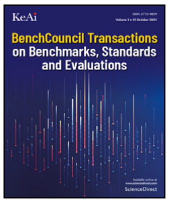
Contents lists available at ScienceDirect

BenchCouncil Transactions on Benchmarks, Standards and Evaluations

journal homepage: https://www.keaipublishing.com/en/journals/benchcouncil-transactions-on-benchmarks-standards-and-evaluations/



Benchmarking for Observability: The Case of Diagnosing Storage Failures

Duo Zhang, Mai Zheng [*]

Department of Electrical and Computer Engineering, Iowa State University, Ames, IA, USA

ARTICLE INFO

Keywords:
Storage systems
Failure diagnosis
Observability
Reproducibility
Reliability
Robustness
Benchmarking
Debugging
Tracing
Record and replay

ABSTRACT

Diagnosing storage system failures is challenging even for professionals. One recent example is the "When Solid State Drives Are Not That Solid" incident occurred at Algolia data center, where Samsung SSDs were mistakenly blamed for failures caused by a Linux kernel bug. With the system complexity keeps increasing, diagnosing failures will likely become more difficult.

To better understand real-world failures and the potential limitations of state-of-the-art tools, we first conduct an empirical study on 277 user-reported storage failures in this paper. We characterize the issues along multiple dimensions (e.g., time to resolve, kernel components involved), which provides a quantitative measurement of the challenge in practice. Moreover, we analyze a set of the storage issues in depth and derive a benchmark suite called $BugBench^k$. The benchmark suite includes the necessary workloads and software environments to reproduce 9 storage failures, covers 4 different file systems and the block I/O layer of the storage stack, and enables realistic evaluation of diverse kernel-level tools for debugging.

To demonstrate the usage, we apply $BugBench^k$ to study two representative tools for debugging. We focus on measuring the observations that the tools enable developers to make (i.e., observability), and derive concrete metrics to measure the observability qualitatively and quantitatively. Our measurement demonstrates the different design tradeoffs in terms of debugging information and overhead. More importantly, we observe that both tools may behave abnormally when applied to diagnose a few tricky cases. Also, we find that neither tool can provide low-level information on how the persistent storage states are changed, which is essential for understanding storage failures. To address the limitation, we develop lightweight extensions to enable such functionality in both tools. We hope that $BugBench^k$ and the enabled measurements will inspire follow-up research in benchmarking and tool support and help address the challenge of failure diagnosis in general.

1. Introduction

The storage stack in the Linux kernel is witnessing a sea-change driven by the advances in non-volatile memory (NVM) technologies [1–13]. For example, the SCSI subsystem and the Ext4 file system, which have been optimized for hard disk drives (HDDs) for decades, are adding multi-queue support [14–16] and DAX support [17] for flash-based solid state drives (SSDs) and persistent memories (PMs), respectively. While such modifications may offer higher performance in general, the implications on system reliability is much less measured or understood.

One real-world example is the "When Solid-State Drives Are Not That Solid" incident occurred in Algolia data center [18], where a random subset of SSD-based servers crashed and corrupted files for unknown reasons. The developers "spent a big portion of two weeks just isolating machines and restoring data as quickly as possible". After trying to diagnose almost all software in the stack (e.g., Ext4, Software RAID [19]), they finally (mistakenly) concluded that it was Samsung's SSDs to blame. Samsung's SSDs were criticized and blacklisted, until one month later Samsung engineers found that it was a TRIM-related Linux kernel bug that caused the failure [20]. Similar confusing failures will likely increase in the foreseeable future as the system complexity keeps increasing [21–24].

Addressing the grand challenge will require cohesive efforts from the communities. Among others, a better understanding of real-world failure incidents and the potential limitations of state-of-the-art tools is critical. To this end, we first conduct an empirical study on 277 real-world storage failure issues in this paper. We characterize the issues along multiple dimensions (e.g., time to resolve, kernel components involved, hardware dependency), which enables us to quantitatively measure the reliability challenge as well as the need for better solutions.

Moreover, we analyze a set of storage issues in depth. By examining the user reports and bug patches meticulously and experimenting on real storage systems, we derive the necessary conditions (e.g., user/workload operations, library/kernel versions, system configurations) for triggering the failures deterministically. At the time of this

* Corresponding author.
E-mail addresses: duozhang@iastate.edu (D. Zhang), mai@iastate.edu (M. Zheng).

https://doi.org/10.1016/j.tbench.2021.100006
Received 6 August 2021; Received in revised form 11 October 2021; Accepted 20 October 2021
Available online 12 November 2021

writing, we are able to reproduce nine cases successfully, which covers four different file systems as well as the low-level block I/O layer of the Linux storage stack.

Based on the reproducible cases, we create a benchmark suite called $BugBench^k$, which includes a set of portable virtual machine (VM) images containing all the necessary software programs and environments to reproduce the nine storage failures caused by kernel-level bugs.[1] Complementary to existing benchmark suites which are mostly designed for measuring the performance [26–29], $BugBench^k$ enables realistic evaluation on the capability of diverse reliability tools (e.g., kernel bug detectors [30,31], tracers [32], record & replay tools [33]), which is critical for identifying the potential limitations as well as the opportunities for further improvement.

To demonstrate the usage, we leverage $BugBench^k$ to analyze two representative tools for debugging: (1) *FTrace*, the Linux kernel internal tracer [32]; and (2) *PANDA*, a VM-based record & replay tool [33]. Different from existing studies which mostly measure the tools' runtime overhead [34], we focus on measuring the *observability*, which means the observations that the tool allows the developers to make in order to diagnose the failure symptoms [35]. While the basic concept of observability is not new [35], we derive a set of concrete metrics to quantitatively and/or qualitatively measure the observability based on $BugBench^k$. Our experiments demonstrate the different design tradeoffs of the tools in terms of debugging information and space overhead. More importantly, we find that there are multiple tricky failure cases where both tools may fail to function properly. In other words, the usage of the tools may introduce interference to the target storage stack and make the failure symptom un-reproducible.

In addition, we find that neither tool can directly provide low-level information (e.g., storage device commands) on how the persistent storage states are changed, which is crucial for understanding host-device interactions in the storage stack. To address the limitation, we explore different ways to extend both FTrace and PANDA, and shows that it is possible to enhance both of them with such low-level observability without relying on special hardware (e.g., bus analyzer [36,37]).

It is well acknowledged that effective benchmark suites and tools are essential for improving various computer systems; on the other hand, building effective benchmark suites and tools is a long-term, iterative process that requires cohesive efforts from broad communities [25,34,38]. To the best of our knowledge, this work is the first effort to benchmark the observability of debugging tools with concrete metrics. We hope the $BugBench^k$ prototype as well as the initial efforts demonstrated in this paper will inspire follow-up research on benchmarking and tool support for reliability, and help improve computer systems in general.[2]

The rest of the paper is organized as follows: in Section 2, we describe the background and extended motivation; in Section 3, we characterize real-world storage failures and derives the $BugBench^k$; in Section 4, we measure the observability of FTrace and PANDA based on $BugBench^k$ and describe our extensions; in Section 5, we discuss related work; finally, we conclude the paper in Section 6.

2. Background & motivation

2.1. The storage stack & reliability challenge

Fig. 1 shows the typical storage stack,[3] which traditionally includes several major layers such as file systems, the block I/O layer, and device drivers. The recent introduction of PM technologies can provide access latencies less than 3X of DRAM while maintaining durability

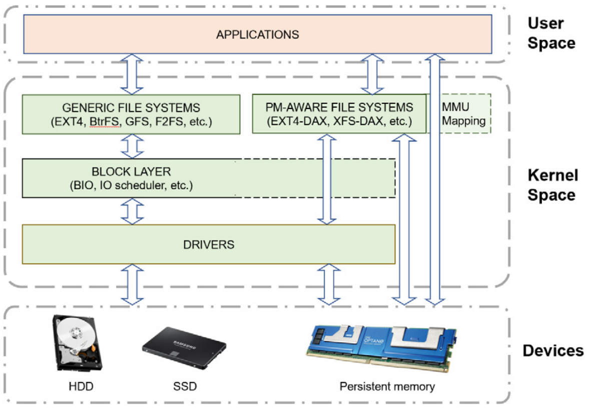

APPLICATIONS
User Space
GENERIC FILE SYSTEMS (EXT4, BtrFS, GFS, F2FS, etc.)
PM-AWARE FILE SYSTEMS (EXT4-DAX, XFS-DAX, etc.)
MMU Mapping
BLOCK LAYER (BIO, IO scheduler, etc.)
Kernel Space
DRIVERS
HDD
SSD
Persistent memory
Devices

Fig. 1. The Storage Stack. The kernel-level software modules (green) are the major focus of this work.
Source: Adapted from [39].

guarantee [40], which blurs the line between the storage management and the memory management in the kernel. Consequently, the memory management subsystem is also becoming part of the storage stack for persistent data.

The storage system is notoriously complex and difficult to get right despite decades of efforts [41–44]. Moreover, almost all layers in the storage stack are being optimized aggressively in recent years. For example, SSDs and PMs are replacing HDDs as the durable device [10–13, 18,45]; NVMe [46] and CXL [47] are redefining the host-device interface; *blk-mq* [48] alleviates the single queue and lock contention bottleneck at the block I/O layer; the SCSI subsystem and the Ext4 file system are being adapted for NVM (e.g., scsi-mq [14–16] and DAX [17]); in addition, various NVM-oriented new designs/optimizations have been proposed (e.g., F2FS [49], NOVA [50], Kevlar [51]), some of which require cohesive modifications throughout the storage stack (e.g., the TRIM support [52]). Such modifications could potentially introduce various software bugs leading to system failures [41,53].

In practice, storage failures may occur due to various reasons including but not limited to software bugs [41–43,53], power outages [41, 54], device errors [42,55,56], etc. Once a failure occurs, it is often difficult to diagnose the root cause due to the complexity of the storage stack, as demonstrated in the Algolia incident described in Section 1. However, despite the various anecdotes, there is little quantitative measurement or understanding of the characteristics of storage failures occurred in the real world. We attempt to address the issue in this work.

2.2. Debugging tools

Many tools have been built to improve system reliability. For example, *testing* tools (e.g., model checkers [41], fuzzers [30,31,57], fault injectors [58–61]) focus on triggering the potential bugs in target systems in a controlled testing environment *before* deployment to reduce the possibility of real-world failures. Once a system failure occurred in practice, however, testing tools can help little for pinpointing the root causes due to the different environments and assumptions.

Another category is *debugging* tools, which aims to facilitate diagnosing the root causes *after* a failure occurred in practice. While a benchmark suite containing real world cases may be used for evaluating both testing tools and debugging tools, we focus on debugging tools in this paper because: (1) they are much less studied compared to the abundant efforts on testing tools; (2) they are much needed in diagnosing real-world failure incidents. We classify existing debugging tools into three types as follows:

Interactive Debuggers. This category includes classic debuggers such as GDB/KDB/KGDB [62–64], which represents the *de facto* way to

[1] Coincidentally, there is an early work on application-level BugBench [25]; we elaborate on the difference and the synergy in Section 5.

[2] We release $BugBench^k$ publicly on GitLab: https://git.ece.iastate.edu/data-storage-lab/prototypes/bugbench.

[3] Adapted from SNIA NVM Programming Model [39].

diagnose software system failures. They usually support fine-grained manual inspection (e.g., setting breakpoints at specific statements, checking the values of specific memory bytes). However, significant human efforts and expertise are needed to harness the power and to diagnose the complicated storage stack efficiently. Such traditional debugging methods are arguably not scalable, because the required manual effort and experience will keep increasing as the system becomes more and more complex. More automation and/or intelligence are probably needed to make debugging scalable.

Software & Hardware Tracers. Software tracers [32,65–69] can collect various events from a running target system automatically, which are typically implemented via dynamic instrumentation. The traced logs can help understand the system anomalies (among other purposes), which are often invaluable for failure diagnosis. Similarly, bus analyzers [36,37] are hardware equipments that can capture the low-level communication data (e.g., SCSI commands [70]) between the storage software and the device. However, since they only trace bus-level information, they cannot help much on understanding system-level behaviors.

Record & Replay Tools. Record & replay tools [33,71] have been applied to debugging for both user-level applications and the kernel. Typically, these tools leverage virtual machines to run the target software stack as a whole. Meanwhile, they record system snapshots as well as non-deterministic events (e.g., interrupts) to ensure replaying system execution faithfully. Developers can replay the recorded whole system execution logs repeatedly without the needs of re-running the workloads. Also, it is possible to integrate record & replay tools with interactive debuggers (e.g., GDB) to perform traditional debugging (e.g., setting breakpoints) during the replay. Additional analysis passes may also be implemented based on the record & replay mechanism (e.g., plugins in PANDA [33]).

Note that all of the three types of tools mentioned above are important debugging tools widely used in practice. But to the best of our knowledge, there is little quantitative measurement of their debugging capability. We attempt to address the deficiency in this work. We focus on software tracers and record & replay tools because they enable different degrees of automation for failure diagnosis, which we believe is critically important for a scalable debugging methodology. We leave the measurement of interactive debuggers (which requires manual effort/expertise that is difficult to quantify) and hardware tracers (which requires special hardware) as future work.

2.3. *Observability of debugging tools*

A few researchers have studied and benchmarked debugging tools [34] due to their prime importance in practice. However, existing studies mostly focus on the performance (e.g., runtime overhead) instead of their effectiveness, largely due to the lack of reliability benchmark suites.

Complementary to the existing efforts, we focus on the effectiveness of failure diagnosis. Specifically, we propose to measure the *observability* of debugging tools, which is a concept proposed recently for improving system reliability [35]. The *observability* includes three desired properties (i.e., visibility, repeatability, and expressibility) for debugging failures. Intuitively, the concept describes the observations that a tool allows the developers to make [35], which is critically important for debugging. While the concept of observability is well known, there is no practical methodologies to measure it to the best of our knowledge. We demonstrate how to measure the observability using realistic cases and concrete metrics in this work.

3. Characterization of storage failures

In this section, we describe how we collect the storage failure dataset (Section 3.1); the overall characteristics of the dataset (Section 3.2); and how we derive the $BugBench^k$ (Section 3.3).

Table 1
Overview of storage issues on Bugzilla.

Group	Count (%)	Avg. Days	Avg. Comments/ Participants
Resolved	136 (49.1%)	146.9	8/3
Unresolved	141 (50.9%)	1444.2	5/2
Overall	277	807.3	6/2

3.1. *Methodology*

To understand the characteristics of real-world problems of the storage stack, we collect failure issues reported by the end users from Linux Bugzilla [72]. We choose this platform because it is one major venue for regular users to report encountered failures to the Linux kernel community, and the reported issues are typically examined by the kernel developers with detailed status updates. Since the platform includes issues of the entire Linux kernel which is beyond the scope of the storage stack, we apply the following methods to refine the dataset.

First, Bugzilla organizes the issues based on major kernel components (e.g., "Process Management", "Networking"), so we search for the issues tagged with storage-related components (e.g., "File Systems", "IO/Storage", "Memory Management", "Drivers") or generic components (i.e., "Others"); also, the time of the issues is limited to the recent ten years. Next, in order to identify a manageable and important subset for study, we refine the dataset further by using a set of keywords implying severe failure consequences (e.g., "data loss", "corrupt"). Moreover, for the "Other" category, we further analyze the issues manually based on our domain knowledge and only keep storage-related ones (e.g., keeping software RAID issues but excluding GPU buffer corruptions). The resulting dataset contains 277 issues in total, which represents a subset of severe storage failures experienced by Linux end users. Note that this method is similar to the keyword search in previous studies [53,73].

Threats to validity. The characterization results presented in this section should be interpreted with the methodology in mind. In particular, the dataset was refined via critical keywords and manual efforts, which might be incomplete. Also, only a limited subset of Linux end users are aware of Linux Bugzilla and only a limited subset of them would report issues. Therefore, it is likely that there are more storage-related issues occurred in the wild but not captured in this study. Nevertheless, we believe our effort is one important step toward addressing the challenge.

3.2. *Overall characteristics*

Bugzilla maintains various status tags for the issues reported (e.g., "new", "closed"). For simplicity, we classify the issues into two groups based on their status tags: (1) *Resolved*, which includes issues with the "resolved" or "closed" status; (2) *Unresolved*, which includes issues with "new", "assigned", "reopened", or "verified" status. We summarize the two groups in Table 1. The second column shows the number of issues (and the percentage) in each group. The third column shows the average duration of the issues in days. For the Resolved group, the duration is calculated based on the report date and the date of the last comment; for the Unresolved group, it is the period between the report date and the time of this writing. The last column shows the average numbers of comments and participants in resolving the issues. We make multiple observations as follows:

First, *the issues took multiple months to resolve on average* (e.g., 146.9 days for the Resolved group in Table 1), and the diagnosis process typically involve multiple rounds of discussions and multiple people (e.g., 8 comments and 3 participants for the Resolved group), which quantitatively suggests the difficulty of diagnosing storage failures.

Second, *the issues involve all major components in the storage software stack*. As shown in Fig. 2, both Resolved and Unresolved groups span

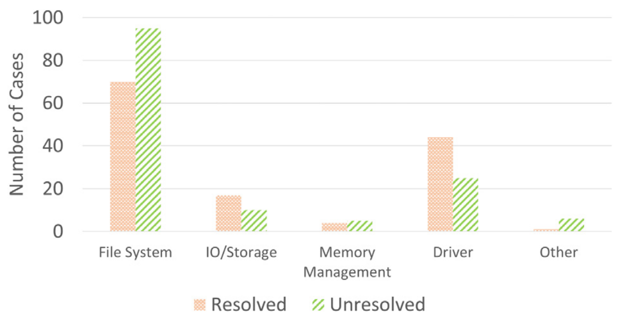


Fig. 2. Distributions of Resolved (orange) and Unresolved (green) Issues across Different Storage Components.

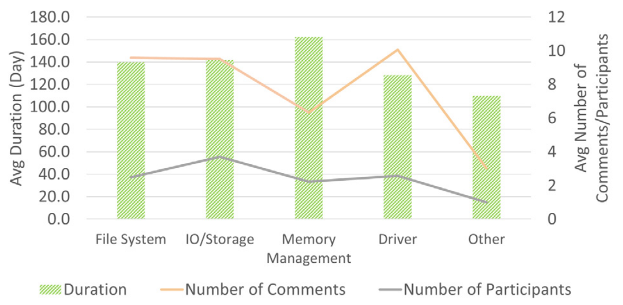


Fig. 3. Characteristics of Resolved Issues across Storage Components in terms of Average Duration (green bar), Average Number of Comments (orange line) and Average Number of Participants (gray line).

over all the five storage components studied. This implies that an ideal debugging tool should provide the full-stack observability. In particular, "File System" and "Driver" contains the most issues reported, which is consistent with previous studies [53].

Third, *for the resolved issues, the average debugging time is consistently long across components*. As shown in Fig. 3, the average debugging time of all five components is more than 100 days. This implies that due to the complexity of the storage stack, no single component is particularly easier to diagnose, which again suggests the importance of capturing the full-stack observability for debugging tools.

Fourth, *37 out of 136 (26.3%) resolved issues involve multiple OS distributions or kernel versions*. The manifestation symptoms of the issues often differ on different systems, which suggests that the software environment (e.g., kernels, libraries) is critically important for reproducing the failures for diagnosis.

Fifth, *only 5 out of 136 (3.7%) resolved issues were caused by hardware*. This implies that software remains the major source of storage failures, which is consistent with previous studies [74]. Also, it suggests that observing the behavior of the storage software stack is critically important for failure diagnosis.

3.3. *BugBench*^{k}

To identify the limitations of state-of-the-art debugging tools as well as the opportunities for further improvement, it is necessary to have a set of *reproducible* failure cases, so that we can apply the target tools and conduct the measurement. To this end, we analyze a set of storage failure issues in depth, identify the specific conditions required to trigger the issues (e.g., user/workload operations, software libraries involved, Linux kernel versions and configurations), and attempt to reproduce the cases on our server machines. This turns out to be a challenging and time-consuming process due to the complexity of the Linux kernel as well as the diversity of the Linux end users' system setups. For example, the reproducing procedure typically requires finding and (re)compiling specific versions of the Linux kernel with non-default configurations, pulling specific software packages which may not be well maintained, deriving workload programs to emulate various users' inputs, etc. At the time of this writing, we have identified 61 cases with relatively complete information for reproducing, and we are able to reproduce 3 cases successfully in our environment. This first-hand experience further confirms the challenge of failure diagnosis and the needs for a readily reproducible benchmark suite.

To ensure that the cases can be reliably reproduced and to enable easy sharing of the reproduced cases in the communities, we package all the required software programs and system environments in VM images. We create two VM images for each of the successfully reproduced cases: the first VM image contains the buggy storage stack and the necessary workload programs, libraries, etc. for reproducing the case; the second VM image contains the patched kernel (i.e., the bug in the storage stack has been fixed by the corresponding patch) to serve as a reference for verification.

In addition, to improve the case count as well as the diversity of the reproducible cases, we collect additional storage-related bug cases from the Linux mailing lists [34]. The cases reported on the Linux mailing lists are often discovered by the developers directly during the internal regression testing, so they may not contain the same information as the issues reported by the end users on Bugzilla (e.g., no user experienced consequences, user environments, or resolving status). Such developer-discovered cases are relatively less valuable for characterizing the real-world failure impact or diagnosis difficulty (as in Section 3.2). However, these cases are still realistic in that they may affect Linux distributions released earlier (i.e., before the bug patch). In other words, if they can be reproduced readily, they are as valuable as the Bugzilla issues for measuring debugging tools and other reliability utilities. At the time of this writing, we are able to reproduce 6 storage-related cases from the Linux mailing list successfully. We also create the corresponding VM images for the 6 cases to facilitate future reproducible research.

Based on the 9 reproducible cases (i.e., 3 from Bugzilla and 6 from the Linux mailing list), we have created a benchmark suite called *BugBench*^{k}, which includes a set of VM images containing all the necessary workloads and system environments/configurations to reproduce the 9 cases. We summarize the 9 cases in Table 2. As shown in the table, the current prototype of *BugBench*^{k} covers 4 different file systems, including 2 cases for Ext4 (i.e., "1-EXT4", "2-EXT4"), 3 cases for Btrfs file system (i.e., "3-BTRFS", "4-BTRFS", "5-BTRFS"), 1 case for F2FS (i.e., "6-F2FS"), and 1 case for GFS2 (i.e., "7-GFS"). Moreover, there are 2 cases for the low-level block I/O layer of the storage stack (i.e., "8-BLK" and "9-BLK").

The "Critical Function" column in Table 2 shows the major kernel functions that are identified as problematic for each case. We can see that the number of critical functions ranges from 1 to 7 (in "6-F2FS"), depending on the complexity of the bug fixes.

The root causes of the 9 cases can be classified into either "Semantics" bugs (7 cases) or "Memory" bugs (2 cases) based on the bug patterns defined in the literature [53,73,75]. Unlike other types of bugs that have well-studied patterns to understand (e.g., deadlocks, data races), "Semantics" bugs is among the hardest issues in practice because they typically require deep understanding of system design logic to detect or diagnose. In other words, the cases included *BugBench*^{k} require sophisticated methodologies to diagnose effectively.

The "Bug Size" is defined as the sum of lines of insertion and deletion code (LoC) in the bug patch, which ranges from 6 (in "2-EXT4') to 121 (in "4-BTRFS") LOC depending on the complexity of the cases. The last column shows the language we used to implement the bug triggering workloads. We use Bash, C, or a combination of both to implement the workloads, depending on the input characteristics described in the user reports (for Bugzilla cases) or bug patches (for Linux mailing list cases).

Table 2
Overview of reproducible cases in *BugBench*^{k}.

Case ID	OS Image	Linux Kernel	Storage Component	Critical Function	Bug Type	Bug Size	Workload Language
1-EXT4	Ubuntu 20.04	v5.4.0	Ext4 File System	ext4_do_update_inode, ext4_isize_set, ext4_clear_inode_state, cpu_to_le16,cpu_to_le32, ext4_update_inode_fsync_trans	Semantics	8	C
2-EXT4	Ubuntu 20.04	v5.4.0	Ext4 File System	parse_options	Semantics	6	C & Bash
3-BTRFS	Ubuntu 16.04	v4.4.107	BTRFS File System	btrfs_ioctl_snap_destroy, btrfs_record_snapshot_destroy, btrfs_set_log_full_commit, check_parent_dirs_for_sync, btrfs_log_inode, btrfs_release_path	Semantics	71	C
4-BTRFS	Ubuntu 16.04	v4.4.107	BTRFS File System	btrfs_log_trailing_hole	Semantics	121	C
5-BTRFS	Ubuntu 20.04	v5.4.0	BTRFS File System	btrfs_log_all_parents, btrfs_log_inode, btrfs_must_commit_transaction, btrfs_record_unlink_dir	Semantics	13	C
6-F2FS	Ubuntu 16.04	v4.15.0	F2FS File System	f2fs_submit_page_bio, f2fs_is_valid_blkaddr, verify_block_addr, zero_user_segment,, validate_checkpoint, datalock_addr	Memory	94	C & Bash
7-GFS	Ubuntu 16.04	v4.4.0	GFS2 File System	gfs2_check_sb, fs_warn	Memory	18	Bash
8-BLK	Ubuntu 20.04	v5.4.0	Block Layer	blkdev_fsync, sync_blkdev	Semantics	12	C
9-BLK	Ubuntu 18.10	v4.19.1	Block Layer	_blk_mq_issue_directly, blk_mq_update_dispatch_busy, _blk_mq_requeue_request	Semantics	9	Bash

We choose Ubuntu to reproduce the cases by default because Ubuntu is one of the most well supported OS distributions for many Linux tools. Also, since many utilities and packages are outdated or even not usable on old kernels, we port the cases to the latest kernel (i.e., v5.4.0) when possible. For 5 cases (i.e., "3-BTRFS", "4-BTRFS", "6-F2FS", "7-GFS", "9-BLK"), we are not able to reproduce the cases in the latest kernel because the affected kernel structures have been changed significantly and the original problematic functions are no longer compatible with the latest kernel. Therefore, we have to reproduce them in relatively old versions (e.g., v4.4.107, v4.15.0, v4.4.0) where the cases are still reproducible.

To sum up, the current prototype of *BugBench*^{k} includes a set of VM images for reproducing 9 realistic storage failure cases. These reproducible cases, including the complete software workloads and environments encapsulated in VMs, enable us to measure and evaluate the effectiveness of tools conveniently. We demonstrate the usage of *BugBench*^{k} in the context of two representative debugging tools in the next section (Section 4).

4. Measuring the observability

In this section, we apply *BugBench*^{k} to study FTrace and PANDA, both of which are state-of-the-art tools widely used for debugging (among other usages). We find that *BugBench*^{k} can help measuring tools with completely different design principles. Also, we find that both FTrace and PANDA may provide useful information for the majority of the cases evaluated. On the other hand, both of them may behave abnormally when diagnosing some tricky cases. We elaborate on the experimental results of FTrace and PANDA in Section 4.1 and Section 4.2, respectively. In addition, we find that both tools fall short of providing low-level information on how the persistent states are changed. We discuss our extensions to improve their low-level observability in Section 4.3.

4.1. FTrace

FTrace is the Linux kernel internal tracer that has been included in the mainline Linux since v2.6.27 [32]. We measure the observability of its major feature (i.e., kernel function tracing) in this subsection.

Table 3 summarizes the results of applying FTrace to diagnose the 9 cases in *BugBench*^{k} (labeled from "1-EXT4" to "9-BLK" in the first column). The second column ("Still Reproducible") shows whether the bug cases can still be reproduced when enabling FTrace to trace the target storage stack. We can see that FTrace do not affect the reproducibility of any case. In other words, the tool is non-intrusive for debugging all the cases in *BugBench*^{k}.

The third column shows the total number of functions (Func.) traced by FTrace in each case, which may include duplicated entries if a kernel function is invoked multiple times during the workload execution. We run FTrace for three times and calculate the average count (e.g., "12,506" for "1-EXT4") and the range of variance (e.g., "±4.1%"). We can see that FTrace can generate a large amount of functions for most cases, ranging from "6867" (in "8-BLK" case) to "110,772,722" (in "9-BLK" case), which implies that the tool can provide rich function-level information for diagnosing the target system behavior.

Similarly, the fourth column shows the number of unique functions traced in each case (i.e., excluding duplicated entries), which is generally much smaller compared to the total number of functions traced (i.e., the third column). This implies that the same kernel functions may be invoked many times in all the failure cases. From debugging's perspective, the large redundancy in the trace could exacerbate the challenge of diagnosing system behavior.

The fifth column ("Critical Func. Observed") measures how many of the critical functions can be observed by FTrace in each case. As mentioned in Section 3.3, a critical function is a problematic kernel

Table 3
FTrace results on 9 *BugBench*^{k} cases.

Case ID	Still Reproducible?	Total # of Func. Traced	Total # of Unique Func.	Critical Func. Observed	Shortest Distance	Log Size (MB)
1-EXT4	Yes	12,506 (±4.1%)	1,152 (±0.7%)	1/7	–	2.07 (±0.01)
2-EXT4	Yes	54,796 (±2.3%)	1,436 (±15.9%)	0/1	2	9.17 (±0.03)
3-BTRFS	Yes	46,370 (±5.6%)	1,339 (±1.5%)	3/6	–	6.87 (±0.10)
4-BTRFS	Yes	92,476 (±5.5%)	1,381 (±1.0%)	0/1	1	14.1 (±0.43)
5-BTRFS	Yes	30,528 (±3.6%)	1,419 (±1.5%)	3/4	–	5.2 (±0.03)
6-F2FS	Yes	0	0	0/7	–	0
7-GFS	Yes	0	0	0/2	–	0
8-BLK	Yes	6,867 (±2.7%)	901 (±4.3%)	1/2	–	1.1 (±0)
9-BLK	Yes	110,772,722(±6.4%)	1,165(±0.8%)	2/3	–	7,496.2 (±153.1)

function that contributes to the storage failure. A failure case may have multiple critical functions as the root cause, depending on the complexity of the failure. We can see that although FTrace can trace many functions, it may not be able to capture the critical functions effectively for the cases in *BugBench*^{k}. For example, in "1-EXT4", there are 7 critical functions but only one of them can be captured by FTrace (i.e., "1/7"). Similarly, in four other cases (i.e., "3-BTRFS", "5-BTRFS", "8-BLK", and "9-BLK"), only partial critical functions can be observed (i.e., "3/6", "3/4", "1/2", and "2/3", respectively).

In terms of "2-EXT4" and "4-BTRFS", none of the critical functions in the two cases can be directly observed by FTrace (i.e., "0/1" in the fifth column for both cases). To measure the relevance of the traced functions in these two cases, we further calculate the "Shortest Distance" (the sixth column), which is defined by the minimum number of function invocations needed from the traced functions to the critical functions. We find that although FTrace misses the critical function in "4-BTRFS", it actually captures the parent function (i.e., the "Shortest Distance" is "1") correctly. Similarly, it captures the parent's parent function of the missing critical function in "2-EXT4" (i.e., the "Shortest Distance" is "2"). This implies that FTrace may still be helpful for diagnosing failures even if it may miss some specific functions.

To understand why FTrace may not be able to trace all critical functions for debugging the cases in *BugBench*^{k}, we look into the internals of FTrace. We find that FTrace relies on a pre-defined list for identifying traceable functions, which is stored in *available_filter_functions* file in the *debugfs* of the target system. Moreover, the default list may contain different functions on different kernel versions we evaluated. This list fundamentally limits the observability of FTrace for debugging diverse failure scenarios, as exposed by the incomplete critical functions in the fifth column ("Critical Func. Observed"),

In terms of "6-F2FS" and "7-GFS", the two cases are still reproducible with FTrace enabled, but FTrace cannot help much in either case (i.e., "0" in "Total # of Func. Traced"). This is because the manifestation of the two cases is kernel panics. Under such a scenario, FTrace cannot function normally. This result exposes a fundamental limitation of FTrace for debugging the storage stack in the kernel: FTrace itself depends on the probes or tracepoints embedded in the kernel, so it cannot survive severe kernel problems (e.g., kernel panics), let alone help diagnosing the problem in such severe scenarios.

The last column shows the size of the logs generated by FTrace under *BugBench*^{k}. We can see that FTrace consumes a relatively small amount of storage space for most cases, ranging from 1.1MB ("8-BLK") to 14.1MB ("4-BTRFS"). Since the log size largely depends on the amount of workload operations, the low storage overhead implies that workloads included in *BugBench*^{k} are concise and effective for triggering the 7 cases.

On the other hand, the last case ("9-BLK") incurs a relatively large amount of storage overhead (i.e., around 7496 MB). This is because the failure requires a relatively heavy workload to trigger. Specifically, the workload includes pulling and installing many software packages from the Internet via the *dpkg* package manager, which involves both the network subsystem and the storage stack and leads to a large amount of kernel functions being traced (i.e. , "110,772,722"). Since only 3 critical functions contribute to the failure in the "9-BLK" case, the substantial amount of traced functions may dilute the debugging focus. In other words, more intelligent methodologies are likely needed to help derive insights from the abundant FTrace logs for debugging.

4.2. PANDA

PANDA (Platform for Architecture-Neutral Dynamic Analysis) is an open-source platform for program analysis [33]. By leveraging virtualization (i.e., QEMU [76]) and the LLVM compiler infrastructure [77], PANDA can help understand the behavior of the entire storage software stack. We focus on measuring its major feature (i.e., record & replay) and 4 related plugins (i.e., Show Instructions, Taint Analysis, Identify Processes, Process-Block Relationship) in this subsection because they are most relevant for diagnosing storage failures.

Since PANDA records the full state of a target system hosted in QEMU as well as all non-deterministic events in snapshots, it can achieve full-stack, all-instruction observability by design (i.e., all executed instructions are observable by replaying). Therefore, we do not calculate the function-based metrics as used in measuring FTrace (Section 4.1). Instead, we qualitatively measure if the target features are applicable in diagnosing failures.

Table 4 summarizes the results of applying PANDA to diagnose the 9 cases in *BugBench*^{k}. Similar to Table 3, the second column ("Still Reproducible") shows whether the bug cases can still be reproduced when using PANDA. We observe that PANDA do not introduce any interference for the first 8 cases, similar to FTrace.

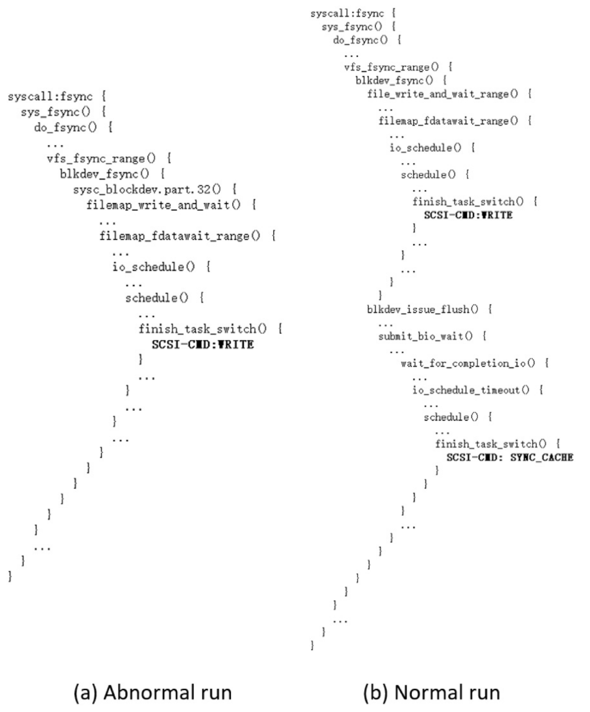
Nevertheless, PANDA fails in the last case ("9-BLK"). Specifically, we observe that the guest VM is hanging when applying PANDA to diagnose the "9-BLK" case. Multiple factors may contribute to the hang. First, as mentioned in Section 4.1, the workload requires installing many packages which are pulling from the Internet via *dpkg*. In other words, this workload involves the network subsystem and tends to generate many non-deterministic events within the kernel. Secondly, the QEMU-based PANDA needs to record all such events in order to ensure a successful replay, which incurs significant overhead in the critical path of QEMU's translation of guest instructions. As a result, QEMU is overloaded by PANDA's event recording, and cannot finish the translation of guest instructions on time. Eventually, the guest kernel (to be diagnosed) hangs in the QEMU VM. This result suggests that the state-of-the-art record & replay mechanisms may not be lightweight enough for diagnosing tricky storage failures.

For the remaining 8 cases, we find that PANDA's major record & replay feature and the 4 relevant plugins can all work normally (i.e., "✓") to support full-stack observability. On the other hand, the full-stack, all-instruction observability comes at the cost of overhead. The last column shows that PANDA incurs hundreds of MB storage overhead for its snapshots and event logs in most cases, which is orders of magnitude larger than the logs generated by FTrace on the same cases (Table 3).

Note that in terms of "6-F2FS" and "7-GFS" where FTrace fails due to kernel panics, PANDA can still work properly. This suggests a unique advantage of VM-based debugging tools like PANDA: by isolating the target storage software stack in the guest VM, the tool itself can survive severe problems of the target system and still provide effective support for diagnosing the problem.

Table 4
PANDA results on 9 $BugBench^k$ cases.

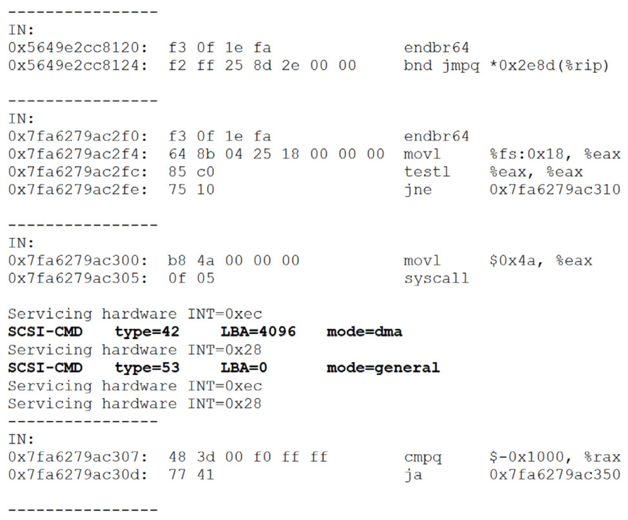
Case ID	Still Reproducible?	Record & Replay	Plugin				Log Size (MB)
			Show Instructions	Taint Analysis	Identify Processes	Process-Block Relationship	
1-EXT4	Yes	✓	✓	✓	✓	✓	659.9
2-EXT4	Yes	✓	✓	✓	✓	✓	671.9
3-BTRFS	Yes	✓	✓	✓	✓	✓	380.7
4-BTRFS	Yes	✓	✓	✓	✓	✓	811.7
5-BTRFS	Yes	✓	✓	✓	✓	✓	683.1
6-F2FS	Yes	✓	✓	✓	✓	✓	451.7
7-GFS	Yes	✓	✓	✓	✓	✓	408.7
8-BLK	Yes	✓	✓	✓	✓	✓	658.7
9-BLK	No	N/A	N/A	N/A	N/A	N/A	N/A


(a) Abnormal run
(b) Normal run

```
syscall:fsync {
  sys_fsync() {
    do_fsync() {
      ...
      vfs_fsync_range() {
        blkdev_fsync() {
          sysc_blockdev.part.32() {
            filemap_write_and_wait() {
              ...
              filemap_fdatawait_range() {
                ...
                io_schedule() {
                  ...
                  schedule() {
                    ...
                    finish_task_switch() {
                      SCSI-CMD:WRITE
                    }
                    ...
                  }
                  ...
                }
                ...
              }
            }
          }
        }
      }
    }
    ...
  }
}
```

```
syscall:fsync {
  sys_fsync() {
    do_fsync() {
      ...
      vfs_fsync_range() {
        blkdev_fsync() {
          file_write_and_wait_range() {
            ...
            filemap_fdatawait_range() {
              ...
              io_schedule() {
                ...
                schedule() {
                  ...
                  finish_task_switch() {
                    SCSI-CMD:WRITE
                  }
                  ...
                }
                ...
              }
            }
          blkdev_issue_flush() {
            ...
            submit_bio_wait() {
              ...
                wait_for_completion_io() {
                  ...
                  io_schedule_timeout() {
                    ...
                    schedule() {
                      ...
                      finish_task_switch() {
                        SCSI-CMD: SYNC_CACHE
                      }
                    }
                  }
                }
                ...
              }
            }
          }
        }
      }
    }
    ...
  }
}
```

Fig. 4. An Example of Augmented FTrace Log for 8-BLK.



```
----------------
IN:
0x5649e2cc8120:  f3 0f 1e fa                endbr64
0x5649e2cc8124:  f2 ff 25 8d 2e 00 00       bnd jmpq *0x2e8d(%rip)

----------------
IN:
0x7fa6279ac2f0:  f3 0f 1e fa                endbr64
0x7fa6279ac2f4:  64 8b 04 25 18 00 00 00    movl     %fs:0x18, %eax
0x7fa6279ac2fc:  85 c0                      testl    %eax, %eax
0x7fa6279ac2fe:  75 10                      jne      0x7fa6279ac310

----------------
IN:
0x7fa6279ac300:  b8 4a 00 00 00             movl     $0x4a, %eax
0x7fa6279ac305:  0f 05                      syscall

Servicing hardware INT=0xec
SCSI-CMD    type=42    LBA=4096    mode=dma
Servicing hardware INT=0x28
SCSI-CMD    type=53    LBA=0       mode=general
Servicing hardware INT=0xec
Servicing hardware INT=0x28
----------------
IN:
0x7fa6279ac307:  48 3d 00 f0 ff ff          cmpq     $-0x1000, %rax
0x7fa6279ac30d:  77 41                      ja       0x7fa6279ac350

----------------
```

Fig. 5. An Example of Augmented QEMU Log for 8-BLK.

4.3. Enhancing low-level observability

Through the experiments with $BugBench^k$, we find that neither FTrace nor PANDA can provide direct observability on the lowest level of information communicated between the storage software and the storage device, i.e., the device commands (e.g., SCSI [70]). Such command-level information is valuable in diagnosing storage failures because the persistent storage states are changed by the device commands directly. Traditionally, bus analyzers [36,37] are used to capture such command-level information. But as mentioned in Section 2.2, bus analyzers are hardware-based tools which are not as convenient as software tools. We introduce software extensions to capture bus-analyzer-like command information and enhance the low-level observability of both FTrace and PANDA in this subsection.

FTrace Extension. To avoid introducing unnecessary complexity to FTrace's probe mechanism in the kernel, we use an indirect way to extend FTrace. Specifically, we use a customized iSCSI driver [78] to collect device commands with timestamp, and align the collected device commands with the original FTrace logs based on timestamp. In doing so, the kernel functions are augmented with low-level device commands under the corresponding critical I/O paths.

We have verified that this extension method works for all the cases where FTrace can work normally without the extension. As an example, Fig. 4 shows a simplified version of the extended FTrace logs for the "8-BLK" case. The traced kernel functions are augmented with additional SCSI commands (bolded) based on timestamp. Fig. 4(a) shows the augmented log of an abnormal run (i.e., the bug is triggered), where we can see that the function $blkdev\_fsync$ eventually generates a write command to the device ("SCSI-CMD: WRITE"). This is problematic because the high-level sync function (i.e., $blkdev\_fsync$) should generate a low-level sync operation (instead of simply a regular write operation) at the device command level. Fig. 4(b) shows the augmented log of a corresponding normal run. We can see that an additional sync command ("SCSI-CMD: SYNC_CACHE") is actually generated within the scope of the $blkdev\_fsync$ function, which is expected.

Essentially, our extension combines the features of FTrace and the traditional hardware-based bus analyzer [36,37]. By extending FTrace logs with the command-level information in this way, we enhance the low-level observability of FTrace without using special hardware.

PANDA Extension. As mentioned in Section 4.2, PANDA uses QEMU to host the entire storage software stack in the guest VM, so the iSCSI driver solution for FTrace does not work for PANDA. Instead, we modify QEMU to capture all command-level information and leverage QEMU's internal logging mechanism to align commands with instructions.

Specifically, in QEMU, the guest OS kernel communicates with a SCSI device by sending Command Descriptor Blocks (CDBs) over the bus. QEMU maintains a 'struct SCSICommand' for each SCSI command, which contains a 16-byte buffer ('SCSICommand->buf') holding the CDB. Every SCSI command type is identified by the opcode at the beginning of the CDB, and the size of CDB is determined by the opcode. For example, the CDB for the WRITE_10 command is represented by the first 10 bytes of the buffer. For simplicity, we always transfer 16 bytes from the buffer to the command log and use the opcode to identify valid bytes. QEMU classifies SCSI commands into either Direct Memory Access (DMA) commands (e.g., READ_10) or Admin commands (e.g., VERIFY_10), and both are handled in the same way in our extension since they share the same data structure.

As an example, Fig. 5 shows a simplified version of the augmented QEMU log for the "8-BLK" case. The two bold lines (i.e., "SCSI-CMD ...") are the added device command information. and the remaining lines are the original QEMU log which includes both instructions (i.e., lines starting with addresses "0x5649e2..." etc.) and interrupts (e.g., "Servicing hardware INT=0xec"). The dash lines show the translation iteration of QEMU, each of which includes one basic block of instructions and the relevant device commands (if any). Similar to the FTrace extension, we enhance the low-level observability of PANDA/QEMU log without relying on special hardware.

4.4. Summary & discussion

To sum up, we have measured and evaluated the debugging observability of FTrace and PANDA via $BugBench^k$. Through the experiments, it is clear that FTrace and PANDA have different design tradeoffs and provide different level of observability. Moreover, we have demonstrated that it is possible to enhance their low-level observability via different lightweight extensions without hardware.

In particular, the results of FTrace suggest that tracing-based tools may be fundamentally limited for diagnosing storage failures in two aspects: (1) they may trace too many functions/events most of which may not be relevant to the root cause; (2) they may fail to function properly when the target storage system is malfunctional severely (e.g., kernel panics as in "6-F2FS" and "7-GFS").

On the other hand, the results of PANDA suggest that VM-based tools may be more viable for diagnosing storage failures because they can isolate the entire storage software stack from the core debugging functionality. However, in complicated failure scenarios, the events monitored may overwhelm the virtualization layer (e.g., "9-BLK" for PANDA), which suggests that more lightweight and less intrusive methods are needed to leverage virtualization for debugging.

We focus on measuring the observability of the debugging tools in this work because this is one of the most important metrics for debugging failures [35]. We envision many opportunities for further improvements based on the initial effort. For example, we recognize that observability is only one desired property proposed recently for improving system reliability [35]. There are other important properties and tools which may be measured by using $BugBench^k$ (e.g., runtime overhead of debugging tools, false positive rates of bug detection tools). Also, the current prototype of $BugBench^k$ only contains 9 reproducible cases due to the difficulty of reproducing real-world storage failures with incomplete information (as discussed in Section 3.3). And unfortunately, based on our investigation, none of the 277 issues collected in our dataset (Section 3) are directly related to the PM modules introduced to the storage stack recently. In terms of debugging tools, we only measure the core features of FTrace (i.e., kernel function tracing) and PANDA (i.e., record & replay and 4 related plugins) in our experiments. In fact, both FTrace and PANDA provide a rich set of additional features which might also be helpful for failure diagnosis. For example, FTrace allows users to add additional events tracing based on tracepoints [32]. Similarly, PANDA has additional plugins built on top of its record & replay framework. We leave reproducing PM-specific cases, deriving additional metrics, and measuring other debugging features and tools as future work.

While we only touch the observability of diagnosing the storage stack in this paper, the concept is also applicable in other contexts. In particular, researchers and practioners have recognized the critical importance of observability in the Cloud Native environment [79], where various components have been developed to enhance the observability to meet service-level objectives (SLOs) [79]. However, different from the modern Cloud-Native environment which supports loosely-coupled microservices and enables flexible integration of monitoring, tracing, logging, etc. services for observability, the storage stack in the monolithic Linux kernel has more constraints. How to improve the observability for the monolithic kernel with minimal intrusion remains an open question. Our effort on measuring the observability of state-of-the-art tools is one first step toward addressing the challenge.

Finally, we would like to point out that our goal of measuring the observability of different debugging tools in this work is not to imply which one is better. Instead, we hope to identify the potential limitations of the state-of-the-art tools in the context of diagnosing realistic storage failures, and inspire further improvements to address the debugging challenge. And as shown in our experiments, although the total number of reproducible cases is relatively small, $BugBench^k$ and the associated metrics have already exposed the limitations of the state-of-the-art tools evaluated, which suggests the needs and opportunities for more advanced diagnosis support. We hope that our initial $BugBench^k$ effort and the proof-of-concept extensions can inspire more follow-up research efforts in the communities, and contribute to the development of benchmarking for system reliability in general.

5. Related work

In this section, we discuss four categories of related work that have not been covered sufficiently in the previous sections.

Benchmarking Storage Systems. Great efforts have been made to benchmark and measure various storage systems [25–28,80,81]. For example, FIO [27] allows specifying diverse I/O patterns (e.g., sequential/random/mixed read or write operations) in multiple threads or processes. WHISPER [80] includes ten PM applications covering three types of access interfaces to PM, which enables analyzing the characteristics of PM applications (e.g., percentage of writes to PM, number of ordering points). Complementary to these existing benchmarking efforts which mostly focus on measuring the performance metrics, we introduce $BugBench^k$ and a set of metrics to enable quantitative and qualitative measurement of debugging observability.

Coincidentally, there is an early work by Lu et al. which is called BugBench [25]. The authors collected 17 bug cases in C/C++ applications, and proposed to evaluate bug detections tools based on the bug cases. Different from BugBench which includes application-level bug cases, $BugBench^k$ includes buggy OS kernels covering major components of the storage stack (e.g., multiple file systems, and low-level block I/O software), which are arguably more difficult to package, reproduce, or diagnose compared to user-level applications. Also, different from BugBench which focuses on evaluating user-level *testing* tools, we focus on evaluating the `debugging` tools for diagnosing failures, which is critically important for resolving failures in the real world but unfortunately is much less investigated compared to the abundant research on bug detection [30,31]. On the other hand, both BugBench and $BugBench^k$ focus reliability-oriented metrics (e.g., false positive rates, observability) and aims to improve the system robustness, which is different from traditional performance-oriented metrics. Therefore, we view the two efforts as complementary to each other. We hope that by reviving the concept of benchmarking for observability and other important reliability-oriented properties of computer systems, this work will inspire follow-up research and help improve the robustness of systems in general.

Studies of Software Bugs and Failures. Many researchers have performed empirical studies on bugs or failures in software systems [73–75,82–84]. For example, Lu et al. [75] studied 5079 patches from 6 Linux file systems and identified evolution trends; Lu et al. [73] studied 105 concurrency bugs from 4 applications and identified a number of common bug patterns (e.g., atomicity-violation and order-violation); Duo et al. [53] studied 1350 PM-related kernel patches and identified a number of PM bug characteristics including PM patch categories, PM bug patterns, consequences, and fix strategies; Gunawi et al. [85] studied 597 cloud service outages and derived multiple lessons including the outage impacts, causes, etc; Liu et al. [86] studied hundreds of incidents in Microsoft Azure.

Generally, our work is complementary to the existing ones as we focus on bugs in the entire storage stack experienced by the end users, which has a different scope compared to most of the existing studies. Moreover, we reproduce a set of cases and derive a $BugBench^k$ to measure representative debugging tools, which is beyond the scope of existing empirical studies.

Characterizing Storage Devices. Many researchers have studied the behaviors of storage devices in depth, including both HDDs [87–89] and SSDs [1,55,56,59,90–96]. For example, Bairavasundaram et al. [88] analyze the data corruption and latent sector errors in production systems containing 1.53 million HDDs; Maneas et al. [56] study the reliability of 1.4 million SSDs deployed in NetApp RAID systems. Schroeder et al. [89] analyze the disk replacement data of seven production systems over five years. Generally, these studies may provide valuable insights for reasoning complex storage failures caused by device. Different from these device-level studies, we analyze the storage failures at the system level involving different kernel components, which is complementary to the existing work.

Testing Storage Software. Great efforts have been made to test various storage software systems [41,54,58,60,61,97–101], with the goal of exposing bugs that could lead to failures. For example, EXPLODE [41] uses modeling checking to find storage system bugs [41], and B^3 applies bounded black-box testing to detect crash-consistency bugs in file systems [58]. However, testing tools are generally not suitable for diagnosing system failures because they typically require a well-controlled environment (e.g., a highly customized kernel [41,58]), which may be substantially different from the storage stack that need to be diagnosed. While the goal of this work is not to develop a new bug detection tool, the $BugBench^k$ created in this work may be used to evaluate such tools as well (e.g., false positive rate on detecting the reproducible bugs), which we leave as future work.

6. Conclusions

We have studied 277 real-world storage failures to quantitatively understand their characteristics. Based on the characterization, we derived a $BugBench^k$ which includes the necessary workloads and software environments to reproduce 9 realistic storage failure cases. We applied $BugBench^k$ to study two representative open source tools and derived concrete metrics to quantitatively/qualitatively measure their debugging observability. Moreover, we demonstrated that it is possible to enhance the observability of the state-of-the-art tools via lightweight extensions.

To the best of our knowledge, this work is the first effort to measure the observability of debugging tools. The work demonstrated in this paper suggests many opportunities for further improvements such as reproducing and packaging other types of bugs cases, deriving additional metrics for other desirable system properties, and measuring other tools or features, which we leave as future work. We hope that our initial effort will inspire follow-up research in the communities and help measure and improve the robustness of computer systems in general.

Acknowledgments

We thank the anonymous reviewers for their insightful feedback. We also thank researchers from Western Digital including Adam Manzanares, Filip Blagojevic, Qing Li, and Cyril Guyot for valuable discussions on the internals of the storage stack and the latest storage technology. In addition, we thank Wei Xu, Om Rameshwar Gatla, Prakhar Bansal, Runzhou Han, and Philip Ma for the help on reproducing and/or analyzing failure reports and bug patches. This work was supported in part by NSF, United States under grants CNS-1566554 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] Ryan Gabrys, Eitan Yaakobi, Laura M. Grupp, Steven Swanson, Lara Dolecek, Tackling intracell variability in TLC Flash through tensor product codes, in: 2012 IEEE International Symposium on Information Theory Proceedings, 2012.
[2] Yu Cai, Erich F. Haratsch, Onur Mutlu, Ken Mai, Error patterns in MLC NAND flash memory: Measurement, characterization, and analysis, in: Proceedings of the Conference on Design, Automation and Test in Europe, DATE, 2012.
[3] Laura M. Grupp, Adrian M. Caulfield, Joel Coburn, Steven Swanson, Eitan Yaakobi, Paul H. Siegel, Jack K. Wolf, Characterizing flash memory: anomalies, observations, and applications, in: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO, 2009.
[4] H Kurata, K Otsuga, A Kotabe, S Kajiyama, T Osabe, Y Sasago, S Narumi, K Tokami, S Kamohara, O Tsuchiya, The impact of random telegraph signals on the scaling of multilevel flash memories, in: VLSI Circuits, 2006. Digest of Technical Papers, 2006.
[5] Hanmant P Belgal, Nick Righos, Ivan Kalastirsky, Jeff J Peterson, Robert Shiner, Neal Mielke, A new reliability model for post-cycling charge retention of flash memories, in: Proceedings of the 40th Annual Reliability Physics Symposium, 2002.
[6] Kang-Deog Suh, Byung-Hoon Suh, Young-Ho Lim, Jin-Ki Kim, Young-Joon Choi, Yong-Nam Koh, Sung-Soo Lee, Suk-Chon Kwon, Byung-Soon Choi, Jin-Sun Yum, Jung-Hyuk Choi, Jang-Rae Kim, Hyung-Kyu Lim, A 3.3v 32mb NAND flash memory with incremental step pulse programming scheme, in: IEEE J. Solid-State Circuits (JSSC), 1995.
[7] T. Ong, A. Frazio, N. Mielke, S. Pan, N. Righos, G. Atwood, S. Lai, Erratic erase in ETOX/sup TM/ flash memory Array, in: Symposium on VLSI Technology, VLSI, 1993.
[8] Adam Brand, Ken Wu, Sam Pan, David Chin, Novel read disturb failure mechanism induced by FLASH cycling, in: Proceedings of the 31st Annual Reliability Physics Symposium, 1993.
[9] Jihye Seo, Wook-Hee Kim, Woongki Baek, Beomseok Nam, Sam H Noh, Failure-atomic slotted paging for persistent memory, ACM SIGPLAN Not. (2017).
[10] Advanced Flash Technology Status, Scaling Trends & Implications to Enterprise SSD Technology Enablement, https://www.flashmemorysummit.com/English/Collaterals/Proceedings/2012/20120821_TA12_Yoon_Tressler.pdf.
[11] Justin Meza, Qiang Wu, Sanjeev Kumar, Onur Mutlu, A large-scale study of flash memory failures in the field, in: ACM SIGMETRICS Performance Evaluation Review, 2015.
[12] Flash array, https://patents.google.com/patent/US4101260A/en.
[13] Basic Performance Measurements of the Intel Optane DC Persistent Memory Module, https://arxiv.org/abs/1903.05714.
[14] Scsi-mq, 2017, https://lwn.net/Articles/602159/, March 20.
[15] Blake Caldwell, Improving block-level efficiency with scsi-mq, 2015, arXiv preprint arXiv:1504.07481.
[16] Bart Van Assche, Increasing SCSI LLD driver performance by using the SCSI multiqueue approach, 2015.
[17] DAX: Page cache bypass for filesystems on memory storage, https://lwn.net/Articles/618064/.
[18] When solid state drives are not that solid, 2015, https://blog.algolia.com/when-solid-state-drives-are-not-that-solid/, June 15.
[19] A guide to mdadm, https://raid.wiki.kernel.org/index.php/A_guide_to_mdadm.
[20] raid0: data corruption when using trim, 2015, https://www.spinics.net/lists/raid/msg49440.html, July 19.
[21] Failure on freebsd/SSD: Seeing data corruption with zfs trim functionality , 2013, https://lists.freebsd.org/pipermail/freebsd-fs/2013-April/017145.html, April 29.
[22] Discussion on kernel TRIM support for SSDs: [1/3] libata: Whitelist SSDs that are known to properly return zeroes after TRIM , 2014, http://patchwork.ozlabs.org/patch/407967/, Nov 7 - Dec 8.
[23] Discussion on data loss on mSATA SSD module and Ext4 , 2016, http://pcengines.ch/msata16a.htm.
[24] HP warns that some SSD drives will fail at 32,768 hours of use, https://www.bleepingcomputer.com/news/hardware/hp-warns-that-some-ssd-drives-will-fail-at-32-768-hours-of-use/.
[25] Shan Lu, Zhenmin Li, FengQin, Lin Tan, Pin Zhou, YuanyuanZhou, BugBench: Benchmarks for evaluating bug detection tools, in: Workshop on the Evaluation of Software Defect Detection Tools, 2005.
[26] FIO Benchmark, https://fio.readthedocs.io/en/latest/fio_doc.html.
[27] Vasily Tarasov, Erez Zadok, Spencer Shepler, Filebencha flexible framework for file system benchmarking, in: Login Usenix Magazine, 2016.
[28] TPC Benchmarks, http://tpc.org/information/benchmarks5.asp.
[29] SPEC's benchmarks, https://www.spec.org/benchmarks.html.

[30] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, Taesoo Kim, Krace: Data race fuzzing for kernel file Systems, in: 2020 IEEE Symposium on Security and Privacy, SP, 2014.

[31] Dae R. Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, Insik Shin, Razzer: Finding kernel race bugs through fuzzing, in: 2019 IEEE Symposium on Security and Privacy, SP, 2019.

[32] ftrace, https://elinux.org/Ftrace.

[33] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, Ryan Whelan, Repeatable reverse engineering with PANDA, in: Proceedings of the 5th Program Protection and Reverse Engineering Workshop, 2015.

[34] Mohamad Gebai, Michel R. Dagenais, Survey and analysis of kernel and userspace tracers on linux: Design implementation and overhead, in: ACM Computing Survey, 2018.

[35] Andrew Quinn, Jason Flinn, Michael Cafarella, You can't debug what you can't see: Expanding observability with the OmniTable, in: Proceedings of the Workshop on Hot Topics in Operating Systems, HotOS, 2019.

[36] How to Read a SCSI Bus Trace, https://www.drdobbs.com/how-to-read-a-scsi-bus-trace/199101012.

[37] SCSI bus analyzer, https://www.ibm.com/support/knowledgecenter/en/ssw_aix_72/diagnosticsubsystem/header_54.html.

[38] Benchmarks by BenchCouncil, https://benchcouncil.org/benchmarks.html.

[39] SNIA NVM Programming Model (NPM).

[40] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, Steve Swanson, An empirical guide to the behavior and use of scalable persistent memory, 18th USENIX Conference on File and Storage Technologies, FAST, 2020.

[41] Junfeng Yang, Can Sar, Dawson Engler, Explode: a lightweight, general system for finding serious storage system errors, in: Proceedings of the 7th Symposium on Operating Systems Design and Implementation, OSDI, 2006.

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

[43] Lanyue Lu, Yupu Zhang, Thanh Do, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Physical disentanglement in a container-based file system, in: 11th USENIX Symposium on Operating Systems Design and Implementation, OSDI, 2014.

[44] Richard Stallman, Roland Pesch, Stan Shebs, et al., Debugging with GDB, Free Software Foundation, 2002.

[45] Brent Welch, Geoffrey Noer, Optimizing a hybrid SSD/HDD HPC storage system based on file size distributions, in: 2013 IEEE 29th Symposium on Mass Storage Systems and Technologies, MSST, 2013.

[46] NVM Express, 2016, https://nvmexpress.org/.

[47] Computeexpresslink(CXL), https://www.computeexpresslink.org/.

[48] Matias Bjorling, Jens Axboe, David Nellans, Philippe Bonnet, Linux Block IO: Introducing multi-queue SSD access on multi-core systems, in: Proceedings of the 6th International Systems and Storage Conference, SYSTOR, 2013.

[49] Changman Lee, Dongho Sim, Joo-Young Hwang, Sangyeun Cho, F2FS: A new file system for flash storage, in: Proceedings of the 13th USENIX Conference on File and Storage Technologies, FAST, 2015.

[50] Jian Xu, Steven Swanson, NOVA: A log-structured file system for hybrid volatile/non-volatile main memories, in: 14th USENIX Conference on File and Storage Technologies, FAST, 2016.

[51] Vaibhav Gogte, William Wang, Stephan Diestelhorst, Aasheesh Kolli, Peter M. Chen, Satish Narayanasamy, Thomas F. Wenisch, Software wear management for persistent memories, in: 17th USENIX Conference on File and Storage Technologies, FAST, 2019.

[52] Libata: add TRIM support, 2009, https://lwn.net/Articles/362108/, November 15.

[53] Duo Zhang, Om Rameshwar Gatla, Wei Xu, Mai Zheng, A study of persistent memory bugs in the linux kernel, in: Proceedings of the 14th ACM International Systems and Storage Conference, SYSTOR, 2021.

[54] Jinrui Cao, Om Rameshwar Gatla, Mai Zheng, Dong Dai, Vidya Eswarappa, Yan Mu, 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.

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

[56] Stathis Maneas, Kaveh Mahdaviani, Tim Emami, Bianca Schroeder, A study of SSD reliability in large scale enterprise storage deployments, in: 18th USENIX Conference on File and Storage Technologies, FAST, 2020.

[57] Seulbae Kim, Meng Xu, Sanidhya Kashyap, Jungyeon Yoon, Wen Xu, 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, 2019.

[58] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, Vijay Chidambaram, Finding crash-consistency bugs with bounded black-box crash testing, in: 13th USENIX Symposium on Operating Systems Design and Implementation, OSDI, 2018.

[59] Mai Zheng, Joseph Tucek, Feng Qin, Mark Lillibridge, Understanding the robustness of SSDs under power fault, in: Proceedings of the 11th USENIX Conference on File and Storage Technologies, FAST, 2013.

[60] Mai Zheng, Joseph Tucek, Dachuan Huang, Feng Qin, Mark Lillibridge, Elizabeth S. Yang, Bill W Zhao, Shashank Singh, Torturing databases for fun and profit, in: 11th USENIX Symposium on Operating Systems Design and Implementation, OSDI, 2014.

[61] Om Rameshwar Gatla, Muhammad Hameed, Mai Zheng, Viacheslav Dubeyko, Adam Manzanares, Filip Blagojević, Cyril Guyot, Robert Mateescu, Towards robust file system checkers, in: 16th USENIX Conference on File and Storage Technologies, FAST, 2018.

[62] GDB: The GNU Project Debugger, https://www.gnu.org/software/gdb/.

[63] Peter A. Buhr, Martin Karsten, Jun Shih, KDB: a multi-threaded debugger for multi-threaded applications, in: Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools, SPDT, 1996.

[64] Kgdb, https://elinux.org/Kgdb.

[65] lTTng, https://lttng.org/.

[66] SystemTap, https://sourceware.org/systemtap/.

[67] XRay: A function call tracing system, https://research.google/pubs/pub45287/.

[68] An introduction to KProbes, https://lwn.net/Articles/132196/.

[69] Dtrace, http://dtrace.org/blogs/.

[70] SCSI Commands Reference Manual by Seagate, https://www.seagate.com/files/staticfiles/support/docs/manual/Interface%20manuals/100293068j.pdf.

[71] Samuel T. King, George W. Dunlap, Peter M. Chen, Debugging operating systems with time-traveling virtual machines, in: Proceedings of the 2005 USENIX Technical Conference, 2005.

[72] Kernel bugzilla, https://www.bugzilla.org/.

[73] Shan Lu, Soyeon Park, Eunsoo Seo, Yuanyuan Zhou, Learning from mistakes: a comprehensive study on real world concurrency bug characteristics, in: Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS, 2008.

[74] Haryadi S. Gunawi, Thanh Do, Agung Laksono, Mingzhe Hao, Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Riza O. Suminto, What bugs live in the cloud? A study of issues in scalable distributed systems, in: Proceedings of the ACM Symposium on Cloud Computing, SOCC, 2014.

[75] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Shan Lu, A study of linux file system evolution, in: Proceedings of the 11th USENIX Conference on File and Storage Technologies, FAST, 2013.

[76] Fabrice Bellard, QEMU, a fast and portable dynamic translator, in: USENIX Annual Technical Conference, FREENIX Track, 2005.

[77] The LLVM Compiler Infrastructure, https://llvm.org/.

[78] Linux SCSI target framework (tgt), http://stgt.sourceforge.net/.

[79] Catherine Paganini, Danyel Fisher, Franciss Espenido, Gabriel H. Dinh, Heather Joslyn, Jason Morgan, Joab Jackson, Judy Williams, Libby Clark, Peter Putz, Steve Tidwell, Susan Hall, Cloud native observability for Devsops teams, in: The New Stack, 2021.

[80] Sanketh Nalli, Swapnil Haria, Mark D. Hill, Michael M. Swift, Haris Volos, An analysis of persistent memory use with WHISPER, in: Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS, 2017.

[81] Yahoo! Cloud Serving Benchmark, https://en.wikipedia.org/wiki/YCSB.

[82] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, DawsonEngle, An empirical study of operating systems errors, in: Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles, SOSP, 2001.

[83] David Lazar, Haogang Chen, Xi Wang, Nickolai Zeldovich, Why does cryptographic software fail? A case study and open problems, in: Proceedings of the Second Asia-Pacific Workshop on Systems, APSys, 2014.

[84] Haogang Chen, Yandong Mao, Xi WangDong Zhou, Nickolai Zeldovich, M. Frans Kaashoek, Linux kernel vulnerabilities:State-of-the-art defenses and open problem, in: Proceedings of the Second Asia-Pacific Workshop on Systems, APSys, 2014.

[85] Haryadi S. Gunawi, Mingzhe Hao, Riza O. Suminto, Agung Laksono, Anang D. Satria, Jeffry Adityatama, Kurnia J. Eliazar, Why does the cloud stop computing? Lessons from hundreds of service outages, in: Proceedings of the ACM Symposium on Cloud Computing, SOCC, 2016.

[86] Haopeng Liu, Shan Lu, Madan Musuvathi, Suman Nath, What bugs cause production cloud incidents? in: Proceedings of the Workshop on Hot Topics in Operating Systems, HotOS, 2019.

[87] Lakshmi N. Bairavasundaram, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Garth R. Goodson, Bianca Schroeder, An analysis of data corruption in the storage stack, Trans. Storage (2008).

[88] Lakshmi N. Bairavasundaram, Garth R. Goodson, Shankar Pasupathy, Jiri 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, 2007.

[89] Bianca Schroeder, Garth 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), 2007.

[90] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, Rina Panigrahy, Design tradeoffs for SSD performance, in: USENIX 2008 Annual Technical Conference, ATC, 2008.

[91] H Kurata, K Otsuga, A Kotabe, S Kajiyama, T Osabe, Y Sasago, S Narumi, K Tokami, S Kamohara, O Tsuchiya, The impact of random telegraph signals on the scaling of multilevel Flash memories, in: 2006 Symposium on VLSI Circuits, 2006.

[92] Jiangpeng Li, Kai Zhao, Xuebin Zhang, Jun Ma, Ming Zhao, Tong Zhang, How much can data compressibility help to improve NAND flash memory lifetime? in: Proceedings of the 13th USENIX Conference on File and Storage Technologies, FAST, 2015.

[93] Bianca Schroeder, Raghav Lagisetty, Arif Merchant, Flash reliability in production: The expected and the unexpected, in: 14th USENIX Conference on File and Storage Technologies, FAST, 2016.

[94] Huang-Wei Tseng, Laura M. Grupp, Steven Swanson, Understanding the impact of power loss on flash memory, in: Proceedings of the 48th Design Automation Conference, DAC, 2011.

[95] Erci Xu, Mai Zheng, Feng Qin, Yikang Xu, Jiesheng Wu, Lessons and actions: What we learned from 10K SSD-related storage system failures, in: Proceedings of the 2019 USENIX Annual Technical Conference, ATC, 2019.

[96] Erci Xu, Mai Zheng, Feng Qin, Jiesheng Wu, Yikang Xu, Understanding SSD Reliability in Large-scale Cloud Systems, in: Proceedings of the 3rd IEEE/ACM International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems, PDSW, 2018.

[97] Changwoo Min, Sanidhya Kashyap, Byoungyoung Lee, Chengyu Song, Taesoo Kim, Cross-checking semantic correctness: The case of finding file system bugs, in: Proceedings of the 25th Symposium on Operating Systems Principles, 2015.

[98] Jinrui Cao, Simeng Wang, Dong Dai, Mai Zheng, Yong Chen, A generic framework for testing parallel file systems, in: Proceedings of the 1st Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems, PDSW, 2016.

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

[100] Runzhou Han, Om Rameshwar Gatla, Mai Zheng, Jinrui Cao, Di Zhang, Dong Dai, Yong Chen, Jonathan Cook, A study of failure recovery and logging of high-performance parallel file systems, in: ACM Transactions on Storage TOS, 2021.

[101] Runzhou Han, Duo Zhang, Mai Zheng, Fingerprinting the checker policies of parallel file systems, in: Proceedings of the 5th ACM/IEEE International Parallel Data Systems Workshop, PDSW, 2020.