Specifying and Checking File System Crash-Consistency Models

James Bornholt Antoine Kaufmann Jialin Li Arvind Krishnamurthy Emina Torlak Xi Wang
University of Washington
{bornholt,antoinek,lijl,arvind,emina,xi}@cs.washington.edu

Abstract

Applications depend on persistent storage to recover state after system crashes. But the POSIX file system interfaces do not define the possible outcomes of a crash. As a result, it is difficult for application writers to correctly understand the ordering of and dependencies between file system operations, which can lead to corrupt application state and, in the worst case, catastrophic data loss.

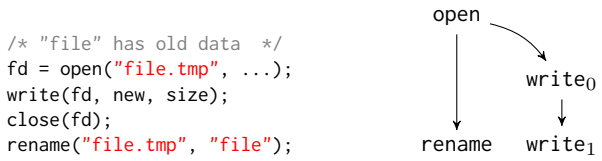
This paper presents *crash-consistency models*, analogous to memory consistency models, which describe the behavior of a file system across crashes. Crash-consistency models include both litmus tests, which demonstrate allowed and forbidden behaviors, and axiomatic and operational specifications. We present a formal framework for developing crash-consistency models, and a toolkit, called FERRITE, for validating those models against real file system implementations. We develop a crash-consistency model for `ext4`, and use FERRITE to demonstrate unintuitive crash behaviors of the `ext4` implementation. To demonstrate the utility of crash-consistency models to application writers, we use our models to prototype proof-of-concept verification and synthesis tools, as well as new library interfaces for crash-safe applications.

Categories and Subject Descriptors D.4.3 [*Operating Systems*]: File Systems Management; D.2.4 [*Software Engineering*]: Software/Program Verification

Keywords Crash consistency; file systems; verification

1. Introduction

Many applications interact with file systems and use them as persistent storage that will be preserved in the face of hardware or software crashes. To achieve the integrity properties they desire, these applications must use file system interfaces correctly. Failure to do so can lead to corrupted application state and catastrophic data loss [83]. Recent studies by Pillai et al. [62] and Zheng et al. [92] show that even mature applications contain serious persistence bugs, making incorrect assumptions about the behavior of the file system interface.



(a) (b)

`file`'s on-disk state	possible executions seen on disk
new	`open`, $\texttt{write}_0$, $\texttt{write}_1$, `rename`, ...
	`open`, $\texttt{write}_0$, `rename`, $\texttt{write}_1$, ...
	`open`, `rename`, $\texttt{write}_0$, $\texttt{write}_1$, ...
old	`open`, crash
	`open`, $\texttt{write}_0$, crash
	`open`, $\texttt{write}_0$, $\texttt{write}_1$, crash
empty	`open`, `rename`, crash
partial new	`open`, $\texttt{write}_0$, `rename`, crash
	`open`, `rename`, $\texttt{write}_0$, crash

(c)

Figure 1. A common replace-via-rename pattern (a) caused the "`ext4` data loss" incident of 2009 [84]. Only the ordering edges in (b) are enforced by the `ext4` file system, which may split the single `write` into smaller actions $\texttt{write}_n$. Unexpected on-disk states (c) arise if the system crashes during execution.

To illustrate the challenges in writing crash-safe applications, consider the 2009 "`ext4` data loss" incident, where multiple users reported that "pretty much any file written to by any application" became empty after a system crash [19, 84]. The root cause was the pattern in Figure 1(a), intended to atomically update "`file`" using a temporary file. Due to modern file system optimizations, the effects of the program do not necessarily reach disk in the order they are executed—only the orderings in Figure 1(b) are enforced. These ordering relaxations are usually invisible to applications and provide significant performance gains. But if the machine crashes during out-of-order execution, the relaxations can become visible. In this case, the `rename` can reach the disk before the

ASPLOS '16 April 2–6, 2016, Atlanta, Georgia, USA
Copyright © 2016 ACM 978-1-4503-4091-5/16/04. . . $15.00
DOI: http://dx.doi.org/10.1145/2872362.2872406

writes. If the machine crashes before the writes persist, users can see empty or partial files, and lose the old file completely, as Figure 1(c) shows. One possible fix is to add an fsync(fd) before close to ensure atomicity: applications will see either the old or the new data.

Who is to blame for such a bug? On one hand, application writers argue that file systems are "broken" [56] and are too aggressive about buffering and reordering file operations. On the other hand, file system developers argue that this behavior is allowed by POSIX and, as with relaxed memory orderings, is necessary for performance [41]. This trade-off between consistency and performance has long been a point of contention in file system design [26].

The key challenge for application writers is to understand the precise behavior of file systems across system crashes. The POSIX standard is largely silent on the guarantees file system interfaces should provide in the event of crashes (see §2.1). In practice, application writers make assumptions about the crash guarantees provided by file systems, and base their applications on these assumptions (e.g., file system-specific optimizations adopted by GNOME [45] and SQLite [77]). Being too optimistic about crash guarantees has led to serious data losses [10, 46, 62, 92]. But being too conservative leads to expensive and unnecessary synchronization, costing energy, performance, and hardware lifespan [55]. It is thus critical to describe crash guarantees of file systems in an unambiguous and accessible manner.

We propose to describe the behavior of file systems across crashes as *crash-consistency models*, analogous to the memory consistency models [1, 75] used to describe relaxed memory orderings. For example, the application in Figure 1 assumes *sequential crash-consistency*—after a crash, the system appears as if it committed the system calls in order. The ext4 file system, however, implements a weaker model, which allows rename to be reordered across a write.

Crash consistency models, as with memory consistency models, take two forms:

- *Litmus tests*: small programs that demonstrate allowed or forbidden behaviors of file systems across crashes; and
- *Formal specifications*: axiomatic and operational descriptions of crash-consistency behavior using logic and (nondeterministic) state machines, respectively.

These forms of specification serve different purposes, and neither can supplant the other. Formal specifications provide complete descriptions of (dis)allowed crash behaviors and, as such, provide a basis for automated reasoning about crash guarantees of applications. Litmus tests, on the other hand, provide a precise and intuitive description that is well suited for communicating to application developers, as well as for validating formal models against file system implementations.

We have developed FERRITE, a toolkit for executing litmus tests against formal specifications of crash-consistency models (to ensure that the specifications agree with the tests) and against real file systems (to validate our models). FERRITE exhaustively explores all possible crash behaviors of a given litmus test, both by explicit enumeration and SMT-based model checking. The enumerator (built on QEMU [7]) executes tests against actual file system implementations, while the symbolic model checker (built in Rosette [80]) executes tests against formal specifications. We have used FERRITE both to confirm known crash-safety problems and to uncover new problems in existing work. Unlike FiSC [88] and eXplode [89], which use model checking to discover file system bugs, FERRITE focuses on distilling the crash guarantees provided by a file system interface.

We show the utility of crash-consistency models with two proof-of-concept tools for application writers:

- A verifier that proves that a program provides intended crash guarantees (such as atomicity and durability) with respect to an operational crash-consistency model;
- A synthesizer that inserts a minimal set of fsync invocations into a program to make it satisfy intended guarantees with respect to an axiomatic crash-consistency model; and,

Crash-consistency models also suggest new OS extensions to better support writing crash-safe applications. Our experience with designing and using these tools shows that crash-consistency models offer a pragmatic formal basis for reasoning about crash safety, much as memory consistency models do for reasoning about the correctness of parallel programs.

In summary, the main contributions of this paper are: (1) litmus tests and formal specifications for precisely describing file system crash-consistency models, (2) a toolkit called FERRITE for running litmus tests against formal specifications and file system implementations, and (3) example applications of crash-consistency models.

The rest of this paper is organized as follows. §2 gives background on POSIX file systems. §3 introduces litmus tests and their results using FERRITE. §4 presents formal specifications. §5 describes how FERRITE works. §6 demonstrates applications of crash-consistency models. §7 relates to previous work. §8 concludes.

2. Background

This section provides background on POSIX file systems and on out-of-order writes to disk in real-world implementations.

2.1 The POSIX file system interface

The POSIX standard [30] defines a set of system calls for file system access [67]. Applications allocate a file descriptor for a file or directory using open, to further perform file operations (e.g., write and read) or directory operations (e.g., link, unlink, mkdir, and their *at variants [79]). They may explicitly flush data to disk using sync or fsync, and deallocate a file descriptor using close. A call to close frees in-memory data structures but need not flush data to disk [43].

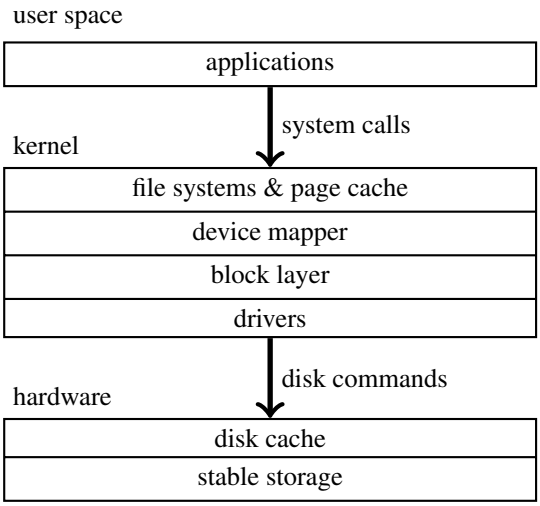


Figure 2. Linux I/O path example: from applications to disk.

As the effects of file and directory operations may be cached in memory, the `fsync` system call is key to providing data integrity in the face of crashes. POSIX defines the rationale for `fsync` [58]:

> The `fsync()` function is intended to force a physical write of data from the buffer cache, and to assure that after a system crash or other failure that all data up to the time of the `fsync()` call is recorded on the disk.

But POSIX is largely silent on crash guarantees of calls other than `fsync`. For example, the standard requires `rename` to be atomic, in the sense that when there is no crash, reading from the destination path can return only the old or new file content, not an intermediate state. But POSIX does not specify what should happen when a crash occurs [6]. As shown in Figure 1, file systems can exhibit unexpected crash behaviors for system calls such as `rename`.

2.2 Out-of-order writes

File systems implement a variety of mechanisms to provide data integrity in the face of crashes. Early file systems relied on check-and-repair tools such as `fsck` [51] to fix inconsistent state on disk after a crash. Modern file systems adopt techniques such as write-ahead logging (or journaling) [27, 28, 54], log-structured file systems [69], copy-on-write (or shadowing) [11, 17, 29, 44, 68], and soft updates [25, 50], among many others [15, 16, 24, 57, 59, 85]. But the modern operating system's I/O stack complicates reasoning about file system guarantees, as it often consists of multiple layers, each with its own caching and reordering policies [72].

As an example, Figure 2 shows a typical data path taken by applications when storing data to disk on Linux. First, applications invoke system calls such as `write` to transfer data from user space into the kernel. A file system may simply copy the data to the page cache and return to user space; pushing data further down the stack will happen later, when the kernel decides to flush the data or applications explicitly invoke system calls like `fsync`. Lower-level layers perform their own optimizations by caching and batching writes. For example, the device mapper may cache data using a faster, secondary disk (e.g., `bcache` and `dm-cache`), while the block layer may sort and merge writes, as well as device drivers and disk controllers [32].

Due to these optimizations, the I/O stack generally persists data to disk out of program order. Ordinarily, the intermediate states are invisible to applications, which retrieve data from the cached, up-to-date version, and so behave as if the system calls were executed sequentially (as required by POSIX). But if the system crashes, the intermediate states can be exposed. Our goal in this paper is to precisely describe the guarantees provided by file systems in the face of crashes, without the need to understand details of the I/O stack or file system implementation.

2.3 Assumptions

Unless otherwise specified, we assume that file systems are POSIX-compliant, running in their default configuration mode, on a single disk, and accessed by a single-threaded application. We focus on what applications should expect from file systems upon a clean reboot after a fail-stop crash. We do not consider disk corruption caused by buggy kernels, broken hardware, or malicious users.

3. Litmus tests

In memory consistency models, a litmus test is a small program and an outcome that communicates the reordering behavior of the memory system to application writers [4, 31]. We adopt litmus tests to document the crash behavior of file systems. A file system litmus test shows a guarantee the application writer might expect to hold—for example, that a later write being persisted implies an earlier write is also persisted. Developers can precisely document the behavior of their file system by listing whether certain litmus tests hold or not. Litmus tests are also useful for cross-validating formal specifications against real implementations.

This section illustrates file system crash-consistency behaviors using litmus tests. Our FERRITE toolkit can execute these litmus tests against a variety of file systems. We show several examples in which a litmus test clarifies ambiguous documentation or corrects misconceptions in prior research.

3.1 Specifying litmus tests

In FERRITE, a litmus test consists of three parts: initial setup, main body, and final checking. It can include POSIX file system calls and control flow constructs (e.g., dealing with errors if a call fails). For presentation purposes, we omit error handling in this section, assuming that all system calls succeed.

Below is a simple example of a litmus test:

```
initial:
  f ← creat("f", 0600)
main:
  write(f, "data")
  fsync(f)
  mark("done")
  close(f)
exists?:
  marked("done") ∧ content("f") ≠ "data"
```

The initial setup is optional. It starts with "`initial:`" and contains statements to create an initial file system state for the main body. It ends with an implicit `sync` (i.e., whole-system flush). If the initial setup is omitted, the main body of the test runs against an empty file system.

The main body starts with "`main:`" and runs after the initial setup. The program may crash at any point in the main body. A special pseudo-function `mark` labels externally visible events (e.g., printing messages on screen or responding to users); the label can be any unique string. We show next how to use `marks` to describe durability properties.

The final part is a list of predicates to be tested. A predicate in the list holds if there exists a (possibly crashing) execution of the program whose final state satisfies the predicate. The helper function `content`$(name)$ returns the file content for the given file name, or $\varnothing$ if the file does not exist; `marked`$(label)$ returns true iff the program crashes after the given label.

In the above example, the initial state consists of an empty file `f`. The main body of the test appends "data" to the file, calls `fsync`, and closes the file. In the final part, the predicate checks whether there exists an execution in which the file does not contain "data" after the "done" point.

In the rest of this section, we use litmus tests to show file system behavior that may be unintuitive to application writers. The predicates in the final section are *surprising* outcomes: a "sequential" file system that performs no reordering will disallow all these surprising behaviors (i.e., none of the tests' predicates will hold).

3.2 Litmus tests for file operations

We use FERRITE to develop existing litmus tests for two types of file writes: append and overwrite. Append writes are generally more complicated, involving multiple steps (e.g., updating inode, data block, and bitmap [5]) and delayed allocation [78]. Moreover, the on-disk state depends on how the file system orders updates to metadata (e.g., file size) and new data. The following litmus tests elucidate the intended behavior of appends and overwrites.

Prefix-append (PA). The prefix-append (PA) litmus test checks whether, in the event of a crash, a file always contains a prefix of the data that has been appended to it:

```
initial:
  N ← 2500
  as, bs ← "a" * N, "b" * N
  f ← creat("file", 0600)
  write(f, as)
main:
  write(f, bs)
exists?:
  content("file") ⋤ as + bs
```

This property, also referred as "safe append" [61, 77], ensures that no out-of-thin-air or garbage data can appear in a file. Many applications (e.g., Chrome [10]) assume prefix append. But popular file systems, such as `ext4` in default configuration, do *not* guarantee this property, as we show in §3.5.

Ordered-file-overwrites. Most file systems reorder file writes, which can lead to surprising results for application writers who assume that these writes happen in program order. The following two litmus tests express this common but unsound assumption.

The **ordered-same-file-overwrites** test checks whether overwrites to the same file are ordered:

```
initial:
  N ← 40 * 1024
  f ← creat("f", 0600)
  write(f, "0" * N)
main:
  pwrite(f, "1", N - 1) # overwrite at offset N - 1
  pwrite(f, "1", 0)     # overwrite at offset 0
exists?:
  content("f")[0] = "1" ∧ content("f")[N - 1] = "0"
```

The **ordered-two-file-overwrites** test checks whether overwrites to different files are ordered:

```
initial:
  f ← creat("f", 0600)
  g ← creat("g", 0600)
  write(f, "0")
  write(g, "0")
main:
  pwrite(f, "1", 0) # overwrite f at offset 0
  pwrite(g, "1", 0) # overwrite g at offset 0
exists?:
  content("f") = "0" ∧ content("g") = "1"
```

3.3 Litmus tests for directory operations

Litmus tests for directory operations check whether operations to the same directory or to two different directories can be reordered. The tests are similar to those for file overwrites and so we omit them. There is a debate about whether POSIX prescribes different guarantees for directory operations as opposed to file operations [6].

3.4 Litmus tests for file-directory operations

The interplay between file and directory operations is subtle and can have surprising consequences. We develop three litmus tests to demonstrate such behavior. The first test checks whether updates to a file and to its parent directory can be reordered. The next two tests check two widely assumed *atomicity properties* of `rename`, which are referred to as "safe rename" in the literature. We show (§3.5) that "safe rename" properties are not guaranteed by most file systems, despite widespread beliefs otherwise [46, 61, 62].

file system	kernel	PA	ARVR	ACVR
journaling				
`ext4`	Linux	●	●	●
`xfs`	Linux	○	●	●
log-structured				
`f2fs` [39]	Linux	●	●	●
`nilfs2`	Linux	○	●	●
copy-on-write				
`btrfs` [68]	Linux	○	○	●
soft updates (SU+J)				
`ufs2` [50]	FreeBSD	●	●	●

Figure 3. Characterizing file system implementations with litmus tests using FERRITE:
○: FERRITE found no execution satisfying any of the test's predicates.
●: FERRITE found an execution which allows unintuitive behavior and may surprise application writers.

Implied-directory-fsync. This test checks whether an `fsync`-ed update to a file can be reordered with the creation of that file (i.e., an update to its parent directory):

```
main:
  f ← creat("file", 0600)
  write(f, data)
  fsync(f)
  mark("written")
exists?:
  marked("written") ∧ content("file") = ∅
```

Atomic-replace-via-rename (ARVR). The ARVR test checks whether replacing file contents via `rename` is atomic across crashes (i.e., the file contains either old or new data):

```
initial:
  g ← creat("file", 0600)
  write(g, old)
main:
  f ← creat("file.tmp", 0600)
  write(f, new)
  rename("file.tmp", "file")
exists?:
  content("file") ≠ old ∧ content("file") ≠ new
```

Atomic-create-via-rename (ACVR). The ACVR test is a variation of the ARVR test in which the destination file name does not exist initially. The test checks whether the destination file is either absent or present in its entirety:

```
main:
  f ← creat("file.tmp", 0600)
  write(f, data)
  rename("file.tmp", "file")
exists?:
  content("file") ≠ ∅ ∧ content("file") ≠ data
```

3.5 Characterizing file systems with litmus tests

We applied FERRITE to check the litmus tests presented in this section against commonly used file systems on Linux 4.1 and FreeBSD 10.2, all in default configuration mode (§5 will detail how FERRITE works). Our evaluation covers file systems using a wide range of crash recovery techniques. As explained earlier, most file systems reorder file overwrites and some even reorder directory operations; this is consistent with previous reports and we omit the corresponding results.

We highlight three tests—PA, ARVR, and ACVR—for which FERRITE's results differ from common belief and previous studies. The results are shown in Figure 3. Unlike ambiguous documentation, these litmus tests show precisely which unexpected behaviors are allowed by each file system.

Prefix-append (PA) is not guaranteed by several file systems, including `ext4`, Samsung's new file system `f2fs`, and FreeBSD's default `ufs2`. This is particularly surprising for `ext4`'s default `data=ordered` mode. According to this mode's documentation [42],

> [`ext4`] logically groups metadata information related to data changes with the data blocks into a single unit called a transaction. When it's time to write the new metadata out to disk, the associated data blocks are written first.

Many application writers and researchers interpreted this to mean that metadata updates (e.g., file size) would always reach disk after file data, and thus the prefix-append property should hold [10, 55, 62].

However, we observed files with out-of-thin-air trailing nulls (i.e., binary zeros): a crash after appending 2500 "a"s and then 2500 "b"s resulted in a file of 4096 bytes, with 2500 "a"s and 1596 nulls. We further observed that `ext4` updated the file size to 4096 without writing the file data block (i.e., 2500 "a"s and 1596 "b"s) first, which it did not consider as an "associated data block." When ChromeOS developers observed similar behavior, the `ext4` developers explained that it is "working as intended" [10]. Failing to anticipate this behavior causes application crashes (upon recovery) or even corrupted data.

Atomic replace-via-rename (ARVR) is not guaranteed by most file systems, including `ext4`. After the "`ext4` data loss" incident [84], the `ext4` developers changed the behavior of the file system to accommodate applications that use the replace-via-rename pattern but do not call `fsync` before `rename` (see Figure 1). The new behavior is described in the `ext4` documentation [42]:

> [T]he data blocks of the new file are forced to disk before the `rename()` operation is committed. This provides roughly the same level of guarantees as `ext3`, and avoids the "zero-length" problem that can happen when a system crashes before the delayed allocation blocks are forced to disk.

The documentation seems to imply that it is safe to omit `fsync` in replace-via-rename, as understood by many application writers and researchers [41, 46, 62]. For example, GNOME

and dpkg removed these calls to `fsync` on `ext4` to improve performance, causing some users to lose data.

Applying FERRITE to the ARVR litmus test shows this optimization to be unsound on `ext4`. Recent `ext4` *does* have a special workaround to reduce the chance of replace-via-rename producing zero-length files by forcing out the new data blocks upon `rename`. However, `rename` does *not* wait for this flush to complete, and therefore provides no atomicity guarantee—it is possible to end up with only partial new content after a crash. Of the file systems we tested, `btrfs` is the only one that provides the replace-via-rename atomicity guarantee [13].

Atomic create-via-rename (ACVR) offers atomicity guarantees different to those of atomic replace-via-rename (ARVR) on some file systems. For example, `btrfs` guarantees ARVR, but not ACVR. For this reason, the term "safe rename" [62] can cause confusion for application writers.

Our results show that litmus tests provide a precise and intuitive way of communicating crash-consistency behavior. In contrast, documentation for file systems is often ambiguous, and so the intent of file system developers can be easily misinterpreted by application writers.

4. Formal specifications

Litmus tests are an effective medium for communicating file system crash behavior to application writers. But tests alone are insufficient for constructing automated reasoning tools, which improve application reliability and programmer productivity. This section presents a formal framework for specifying file system crash-consistency models, analogous to memory consistency models. In §6, we use formal crash-consistency models expressed in this framework to develop automated verification and synthesis tools.

A crash-consistency model defines the permissible states of a file system after a crash. We develop two styles of specification: *axiomatic* and *operational*. Axiomatic models describe valid crash behaviors declaratively, using a set of axioms and ordering relations, while operational models are abstract machines that simulate relevant aspects of file system behavior (such as crashes, caching, etc.). We show both kinds of models for two example file systems: `seqfs`, an ideal file system with strong crash consistency guarantees (analogous to sequential consistency [38]), and `ext4`, a real file system with weak consistency guarantees (analogous to a weak memory model).

4.1 Axiomatic specifications

An axiomatic crash-consistency model consists of a set of rules, or axioms, that specify whether a given *execution* of a program is allowed. We first define programs and their executions (Definitions 1–3), and then define crash-consistency models (Definitions 4–7).

Program Executions. Programs update the state of an underlying *file system* (Definition 1). As is common in axiomatic specifications of memory models, we describe executions of programs in terms of the *events* (Definition 2) that they generate, such as writing a block to a file or renaming a file. These events are atomic, so we model high-level operating system procedures as emitting sequences of atomic events (e.g., a system call to write $k \times n$ bytes generates a sequence of n atomic write events that write k bytes each). We distinguish between two kinds of events: *update events* modify the state of the underlying file system, while *synchronization events* constrain the order in which events may be generated when executing a program. A sequence of events is called a *trace* (Definition 3), and a crash-consistency model decides whether a given trace is permissible for a program.

Definition 1 (File Systems)**.** A *file system* σ is a tuple $\langle \sigma_{meta}, \sigma_{data} \rangle$, where σ_{meta} is a map from *object identifiers* (i.e., names) to natural numbers (analogous to inode numbers), and σ_{data} is a map from natural numbers (inode numbers) to *file system objects* (analogous to inodes). Object identifiers i are drawn from a countably infinite set of symbols $\mathbb{I} = \mathbb{F} \cup \mathbb{D}$, consisting of disjoint sets of file identifiers $\mathbb{F}$ and directory identifiers $\mathbb{D}$. We write f and d to denote file and directory identifiers, respectively, and we write $\sigma(i)$ to denote the object $\sigma_{data}(\sigma_{meta}(i))$. A file object $\sigma(f)$ is a tuple $\langle b, m \rangle$, where b is a finite string of bits, and m is a finite key-value map of file metadata. A directory object $\sigma(d)$ is a map from object identifiers to natural numbers. We write $\sigma(i) = \bot$ when the object $\sigma(i)$ has not been created; that is, $\sigma_{meta}(i) = \bot$.

Definition 2 (Events)**.** A *file system event* represents an atomic access to the file system σ. There are two kinds of events: *update events* and *synchronization events*. Update events atomically modify the file and directory objects in σ. Synchronization events synchronize accesses to (parts of) σ.

Update events include writes to file (meta)data and updates to directory maps:

- *write*(f, a, d) updates the address a of the file f to contain the data d (i.e., $b[a] = d$ for $\sigma(f) = \langle b, m \rangle$ after the update).
- *setattr*(f, k, v) updates the metadata attribute k of the file f to contain the value v (i.e., $m[k] = v$ for $\sigma(f) = \langle b, m \rangle$ after the update).
- *extend*(f, a, d, s) sets the "size" attribute of f to s, and writes the data d to the address a (i.e., $m[\text{"size"}] = s$ and $b[a] = d$ for $\sigma(f) = \langle b, m \rangle$ after the update). *extend* is used to implement file append operations.
- *link*(i_1, i_2) updates i_2's binding to reflect that of i_1 (i.e., $\sigma_{meta}(i_2) = \sigma_{meta}(i_1)$ and $\sigma_{meta}(i_1)$ is unchanged after the update).
- *unlink*(i) removes the binding for i (i.e., $\sigma_{meta}(i) = \bot$ after the update).

- *rename*(i_1, i_2) links i_2 to i_1 and unlinks i_1 when $i_1 \neq i_2$; otherwise nothing changes.

Synchronization events include write barriers, as well as externally observable non-file-system events (such as sending a message over the network) and events for the beginning and end of a transaction:

- *fsync*(i) synchronizes accesses to the file or directory i.
- *sync*$()$ synchronizes accesses to all files and directories in the file system.
- *mark*(l) marks an externally observable event identified by the unique label l.
- *begin*$()$ begins a new transaction.
- *commit*$()$ ends the current transaction.

Definition 3 (Traces)**.** A *trace* t_P is a sequence of file system events generated during the execution of a program P. We write $\leq_{t_P}$ to denote the total order on events induced by the trace t_P: $e_1 \leq_{t_P} e_2$ iff e_1 occurs before e_2 in the trace t_P. Each program P has a *canonical trace*, τ_P, which is free of crashes and generates events in the order specified by the syntax of P. A trace t_P is *valid* iff it satisfies the following conditions:

- t_P is a permutation of τ_P.
- t_P respects the synchronization semantics of τ_P. That is, $e_i \leq_{t_P} e_j$ when $e_i \leq_{\tau_P} e_j$ and any of the following hold:
  - e_i is an *fsync*, *sync*, *mark*, *begin*, or *commit* event;
  - e_j is a *sync*, *begin*, or *commit* event;
  - e_j is an *fsync* event on i and e_i is an update event on i.
- t_P respects the update semantics of τ_P. That is, applying the update events in the order specified by t_P to a file system σ leaves σ in the same state as applying the update events in the order specified by τ_P.

A *crash* trace, c_P, is a prefix of a valid trace t_P that respects transactional semantics: c_P contains the same number of *begin* and *commit* events. Both the empty trace and t_P itself are crash traces of t_P. We omit the subscript P from the notation when the program P is irrelevant or clear in context.

Example 1. To illustrate our model of file systems, events, and traces, consider the following litmus test:

```
initial:
  f ← creat("f", 0600)
  g ← creat("g", 0600)
  write(f, "0")
  write(g, "0")
main:
  pwrite(f, "1", 0)
  pwrite(g, "1", 0)
  fsync(g)
exists?:
  content("f") = "0" ∧ content("g") = "1"
```

Suppose that we execute this test on an empty file system σ (i.e., $\sigma(i) = \bot$ for all identifiers i). The initial setup updates the abstract state of σ to include the bindings $\sigma(\mathsf{f}) = \langle b, m\rangle$ and $\sigma(\mathsf{g}) = \langle b, m\rangle$, where $b =$ "0" and $m = \{\mathsf{permissions} \mapsto \text{"0600"}, \ldots\}$. Assuming that the `pwrite` operations generate one *write* event each, the canonical trace for the litmus test is $\tau = [e_0, e_1, e_2]$, where $e_0 =$ *write*$(\mathsf{f}, 0, \text{"1"})$, $e_1 =$ *write*$(\mathsf{g}, 0, \text{"1"})$, and $e_2 =$ *fsync*(g). The valid traces, besides τ, are $t_1 = [e_1, e_0, e_2]$ and $t_2 = [e_1, e_2, e_0]$. None of the other permutations of τ are valid traces because they violate barrier semantics. Any prefix of τ, t_1, or t_2 forms a crash trace—intuitively, a crash trace represents the crash of a valid execution at a particular point.

Crash-Consistency Models. A crash-consistency model (Definition 4) determines which valid program traces are permissible, and therefore, what file system states may be observed after a crash. The strongest consistency model is *Sequential Crash-Consistency* (SCC) (Definition 5), which permits no re-ordering of events in the canonical trace. That is, an execution of a program P may only emit the canonical trace τ_P. As a result, the crash behavior of an application running under SCC is easy to reason about, and it can be connected directly to program text. SCC can be thought of as the analogue of sequential consistency for memory models.

Real file systems, however, implement *weaker* models (Definition 6), which permit additional valid traces. In particular, we say that a crash consistency model M_1 is weaker than a model M_2 iff M_1 permits a superset of the valid traces permitted by M_2 for every program. The model provided by the `ext4` file system, for example, is weaker than SCC, permitting re-orderings of updates to different files (among other relaxations). Definition 7 gives an axiomatic specification of `ext4` in our framework, obtained by reading the documentation and empirically observing its behavior on litmus tests using FERRITE. As we show in §6, such a specification clarifies the contract between the application and the underlying file system, and enables automatic synthesis of a minimal set of barriers that ensure atomicity and durability after a crash.

Definition 4 (Crash-Consistency Model)**.** A crash-consistency model M relates an arbitrary valid trace of a program P to its canonical trace. We say that M *permits* t_P iff $M(t_P, \tau_P)$ evaluates to true.

Definition 5 (Sequential Crash-Consistency (SCC))**.** A valid trace t_P of a program P is *sequentially crash-consistent* iff $t_P = \tau_P$. That is, $\mathsf{SCC}(t_P, \tau_P) \triangleq t_P = \tau_P$.

Example 2. Consider the litmus test from Example 1. For this test, SCC permits only the canonical trace $\tau = [e_0, e_1, e_2]$. No crash trace (i.e., prefix) of τ results in the specified final state $\mathsf{content}(f) =$ "0" $\wedge$ $\mathsf{content}(g) =$ "1", and so the SCC model does not allow the surprising behavior of this test.

Example 3. FSCQ [14] is a file system that is verified to be crash-safe: a machine-checkable proof in Coq shows that FSCQ will always recover correctly after a crash. FSCQ wraps each POSIX system call into a transaction committed to a write-ahead log, and so each system call is both atomic

and persistent. In our formalization, the canonical trace τ_P for a program P running on FSCQ includes a *begin* event before the sequence of events generated by each system call, and a *commit* event after those events. For example, a `write` system call on FSCQ may produce the canonical trace $\tau_P = [\mathit{begin}, \mathit{write}_0, \ldots, \mathit{write}_n, \mathit{commit}]$ (with one write_i per block). This canonical trace yields only two valid crash traces: the empty trace and τ_P itself.

Definition 6 (Weaker Crash-Consistency Model)**.** A crash-consistency model M_1 is *weaker* than the model M_2 iff $M_2(t_P, \tau_P) \implies M_1(t_P, \tau_P)$ for every valid trace t_P and its corresponding canonical trace τ_P.

Definition 7 (`ext4` Crash-Consistency)**.** Let t_P be a valid trace and τ_P the corresponding canonical trace. We say that t_P is `ext4` *crash-consistent* iff $e_i \leq_{t_P} e_j$ for all events e_i, e_j such that $e_i \leq_{\tau_P} e_j$ and at least one of the following conditions holds:

1. e_i and e_j are metadata updates to the same file: $e_i = \mathit{setattr}(f, k_i, v_i)$ and $e_j = \mathit{setattr}(f, k_j, v_j)$.
2. e_i and e_j are writes to the same block in the same file: $e_i = \mathit{write}(f, a_i, d_i)$, $e_j = \mathit{write}(f, a_j, d_j)$, and $\mathit{sameBlock}(a_i, a_j)$, where *sameBlock* is an implementation-specific predicate.
3. e_i and e_j are updates to the same directory: $\mathit{args}(e_i) \cap \mathit{args}(e_j) \neq \emptyset$, where $\mathit{args}(\mathit{link}(i_1, i_2)) = \{i_1, i_2\}$, $\mathit{args}(\mathit{unlink}(i_1)) = \{i_1\}$, and $\mathit{args}(\mathit{rename}(i_1, i_2)) = \{i_1, i_2\}$.
4. e_i is a write and e_j is an extend to the same file: $e_i = \mathit{write}(f, a_i, d_i)$ and $e_j = \mathit{extend}(f, a_j, d_j, s)$.

Example 4. Consider again the litmus test from Example 1. The `ext4` crash-consistency model permits every valid trace (i.e., τ, t_1, and t_2) of the program. Crash traces (i.e., prefixes) of two of these traces (t_1 and t_2) satisfy the predicate $\texttt{content}(f) =$ "0" $\wedge$ $\texttt{content}(g) =$ "1", and so unlike SCC, `ext4` allows the surprising behavior of this test. The programmer can ensure that the test exhibits only the SCC behaviors on `ext4` by inserting an `fsync(f)` barrier after the write to `f`.

Note that a crash-consistency model does not include an explicit model of hardware disk behavior. Instead, the reorderings and caching behaviors of hardware disks are implicit in the allowed reorderings and in the semantics of update and synchronization events. In §5, we describe how FERRITE simulates the intermediate states of the hardware disk after a crash, and so a crash-consistency model developed with FERRITE implicitly captures hardware behavior.

4.2 Operational specifications

An operational crash-consistency model takes the form of a non-deterministic state machine M, which uses idealized components—maps and tuples—to abstract the implementation details of real file systems. We model the persistent (on-disk) and volatile (in-core) state of a file system σ with a tuple $\langle \sigma_{\mathit{inCore}}, \sigma_{\mathit{onDisk}} \rangle$, where σ_{inCore} and σ_{onDisk} are themselves tuples of the form $\langle \sigma_{\mathit{meta}}, \sigma_{\mathit{data}} \rangle$, as defined previously (Definition 1). Events update the volatile state of the files they interact with, occurring in the order specified by the syntax of a program P. The state machine M non-deterministically chooses to persist (some of) these updates to disk, or to crash.

$$\frac{\sigma' = \textsc{Flush}(\textsc{Apply}(P[p], \sigma))}{\langle \sigma, p \rangle \to \langle \sigma', p+1 \rangle}\ \textsc{StepSeq}$$

$$\frac{}{\langle \sigma, p \rangle \to \langle \sigma, \bot \rangle}\ \textsc{Crash}$$

Figure 4. An operational model for SCC (Definition 5).

$$\frac{\sigma' = \textsc{Apply}(P[p], \sigma)}{\langle \sigma, p \rangle \to \langle \sigma', p+1 \rangle}\ \textsc{Step} \qquad \frac{}{\langle \sigma, p \rangle \to \langle \sigma, \bot \rangle}\ \textsc{Crash}$$

$$\frac{\sigma' = \textsc{PartialFlush}(\sigma)}{\langle \sigma, p \rangle \to \langle \sigma', p \rangle}\ \textsc{NonDet}$$

Figure 5. A sample of the `ext4` operational model.

Figure 4 shows an example operational specification of sequential crash-consistency. We model the machine state as a pair $\langle \sigma, p \rangle$, where $\sigma = \langle \sigma_{\mathit{inCore}}, \sigma_{\mathit{onDisk}} \rangle$ is the current state of the file system, and p the program counter (an index into the program P). The APPLY function executes the semantics of an event as given in Definition 2. The FLUSH function flushes the volatile state of all files to disk. The STEPSEQ rule says that the volatile state is flushed to disk at every step. In other words, $\sigma_{\mathit{inCore}} = \sigma_{\mathit{onDisk}}$ at any point during execution. The non-deterministic rule CRASH can halt the program at any point (denoted by setting the program counter to $\bot$). It is easy to see that a machine executing these two rules can only produce the canonical trace (or its prefix) of any program P.

Figure 5 shows the key rules for an operational specification of `ext4`. The `ext4` model permits more non-determinism than the sequential model: the NONDET rule may transition without performing the next event in the program. The PARTIALFLUSH function non-deterministically flushes none, some, or all of each file in the file system. For space reasons, we omit a full description of the rules for `ext4`, but in §6.2, we describe a Dafny-based verifier for application-level properties that uses this operational crash-consistency model.

5. Making specifications executable

This section describes FERRITE, a suite of automated tools for reasoning about crash-consistency models. FERRITE consists of two tools—an explicit enumerator and a bounded model checker—which exhaustively explore all possible crash behaviors of a litmus test (§3). The enumerator executes litmus tests against actual file system implementations to determine the set of all possible crash behaviors. The model checker executes litmus tests symbolically against an axiomatic specification (§4.1). Together, these tools enable

disk commands	description
`write`($blockID$, $data$)	write data to a given block
`flush`()	flush disk cache to stable storage
`trim`($blockID$)	wipe a given block
`mark`($label$)	label externally observable event

Figure 6. Disk commands supported by FERRITE's virtual disk; `mark` is a pseudo-command that corresponds to the pseudo-function of the same name in litmus tests (see §3.1).

file system developers to create high fidelity specifications of crash behavior: the model checker ensures that the formalizations (dis)allow representative behaviors encoded in litmus tests, and the enumerator ensures that the litmus tests are indeed representative of the behavior of the actual file system implementations.

5.1 Executing tests against file system implementations

The possible reordering behaviors of a file system are often subtle even to their developers. Prior experience with memory models has demonstrated the utility of tools such as DIY [3] that execute litmus tests against real hardware to observe possible outcomes. To help specification writers understand the behaviors allowed by a file system, FERRITE includes an explicit enumerator that executes all possible crash traces of a litmus test against an actual file system implementation.

The enumerator takes as input a litmus test P and a target file system, and reports all possible outcomes of P on that file system, including potential crashes. The system calls made by P pass through several layers of the I/O stack (Figure 2), each of which is free to reorder or buffer them according to its own rules. Eventually, the operating system produces *disk commands*, shown in Figure 6, which are sent to the hardware storage device. The storage device is then free to reorder and buffer those events according to its own rules. The state of persistent storage after a crash therefore depends on both the trace of disk commands and the behavior of the storage device.

To generate all possible crash states of the test P, FERRITE needs to observe two sets of traces: (1) all possible sequences of disk commands sent to the hardware, and (2) all possible reorderings of those commands made by the hardware. FERRITE determines the possible sequences of disk commands by executing P many times, as is common for testing memory models [3, 4]. But producing hardware reorderings is trickier, since it also requires producing all intermediate states of the persistent storage in order to simulate crashes. In particular, interposing on the hardware to inspect intermediate states or produce a crash at a certain point would be expensive and imprecise. Instead, FERRITE abstracts the behavior of storage hardware with a *disk model*, producing only the command reorderings that satisfy a chosen disk model.

A disk model captures the allowed behaviors of the disk hardware. FERRITE implements a default disk model that resembles a disk with a large volatile cache and arbitrary schedules. Specifically, it allows two disk operations o_i and o_j to be reordered unless one of the following is true:

- either o_i or o_j is a `flush`;
- o_i and o_j are changes to the same block;
- o_i is a `mark` command.

Capturing hardware behavior in a disk model allows FERRITE to easily simulate a crash at any point during P's execution, and observe the resulting intermediate state. This guarantees that all possible crash states are observed.

The enumerator executes litmus tests by intercepting system calls and forwarding them to an underlying guest OS running in QEMU [7]. The guest OS has an attached virtual disk provided by FERRITE. The target file system, running in the guest OS, issues disk commands to this virtual disk, which are forwarded to FERRITE to be recorded in a trace. Once execution of P is complete, the enumerator produces all possible reorderings of the trace according to the chosen disk model, and for each trace, produces all possible prefixes. Each prefix produces a disk image file that corresponds to a possible disk state after a crash. The enumerator finally mounts the disk image as a loopback device in the guest OS to recover the state of the files written by P. Optionally, the enumerator can then verify the predicates in the final section of the litmus test (§3.1) against the possible states. We have applied enumeration to run litmus tests against file systems on Linux and FreeBSD, with results presented in §3.5 earlier.

Limitations. The result of the enumerator is dependent on a particular implementation of the file system and the rest of the I/O stack. It is useful to show that certain system calls *can* be reordered by the file system (assuming QEMU and the I/O stack are correct). It cannot prove that two system calls must always be ordered even if it never sees a counter-example in the output; we cross-validated with documentation and with file system developers for those cases.

5.2 Executing tests against specifications

Axiomatic specifications of crash consistency are useful only to the extent that they faithfully capture the behavior of file system implementations. Prior experience with memory models [82] has shown that it is easy to accidentally write a specification that is under-constrained (allowing behaviors that should be forbidden) or over-constrained (forbidding behaviors that should be allowed). To help file system developers avoid these pitfalls when specifying crash-consistency models, FERRITE includes a model checker that symbolically executes a litmus test against a specification, ensuring that the specification (dis)allows the behaviors encoded in the test.

The model checker takes as input a litmus test P (§3) and an axiomatic specification of a crash-consistency model M (§4), and checks whether the predicates specified in the `exists?` section of the test are satisfied by any execution of the body. Conceptually, the model checker works by generating all crash traces of P that are permitted by M

```
; SCC permits no reorderings
(define (SCC-reorder? e1 e2) #f)

; ext4 forbids the reorderings in Definition 7
(define (ext4-reorder? e1 e2)
  (and
   (not (metadata-same-ino-deps? e1 e2))            ; (1)
   (not (same-file-block-deps? e1 e2 BLOCK_SIZE))   ; (2)
   (not (dir-same-ino-deps? e1 e2))                 ; (3)
   (not (file-write-extend-deps? e1 e2))))          ; (4)
```

Figure 7. Rosette specifications of the reordering rules for the SCC (Definition 5) and `ext4` (Definition 7) models.

(Definition 4), and checking each predicate against that set. In particular, the checker produces the set of crash traces $T = \{c_P \mid c_P \text{ is a crash trace of } t_P \text{ and } M(t_P, \tau_P)\}$, where τ_P is the canonical trace of P. Then, it checks whether any predicate in P is satisfied by some trace in T. If so, the crash-consistency model M allows some surprising behavior. This outcome can be cross-checked against the actual file system implementation using FERRITE to confirm the model is not too weak.

We implemented the model checker in Rosette [80], a programming language that extends Racket [23] with features for program synthesis and verification, based on an underlying SAT or SMT solver [21, 81]. The specification writer uses Rosette to encode the axiomatic specification of the crash-consistency model M, and the model checker provides a simple DSL for expressing the litmus test P. The specification M is written simply as a procedure that takes as input two events from the canonical trace, $e_1 \leq_{\tau_P} e_2$, and returns true iff M allows these two events to be reordered. Figure 7 shows the specifications for the SCC and `ext4` models, both of which are direct translations of Definitions 5 and 7, respectively, into Rosette's syntax. Given these inputs, the model checker uses Rosette to symbolically execute P against M, relying on the SMT solver to implicitly check P's `exists?` predicates against the traces permitted by M, as described above.

We applied the model checker to validate the SCC and `ext4` specifications against our litmus tests (§3). The SCC specification, by construction, does not permit any of the surprising results in the tests, and our model checker confirmed this to be the case. The `ext4` specification, on the other hand, is expected to allow some of the litmus tests, since the actual `ext4` implementation does as well (§3.5). The model checker confirms that the specification behaves like the implementation on all tests. All of the checks complete within a few seconds. In general, implementing a SAT/SMT-based tool that scales to our litmus tests—which manipulate files with thousands of bytes—would require hand-crafted encodings and significant time investment (as with the specialized encodings [48, 82, 90] for checking litmus tests against memory models). However, Rosette's aggressive partial evaluation allows our model checker to scale to these litmus tests without manual intervention.

6. Experience with specifications

In this section we demonstrate that formal specifications are useful for building tools that can aid application writers. We also discuss how applications can benefit from support from the operating system and hardware.

6.1 Synthesis

Rather than require application writers to manually ensure crash safety of their programs, formal crash-consistency specifications allow us to *synthesize* sufficient barriers for an implementation so that it satisfies the desired crash-safety properties. In particular, the application writer develops the program P assuming sequential crash-consistency. The synthesizer then transforms P, by inserting a minimal set of barriers, so that the resulting program P' behaves just like P under a given weaker crash-consistency model, such as `ext4`.

We built a proof-of-concept synthesizer on top of the FERRITE model checker described in §5.2. The synthesizer uses Rosette's implementation [80] of counterexample-guided inductive synthesis (CEGIS) [73, 74] to generate candidate programs P' and verify that they preserve P's guarantees on every permitted trace. The synthesizer generates candidate programs P' by inserting new `fsync` invocations into P. Moreover, the synthesizer automatically *optimizes* the insertion of barriers, such that the program P' contains the minimal number of additional `fsync` invocations necessary to guarantee the desired safety properties.

To illustrate, consider the atomic-replace-via-rename (ARVR) litmus test in §3.4. This program guarantees atomicity under SCC. But recall from Figure 3 that it does not do so on `ext4`: after a crash, it is possible for "`file`" to contain neither the old nor the new file contents. To guarantee atomicity, we must insert additional synchronization. A naive solution would be to insert an `fsync` after each call in this program, but this is unnecessarily conservative and potentially inhibits performance. Our synthesizer instead chooses to insert an `fsync(f)` after the `write` operation. This single `fsync` is sufficient to guarantee atomicity for this program on `ext4`. The synthesizer automatically generates and verifies this program in under 30 seconds, with similar performance on all the other litmus tests in §3. While our synthesizer is a proof-of-concept, common optimizations (e.g., Fender [37]) would allow it to scale to larger programs.

6.2 Verification

To demonstrate the use of operational crash-consistency models (§4.2), we developed a verification framework in Dafny [40] based on our operational model of `ext4`, and we applied it to prove crash safety of several small programs. Dafny proofs in our framework are unbounded, unlike those produced by the FERRITE model checker. In particular, given

a procedure that takes as input a file, Dafny proves its crash-safety with respect to all possible file sizes. FERRITE, in contrast, expects a bound on the file size, as provided in our litmus tests.

Our framework models the file system with Dafny's built-in strings and maps: files are strings, and directories are maps. Each system call is simply a procedure with associated pre- and post-conditions. We express non-determinism using Dafny's havoc statements. For example, the CRASH rule in Figure 5 can be modeled as "`if (*) crash();`," where the havoc statement $*$ means that the given branch is taken non-deterministically. We similarly use havoc statements to implement non-deterministic flushes (the NONDET rule in Figure 5) and to represent garbage data. Because Dafny closely resembles a conventional imperative language, applications can be implemented in Dafny against our abstract interface, and extracted to implementations against real file system interfaces.

We verified crash safety of several small programs using our Dafny framework. For example, we verified the unbounded version of the ARVR litmus test used in synthesis above. Our framework's use of Dafny's language features made proof automation highly effective: it required only 6 annotations to verify ARVR.

6.3 OS support

Recent research in memory consistency models [1] designs programming abstractions that hide the details of memory reordering where possible (using concurrency libraries), but exposing them when necessary for peak performance (such as through C11 atomics). In addition to providing a basis for synthesis and verification tools, crash-consistency models also suggest a similar design for programming abstractions that offer crash-consistency guarantees. This section discusses two such interfaces. The first is a simple interface to expose the key characteristics of a file system's crash-consistency behavior to the application, akin to C11 atomics. The second is a radical design that provides applications with sequential crash-consistency (SCC) guarantees. We prototyped both designs using our litmus tests and Dafny verification framework.

Exposing file system characteristics. File system developers often hesitate to promise strong crash-consistency guarantees. Instead, many applications try to exploit the characteristics of specific file systems and implement per-file-system optimizations. Consider the following code snippet from GNOME [45], which skips an `fsync` call for replace-via-rename if the file system magic number is a known constant:

```
struct statfs buf;
fstatfs(fd, &buf);
if (buf.f_type == BTRFS_SUPER_MAGIC || buf.f_type == ...)
      // ... skip fsync(fd) ...
```

This practice is risky because the file system's behavior may depend on specific versions or mount options, which applications can easily overlook. One solution is to provide interfaces for applications to query the possible behaviors of the current file system. For example, a flag could expose whether the file system guarantees atomic replace-via-rename (§3):

```
if (buf.f_charcs & FC_ATOMIC_REPLACE_VIA_RENAME) ...
```

Applications can then make their optimizations and expectations explicit, making them more portable and easier to reason about. We implemented a prototype on Linux by exposing such information through the reserved bits of `statfs`. §7 discusses other proposed high-level interfaces (e.g., file system transactions and barriers).

End-to-end I/O stack. Reasoning about crash consistency is made difficult by the complexity of the I/O stack, as shown in §2.2. An alternative approach, embraced by exokernel designs [22, 34], is to reduce or eliminate the kernel from the I/O path. While these designs focus on improving end-to-end performance, we believe that they can also offer better reliability and more explicit crash-consistency guarantees.

As a demonstration, we implemented a prototype I/O stack by porting the persistent log and a `megaraid` driver from the Arrakis operating system [60] to Linux. It has a small code base—200 LOC for the log and 4,000 LOC for the driver—and completely bypasses the existing file system I/O stack. The persistent log provides strong crash-consistency guarantees: log entries reach disk in same order as they are appended, and applications will see only complete log entries after the system crashes. We derived a formal proof of the properties of the persistent log implementation using Dafny (with 25 annotations). To measure performance, we modified LevelDB 1.18 to use the log and measured the time to append 100,000 `Put(key, val)` entries to the log; the backend was an LSI Logic MegaRAID SAS-3 3108 RAID controller. The time reduced from 14.4 s on `ext4` to 4 s when using the log. We believe that this is a promising direction to build end-to-end storage systems with well-defined crash behavior.

7. Related work

Our work is motivated by past research that finds crash-safety bugs in both applications and file systems, that explores better programming abstractions for crash safety, and that provides formal models of file systems.

Bugs in storage systems. Storage systems need to tolerate system crashes and are difficult to get right [47, 64]. Techniques such as model checking [88, 89] and static analysis [70] have been proven to be effective in finding bugs in applications and file systems. FERRITE differs from these tools in two ways. First, it does not focus on finding bugs in file systems. Rather, §3.5 shows incorrect or imprecise crash models used in previous work (i.e., bugs in their models). Second, Ferrite presents the first formalization of file system crash-consistency models; these models can be used with automated reasoning tools, as we showed through synthesis

and verification in §6. The models in previous work are not formal models and cannot be used this way.

FERRITE was inspired by recent work from Pillai et al. [62] and Zheng et al. [92] on crash-safety bugs. Their work showed that even applications carefully designed and engineered to tolerate crashes may be vulnerable to data losses, and that it remains challenging to develop crash-safe applications on top of POSIX file systems.

A major challenge for both application writers and file system developers is that the POSIX file interface under-specifies crash behaviors, leading to conflicting interpretations. Our crash-consistency models address this challenge by providing litmus tests and formal specifications for precisely describing crash behavior. Precise, formal models clarify misinterpretations and provide a basis for automated reasoning tools.

Formal specification and verification of file systems. Several projects have explored formally specifying and verifying file systems. Bevier et al. specified the Synergy file system in Z [9] and verified its implementation in ACL2 [8]. Kang and Jackson used Alloy [35] to model a file system for flash-based storage devices, with a focus on flash-specific features such as wear leveling. Wenzel used Isabelle/HOL [86] to formalize common Unix file system abstractions and reason about their security properties. The Commuter tool [18] formalizes the commutativity of the POSIX interface and studies its scalability implications. SibylFS [66] is a recent effort in formalizing POSIX and testing implementation conformance. To our knowledge, we propose the first formal framework for describing crash-consistency behaviors of POSIX file systems. On the other hand, our framework focuses on crash consistency, and we do not model POSIX details such as permissions and symbolic links.

Another line of work is applying formal verification to implementing a POSIX-like file system [33]. On-going work includes BilbyFs [36], FSCQ [14], and Schellhorn et al.'s verified flash file system [71]. It would be interesting to apply our framework to analyzing the crash-consistency guarantees of these file systems once they are complete and available.

File system interfaces. One limitation of POSIX is that applications have to resort to expensive `fsync` calls to express ordering requirements even when they do not need durability. Kernel developers have proposed to extend Linux with an `fbarrier` system call [20] to improve performance for such applications. Other extensions to better support ordering include Featherstitch [24], OptFS [16], and xsyncfs [57].

A number of proposals offer transactional semantics for file system operations [52, 59, 63, 65, 76, 85, 87]. Emerging storage devices provide new hardware primitives, such as Fusion-io's multi-block atomic writes and new SCSI commands (ATOMIC WRITE and draft SCATTERED WRITE), that could be used to simplify file system implementations and provide stronger crash-consistency guarantees [32, 53].

Memory consistency models. Development and analysis of memory consistency models is an active area of research, and our approach draws on the general themes and ideas from this work. A memory model determines the semantics of reads in a multi-threaded shared-memory system; in particular, it specifies which writes to a memory location a given read may observe. Memory models are described through a combination of litmus tests (e.g., [3, 4]), axiomatic specifications (e.g., [2, 82, 90]), and operational specifications (e.g., [12, 91]). Litmus tests have the advantage of being easy to understand; operational models offer a direct basis for building simulation tools; and axiomatic models are best suited for formal reasoning and concise descriptions of complex systems (e.g., [49]). Many tools have been developed to check that these forms of specification agree with each other (e.g., [48, 82, 90]), as well as with the behavior exhibited by hardware (e.g., [4]).

Compared to memory models, crash-consistency models have simpler ordering constraints: all possible program behaviors can be described in terms of reorderings of a canonical trace. Memory models, in contrast, generally involve constraints over many ordering relations [2] and even many speculative executions [49]. On the other hand, crash-consistency models involve richer data structures, such as files and directories, and the notion of crashes.

8. Conclusion

The guarantees that file systems should provide in the face of crashes are under-specified by the POSIX standard, leaving application writers to guess how to achieve data integrity and improve performance. This paper showed that crash-consistency models, in the form of litmus tests and formal specifications, effectively describe the contract between the file system and the application. We presented FERRITE, a toolkit for constructing crash-consistency models by exploring the possible crash behaviors of a file system. While developing models for common file systems, we identified and clarified several widely misunderstood guarantees in the literature.

Crash-consistency models enable tools for writing crash-safe applications. We demonstrated proof-of-concept synthesis and verification tools, and discussed opportunities for improved programming interfaces that build on these models. We hope that both file system developers and application writers can benefit from precise specifications of crash behavior. All of FERRITE's source code is publicly available at: `http://sandcat.cs.washington.edu/ferrite`.

Acknowledgments

We thank Zach Tatlock, our shepherd Joseph Tucek, and the anonymous reviewers for feedback on earlier versions of this paper. This work was supported in part by NSF under grant #1064497, by DARPA under contract FA8750-16-2-0032, and by a gift from the VMware University Research Fund.

References

[1] S. V. Adve and H.-J. Boehm. Memory models: A case for rethinking parallel languages and hardware. *Communications of the ACM*, 53(8):90–101, Aug. 2010.

[2] J. Alglave. A formal hierarchy of weak memory models. *Formal Methods in System Design*, 41(2):178–210, Oct. 2012.

[3] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Fences in weak memory models. In *Proceedings of the 22nd International Conference on Computer Aided Verification (CAV)*, pages 258–272, Edinburgh, UK, July 2010.

[4] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Litmus: Running tests against hardware. In *Proceedings of the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 41–44, Saarbrücken, Germany, Mar.–Apr. 2011.

[5] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 0.90 edition, Mar. 2015.

[6] Austin Group. 0000672: Necessary step(s) to synchronize filename operations on disk, 2013. `http://austingroupbugs.net/view.php?id=672`.

[7] F. Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the 2005 USENIX Annual Technical Conference*, pages 41–46, Anaheim, CA, Apr. 2005.

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

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

[10] N. Boichat. Issue 502898: ext4: Filesystem corruption on panic, June 2015. `https://code.google.com/p/chromium/issues/detail?id=502898`.

[11] J. Bonwick. ZFS: The last word in filesystems, Oct. 2005. `https://blogs.oracle.com/bonwick/entry/zfs_the_last_word_in`.

[12] G. Boudol and G. Petri. Relaxed memory models: An operational approach. In *Proceedings of the 36th ACM Symposium on Principles of Programming Languages (POPL)*, pages 392–403, Savannah, GA, Jan. 2009.

[13] Btrfs. What are the crash guarantees of overwrite-by-rename? `https://btrfs.wiki.kernel.org/index.php/FAQ`.

[14] 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)*, Monterey, CA, Oct. 2015.

[15] V. Chidambaram, T. Sharma, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Consistency without ordering. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST)*, pages 101–116, San Jose, CA, Feb. 2012.

[16] 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.

[17] H. Chu. MDB: A memory-mapped database and backend for OpenLDAP. In *Proceedings of the 3rd International Conference on LDAP*, Heidelberg, Germany, Oct. 2011.

[18] 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.

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

[20] J. Corbet. That massive filesystem thread, Mar. 2009. `https://lwn.net/Articles/326471/`.

[21] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Budapest, Hungary, Mar.–Apr. 2008.

[22] D. R. Engler, M. F. Kaashoek, and J. W. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, pages 251–266, Copper Mountain, CO, Dec. 1995.

[23] M. Flatt and PLT. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Design Inc., 2010. `http://racket-lang.org/`.

[24] C. Frost, M. Mammarella, E. Kohler, A. de los Reyes, S. Hovsepian, A. Matsuoka, and L. Zhang. Generalized file system dependencies. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, pages 307–320, Stevenson, WA, Oct. 2007.

[25] 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.

[26] D. Giampaolo. *Practical File System Design with the BE File System*. Morgan Kaufmann Publishers, 1999.

[27] J. Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, pages 393–481. Springer-Verlag, 1977.

[28] R. Hagmann. Reimplementing the cedar file system using logging and group commit. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP)*, pages 155–162, Austin, TX, Nov. 1987.

[29] D. Hitz, J. Lau, and M. Malcolm. File system design for an NFS file server appliance. In *Proceedings of the Winter 1994 USENIX Technical Conference*, San Francisco, CA, Jan. 1994.

[30] IEEE and The Open Group. The open group base specifications issue 7, 2013.

[31] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*, 2015. rev. 57.

[32] W. K. Josephson, L. A. Bongo, D. Flynn, and K. Li. DFS: A file system for virtualized flash storage. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST)*, pages 1–15, San Jose, CA, Feb. 2010.

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

[34] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on exokernel systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, pages 52–65, Saint-Malo, France, Oct. 1997.

[35] 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.

[36] G. Keller, T. Murray, S. Amani, L. O'Connor, Z. Chen, L. Ryzhyk, G. Klein, and G. Heiser. File systems deserve verification too. In *Proceedings of the 7th Workshop on Programming Languages and Operating Systems*, Farmington, PA, Nov. 2013.

[37] M. Kuperstein, M. Vechev, and E. Yahav. Automatic inference of memory fences. In *Proceedings of 10th International Conference on Formal Methods in Computer-Aided Design*, pages 111–120, Lugano, Switzerland, Oct. 2010.

[38] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 26(9):690–691, Sept. 1979.

[39] C. Lee, D. Sim, J.-Y. Hwang, and S. Cho. F2FS: A new file system for flash storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*, pages 273–286, Santa Clara, CA, Feb. 2015.

[40] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, pages 348–370, Dakar, Senegal, Apr.–May 2010.

[41] Linux kernel. Bug 15910 - zero-length files and performance degradation, 2010. `https://bugzilla.kernel.org/show_bug.cgi?id=15910`.

[42] Linux kernel. Ext4 filesystem, 2015. `https://www.kernel.org/doc/Documentation/filesystems/ext4.txt`.

[43] Linux man-pages. `close` - close a file descriptor, 2013. `http://man7.org/linux/man-pages/man2/close.2.html`.

[44] R. A. Lorie. Physical integrity in a large segmented database. *ACM Transactions on Database Systems*, 2(1):91–104, Mar. 1977.

[45] R. Lortie. more on dconf performance, btrfs and fsync, Dec. 2010. `https://blogs.gnome.org/desrt/2010/12/19/more-on-dconf-performance-btrfs-and-fsync/`.

[46] R. Lortie. ext4 file replace guarantees, June 2013. `http://www.spinics.net/lists/linux-ext4/msg38774.html`.

[47] 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.

[48] S. Mador-Haim, R. Alur, and M. M. K. Martin. Generating litmus tests for contrasting memory consistency models. In *Proceedings of the 22nd International Conference on Computer Aided Verification (CAV)*, pages 273–287, Edinburgh, UK, July 2010.

[49] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *Proceedings of the 32nd ACM Symposium on Principles of Programming Languages (POPL)*, pages 378–391, Long Beach, CA, Jan. 2005.

[50] M. K. McKusick. Journaled soft-updates. In *BSDCan*, Ottawa, Canada, May 2010.

[51] M. K. McKusick and T. J. Kowalski. Fsck: The UNIX file system check program. UNIX System Manager's Manual (SMM), Oct. 1996.

[52] Microsoft. Alternatives to using Transactional NTFS, 2015. `https://msdn.microsoft.com/en-us/library/windows/desktop/bb968806(v=vs.85).aspx`.

[53] C. Min, W.-H. Kang, T. Kim, S.-W. Lee, and Y. I. Eom. Lightweight application-level crash consistency on transactional flash storage. In *Proceedings of the 2015 USENIX Annual Technical Conference*, pages 221–234, Santa Clara, CA, July 2015.

[54] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, 17(1):94–162, Mar. 1992.

[55] Mozilla. Bug 421482 - Firefox 3 uses `fsync` excessively, 2008–2015. `https://bugzilla.mozilla.org/show_bug.cgi?id=421482`.

[56] S. Neumann. Re: fsync in glib/gio, Mar. 2009. `https://mail.gnome.org/archives/gtk-devel-list/2009-March/msg00098.html`.

[57] E. B. Nightingale, K. Veeraraghavan, P. M. Chen, and J. Flinn. Rethink the sync. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–14, Seattle, WA, Nov. 2006.

[58] Open Group. fsync - synchronise changes to a file. The Single UNIX Specification, Version 2, 1997. `http://pubs.opengroup.org/onlinepubs/7908799/xsh/fsync.html`.

[59] S. Park, T. Kelly, and K. Shen. Failure-atomic `msync()`: A simple and efficient mechanism for preserving the integrity of durable data. In *Proceedings of the ACM EuroSys Conference*, pages 225–238, Prague, Czech Republic, Apr. 2013.

[60] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The operating system is the control plane. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–16, Broomfield, CO, Oct. 2014.

[61] T. S. Pillai, V. Chidambaram, J.-Y. Hwang, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Towards efficient, portable application-level consistency. In *Proceedings of the 9th Workshop on Hot Topics in Dependable Systems*, Farmington, PA, Nov. 2013.

[62] 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.

[63] D. E. Porter, O. S. Hofmann, C. J. Rossbach, A. Benn, and E. Witchel. Operating systems transactions. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 161–176, Big Sky, MT, Oct. 2009.

[64] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Model-based failure analysis of journaling file systems. In *Proceedings of the 35th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 802–811, Yokohama, Japan, June–July 2005.

[65] V. Prabhakaran, T. L. Rodeheffer, and L. Zhou. Transactional flash. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 147–160, San Diego, CA, Dec. 2008.

[66] 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)*, Monterey, CA, Oct. 2015.

[67] D. M. Ritchie and K. Thompson. The UNIX time-sharing system. *Communications of the ACM*, 17(7):365–375, July 1974.

[68] O. Rodeh, J. Bacik, and C. Mason. BTRFS: The Linux B-tree filesystem. *ACM Transactions on Storage*, 9(3), Aug. 2013.

[69] 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.

[70] C. Rubio-González, H. S. Gunawi, B. Liblit, R. H. Arpaci-Dusseau, and A. C. Arpaci-Dusseau. Error propagation analysis for file systems. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 270–280, Dublin, Ireland, June 2009.

[71] G. Schellhorn, G. Ernst, J. Pfähler, D. Haneberg, and W. Reif. Development of a verified flash file system. In *Proceedings of the 4th International ABZ Conference*, pages 9–24, Toulouse, France, June 2014.

[72] K. Shen, S. Park, and M. Zhu. Journaling of journal is (almost) free. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST)*, pages 287–293, Santa Clara, CA, Feb. 2014.

[73] A. Solar-Lezama. *Program synthesis by sketching*. PhD thesis, University of California, Berkeley, 2008.

[74] A. Solar-Lezama, L. Tancau, R. Bodik, V. Saraswat, and S. Seshia. Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 404–415, San Jose, CA, Oct. 2006.

[75] D. J. Sorin, M. D. Hill, and D. A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool, 2011.

[76] R. P. Spillane, S. Gaikwad, M. Chinni, E. Zadok, and C. P. Wright. Enabling transactional file access via lightweight kernel extensions. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST)*, pages 29–42, San Francisco, CA, Feb. 2009.

[77] SQLite. Atomic commit in SQLite, 2013. `https://www.sqlite.org/atomiccommit.html`.

[78] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS file system. In *Proceedings of the 1996 USENIX Annual Technical Conference*, San Diego, CA, Jan. 1996.

[79] The Open Group. Technical standard: Extended API set part 2, Oct. 2006.

[80] E. Torlak and R. Bodik. A lightweight symbolic virtual machine for solver-aided host languages. In *Proceedings of the 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 530–541, Edinburgh, UK, June 2014.

[81] E. Torlak and D. Jackson. Kodkod: A relational model finder. In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 632–647, Braga, Portugal, Mar.–Apr. 2007.

[82] E. Torlak, M. Vaziri, and J. Dolby. MemSAT: Checking axiomatic specifications of memory models. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 341–350, Toronto, Canada, June 2010.

[83] L. Tung. Bitcoin developers offer $10,000 virtual bounty to fix mystery Mac bug, Nov. 2013. `http://goo.gl/Ssbj8T`.

[84] Ubuntu. Bug #317781: Ext4 data loss, Jan. 2009. `https://bugs.launchpad.net/ubuntu/+source/linux/+bug/317781`.

[85] R. Verma, A. A. Mendez, S. Park, S. Mannarswamy, T. Kelly, and C. B. M. III. Failure-atomic updates of application data in a Linux file system. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*, pages 203–211, Santa Clara, CA, Feb. 2015.

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

[87] C. P. Wright, R. Spillane, G. Sivathanu, and E. Zadok. Extending ACID semantics to the file system. *ACM Transactions on Storage*, 3(2):1–42, June 2007.

[88] 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.

[89] 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.

[90] Y. Yang, G. Gopalakrishnan, G. Lindstrom, and K. Slind. Nemos: a framework for axiomatic and executable specifications of memory consistency models. In *IPDPS*, 2004.

[91] Y. Yang, G. Gopalakrishnan, and G. Lindstrom. UMM: An operational memory model specification framework with inte-

grated model checking capability. *Concurrency and Computation: Practice & Experience*, 17:465–487, Apr. 2005.

[92] 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.