Can Applications Recover from fsync Failures?

ANTHONY REBELLO and YUVRAJ PATEL, University of Wisconsin–Madison, USA
RAMNATTHAN ALAGAPPAN, VMware Research Group, USA
ANDREA C. ARPACI-DUSSEAU and REMZI H. ARPACI-DUSSEAU, University of Wisconsin–Madison, USA

We analyze how file systems and modern data-intensive applications react to fsync failures. First, we characterize how three Linux file systems (ext4, XFS, Btrfs) behave in the presence of failures. We find commonalities across file systems (pages are always marked clean, certain block writes always lead to unavailability) as well as differences (page content and failure reporting is varied). Next, we study how five widely used applications (PostgreSQL, LMDB, LevelDB, SQLite, Redis) handle fsync failures. Our findings show that although applications use many failure-handling strategies, none are sufficient: fsync failures can cause catastrophic outcomes such as data loss and corruption. Our findings have strong implications for the design of file systems and applications that intend to provide strong durability guarantees.

CCS Concepts: • **Software and its engineering** → **File systems management**; • **General and reference** → **Reliability**; • **Computer systems organization** → *Reliability*; • **Software and its engineering** → **Software fault tolerance**; • **Information systems** → *Database recovery*;

Additional Key Words and Phrases: Durability, persistence, fsync failures, fsync, file system

ACM Reference format:
Anthony Rebello, Yuvraj Patel, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2021. Can Applications Recover from fsync Failures? *ACM Trans. Storage* 17, 2, Article 12 (June 2021), 30 pages.
https://doi.org/10.1145/3450338

1 INTRODUCTION

Applications that care about data must care about how data is written to stable storage. Issuing a series of write system calls is insufficient. A write call only transfers data from application memory into the operating system; the OS usually writes this data to disk lazily, improving performance via batching, scheduling, and other techniques [28, 48, 57, 58].

Ramnatthan Alagappan work done while at University of Wisconsin-Madison.
We thank our sponsors: VMware, Intel, Seagate, and Samsung, for their generous support. This material was also supported by funding from NSF under award numbers CNS-1421033, CNS-1763810, and CNS-1838733, and DOE DE-SC0014935. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF, DOE, or any other institutions.

Authors' addresses: A. Rebello, Y. Patel, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, University of Wisconsin–Madison, Department of Computer Sciences, Madison, WI, 53706; emails: arebello@wisc.edu, {yuvraj, dusseau, remzi}@cs.wisc.edu; R. Alagappan, VMware Research Group, Madison, WI, 53706; email: ralagappan@vmware.com.

1553-3077/2021/06-ART12 $15.00
https://doi.org/10.1145/3450338

To update persistent data correctly in the presence of failures, the order and timing of flushes to stable storage must be controlled by the application. Such control is usually made available to applications in the form of calls to fsync [9, 51], which forces unwritten ("dirty") data to disk before returning control to the application. Most update protocols, such as write-ahead logging or copy-on-write, rely on forcing data to disk in particular orders for correctness [33, 34, 38, 41, 50, 61].

Unfortunately, recent work has shown that the behavior of fsync during failure events is ill-defined [60] and error prone. Some systems, for example, mark the relevant pages clean upon fsync failure, even though the dirty pages have not yet been written properly to disk. Simple application responses, such as retrying the failed fsync, will not work as expected, leading to potential data corruption or loss.

In this article, we ask and answer two questions related to this critical problem. The first question (Section 3) relates to the file system itself: Why does fsync sometimes fail, and what is the effect on file-system state after the failure event?

To answer this first question, we run carefully crafted micro-workloads on important and popular Linux file systems (ext4 [47], XFS [59], Btrfs [55]) and inject targeted block failures in the I/O stream using dm-loki—our custom built device-mapper target for deterministic fault injection. We then use blockviz—a block trace visualization tool that enriches block access patterns with file-system specific information to examine the results. We provide the traces generated by blockviz to serve as reference for current file system error-handling behavior. Our findings show commonalities across file systems as well as differences. For example, all three file systems mark pages clean after fsync fails, rendering techniques such as application-level retry ineffective. However, the content in said clean pages varies depending on the file system; ext4 and XFS contain the latest copy in memory, while Btrfs reverts to the previous consistent state. Failure reporting is varied across file systems; for example, ext4 data mode does not report an fsync failure immediately in some cases, instead (oddly) failing the subsequent call. Failed updates to some structures (e.g., journal blocks) during fsync reliably lead to file-system unavailability. And finally, other potentially useful behaviors are missing; for example, none of the file systems alert the user to run a file-system checker after the failure.

The second question we ask is (Section 4): How do important data-intensive applications react to fsync failures? To answer this question, we build CuttleFS, a FUSE file system that can emulate different file system fsync failures. CuttleFS maintains its own page cache in user-space memory, separate from the kernel page cache, allowing application developers to perform durability tests against characteristics of different file systems, without interference from the underlying file system and kernel.

With this test infrastructure, we examine the behavior of five widely used data-management applications: Redis [21], LMDB [18], LevelDB [13], SQLite [23] (in both RollBack [1] and WAL modes [24]), and PostgreSQL [18] (in default and DirectIO modes). Our findings, once again, contain both specifics per system, as well as general results true across some or all. Some applications (Redis) are surprisingly careless with fsync, not even checking its return code before returning success to the application-level update; the result is a database with old, corrupt, or missing keys. Other applications (LMDB) exhibit false-failure reporting, returning an error to users even though on-disk state is correct. Many applications (Redis, LMDB, LevelDB, SQLite) exhibit data corruptions; for example, SQLite fails to write data to its rollback journal and corrupts in-memory state by reading from said journal when a transaction needs to be rolled back. While corruptions can cause some applications to reject newly inserted records (Redis, LevelDB, SQLite), both new and old data can be lost on updates (PostgreSQL). Finally, applications (LevelDB, SQLite, PostgreSQL) sometimes seemingly work correctly as long as the relevant data remains in the file-system cache;

when said data is purged from the cache (due to cache pressure or OS restart), however, the application then returns stale data (as retrieved from disk).

We also draw high-level conclusions that take both file-system and application behavior into account. We find that applications expect file systems on an OS platform (e.g., Linux) to behave similarly, and yet file systems exhibit nuanced and important differences. We also find that applications employ numerous different techniques for handling `fsync` failures, and yet none are (as of today) sufficient; even after the PostgreSQL `fsync` problem was reported [60], no application yet handles its failure perfectly. We also determine that application recovery techniques often rely upon the file-system page cache, which does not reflect the persistent state of the system and can lead to data loss or corruption; applications should ensure recovery protocols only use existing persistent (on-disk) state to recover. Finally, in comparing ext4 and XFS (journaling file systems) with Btrfs (copy-on-write file system), we find that the copy-on-write strategy seems to be more robust against corruptions, reverting to older states when needed.

The rest of this article is organized as follows: First, we motivate why this study is necessary (Section 2), followed by a file-system study (Section 3). Next, we study how applications react to `fsync` failures (Section 4). We then discuss the implications of our findings (Section 5), discuss related work (Section 6), and conclude (Section 7).

2 MOTIVATION

Applications that manage data must ensure that they can handle and recover from any fault that occurs in the storage stack. Recently, a PostgreSQL user encountered data corruption after a storage error and PostgreSQL played a part in that corruption [20]. Because of the importance and complexity of this error, we describe the situation in detail.

PostgreSQL is an RDBMS that stores tables in separate files and uses a **write-ahead log** *(wal)* to ensure data integrity [19]. On a transaction commit, the entry is written to the log and the user is notified of the success. To ensure that the log does not grow too large (as it increases startup time to replay all entries in the log), PostgreSQL periodically runs a checkpoint operation to flush all changes from the log to the different files on disk. After an `fsync` is called on each of the files, and PostgreSQL is notified that everything was persisted successfully, the log is truncated.

Of course, operations on persistent storage do not always complete successfully. Storage devices can exhibit many different types of partial and transient failures, such as latent sector errors [30, 44, 56], corruptions [29], and misdirected writes [46]. These device faults are propagated through the file system to applications in a variety of ways [43, 53], often causing system calls such as `read`, `write`, and `fsync` to fail with a simple return code.

When PostgreSQL was notified that `fsync` failed, it retried the failed `fsync`. Unfortunately, the semantics for what should happen when a failed `fsync` is retried are not well defined. While POSIX aims to standardize behavior, it only states that outstanding IO operations are not guaranteed to have been completed in the event of failures during `fsync` [17]. As we shall see, on many Linux file systems, data pages that fail to be written are simply marked clean in the page cache when `fsync` is called and fails. As a result, when PostgreSQL retried the `fsync` a second time, there were no dirty pages for the file system to write, resulting in the second `fsync` succeeding without actually writing data to disk. PostgreSQL assumed that the second `fsync` persisted data and continued to truncate the write-ahead log, thereby losing data. PostgreSQL had been using `fsync` incorrectly for 20 years [60].

After identifying this intricate problem, developers changed PostgreSQL to respond to the `fsync` error by crashing and restarting without retrying the `fsync`. Thus, on restart, PostgreSQL rebuilds state by reading from the *wal* and retrying the entire checkpoint process. The hope and intention is that this crash-and-restart approach will not lose data. Many other applications such as WiredTiger/MongoDB [27] and MySQL [3] followed suit in fixing their fsync retry logic.

This experience leads us to ask a number of questions. As application developers are not certain about the underlying file-system state on fsync failure, the first part of our study answers what happens when fsync fails. How do file systems behave after they report that an fsync has failed? Do different Linux file systems behave in the same way? What can application developers assume about the state of their data after an fsync fails? Thus, we perform an in-depth study into the fsync operation for multiple file systems.

The second part of our study looks at how data-intensive applications react to fsync failures. Does the PostgreSQL solution indeed work under all circumstances and on all file systems? How do other data-intensive applications react to fsync failures? For example, do they retry a failed fsync, avoid relying on the page cache, crash and restart, or employ a different failure-handling technique? Overall, how well do applications handle fsync failures across diverse file systems?

3 FILE SYSTEM STUDY

Our first study explores how file systems behave after reporting that an fsync call has failed. After giving a brief background of caching in file systems, we describe our methodology and our findings for the three Linux file systems.

3.1 Background

File systems provide applications with open, read, and write system calls to interact with the underlying storage media. Since block devices such as hard disks and solid state drives are much slower than main memory [63], the operating system maintains a page cache of frequently used pages of files in kernel space in main memory.

When an application calls read, the kernel first checks if the data is in the page cache. If not, then the file system retrieves the data from the underlying storage device and stores it in the page cache. When an application calls write, the kernel only *dirties* the page in memory while notifying the application that the write succeeded; there is now a mismatch between the data in memory and on the device and data can potentially be lost. For durability, the file system periodically synchronizes content between memory and disk by *flushing* dirty pages and marking them clean. Applications that require stronger durability guarantees can force the dirty pages to disk using the fsync system call.

Applications can choose to bypass the page cache altogether by opening files with *O_DIRECT* (DirectIO). For caching, applications must perform their own in user space. Calls to fsync are still required, since data may be cached within the underlying storage media; an fsync issues a FLUSH command to the underlying device so it pushes data all the way to stable storage.

3.2 Methodology

To understand how file systems should behave after reporting an fsync failure, we begin with the available documentation. The fsync man pages [9] report that fsync may fail for many reasons: the underlying storage medium has insufficient space (*ENOSPC* or *EDQUOT*), the file descriptor is not valid (*EBADF*), or the file descriptor is bound to a file that does not support synchronization (*EINVAL*). Since these errors can be discovered by validating input and metadata before initiating write operations, we do not investigate them further.

We focus on errors that are encountered only after the file system starts synchronizing dirty pages to disk; in this case, fsync signals an *EIO* error. *EIO* errors are difficult to handle, because the file system may have already begun an operation (or changed state) that it may or may not be able to revert.

To trigger *EIO* errors, we consider single, transient, write faults in line with the fail-partial failure model [52, 53]. When the file system sends a write request to the storage device, we inject a fault for a single sector or block within the request. Specifically, we build a kernel module device-mapper target called dm-loki that intercepts block-device requests from the file system and fails a particular write request to a particular sector or block while letting all other requests succeed; this allows us to observe the impact on an unmodified file system.

3.2.1 Workloads. To exercise the fsync path, we create three simple workloads that are representative of common write patterns seen in data-intensive applications.

Single Block Update (w_{su}): Open an existing file containing three pages (12 KB) and modify the middle page. This workload resembles many applications that modify the contents of existing files: LMDB always modifies the first two metadata pages of its database file; PostgreSQL stores tables as files on disk and modifies them in-place. Specifically, w_{su} issues system calls in the following sequence: open, lseek(4K), write(4K), *fsync*, *fsync*, sleep(40), close. The first fsync forces the dirty page to disk. While one fsync is sufficient in the absence of failures, we are interested in the impact of fsync retries after a failure; therefore, w_{su} includes a second fsync. Finally, since ext4, XFS, and Btrfs write out metadata and checkpoint the journal periodically, w_{su} includes a sleep for 40 seconds.

Multi Block Append (w_{ma}): Open a file in append mode and write a page followed by an fsync; writing and fsyncing is repeated after sleeping. This workload resembles many applications that periodically write to a log file: Redis writes every operation that modifies its in-memory data structures to an append only file; LevelDB, PostgreSQL, and SQLite write to a write-ahead-log and fsync the file after the write. w_{ma} repeats these operations after a delay to allow checkpointing to occur; this is realistic, as clients do not always write continuously and checkpointing may occur in those gaps. Specifically, w_{ma} issues system calls in the following sequence: open (in append mode), write(4K), *fsync*, sleep(40), write(4K), *fsync*, sleep(40), close.

Multi File Create (w_{dir}): Create a new file within a directory and then fsync both the file and the directory. This workload resembles file creation in many applications that care about durability. The ALICE framework [51] analyzes multiple applications and lists vulnerabilities that arise from not issuing an fsync on the parent directory after creating and calling fsync on a file. w_{dir} repeats these operations after a delay to allow checkpointing to occur; a realistic scenario, as applications often create files periodically. Specifically, w_{dir} issues system calls in the following sequence: open(dir), creat(file1), fsync(file1), fsync(dir), sleep(40), creat(file2), fsync(file2), fsync(dir), sleep(40), close(file1,file2,dir)[1].

3.2.2 dm-loki. To study file system behavior on fsync failure, we require a tool that injects failures deterministically. For example, always failing the *i*th write to a particular sector or block. Additionally, as file systems may overwrite a block multiple times, capturing disk state before and after an experiment is insufficient. We require the content of each read or write request. We built dm-loki [4], a loadable kernel module device-mapper target, to satisfy both requirements.

Contrast with current fault injection device-mapper targets such as dm-error and dm-flakey [5], dm-loki can change its fault injection configuration dynamically via messages through the dmsetup message [15] interface. A user may start dm-loki without any failure points and then send a message to the target to start failing certain sectors or blocks. Fault injection for a particular block is expressed as character sequences where the index in the sequence is incremented every time the block is written to. dm-loki decides to fail a particular access if the character at the current index indicates failure. For example, the sequence string wwxxw describes a pattern

[1] We use the notation creat for conciseness, but we actually use the open system call with flags O_WRONLY | O_CREAT.

where the first two writes succeed, the third and fourth fail, and all writes after succeed. For a specific block or sector failure sequence, the lowercase letters `w` and `x` at an index `i` decide whether the *i*th request is sent to the underlying device or failed. For accesses greater than the string length, we refer the last character to decide.

A user may also enable or disable request logging using the `dmsetup message` interface. `dm-loki` logs all read and write requests with associated data and flags to a file. Additionally, a user may inject "tags" via messages that are also logged. Injecting tags with a system call name and arguments right before its invocation allows us to identify the origin of each request. For example, all requests immediately preceded by a 30-second sleep tag implied that the requests were generated periodically for checkpointing.

3.2.3 blockviz. File systems need not treat all block write failures equally. Data block write failures may not be treated the same way as metadata block write failures. Additionally, file systems may treat different types of metadata differently. While `dm-loki` provides the functionality to inject a failure and log all BIO requests, it has no file-system level context about the specific sector or block. We build `blockviz`, an interactive Jupyter Notebook [45] widget in Python that takes BIO requests logged by `dm-loki` as input and enriches them with file-system specific information. We describe `blockviz`'s main features that help us in characterizing file-system behavior.

`dm-loki`'s ability to inject "tags" into the logs make it easier to visualize requests with `blockviz`. In our workloads, before issuing a system call, we inject a "tag" to `dm-loki`, specifying the system call and its arguments. `blockviz` visualizes the traces with tags, making it easier to identify the origin of every BIO request. Furthermore, as an interactive widget, clicking a particular request provides more file system-specific information. Using a combination of existing tools such as `debugfs` and `xfs_db`, and custom code to parse metadata blocks (such as XFS headers and Btrfs tree nodes), `blockviz` provides more information about every block in the trace.

As a particular block may be read from or written to multiple times, `blockviz` allows a user to compare different blocks in the trace. Since `dm-loki` logs all data read or written, `blockviz` creates checksums of the data for fast searches; a useful feature when trying to match content written in a journal block to content written to a metadata block during checkpointing. `blockviz` also allows metadata specific comparisons such as highlighting differences between two inode table entries in ext4 or identifying bitmap differences.

Figure 1 shows two sample traces from blockviz for ext4 ordered mode running w_{su}. We provide interpretations for these traces as they are used frequently in our findings (Section 3.3).

Figure 1(a) can be read as follows:

1 `open(/f1)` triggers a read request for the root directory data block.
2 There are no BIO requests during `write`.
3 On `fsync`, the data block for `/f1` is written to disk and the inode table is journaled.
4 There are no BIO requests during the second `fsync`.
5 During `sleep`, the journaled inode table is written to its actual location.
6 There are no BIO requests during `close`.

Figure 1(b) can be read as follows:

1–2 Same as Figure 1(a).
3 On `fsync`, the data block write for `/f1` fails and nothing is journaled. The user is notified of the failure through a syslog entry and `fsync` returns −1 with errno set to EIO.
4 The second `fsync` writes the inode table to the journal.
5–6 Same as Figure 1(a).

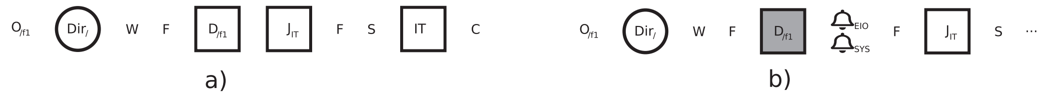

Fig. 1. Sample blockviz traces and how to interpret them:
The figure shows two sample traces from blockviz for w_{su} on ext4 ordered mode. The first (a) represents a normal run without any failures while the second (b) contains a grey-shaded block indicating a write failure.
System calls are represented by their first letter (in boldfaced font): **O**pen, **W**rite, **F**sync, **S**leep, **C**lose.
In some cases, a system call contains the file or directory path. As w_{su} opens file `/f1`, the first symbol is **O**$_{/f1}$.
BIO read requests are depicted using circles ○ and write requests with squares □. The letters within a BIO request are file-system dependent and are explained when first used. In this blockviz ext4 trace:

Dir$_{/}$ is the directory data block for the root(/) directory.
D$_{/f1}$ is the data block for file `/f1`.
IT is a block that contains the inode table entry for `/f1`.
J$_{IT}$ is a journal block containing the inode table entry.

In workloads that have multiple data block writes such as w_{ma}, we use `D'` for the data corresponding to the second `write` system call.
Trace (b) contains a grey-shaded block ■ indicating that the write to data block D textsubscript/f1 failed.
It is followed by two bells that symbolize notifications:

🔔$_{\text{EIO}}$: The immediate left system call (the `fsync`) is failed and errno is set to EIO.
🔔$_{\text{SYS}}$: The error is written to syslog.

As seen at the end of trace (b), if the rest of the trace is similar to the trace without failures, then we use a set of dots indicating an ellipsis.
For better readability, a dotted line (as seen below) separates this caption from the main text.

....

BIO request traces from `blockviz` contain too much low-level information. For instance, journaling in ext4 ordered mode involves writing a journal descriptor block that describes the following blocks, the actual block data to be journaled, a BIO flush request, and finally, a journal commit block with the **Force Unit Access (FUA)** flag set. For simplicity and conciseness, our traces in this article do not include the flush requests and BIO flags. For ext4 specifically, we also omit the journal descriptor and commit blocks from the traces.

3.2.4 Experiment Overview. We run the workloads on three different file systems: ext4, XFS, and Btrfs, with default mkfs and mount options. We evaluate both ext4 with metadata ordered journaling (data=ordered) and full data journaling (data=journal). We use an Ubuntu OS with Linux kernel version 5.2.11.

We run mkfs on loop devices. Since our workloads are small, the loop devices are backed by files of size 1 GB (images). The images are created using the `dd if=/dev/zero` command to ensure a clean initial zero state. The 1 GB size ensures that the block size of the file systems is 4 KB by default.[2] Since workloads w_{su} and w_{ma} require an existing file to operate on, we mount the file system, create an existing file of required size, and unmount. The images are now considered ready for the workloads.

For each file system and workload, we conduct experiments as follows:

We create a loop device (say, loop0) from the prepared image using the `losetup` [16] command. Then, using the `dmsetup` command, we set up a device-mapper device `/dev/dm/lokidev` that forwards all requests to the `dm-loki` target. We then run the workload with no fault points configured.

[2] A smaller file size can change the block size to 1 KB on ext4.

For each file system and workload, we first trace the block write access pattern. We then repeat the workload multiple times, each time configuring the fault injector to fail the *i*th write access to a given sector or block. We only fail a single block or sector within the block in each iteration. We use `blockviz` to analyze the traces and SystemTap [25] to examine the state of relevant buffer heads and pages associated with data or metadata in the file system.

3.2.5 Behavior Inference. We answer the following questions for each file system:

Basics of `fsync` Failures:

Q1 Which block (data, metadata, journal) failures lead to `fsync` failures?
Q2 Is metadata persisted if a data block fails?
Q3 Does the file system retry failed block writes?
Q4 Are failed data blocks marked clean or dirty in memory?
Q5 Does in-memory page content match what is on disk?

Failure Reporting:

Q6 Which future `fsync` will report a write failure?
Q7 Is a write failure logged in the syslog?

After Effects of `fsync` Failure:

Q8 Which block failures lead to file-system unavailability?
Q9 How does unavailability manifest? Does the file system shutdown, crash, or remount in read-only mode?
Q10 Does the file suffer from holes or block overwrite failures? If so, in which parts of a file can they occur?[3]

Recovery:

Q11 If there is any inconsistency introduced due to `fsync` failure, can fsck detect and fix it?

3.3 Findings

We now describe our findings for the three file systems we have characterized: ext4, XFS, and Btrfs. Our answers to our posed questions are summarized in Table 1.

3.3.1 Ext4. The ext4 file system is a commonly used journaling file system on Linux. The two most common options when mounting this file system are *data=ordered* and *data=journal*, which enable ext4 ordered mode and ext4 data mode, respectively. Ext4 ordered mode writes metadata to the journal, whereas ext4 data mode writes both data and metadata to the journal.

Ext4 ordered mode: We give an overview of ext4 ordered mode by describing how it behaves for our three representative workloads when no failures occur.

Single Block Update ($\mathbf{w}_{su}$). When no fault is injected and `fsync` is successful, ext4 ordered mode behaves as follows: During the `write` (Step 1), ext4 updates the page in the page cache with the new contents and marks the page dirty. On `fsync`, the page is written to a data block; after the data-block write completes successfully, the metadata (i.e., the inode with a new modification time) is written to the journal, and `fsync` returns 0 indicating success (Step 2). After the `fsync`, the dirty page is marked clean and contains the newly written data. On the second `fsync`, as there are

[3] In file-system terminology, a hole is a region in a file for which there is no block allocated. If a block is allocated but not overwritten with the new data, then we consider the file to have a *non-overwritten block* and suffer from *block overwrite failure.*

Table 1. Behavior of Various File Systems When fsync Fails

		fsync Failure Basics					Error Reporting		After Effects			Recovery
		Which block failure causes fsync failure?	Is metadata persisted on data block failure?	Which block failures are retried?	Is the page dirty or clean after failure?	Does the in-memory content match disk?	Which fsync reports the failure?	Is the failure logged to syslog?	Which block failure causes unavailability?	What type of unavailability?	Holes or block over-write failures? If yes where do they occur?	Can fsck help detect holes or block over-write failures?
		Q1	**Q2**	**Q3**	**Q4**	**Q5**	**Q6**	**Q7**	**Q8**	**Q9**	**Q10**	**Q11**
ext4	**ordered**	data, jrnl	yes ^{A}		clean ^{B}	no ^{B}	immediate	yes	jrnl	remount-ro	NOB, anywhere ^{A}	no
	data	data, jrnl	yes ^{A}		clean ^{B}	no ^{B}	next ^{C}	yes	jrnl	remount-ro	NOB, anywhere ^{A}	no
XFS		data, jrnl	yes ^{A}	meta	clean ^{B}	no ^{B}	immediate	yes	jrnl, meta	shutdown	NOB, within ^{A}	no
Btrfs		data, jrnl	no		clean	yes	immediate	yes	jrnl, meta	remount-ro	HOLE, within ^{D}	yes

[A] Non-overwritten blocks (Q10) occur because metadata is persisted despite data-block failure (Q2).

[B] Marking a dirty page clean (Q4) even though the content does not match the disk (Q5) is problematic.

[C] Delayed reporting (Q6) of fsync failures may confuse application error-handling logic.

[D] Continuing to write to a file after an fsync failure is similar to writing to an offset greater than file size, causing a hole in the skipped portion (Q10).

The table summarizes the behavior of the three file systems: ext4, XFS, and Btrfs, according to the questions posed in Section 3.2.5. The questions are divided into four categories mentioned at the top. For questions that require identifying a block type, we use the following abbreviations: Data Block (data), Journal Block (jrnl), Metadata Block (meta). In Q9, Remount-**ro** denotes remounting in read-only mode. In Q10, "anywhere" and "within" describe the locations of the holes or non-overwritten blocks (NOB); "within" does not include the end of the file. Entries with a superscript denote a problem.

no dirty pages, no block writes occur, and as there are no errors, fsync returns 0 (Step 3). During sleep, the metadata in the journal is checkpointed to its final in-place block location (Step 4). No writes or changes in page state occur during the close (Step 5). The trace for this experiment can be seen in Figure 2(a).

If fsync fails (i.e., returns −1 with errno set to *EIO*), then a variety of write problems could have occurred. For example, the data-block write could have failed (trace in Figure 2(b)); if this happens, then ext4 does not write the metadata to the journal. However, the updated page is still marked clean and contains the newly written data from Step 1, causing a discrepancy with the contents on disk. Furthermore, even though the inode table was not written to the journal at the time of the data fault, the inode table containing the updated modification time is written to the journal on the second fsync in Step 3. Steps 4 and 5 are the same as above, and thus the inode table is checkpointed.

Thus, applications that read this data block while the page remains in the page cache (i.e., the page has not been evicted and the OS has not been rebooted) will see the new contents of the data; however, when the page is no longer in memory and must be read from disk, applications will see the old contents.

Alternatively, if fsync failed, then it could be because a write to one of the journal blocks failed (trace in Figure 2(c)). In this case, ext4 aborts the journal transaction and remounts the file system in read-only mode, causing all future writes to fail.

Multi Block Append (w_{ma}). This next workload exercises additional cases in the fsync error path. If there are no errors and all fsyncs are successful (trace in Figure 3(a)), then the multi-block append workload on ext4 behaves as follows: First, during write, ext4 creates a new page with the new contents and marks it dirty (Step 1). On fsync, the page is written to a newly allocated on-disk data block; after the data-block write completes successfully, the relevant metadata (i.e., both the inode table and the block bitmap) are written to the journal, and fsync returns success (Step 2). As in w_{su}, the page is marked clean and contains the newly written data. During sleep,

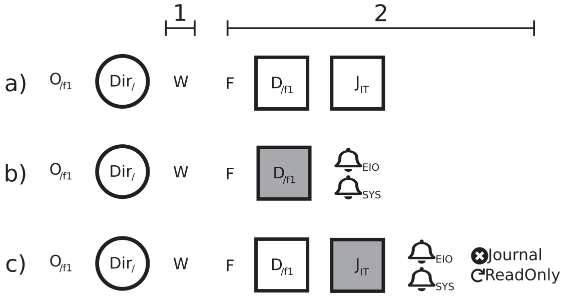
1
2
3
4
5
a) O/f1 Dir/ W F D/f1 JIT F S IT C
b) O/f1 Dir/ W F D/f1 EIO SYS F JIT S IT C
c) O/f1 Dir/ W F D/f1 JIT EIO SYS Journal ReadOnly

Fig. 2. Blockviz traces for w_{su} on ext4 ordered mode:
The figure shows three traces corresponding to different fault injection configurations of dm-loki.

(a) when no faults are injected: `open(/f1)` triggers a read request for the directory data block `/`. On `fsync`, the data block for `/f1` is written to disk and the Inode Table (IT) is written to the Journal (J_{IT}). During `sleep`, the Inode Table is checkpointed.

(b) dm-loki configured to fail the data block write: On data block write failure, the error is logged to syslog (🔔$_{SYS}$) and fsync fails with errno=EIO (🔔$_{EIO}$).

(c) dm-loki configured to fail the journal block write: On journal block write failure, in addition to the syslog and EIO notifications, ext4 aborts the journal (⊗Journal) and remounts in read-only mode (↻ReadOnly).

The figure is also annotated with steps (the first horizontal row with lines and numbers) that are referred to in the main text.

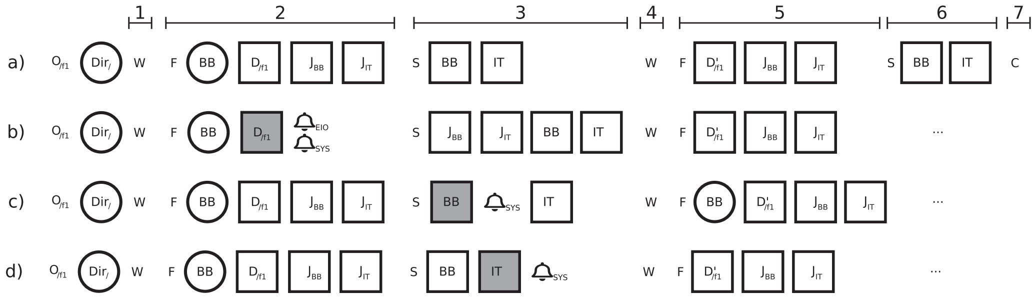
1
2
3
4
5
6
7
a) O/f1 Dir/ W F BB D/f1 JBB JIT S BB IT W F D'/f1 JBB JIT S BB IT C
b) O/f1 Dir/ W F BB D/f1 EIO SYS S JBB JIT BB IT W F D'/f1 JBB JIT ...
c) O/f1 Dir/ W F BB D/f1 JBB JIT S BB SYS IT W F BB D'/f1 JBB JIT ...
d) O/f1 Dir/ W F BB D/f1 JBB JIT S BB IT SYS W F D'/f1 JBB JIT ...

Fig. 3. Blockviz traces for w_{ma} on ext4 ordered mode:
The figure shows four traces corresponding to different fault injection configurations of dm-loki.

(a) when no faults are injected: Since w_{ma} involves appends, ext4 must allocate new blocks to the inode. Allocation is done on `fsync` as ext4 ordered mode uses delayed allocation by default. To allocate a block, the data block bitmap (BB) is read from disk if not already cached. While similar to w_{su}, an fsync in w_{ma} involves writing both the inode table (IT) and the block bitmap (BB) to the journal and later to their actual locations during checkpointing.

(b) dm-loki configured to fail the data block write: While the data block bitmap (BB) and inode table (IT) are not journaled after data block failure, they are still journaled and written to disk during checkpointing.

(c) dm-loki configured to fail the data block bitmap (BB) write: Failures during checkpointing are only logged to syslog while checkpointing continues. A stale block bitmap is re-read from disk after bitmap block failure.

(d) dm-loki configured to fail the inode table (IT) write: Unlike (c), inode tables are never read again from disk despite failure.

the metadata is checkpointed to disk (Step 3); specifically, the inode contains the new modification time and a link to the newly allocated block, and the block bitmap now indicates that the newly allocated block is in use. The pattern is repeated for the second `write` (Step 4), `fsync` (Step 5), and `sleep` (Step 6). As in w_{su}, there are no write requests or changes in page state during `close` (Step 7).

An `fsync` failure could again indicate numerous problems. First, a write to a data block could have failed in Step 2 (trace in Figure 3(b)). If this is the case, then the `fsync` fails and the page is marked clean; as in w_{su}, the page cache contains the newly written data, differing from the on-disk block that contains the original block contents. The inode table and block bitmap are first journaled and then written to disk in Step 3; thus, even though the data itself has not been written, the inode is modified to reference this block and the corresponding bit is set in the block bitmap. When the workload writes another 4 KB of data in Step 4, this write continues oblivious of the previous fault and Steps 5, 6, and 7 proceed as usual.

Thus, with a data-block failure, the on-disk file contains a non-overwritten block where it was supposed to contain the data from Step 1. A similar possibility is that the write to a data block in Step 5 fails; in this case, the file has a non-overwritten block at the end instead of somewhere in the middle. Again, an application that reads any of these failed data blocks while they remain in the page cache will see the newly appended contents; however, when any of those pages are no longer in memory and must be read from disk, applications will read the original block contents.

An `fsync` failure could also indicate that a write to a journal-block failed. In this case, as in w_{su}, the `fsync` returns an error and the following `write` fails, since ext4 has been remounted in read-only mode.

Because this workload contains an `fsync` after the metadata has been checkpointed in Step 3, it also illustrates the impact of faults when checkpointing the inode table and block bitmap. We find that ext4 reacts differently to block bitmap and inode table write failures (traces in Figures 3 (c) and (d)). In both cases, the failure is only logged to syslog, checkpointing proceeds to write other metadata, and the following fsync does not return an error. However, when ext4 fails to write the block bitmap, it marks the associated buffer head `!uptodate`, indicating that a future read must first retrieve the on-disk contents. On `fsync` in Step 5 (or `write` in Step 4 if there is no delayed allocation), ext4 must query the block bitmap to allocate a new block, reloading the stale on-disk block bitmap. With no more write failures, the fsync in Step 5 succeeds and checkpointing proceeds to write the new block bitmap—a version where only the bit for the second block in the file is set. The filesystem is now in an inconsistent state, with an inode pointing to a block whose bit is not set in the bitmap. While `fsck` can fix this inconsistency, it has to run in force mode (`fsck -f`), as ext4 incorrectly marks the filesystem clean on unmount.

We do not observe such inconsistencies with inode table write failures, as ext4 ignores the `!uptodate` flag on inode table buffer heads. Despite being `!uptodate`, ext4 continues to read and write to the latest in-memory inode table. Future successful writes to the on-disk inode table are guaranteed to have all the changes.

We note that for none of these `fsync` and metadata checkpoint failures does ext4 ordered mode recommend running the file system checker; furthermore, running the checker does not identify or repair any of the preceding problems. Finally, future calls to `fsync` never retry previous data writes that may have failed; neither are failed metadata writes during checkpointing. These results for ext4 ordered mode are all summarized in Table 1.

The ext4 file system also offers functionality to abort the journal if an error occurs in a file data buffer (mount option `data_err=abort`) and remount the file system in read-only mode on an error

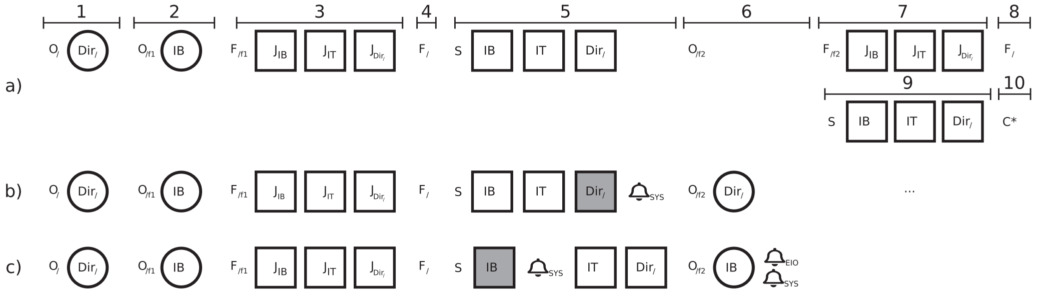

Fig. 4. Blockviz traces for w_{dir} on ext4 ordered mode:
The figure shows three traces corresponding to different fault injection configurations of dm-loki.

(a) when no faults are injected: Since w_{dir} creates new inodes, when the first file is created ($O_{/f1}$) the inode bitmap (IB) is read from disk if not already in cache. We use C* to denote closing all open file descriptors (`/`, `/f1`, and `/f2`).

(b) dm-loki configured to fail the directory data block write: A stale version of the directory data block is read from disk (as seen after $O_{/f2}$) if ext4 encounters a write failure during checkpointing for the same block.

(c) dm-loki configured to fail the inode bitmap block (IB) write: A stale version of the inode bitmap is read from disk, but ext4 fails the inode creation ($O_{/f2}$), logging the error to syslog and setting errno to EIO.

(mount option `errors=remount-ro`). However, we observe that the results are identical with and without the mount options.[4]

Multi File Create (w_{dir}). While w_{su} and w_{ma} address data-block, inode-table, and data-block-bitmap failures, w_{dir} exercises failures related to directory data blocks and inode bitmap blocks (trace in Figure 4(a)). If there are no errors and all `fsyncs` are successful, then the multi-file create workload on ext4 behaves as follows: First, during `open(dir)`, the directory data block is read from disk if not already cached (Step 1). Next, on `creat(file1)`, to allocate a new inode, ext4 reads the corresponding inode bitmap block from disk if not already cached (Step 2). Ext4 proceeds to modify the following data structures and marks them dirty: inode bitmap for file1's inode, inode table entry for file1 and dir, dir's directory data block that contains file1's name-to-inode mapping. On `fsync(file1)`, the dirtied metadata is written to the journal and `fsync(file1)` returns success (Step 3). Ext4 treats directory data blocks as metadata, so unlike w_{su} and w_{ma}, this workload does not write data blocks to disk during an `fsync`. On `fsync(dir)`, we observe no read or write requests as `dir` is already synced during `fsync(file1)` (Step 4). During `sleep`, the metadata is checkpointed to disk (Step 5); specifically, the inode bitmap has a previously cleared bit set, the inode table entry for the directory has a new modification time and updated size, the inode table entry for the file is initialized, and the directory data block has a new name-to-inode mapping. The pattern is repeated for `creat(file2)` (Step 6), `fsync(file2)` (Step 7), `fsync(dir)`, and `sleep`. As in the previous two workloads, we observe no bio requests during `close`.

As `fsync` in Step 3 and Step 7 only involve journal-block writes, similar to w_{su} and w_{ma}, a block write failure during `fsync` in w_{dir} will always return an error and trigger a remount in read-only mode.

Because this workload contains a `sleep` in Step 5, it also illustrates the impact of faults when checkpointing the inode table, inode bitmap, and directory data block. Inode table failures behave exactly as described for w_{ma}. Similar to block bitmap write failures, inode bitmap and directory data block write failures both mark the associated buffer heads `!uptodate` and trigger a read of

[4] We verified our observations by reproducing them using standard Linux tools and have filed a bug report for the same [2].

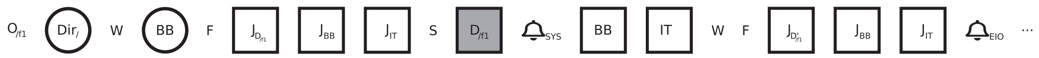

Fig. 5. Blockviz trace for w_{ma} on ext4 data mode, data block failures:
Unlike ext4 ordered mode, data blocks are written to the journal during `fsync` ($J_{D_{/f1}}$) and the data block bitmap is read on `write` instead of `fsync` as delayed allocation is disabled. On data block write failure during checkpointing, the error is logged to syslog. The second fsync writes the data and metadata from the second `write` to the journal but fails the `fsync` call with errno set to EIO.

the stale on-disk version during `creat(file2)` in Step 6. However, directory data block failures are problematic while inode bitmap failures are benign.

We find that files may disappear from directories even while the file system is running. After a directory data block write failure during checkpointing (trace in Figure 4(b)), because of the `!uptodate` flag, ext4 reads and modifies a stale version during `create(file2)`; the in-memory name-to-inode mapping for file1 is lost and the inode for file1 is an orphaned inode.[5] Future calls to `readdir(dir)` either directly or through the `ls` command will not contain `file1`. Although ext4 does not prompt us to run a checker, running `fsck -f` can detect orphaned inodes and place them in the `lost+found` directory. However, applications that encode information in the filename still suffer data loss.

Since ext4 must refer to both the inode bitmap and inode table when allocating a new inode, it detects the inconsistency and fails the system call. Additionally, to prevent further errors on `creat`, ext4 locks the entire group described by the inode bitmap and recommends running `fsck` to set the bit and unlock the group. This trace can be found in Figure 4(c).

Ext4 Data Mode: Ext4 data mode differs from ordered mode in that data blocks are first written to the journal and then later checkpointed to their final in-place block locations.

As shown in Table 1, the behavior of `fsync` in ext4 data mode is similar to that in ext4 ordered mode for most cases: For example, on a write error, pages may be marked clean even if they were not written out to disk, the file system is remounted in read-only mode on journal failures, metadata failures are not reported by `fsync`, and files can end up with non-overwritten blocks in the middle or end.

However, the behavior of ext4 data mode differs in one important scenario. Because data blocks are first written to the journal and later to their actual block locations during checkpointing, the first `fsync` after a write may succeed even if a data block will not be successfully written to its permanent in-place location. As a result, a data-block fault causes the second `fsync` to fail instead of the first; in other words, the error reporting by `fsync` is delayed due to a *failed intention* [39]. This trace can be seen in Figure 5.

3.3.2 *XFS.*

XFS is a journaling file system that uses B-trees. Instead of performing physical journaling like ext4, XFS journals logical entries for changes in metadata.

Figure 6(a) shows a trace of XFS without any failures for w_{su}. As shown in Table 1, from the perspective of error reporting and `fsync` behavior, XFS is similar to that of ext4 ordered mode. Specifically, failing to write data blocks (trace in Figure 6(b)) leads to `fsync` failure and the faulty data pages are marked clean even though they contain new data that has not been propagated to disk; as a result, applications that read this faulty data will see the new data only until the page has been evicted from the page cache. Similarly, failing to write a journal block will cause `fsync` failure (trace in Figure 6(c)), while failing to write a metadata block will not. XFS remains available for reads and writes after data-block faults.

[5] Orphaned inodes are inodes that can never be accessed, as no directory points to them.

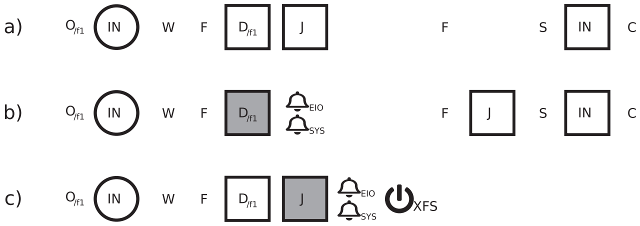

Fig. 6. Blockviz traces for w_{su} on XFS:
The figure shows three traces corresponding to different fault injection configurations of dm-loki.

(a) when no faults are injected: On open, XFS reads inode information from disk if not already cached (IN); it includes directory entries. On `fsync`, like ext4, XFS writes the data block and then journals metadata related to the changes. During a checkpoint, the inode information (IN) with updated mtime is written to disk.
(b) dm-loki configured to fail the data block write: XFS immediately fails the `fsync` after a data block failure. However, the updated mtime is journaled in the second `fsync` and checkpointed during the `sleep`.
(c) dm-loki configured to fail the journal block write: On journal block failure, XFS fails the fsync and shuts down the file system (⏻$_{\text{XFS}}$).

XFS handles `fsync` failures in a few ways that are different than ext4 ordered mode. First, on a journal-block fault, XFS shuts down the file system entirely (Figure 6(c)) instead of merely remounting in read-only mode; thus, all subsequent read and write operations fail. Second, XFS retries metadata writes when it encounters a fault during checkpointing; the retry limit is determined by a value in `/sys/fs/xfs/*/error/metadata/*/max_retries`, but is infinite by default. If the retry limit is exceeded, then XFS again shuts down the file system. We provide traces for w_{ma} in Figure 7 to highlight the retries.

The multi-block append workload illustrates how XFS handles metadata when writes to related data blocks fail. If the write to the first data block fails, then XFS writes no metadata to the journal and fails the `fsync` immediately. When later data blocks are successfully appended to this file, the metadata is updated, which creates a non-overwritten block in the file corresponding to the first write. However, if no new data blocks are successfully appended, then the on-disk metadata is not updated to reflect any of these last writes (i.e., the size of the file is not increased).[6] Thus, while in ext4 a failed write always causes a non-overwritten block, in XFS, non-overwritten blocks cannot exist at the end of a file. However, for either file system, if the failed blocks remain in the page cache, then applications can read those blocks regardless of whether they are in the middle or the end of a file.

During checkpointing, since XFS either shuts down or retries writes on metadata failures, we do not observe the same inconsistencies as described for ext4 when running the multi-file create workload w_{dir}.

3.3.3 *Btrfs.*

Btrfs is a copy-on-write file system that avoids writing to the same block twice, except for the superblock, which contains root-node information. Figure 8(a) provides a trace of w_{su} without any failures along with a description of Btrfs's data structures. At a high level, some of the actions in Btrfs are similar to those in a journaling file system: Instead of writing to a journal, Btrfs writes to a log tree to record changes when an `fsync` is performed; instead of checkpointing

[6] To be precise, the mtime and ctime of the file are updated, but not the size of the file. Additional experiments removed for space confirm this behavior.

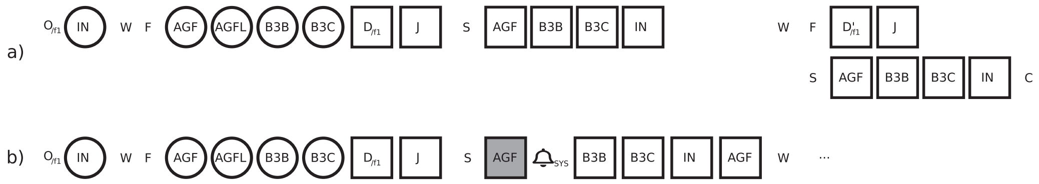

Fig. 7. Blockviz traces for w_{ma} on XFS:
The figure shows two traces corresponding to different fault injection configurations of dm-loki.

(a) when no faults are injected: Like ext4 ordered mode, XFS uses delayed allocation and must allocate new blocks to the inode, since w_{ma} is appending to the file. Unlike ext4's free space block bitmap, XFS tracks free space for every allocation group (AG—groups of inodes) using two B+trees that need to be read from disk if not cached. First, it reads the allocation group free space block (AGF) that contains information about the B+trees. Next, it reads the allocation group free list block (AGFL) that contains pointers to free space for growing the B+trees. It then reads the two B+trees, and the first is sorted by block number (B3B) to quickly find space closer to a given block. The second tracks space by size (B3C) to quickly find free space of a given size. On `fsync`, the data block is written and the free space changes are journaled along with the inode changes. During checkpointing, these changes are written to their actual disk locations. Since the B+trees did not need to grow, there are no modifications to AGFL. With no failures, the pattern repeats for the next `write`, `fsync`, and `sleep`.

(b) dm-loki configured to fail a metadata block during checkpointing: During checkpointing, XFS retries the failed write to the allocation group free space block (AGF). The retry limit is configurable and set to infinity by default. However, if the limit is reached, then XFS shuts down the file system and recommends running `fsck`. While this trace shows faults for AGF, we observe similar behavior for failures on B3B, B3C, and IN.

to fixed in-place locations, Btrfs writes to new locations and updates the roots in its superblock. However, since Btrfs is based on copy-on-write, it has a number of interesting differences in how it handles `fsync` failures compared to ext4 and XFS, as shown in Table 1.

Like ext4 ordered mode and XFS, Btrfs fails `fsync` when it encounters data-block faults (trace in Figure 8(b)). However, unlike ext4 and XFS, Btrfs effectively reverts the contents of the data block (and any related metadata) back to its old state (and marks the page clean). Thus, if an application reads the data after this failure, then it will never see the failed operation as a temporary state. As in the other file systems, Btrfs remains available after this data-block fault.

Similar to faults to the journal in the other file systems, faults to Btrfs's log tree can result in a failed `fsync` and a remount in read-only mode. However, as seen in Figure 8(c), Btrfs can recover from log-tree failures by attempting a full-tree commit immediately after the failure (as opposed to periodically during checkpointing). If the full-tree commit succeeds, then Btrfs ignores the log-tree failure and returns success for `fsync`. However, if there were another failure during the full-tree commit, then Btrfs would fail the `fsync` and remount in read-only mode. As Btrfs also performs a full-tree commit periodically during checkpointing, unlike ext4 and XFS, faults during checkpointing (trace in Figure 8(d)) result in a remount in read-only mode.

The multi-block append workload illustrates interesting behavior in Btrfs block allocation. If the first append fails, then the state of the file system, including the B-tree that tracks all free blocks, is reverted. However, the next append will continue to write at the (incorrectly) updated offset stored in the file descriptor, creating a hole in the file. Since the state of the B-tree was reverted, the deterministic block allocator will choose to allocate the same block again for the next append operation. Thus, if the fault to that particular block was transient, then the next `write` and `fsync` will succeed and there will simply be a one block hole in the file. If the fault to that particular block

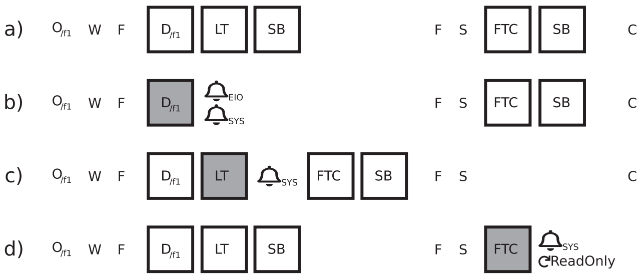
a) O/f1 W F D/f1 LT SB F S FTC SB C
b) O/f1 W F D/f1 EIO SYS F S FTC SB C
c) O/f1 W F D/f1 LT SYS FTC SB F S C
d) O/f1 W F D/f1 LT SB F S FTC SYS ReadOnly

Fig. 8. Blockviz traces for w_{su} on Btrfs:
The figure shows four traces corresponding to different fault injection configurations of dm-loki.

(a) when no faults are injected: In a copy-on-write file system like Btrfs, any modification to data or metadata involves creating a copy of the modified nodes in the tree. To avoid too much redundant I/O by forcing repeated copy-on-write for modified parts of the trees, Btrfs journals fsync-triggered copy-on-writes to a log tree (LT). The super block (SB) is also updated, as it contains a reference to the updated log tree root. During `sleep`, Btrfs checkpoints state by performing a full tree commit (FTC), which involves writing all modified trees and deleting items from the log tree, followed by a write to the superblock that contains references to all the tree roots.

(b) dm-loki configured to fail the data block write: On data block failure, Btrfs fails the `fsync` and reverts state. Unlike ext4 and XFS, we observe no write requests during the second `fsync`. However, the modification and reversal trigger an unnecessary full tree commit during `sleep`.

(c) dm-loki configured to fail a block write in the log tree: When Btrfs encounters a log tree write failure, it logs the error to syslog and starts a full tree commit. Since we only fail one particular block, the full tree commit succeeds and fsync does not fail. As there are no changes after the last full tree commit, there are no write requests during `sleep`.

(d) dm-loki configured to fail a block write during a full tree commit: If Btrfs encounters a write failure during a full tree commit, then it logs the error to syslog and remounts in read-only mode (↻ReadOnly). We observe this behavior for any full tree commit, both periodically (in w_{su}, w_{ma}, w_{dir} during `sleep`) and triggered on log tree failures as seen in subfigure (c). When triggered due to log tree failures, the fsync fails with errno set to EIO.

occurs multiple times, then future writes will continue to fail; as a result, Btrfs may cause more holes within a file than ext4 and XFS. However, unlike ext4 and XFS, the file does not have block overwrite failures.

During checkpointing, since Btrfs remounts in read-only mode on metadata write failures, we do not observe the same inconsistencies as described for ext4 when running the multi-file create workload w_{dir}.

3.3.4 File System Summary. We now present a set of observations for the file systems based on the questions from Section 3.2.5.

File System Behavior to `fsync` Failures. On all the three file systems, only data and journal-block failures lead to `fsync` failures (Q1). Metadata-block failures do not result in `fsync` failures as metadata blocks are written to the journal during an `fsync`. However, during a checkpoint, any metadata failure on XFS and Btrfs lead to unavailability (Q8) while ext4 logs the error and continues.[7]

[7] Ext4's error handling behavior for metadata has unintended side-effects, but we omit the results, as the rest of the article focuses on data-block failures.

On both modes of ext4 and XFS, metadata is persisted even after the file system encounters a data-block failure (Q2); timestamps are always updated in both file systems. Additionally, ext4 appends a new block to the file and updates the file size, while XFS does so only when followed by a future successful `fsync`. As a result, we find non-overwritten blocks in both the middle and end of files for ext4, but in only the middle for XFS (Q10). Btrfs does not persist metadata after a data-block failure. However, because the process file-descriptor offset is incremented, future `writes` and `fsyncs` cause a hole in the middle of the file (Q10).

Among the three, XFS is the only file system that retries metadata-block writes. However, none of them retry data or journal-block writes (Q3).

All the file systems mark the page clean even after `fsync` fails (Q4). In both modes of ext4 and XFS, the page contains the latest write, while Btrfs reverts the in-memory state to be consistent with what is on disk (Q5).

We note that even though all the file systems mark the page clean, this is not due to any behavior inherited from the VFS layer. Each file system registers its own handlers to write pages to disk (`ext4_writepages`, `xfs_vm_writepages`, and `btrfs_writepages`). However, each of these handlers call `clear_page_dirty_for_io` before submitting the bio request and do not set the dirty bit in case of failure to avoid memory leaks,[8] replicating the problem independently.

Failure Reporting. While all file systems report data-block failures by failing `fsync`, ext4 ordered mode, XFS, and Btrfs fail the immediate `fsync`. As ext4 data mode puts data in the journal, the first `fsync` succeeds and the next `fsync` fails. (Q6). All block write failures, irrespective of block type, are logged in the syslog (Q7).

After Effects. Journal block failures always lead to file-system unavailability. On XFS and Btrfs, metadata-block failures do so as well (Q8). While ext4 and Btrfs remount in read-only mode, XFS shuts down the file system (Q9). Holes and non-overwritten blocks (Q10) have been covered previously as part of Q2.

Recovery. None of the file systems alert the user to run a file-system checker. However, as Btrfs records intentionally created holes as zero-byte extents, holes created through fsync failures (as seen in w_{ma}) can be detected by btrfsck due to missing zero-byte extent information (Q11).

4 APPLICATION STUDY

We now focus on how applications are affected by `fsync` failures. In this section, we first describe our fault model with CuttleFS, followed by a description of the workloads, execution environment, and the errors we look for. Then, we present our findings for five widely used applications: Redis (v5.0.7), LMDB (v0.9.24), LevelDB (v1.22), SQLite (v3.30.1), and PostgreSQL (v12.0).

4.1 CuttleFS

We limit our study to how applications are affected by data-block failures, as journal-block failures lead to unavailability and metadata-block failures do not result in `fsync` failures (Section 3.3). Our fault model is simple: When an application writes data, we inject a single fault to a data block or a sector within it.

We build CuttleFS[9] [10]—a FUSE [42] file system to emulate the different file-system reactions to failures defined by our fault model. Instead of using the kernel's page cache, CuttleFS maintains its own page cache in user-space memory. Write operations modify user-space pages and mark them

[8] Ext4 focuses on the common case of users removing USB sticks while still in use. Dirty pages that can never be written to the removed USB stick have to be marked clean to unmount the file system and reclaim memory [26].

[9] Cuttlefish are sometimes referred to as the "chameleons of the sea" because of their ability to rapidly alter their skin color within a second. CuttleFS can change characteristics much faster.

dirty while read operations serve data from these pages. When an application issues an `fsync` system call, CuttleFS synchronizes data with the underlying file system.

CuttleFS has two modes of operation: trace mode and fault mode. In trace mode, CuttleFS tracks writes and identifies which blocks are eventually written to disk. This is different from just tracing a `write` system call, as an application may write to a specific portion of a file multiple times before it is actually flushed to disk.

In fail mode, CuttleFS can be configured to fail the *i*th write to a sector or block associated with a particular file. On `fsync` failure, as CuttleFS uses in-memory buffers, it can be directed to mark a page clean or dirty, keep the latest content, or revert the file to the previous state. Error reporting behavior can be configured to report failures immediately or on the next `fsync` call. In short, CuttleFS can react to `fsync` failures in any of the ways mentioned in Table 1 (Q4,5,6). Additionally, CuttleFS accepts commands to evict all or specific clean pages.

We configure CuttleFS to emulate the failure reactions of the file systems studied in Section 3.3. For example, to emulate ext4 ordered mode and XFS (as they both have similar failure reactions), we configure CuttleFS to mark the page clean, keep the latest content, and report the error immediately. Henceforth, when presenting our findings and referring to characteristics emulated by CuttleFS, we use CuttleFS$_{\text{ext4o,xfs}}$ for the above configuration. When the page is marked clean, has the latest content, but the error is reported on the next `fsync`, we use CuttleFS$_{\text{ext4d}}$. When the page is marked clean, the content matches what is on disk, and the error is reported immediately, we refer to it as CuttleFS$_{\text{btrfs}}$.

4.2 Workloads and Execution Environment

We run CuttleFS in trace mode and identify which blocks are written to by an application. For each application, we choose a simple workload that inserts a single key-value pair, a commonly used operation in many applications. We perform experiments both with an existing key (update) as well as a new key (insert). The keys can be of size 2 B or 1 KB.[10] The values can be of size 2 B or 12 KB. We run experiments for all four combinations. The large keys allow for the possibility of failing a single sector within the key and large values for pages within a value. Since SQLite and PostgreSQL are relational database management systems, we create a single table with two columns: keys and values.

Using the trace, we generate multiple failure sequences for each of the identified blocks and sectors within them. We then repeat the experiment multiple times with CuttleFS in fault mode, each time with a different failure sequence and file-system reaction. To observe the effects after a fault, we dump all key-value pairs before and after the workload.

We look for the following types of errors when performing the experiments:

- **OldValue (OV):** The system returns the new value for a while but then reverts to an old value, or the system conveys a successful response but returns the old value later on.
- **FalseFailure (FF):** The system informs the user that the operation failed but returns the new value in the future.
- **KeyCorruptions (KC)** and **ValueCorruptions (VC):** Corrupted keys or values are obliviously returned.
- **KeyNotFound (KNF):** The system informs the user that it has successfully inserted a key but it cannot be found later on, or the system fails to update a key to a new value but the old key-value pair disappears as well.

[10] As LMDB limits key sizes to 511 B, we use key sizes of 2 B and 511 B for LMDB experiments.

Table 2. Findings for Applications on fsync Failure

	A=KeepGoing	A=Restart
BC=Keep	╲	╱
BC=Evict	│	−

Applications		ext4o,xfs = clean differs immediate					ext4d = clean differs next fsync					btrfs = clean matches immediate				
		OV	**FF**	**KC**	**VC**	**KNF**	**OV**	**FF**	**KC**	**VC**	**KNF**	**OV**	**FF**	**KC**	**VC**	**KNF**
Redis		−		−	−	−	−		−	−	−	⌿				⌿
LMDB			−				+			+						
LevelDB			╱				−		+	+	+					
SQLite	Rollback		+		+			+		−	−		✳			
	WAL		╱							−	−					
PostgreSQL	Default		⌿				−				−					
	Direct I/O						−				−					

The table lists the different types of errors that manifest for applications when fsync fails due to a data-block write fault. The errors (OV, FF, KC, VC, KNF) are described in Section 4.2. We group columns depending on how a file system reacts to an fsync failure according to our findings in Section 3.3 for Q4, Q5, and Q6. For example, both ext4 ordered and XFS (ext4o,xfs) mark a page **clean**, the page **differs** in in-memory and on-disk content, and the fsyncfailure is reported **immediately**. For each application, we describe when the error manifests, in terms of combinations of the four different execution environment factors (Section 4.2) whose symbols are provided at the top left corner. For example, OldValue manifests in Redis in the first group (ext4-ordered, XFS) only on (A)App=Restart,(BC)BufferCache=Evict. However, in the last group (Btrfs), the error manifests both on App=Restart,BufferCache=Evict as well as App=Restart,BufferCache=Keep, depicted as a combination of the two symbols.

We also identify the factors within the execution environment that cause all these errors to be manifested. If an application maintains its own in-memory data structures, then some errors may occur only when an application restarts and rebuilds in-memory state from the file system. Alternatively, the manifestation of these errors may depend on state changes external to the application, such as a single page eviction or a full page cache flush. We encode these different scenarios as:

- **App=KeepGoing:** The application continues without restarting.
- **App=Restart:** The application restarts either after a crash or a graceful shutdown. This forces the application to rebuild in-memory state from disk.
- **BufferCache=Keep:** No evictions take place.
- **BufferCache=Evict:** One or more clean pages are evicted.

Note that BufferCache=Evict can manifest by clearing the entire page cache, restarting the file system, or just evicting clean pages due to memory pressure. A full system restart would be the combination of App=Restart and BufferCache=Evict, which causes a loss of both clean and dirty pages in memory while also forcing the application to restart and rebuild state from disk.

Configuring CuttleFS to fail a certain block and react according to one of the file-system reactions while the application runs only addresses App=KeepGoing and BufferCache=Keep. The remaining three scenarios are addressed as follows: To simulate App=Restart and BufferCache=Keep, we restart the application and dump all key-value pairs, ensuring that no page in CuttleFS is evicted. To address the remaining two scenarios, we instruct CuttleFS to evict clean pages for both App=KeepGoing and App=Restart.

4.3 Findings

We configured all five applications to run in the form that offers most durability and discuss what they are in their respective sections. Table 2 summarizes the per-application results across different failure characteristics.

Note that these results are only for the simple workload that inserts a single key-value pair. A complex workload may exhibit more errors or mask the ones we observe.

Redis: Redis is an in-memory data-structure store, used as a database, cache, and message broker. By default, it periodically snapshots in-memory state to disk. However, for better durability guarantees, it provides options for writing every operation that modifies the store to an append-only file *(aof)* [22] and how often to fsync the *aof*. In the event of a crash or restart, Redis rebuilds in-memory state by reading the contents of the *aof*.

We configure Redis to fsync the file for every operation, providing strong durability. Thus, whenever Redis receives a request like an insert operation that modifies state, it writes the request to the *aof* and calls fsync. However, Redis trusts the file system to successfully persist the data and does not check the fsync return code. Regardless of whether fsync fails or not, Redis returns a successful response to the client.

As Redis returns a successful response to the client irrespective of fsync failure, FalseFailures do not occur. Since Redis reads from disk only when rebuilding in-memory state, errors may occur only during App=Restart.

On CuttleFS$_{\text{ext4o,xfs}}$ and CuttleFS$_{\text{ext4d}}$, Redis exhibits OldValue, KeyCorruption, ValueCorruption, and KeyNotFound errors. However, as seen in Table 2, these errors occur only on BufferCache=Evict and App=Restart. On BufferCache=Keep, the page contains the latest write, which allows Redis to rebuild the latest state. However, when the page is evicted, future reads will force a read from disk, causing Redis to read whatever is on that block. OldValue and KeyNotFound errors manifest when a fault corrupts the *aof* format. When Redis restarts, it either ignores these entries when scanning the *aof*, or recommends running the *aof checker*, which truncates the file to the last non-corrupted entry. A KeyCorruption and ValueCorruption manifest when the fault is within the key or value portion of the entry.

On CuttleFS$_{\text{btrfs}}$, Redis exhibits OldValue and KeyNotFound errors. These errors occur on App=Restart, regardless of buffer-cache state. When Redis restarts, the entries are missing from the *aof* as the file was reverted, and thus, the insert or update operation is not applied.

LMDB: **Lightning Memory-Mapped Database (LMDB)** is an embedded key-value store that uses B+Tree data structures whose nodes reside in a single file. The first two pages of the file are metadata pages, each of which contain a transaction ID and the location of the root node. Readers always use the metadata page with the latest transaction ID, while writers make changes and update the older metadata page.

LMDB uses a copy-on-write bottom-up strategy [14] for committing write transactions. All new nodes from leaf to root are written to unused or new pages in the file, followed by an fsync. An fsync failure terminates the operation without updating the metadata page and notifies the user. If fsync succeeds, then LMDB proceeds to update the old metadata page with the new root location and transaction ID, followed by another fsync.[11] If fsync fails, then LMDB writes an old transaction ID to the metadata page in memory, preventing future readers from reading it.

On CuttleFS$_{\text{ext4o,xfs}}$, LMDB exhibits FalseFailures. When LMDB writes the metadata page, it only cares about the transaction ID and new root location, both of which are contained in a single sector. Thus, even though the sector is persisted to disk, failures in the seven other sectors of the metadata page can cause an fsync failure.[12] As mentioned earlier, LMDB writes an old transaction ID (say, ID1) to the metadata page in memory and reports a failure to the user. However, on BufferCache=Evict and App=Restart (such as a machine crash and restart), ID1 is lost, as it was

[11] To be precise, LMDB does not do a write followed by an fsync for metadata page updates. Instead, it uses a file descriptor that is opened in O_SYNC mode. On a write, only the metadata page is flushed to disk. On failure, it uses a normal file descriptor.

[12] CuttleFS can fail the *i*th write to a sector or block(Section 4.1). We observed FalseFailures in LMDB when CuttleFS was configured to fail writes to sectors in the metadata pages.

only written to memory and not persisted. Thus, readers read from the latest transaction ID, which is the previously failed transaction.

LMDB does not exhibit FalseFailures in CuttleFS$_{ext4d}$, as the immediate successful fsync results in a success to the client. Instead, ValueCorruptions and OldValue errors occur on BufferCache=Evict, regardless of whether the application restarts or not. ValueCorruptions occur when a block containing a part of the value experiences a fault. As LMDB *mmaps*() the file and reads directly from the page cache, BufferCache=Evict such as a page eviction leads to reading the value of the faulted block from disk. OldVersion errors occur when the metadata page experiences a fault. The file system responds with a successful fsync initially (as data is successfully stored in the ext4 journal). For a short time, the metadata page has the latest transaction ID. However, when the page is evicted, the metadata page reverts to the old transaction ID on disk, resulting in readers reading the old value. KeyCorruptions do not occur, as the maximum allowed key size is 511 B.

As CuttleFS$_{btrfs}$ reports errors immediately, it does not face the problems seen in CuttleFS$_{ext4d}$. FalseFailures do not occur, as the file is reverted to its previous consistent state. We observe this same pattern in many of the applications and omit them from the rest of the discussion unless relevant.

LevelDB: LevelDB is a widely used key-value store based on LSM trees. It stores data internally using MemTables and SSTables [36]. Additionally, LevelDB writes operations to a log file before updating the MemTable. When a MemTable reaches a certain size, it becomes immutable and is written to a new file as an SSTable. SSTables are always created and never modified in place. On a restart, if a log file exists, then LevelDB creates an SSTable from its contents.

We configure LevelDB to fsync the log after every write, for stronger durability guarantees. If fsync fails, then the MemTable is not updated and the user is notified about the failure. If fsync fails during SSTable creation, then the operation is cancelled and the SSTable is left unused.

On CuttleFS$_{ext4o,xfs}$, as seen in Table 2, LevelDB exhibits FalseFailures only on App=Restart with BufferCache=Keep. When LevelDB is notified of fsync failure to the log file, the user is notified of the failure. However, on restart, since the log entry is in the page cache, LevelDB includes it while creating an SSTable from the log file. Read operations from this point forward return the new value, reflecting FalseFailures. FalseFailures do not occur on BufferCache=Evict as LevelDB is able to detect invalid entries through CRC checksums [36]. Faults in the SSTable are detected immediately and do not cause any errors, as the newly generated SSTable is not used by LevelDB in case of a failure.

On CuttleFS$_{ext4d}$, LevelDB exhibits KeyNotFound and OldVersion errors when faults occur in the log file. When inserting a key-value pair, fsync returns successfully, allowing future read operations to return the new value. However, on BufferCache=Evict and App=Restart, LevelDB rejects the corrupted log entry and returns the old value for future read operations. Depending on whether we insert a new or existing key, we observe KeyNotFound or OldVersion errors when the log entry is rejected. Additionally, LevelDB exhibits KeyCorruption, ValueCorruption, and KeyNotFound errors for faults that occur in the SSTables. Ext4 data mode may only place the data in the journal and return a successful fsync. Later, during checkpointing, the SSTable is corrupted due to the fault. These errors manifest only on BufferCache=Evict, either while the application is running or on restart, depending on when the SSTable is read from disk.

SQLite: SQLite is an embedded RDBMS that uses BTree data structures. A separate BTree is used for each table and index but all BTrees are stored in a single file on disk, called the “main database file” *(maindb)*. During a transaction, SQLite stores additional information in a second file called the “rollback journal” *(rj)* or the “write-ahead log” *(wal)* depending on which mode it is operating in. In the event of a crash or restart, SQLite uses these files to ensure that committed or rolled-back transactions are reflected in the *maindb*. Once a transaction completes, these files are deleted. We perform experiments for both modes.

SQLite RollBack: In rollback journal mode, before SQLite modifies its user-space buffers, it writes the original contents to the *rj*. On commit, the *rj* is `fsyncd`. If it succeeds, then SQLite writes a header to the *rj* and `fsyncs` again (2 `fsyncs` on the *rj*). If a fault occurs at this point, then only the state in the user-space buffers need to be reverted. If not, then SQLite proceeds to write to the *maindb* so it reflects the state of the user-space buffers. *maindb* is then `fsyncd`. If the `fsync` fails, then SQLite needs to rewrite the old contents to the *maindb* from the *rj* and revert the state in its user-space buffers. After reverting the contents, the *rj* is deleted.

On CuttleFS$_{\text{ext4o,xfs}}$, SQLite Rollback exhibits FalseFailures and ValueCorruptions on BufferCache=Evict, regardless of whether the application restarts or not. When faults occur in the *rj*, SQLite chooses to revert in-memory state using the *rj* itself, as it contains just enough information for a rollback of the user-space buffers. This approach works well as long as the latest contents are in the page cache. However, on BufferCache=Evict, when SQLite reads the *rj* to rollback in-memory state, the *rj* does not contain the latest write. As a result, SQLite's user-space buffers can still have the new contents (FalseFailure) or a corrupted value, depending on where the fault occurs.

SQLite Rollback exhibits FalseFailures in CuttleFS$_{\text{ext4d}}$ for the same reasons mentioned above as the `fsync` failure is caught on the second `fsync` to the *rj*. Additionally, due to the late error reporting in CuttleFS$_{\text{ext4d}}$, SQLite Rollback exhibits ValueCorruption and KeyNotFound errors when faults occur in the *maindb*. SQLite sees a successful `fsync` after writing data to the *maindb* and proceeds to delete the *rj*. However, on App=Restart and BufferCache=Evict, the above-mentioned errors manifest depending on where the fault occurs.

On CuttleFS$_{\text{btrfs}}$, SQLite Rollback exhibits FalseFailures for the same reasons mentioned above. However, they occur irrespective of whether buffer-cache state changes due to the fact that the contents in the *rj* are reverted. As there is no data in the *rj* to recover from, SQLite leaves the user-space buffers untouched. ValueCorruptions cannot occur, as no attempt is made to revert the in-memory content.

SQLite WAL: Unlike SQLite Rollback, changes are written to a write-ahead log *(wal)* on a transaction commit. SQLite calls `fsync` on the *wal* and proceeds to change in-memory state. If `fsync` fails, then SQLite immediately returns a failure to the user. If SQLite has to restart, then it rebuilds state from the *maindb* first and then changes state according to the entries in the *wal*. To ensure that the *wal* does not grow too large, SQLite periodically runs a Checkpoint Operation to modify *maindb* with the contents from the *wal*.

On CuttleFS$_{\text{ext4o,xfs}}$, as seen in Table 2, SQLite WAL exhibits FalseFailures only on App=Restart with BufferCache=Keep, for reasons similar to LevelDB. It reads valid log entries from the page cache even though they might be invalid due to faults on disk.

On CuttleFS$_{\text{ext4d}}$, SQLite WAL exhibits ValueCorruption and KeyNotFound Errors when there are faults in the *maindb* during a Checkpoint Operation for the same reasons mentioned in SQLite Rollback.

PostgreSQL: PostgreSQL is an object-relational database system that maintains one file per database table. On startup, it reads the on-disk tables and populates user-space buffers. Similar to SQLite WAL, PostgreSQL reads entries from the write-ahead log *(wal)* and modifies user-space buffers accordingly. Similar to SQLite WAL, PostgreSQL runs a checkpoint operation, ensuring that the *wal* does not grow too large. We evaluate two configurations of PostgreSQL: the default configuration and a DirectIO configuration.

PostgreSQL Default: In the default mode, PostgreSQL treats the *wal* like any other file, using the page cache for reads and writes. PostgreSQL notifies the user of a successful *commit* operation only after an `fsync` on the *wal* succeeds. During a checkpoint, PostgreSQL writes data from its user-space buffers into the table and calls `fsync`. If the `fsync` fails, then PostgreSQL, aware of the problems with `fsync` [8], chooses to crash. Doing so avoids truncating the *wal* and ensures that checkpointing can be retried later.

On CuttleFS$_{\text{ext4o,xfs}}$, PostgreSQL exhibits FalseFailures for reasons similar to LevelDB. While App=Restart is necessary to read the entry from the log, BufferCache=Evict is not. Further, the application restart cannot be avoided, as PostgreSQL intentionally crashes on an `fsync` failure. On BufferCache=Keep, PostgreSQL reads a valid log entry in the page cache. On BufferCache=Evict, depending on which block experiences the fault, PostgreSQL either accepts or rejects the log entry. FalseFailures manifest when PostgreSQL accepts the log entry. However, if the file system were to also crash and restart, then the page cache would match the on-disk state, causing PostgreSQL to reject the log entry. Unfortunately, ext4 currently does not behave as expected with mount options `data_err=abort` and `errors=remount-ro` (Section 3.3.1).

Due to the late error reporting in CuttleFS$_{\text{ext4d}}$, as seen in Table 2, PostgreSQL exhibits OldVersion and KeyNotFound Errors when faults occur in the database table files. As PostgreSQL maintains user-space buffers, these errors manifest only on BufferCache=Evict with App=Restart. During a checkpoint operation, PostgreSQL writes the user-space buffers to the table. As the fault is not yet reported, the operation succeeds and the *wal* is truncated. If the page corresponding to the fault is evicted and PostgreSQL restarts, then it will rebuild its user-space buffers using an incorrect on-disk table file. The errors are exhibited depending on where the fault occurs. While KeyNotFound errors occur in other applications when a new key is inserted, PostgreSQL *loses existing keys on updates* as it modifies the table file in-place.

PostgreSQL DIO: In the DirectIO mode, PostgreSQL bypasses the page cache and writes to the *wal* using DirectIO. The sequence of operations during a transaction commit and a checkpoint are exactly the same as the default mode.

FalseFailures do not occur as the page cache is bypassed. However, OldVersion and KeyNotFound errors still occur in CuttleFS$_{\text{ext4d}}$ for the same reasons mentioned above, as writes to the database table files do not use DirectIO.

5 DISCUSSION

We now present a set of observations and lessons for handling `fsync` failures across file systems and applications.

#1: Existing file systems do not handle fsync failures uniformly. In an effort to hide cross-platform differences, POSIX is intentionally vague on how failures are handled. Thus, different file systems behave differently after an `fsync` failure (as seen in Table 1), leading to non-deterministic outcomes for applications that treat all file systems equally. *We believe that the POSIX specification for* `fsync`*needs to be clarified and the expected failure behavior described in more detail.*

#2: Copy-on-Write file systems such as Btrfs handle fsync failures better than existing journaling file systems such as ext4 and XFS. Btrfs uses new or unused blocks when writing data to disk; the entire file system moves from one state to another on success and no in-between states are permitted. Such a strategy defends against corruptions when only some blocks contain newly written data. *File systems that use copy-on-write may be more generally robust to* `fsync`*failures than journaling file systems.*

#3: Ext4 data mode provides a false sense of durability. Application developers sometimes choose to use a data journaling file system despite its lower performance, because they believe data mode is more durable [12]. Ext4 data mode does ensure data and metadata are in a "consistent state," but only from the perspective of the file system. As seen in Table 2, application-level inconsistencies are still possible. Furthermore, applications cannot determine whether an error received from `fsync` pertains to the most recent operation or an operation sometime in the past. *When failed intentions are a possibility, applications need a stronger contract with the file system, notifying them of relevant context such as data in the journal and which blocks were not successfully written.*

#4: Existing file-system fault-injection tests are devoid of workloads that continue to run post failure. While all file systems perform fault-injection tests, they are mainly to ensure that the file system is consistent after encountering a failure. Such tests involve shutting down the file system soon after a fault and checking if the file system recovers correctly when restarted. *We believe that file-system developers should also test workloads that continue to run post failure, and see if the effects are as intended.* Such effects should then be documented. File-system developers can also quickly test the effect on certain characteristics by running those workloads on CuttleFS before changing the actual file system.

#5: Application developers write OS-specific code, but are not aware of all OS-differences. The FreeBSD VFS layer chooses to re-dirty pages when there is a failure (except when the device is removed) [6] while Linux hands over the failure handling responsibility to the individual file systems below the VFS layer (Section 3.3.4). *We hope that the Linux file-system maintainers will adopt a similar approach in an effort to handle* `fsync`*failures uniformly across file systems.* Note that it is also important to think about when to classify whether a device has been removed. For example, while storage devices connected over a network are not really as permanent as local hard disks, they are more permanent than removable USB sticks. Temporary disconnects over a network need not be perceived as device removal and re-attachment; pages associated with such a device can be re-dirtied on write failure.

#6: Application developers do not target specific file systems. We observe that data-intensive applications configure their durability and error-handling strategies according to the OS they are running on, but treat all file systems on a specific operating system equally. Thus, as seen in Table 2, a single application can manifest different errors depending on the file system. *If the POSIX standard is not refined, then applications may wish to handle* `fsync`*failures on different file systems differently.* Alternatively, applications may choose to code against *failure handling characteristics* as opposed to specific file systems, but this requires file systems to expose some interface to query characteristics such as "Post Failure Page State/Content" and "Immediate/Delayed Error Reporting."

#7: Applications employ a variety of strategies when fsync fails, but none are sufficient. As seen in Section 4.3, Redis chooses to trust the file system and does not even check `fsync` return codes; LMDB, LevelDB, and SQLite revert in-memory state and report the error to the application while PostgreSQL chooses to crash. We have seen that none of the applications retry `fsync` on failure; application developers appear to be aware that pages are marked clean on `fsync` failure and another `fsync` will not flush additional data to disk. Despite the fact that applications take great care to handle a range of errors from the storage stack (e.g., LevelDB writes CRC Checksums to detect invalid log entries and SQLite updates the header of the rollback journal only after the data is persisted to it), data durability cannot be guaranteed as long as `fsync` errors are not handled correctly. *While no one strategy is always effective, the approach currently taken by PostgreSQL to use direct IO may best handle* `fsync`*failures.* If file systems do choose to report failure handling characteristics in a standard format, then applications may be able to employ better strategies. For example, applications can choose to keep track of dirtied pages and re-dirty them by reading and writing back a single byte if they know that the page content is not reverted on failure (ext4, XFS). On Btrfs, one would have to keep track of the page as well as its content. For applications that access multiple files, it is important to note that the files can exist on different file systems.

#8: Applications run recovery logic that accesses incorrect data in the page cache. Applications that depend on the page cache for faster recovery are susceptible to FalseFailures. As seen in LevelDB, SQLite, and PostgreSQL, when the *wal* incurs an fsync failure, the applications fail the operation and notify the user; in these cases, while the on-disk state may be corrupt, the entry in the page cache is valid; thus, an application that recovers state from the *wal* might read partially valid entries from the page cache and incorrectly update on-disk state. *Applications should read the on-disk content of files when performing recovery.*

#9: Application recovery logic is not tested with low-level block faults. Applications test recovery logic and possibilities of data loss by either mocking system call return codes or emulating crash-restart scenarios, limiting interaction with the underlying file system. As a result, failure handling logic by the file system is not exercised. *Applications should test recovery logic using low-level block injectors that force underlying file-system error handling.* Alternatively, they could use a fault injector like CuttleFS that mimics different file-system error-handling characteristics.

6 RELATED WORK

In this section, we discuss how our work builds upon and differs from past studies in key ways. We include works that study file systems through fault injection, error handling in file systems, and the impact of file-system faults on applications.

Our study on how file systems react to failures is related to work done by Prabhakaran et al. with IRON file systems [53] and a more recent study conducted by Jaffer et al. [43]. Other works study specific file systems such as NTFS [31] and ZFS [62]. All these studies inject failures beneath the file system and analyze if and how file systems detect and recover from them. These studies use system-call workloads (e.g., writes and reads) that make the file system interact with the underlying device.

While prior studies do exercise some portions of the fsync path through single system-call operations, they do not exercise the checkpoint path. More importantly, in contrast to these past efforts, our work focuses specifically on the *in-memory* state of a file system and the effects of *future operations* on a file system that has encountered a write fault. Specifically, in our work, we choose workloads that continue after a fault has been introduced. Such workloads help in understanding the after-effects of failures during fsync such as masking of errors by future operations, fixing the fault, or exacerbating it.

Mohan et al. [49] use bounded black-box crash testing to exhaustively generate workloads and discover many crash-consistency bugs by simulating power failures at different persistence points. Our work focuses on transient failures that may not necessarily cause a file system to crash and the effect on applications even though a file system may be consistent. Additionally, we inject faults in the middle of an fsync as opposed to after a successful fsync (persistence point).

Gunawi et al. describe the problem of failed intentions [39] in journaling file systems and suggest chained transactions to handle such faults during checkpointing. Another work develops a static-analysis technique named Error Detection and Propagation [40] and conclude that file systems neglect many write errors. Even though the Linux kernel has improved its block-layer error handling [11], file systems may still neglect write errors. Our results are purely based on injecting errors in bio requests that the file system can detect.

Vondra describes how certain assumptions about fsync behavior led to data loss in PostgreSQL [60]. The data loss behavior was reproduced using a device mapper with the dm-error target, which inspired us to build our own fault injector (dm-loki [4]) atop the device mapper, similar to dm-inject [43]. Additionally, the FSQA suite (xfstests) [7] emulates write errors using the dm-flakey target [5]. While dm-flakey is useful for fault-injection testing, faults are injected based on current time; the device is available for x seconds and then exhibits unreliable behavior for y seconds (x and y being configurable). Furthermore, any change in configuration requires suspending the device. To increase determinism and avoid relying on time, dm-loki injects faults based on access patterns (e.g., fail the second and fourth write to block 20) and is capable of accepting configuration changes without device suspension.

Recent work has shifted the focus to study the effects of file-system faults in distributed storage systems [37] and high-performance parallel systems [32]. Similarly, our work focuses on understanding how file systems and applications running on top of them behave in the presence of failures.

7 CONCLUSIONS

Applications that care about data must care about how data is written to stable storage. IO requests such as `read` and `write` system calls do not always translate to device-level requests, as the page cache buffers contents in volatile memory. Therefore, applications are able to leverage the performance benefits of the page cache without any modification to their source code. However, in cases where applications require specific data in the page cache to be written to disk immediately, `fsync` must be invoked.

In this article, we described why `fsync` may fail and how three different file systems (ext4, XFS, Btrfs) currently handle these failures. We also described how applications use `fsync` to provide durability guarantees and the consequences of incorrectly handling `fsync` failures.

7.1 File Systems

File systems are tasked with a heavy burden. They must be able to support applications that care about performance as well as those that care about durability. They must make both options efficient while leaving the tradeoff decision to the application developers. File-system developers have spent considerable time and effort optimizing the common case and ensuring durability, year after year, as hardware technology advances. However, while most hardware advancements introduce increased performance and robustness, advancements in availability have changed the way devices fail. Changes from the fail-stop model to the fail-partial model [52, 53] required changes in how failures were handled. With incomplete specifications for failure handling, file-system developers chose to interpret the standard differently, causing non-uniform behavior across file systems, as seen in Section 3.3.

We hope that file-system developers will eventually agree on a standard way to handle `fsync` failures. However, the current situation requires application developers to be aware of differences between file systems. Unfortunately, the post-failure characteristics described in this article were either not documented or not easily accessible; most of the error-handling knowledge lies with file-system maintainers. Tracing and visualization tools like `blockviz` (Section 3.2.3) can provide useful insights into file-system behavior. While application developers are free to download and use these tools, we hope that file-system maintainers can use them to aid in better documentation; short traces can clearly explain file-system behavior for common workloads.

A good fault-injection toolchain facilitates easier failure-handling studies and testing. While the `device-mapper` framework has a few modules to inject errors, it is far from complete. The `dm-loki` kernel module (Section 3.2.2) and an error emulation tool in prior work by Jaffer et al. [43] are useful contributions that can aid future fault-injection studies. However, such tools currently work under the neatly abstracted block layer in the kernel. With the advent of low-latency storage devices and kernel-bypass techniques, fault-injection toolchains must change accordingly.

7.2 Applications

Most applications that provide durability guarantees are aware that fsync can fail. Of the five applications we studied, Redis was the only one that did not have any fsync error-handling code. However, due to the differences in file-system fsync failure handling, no single application-level error-handling strategy works well for all file systems.

One way that applications expand their user base is by supporting multiple operating systems. Making an application portable can take significant effort just for the commonly used features. Given the current state of non-uniform post-failure characteristics, achieving correct failure handling on every file system and operating system is a herculean task. However, informing users of durability guarantees and potential data-loss concerns on different file systems they support would be an excellent addition to the already well maintained online documentation that current applications provide.

While many of our application findings are tied to the fact that file systems mark dirty pages clean even on a failed write, they may remain relevant even after the issue is fixed. If a file system were to re-dirty the previously marked clean pages belonging to the failed write, then an application may still recover state from the dirty pages rather than from disk. Such an application may then proceed to serve results that could potentially be lost if the machine crashes before the dirty pages are written to disk successfully. However, if file systems choose to revert the content of all dirty pages involved in a failed fsync (as seen in Btrfs), then our findings will no longer be applicable to applications using write-ahead log strategies with fsync error-handling code.

Applications that choose to test their error-handling strategies and provide durability guarantees must not rely entirely on mocking system call return values. Since error-handling code may involve accessing state on the erroneous system, mocks must also replicate state of said system after the error. While injecting faults at the device level is a more reliable method, it is challenging to express fault-injection intents at the block level without access to file-level abstractions. We believe that tools like CuttleFS (Section 4.1) that offer the ability to inject faults for certain file offsets and evict cached pages on demand will be useful for applications that wish to test the effectiveness of their error-handling strategies.

ACKNOWLEDGMENTS

We thank the *ACM TOS* reviewers for their detailed reviews on this extended version. We thank Peter Macko (our shepherd), the anonymous reviewers, and the members of ADSL for their insightful comments and suggestions for our prior submission to ATC'20 [54]. We thank CloudLab [35] for providing a great environment to run our experiments.

REFERENCES

[1] SQLite. 2020. Atomic Commit in SQLite. Retrieved from https://www.sqlite.org/atomiccommit.html.

[2] Anthony Rebello. 2020. Bug-207729 Mounting EXT4 with data_err=abort does not abort journal on data block write failure. Retrieved from https://bugzilla.kernel.org/show_bug.cgi?id=207729.

[3] Allen Lai. 2020. Bug-27805553 HARD ERROR SHOULD BE REPORTED WHEN FSYNC() RETURN EIO. Retrieved from https://github.com/mysql/mysql-server/commit/8590c8e12a3374eeccb547359750a9d2a128fa6a.

[4] Anthony Rebello. 2020. Custom Fault Injection Device Mapper Target: dm-loki. Retrieved from https://github.com/WiscADSL/dm-loki.

[5] The Linux Kernel Organization. 2020. Device Mapper: dm-flakey. Retrieved from https://www.kernel.org/doc/html/latest/admin-guide/device-mapper/dm-flakey.html.

[6] The FreeBSD Project. 2020. FreeBSD VFS Layer re-dirties pages after failed block write. Retrieved from https://github.com/freebsd/freebsd/blob/0209fe3398be56e5e042c422a96a4fbc654247f4/sys/kern/vfs_bio.c#L2646.

[7] The Linux Kernel Organization. 2020. FSQA (xfstests). Retrieved from https://git.kernel.org/pub/scm/fs/xfs/xfstests-dev.git/about/.

[8] Thomas Munro and Craig Ringer. 2020. Fsync Errors—PostgreSQL wiki. Retrieved from https://wiki.postgresql.org/wiki/Fsync_Errors.

[9] The Linux Kernel Organization. 2020. fsync(2)—Linux Programmer's Manual. Retrieved from http://man7.org/linux/man-pages/man2/fdatasync.2.html.

[10] Anthony Rebello. 2020. FUSE file system to emulate different file-system failure reactions: CuttleFS. Retrieved from https://github.com/WiscADSL/cuttlefs.

[11] Jonathan Corbet. 2020. Improved block-layer error handling. Retrieved from https://lwn.net/Articles/724307/.

[12] The Stack Exchange network. 2020. Is data=journal safer for Ext4 as opposed to data=ordered? Retrieved from https://unix.stackexchange.com/q/127235.

[13] Google. 2020. LevelDB. Retrieved from https://github.com/google/leveldb.

[14] Howard Chu. 2020. Lightning Memory-Mapped Database Manager (LMDB). Retrieved from http://www.lmdb.tech/doc/.

[15] The Linux Kernel Organization. 2020. Man Pages: dmsetup. Retrieved from https://man7.org/linux/man-pages/man8/dmsetup.8.html.

[16] The Linux Kernel Organization. 2020. Man Pages: losetup. Retrieved from https://man7.org/linux/man-pages/man8/losetup.8.html.

[17] IEEE and The Open Group. 2020. POSIX Specification for fsync. Retrieved from https://pubs.opengroup.org/onlinepubs/9699919799/functions/fsync.html.

[18] The PostgreSQL Global Development Group. 2020. PostgreSQL. Retrieved from https://www.postgresql.org/.

[19] The PostgreSQL Global Development Group. 2020. PostgreSQL: Write-Ahead Logging (WAL). Retrieved from https://www.postgresql.org/docs/current/wal-intro.html.

[20] Craig Ringer. 2020. PostgreSQL's handling of fsync() errors is unsafe and risks data loss at least on XFS. Retrieved from https://www.postgresql.org/message-id/flat/CAMsr+YHh+5Oq4xziwwoEfhoTZgr07vdGG+hu=1adXx59aTeaoQ@mail.gmail.com

[21] Redis Labs. 2020. Redis. Retrieved from https://redis.io/.

[22] Redis Labs. 2020. Redis Persistence. Retrieved from https://redis.io/topics/persistence.

[23] SQLite. 2020. SQLite. Retrieved from https://www.sqlite.org/index.html.

[24] SQLite. 2020. SQLite Write-Ahead Logging. Retrieved from https://www.sqlite.org/wal.html.

[25] SystemTap. 2020. SystemTap. Retrieved from https://sourceware.org/systemtap/.

[26] Theodore Ts'o. 2020. Why does ext4 clear the dirty bit on I/O error? Retrieved from https://www.postgresql.org/message-id/edc2e4d5-5446-e0db-25da-66db6c020cc3%40commandprompt.com

[27] WiredTiger. 2020. WT-4045 Don't retry fsync calls after EIO failure. Retrieved from https://github.com/wiredtiger/wiredtiger/commit/ae8bccce3d8a8248afa0e4e0cf67674a43dede96.

[28] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. 2018. *Operating Systems: Three Easy Pieces (1st ed.)*. Arpaci-Dusseau Books.

[29] Lakshmi N. Bairavasundaram, Garth Goodson, Bianca Schroeder, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2008. An analysis of data corruption in the storage stack. In *Proceedings of the 6th USENIX Symposium on File and Storage Technologies (FAST'08)*. 223–238.

[30] Lakshmi N. Bairavasundaram, Garth R. Goodson, Shankar Pasupathy, and Jiri Schindler. 2007. An analysis of latent sector errors in disk drives. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'07)*. 289–300.

[31] Lakshmi N. Bairavasundaram, Meenali Rungta, Nitin Agrawal, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Michael M. Swift. 2008. Analyzing the effects of disk-pointer corruption. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'08)*. 502–511.

[32] Jinrui Cao, Om Rameshwar Gatla, Mai Zheng, Dong Dai, Vidya Eswarappa, Yan Mu, and Yong Chen. 2018. PFault: A general framework for analyzing the reliability of high-performance parallel file systems. In *Proceedings of the International Conference on Supercomputing*. 1–11.

[33] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2013. Optimistic crash consistency. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*. 228–243.

[34] Vijay Chidambaram, Tushar Sharma, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2012. Consistency without ordering. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST'12)*. 101–116.

[35] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott,

Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'19)*. 1–14.

[36] Christian Forfang. 2014. *Evaluation of High Performance Key-value Stores*. Master's thesis. Norwegian University of Science and Technology.

[37] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2017. Redundancy does not imply fault tolerance: Analysis of distributed storage reactions to single errors and corruptions. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST'17)*. 149–165.

[38] Gregory R. Ganger and Yale N. Patt. 1994. Metadata update performance in file systems. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation (OSDI'94)*. 49–60.

[39] Haryadi S. Gunawi, Vijayan Prabhakaran, Swetha Krishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2007. Improving file system reliability with I/O shepherding. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP'07)*. 293–306.

[40] Haryadi S. Gunawi, Cindy Rubio-González, Remzi H. Arpaci-Dusseau Andrea C. Arpaci-Dusseau, and Ben Liblit. 2008. EIO: Error handling is occasionally correct. In *Proceedings of the 6th USENIX Symposium on File and Storage Technologies (FAST'08)*. 207–222.

[41] Robert Hagmann. 1987. Reimplementing the Cedar file system using logging and group commit. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP'87)*. 155–162.

[42] FUSE (Filesystem in Userspace). 2020. The reference implementation of the Linux FUSE (Filesystem in Userspace) interface. Retrieved from https://github.com/libfuse/libfuse.

[43] Shehbaz Jaffer, Stathis Maneas, Andy Hwang, and Bianca Schroeder. 2019. Evaluating file system reliability on solid state drives. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'19)*. 783–797.

[44] Hannu H. Kari. 1997. *Latent Sector Faults and Reliability of Disk Arrays*. Ph.D. Dissertation. Helsinki University of Technology.

[45] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, Carol Willing, and Jupyter development team. 2016. Jupyter notebooks—A publishing format for reproducible computational workflows. In *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, Fernando Loizides and Birgit Scmidt (Eds.). IOS Press, Netherlands, 87–90. Retrieved from https://eprints.soton.ac.uk/403913/.

[46] Andrew Krioukov, Lakshmi N. Bairavasundaram, Garth R. Goodson, Kiran Srinivasan, Randy Thelen, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2008. Parity lost and parity regained. In *Proceedings of the 6th USENIX Symposium on File and Storage Technologies (FAST'08)*. 127–141.

[47] Avantika Mathur, Mingming Cao, and Andreas Dilger. 2007. Ext4: The next generation of the Ext3 file system. *Usenix Assoc.* 32, 3 (June 2007), 25–30.

[48] Jeffrey C. Mogul. 1994. A better update policy. In *Proceedings of the USENIX Summer Technical Conference (USENIX Summer'94)*. 99–111.

[49] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. 2018. Finding crash-consistency bugs with bounded black-box crash testing. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*. 33–50.

[50] Thanumalayan Sankaranarayana Pillai, Ramnatthan Alagappan, Lanyue Lu, Vijay Chidambaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2017. Application crash consistency and performance with CCFS. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST'17)*. 181–196.

[51] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2014. All file systems are not created equal: On the complexity of crafting crash-consistent applications. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI'14)*. 433–448.

[52] Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2005. Model-based failure analysis of journaling file systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'05)*. 802–811.

[53] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2005. IRON file systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP'05)*. 206–220.

[54] Anthony Rebello, Yuvraj Patel, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2020. Can applications recover from fsync failures? In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'20)*. 753–767.

[55] Ohad Rodeh, Josef Bacik, and Chris Mason. 2013. BTRFS: The Linux B-tree filesystem. *ACM Trans.* 9, 3 (Aug. 2013), 1–32.

[56] Bianca Schroeder, Sotirios Damouras, and Phillipa Gill. 2010. Understanding latent sector errors and how to protect against them. In *Proceedings of the 8th USENIX Symposium on File and Storage Technologies (FAST'10)*. 71–84.

[57] Margo Seltzer, Peter Chen, and John Ousterhout. 1990. Disk scheduling revisited. In *Proceedings of the Winter 1990 USENIX Conference*. 313–323.

[58] Chuck Silvers. 2000. UBC: An efficient unified I/O and memory caching subsystem for NetBSD. In *Proceedings of the USENIX Annual Technical Conference: FREENIX Track*. 285–290.

[59] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. 1996. Scalability in the XFS file system. In *Proceedings of the USENIX Annual Technical Conference*.

[60] Tomas Vondra. 2019. PostgreSQL vs. fsync. How is it possible that PostgreSQL used fsync incorrectly for 20 years, and what we'll do about it.Brussels, Belgium. Retrieved from https://archive.fosdem.org/2019/schedule/event/postgresql_fsync/.

[61] Youjip Won, Jaemin Jung, Gyeongyeol Choi, Joontaek Oh, Seongbae Son, Jooyoung Hwang, and Sangyeun Cho. 2018. Barrier-enabled IO stack for flash storage. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST'18)*. 211–226.

[62] Yupu Zhang, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2010. End-to-end data integrity for file systems: A ZFS case study. In *Proceedings of the 8th USENIX Symposium on File and Storage Technologies (FAST'10)*. San Jose, CA, 29–42.

[63] Yiying Zhang and Steven Swanson. 2015. A study of application performance with non-volatile main memory. In *Proceedings of the 31st IEEE Conference on Massive Data Storage (MSST'15)*. Santa Clara, CA, 1–10.

Received November 2020; accepted February 2021