in order to qualitatively measure the checker policies, we develop a taxonomy to characterize the behaviors in details (§III-B).

A. Type-Aware Fault Injection

The behaviors of PFS checkers depend on the PFS states, which in turn depends on two major factors: the workloads applied to the PFS, and the corruptions occurred to the PFS. Therefore, the first step of our type-aware fault injection is to apply workloads to age the PFS and generate representative metadata layouts. Inspired by a recent work on aging file systems [21], we apply a git-based workload to create the aging effect efficiently. We plan to incorporate other I/O intensive workloads in future work (§V).

Next, after obtaining a meaningful metadata layout via aging, we inject faults that emulate realistic corruptions. Many existing works [18], [25], [41], [47] inject faults to the target system in a coarse-grained type-oblivious manner, i.e., they do

TABLE I: **PFS Data Structures.** The table shows the data structures of Lustre and BeeGFS used in our type-aware fault injection. For each structure, we list the name, the node location, the relation to local FS, and the usage. Note that the information is derived from the source codes and (incomplete) documentations based on our best efforts.

Lustre Structures	Node	Relation to Local FS	Usage
MDT-object	MDT	regular file (empty)	Provides extended attributes for Lustre metadata
OST-object	OST	regular file	Data stripes for user data
llog record	MDT	regular file	Stores Lustre transaction info
FID	MDT & OST	extended attribute	A unique, global ID of a Lustre file
FID location database (FLDB)	MDT	regular file	Maps FID.sequence_number to MDT/OSTs
object index (OI) table	MDT & OST	regular file	Maps FIDs to on-disk inode numbers
LOV EA	MDT	extended attribute	Stores one or more child OST-objects' FIDs
parent FID (PFID)	OST	extended attribute	Stores parent MDT-object's FID
linkEA	MDT	extended attribute	Stores MDT-object's name & its parent dir.'s FID
nlink	MDT	inode field	Number of hard links of the inode
BeeGFS Structures	Node	Relation to Local FS	Usage
dentry-by-name (MDT-object)	MDT	regular file (empty)	Stores user file name in its name entry
dentry-by-ID (MDT-object)	MDT	regular file (empty)	Stores child chunks' ID in its name entry
fhgfs (MDT-object)	MDT	extended attribute	Stores BeeGFS metadata
chunk (OST-object)	OST	regular file	Data stripes for user data & ID in its name entry
content directory	MDT	directory file (dir.)	List of MDT-objects in directory
nlink	MDT	inode field	Number of hard links of the inode
size	OST	inode field	Size of a chunk

not consider how a byte is being used by the target system. As a result, the injected faults often affect the entire software stack and trigger complex behaviors (e.g., crash, reboot) that are time-consuming and difficult to analyze.

To address the limitation, we inject well-defined faults to specific types of the PFS structures. Such fine granularity allows us to examine the different rules that the PFS checker applies for its different internal structures efficiently. The types of PFS data structures we test are listed in TABLE I, which includes both Lustre and BeeGFS. Note that both PFSes have data structures closely interleaved with the local file system (e.g., Lustre's `FID` and BeeGFS' `fhgfs` are embedded in the extended attributes of local inodes), which makes the type-awareness critically important for avoiding the complexity introduced by local file systems.

For the target PFS structures, we consider the following fault models, which are derived from the literature [18], [19], [24], [25], [39], [44], [48], [49] as well as the design documents of the PFS checkers [16], [33]. These fault models capture the typical corruptions that may occur in the local storage stack and be exposed to the PFS checker:

- *junk*: bytes of the structure are replaced by random values. This is the scenario of the two examples in Fig. 1, and as mentioned in §II, LFSCK may or may not be able to repair depending on the affected types.
- *zero*: bytes of the structure are replaced by zero. This is a special case of *junk*.
- *duplicate*: the structure has the same value as another structure of the same type. This may occur when doing file-level backup and restore on MDT servers [13].
- *out-of-sync*: In-memory copy of the structure is inconsistent with on-disk copy. This may be caused by memory/disk corruptions and software bugs.

For PFS structures stored as regular files/directories on the local file system, we use regular file operations to inject faults (e.g., `dd`, `rm`); for other structures stored as the extended attributes of local inodes, we use file system specific tools (e.g. , `debugfs` [3]). All operations maintain the integrity of the local file system. Also, to examine the checker policy precisely, we only inject one single fault at a time, similar to the previous work [24].

B. PFS Checker Taxonomy

Since the specific PFS structures and the consistency relations among them are diverse and complex, the detection and repair policies of PFS checkers are complicated too. To make the measurement and comparison more accurate, we develop a unified taxonomy based on the high-level similarity of PFSes.

First, inspired by the classic memory consistency models [36], we propose a *PFS consistency model*, which describes the general principle that PFS checkers should ensure in order to maintain PFS integrity. This model is based on the observation that many policies can be described as relations between objects on MDT and objects on OSTs. Specifically, the PFS consistency model includes the following definition and consistency rules:

- *Consistency Group (CG)*: an MDT-object and all its associated child OST-objects form a CG;
- *CG-rule1*: every object in a CG should be consistent individually (e.g., has a valid format within the structure);
- *CG-rule2*: one MDT-object of a client directory maps to no child OST-object;
- *CG-rule3*: one MDT-object of a client file maps to at least one child OST-object;

- *CG-rule4*: one OST-object maps to one and only one parent MDT-object;
- *CG-rule5*: the mapping between a parent MDT-object and a child OST-object is bidirectional;
- *CG-rule6*: an object violating the above rules may only exist in a specified location (e.g., "`lost+found`").

If a corruption breaks one or more of the consistency rules, we expect the PFS checker to be able to detect the inconsistency. In case there is redundancy in PFS structures, the PFS checker may be able to repair the inconsistency; otherwise, the corrupted objects should be reclaimed to a specified location (e.g., "`lost+found`").

Based on the model, we propose a taxonomy to characterize the checker policies of PFSes meticulously, which includes multiple **D**etection and **R**epair levels. Detection levels describe the capability of the PFS checker to detect corruptions violating the consistency model:

- $\mathtt{D}_{abn.}$: the PFS checker behaves abnormally (e.g., hangs or crashes) without reporting detection results;
- $\mathtt{D}_{zero}$: the PFS checker finishes normally but misses the corruption, which may occur when the checker skips an entire data structure;
- $\mathtt{D}_{par.}$: the PFS checker partially detects the corruption, which may occur when the checker only examines partial bytes of a CG;
- $\mathtt{D}_{com.}$: the ideal case when the PFS checker detects the corruption completely.

Similarly, Repair levels describe the capability of the PFS checker to repair the corruptions:

- $\mathtt{R}_{wro.}$: the PFS checker fixes the corruption in a wrong way (e.g., a corrupted inode extended attribute is "repaired" by a wrong value).
- $\mathtt{R}_{zero}$: the PFS checker reports failure on repair; note that $\mathtt{D}_{zero}$ implies $\mathtt{R}_{zero}$.
- $\mathtt{R}_{par.}$: the PFS checker partially fixes the corruption;
- $\mathtt{R}_{com.}$: the PFS checker fixes the corruption completely according to the PFS consistency model.

Note that our current model and taxonomy may not cover all potential cases of PFS corruptions. For example, if MDT-object `A` mistakenly refers to MDT-object `B`'s child OST-object `b`, and `b` also mistakenly refers to `A`. Based on the model, a PFS checker may move MDT-object `B` to "`lost+found`", while `A` and `b` may legally refer to each other. And the PFS checker may be classified as $\mathtt{D}_{com.}$ and $\mathtt{R}_{com.}$. However, such corner cases require collaborative corruptions on multiple servers simultaneously, which is rare in practice. Therefore, we leave more sophisticated cases as future work.

IV. Experimental Results

In this section, we present the results of fingerprinting the policies of LFSCK (§IV-A) and BeeGFS-FSCK (§IV-B). Overall, we find that both checkers can detect and repair a variety of corruptions. However, there are still cases where the checkers behave sub-optimally (e.g., kernel panics, wrong repair, invalid option). All experiments were performed on a six-node cluster: one MGS/MGT, one MDS/MDT, three OSS/OSTs, and one client. A subset of the experiments have been repeated and verified on a Lustre profile and a BeeGFS profile on CloudLab [1]. The experiment scripts, corrupted images, and the CloudLab profiles are publicly available [8].

A. LFSCK Policy

LFSCK is a distributed checker with one master engine on MDT and slave engines on OSTs. All engines can examine the Lustre structures locally. Moreover, the master engine inquires metadata from OSTs to check global consistency with kernel threads. Table II summarizes the results of LFSCK using the taxonomy defined in §III. Overall, we find that LFSCK is able to detect and repair corruptions on a variety of structures completely (i.e., $\mathtt{D}_{com.}$ and $\mathtt{R}_{com.}$). We discuss a few sub-optimal cases in details below.

Kernel panic ($\mathtt{D}_{abn.}$). When the in-memory copy of the inode (stands for Ext4 inode here) of "MDT-object" or "OST-object" is inconsistent with the on-disk copy, LFSCK kernel threads may trigger kernel panics. This implies a potential gap in the contract between LFSCK and local file system: LFSCK assumes the object files exist, which may not always be true in practice (e.g., local inodes may be corrupted or reclaimed by `e2fsck` [4]). A more elegant policy could be verifying the existence of the file first. This abnormal behavior has been confirmed by Lustre developers, and has led to a new patch.

Skipping on-disk copy ($\star\mathtt{D}_{com.}$). For most data structures, LFSCK scans the on-disk copy and examines the consistency. However, for "OI table" and "LOV EA", LFSCK only checks the in-memory copy of the structure without examining the actual on-disk copy. As a result, the on-disk corruption may not be detected immediately.

Wrong repair ($\mathtt{R}_{wro.}$). LFSCK may use redundancy in Lustre structures to repair corrupted structures. However, in case of *junk* or *out-of-sync* in "FID on MDT", LFSCK may update the FID on MDT using a new value, but forget to update the corresponding PFID on OST. As a result, the mapping between the MDT-object and the OST-object is still invalid.

Unbounded repair ($+\mathtt{R}_{com.}$). When "linkEA" is duplicated, LFSCK appears to be able to repair the structure initially. However, we observe that when the structure is corrupted and repaired repeatedly, linkEA will be extended with extra bytes and LFSCK will eventually hang.

B. BeeGFS-FSCK Policy

Different from LFSCK, BeeGFS-FSCK loads structures to a SQLite database and checks BeeGFS using SQL queries. As shown in Table II, BeeGFS-FSCK can also detect or repair diverse corruptions (i.e., $\mathtt{D}_{com.}$ and $\mathtt{R}_{com.}$), We discuss a few sub-optimal cases below.

Wrong repair ($\mathtt{R}_{wro.}$). BeeGFS uses two empty files (i.e., `dentry-by-name` and `dentry-by-ID`) to represent client file's name and its data chunk ID on OST. As shown in Fig. 2, `dentry-by-name` and `dentry-by-ID` share the same inode via hard link [5]. However, when the "fghfs" extended

TABLE II: **Summary of Checker Policy.** This table summarizes the checker policies of LFSCK and BeeGFS observed in our experiments based on the detection (D) and repair (R) levels defined in §III. Green color means the corruption is detected or repaired completely. Red color means the checker has abnormal behaviors or repairs the corruption in a wrong way. "*" means the checker only checks the in-memory copy of the structure. "+" means the checker will hang if the structure is corrupted and repaired repeatedly. "—" means the fault is not applicable to the structure.

Lustre Structures	*junk*	*zero*	*duplicate*	*out-of-sync*
MDT-object	—	—	—	$D_{abn.}$|R_{zero}
OST-object	D_{zero}|R_{zero}	D_{zero}|R_{zero}	—	$D_{abn.}$|R_{zero}
llog record	D_{zero}|R_{zero}	D_{zero}|R_{zero}	—	D_{zero}|R_{zero}
FID on MDT	$D_{com.}$|$R_{wro.}$	$D_{com.}$|R_{zero}	$D_{com.}$|$R_{wro.}$	$D_{com.}$|$R_{wro.}$
FID on OST	$D_{com.}$|R_{zero}	$D_{com.}$|R_{zero}	$D_{com.}$|R_{zero}	D_{zero}|R_{zero}
FLDB	D_{zero}|R_{zero}	D_{zero}|R_{zero}	—	D_{zero}|R_{zero}
OI table	$*D_{com.}$|R_{zero}	$*D_{com.}$|R_{zero}	—	$*D_{com.}$|$R_{com.}$
LOV EA	$*D_{com.}$|$R_{com.}$	$*D_{com.}$|$R_{com.}$	$*D_{com.}$|$R_{com.}$	$*D_{com.}$|$R_{com.}$
PFID	$D_{par.}$|$R_{par.}$	$D_{com.}$|$R_{com.}$	$D_{com.}$|$R_{com.}$	D_{zero}|R_{zero}
linkEA	$D_{com.}$|$R_{com.}$	$D_{com.}$|$R_{com.}$	$D_{com.}$|$+R_{com.}$	$D_{com.}$|$R_{com.}$
nlink	D_{zero}|R_{zero}	$*D_{com.}$|R_{zero}	—	—
BeeGFS Structures	*junk*	*zero*	*duplicate*	*out-of-sync*
dentry-by-name (MDT-object)	—	—	—	$D_{com.}$|$R_{com.}$
dentry-by-ID (MDT-object)	—	—	—	$D_{com.}$|$R_{com.}$
fhgfs (MDT-object)	$D_{com.}$|$R_{wro.}$	$D_{com.}$|$R_{wro.}$	$D_{com.}$|R_{zero}	$D_{com.}$|$R_{wro.}$
chunk (OST-object)	D_{zero}|R_{zero}	D_{zero}|R_{zero}	—	$D_{par.}$|R_{zero}
content directory	—	—	—	$D_{com.}$|$R_{com.}$
nlink	$*D_{com.}$|R_{zero}	$*D_{com.}$|R_{zero}	—	—
size	$*D_{com.}$|R_{zero}	$*D_{com.}$|R_{zero}	—	—

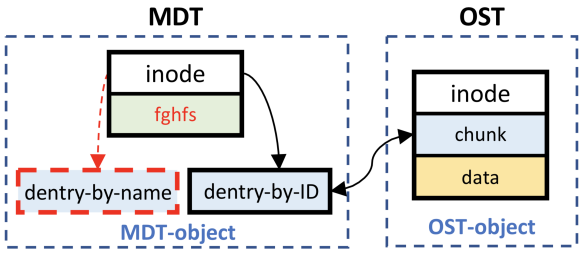

Fig. 2: **Example of wrong repair operation on BeeGFS.** White/green/blue/yellow colors represent inode, extended attribute, regular file, and user data, respectively.

attribute of the inode is corrupted, BeeGFS-FSCK will remove the intact `dentry-by-name`. As a result, the client data cannot be accessed.

Invalid option (R_{zero}). BeeGFS-FSCK seems to be more user-friendly as it often prompts to users and provides repairing options. However, the specified option may be ineffective. For example, in case of *junk* or *zero* on "nlink" or "size", BeeGFS-FSCK offers to update the corresponding structure, but we find that the structure is not actually updated.

C. Need for More Complete Test Cases

Both Lustre and BeeGFS have built-in test suites for their checkers. For example, there are test cases designed for Lustre's `FID`, `nlink`, and BeeGFS' `chunk`, `nlink`, `size`. However, we still observe sub-optimal checking of these structures. This implies that existing test suites are not enough, which is consistent with previous studies [17].

V. Discussion & Future Work

The work presented in this paper suggests many opportunities for further improvements. We discuss two major directions in this section.

Scalability & Automation. Our current study involves much manual effort (e.g., identifying PFS metadata types). To make the methodology more scalable, we are building a framework named PCheck. We are exploring static analysis to identify PFS metadata structures precisely. Also, to generate more comprehensive PFS states, we are exploring the fuzzing technique [31], [45] and more HPC workloads. Our goal is to develop PCheck into a generic framework that can fingerprint the checker policies of diverse PFSes thoroughly and efficiently, including PFSes without using local file systems (e.g., IBM Spectrum Scale).

Enhancing PFS Checkers. Based on the fingerprinting results, we are exploring methods to improve state-of-the-art PFS checkers. We will enrich the proposed PFS consistency model and taxonomy to serve as a foundation for designing complete checker policies. We plan to collaborate with PFS developers and the community to explore new designs, so that the robustness of PFSes can be improved to the next level.

VI. Acknowledgments

We thank the anonymous reviewers for their insightful feedback. We also thank Andreas Dilger for discussion on the kernel panic problem. This work was supported in part by NSF under grants CNS-1566554/1855565, 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] CloulLab. In *https://www.cloudlab.us/p/ISU-Cloud/pcheck*.

[2] DDN acquires Lustre business from Intel. In *https://datacentrenews.eu/story/ddn-acquires-lustre-business-intel-significant-investment-come*.

[3] debugfs. In *https://man7.org/linux/man-pages/man8/debugfs.8.html*.

[4] e2fsck. In *https://linux.die.net/man/8/e2fsck*.

[5] hardlink. In *https://man7.org/linux/man-pages/man1/hardlink.1.html*.

[6] LIVERMORE COMPUTING CENTER . In *https://hpc.llnl.gov/hardware/file-systems/parallel-file-systems*.

[7] Lustre Metadata Service (MDS) . In *http://wiki.lustre.org/*.

[8] pcheck. In *https://git.ece.iastate.edu/data-storage-lab/prototypes/pcheck*.

[9] RAID. In *https://en.wikipedia.org/wiki/RAID*.

[10] xfs-check. In *https://linux.die.net/man/8/xfs_check*.

[11] PFS corruption after upgrading from SQL Server 2014. In *https://www.sqlskills.com/blogs/paul/pfs-corruption-after-upgrading-from-sql-server-2014/*, 2014.

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

[13] Lustre software release 2.x operations manual, 2017.

[14] Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau. Information and control in gray-box systems. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, page 43–56, New York, NY, USA, 2001. Association for Computing Machinery.

[15] Remzi H. Arpaci-Dusseau. Operating systems: Three easy pieces. 2017.

[16] BeeGFS File System Check (beegfs fsck). https://www.beegfs.io/wiki/fscheck.

[17] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. Exe: Automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS)*, 2006.

[18] Jinrui Cao, Om Rameshwar Gatla, Mai Zheng, Dong Dai, Vidya Eswarappa, Yan Mu, and Yong Chen. PFault: A General Framework for Analyzing the Reliability of High-Performance Parallel File Systems. In *Proceedings of the 2018 International Conference on Supercomputing (ICS)*, 2018.

[19] Jinrui Cao, Simeng Wang, Dong Dai, Mai Zheng, and Yong Chen. A generic framework for testing parallel file systems. In *2016 1st Joint International Workshop on Parallel Data Storage and data Intensive Scalable Computing Systems (PDSW-DISCS)*, 2016.

[20] Joao Carreira, Rodrigo Rodrigues, George Candea, and Rupak Majumdar. Scalable Testing of File System Checkers. In *EuroSys12*, 2008.

[21] Alexander Conway, Ainesh Bakshi, Yizheng Jiao, William Jannen, Yang Zhan, Jun Yuan, Michael A Bender, Rob Johnson, Bradley C Kuszmaul, Donald E Porter, et al. File systems fated for senescence? nonsense, says science! In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST)*, pages 45–58. USENIX Association, 2017.

[22] Haryadi S. Gunawi et. al., Riza O. Suminto, Russell Sears, Casey Golliher, Swaminathan Sundararaman, Xing Lin, Tim Emami, Weiguang Sheng, Nematollah Bidokhti, Caitie McCaffrey, ary Grider, Parks M. Fields, Kevin Harms, Robert B. Ross, Andree Jacobson, Robert Ricci, Kirk Webb, Peter Alvaro, H. Birali Runesha, Mingzhe Hao, and Huaicheng Li. Fail-Slow at Scale: Evidence of Hardware Performance Faults in Large Production Systems. In *Proceedings of the 16th USENIX Conference on File andStorage Technologies (FAST)*, 2010.

[23] Philip Schwan et al. Lustre: Building a file system for1000-node clusters. In *InProceedings of the 2003 Linuxsymposium*, pages 380–386, 2003.

[24] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Redundancy does not imply fault tolerance: Analysis of distributed storage reactions to file-system faults. *ACM Trans. Storage*, 13(3), September 2017.

[25] Om Rameshwar Gatla, Muhammad Hameed, Mai Zheng, Viacheslav Dubeyko, Adam Manzanares, Filip Blagojević, Cyril Guyot, and Robert Mateescu. Towards Robust File System Checkers. In *16th USENIX Conference on File and Storage Technologies (FAST)*, 2018.

[26] Om Rameshwar Gatla and Mai Zheng. Understanding the fault resilience of file system checkers. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2017.

[27] Om Rameshwar Gatla, Mai Zheng, Muhammad Hameed, Viacheslav Dubeyko, Adam Manzanares, Filip Blagojevic, Cyril Guyot, and Robert Mateescu. Towards robust file system checkers. *ACM Trans. Storage (TOS)*, 2018.

[28] Haryadi S. Gunawi, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi H. SQCK: A Declarative File System Checker. In *OSDI'08: Proceedings of the 8th USENIX conference on Operating systems design and implementation*, 2008.

[29] Jan Heichler. An introduction to BeeGFS. 2014.

[30] HPC User Site Census. 2016.

[31] Seulbae Kim, Meng Xu, Sanidhya Kashyap, Jungyeon Yoon, Wen Xu, and Taesoo Kim. Finding semantic bugs in file systems with an extensible fuzzing framework. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 147–161, New York, NY, USA, 2019. Association for Computing Machinery.

[32] Jharrod LaFon, Satyajayant Misra, and Jon Bringhurst. On distributed file tree walk of parallel file systems. In *InProceedings of the International Conference on HighPerformance Computing, Networking, Storage and Anal-ysis (SC'12)*, 2012.

[33] LFSCK: an online file system checker for Lustre. 2017.

[34] Lustre Software Release 2.x: Operations Manual. 2017.

[35] Ao Ma, Chris Dragga, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. ffsck: The Fast File System Checker. In *Proceedings of the 11th USENIX conference on File and Storage Technologies*, 2013.

[36] David Mosberger. Memory consistency models. *SIGOPS Oper. Syst. Rev.*, 27(1):18–26, January 1993.

[37] Sarp Oral, Feiyi Wang, David Dillow, Galen Shipman, Ross Miller, and Oleg Drokin. Efficient object storagejournaling in a distributed parallel file system. . In *InPro-ceedings of the 8th USENIX Conference on File andStorage Technologies (FAST'10)*, 2010.

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

[39] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. IRON File Systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, 2005.

[40] Kai Ren, Qing Zheng, Swapnil Patil, and Garth Gibson. IndexFS: Scaling file system metadata performance withstateless caching and bulk insertion. In *Proc. of the Intl. Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2014.

[41] Jinghan Sun, Chen Wang, Jian Huang, and Marc Snir. Understanding and Finding Crash-Consistency Bugs in Parallel File Systems. In *HotStorage*, 2020.

[42] Ceph File System. http://docs.ceph.com/docs/master/.

[43] OrangeFS Team. The OrangeFS project. 2015.

[44] Erci Xu, Mai Zheng, Feng Qin, Yikang Xu, and Jiesheng Wu. Lessons and Actions: What We Learned from 10K SSD-Related Storage System Failures. In *2019 USENIX Annual Technical Conference (ATC)*, 2019.

[45] Wen Xu, Hyungon Moon, Sanidhya Kashyap, Po-Ning Tseng, and Taesoo Kim. Fuzzing file systems via two-dimensional input space exploration. *2019 IEEE Symposium on Security and Privacy (SP)*, pages 818–834, 2019.

[46] Fan Yong. Lustre consistency verification, 2014.

[47] Mai Zheng, Joseph Tucek, Dachuan Huang, Feng Qin, Mark Lillibridge, Elizabeth Yang, Bill Zhao, and Shashank Singh. Torturing Databases for Fun and Profit. In *Proceedings of 11th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2014.

[48] Mai Zheng, Joseph Tucek, Feng Qin, and Mark Lillibridge. Understanding the Robustness of SSDS under Power Fault. In *Proceedings of 11th USENIX Conference on File and Storage Technologies (FAST)*, 2013.

[49] Mai Zheng, Joseph Tucek, Feng Qin, Mark Lillibridge, Bill W. Zhao, and Elizabeth S. Yang. Reliability analysis of ssds under power fault. *ACM Trans. Comput. Syst. (TOCS)*, 2016.