Verifying a high-performance crash-safe file system using a tree specification

Haogang Chen,[†] Tej Chajed, Alex Konradi,[‡] Stephanie Wang,[§] Atalay İleri,
Adam Chlipala, M. Frans Kaashoek, Nickolai Zeldovich
MIT CSAIL

ABSTRACT

DFSCQ is the first file system that (1) provides a precise specification for `fsync` and `fdatasync`, which allow applications to achieve high performance and crash safety, and (2) provides a machine-checked proof that its implementation meets this specification. DFSCQ's specification captures the behavior of sophisticated optimizations, including log-bypass writes, and DFSCQ's proof rules out some of the common bugs in file-system implementations despite the complex optimizations.

The key challenge in building DFSCQ is to write a specification for the file system and its internal implementation without exposing internal file-system details. DFSCQ introduces a *metadata-prefix* specification that captures the properties of `fsync` and `fdatasync`, which roughly follows the behavior of Linux ext4. This specification uses a notion of *tree sequences*—logical sequences of file-system tree states—for succinct description of the possible states after a crash and to describe how data writes can be reordered with respect to metadata updates. This helps application developers prove the crash safety of their own applications, avoiding application-level bugs such as forgetting to invoke `fsync` on both the file and the containing directory.

An evaluation shows that DFSCQ achieves 103 MB/s on large file writes to an SSD and durably creates small files at a rate of 1,618 files per second. This is slower than Linux ext4 (which achieves 295 MB/s for large file writes and 4,977 files/s for small file creation) but much faster than two recent verified file systems, Yggdrasil and FSCQ. Evaluation results from application-level benchmarks, including TPC-C on SQLite, mirror these microbenchmarks.

SOSP'17, October 28–31, 2017, Shanghai, China.
© 2017 Copyright is held by the owner/author(s).
ACM ISBN 978-1-4503-5085-3/17/10.
https://doi.org/10.1145/3132747.3132776

1 INTRODUCTION

File systems achieve high I/O performance and crash safety by implementing sophisticated optimizations to increase disk throughput. These optimizations include deferring writing buffered data to persistent storage, grouping many transactions into a single I/O operation, checksumming journal entries, and bypassing the write-ahead log when writing to file data blocks. The widely used Linux ext4 is an example of an I/O-efficient file system; the above optimizations allow it to batch many writes into a single I/O operation and to reduce the number of disk-write barriers that flush data to disk [33, 56]. Unfortunately, these optimizations complicate a file system's implementation. For example, it took 6 years for ext4 developers to realize that two optimizations (data writes that bypass the journal and journal checksumming) taken together can lead to disclosure of previously deleted data after a crash [30]. This bug was fixed in November of 2014 by forbidding users from mounting an ext4 file system with both journal bypassing for data writes and journal checksumming. A comprehensive study of several file systems in Linux also found a range of other bugs [34: §6].

Somewhat surprisingly, there exists no precise specification that would allow proving the correctness of a high-performance file system, ruling out bugs like the ones described above. For example, the POSIX standard is notoriously vague on what crash-safety guarantees file-system operations provide [44]. Of particular concern are the guarantees provided by `fsync` and `fdatasync`, which give applications precise control over what data the file system flushes to persistent storage. Unfortunately, file systems provide imprecise promises on exactly what data is flushed, and, in fact, for the Linux ext4 file system, it depends on the options that an administrator specifies when mounting the file system [40]. Because of this lack of precision, applications such as databases and mail servers, which try hard to make sequences of file creates, writes, and renames crash-safe by issuing `fsync`s and `fdatasync`s, may still lose data if the file system crashes at an inopportune time [40]. For example, SQLite and LightningDB, two popular databases, improperly used `fsync` and `fdatasync`, leading to possible data loss [65].

[†] Now at Databricks. [‡] Now at Google. [§] Now at UC Berkeley.

A particularly challenging behavior to specify is *log-bypass writes*[1]. A file system typically uses a write-ahead log to guarantee that changes are flushed to disk atomically and in order. The one exception is data writes: to avoid writing data blocks to disk twice (once to the log and once to the file's blocks), file systems bypass the log when writing file data. Since data blocks are not written to the log, they may be re-ordered with respect to logged metadata changes, and this reordering must be precisely captured in the specification. A further complication arises from the fact that this reordering can lead to corruption in the presence of block reuse. For example, a bypass write to file f can modify block b on disk, but the metadata update allocating b to f has not been flushed to disk yet; b may still be in use by another file or directory, which might be corrupted by the bypass write. The specification must exclude this possibility.

This paper makes several contributions:

(1) An approach to specifying file-system behavior purely in terms of abstract *file-system trees*, as opposed to exposing the internal details of file-system optimizations, such as log-bypass writes directly updating disk blocks. A purely tree-based specification shields application developers from having to understand and reason about low-level file-system implementation details.

(2) A precise *metadata-prefix* specification that describes the file system's behavior in the presence of crashes (in addition to describing non-crash behavior), including the `fsync` and `fdatasync` system calls and sophisticated optimizations such as log-bypass writes, using the tree-based specification approach. Intuitively, the specification says that if an application makes a sequence of metadata changes, the file system will persist these changes to disk in order, meaning that the state after a crash will correspond to a *prefix* of the application's changes. On the other hand, data updates need not be ordered; thus, the name *metadata-prefix*. Our specification is inspired by the observed behavior of the Linux ext4 file system,[2] as well as other prior work [11, 35].

(3) A formalization of this metadata-prefix specification. A precise formal specification is important for two reasons. First, it allows developers of a mission-critical application such as a database to prove that they are using `fsync` correctly and that the database doesn't run the risk of losing data. Second, it allows file-system developers to prove that the optimizations that they implement will still provide the same guarantees, embodied in the metadata-prefix specification, that applications may rely on.

The main challenge in formalizing the metadata-prefix specification and proving that a file system obeys this specification lies in capturing the two degrees of nondeterminism due to crashes: which prefix of metadata updates has been committed to disk and which log-bypass writes have been flushed. This paper uses the notion of *tree sequences* to capture the metadata-prefix specification. Abstractly, the state of the file system is a sequence of trees (directories and files), each corresponding to a metadata update performed by the application, in the order the updates were issued; the sequence captures the metadata order. Data writes apply to every tree in the sequence, reflecting the fact that they can be reordered with respect to metadata. After a crash, the file system can end up in any tree in the sequence. Calling `fsync` logically truncates the sequence of trees to just the last one, and `fdatasync` makes file data writes durable in every tree in the sequence.

(4) To demonstrate that the metadata-prefix specification is easy to use, we implemented, specified, and proved the correctness of a core pattern used by applications to implement crash-safe updates. This code pattern modifies a destination file atomically, even if the file system crashes during the update. That is, if a crash happens, either the destination file exists with a copy of *all* the new bytes that should have been written to it, or the destination file is not modified. This pattern captures the core goal of many crash-safe applications—ensuring that some sequence of file creates, writes, renames, `fsync`s, and `fdatasync`s is crash-safe. If an application's use of `fsync` and `fdatasync` does not ensure crash safety, the application developer is unable to prove the application correct.

(5) To demonstrate that the metadata-prefix specification allows proving the correctness of a high-performance file system, we built DFSCQ (Deferred-write FSCQ), based on FSCQ [10] and its Crash-Hoare Logic infrastructure.[3] DFSCQ implements standard optimizations such as deferring writes, group commit, non-journaled writes for file data, checksummed log commit, and so on. We report the results of several benchmarks on top of DFSCQ, compared to Linux ext4 and two recent verified file systems.

DFSCQ has several limitations. First, DFSCQ (and the metadata-prefix specification) does not capture concurrency; system calls are executed one at a time. Second, DFSCQ runs as a Haskell program, which incurs a large trusted computing base (the Haskell compiler and runtime) as well as CPU performance overhead. Addressing these limitations remains an interesting open problem for future work.

2 RELATED WORK

DFSCQ is the first file system with a machine-checked proof of crash safety in the presence of log-bypass writes, and the metadata-prefix specification is the first to precisely capture the behavior of both `fsync` and `fdatasync`. The rest of this section relates prior work to DFSCQ and the metadata-prefix specification.

File-system verification. There has been significant progress on machine-checked proofs of file-system correctness; the most recent results include Yggdrasil [48],

[1]File systems use different terms for this optimization. For example, ext4 often uses the term direct writes, but that is easily confused with direct I/O, which refers to bypassing the buffer cache.

[2]ext4's implementation flushes metadata changes in order, similar to what our specification requires. However, ext4 has no specification.

[3]The source code is at https://github.com/mit-pdos/fscq.

FSCQ [10], and Cogent's BilbyFS [1, 2]. The main contribution of this paper in relation to this prior work is the formalization and proof of sophisticated optimizations, such as log-bypass writes, in the presence of crashes, as well as proving the correctness and crash safety of application code on top of a file system. Earlier work on file-system verification [3, 5, 6, 17–19, 21, 24, 28, 29, 29, 58] also cannot prove the crash safety of a file system in the presence of such sophisticated optimizations.

Efforts to find bugs in file-system code have been successful in reducing the number of bugs in real-world file systems and other storage systems [27, 32, 62–64]. However, these approaches cannot guarantee the absence of bugs.

Application bugs. It is widely acknowledged that it is easy for application developers to make mistakes in ensuring crash safety for application state [40]. For instance, a change in the ext4 file-system *implementation* changed the observable crash behavior of the file system. This change led to many applications losing data after a crash [14, 22], because of a missing `fsync` call whose absence did not previously keep the contents of a new file from being flushed to disk [8]. The ext4 developers, however, maintained that the file system never promised to uphold the earlier behavior, so this was an application bug. Similar issues crop up with different file-system options, which often lead to different crash behavior [40]. This paper is the first to demonstrate that it is possible to prove a challenging, commonly used application pattern correct and compose it, in a provable way, with a verified file system, to produce an end-to-end proof of crash safety and application correctness.

Formalization techniques. Contributions of this paper are the metadata-prefix specification and the notion of tree sequences for writing the specifications of a high-performance crash-safe file system. Prior work has focused on specifying the file-system API [43], including crashes [8], using trace-based specifications [23] or abstract state machines [1]. However, no prior specifications have addressed bypassing the log for data writes and `fdatasync`.

3 MOTIVATION

To handle crashes, many modern file systems run each system call in a transaction using write-ahead logging. For example, if the file system is creating a new file on disk, it needs to modify two disk blocks: one containing the directory and one containing the file's inode. If the computer were to crash in between these two block writes, the on-disk state after a crash may be inconsistent: in particular, the directory may have been updated, but the file's inode may not have been updated yet. Logging ensures that, after a crash, the file system can either apply or roll back the changes, ensuring that either all of the updates are applied, or none are.

3.1 Why `fsync` and `fdatasync`?

Extending the file-system interface with `fsync` and `fdatasync` enables two optimizations on top of logging. First, a file system can buffer transactions in volatile memory and defer committing them to persistent storage until an application explicitly requests it using `fsync` or `fdatasync`. This allows the file system to send changes in batches to the storage device.

The second optimization targets applications that modify large files, such as virtual-machine disk images. Making these changes through the log requires writing the data to disk twice: first when the change is written to the log and second when the change is applied to the file's data blocks. To avoid double writes, file systems bypass the log and update on-disk file data in-place (e.g., when the application calls `fsync` or `fdatasync`, or the file system decides to flush its own buffer cache).

Table 1 demonstrates the importance of the log-bypass optimization by showing the throughput of two microbenchmarks on a fast SSD with and without log bypass. The first benchmark, **smallfile**, durably creates small files by calling `fsync` after creating a file and writing 1 KB to it; the second benchmark, **largefile**, overwrites a 50 MB file using 4KB writes with an `fdatasync` every 10 MB. "ext4 ordered," which implements log-bypass writes, achieves 1.9× the throughput of "ext4 journaled" on the largefile benchmark, which writes data blocks to the journal, and also achieves 1.2× the throughput for the smallfile benchmark.

	smallfile	**largefile**
ext4 journaled	3720 files/s	152 MB/s
ext4 ordered	4560 files/s	294 MB/s

Table 1: Linux ext4 performance on an Intel S3700 SSD in two configurations: "journaled," where file data writes are journaled, and "ordered," which implements log-bypass writes for file data.

Although these optimizations pay off even on a fast modern SSD, it is an open question whether these optimizations will remain important with newer persistent-memory technologies, such as NVM, which don't lose their contents after a power failure [61]. For instance, fast persistent memory may allow local file systems to be synchronous, avoiding buffering altogether.

3.2 Complex interface

The two optimizations described above improve file-system performance but make it more difficult for application developers to write crash-safe code.

Deferred durability. Figure 1 shows how a prototypical application pattern of `fsync` and `fdatasync`, in combination with other file-system APIs, updates a file in a crash-safe manner. This pattern shows up in many applications, such as mail servers, text editors, databases, etc. [40, 65]. Even layered file systems, such as overlayfs [9], use this pattern and sometimes get it wrong [50]. The example code assumes that the application never runs this procedure concurrently.

```
tmpfile = "crashsafe.tmp"

def crash_safe_update(filename, data_blocks):
  f = open(tmpfile, O_CREAT | O_TRUNC | O_RDWR)
  for block in data_blocks:
    f.write(block)
  f.close()

  fdatasync(tmpfile)
  rename(tmpfile, filename)
  fsync(dirname(filename))

def crash_safe_recover():
  unlink(tmpfile)
```

Figure 1: Pseudocode for an application library that updates the contents of a file in a crash-safe manner.

`crash_safe_update(f, data)` ensures that, after a crash, file `f` will have either its old contents or the new `data`, but not a mixture of them. To ensure this property, `crash_safe_update` writes the new data into a temporary file. After `crash_safe_update` has finished writing data to the temporary file, it invokes `fdatasync` to persist the file's data blocks on disk. After `fdatasync` returns, `crash_safe_update` replaces the original file with the new temporary file using `rename`. Finally, `crash_safe_update` uses `fsync` to flush its change to the directory, so that upon return, an application can be sure that the new data will survive a crash.

All parts of this example are necessary to update the file in a crash-safe manner. The temporary file and `rename` ensure that the file is not partially updated after a crash. The `fdatasync` ensures that the file system does not flush the `rename` call before it flushes the contents of the file (thus resulting in an empty file). The final `fsync` ensures that the `rename` is flushed to disk before `crash_safe_update` returns. Note that there is no need to call `fsync` after creating the temporary file.

If the system crashes while executing `crash_safe_update`, it first executes the file system's recovery code (which may replay transactions that have been committed but not applied), followed by recovery code of applications. In our example, the application-specific recovery code `crash_safe_recover` simply deletes the temporary file if one exists. This is sufficient for our example, since if the temporary file exists, we must have crashed in the middle of `crash_safe_update`, and thus the original file still has its old contents.

Log-bypass writes. The second optimization, bypassing the log for file data writes, brings another set of complications for the application programmer, because data writes can be reordered with respect to metadata updates. For example, if `crash_safe_update` did not invoke `fdatasync`, some file systems may buffer the data write in memory and flush the `rename` to disk before writing the file's contents. This can result in an empty or zero-filled file after a crash, which does not correspond to any prefix of system calls.

Background flushes. A final complication due to buffering changes in memory is that the file system may run out of memory if an application makes many changes. File systems deal with this by flushing changes in the background at any time; once a change is written durably to disk, it can be discarded from memory. File systems typically do not make guarantees about the order in which they flush changes to disk in the background.

3.3 Complex implementation

Exploiting deferred durability and log-bypass writes leads to significant complexity inside of the file system, especially when the two optimizations interact.

Block reuse. With log-bypass writes, it is important that blocks are not reused until all metadata changes are flushed to disk. Consider a situation where an application deletes an old file, creates a new file, and writes some data to the new file. If the file system were to reuse the old file's blocks for the new file, writes to the new file would bypass the log and modify the same blocks that used to belong to the old file. If the file system crashes at this point, before flushing the in-memory transactions that deleted the old file and created the new one, the disk would contain the old file whose data blocks contain the new file's data. To avoid this problem, file systems typically delay the reuse of freed disk blocks until in-memory transactions are flushed to disk.

Exposing uninitialized data. Log-bypass writes present a complication where the old contents of a disk block may be exposed in a newly created file after a crash. Consider an application that grows a file using `ftruncate`. This causes the file system to do two things: first, to find an unused disk block and zero it out, and second, to mark the block as allocated and add it to the file's block list. With the log-bypass write optimization, an implementation may treat the zeroing of the disk block as a data write and bypass the log, but perform the update of the free bitmap and the file's block list in a transaction. If the transaction is flushed to disk without the preceding zero write, and the computer crashes, the disk contains a new file pointing to the old disk block whose data has not been erased yet. This can expose one user's data to another user.

It may seem relatively straightforward to avoid a problem like this [20]. However, two further optimizations make the problem less obvious and led to deleted file contents being exposed after a crash in Linux ext4; this bug existed for six years before it was discovered [30]. First, a modern disk has an internal write buffer that allows the disk's controller to reorder writes, until the file system issues a *write barrier*. Second, ext4 can use checksums to ensure the integrity of the on-disk log after a crash, allowing it to commit transactions using just one write barrier instead of two. Under the combination of these optimizations, ext4 performs all disk writes in a batch: the zeroes to the old file blocks and the new transaction adding the block to the file's block list.

Due to the checksum optimization, ext4 did not issue a write barrier between writing the zeroes and the transaction, which can lead to the problem described above. The disk may choose to persist the transaction first, before zeroing out the data block. A crash between these two steps results in a new file pointing to old disks blocks whose data have not been erased. On recovery, the file system will see that the log is valid (because the checksum matches) and will replay the creation of the file. However, the zero-ing hasn't been performed, and thus the block pointers in the new file may point to blocks with content from previous files or directories. Developers of ext4 considered the two optimizations incompatible and proposed to disallow enabling both modes at the same time [30].

3.4 Goal: specification for formal verification

The above examples illustrate the tension between performance and correctness in file systems and their applications. This paper explores formal verification as a way of reconciling high-performance file systems and the bugs that arise due to their complexity. Formal verification is a promising approach because it can help both file-system and application developers to reason about all possible corner cases in their sophisticated optimized implementations, and prove that their code is correct regardless of when the computer may crash. However, this requires a specification describing the behavior of the file system that allows for the above optimizations.

4 METADATA-PREFIX SPECIFICATION

The POSIX specification describes the expected behavior of most file-system operations [26]. Unfortunately, it does not specify much about the behavior of `fsync` and `fdatasync`, or about what happens after a crash for any other system call. This omission has been brought to the attention of the POSIX committee [44], but they have been unable to reach consensus so far on what POSIX should promise about crash safety.

In lieu of formal guidance from POSIX, file systems have implemented a wide range of guarantees; these guarantees differ even within the same file system depending on which variant of the implementation is used (as chosen through mount options) [40]. This makes it difficult for an application developer to know what crash-safety guarantees they can or cannot rely on if they want to achieve high performance.

To understand the issues involved, consider an application that calls `rename("d1/f", "d2/f")`, followed by `fsync("d1")`. For performance reasons, a POSIX-compliant implementation might flush just the content of the specific directory (i.e., `d1`), allowing the file systems to parallelize `fsync` on different files and directories. However, in our example, this would mean that the file `f` could be lost after a crash: it would be gone from `d1`, because `d1` was synced to disk, but it would not yet appear in `d2`, because `d2` had not been synced. Thus, if the `fsync` specification required that just the directory itself was flushed, the file system may be able to get good performance, but it would be difficult for application developers to use such an API in the presence of crashes.

To make it easier for application developers to build crash-safe applications, the file system could provide a different specification, mirroring the traditional BSD semantics, where all metadata operations are synchronous (written to disk immediately), but file contents are asynchronous. This means that an application need not worry about calling `fsync` on a directory; the `rename` operation from the `crash_safe_update` example in the previous section would be persisted to disk upon return. While this is simple to reason about, it achieves low performance for metadata-heavy workloads (such as extracting a tar file or deleting many files).

As can be seen from these different specifications, defining the specification for directory `fsync` requires a balance between achieving good performance and enabling application developers to reason about application-level crash safety.

Approach. We approach this challenge by proposing a practical specification that is both easy use by application developers and allows for efficient file-system implementations. Specifically, the metadata-prefix specification says:

1. `fdatasync(f)` on file `f` flushes just the data of `f`. This allows for *log bypass* when writing `f`'s data blocks, which is important to avoid writing file data to disk twice. For example, this allows a database server to write to the disk (through a large preallocated file) without incurring file-system logging overheads.

2. `fsync(f)` is a superset of `fdatasync(f)`: it flushes both data and metadata. Furthermore, `fsync` flushes *all* pending metadata changes; i.e., if `fsync(d)` is called on directory `d`, it effectively ignores the argument and flushes changes to all other unrelated directories.

3. Finally, the file system is always allowed to flush any file's data or all of the metadata operations performed up to some point in time. This allows the file system to reorder data and metadata writes to disk. After a crash, metadata updates will be consistently ordered (i.e., if the file system performed two operations, *a* and *b*, in that order, then after a crash, if *b* appears on disk, then so must *a*), but data writes can appear out-of-order.

This metadata-prefix specification provides a clear contract between applications and a file system. Ensuring metadata ordering helps developers reason about the possible states of the directory structure after a crash: if some operation survives a crash, then all preceding operations must have also survived. At the same time, the specification allows for high-performance file-system implementations: on the metadata side, it allows batching of metadata changes (until the next `fsync` call), and on the data side, it allows for log bypass for file data writes and allows for each file's *data* to be flushed independently of other files and of the metadata. This enables high performance for applications that modify data in-place.

Discussion. One benefit of the metadata-prefix specification is that, by flushing all metadata changes in order, it simplifies the application code for durably creating files. For example, in the `crash_safe_update` procedure from Figure 1, the developer knows that all directory changes have been flushed to disk once `fsync(dirname(filename))` returns. Notably, this includes any possible pending changes to parent directories as well, for instance if the application had just created the parent directory prior to calling `crash_safe_update`.

One limitation of the metadata-prefix specification is that it can flush unrelated changes to disk when an application invokes `fsync`, since the specification requires all metadata changes to be flushed together. In practice, the metadata-prefix specification captures behavior similar to that provided by the ext4 implementation (in its default configuration), largely as a side effect of ext4 having a system-wide log for all metadata. This suggests that the specification is amenable to high-performance implementations and high-performance applications, and our prototype implements many of the optimizations found in ext4.

Recent work has explored the performance benefits of avoiding flushing unrelated metadata changes [7, 38, 41]. An interesting direction for future work would be to come up with a corresponding specification where it is still easy enough to prove application-level correctness.

5 FORMALIZING THE METADATA-PREFIX SPECIFICATION

To verify a file-system implementation, we must formalize the metadata-prefix specification described above, which means precisely specifying the states in which a system can crash at every point in its execution. As in Crash Hoare Logic [10], we explicitly specify these crash states by describing all possible contents of the disk at the time of the crash. System calls that modify metadata run in a transaction, which ensures that all metadata changes are applied to disk atomically. Deferred writes lead to many more possible crash states: when a computer crashes, the state of the disk could reflect the changes of any system calls since the last `fsync`. The challenging aspect of formalizing these crash states is to describe them in a way that is easy for applications to reason about, specifically in specifying the behavior of bypass writes and `fdatasync`.

5.1 Strawman: operational specification

To understand why bypass writes and `fdatasync` complicate a specification, consider an example application that renames a file from `f1` to `f2` and writes to `f2`. When the computer crashes, the file system can recover in a state where the application's writes are applied to `f1`. This situation arises because the file system's bypass writes directly update the on-disk blocks corresponding to `f2`, which happen to also correspond to `f1` if the rename has not been flushed yet.

A natural way to specify bypass writes is to describe what the file system does in an operational manner: e.g., after a bypass write to `f2`, `f2`'s data block `bn` is modified, and after an `fdatasync`, the file's disk blocks are flushed. This matches the informal specification from POSIX [52] and from the Linux manpage for `fdatasync` [51].

This description makes it easy for an application to reason about the state of the system in the absence of crashes: `f2` now has the new data. However, to reason about crashes, the application must consider all possible metadata states on disk after a crash (e.g., whether the rename succeeded) and deduce what effect modifying `bn` would have in that situation. This may be difficult for some metadata operations, such as creating or deleting a file, or freeing and reusing data blocks.

For instance, if an application deletes `f1`, then creates `f2` before writing to it, and then the computer crashes, `f1` may still appear to be on disk (because the deletion metadata was not flushed yet). Applications likely expect that the bypass write to `f2` should not corrupt `f1`'s contents in this case, even though this was acceptable when the application renamed `f1` to `f2`. As yet another example, applications also expect that the bypass write to `f2` not corrupt other file-system state (e.g., the contents of some unrelated directory). Reasoning about these possible crash effects requires the application to precisely understand which disk blocks correspond to which parts of the file system (e.g., data blocks of one file or another file, or metadata blocks corresponding to a directory, a free bitmap, the log, etc.).

The above example illustrates why the operational way of describing bypass writes and `fdatasync` is not condusive to proving application crash safety: applications must in effect prove the correctness of file-system internals, reasoning about disk-block allocation and reuse.

5.2 Our approach: tree-based specification

Our approach is to specify the effects of bypass writes and `fdatasync` purely at the level of file-system trees, without mentioning disk blocks. This allows the application developer to reason purely about file-system trees that can appear after a crash, and requires the file-system developer to prove that bypass writes meet the tree-level specification (e.g., by proving that file data blocks are never reused in a way that can corrupt other files or metadata).

Our specification of the file-system state reflects the two degrees of nondeterminism related to crashes: first, which metadata updates have been committed to disk after a crash, and second, which bypass writes have been reflected in the on-disk state. The next two subsections describe how we model these two aspects.

5.3 Representing deferred commit

In the presence of the deferred commit optimization (where a file system buffers committed transactions in volatile memory), it is difficult to describe the tree that might be on disk after a crash. For instance, a synchronous specification for the `unlink` system call might say that after `unlink` returns, the file is removed from the tree, and if a crash occurs during `unlink`, the file might or might not have been removed. With deferred commit, if a crash occurs during `unlink`, the on-disk state might have little to do with `unlink` itself and instead might reflect the operations that were performed on the tree before `unlink` was called. For instance, the directory in which `unlink` is called might not have been created yet.

To describe these crash states succinctly, we avoid reasoning about the contents of a single tree and instead represent the possible on-disk state as a sequence of trees. Each tree represents the state of the file system after some system call, and each system call adds a new tree to the sequence.

Figure 2 shows an example tree sequence. On the left, the first tree in the sequence represents the persistent state stored on disk. The list of transactions on the bottom corresponds to the system calls that have committed in memory but whose changes have not been flushed to disk yet. Instead of describing individual transactions, the specification talks about a sequence of trees, where each tree represents the state that would arise if one were to apply the in-memory transactions, in order, up to that point. Specifically, the middle row of Figure 2 shows the disk contents that would arise after applying a prefix of in-memory transactions, and the top row shows the corresponding abstract file-system trees.

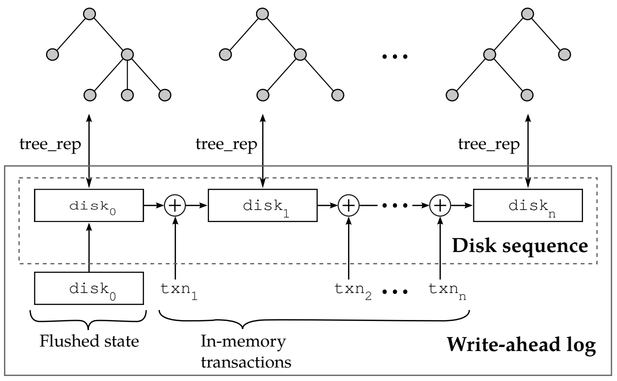


Figure 2: An illustration of the tree-sequence abstraction.

Tree sequences simplify specifications. For instance, consider the specification of the `unlink` system call, shown in Figure 3, written using Crash Hoare Logic [10].[4] The specification has a precondition, a postcondition, and a crash condition. The specification says that, if the precondition is true when `unlink` is invoked, then if `unlink` returns, the postcondition will be true, and if the computer crashes before `unlink` returns, the crash condition will be true.

In our `unlink` example, the precondition describes the state of the system before `unlink` is called, by saying that

[4]In our implementation of Crash Hoare Logic, these specifications are written in Coq's programming language [13]; in this paper we show an easier-to-read version, but the full source code is available at `https://github.com/mit-pdos/fscq`.

SPEC	unlink(*cwd_ino*, *pathname*)
PRE	**disk**: tree_rep(*tree_seq*)
POST	**disk**: tree_rep(*tree_seq* ++ [*new_tree*]) ∧ new_tree = tree_prune(*tree_seq*.latest, *cwd_ino*, *pathname*)
CRASH	**disk**: tree_intact(*tree_seq* ++ [*new_tree*])

Figure 3: Specification for `unlink`.

there is some sequence of trees, called *tree_seq*, representing system calls that have been executed since the last `fsync`. `tree_rep` is a representation invariant connecting the physical state of the disk and the in-memory state to their logical representation as a tree sequence. The postcondition adds a new tree to this sequence, where the unlinked file is removed. The crash condition simply says that `unlink` can crash with any of the trees from the original tree sequence (corresponding to earlier system calls) or with the new tree that `unlink` added to this sequence.

Tree sequences naturally capture the metadata-prefix property, because the tree sequence is built up by applying the application's system calls in order. As a result, we can succinctly describe the metadata-prefix property by saying that a crash during a system call can result in any of the trees from the tree sequence.

Tree sequences also allow for a concise specification of `fsync` for directories, as shown in Figure 4. The specification simply says that, after `fsync` on a directory returns, the tree sequence contains only the latest tree from before `fsync`. This latest tree reflects all of the system calls issued by the application up to its call to `fsync`.

SPEC	fsync(*dir_ino*)
PRE	**disk**: tree_rep(*tree_seq*)
POST	**disk**: tree_rep([*tree_seq*.latest])
CRASH	**disk**: tree_intact(*tree_seq*)

Figure 4: Specification for `fsync`; not shown is the part of the precondition that checks for `dir_ino` pointing to a directory inode.

5.4 Representing log-bypass writes

Writes to file data that bypass the log can cause the state of the file system after a crash to violate the order in which system calls were issued, since log-bypass writes are not ordered with respect to other updates that use the write-ahead log. For instance, if an application writes to an existing file and then renames the file, after a crash the file may have the new name but the old contents. Conversely, if the application first renames the file and then writes to it, after a crash the file may have the old name but the new contents.

As explained above, an operational specification of bypass writes is succinct (e.g., "block `bn` of file `f2` was updated") but is difficult for applications to reason about. Our specification captures the effect of bypass writes in terms of how they modify the trees in a tree sequence.

Figure 5 illustrates how our specification integrates log-bypass writes with tree sequences. A log-bypass write (shown by blue in the figure) can affect every tree in the tree sequence, because under the covers, the tree sequence is simply a logical sequence of trees that would arise if one were to apply the in-memory transactions to the real on-disk state. Thus, modifying the real disk state can change all of the trees in the tree sequence. The figure illustrates two subtleties with log-bypass writes that our specification captures. If the file being modified was present elsewhere in the file-system hierarchy in a previous tree, it will also be affected by a log-bypass write. However, if the file is nowhere to be found, then the log-bypass write must have no effect on the corresponding tree (that is, it is not allowed to change the contents of another file or corrupt some metadata block).

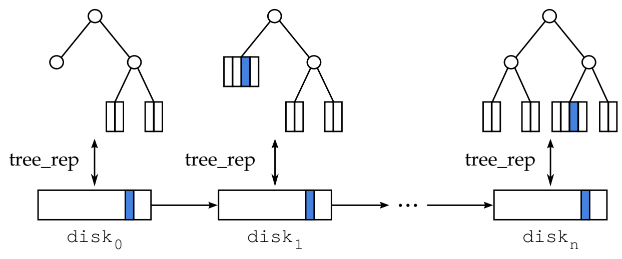


Figure 5: Interaction between log bypass and tree sequences.

This tree-level description of bypass writes is easier for application developers to reason about because the application developer needs to consider which tree in the tree sequence might appear after a crash (based on which metadata updates were committed) and then consider the files with outstanding bypass writes in that tree. Applications need not consider the possibility of bypass writes modifying data blocks belonging to other files, because the specification of bypass writes precisely describes which file(s) in the tree can be affected.

SPEC pwrite(*ino*, *off*, *buf*)
PRE **disk**: tree_rep(*old_tree_seq*) $\wedge$ $\exists$*path*,
find_subtree(*old_tree_seq*.latest, *path*) = $\langle ino, f \rangle$ $\wedge$
off $\in$ *f*
POST **disk**: tree_rep(*new_tree_seq*) $\wedge$ $\forall i$,
if $\exists pn, \exists f_i$, such that *old_tree_seq*[*i*][*pn*] = $\langle ino, f_i \rangle$,
then *new_tree_seq*[*i*] = tree_update(*old_tree_seq*[*i*], *pn*,
f_i.overwrite(*off*, *buf*))
else *new_tree_seq*[*i*] = *old_tree_seq*[*i*]
CRASH **disk**: tree_intact(*new_tree_seq*)

Figure 6: Specification for the `pwrite` system call that bypasses the log. This simplified specification assumes that `pwrite` does not extend the file.

Consider Figure 6, which shows a simplified specification for the `pwrite` system call, assuming that `pwrite` does not grow the file. The postcondition states that, if a file with the same inode number exists in any earlier tree, it will be modified, at the same offset, and otherwise the `pwrite` will have no effect on that earlier tree. The abstract file's `overwrite()` method discards writes past the end of the file; one implication is that the specification says that writes to a file that was recently extended may be applied partially (or not at all) in earlier trees in the tree sequence. This gives application developers a clear guarantee about what can and cannot happen to other files after a crash. Note that the specification allows the file to have a different path name in a previous tree, corresponding to our example where `f1` was renamed to `f2` before `f2` was modified.

SPEC fdatasync(*ino*)
PRE **disk**: tree_rep(*old_tree_seq*) $\wedge$ $\exists$*path*,
find_subtree(*old_tree_seq*.latest, *path*) = $\langle ino, f \rangle$
POST **disk**: tree_rep(*new_tree_seq*) $\wedge$ $\forall i$,
if $\exists pn, \exists f_i$, such that *old_tree_seq*[*i*][*pn*] = $\langle ino, f_i \rangle$
then *new_tree_seq*[*i*] = tree_update(*old_tree_seq*[*i*], *pn*,
f_i.sync_data())
else *new_tree_seq*[*i*] = *old_tree_seq*[*i*]
CRASH **disk**: tree_intact(*old_tree_seq*)

Figure 7: Specification for `fdatasync`.

The specification for `fdatasync`, shown in Figure 7, is similar. The main difference in the postcondition is that, instead of updating the file's contents using the abstract file's `overwrite()` method, it flushes the file's data, represented using the abstract file's `sync_data()` method.

5.5 Representing crash nondeterminism

After a system crashes and recovers, the metadata-prefix property requires that the resulting file-system state reflects a prefix of metadata changes, along with any subset of file data writes, since data writes may bypass the log. Tree sequences concisely represent the prefix of metadata writes that survive a crash, by nondeterministically choosing one of the trees from the tree sequence.

To represent the effects of reordered data writes on that tree, we model the contents of a file's block as a history of values that have been written to that block (similarly to how FSCQ models asynchronous disk writes [10]). Writing to a block adds a new value to the history. Reading from a block returns the latest value from the history. Calling `fdatasync` deletes all history except for the last write. A crash nondeterministically selects one of the block values from the history.

SPEC recover()
PRE **disk**: tree_intact(*old_tree_seq*)
POST **disk**: tree_rep([crash_xform(*t*)]) $\wedge$ $t \in$ *old_tree_seq*
CRASH **disk**: tree_intact(*old_tree_seq*)

Figure 8: Specification for `recover`.

Figure 8 formally describes how tree sequences capture crash recovery. The postcondition of `recover` says that the file-system state is a singleton tree sequence, whose tree is the crash transform of some tree t that appeared in the pre-crash tree sequence. The `crash_xform` function implements the nondeterministic choice of data blocks from each file's history, as described above. Since the crash condition implies the precondition, `recover` is idempotent with respect to crashes and can thus handle crashes during recovery.

DFSCQ makes specifications more concise by taking advantage of the fact that some operations shrink the set of possible crashes while others grow the set. For example, Figure 6 specifies that `pwrite`'s crash condition corresponds to the new tree, which includes all of the possible ways that the old tree could have crashed. In contrast, `fdatasync`'s crash condition in Figure 7 corresponds to the old tree, since that tree subsumes all possible crashes after `fdatasync`.

5.6 Log checksums

Many file systems use checksums to ensure the integrity of a log after a crash. Although this may seem like an implementation issue, there is a risk that it shows up in the specification because of the possibility of checksum collisions. Specifically, there is a small (but nonzero) probability that, after a crash, the recovery procedure reads the on-disk log and decides that the log contents are valid, because the checksum matched due to an inadvertent checksum collision, even though the file system did not write such a valid log entry to disk. One approach to handling this nonzero probability is to have the file-system specifications explicitly state that there is a small probability that the specification doesn't hold (because of a hash collision). The downside of this approach is that we must prove that the implementation indeed guarantees that the probability is small. This requires probabilistic reasoning in any layer above the logging subsystem, which would complicates proofs and is also not well-supported by the Coq ecosystem.

Another approach is to state an axiom that collisions will not happen, matching how programmers typically ignore the possibility of hash collisions. Using this axiom, however, we can prove a contradiction using a pigeon-hole argument in Coq (and any other formal proof system). This would in turn make all of DFSCQ's specifications unsound, because any conclusion follows from a contradiction.

We avoid reasoning about probabilities and unsoundness of specifications by taking advantage of the fact that Hoare-logic specifications deal with terminating procedures, so we can model the negligible hash collisions as procedure nontermination, a trick adopted from previous work [4]. We introduce a new opcode in Crash Hoare Logic, called *Hash*, which computes the hash value of its input. The formal semantics models the *Hash* opcode by keeping track of all inputs ever hashed, and by storing their corresponding hash values (purely for proofs; not at execution time). If *Hash* is presented with an input that hashes to the same result as an earlier, different input, then the semantics says that the *Hash* opcode enters an infinite loop and never returns. The details of this plan are described in Wang's thesis [57].

This formalization achieves our goals. First, it allows developers to conclude that, if *Hash* returns the same hash value twice, the inputs must have been equal (because otherwise, according to the formal semantics, *Hash* would not have returned), without reasoning about probabilities. This allows writing specifications about entire file-system operations, such as `rename`, saying that *if the operation returns*, then a transaction must have been committed. Second, this formalization is sound, because it does not prohibit the possibility of hash collisions, instead explicitly taking them into account (by formally treating the program as entering an infinite loop on a collision).

At runtime, of course, the *Hash* opcode is implemented using a collision-resistant hash function. This hash function does *not* enter an infinite loop when presented with a colliding input, and consequently it can differ from the formal semantics of *Hash*. However, since we know that our hash function is collision-resistant, we know that the possibility of finding a collision is negligible, and thus the possibility that the real execution semantics will differ from the formal ones is also negligible. Consequently, using a collision-resistant function for *Hash* at runtime allows us to capture the standard assumption made by developers (that hash collisions do not happen).

6 IMPLEMENTATION

To demonstrate that the metadata-prefix specification allows for a verified high-performance file-system implementation, we built a prototype file system called DFSCQ. DFSCQ is based on FSCQ but implements a wide range of I/O and CPU performance optimizations and provides a number of features missing from FSCQ, such as implementing doubly indirect blocks to support large files and proving that `mkfs` is correct. DFSCQ is not as complete as ext4; for example, it does not support extended attributes or permissions, and it does not implement the cylinder-group optimization. Furthermore, ext4 can run system calls concurrently, while DFSCQ has no support for concurrency.

Following FSCQ's approach, DFSCQ's implementation is extracted from Coq into Haskell code and compiled into a user-space FUSE server. Table 2 shows the lines of code for different components of DFSCQ. The proofs are fully checked by Coq for correctness and guarantee that DFSCQ's implementation meets DFSCQ's metadata-prefix specification, which covers all system calls implemented by DFSCQ (a few of them were shown earlier in this paper). The total development effort involved 5 people over two years, but none of these people worked full-time on DFSCQ; the total effort behind DFSCQ is much less than 10 person-years.

The correctness argument of DFSCQ assumes that trusted components are correct; in the case of our prototype, the TCB includes the specifications, the formal execution semantics, Coq's proof checker, the Haskell extraction process, and the runtime environment (which includes the Haskell runtime, the FUSE driver, and the Linux disk device driver).

Component	Lines of code
FSCQ infrastructure	24,032
Hashing semantics	458
General data structures	7,216
Buffer cache	2,453
Write-ahead log	11,791
Inodes and files	9,521
Directories	10,310
Top-level file-system API	2,320
Byte files	6,453
Tree sequences	5,630
crash_safe_update	3,634
Haskell extraction support	109
Total	83,927

Table 2: Lines of spec, code, and proof for DFSCQ

In the rest of this section, we provide more details about the most interesting aspects of DFSCQ: block reuse, write-ahead logging, CPU performance optimizations, and proofs.

6.1 Block reuse

Implementing log-bypass writes correctly is challenging. Consider a disk block b that belongs to directory d, and suppose an application calls `rmdir`(d) and then the file system reuses b for a new file f. If the `rmdir` is not yet flushed to disk, log-bypass writes to b will overwrite the on-disk contents of d. After a crash, d will be corrupted with f's contents. The typical approach for avoiding this problem is to ensure that data blocks are not reused until after the metadata log is applied to disk. DFSCQ implements a variant of this approach, by maintaining two block allocators, one of which is used for allocating blocks and the other for freeing blocks. Flushing the metadata log forms a barrier, which ensures that there are no possible on-disk pointers to free disk blocks. DFSCQ swaps the two allocators on every log flush, since it is now safe to reuse blocks that were freed before the log flush.

To prove the safety of this plan, DFSCQ maintains an internal invariant, called *bypass safety*, which is a pairwise relation between every adjacent pair of trees in the tree sequence. This relation states that, for two trees (one old and one new), for every block that belongs to some file in the new tree, that same block must either belong to the same file in the old tree (with the same inode number, at the same offset), or that block must be unallocated in the old tree. Since this invariant is transitive, it allows us to prove that updating a block in the latest tree will never affect other files or directories. We prove that this invariant is maintained throughout DFSCQ. If the implementation did not correctly handle block reuse, it would violate the specification, because log-bypass writes to a reused block may affect an unrelated file or unrelated metadata, which would alter the tree in a way prohibited by the specification (for instance, as shown in Figure 6 and Figure 7), and we would be unable to prove that the implementation meets the specification.

Another block-reuse complication arises when an application resizes a file and then invokes `fdatasync`. For example, if a file was truncated and then regrown, the file may have the same length, but the underlying disk blocks are different (because they were not reused). At this point, if the application invokes `fdatasync`, the file system does not know which blocks the file used to have in the past and, as a result, cannot flush the file's old blocks to disk. This makes it difficult to satisfy the metadata-prefix specification shown in Figure 7, which requires that the file's data be properly flushed in *all* past trees.

To prove that applications use `fdatasync` correctly, DFSCQ introduces another relation called *block stability*. Block stability says that a file's list of blocks grows monotonically within a tree sequence. Growing a file maintains that file's block stability. Truncating a file gives up block stability of that file, because now `fdatasync` will not flush the blocks that the file used to have. Flushing the log reestablishes block stability of every file, because there is now exactly one tree in the tree sequence. Block stability is a per-file property, and the postconditions of our specifications for `pwrite` and `fdatasync` are conditional on the file's block stability (not shown in the simplified specs earlier in the paper).

6.2 Write-ahead logging and log-bypass writes

DFSCQ implements write-ahead logging much like other file systems, including sophisticated optimizations like deferred commit, deferred apply, group commit, checksums, etc. The unique challenge faced by DFSCQ lies in proving the correctness of this write-ahead log. DFSCQ's implementation breaks up the write-ahead log into smaller logical modules (as shown in Figure 9), whose correctness can be proven independently and combined to provide a single logical abstract state: a logged disk.

One complication we did not initially expect is reconciling the log with log-bypass writes. Bypass writes still interact with the log in two ways. First, bypass writes need to be reflected in the abstract state exposed by the log. Second, if the file system issues a bypass write to block b, and there is an unapplied transaction that modified block b, it is important that this transaction does not later overwrite b's contents.

We resolve this complication by making the log aware of bypass writes; the log exposes two functions: `write`, which logs the update, and `dwrite`, which bypasses the log. Both update the abstract logged disk state, but `write` makes a synchronous update whereas `dwrite` adds a pending write. `dwrite`'s implementation also checks if there was a previous logged write to the same address as the bypass write.

Internally, DFSCQ implements two ways of updating file data, just like ext4's two mount options: one where file data writes bypass the log and one where file data writes are logged. Both modes of operation are proven to obey the same specification. Unlike ext4, both modes are compatible with the log-checksum optimization, which, as a result, is always enabled. Furthermore, DFSCQ does not require choosing one of the two modes at the time the file system is mounted.

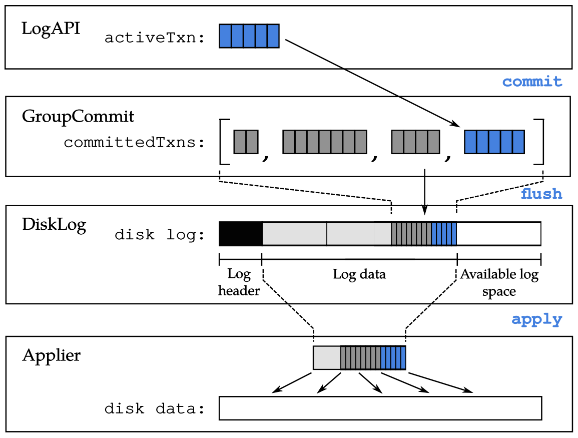


Figure 9: Illustration of log layers and the timeline of a transaction. Transactional writes are added to an `activeTxn` map in the LogAPI layer. When the application calls `commit`, `activeTxn` is buffered in a list of pending transactions in GroupCommit. When `flush` is called on GroupCommit, all pending transactions are flushed to disk together as a single transaction in the DiskLog layer. At this point, the transactions are durable, and Applier can lazily apply and truncate the log records.

Instead, DFSCQ uses a simple heuristic to decide which mode to use: when appending to a file, writes are logged (since DFSCQ must log the inode's block-pointer update anyway), and when overwriting a file, writes bypass the log, since no metadata updates are needed. One downside of this simple heuristic is that DFSCQ will log writes in some cases (e.g., large file appends) when it would have been better to do bypass writes.

6.3 CPU-performance optimizations

Since DFSCQ generates executable code through Haskell, any CPU inefficiency is multiplied by the overhead of running functional Haskell code. DFSCQ includes a range of common optimizations, which are proven correct. For example, DFSCQ implements specialized caches for directory entries, inodes, dirty blocks, and free bitmap entries (used for both free disk blocks and free inodes), instead of recomputing them based on the buffer cache. DFSCQ has a two-level implementation of a bitmap allocator: instead of treating a disk block as an array of 32768 bitmap bits (as in FSCQ), DFSCQ treats a disk block as an array of 64-bit words, each of which has 64 bitmap bits. DFSCQ implements several workarounds for Haskell-induced overhead, such as ordering operations to avoid excessive garbage collection and lazy evaluation.

6.4 Proving

Proving the correctness of DFSCQ is challenging because DFSCQ maintains complex invariants, some of which were mentioned above. In addition, the metadata-prefix specification also allows for complex behavior. Defining safety relations, such as bypass safety and block stability, and proving their transitivity helped us prove the correctness of several optimizations.

To help applications reason about the contents of trees, we introduced a variant of separation logic [42] for pathnames in trees. The "addresses" in this separation logic are full pathnames, and the "values" are one of three types: either a file (along with the contents of that file), a directory (with no additional information about the directory, since the content of that directory is reflected in the values associated with other pathnames), or "missing," which indicates that the pathname does not exist. (The simplified specifications shown in this paper do not use this separation logic, however.)

In the absence of separation logic, we found that proving application-level code required reasoning about functional tree transformations, such as the result of performing a lookup after `rename` or `unlink`. This in turn required reasoning about whether any pair of names are the same or different, whether one of them might be a directory, etc. By tracking directories and missing pathnames in our separation logic, we are able to precisely capture that `unlink` deletes a file and that creating a file requires that it not already exist and that the parent directory does exist (and is a directory).

7 EVALUATION

This section uses DFSCQ to answer the following questions about the metadata-prefix specification:

- Can developers use the metadata-prefix specification to prove the crash safety of their applications, and are tree sequences helpful? (§7.1)
- Does the metadata-prefix specification help file-system developers avoid bugs? (§7.2)
- Is the metadata-prefix specification, and DFSCQ's implementation of it, correct? (§7.3)
- Does the metadata-prefix specification allow for a high-performance implementation? (§7.4)

7.1 Application crash safety

To provide evidence for whether a tree-sequence-based specification is useful in proving application-level crash safety, we implemented and proved the `crash_safe_update` procedure shown in Figure 1, which captures the core pattern used by applications to perform crash-safe updates.

Proving the correctness of `crash_safe_update` led us to discover several cases where the DFSCQ specification was too weak. For example, in the `read` specification we originally forgot to mention that the data returned by the system call is related to the contents of the file. None of these issues required changing the DFSCQ implementation, and we were able to reprove the correctness of DFSCQ after fixing the specification.

Proving `crash_safe_update` also led us to discover a number of corner cases in `crash_safe_update` itself. For example, we discovered that `crash_safe_update` cannot perform

Bug category and example	**Possible in FSCQ?**	**Prevented by FSCQ?**	**Possible with the M-P spec?**	**Prevented by the M-P spec?**
Logging logic; write/barrier ordering [16, 30, 53]	Some (no checksumming)	Yes	Yes	Yes
Misuse of logging API [46, 54]	Some (no log bypass)	Yes	Yes	Yes
Bugs in recovery protocol [25, 36]	Yes	Yes	Yes	Yes
Improper corner-case handling [60]	Yes	Yes	Yes	Yes
Low-level bugs [30, 37, 59]	Some (memory safe)	Yes	Some (memory safe)	Yes
Concurrency [31, 55]	No	—	No	—

Table 3: Representative bugs found in Linux ext4 and whether the metadata-prefix specification precludes them.

a safe update on a file with the same file name as the temporary file that it uses. Another example is that, after recovery, the destination file could be legitimately *neither* equal to its original contents *nor* the source file: if the source file was modified but not synced prior to calling `crash_safe_update`, the destination file could contain the latest write to the source file, but the source file itself could lose that write after a crash. After fixing the specification to take into account these corner cases, we were able to prove the correctness of `crash_safe_update` when running on top of DFSCQ.

Tree sequences help prove application safety. As a concrete example, DFSCQ has an intermediate-level operational specification for the file system, following the strawman described in §5.1, which describes the effect of log-bypass writes in terms of direct writes to the underlying disk. We tried to prove `crash_safe_update` on top of this operational specification and gave up due to complexity after a significant amount of effort. This directly led us to develop a tree-based specification, which allowed us to prove `crash_safe_update` with a modest amount of effort.

7.2 File-system bugs

This section sheds light in two ways on whether DFSCQ's specification can help prevent real bugs. First, we present a case study of different kinds of bugs that have been discovered in the Linux ext4 file system, and we argue for whether FSCQ (which has a synchronous specification that disallows DFSCQ's optimizations) or DFSCQ avoid them. Second, we describe our own experience in developing DFSCQ, and we point out specific bugs that were caught in the process of proving its correctness.

ext4 bugs case study. We looked through Linux ext4's Git logs starting from 2013 and categorized the bugs fixed in those commits. Table 3 shows the resulting categories, with representative bugs from each category. For instance, this table includes the bug that was mentioned in the introduction, where ext4 disclosed previously deleted file data after a crash [30]. The table also shows whether each bug category could have occurred in the implementations of either FSCQ or DFSCQ; for instance, some bugs arise due to concurrent execution of system calls, which is impossible in both FSCQ and DFSCQ by design (i.e., they are not sophisticated enough to have such a bug). The table also shows whether the theorems of FSCQ and DFSCQ prevent those bugs.

We make four conclusions from this case study. First, DFSCQ is sophisticated enough that its implementation could have had many of the bugs that were fixed in ext4, making verification important. Second, FSCQ was not sophisticated enough to even have many of these bugs, especially the trickier cases dealing with log bypass, log checksums, and so on. Third, the metadata-prefix specification precludes every bug category that was possible in the DFSCQ implementation. This suggests that the metadata-prefix specification is effective at preventing real bugs. Finally, the one category where the metadata-prefix specification is not sophisticated enough to have bugs is concurrency: the specification, as well as DFSCQ, are single-threaded. Verifying a concurrent file system is an open problem and remains future work.

Development experience. While proving the correctness of DFSCQ, we ran into several cases where we were unable to prove a theorem, in the process discovering an underlying implementation issue. For instance, when `mknod` was invoked on an existing pathname, it would delete the old file. This was allowed by the specification (which in itself could have arguably been a bug), but more importantly it failed to deallocate the old file's blocks. This violated bypass safety, and we were unable to prove that log bypass would be safe after `mknod`. Another example is log bypass writes: while trying to prove that it is safe to bypass the log for modifying a file data block, we realized that there could be a pending non-bypass write to that same block in the write-ahead log. This forced us to change the system's design for handling log-bypass writes, as described in §6. These examples show that proofs are good at bringing out corner cases that are easy to overlook during development and testing.

7.3 Are DFSCQ specs correct?

Proving `crash_safe_update` demonstrates that the specification provides strong guarantees that can be used by an application. To further demonstrate that DFSCQ's specification are correct, we performed the following experiments, which suggest that DFSCQ's specification and implementation match what developers expect.

fsstress. We ran `fsstress` from the Linux Test Project to check if it finds bugs in DFSCQ. When we first ran `fsstress`, it caused our FUSE file server to crash. However, after some investigation, we discovered that this was due to a bug in our Haskell FUSE bindings that sit between DFSCQ and the

Linux FUSE interface. The bug was due to the developer thinking that some corner case could not be triggered and calling the `error` function in Haskell to panic if that case ever executed. As it turns out, `fsstress` found a way to trigger that corner case. After fixing this bug, `fsstress` ran without problems and did not discover any bugs in DFSCQ's proven code. This bug reflects the fact that DFSCQ has a large TCB.

Enumerating crash states. We ran `crash_safe_update` on DFSCQ while monitoring all of the disk writes and barriers issued by DFSCQ. We then computed all possible subsets and reorderings of DFSCQ's disk writes, subject to its barriers, to produce every possible state in which DFSCQ could have crashed. Finally, we remounted the resulting disk with DFSCQ and examined the file-system state after DFSCQ performed its recovery. This experiment produced 182 possible disks after a crash but only three distinct file-system states after DFSCQ executed its recovery code: neither file existed, the temporary file existed with no contents, or the destination file existed with the written contents. All of these states are safe, since either the destination file didn't exist or it contained the correct data (the empty temporary file is removed during recovery).

7.4 Performance

To evaluate DFSCQ's performance, we use two microbenchmarks and three application workloads. We compare DFSCQ to FSCQ [10], to Yxv6 [48], and to ext4.[5] We run Yxv6 in two modes: the verified synchronous mode where all system calls immediately persist their changes, and the asynchronous mode where system calls are deferred in memory, called `group_commit` by Yggdrasil, which we denote by Yxv6*. This second mode, however, does not have a top-level file-system specification that describes how commits are deferred [47, 49]; as a result, it provides no meaningful proof of crash safety. For ext4, we ran in two modes: one with `data=ordered` (which we denote ext4) and the other with `data=journal,journal_async_commit` mount options (which we denote ext4/J). As a result of the bug we described earlier [30] (§3.3), ext4 prohibits the use of `journal_async_commit` in `data=ordered` mode.

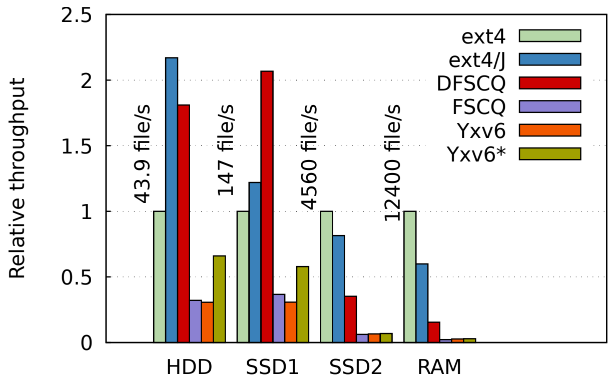
All experiments were run on a Dell PowerEdge R430 server with two Intel Xeon E5-2660v3 CPUs and 64 GB of RAM. We used several disk configurations to compare performance. One is a 7200 rpm WD RE4 2 TB rotational disk, which we denote as HDD. One is an inexpensive Samsung 850 SSD, which we denote as SSD1. The third is an expensive high-performance Intel S3700 SSD, denoted SSD2. Finally, a RAM disk configuration is denoted as RAM as a way of simulating the maximum possible disk performance. We use two SSDs to demonstrate how DFSCQ performs on both high-end and lower-end SSDs.

To confirm that the experimental results are meaningful, we ran all experiments an additional five times. For all experiments, the standard deviation of the measured results was within 10% of the mean; the median standard deviation across all experiments was 1.6%. We expect some variance across runs due to non-deterministic behavior in the Linux I/O stack and the storage devices.

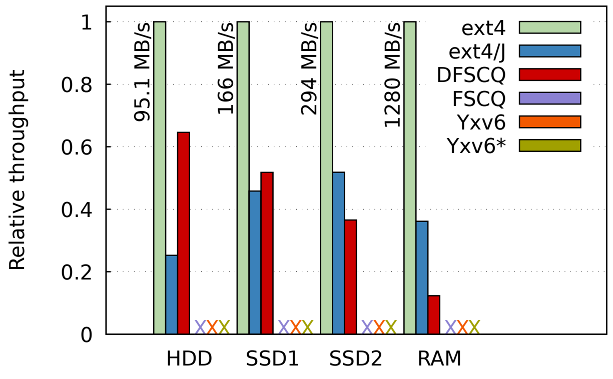
[5]We do not compare to BilbyFS [2] since it is designed to run only on raw flash devices, and it runs only with an older version of the Linux kernel.

Microbenchmarks. The microbenchmarks are intended to measure performance of deferred commit for small file operations and large file writes, inspired by LFS [45]. The `smallfile` benchmark creates 1,000 files; for each, it creates the file, writes 100 bytes to it, and `fsync`s it; we measure throughput in terms of files per second. The `largefile` benchmark overwrites a 50 MB file, calling `fsync` every 10 MBytes; we measure throughput in terms of MB/s.

The results are shown in Figure 10. We draw several conclusions. First, DFSCQ achieves good performance, significantly improving on FSCQ and Yxv6, due to its I/O and CPU optimizations. DFSCQ is also more complete: neither FSCQ nor Yxv6 can run the `largefile` benchmark, because they lack doubly indirect blocks.



(a) The `smallfile` microbenchmark.



(b) The `largefile` microbenchmark.

Figure 10: Performance of Linux ext4, DFSCQ, FSCQ, Yggdrasil for two microbenchmarks. Vertical numbers indicate the absolute throughput of Linux ext4. The ext4 numbers are the same as in Table 1. Benchmarks that did not complete due to file-size limitations are marked with "X."

Second, DFSCQ performance is close to that of ext4 for `smallfile` on HDD and even beats ext4 on SSD1. This is because DFSCQ is just as efficient as ext4 in terms of disk barriers, but ext4 writes out one more block to its journal (to initially zero out the new file), which DFSCQ combines with the subsequent data write. DFSCQ also achieves performance close to that of ext4 for `largefile` on HDD and SSD1. However, on SSD2 and RAM, DFSCQ's performance lags behind that of ext4 due to the CPU overhead of Haskell.

Applications. Figure 11 shows the performance for three applications. mailbench is a qmail-like mail server [12], which we modified to call `fsync` and `fdatasync` to ensure messages are stored durably in the spool and in the user mailbox, using the pattern from Figure 1. The TPC-C benchmark executes a TPC-C-like workload [39] on a SQLite database. "Dev. mix" is measuring the result of running `git clone` on the xv6 source-code repository [15] followed by running `make` on it.

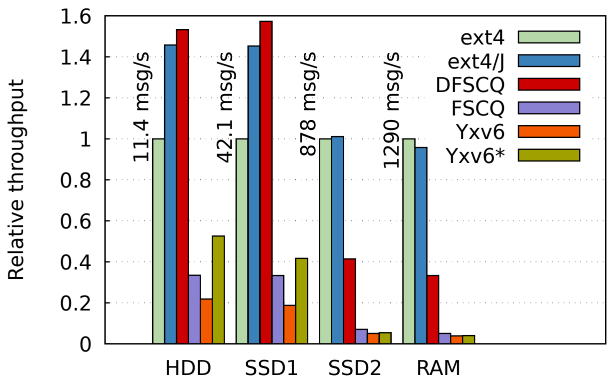
The results mirror the conclusions from the microbenchmarks. DFSCQ significantly outperforms other verified file systems and is able to run applications that others cannot. DFSCQ's performance on HDD and SSD1 is comparable to ext4's, but DFSCQ's Haskell overhead becomes much more significant with SSD2 and RAM in particular. On the TPC-C benchmark, DFSCQ outperforms ext4 on HDD because DFSCQ writes less data to disk compared to ext4, due to the fact that DFSCQ combines in-memory transactions and eliminates duplicate writes before writing to the on-disk journal, and ext4 does not.

8 CONCLUSION

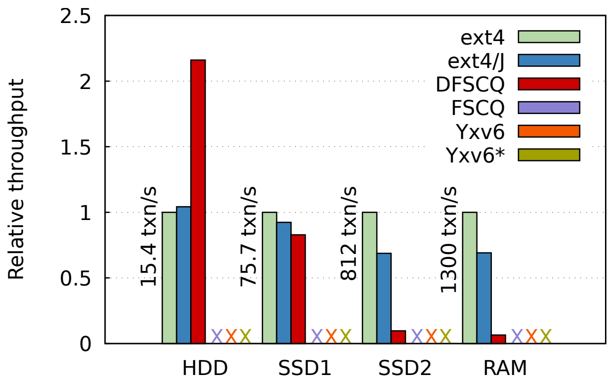
DFSCQ is the first verified file system that implements sophisticated optimizations and achieves both correctness and good performance. The metadata-prefix specification, implemented by DFSCQ, is the first to describe file-system behavior in the presence of crashes, deferred commit, `fsync`, and `fdatasync`. Using tree sequences, DFSCQ represents the metadata-prefix specification in a way that is amenable to proving correctness of both the file-system implementation and application-level code. We hope that our specification techniques will help others to reason about their storage systems.

ACKNOWLEDGMENTS

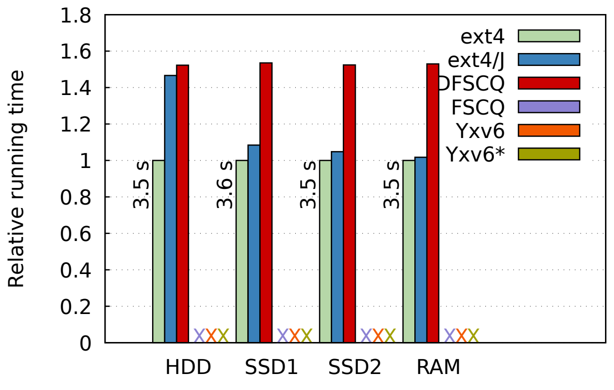
We thank Robert Morris, the anonymous reviewers, and our shepherd, Jon Howell, for their feedback. This research was supported by NSF award CNS-1563763 and by Google.



(a) mailbench.



(b) TPC-C on SQLite.



(c) Dev. mix.

Figure 11: Performance of Linux ext4, DFSCQ, FSCQ, Yggdrasil for three application workloads. Vertical numbers indicate the absolute throughput of Linux ext4. mailbench and TPC-C on SQLite measure throughput; higher is better. Dev. mix measures running time; lower is better. Benchmarks that did not complete due to file-size limitations are marked with "X."

REFERENCES

[1] S. Amani and T. Murray. Specifying a realistic file system. In *Proceedings of the Workshop on Models for Formal Analysis of Real Systems*, pages 1–9, Suva, Fiji, Nov. 2015.

[2] S. Amani, A. Hixon, Z. Chen, C. Rizkallah, P. Chubb, L. O'Connor, J. Beeren, Y. Nagashima, J. Lim, T. Sewell, J. Tuong, G. Keller, T. Murray, G. Klein, and G. Heiser. COGENT: Verifying high-assurance file system implementations. In *Proceedings of the 21st International*

Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 175–188, Atlanta, GA, Apr. 2016.

[3] K. Arkoudas, K. Zee, V. Kuncak, and M. Rinard. Verifying a file system implementation. In *Proceedings of the 6th International Conference on Formal Engineering Methods*, Seattle, WA, Nov. 2004.

[4] G. Barthe, C. Fournet, B. Grégoire, P.-Y. Strub, N. Swamy, and S. Zanella-Béguelin. Probabilistic relational verification for cryptographic implementations. In *Proceedings of the 41st ACM Symposium on Principles of Programming Languages (POPL)*, San Diego, CA, Jan. 2014.

[5] W. R. Bevier and R. M. Cohen. An executable model of the Synergy file system. Technical Report 121, Computational Logic, Inc., Oct. 1996.

[6] W. R. Bevier, R. M. Cohen, and J. Turner. A specification for the Synergy file system. Technical Report 120, Computational Logic, Inc., Sept. 1995.

[7] S. S. Bhat, R. Eqbal, A. T. Clements, M. F. Kaashoek, and N. Zeldovich. Scaling a file system to many cores using an operation log. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, Shanghai, China, Oct. 2017.

[8] J. Bornholt, A. Kaufmann, J. Li, A. Krishnamurthy, E. Torlak, and X. Wang. Specifying and checking file system crash-consistency models. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 83–98, Atlanta, GA, Apr. 2016.

[9] N. Brown. Overlay filesystem. https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt.

[10] H. Chen, D. Ziegler, T. Chajed, A. Chlipala, M. F. Kaashoek, and N. Zeldovich. Using Crash Hoare Logic for certifying the FSCQ file system. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, pages 18–37, Monterey, CA, Oct. 2015.

[11] V. Chidambaram, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Optimistic crash consistency. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 228–243, Farmington, PA, Nov. 2013.

[12] A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–17, Farmington, PA, Nov. 2013.

[13] Coq development team. *The Coq Proof Assistant Reference Manual, Version 8.5pl2*. INRIA, July 2016. http://coq.inria.fr/distrib/current/refman/.

[14] J. Corbet. ext4 and data loss. http://lwn.net/Articles/322823/, Mar. 2009.

[15] R. Cox, M. F. Kaashoek, and R. T. Morris. Xv6, a simple Unix-like teaching operating system, 2016. http://pdos.csail.mit.edu/6.828/xv6.

[16] L. Czerner. [PATCH] ext4: Fix data corruption caused by unwritten and delayed extents. https://lwn.net/Articles/645722, Apr. 2015.

[17] G. Ernst, G. Schellhorn, D. Haneberg, J. Pfähler, and W. Reif. Verification of a virtual filesystem switch. In *Proceedings of the 5th Working Conference on Verified Software: Theories, Tools and Experiments*, Menlo Park, CA, May 2013.

[18] M. A. Ferreira and J. N. Oliveira. An integrated formal methods tool-chain and its application to verifying a file system model. In *Proceedings of the 12th Brazilian Symposium on Formal Methods*, Aug. 2009.

[19] L. Freitas, J. Woodcock, and A. Butterfield. POSIX and the verification grand challenge: A roadmap. In *Proceedings of 13th IEEE International Conference on Engineering of Complex Computer Systems*, pages 153–162, Mar.–Apr. 2008.

[20] G. R. Ganger and Y. N. Patt. Metadata update performance in file systems. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation (OSDI)*, pages 49–60, Monterey, CA, Nov. 1994.

[21] P. Gardner, G. Ntzik, and A. Wright. Local reasoning for the POSIX file system. In *Proceedings of the 23rd European Symposium on Programming*, pages 169–188, Grenoble, France, 2014.

[22] B. Gribincea et al. Ext4 data loss. https://bugs.launchpad.net/ubuntu/+source/linux/+bug/317781, Jan. 2009.

[23] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages Systems*, 12(3):463–492, 1990.

[24] W. H. Hesselink and M. Lali. Formalizing a hierarchical file system. In *Proceedings of the 14th BCS-FACS Refinement Workshop*, pages 67–85, Dec. 2009.

[25] B. Hutchings. [PATCH 3.2 027/115] jbd2: fix fs corruption possibility in jbd2_journal_destroy() on umount path. https://lkml.org/lkml/2016/4/26/1230, April 2016.

[26] IEEE (The Institute of Electrical and Electronics Engineers) and The Open Group. The Open Group base specifications issue 7, 2013 edition (POSIX.1-2008/Cor 1-2013), Apr. 2013.

[27] D. Jones. Trinity: A Linux system call fuzz tester, 2014. http://codemonkey.org.uk/projects/trinity/.

[28] R. Joshi and G. J. Holzmann. A mini challenge: Build a verifiable filesystem. *Formal Aspects of Computing*, 19 (2):269–272, June 2007.

[29] E. Kang and D. Jackson. Formal modeling and analysis of a Flash filesystem in Alloy. In *Proceedings of the 1st Int'l Conference of Abstract State Machines, B and Z*, pages 294–308, London, UK, Sept. 2008.

[30] J. Kara. [PATCH] ext4: Forbid journal_async_commit in data=ordered mode. http://permalink.gmane.org/gmane.comp.file-systems.ext4/46977, Nov. 2014.

[31] J. Kara. ext4: fix crashes in dioread_nolock mode. http://permalink.gmane.org/gmane.linux.kernel.commits.head/575311, Feb. 2016.

[32] E. Koskinen and J. Yang. Reducing crash recoverability to reachability. In *Proceedings of the 43rd ACM Symposium on Principles of Programming Languages (POPL)*, pages 97–108, St. Petersburg, FL, Jan. 2016.

[33] Linux Kernel Developers. Ext4 filesystem, 2017. https://www.kernel.org/doc/Documentation/filesystems/ext4.txt.

[34] L. Lu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and S. Lu. A study of Linux file system evolution. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)*, pages 31–44, San Jose, CA, Feb. 2013.

[35] J. Mickens, E. Nightingale, J. Elson, B. Fan, A. Kadav, V. Chidambaram, O. Khan, K. Nareddy, and D. Gehring. Blizzard: Fast, cloud-scale block storage for cloud-oblivious applications. In *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI)*, Seattle, WA, Apr. 2014.

[36] K. Mostafa. [PATCH 3.13 075/103] jbd2: fix descriptor block size handling errors with journal_csum. https://lkml.org/lkml/2014/9/30/747, Sept. 2014.

[37] K. Mostafa. ext4: fix null pointer dereference when journal restart fails. https://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/commit/?id=9d506594069355d1fb2de3f9104667312ff08ed3, June 2016.

[38] D. Park and D. Shin. iJournaling: Fine-grained journaling for improving the latency of fsync system call. In *Proceedings of the 2017 USENIX Annual Technical Conference*, pages 787–798, Santa Clara, CA, July 2017.

[39] A. Pavlo. Python implementation of TPC-C, 2017. https://github.com/apavlo/py-tpcc.

[40] T. S. Pillai, V. Chidambaram, R. Alagappan, S. Al-Kiswany, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. All file systems are not created equal: On the complexity of crafting crash-consistent applications. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 433–448, Broomfield, CO, Oct. 2014.

[41] T. S. Pillai, R. Alagappana, L. Lu, V. Chidambaram, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Application crash consistency and performance with CCFS. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST)*, pages 181–196, Santa Clara, CA, Feb.–Mar. 2017.

[42] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, Copenhagen, Denmark, July 2002.

[43] T. Ridge, D. Sheets, T. Tuerk, A. Giugliano, A. Madhavapeddy, and P. Sewell. SibylFS: formal specification and oracle-based testing for POSIX and real-world file systems. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, pages 38–53, Monterey, CA, Oct. 2015.

[44] X. Roche, G. Clare, K. Schwarz, P. Eggert, J. Schilling, A. Josey, and J. Pugsley. Necessary step(s) to synchronize filename operations on disk. Austin Group Defect Tracker, Mar. 2013. http://austingroupbugs.net/view.php?id=672.

[45] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–15, Pacific Grove, CA, Oct. 1991.

[46] E. Sandeen. [PATCH] ext4: fix unjournaled inode bitmap modification. http://permalink.gmane.org/gmane.comp.file-systems.ext4/35119, Oct. 2012.

[47] H. Sigurbjarnarson and X. Wang. Personal communication, Mar. 2017.

[48] H. Sigurbjarnarson, J. Bornholt, E. Torlak, and X. Wang. Push-button verification of file systems via crash refinement. In *Proceedings of the 12th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–16, Savannah, GA, Nov. 2016.

[49] H. Sigurbjarnarson, J. Bornholt, E. Torlak, and X. Wang. The Yggdrasil toolkit, 2017. https://github.com/locore/yggdrasil/.

[50] M. Szeredi. ovl: fsync after copy-up. https://github.com/torvalds/linux/commit/641089c1549d8d3df0b047b5de7e9a111362cdce, Oct. 2016.

[51] The Linux man-pages project. fdatasync(2): Linux man page, 2017. http://man7.org/linux/man-pages/man2/fdatasync.2.html.

[52] The Open Group. fdatasync: synchronize the data of a file, 2016. http://pubs.opengroup.org/onlinepubs/9699919799/functions/fdatasync.html.

[53] T. Ts'o. [PATCH] ext4, jbd2: add REQ_FUA flag when recording an error flag. http://permalink.gmane.org/gmane.comp.file-systems.ext4/49323, July 2015.

[54] T. Ts'o. [PATCH] ext4: use private version of page_zero_new_buffers() for data=journal mode. https://lkml.org/lkml/2015/10/9/1, Oct. 2015.

[55] T. Ts'o. ext4: fix race between truncate and __ext4_journalled_writepage(). https://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/commit/?id=bdf96838aea6a265f2ae6cbcfb12a778c84a0b8e, June 2015.

[56] S. C. Tweedie. Journaling the Linux ext2fs filesystem. In *Proceedings of the 4th Annual LinuxExpo*, Durham, NC, May 1998.

[57] S. Wang. Certifying checksum-based logging in the RapidFSCQ crash-safe filesystem. Master's thesis, Massachusetts Institute of Technology, June 2016.

[58] M. Wenzel. Some aspects of Unix file-system security, Aug. 2014. http://isabelle.in.tum.de/library/HOL/HOL-Unix/Unix.html.

[59] D. J. Wong. jbd2: Fix endian mixing problems in the checksumming code. http://lists.openwall.net/linux-ext4/2013/07/17/1, July 2013.

[60] D. J. Wong. [PATCH] ext4: fix same-dir rename when inline data directory overflows. http://permalink.gmane.org/gmane.comp.file-systems.ext4/45594, Aug. 2014.

[61] J. Xu and S. Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST)*, pages 323–338, Santa Clara, CA, Feb. 2016.

[62] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using model checking to find serious file system errors. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 273–287, San Francisco, CA, Dec. 2004.

[63] J. Yang, C. Sar, P. Twohey, C. Cadar, and D. Engler. Automatically generating malicious disks using symbolic execution. In *Proceedings of the 27th IEEE Symposium on Security and Privacy*, pages 243–257, Oakland, CA, May 2006.

[64] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. eXplode: A lightweight, general system for finding serious storage system errors. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 131–146, Seattle, WA, Nov. 2006.

[65] M. Zheng, J. Tucek, D. Huang, F. Qin, M. Lillibridge, E. S. Yang, B. W. Zhao, and S. Singh. Torturing databases for fun and profit. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 449–464, Broomfield, CO, Oct. 2014.