# SquirrelFS: using the Rust compiler to check file-system crash consistency

Hayley LeBlanc
University of Texas at Austin

Nathan Taylor
University of Texas at Austin

James Bornholt
University of Texas at Austin

Vijay Chidambaram
University of Texas at Austin

## Abstract

This work introduces a new approach to building crash-safe file systems for persistent memory. We exploit the fact that Rust's typestate pattern allows compile-time enforcement of a specific order of operations. We introduce a novel crash-consistency mechanism, *Synchronous Soft Updates*, that boils down crash safety to enforcing ordering among updates to file-system metadata. We employ this approach to build SQUIR-RELFS, a new file system with crash-consistency guarantees that are *checked at compile time*. SQUIRRELFS avoids the need for separate proofs, instead incorporating correctness guarantees into the typestate itself. Compiling SQUIRRELFS only takes tens of seconds; successful compilation indicates crash consistency, while an error provides a starting point for fixing the bug. We evaluate SQUIRRELFS against state of the art file systems such as NOVA and WineFS, and find that SQUIRRELFS achieves similar or better performance on a wide range of benchmarks and applications.

## 1 Introduction

One of the most important properties for file systems is to preserve their integrity and user data in the face of a crash or a power loss [16, 20, 28, 31, 42, 43, 51]. Unfortunately, building crash-consistent file systems is challenging; checking or ensuring crash consistency is even more so [17, 40].

There are two main approaches to building file systems today, as summarized in Table 1. First, we build file systems using low-level languages like C, and we use runtime testing to gain some confidence in the correctness of the systems [36, 37, 41, 46, 47, 59]. Note that this approach is necessarily incomplete; testing can only reveal bugs, not prove their absence. However, this approach allows rapid development, and entire testing ecosystems have sprung up around this basic approach, like the widely-used xfstests [10] and Linux Test Project [5].

A different approach to building file systems is to verify them: we write a high-level specification of correct behavior (including crash behavior) and then prove that the implementation matches the specification [17–19, 29, 53]. This

| Approach | Complete | Dev effort | Time to check |
|---|---|---|---|
| Testing | No | Low | Medium |
| Verification | Yes | High | High |
| This work | Yes | Medium | Low |

Table 1: Comparison of different approaches to ensuring crash consistency in file systems.

approach can prove that the implementation does not have certain classes of bugs; however, it comes at a high cost. For each line of code in the implementation, we may need to write 7–13 lines of proof. Writing and maintaining proofs is time-consuming and requires specialized expertise, constraining rapid development.

In this work, we seek to find a middle ground between these two approaches. We would like to verify some aspects of file systems, but without the burden of having to write and maintain proofs. In particular, we are interested in *crash consistency*, a correctness property that is especially difficult to test for. In order to be crash consistent, systems must ensure that updates become persistent on storage media *in the correct order*; however, hardware or caching layers may reorder updates to improve performance in unanticipated ways. Exposing crash-consistency bugs thus requires one to find and reproduce these low-level orderings, which requires specialized testing software [36, 37, 41, 46, 47, 59]. Our goal is to develop lightweight approaches to statically check for crash-consistency bugs without the overhead of full verification.

We exploit two recent developments to achieve this goal (§2). First, the Rust programming language has a strong type system that supports powerful compile-time safety checks. Our work takes inspiration from Corundum [32], a Rust crate (library) that uses Rust's type system to check low-level PM safety properties. In this work, we observe that Rust's type system can also statically enforce that certain operations are carried out in a given order [27, 38]. Since the root of crash consistency is ordering updates to storage, if we can encode those ordering-based invariants in the type system, the com-
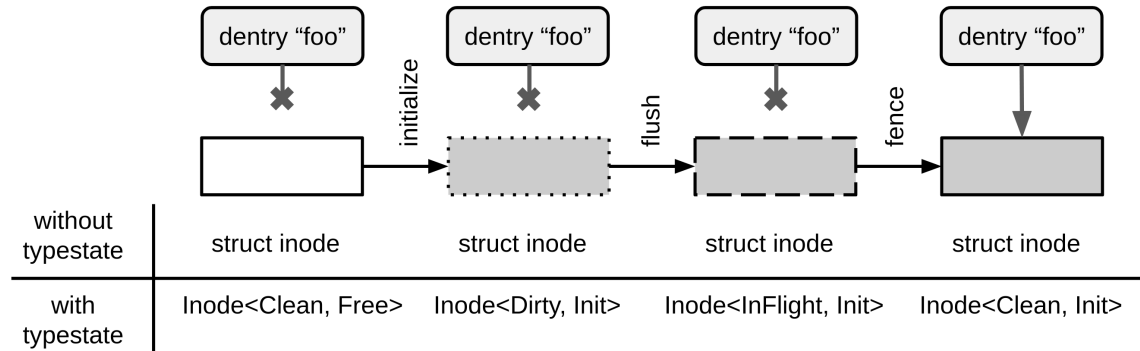
| | | | | |
|---|---|---|---|---|
| | dentry "foo" | dentry "foo" | dentry "foo" | dentry "foo" |
| | ✗ | ✗ | ✗ | |
| | | *initialize* → | *flush* → | *fence* → |
| **without typestate** | struct inode | struct inode | struct inode | struct inode |
| **with typestate** | Inode\<Clean, Free\> | Inode\<Dirty, Init\> | Inode\<InFlight, Init\> | Inode\<Clean, Init\> |

Figure 1: For a soft updates file system to be crash-consistent, directory entries should only point to fully-initialized, durable inodes. In existing file systems, all persistent inodes have the same type, regardless of whether they are durable or have been initialized. With typestate, durability and the inode's contents are reflected in its type.

piler can ensure the invariants hold at compile time.

However, to do so, crash consistency must be derived purely from ordering-based invariants; some mechanisms such as journaling use writes to a log to obtain atomicity, which is harder to encode in the type system. Soft updates achieves crash consistency purely via ordering [43], but the traditional soft updates scheme is complex and hard to implement [13, 25].

We observe that the low latency of persistent memory [56, 59] allows file-system operations to be *synchronous*; all updates to storage media are durable by the time each operation returns [24, 34, 35, 39, 57]. We take advantage of persistent memory's synchronous updates and byte addressability to develop a new mechanism for crash consistency we term *Synchronous Soft Updates*.

Synchronous Soft Updates builds on the classical soft updates mechanism [43], but avoids most of the complexity that prevented the widespread adoption of soft updates [13]. Two of the most complicated aspects of soft updates, dependency structures and cyclic dependency management, arise due to the need to track ordering requirements between block-sized updates across asynchronous operations. Synchronous Soft Updates eliminates these challenges entirely by using fast, fine-grained storage to back synchronous operations.

We ensure that the ordering invariants of Synchronous Soft Updates hold by using the Rust compiler. We take advantage of Rust's support for the *typestate pattern*, an API design pattern where an object's type reflects the operations that have been performed on it [54]. The legal order of operations is encoded in function signatures and enforced by Rust's type-checker. For example, an uninitialized inode has a different type than an initialized one; attempting to use one where the other is expected will result in a compile-time error. Figure 1 illustrates the approach.

We implement Synchronous Soft Updates in a new file system for PM called SQUIRRELFS and use the typestate pattern in Rust to check that update orderings are implemented correctly. SQUIRRELFS provides crash-atomic metadata system calls, including `rename`; on the original soft updates, a crash during rename could result in both the source and destination existing. SQUIRRELFS compiles and typechecks in seconds, whereas running verification on existing storage systems takes minutes or hours. Building SQUIRRELFS required no modifications to the Rust language.

We evaluate SQUIRRELFS by comparing to a number of file systems meant for persistent memory, such as NOVA [57] and WineFS [34] (§5). We use Intel's Optane DC Persistent Memory Module for our comparison, and find that SQUIRRELFS offers comparable or better performance to other PM file systems across a broad range of workloads. The current SQUIRRELFS prototype prioritizes simplicity of update ordering rules over performance in some areas, leading to relatively high mount times and memory utilization; however, these are not fundamental limitations of the design. We also model the design of SQUIRRELFS using the Alloy model-checking language [33] to gain confidence in the correctness of its Synchronous Soft Updates mechanism.

We note that SQUIRRELFS is not fully verified, and thus does not obtain the strong correctness guarantees of verified storage systems like FSCQ [19]. Crash-consistency bugs may still occur in SQUIRRELFS if their root causes are unrelated to ordering, if the ordering rules enforced by the compiler are incorrect, or if trusted code in SQUIRRELFS's implementation or the Rust compiler are buggy. For example, SQUIRRELFS's ordering rules guarantee that inodes are always initialized before they are linked into the file system tree, but they do not guarantee that the contents of the inode are correct. SQUIRRELFS's static checks are also limited by the capabilities of the Rust compiler. For instance, the Rust compiler cannot check properties about variable-sized sets of data structures, as checking such properties is undecidable in general.

SQUIRRELFS offers a useful new point in the spectrum of approaches to building robust storage systems; it provides weaker guarantees than verified systems, but comes at a lower

cost. As such, we hope that it proves useful for developers of storage systems that require strong guarantees, good performance, and rapid development.

In summary, this work makes the following contributions:

- Statically-checked crash consistency, an approach where high-level properties are encoded into the type system and checked at compile time (§3)

- The Synchronous Soft Updates crash-consistency mechanism for persistent-memory file systems (§3.1)

- The SQUIRRELFS prototype, along with a discussion of lessons learned during its development (§4).

SQUIRRELFS and its Alloy model are publicly available at https://github.com/utsaslab/squirrelfs.

## 2 Background and Motivation

### 2.1 Crash Consistency

A file system is *crash consistent* if it can recover to a consistent state after a power loss or a crash [16,20,51]. A consistent file system is one where all the metadata is in sync; for example, two files cannot (mistakenly) claim the same data block. Files present before the crash must exist post-crash, and the data in files must remain valid.

**Crash-consistency mechanisms**. Crash consistency is generally achieved using mechanisms such as journaling [28,48], copy-on-write [1,31,42], or soft updates [43]. The root of crash consistency is correctly *ordering* writes to storage [20]; for example, a data block must be initialized before a file points to it. Soft updates achieves crash consistency by carefully ordering in-place updates to storage such that all possible crash states are consistent [43]. To enforce ordering, soft updates must track updates across asynchronous operations and resolve cyclic dependencies when they arise. Though soft updates is used in FreeBSD [44], it has not been widely adopted due to its high complexity.

**Ensuring crash consistency**. Ensuring that a given file system achieves crash consistency is challenging. There are two main approaches. The first approach is testing, in which possible crash states of a file system are explored and checked for consistency. Obtaining these crash states requires support from tools like eXplode [59], CrashMonkey [46], Hydra [37], Chipmunk [41], or Vinter [36]. While such testing tools can find many bugs, they cannot prove overall correctness or the absence of crash-consistency bugs.

The second approach is to build verified file systems. A developer writes a high-level specification of correctness and a lower-level implementation, and proves that the implementation satisfies the specification. This approach is stronger than testing in that it can prove strong correctness properties and verify that there are no bugs. However, it comes at a high cost: the developer has to write 7–13 lines of proof for every line of code. For example, BilbyFS [12] required 13k lines of proof for 1k lines of implementation code; VeriBetrKV [29] used 45K lines of proof for 6k lines of implementation. Another verified file system, FSCQ [19], has interleaved proof and implementation code that is $10\times$ the size of the most similar unverified system.

This heavy proof burden constrains development in a number of ways. First, building a verified system requires proof-writing expertise, which restricts the set of developers who can work on it. Second, proofs must be written in tandem with the code that they verify, which extends development time. Finally, making changes to the system requires corresponding changes to the proofs, making maintenance slow and preventing rapid updates.

Corundum [32] is a Rust crate for PM systems that, like SQUIRRELFS, uses the Rust type system to enforce certain low-level PM safety properties at compile time. For example, Corundum ensures that every update to PM occurs in a logged transaction, and prevents the storage of pointers to volatile memory in durable structures. SQUIRRELFS was inspired by Corundum and aims to enforce higher-level properties like file-system crash consistency with Rust.

### 2.2 The opportunity: Rust and PM

We observe an opportunity to ensure file-system crash consistency in a cheap manner.

First, we note that the Rust programming language can **statically enforce a specific order on operations** via its support for the *typestate pattern* [9,27]. Briefly, the typestate pattern enables an object's runtime state to be encoded in its type [54]. This state can be checked at compile time via typechecking, ensuring that a given operation can only occur on a specific type. Typestate information is stored in zero-sized types that incur no runtime overhead.

For example, one consistency rule enforced by soft updates is that a directory entry should never point to an uninitialized inode. Listing 1 shows how typestate is used to enforce this rule. To create a new file, we first obtain a free directory entry and inode. Initially, both objects have typestate `Free`. Then, we initialize the directory entry, transitioning its type to `Dentry<Init>`. The listing then has a bug in which the directory entry's inode number is set by `commit_dentry()` before the inode is initialized, breaking the consistency rule. The Rust compiler catches this bug because the inode's current typestate `Free` does not match the typestate `Init` expected by the function.

Since soft updates is entirely built on ordering updates to file-system objects, we can translate the required partial order into a set of types and use Rust's type checking to enforce the order. Thus, the invariants we want to maintain are translated into something the type system and compiler can enforce. We note that we are able to do this with an *unmodified* Rust compiler; the new types introduced are no different to the

```
1  fn new_file() {
2      // Dentry<Free>
3      let d = Dentry::get_free_dentry();
4      // Inode<Free>
5      let i = Inode::get_free_inode();
6      // Dentry<Init>
7      let d = d.set_name("foo");
8      let d = d.commit_dentry(i);
9                      ^ expected 'Inode<Init>',
10                       found 'Inode<Free>'
11 }
```

Listing 1: The listing shows the typecheck process throwing an error when an uninitialized inode is passed to a function that expects an initialized inode.

compiler from existing types in the codebase.

However, implementing soft updates correctly remains challenging even with typestate support. With soft updates, file-system updates are applied to the page cache in DRAM, and then later written to storage in the right order. Determining the right order requires tracking complex dependencies across asynchronous operations. When a single file-system metadata object (such as an inode or a bitmap) is updated multiple times, it can lead to cyclic dependencies.

This leads to our second observation: persistent memory (PM) file systems support synchronous operations thanks to the low latency of the storage media [56, 58]. These file systems write updates directly to storage without first caching them in DRAM [24, 34, 35, 39, 57]. A **synchronous implementation** of soft updates for persistent memory **eliminates** the complexities of asynchronous dependency management, greatly simplifying the mechanism and allowing the relevant invariants to be encoded in Rust's type system.

## 3  SquirrelFS

We now present the design and implementation of SQUIR-RELFS, a novel file system that uses the unmodified Rust compiler to check its crash consistency. If the compilation is successful, it indicates that the ordering-based invariants hold throughout the file system: in other words, the checking is complete. If compilation fails, the error reported by Rust is useful in figuring out which operations are out of order. Compilation takes only seconds, offering quick feedback to developers.

SQUIRRELFS is built on two key ideas:

- A novel crash-consistency mechanism, Synchronous Soft Updates, that achieves crash consistency purely via ordering file-system operations (§3.1)

- Using the Rust typestate pattern to encode ordering invariants into the Rust type system (§3.2)

It is important to note that we are not modifying the Rust compiler in any way. To the Rust compiler, it is no different from type-checking any other code base; we are merely using the type checking to ensure that crash consistency holds in the file system.

We now describe the key ideas in more detail.

### 3.1  Synchronous Soft Updates

We develop *Synchronous Soft Updates* (SSU), a novel crash-consistency mechanism. SSU is based on the traditional soft updates approach, but differs in two key aspects. First, soft updates was designed for asynchronous settings, but all operations are synchronous in SSU. Second, soft updates does not provide atomic rename; a crash during a rename of `src` to `dst` can result in both being present after a crash. SSU fixes this flaw; renames are atomic, and a crash during rename will result in either `src` or `dst` after recovery.

We now discuss why we developed SSU, its key aspects, and how renames are atomic in SSU.

**Why a new mechanism?** To go with our overall approach of encoding ordering-based invariants into the Rust type system, we needed a mechanism that achieves crash consistency purely via ordering file-system updates. This rules out mechanisms such as journaling and copy-on-write that use writes to a log or an extra copy to obtain atomicity. Soft updates [43] obtains crash consistency by enforcing ordering on in-place persistent updates to file-system objects; thus, it was a good match. However, traditional soft updates suffered from two problems that we needed to tackle. The first challenge was that soft updates had significant complexity arising from needing to track dependencies between asynchronous file-system operations; the presence of cyclic dependencies also requires complex roll-back and roll-forward logic. The second challenge is that soft updates does not provide atomic operations, particularly rename; atomic rename is a crucial primitive for a number of POSIX applications [50]. Thus, we need to modify soft updates to tackle both its high complexity and lack of atomic operations.

**Synchronous operations**. We observe that the root of complexity in soft updates (such as cyclic dependencies and structures for tracking dependencies) is *asynchrony*. A *synchronous* implementation of soft updates neatly avoids these complexities. All updates would be made durable by the end of each system call, which would eliminate the need to track cross-operation dependencies. Cyclic dependencies would no longer arise because there are no pending updates that can conflict with each other. The SoupFS [23] soft updates file system for persistent memory eliminated cyclic dependencies using fine-grained updates, but still required asynchronous dependency tracking. A synchronous implementation is necessary to overcome both sources of complexity.

A synchronous version of soft updates was not feasible

until now, as running this on magnetic hard drives or even solid state drives would be prohibitively slow. However, synchronous soft updates is a good match for persistent memory (PM) due to its low latency; system calls in many existing PM file systems are already synchronous [24, 34, 35, 57].

Similar to traditional soft updates, SSU maintains crash consistency by enforcing ordering among updates to file-system objects. SSU implements the original soft updates rules [26]:

1. Never point to a structure before it has been initialized;

2. Never re-use a resource before nullifying all previous pointers to it;

3. Never reset the old pointer to a live resource before the new pointer has been set.

These rules are significantly easier to enforce in a synchronous setting, as there is no need to track dependencies across asynchronous operations. Like soft updates, SSU focuses on the integrity of file system metadata and cannot guarantee that operations on file data are atomic. SSU could be combined with journaling or copy-on-write to obtain stronger data guarantees.

**Atomic rename in SSU**. SSU ensures renames are atomic by cleaning up file-system state after a crash. In traditional soft updates, if there is a rename from src to dst, it is impossible to tell after a crash whether src or dst should be removed. To resolve this, SSU adds an extra field, called the *rename pointer*, to directory entries in order to persistently save enough information to complete the rename operation after a crash. The rename pointer in the destination directory entry points to the physical location of the source directory entry. The rename pointer allows the file system to follow soft updates rule 3 (never reset the old pointer before the new one has been set) while also retaining the ability to distinguish between src and dst after a crash.

Note that this is similar to what journaling-based file systems do; they write a log entry specifying src and dst so that the right clean-up action can be performed. In SSU, the information in this log entry is distributed over the source and destination inodes; taken together, they provide enough information to the file system.

Figure 2 illustrates the process. Step ❶ shows an example system state prior to the rename operation. In ❷, dst's rename pointer (dotted line) is set to src. dst is invalid, and src is still valid. In ❸, we make dst valid; this also logically invalidates src. This is an atomic point; after this step, the file system will always complete the rename operation. If the file system crashes prior to this step, the rename pointer is cleared on recovery. In ❹, we physically mark src as invalid. In ❺, the rename pointer is cleared, and in ❻ src is fully deallocated. Each step either modifies metadata that is invisible to the user (e.g., deallocating an orphaned directory entry) or
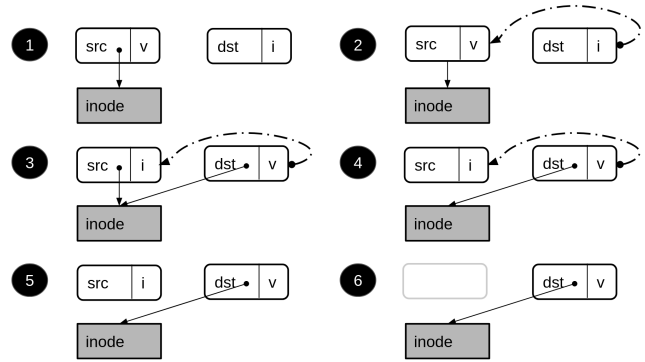


Figure 2: The figure shows the steps in atomic soft updates rename. The dotted lines represent rename pointers and the solid lines represent inode pointers. src and dst are directory entries. The labels "v" and "i" indicate whether a directory entry is valid or invalid.

atomically modifies a single 8-byte value. All modifications must be durable before proceeding to the next step.

A question that arises is how the file system finds src and dst. This is an example of how SSU is tailored for PM file systems. In PM file systems, it is common for the file system to scan persistent objects to construct indexes in DRAM; we add the rename-recovery procedure into this scan. Thus, when building volatile indexes after a crash, the file system also looks for and completes any partially completed rename operations.

## 3.2   Using Rust to enforce ordering

Rust's typestate pattern can be used to ensure that *a set of functions are always called in certain partial order*. A total order is not necessary, as many operations involve independent updates that can be safely reordered. As we discussed previously (§2), an object's typestate is encoded in generic type parameters in its definition, and the partial order is encoded in the function signatures of its associated functions.

We encode two states (as different type parameters) in the types of persistent objects:

- *Persistence* typestate is a representation of whether an object's most recent update(s) have been made durable. We use three persistence typestates: Dirty, InFlight, and Clean.

- *Operational* typestate represents the operations that have been performed on an object and is used to determine what operations can happen next.

Persistence and operational typestate are separate to capture the fact that most storage devices do not synchronously flush updates. For example, in persistent memory, updates go to

```
1  impl Inode<Clean, Free> {
2      fn init_inode(self, ino: u64, ...)
3          -> Inode<Dirty, Init> {...}
4  }
5  impl Dentry<Clean, Alloc> {
6      fn commit_dentry(
7          self,
8          inode: Inode<Clean, Init>
9      ) -> Dentry<Dirty, Committed> {...}
10 }
11 impl<S> Inode<Dirty, S> {
12     fn flush(self) ->
13         Inode<InFlight, S> {...}
14 }
15 impl<S> Inode<InFlight, S> {
16     fn fence(self) ->
17         Inode<Clean, S> {...}
18 }
```

Listing 2: Pseudocode implementations of file system objects with persistence and operational typestate. Typestate arguments are shown in bold.

the CPU caches first, and must be explicitly flushed to the persistent media.

Listing 2 shows implementations of several methods of persistent Inode and Dentry types with persistence and operational typestate as generic type parameters. The methods flush and fence invoke a cache line write back and store fence respectively and are generic with respect to operational typestate. These methods must be used to ensure updates are persistent before continuing; for example, commit_dentry() requires an Inode<Clean, Init> to ensure the inode's initialization cannot be transparently reordered with the directory entry updates.

This formulation of persistence typestate has several performance benefits. First, because the flush and fence methods can only be called on an object whose typestate indicates it is not yet persistent, typechecking will prevent redundant persistence operations (thereby improving performance). Second, developers can wait to flush updates until it is strictly necessary and can write additional transitions to enable multiple updates to share a single fence.

**Why Rust?** In order to obtain useful compiler-checked guarantees from the typestate pattern, each object must have exactly one typestate [54]. Thus, languages with unrestricted aliasing (e.g., C) cannot support the typestate pattern, as different aliases for the same value can have different types. Rust supports the pattern via its ownership type system, which ensures that each value has exactly one owner (and thus exactly one type).
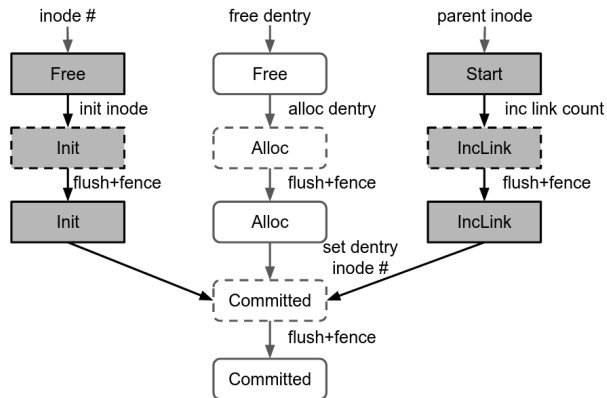


Figure 3: The figure shows the persistent updates and corresponding dependencies made during mkdir. Inodes are dark gray and directory entries are white. Each object is labeled with its operational typestate and its outline indicates whether it is clean (solid) or dirty (dotted).

## 3.3 Example: mkdir

We use mkdir to illustrate the typestates and dependency rules used in SSU. To be crash consistent, an SSU implementation of mkdir must ensure (1) that a structure never points to an uninitialized resource, and (2) that each inode's link count is greater than or equal to its actual number of links. Both rules prevent dangling links in the event of a crash.

Figure 3 illustrates the dependencies in a mkdir operation. During mkdir, three file-system objects are modified: an inode for the new directory, a directory entry for the new directory, and the inode of the parent directory. Note that all three can be modified at the same time in a concurrent fashion, and can share a single store fence at the end (not shown). SQUIRRELFS uses volatile allocation structures, so they are not persisted during mkdir.

The system first finds the parent inode and obtains a free directory entry in one of the parent's pages as well as a free inode. The inode is then initialized (i.e., setting its inode number, link count, timestamps), the directory entry's name is set, and the parent inode's link count is incremented.

Next, we commit the directory entry by setting its inode number. This makes the directory entry valid and connects the inode to the file system tree. Directory entry commit is dependent on inode and directory entry initialization and parent link increment. Committing the directory entry before initializing the inode can result in a directory entry pointing to a garbage inode; committing before incrementing the parent's link count can lead to dangling links.

## 3.4 Implementation

We implemented SQUIRRELFS in Rust with 7500 LOC. It uses bindings from the Rust for Linux project [8] to con-
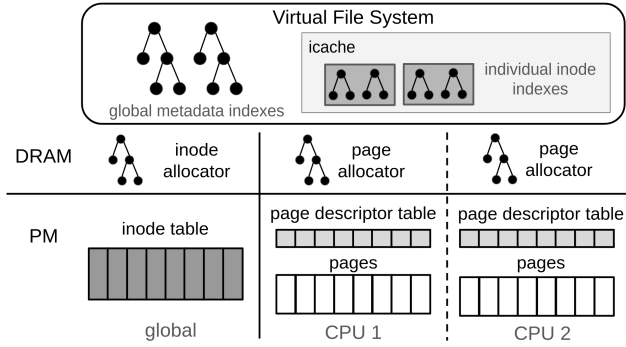
Figure 4: The figure shows the main components of SQUIR-RELFS. Each CPU has its own pool of pages and private page allocator. The inode allocator is shared between all CPUs. Volatile indexes are stored in VFS data structures.

nect to the Linux Virtual File System (VFS) layer. Figure 4 shows SQUIRRELFS's architecture. We also built a model of SQUIRRELFS in the model-checking language Alloy [33] to check its design for crash consistency issues. We describe our experience developing SQUIRRELFS in §4.

**Overview**. The design of SQUIRRELFS combines aspects of FreeBSD's FFS [44] and PM file systems such as NOVA [57] and WineFS [35]. Like FFS, it has a simple on-storage layout, and uses soft updates. Like other PM file systems, SQUIR-RELFS uses volatile index structures that are built when the file system is mounted.

SQUIRRELFS's design was primarily influenced by two factors. First, we wanted to keep dependencies as simple as possible and avoid nested persistent structures that are difficult to represent in typestate. Second, we assume the x86 PM persistence model in which only aligned updates of 8 bytes (or smaller) are crash atomic [24]. Under the x86 model, persistent addresses can be accessed via regular memory stores, but the corresponding cache line must be flushed before updates become persistent; a memory barrier like a store fence must also be invoked to correctly order stores [52]. Durable structures may also be updated via cache-bypassing non-temporal store instructions, which still require a store fence for persistence ordering. This programming model influences the structure of persistent objects and restricts the set of legal orderings.

All system calls in SQUIRRELFS are synchronous, meaning that updates to durable structures made by each system call are durable by the time the system call returns. As such, `fsync` is a no-op in SQUIRRELFS. Metadata-related operations are also crash-atomic. Data-related operations are not atomic in the current SQUIRRELFS prototype, which matches the default behavior of other PM file systems like NOVA [57]. These operations could be made atomic by using copy-on-write to update file contents.

**Persistent layout**. SQUIRRELFS uses a simple layout to reduce the complexity of update dependencies. SQUIRRELFS splits the storage device into four sections: the superblock, the inode table, the page descriptor table, and the data pages. The inode table is an array of all of the inodes in the system. SQUIRRELFS reserves enough space for approximately one inode for every 16KB of data (four pages), the same ratio used by the Linux Ext4 file system.

The page descriptor array contains page metadata. Rather than having inodes point to the pages they own, each page descriptor contains a backpointer to its owner (similar to NoFS [21]) and stores its own metadata (e.g., its offset in the file). This approach simplifies dependency rules for updates involving page allocation and deallocation. All remaining space after the page descriptor table is used for data and/or directory pages.

**Volatile structures**. SQUIRRELFS's persistent layout simplifies typestate and update dependency rules, but it is not amenable to fast lookups. Therefore, SQUIRRELFS uses indexes in DRAM to speed up lookup and read operations. Each inode in the VFS inode cache has a private index for the resources it owns; index data for uncached nodes is stored in the VFS superblock.

Like many other PM file systems, SQUIRRELFS uses volatile allocators: allocation information is not stored in a persistent manner, but rather rebuilt each time the file system is mounted. It uses a per-CPU page allocator and a single shared inode allocator (which could be converted to a per-CPU allocator to improve scalability). The allocators use free lists backed by kernel RB-trees.

SQUIRRELFS's indexes and allocators are rebuilt by scanning the file system when SQUIRRELFS is mounted. An inode, directory entry, or page descriptor is considered allocated if *any* of its bytes are non-zero. Directory entries and page descriptors are only valid if their inode numbers are set; inodes are valid only if they are reachable from the root. Thus, updates that allocate new structures and set non-inode metadata fields need not be crash-atomic.

**Synchronous Soft Updates.** SQUIRRELFS uses an implementation of SSU for crash consistency. As shown in Figure 3, operations that involve creation of new objects must first durably allocate and initialize resources before linking them into the file system (setting the directory entry's inode in the example) to enforce rule 1 (never point to a structure before it has been initialized). Deallocation proceeds in reverse; links are first cleared, then the object itself is deallocated by zeroing all of its bytes. SQUIRRELFS enforces rule 2 of soft updates (never re-use a resource before nullifying all previous pointers to it) by treating durable objects that are not completely zeroed out as allocated and by ensuring via typestate that pointers to the object are cleared before the object can be zeroed.

**Typestate transition functions.** SQUIRRELFS updates the

typestate of objects via *typestate transition functions*. These functions take ownership of the original object, modify it, and return it to the caller with the new typestate. These functions are defined only on certain typestates to ensure they are called in a safe order. For example, the typestate transition function `commit_dentry()`, shown in Listing 2, is only defined for directory entries with type `Dentry<Clean, Alloc>`, and also takes ownership of an inode of type `Inode<Clean, Init>`. Calling `commit_dentry()` out of order – e.g., on a directory entry that has not yet been persistently allocated – is a potential crash-consistency bug and results in a compiler error.

**Concurrency.** SQUIRRELFS supports concurrent file-system operations. It relies on VFS-level locking on durable resources like inodes. This locking, together with Rust's type system, ensures that each resource has only one owner – and only one type – at any time, enabling strong typestate-based compile-time checking. SQUIRRELFS uses internal locks to protect its allocators and indexes.

**Building a model with Alloy**. While the typestate pattern can enforce a given operation order, it cannot verify that this order is crash consistent. To gain more confidence that SQUIRRELFS's design is crash consistent, we built a model of SQUIRRELFS in the Alloy model checking language [33].

Alloy provides a language for specifying transition systems and a model checker to explore possible sequences of states (traces) of these systems. Alloy's implementation is based on a logic of relations; each system is composed of a set of constraints that define a set of structures and the relations between them, and the model checker uses constraint solving to find traces.

In SQUIRRELFS, there is roughly a one-to-one mapping between typestate transitions in the Rust implementation and the next-state predicates in the Alloy model. Each next-state predicate specifies the states in which the transition may occur and the changes it makes to the model's state.

The model includes next-state predicates for typestate transitions and persistent updates. It also includes transitions that model crashes and recovery, which let us check SQUIRRELFS's design for crash-consistency bugs.

Each persistent structure in SQUIRRELFS is represented by a corresponding structure, also called a signature, in Alloy. The model also includes a `Volatile` signature that is used to model volatile aspects of the file system like its indexes. Each typestate is represented by a signature, and instances of persistent structures are mapped to their current typestate. Each file system operation is also represented by a signature, and relations map system calls to instances of persistent objects they are operating on as well as other volatile state (e.g., the locks held by that operation). We use this to model concurrent file-system operations.

## 3.5 Limitations of the approach

It is important to note that the typestate-based approach used in SQUIRRELFS is not as powerful as full verification. Fully-verified systems, such as the FSCQ file system [19], use theorem provers that can prove a wide variety of complex properties. For example, a developer could prove, if required, that the system only uses even-numbered inodes for files.

In contrast, our typestate-based approach can only check *ordering-based* invariants. Our approach could be used to verify that functions are called in a specific order; for example, our approach can ensure that a file is not linked into the file-system tree before it is allocated. However, it does not verify the implementation of each function that is called.

Thus, full verification is significantly more powerful and general, but it pays a cost in terms of complexity and development time. Our approach is more targeted and ordering-based, but allows quick feedback and incremental development.

We believe this approach is a valuable addition to the repertoire of tools we have for building correct file systems. This approach should be used alongside runtime testing and model-checking approaches.

## 3.6 Relevance beyond PM

While we have designed SQUIRRELFS for persistent memory, SQUIRRELFS would be relevant for any storage technology with byte-addressability and low latency. The Compute Express Link [2] standard will support attached memory devices, including PM, via the Type 3 (CXL.mem) protocol. These CXL-attached PM devices will have the same interface and persistence semantics as current NVDIMMs, though performance will be lower [14].

SQUIRRELFS, and SSU file systems in general, could be used on CXL-attached memory. As SQUIRRELFS's mount performance and memory footprint are tied to the size of the device, they may worsen with significantly larger-capacity devices. Further work will be required to optimize file systems based on our approach for such devices.

## 4 Experience developing SQUIRRELFS

We now describe our experience with designing, developing, and testing SQUIRRELFS. We also discuss the challenges we faced during this process.

### 4.1 Development process

**Designing SQUIRRELFS.** Our initial design closely followed that of BSD FFS [43], but most aspects eventually diverged due to differences between storage hardware and typestate considerations. We found that some data structures and crash-consistency properties were better suited for use with the

typestate pattern than others. For example, we chose SQUIR-RELFS's backpointer-based page management approach because it simplifies update dependency rules when allocating or deallocating pages. With backpointers, these operations involve a constant number of persistent updates and involve no additional durable structures. In contrast, tree or log-based approaches need extra persistent metadata and may require additional updates to balance the tree or free log space, both of which complicate dependencies and typestate management.

An important design decision we had to make was how granular typestate would be. One option was to use specific typestates to represent each fine-grained operation; e.g., have one typestate for initializing an inode's link count, another for setting its flags, etc. Another was to make each typestate more general, with transition functions potentially performing multiple persistent updates. More general typestates may sacrifice some bug-finding power, but they make the system easier to understand and develop. In SQUIRRELFS, we aimed to strike a balance by representing only operations that require a specific ordering with typestate. For example, when initializing an inode in SQUIRRELFS, the order in which the values of most fields are set is not relevant to crash consistency, as the contents of the inode are not visible until it is linked into the file system tree. Therefore, SQUIRRELFS uses only a single typestate (`Init`) to represent inode initialization, and another (`Committed`) to indicate when it has been added to the tree.

**Parallel model and system implementation.** We developed the Alloy model alongside SQUIRRELFS. This created a useful feedback loop in which the model supported the Rust implementation, and questions or changes to the implementation could be quickly reflected and checked in the model. We used an incremental development process, incorporating feedback from the Rust compiler and the model immediately as we implemented the system. Many transitions in the model could be translated directly into Rust typestate transitions, making the model a valuable guide for implementing file system operations. When we made mistakes translating the model into Rust, typestate checking quickly caught these issues.

Alloy also includes a graphical user interface for visualizing traces of operations on the model. This was useful for both investigating invariant violations and seeing the set of transitions that occur in a given file system operation, which could be translated directly into system call handler implementations. It also demonstrated locations where multiple updates could safely share a single store fence, which helped us avoid redundant fences.

## 4.2  Finding bugs

While developing SQUIRRELFS, we used a combination of typestate checking, model checking in Alloy, and dynamic testing to find bugs.

**Typestate checking.** Typestate checking in the implementa-

tion was successful at quickly catching both missing persistence primitives and higher-level ordering bugs; we provide an example of each.

- *Missing persistence primitives.* Our initial implementation of `write` was missing flush and fence calls after setting the backpointer of a newly-allocated page. This bug was immediately highlighted as an error by the compiler. Had this bug made it into the implementation, a crash could cause a file to have a size larger than the number of pages associated with it, causing errors when trying to read the file.

- *Incorrect ordering.* Our initial `rename` implementation mistakenly decremented an inode's link count before clearing the corresponding directory entry. A crash could result in a link count that is lower than the true number of links, leading to a dangling link if the inode is subsequently deleted.

Although we did not specifically check execution paths without crashes, the crash-consistency invariants encoded in typestate were general enough to detect some bugs in this code. For example, the compiler caught a bug where pages were not fully deallocated during `unlink`, which did not require a crash to manifest. Typestate-related compiler errors were relatively uncommon overall, since using the model as a guide for implementation helped us get ordering right early. However, it provided a crucial safety net to prevent subtle bugs when we did make mistakes.

**Model checking with Alloy.** The Alloy model found several high-level issues in SQUIRRELFS's design that would have otherwise been difficult to detect and time-consuming to fix, including the following examples.

- We initially believed that crash recovery would not be needed other than to fix space leaks. Alloy found an instance of the model where a crash during rename followed by deallocation of the destination directory entry could cause an invalid directory entry to reappear. Fixing this required the addition of recovery transitions.

- Early designs for SQUIRRELFS stored . and .. directory entries durably. We discovered via model checking that our original dependency rules for handling these directory entries during more complicated operations like `rename` were not correct. Ultimately, we decided to not store these entries, since they can be constructed at runtime using indexed information.

**Testing.** Neither the typestate pattern nor the Alloy model eliminated the need to test SQUIRRELFS. Our primary goal was to check crash-consistency, and we did not check any invariants that only impact regular, non-crash execution. We used handwritten tests and the xfstests suite [10] to test these unchecked parts of the code.

All bugs found through testing were in parts of SQUIRRELFS that were not checked by typestate or directly modeled in Alloy. Most bugs were related to updating volatile indexes or VFS inodes, e.g., failing to remove a deallocated object from an index or setting the wrong value in the VFS inode. There were also bugs in the implementations of typestate transitions, which are not themselves verified; for example, the transition that wrote new file data to a page did not always calculate the offset for non-aligned writes correctly. Implementing bug fixes was quick since we did not need to modify the typestate-restricted interface to objects and there were no proofs to update.

## 4.3 Challenges encountered

**Challenges with typestate.** It is easier to write typestate-checked code than it is to write verified code, but this comes at the cost of less powerful compile-time checking. For example, checking universally-quantified formulas (e.g., all pages in a file are allocated) is undecidable, and unlike verification-aware languages, the Rust compiler has no heuristics to attempt to solve them. As a result, we cannot ensure invariants such as "all objects in a set are in a certain typestate"; specifically, we can't encode this in typestate because the number of objects in the set is not known at compile time.

This became a problem when implementing file-system operations like `unlink`, where we would like to e.g., check that the backpointers of all pages belonging to the file are cleared before deallocating the inode. Such a check ensures that the system always follows soft updates rule 2 (never reuse a resource before nullifying all previous pointers to it); by clearing all of the page backpointers before deleting the inode, we ensure that none remain when the inode is eventually reused. However, it is impossible to check this property on arbitrary sets of pages if each page has its own typestate. We experimented with several workarounds, including forcing `write` operations to update no more than one page at a time (which was prohibitively slow and did not solve the problem for `unlink`), and storing typestate in page structures at runtime and manually adding assertions (which also impacted performance and lost the benefit of static checking). Ultimately, we decided to use a single piece of typestate to represent *ranges* of pages (e.g., all of the pages in a file or a contiguous subsection). Each typestate operation on such a range performs the operation on all pages in the range. This moved some logic into the typestate transitions, making the transition functions themselves more complicated but making page-management logic more centralized and easier to manually audit.

**Challenges with Alloy.** As SQUIRRELFS grew in complexity, it became harder to maintain the model and get useful feedback quickly. The model checker uses a SAT solver to check invariants, and the formulas representing a large model can take days or weeks to solve. We checked that traces with multiple concurrent operations were crash consistent, which increased the size of the problem further. To get faster feedback, we built a custom utility to run multiple independent instances of the model checker in parallel and split larger predicates into smaller, more concrete sub-checks.

It could also be difficult to determine whether a reported failure was a false positive. A particular challenge was dealing with *frame conditions*, predicates that specify what should not change in a given transition. Alloy is free to arbitrarily change any state that the current transition does not explicitly mention, so frame conditions are crucial to constrain the model to real traces. This behavior helps Alloy find corner-case bugs, but it also leads to false positives. To overcome this challenge, we built a syntax-based checker that parses the model using Alloy's API and checks that each transition explicitly mentions all mutable structures in the model. The current version of the checker cannot detect all issues, but it detected many missing conditions that would have otherwise taken hours to catch via model checking.

## 4.4 Typestate beyond SQUIRRELFS

**Costs and benefits of typestate.** We do not have equivalent verified or unverified systems to compare with SQUIRRELFS in terms of development and debugging effort; however, in the authors' experience, designing and implementing SQUIRRELFS required more effort than a typical unverified system, but far less effort than a verified storage system.[1] We believe that debugging SQUIRRELFS was faster and easier than debugging an equivalent unverified system, as following the typestate-enforced ordering rules made it easier to implement the system correctly in the first place and reduced the number of bugs overall.

Using the typestate pattern for crash consistency represents a useful new point in the tradeoff space between runtime testing and full verification. While it comes at the cost of additional development effort compared to unverified systems to determine correct ordering rules and does not gain the same correctness guarantees as verified systems, it does eliminate an entire class of crash consistency bugs that are otherwise difficult to find and fix [37, 41, 46]. Furthermore, as the pattern builds ordering rules directly into a system's implementation, the rules will stay up to date and continue to prevent crash-consistency bugs as the system is developed further [30, 49].

**Broader applicablity.** As the typestate pattern is a general approach for statically checking the order of updates to data structures, it is useful in a broad variety of contexts, several of which are described below.

---

[1]For example, author LeBlanc recently worked on a durable log implemented in a verification-aware programming language, which took about 3 months of full-time work.

- Volatile data structures: SQUIRRELFS does not use type-state to manage updates to volatile data structures, but prior work on typestate verification has focused entirely on such use cases [11, 54].

- Other types of storage systems: The typestate pattern could be used to enforce ordering invariants on durable updates in other types of storage systems (e.g., key-value stores) with different crash-consistency mechanisms. We note that crash-consistency mechanisms like journaling and copy-on-write do not achieve consistency entirely through ordering and would require auxiliary techniques to check properties like atomicity.

- Durable layout: SQUIRRELFS's on-storage layout is tailored to reduce the number of durable updates per file-system operation and to simplify ordering rules. Other layouts could also be used in typestate-checked storage systems, although the complexity of the ordering rules would increase.

- Asynchrony: The typestate pattern is compatible with asynchronous systems, although the ordering rules to enforce are much more complicated in such systems, as updates from different operations may be interleaved.

## 5 Evaluation

We seek to answer the following questions in our evaluation of SQUIRRELFS:

1. What is the latency of different file-system operations on SQUIRRELFS? (§5.2)

2. How does SQUIRRELFS perform on macrobenchmarks? (§5.3)

3. How does SQUIRRELFS perform on real applications? (§5.4)

4. How long does SQUIRRELFS take to mount and recover from crashes? (§5.5)

5. What compilation, memory, and CPU overheads does SQUIRRELFS incur? (§5.6)

6. Is SQUIRRELFS correct? (§5.7)

### 5.1 Experimental setup

We evaluate SQUIRRELFS on a two-socket, 32 core machine with 128GB of memory and one 128GB Intel Optane DC Persistent Memory Module. The evaluation machine runs Debian Bookworm and Linux 6.3.

We compare SQUIRRELFS against ext4-DAX [3], NOVA [57], and WineFS [34]. We configure all three systems to provide metadata consistency but not data consistency to match SQUIRRELFS's guarantees. We cannot compare SQUIRRELFS to SoupFS [23], the only other soft updates PM file system, as it is not open source. Due to time constraints, we were unable to compare against the recent ArckFS [60] file system. We hope to do so in the future. All reported results are the average of multiple trials. The red errors bars in Figure 5 indicate the minimum and maximum values recorded over all trials.

### 5.2 Microbenchmarks

We compare each system's latency by testing several file system operations: appending and reading 1KB and 16KB to a file, file creation, directory creation, renaming a directory, and unlinking a 16KB file. None of the tests call `fsync`.

The average latency over 10 trials of the tested operations are shown in Figure 5(a). The lowest latency file system in each test is either WineFS or SQUIRRELFS. Ext4-DAX has the highest latency on many operations because it interacts with the Linux kernel block layer for tasks like block allocation, which incurs additional software overhead. It achieves similar performance to the other systems on operations that do not go through the block layer (e.g., unlink). NOVA has higher latency on `mkdir` and `rename` than WineFS and SquirrelFS because operations that update multiple inodes require journaling in NOVA.

### 5.3 Macrobenchmarks

We evaluate SQUIRRELFS on the Filebench [4] storage benchmark suite. We run four workloads from the suite – fileserver, varmail, webserver, and webproxy – in their default configurations. Fileserver performs mostly writes with some whole file reads; varmail is half appends and half reads; webproxy appends to each file and reads from it several times; and webserver reads and occasionally appends to a log file. Figure 5(b) shows the average throughput in kops/sec for each file system on each workload. SQUIRRELFS achieves slightly better throughput than the next fastest system on fileserver and varmail (8% and 13% better, respectively) and within 10% of the fastest system on both webserver and webproxy. Fileserver and varmail perform many small appends, which SQUIRRELFS performs well on due to its lack of journaling. Webserver and webproxy are more read-heavy, which all systems perform roughly equally on. Ext4-DAX does not go through the block layer on reads and it benefits from data contiguity awareness, making its performance similar or better than the other systems on these workloads.

### 5.4 Applications

**YCSB on RocksDB.** We evaluate the four systems on RocksDB [7] using YCSB workloads [22]. We run all workloads on a 25GB database with 25M records, 25M operations, and 8 threads. All workloads are run using standard workload configurations and the default settings of YCSB, which uses

**(a) System call latency**

**(b) Filebench**

**(c) YCSB workloads on RocksDB**
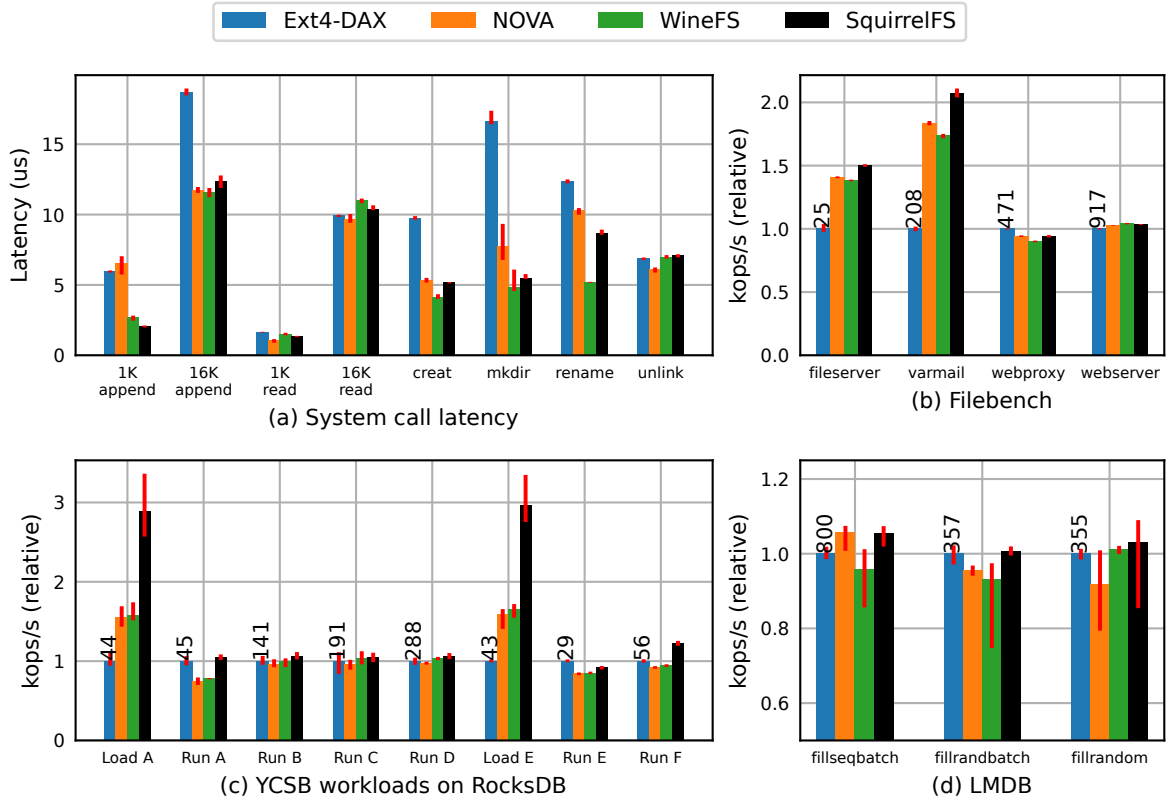
**(d) LMDB**

Figure 5: This figure shows the performance of the evaluated file systems on different benchmarks and applications. (a) shows absolute latency of different file system operations; (b), (c), and (d) show the relative throughput in kops/s of each system relative to Ext4-DAX on filebench, YCSB on RocksDB, and LMDB respectively.

system calls for all operations. Figure 5(c) shows throughput in kops/second relative to Ext4-DAX on each tested workload.

SQUIRRELFS outperforms the other systems on Loads A and E, which are 100% small inserts. As seen on the other benchmarks, SQUIRRELFS performs particularly well on small appends due to its lack of journaling or logging. Writes that require page allocation are particularly expensive in the other systems, as journaling/logging the new metadata incurs an additional 2-3us in NOVA and WineFS and 3-4us in Ext4-DAX. Ext4-DAX and NOVA both also journal or log metadata on every append, spending roughly 30% of each non-allocating call (approx 1-1.5us) managing journals/logs.

All file systems are within 10% of Ext4-DAX's throughput on Runs B, C, and D. All of these workloads are at least 95% small (4KB) reads, which all four systems achieve similar performance on.

SQUIRRELFS achieves the best throughput on Runs A and F, which are 50% reads and 50% updates (Run A) or read-modify-write operations (Run F). Ext4-DAX, NOVA, and WineFS all incur logging/journaling on these workloads; Ext4-DAX outperforms NOVA and WineFS because it has less journaling overhead for in-place updates and is more

aware of data contiguity on reads.

Ext4-DAX achieves the best performance on Run E, which is 95% range scans and 5% inserts. Ext4-DAX's contiguity-awareness and better fragmentation-prevention mechanisms help it outperform the other systems on larger read operations.

**LMDB.** We also run LMDB [6], a memory-mapped database, using db_bench's fillseqbatch, fillrandbatch, and fillrand workloads. Each experiment uses 100M keys on an empty file system. Figure 5(d) shows the throughput in kops/sec for each file system on each workload. Each file system has throughput with 12% of the other systems. Most updates are done to memory-mapped files, so differences in the performance of system calls and metadata management designs have a reduced impact.

**Git.** We also evaluate the performance of SQUIRRELFS by performing git checkout of major Linux kernel versions. The time to check out a given version in each file system is within 8% of the other systems.

## 5.5 Mount time

SQUIRRELFS takes longer to mount than other PM file sys-

| | System state | Mount time (s) |
|---|---|---|
| Normal mount | mkfs | 5.80 |
| | Empty | 5.51 |
| | Full | 30.50 |
| Recovery mount | Empty | 5.76 |
| | Full | 55.50 |

Table 2: Time in seconds to mount SQUIRRELFS file system images in different states. Times in the recovery mount column come from mounting a cleanly-unmounted file system that runs a recovery scan in addition to normal rebuild scans.

| System | LOC | Compile time (s) |
|---|---|---|
| Ext4 | 45K | 38 |
| NOVA | 16K | 20 |
| WineFS | 9K | 13 |
| SQUIRRELFS | 7.5K | 10 |

Table 3: Time to compile different PM file systems as loadable kernel modules. Ext4's line count includes interleaved DAX and non-DAX code.

tems because it must rebuild volatile indices for the entire file system. Table 2 shows how long it takes to mount SQUIRRELFS on a 128GB PM device with different contents. The ≈ 5.5 seconds it takes to initialize or mount an empty system is the overhead of zeroing or scanning the metadata tables and creating volatile allocators. We also measure the time to mount a system with 100% data and inode utilization. Most of this time is spent allocating space for and managing the volatile indexes and allocators.

If SQUIRRELFS detects that it was not unmounted cleanly, it constructs additional structures to keep track of orphaned objects and the true link count of each inode. It fills in these structures during the regular rebuild scan and uses them to free orphans and correct link counts at the end of the mount process. SQUIRRELFS also checks each directory entry for non-null rename pointers and either rolls back or completes any interrupted renames. Table 2 reports the time it takes SQUIRRELFS to perform recovery scans on a cleanly-unmounted device. Mounting with recovery takes longer than a standard mount because the file system must construct orphan-tracking structures and do an extra iteration over all directories to check for rename pointers in addition to building the volatile indices and allocators.

SQUIRRELFS's mount time could be improved by parallelizing some of its rebuild and recovery logic. For example, the inode and page descriptor table scans are completely independent and could be done in parallel. The file system tree rebuild logic could also be distributed across multiple threads.

## 5.6 Resource usage

**Compilation.** SQUIRRELFS takes approximately 10 seconds to compile on our test machine, including typestate checking. This compares well to fully-verified systems; FSCQ [19] takes about 11 hours to verify, and VeriBetrKV [29] takes 1.8 hours (10 minutes when parallelized).

SQUIRRELFS also compiles faster than the other tested systems on the test machine. Table 3 shows the size of each system in lines of code and how long it takes to compile. SQUIRRELFS's more complicated typechecking does not no-

ticeably impact its compilation time.

**Memory.** SQUIRRELFS maintains indexes for fast lookups of files and directory entries. Each regular file has an index mapping its inode number (8 bytes) to each of its pages and their offsets (16 bytes total). Thus, the index entries for a 1MB file use about 4KB of memory. Each directory has a similar inode to page index (without offsets), plus a mapping from directory entry names to metadata like their location on PM and inode number. The current maximum name length is 110 bytes (which makes directory entries 128 bytes) and SQUIRRELFS does not currently hash or compress names. Therefore, each directory entry takes up approximately 250 bytes in the index.

**CPU.** SQUIRRELFS does not start new threads in any of its operations. We leave the use of more threads for operations like freeing pages, running crash recovery, etc. to future work.

## 5.7 Correctness

**Model checking.** We check that a correctness invariant always holds in all traces of our Alloy model. We bound traces to include two operations (which may be concurrent), 10 persistent objects, and up to 30 steps. The invariant includes both sanity checks on the model as well as file system consistency checks. The sanity checks ensure, for example, that objects will never end up with conflicting typestates. The consistency checks ensure that 1) objects always have a legal link count, 2) there are no pointers to uninitialized objects, 3) freed objects do not contain pointers to other objects, and 4) there are no cycles of rename pointers and directory entries are pointed to by at most one rename pointer.

**Testing.** We test SQUIRRELFS using a set of handwritten tests and the xfstests [10] test suite. SQUIRRELFS currently passes all supported tests (67) from xfstests' `generic` test suite. The rest of the tests use system calls or arguments that are currently not supported by SQUIRRELFS.

**Crash consistency.** We used Chipmunk [41] to test SQUIRRELFS for crash-consistency bugs. We modified Chipmunk's test generators to remove several system calls that SQUIRRELFS does not currently support but otherwise ran its full suite of systematically-generated tests and fuzzed the sys-

tem for approximately 24 hours. Chipmunk did not find any ordering-related crash-consistency bugs in SQUIRRELFS, providing evidence that typestate-checked SSU is an effective mechanism for preventing such bugs. Chipmunk did find four crash consistency bugs in unchecked parts of SQUIRRELFS code, three in its rebuilding of volatile data structures and one in the body of typestate transitions in which a cache line flush was issued to the wrong address. As these are not caused by incorrect update ordering, the typestate pattern did not catch them at compile time. We found that using the typestate pattern in SQUIRRELFS made locating and fixing these bugs faster and easier, as we could focus on the specific regions of code that are unchecked and are thus more likely to have bugs.

## 5.8 Summary

SQUIRRELFS provides comparable performance to other PM file systems, while providing strong guarantees about its crash consistency. Due to the innovative use of typestate checking, we were able to implement SSU and gain confidence in its correctness. SQUIRRELFS gains an advantage over other file systems in write-dominated workloads, since soft updates avoids writing to a log or to a second copy of the data. The design of SQUIRRELFS trades off good common-case performance for slightly longer mount times compared to other file systems; we believe this is acceptable since crashes are rare. SQUIRRELFS compiles at the same rate as other PM file systems, despite the strong type checking.

## 6 Related work

**Rust for PM**. SQUIRRELFS was inspired by Corundum [32]. Corundum builds data structures whose low-level properties are checked using Rust's type system. For example, Corundum ensures that there are no pointers to volatile memory stored in persistent memory, and that persistent state is only updated in transactions. It focuses on lower-level persistent memory programming errors and cannot prevent higher-level logical bugs. Corundum also requires *all* updates to PM to be in transactions, which is overly restrictive for many systems. In contrast to Corundum, SQUIRRELFS checks high-level file-system crash-consistency properties using type-checking without placing constraints on how the file system is used.

**Soft updates for PM**. Two PM file systems use soft updates for crash consistency: SoupFS [23] and ArckFS [60]. Unlike SQUIRRELFS, SoupFS is asynchronous and uses background threads to flush updates. It uses byte-addressable updates to eliminate cyclic dependencies. ArckFS is a user-space PM file system built on the Trio architecture that uses synchronous, soft-updates-esque updates for simple operations (e.g., creating a file) and undo journaling in more complicated cases. Unlike ArckFS, SQUIRRELFS uses only synchronous soft

updates for its crash consistency; the novel way in which SQUIRRELFS implements atomic rename (without journaling or copy-on-write) further differentiates it from ArckFS. Both SoupFS and ArckFS are written in C, and do not use Rust's type system to check their crash consistency.

**Storage systems in Rust**. Bento [45] is a framework for building in-kernel file systems in Rust. The corresponding file system from the Bento project, BentoFS, was designed for block devices. Bento does not utilize the type system of Rust to check file-system properties.

ShardStore [15] is a Rust key-value store used in Amazon S3 that uses an asynchronous soft-updates-inspired crash-consistency mechanism. The rules for when something should be written to storage in ShardStore were checked with Dep-Synth [55], a tool for synthesizing soft updates dependency rules. Unlike ShardStore, SQUIRRELFS uses a synchronous version of soft updates, and provides higher-level primitives like atomic rename; ShardStore does not utilize the type system to perform higher-level checks.

## 7 Conclusion

This paper presents a new methodology for crash-consistent file system development. We propose the use of the typestate pattern in Rust to statically check crash-consistency invariants with low proof burden. We also introduce a novel crash-consistency mechanism, synchronous soft updates, that is well-suited to enforcement with the typestate pattern and that eliminates many challenges associated with the original soft updates technique. We develop SQUIRRELFS, a new file system for persistent memory that uses statically-checked synchronous soft updates for crash consistency. SQUIRRELFS achieves comparable or better performance than other PM file systems and required no language modifications or verification expertise to build. SQUIRRELFS, its Alloy model, and our Alloy utilities are available at https://github.com/utsaslab/squirrelfs.

## References

[1] BTRFS documentation. https://btrfs.readthedocs.io/en/latest/.

[2] Compute Express Link (CXL) specification. https://www.computeexpresslink.org/download-the-specification.

[3] Direct Access for files. https://www.kernel.org/doc/Documentation/filesystems/dax.txt.

[4] Filebench. https://sourceforge.net/projects/filebench/.

[5] Linux test project. https://linux-test-project.github.io/.

[6] LMDB. http://www.lmdb.tech.

[7] Rocksdb. https://rocksdb.org/.

[8] Rust for linux. https://rust-for-linux.com/.

[9] Typestate programming. https://docs.rust-embedded.org/book/static-guarantees/typestate-programming.html.

[10] xfstests. https://github.com/kdave/xfstests.

[11] Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. Typestate-oriented programming. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, page 1015–1022, New York, NY, USA, 2009. Association for Computing Machinery.

[12] Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O'Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, Joseph Tuong, Gabriele Keller, Toby Murray, Gerwin Klein, and Gernot Heiser. Cogent: Verifying high-assurance file system implementations. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, page 175–188, New York, NY, USA, 2016. Association for Computing Machinery.

[13] Valerie Aurora. Soft updates, hard problems. https://lwn.net/Articles/339337/, July 2009.

[14] Piotr Balcer. Exploring the Software Ecosystem for Compute Express Link (CXL) Memory. https://pmem.io/blog/2023/05/exploring-the-software-ecosystem-for-compute-express-link-cxl-memory/, May 2023.

[15] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, Jacob Van Geffen, and Andrew Warfield. Using lightweight formal methods to validate a key-value storage node in Amazon S3. In *ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP)*, pages 836–850, October 2021.

[16] James Bornholt, Antoine Kaufmann, Jialin Li, Arvind Krishnamurthy, Emina Torlak, and Xi Wang. Specifying and checking file system crash-consistency models. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS*, pages 83–98, Atlanta, GA, USA, April 2016.

[17] Tej Chajed, Joseph Tassarotti, Mark Theng, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying the DaisyNFS concurrent and crash-safe file system with sequential reasoning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 447–463, Carlsbad, CA, July 2022. USENIX Association.

[18] Haogang Chen, Tej Chajed, Alex Konradi, Stephanie Wang, Atalay İleri, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying a high-performance crash-safe file system using a tree specification. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 270–286, New York, NY, USA, 2017. Association for Computing Machinery.

[19] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Using crash hoare logic for certifying the fscq file system. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 18–37, New York, NY, USA, 2015. Association for Computing Machinery.

[20] Vijay Chidambaram. *Orderless and Eventually Durable File Systems*. PhD thesis, University of Wisconsin, Madison, Aug 2015.

[21] Vijay Chidambaram, Tushar Sharma, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Consistency Without Ordering. In *Proceedings of the 10th Conference on File and Storage Technologies (FAST '12)*, pages 101–116, San Jose, California, February 2012.

[22] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.

[23] Mingkai Dong and Haibo Chen. Soft updates made simple and fast on non-volatile memory. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 719–731, Santa Clara, CA, July 2017. USENIX Association.

[24] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran,

and Jeff Jackson. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, New York, NY, USA, 2014. Association for Computing Machinery.

[25] Christopher Frost, Mike Mammarella, Eddie Kohler, Andrew de los Reyes, Shant Hovsepian, Andrew Matsuoka, and Lei Zhang. Generalized file system dependencies. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, page 307–320, New York, NY, USA, 2007. Association for Computing Machinery.

[26] Gregory R. Ganger and Yale N. Patt. Metadata update performance in file systems. In *First Symposium on Operating Systems Design and Implementation (OSDI 94)*, Monterey, CA, November 1994. USENIX Association.

[27] Jon Gjenset. *Rust for Rustaceans*. No Starch Press, 2022.

[28] Robert B. Hagmann. Reimplementing the cedar file system using logging and group commit. In Les Belady, editor, *Proceedings of the Eleventh ACM Symposium on Operating System Principles, SOSP 1987, Stouffer Austin Hotel, Austin, Texas, USA, November 8-11, 1987*, pages 155–162. ACM, 1987.

[29] Travis Hance, Andrea Lattuada, Chris Hawblitzel, Jon Howell, Rob Johnson, and Bryan Parno. Storage systems are distributed systems (so verify them that way!). In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI'20, USA, 2020. USENIX Association.

[30] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jay Lorch, Bryan Parno, Justine Stephenson, Srinath Setty, and Brian Zill. Ironfleet: Proving practical distributed systems correct. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*. ACM - Association for Computing Machinery, October 2015.

[31] Dave Hitz, James Lau, and Michael A. Malcolm. File system design for an NFS file server appliance. In *USENIX Winter 1994 Technical Conference, San Francisco, California, USA, January 17-21, 1994, Conference Proceedings*, pages 235–246. USENIX Association, 1994.

[32] Morteza Hoseinzadeh and Steven Swanson. Corundum: Statically-enforced persistent memory safety. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 429–442, New York, NY, USA, 2021. Association for Computing Machinery.

[33] Daniel Jackson. *Software Abstractions*. The MIT Press, 2016.

[34] Rohan Kadekodi, Saurabh Kadekodi, Soujanya Ponnapalli, Harshad Shirwadkar, Gregory R. Ganger, Aasheesh Kolli, and Vijay Chidambaram. Winefs: A hugepage-aware file system for persistent memory that ages gracefully. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 804–818, New York, NY, USA, 2021. Association for Computing Machinery.

[35] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. Splitfs: Reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 494–508, New York, NY, USA, 2019. Association for Computing Machinery.

[36] Samuel Kalbfleisch, Lukas Werling, and Frank Bellosa. Vinter: Automatic Non-Volatile memory crash consistency testing for full systems. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 933–950, Carlsbad, CA, July 2022. USENIX Association.

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

[38] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. No Starch Press, USA, 2018.

[39] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A cross media file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 460–477, New York, NY, USA, 2017. Association for Computing Machinery.

[40] Ubuntu Bugs LaunchPad. Bug #317781: Ext4 Data Loss. https://bugs.launchpad.net/ubuntu/+source/linux/+bug/317781?comments=all.

[41] Hayley LeBlanc, Shankara Pailoor, Om Saran K R E, Isil Dillig, James Bornholt, and Vijay Chidambaram. Chipmunk: Investigating crash-consistency in persistent-memory file systems. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, page 718–733, New York, NY, USA, 2023. Association for Computing Machinery.

[42] R. Lorie. Physical Integrity in a Large Segmented Database. *ACM Transactions on Databases*, 2(1):91–104, 1977.

[43] Marshall Kirk McKusick and Gregory R. Ganger. Soft updates: A technique for eliminating most synchronous writes in the fast filesystem. In *1999 USENIX Annual Technical Conference (USENIX ATC 99)*, Monterey, CA, June 1999. USENIX Association.

[44] Marshall Kirk McKusick, George Neville-Neil, and Robert N.M. Watson. *The Design and Implementation of the FreeBSD Operating System*. Addison-Wesley Professional, 2nd edition, 2014.

[45] Samantha Miller, Kaiyuan Zhang, Mengqi Chen, Ryan Jennings, Ang Chen, Danyang Zhuo, and Thomas Anderson. High velocity kernel file systems with bento. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 65–79. USENIX Association, February 2021.

[46] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. CrashMonkey and ACE: Systematically testing file-system crash consistency. *ACM Trans. Storage*, 15(2), apr 2019.

[47] Ian Neal, Ben Reeves, Ben Stoler, Andrew Quinn, Youngjin Kwon, Simon Peter, and Baris Kasikci. AGAMOTTO: How persistent is your persistent memory application? In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1047–1064. USENIX Association, November 2020.

[48] Roger M. Needham, David K. Gifford, and Mike Schroeder. The cedar file system. *Communications of the ACM*, March 1988.

[49] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How amazon web services uses formal methods. *Communications of the ACM*, 2015.

[50] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. All file systems are not created equal: On the complexity of crafting crash-consistent applications. In Jason Flinn and Hank Levy, editors, *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, pages 433–448. USENIX Association, 2014.

[51] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Crash consistency. *Commun. ACM*, 58(10):46–51, 2015.

[52] Andy Rudoff. Persistent memory programming. *;login:*, (42):34–40, 2017.

[53] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. Push-Button verification of file systems via crash refinement. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 1–16, Savannah, GA, November 2016. USENIX Association.

[54] Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, SE-12(1):157–171, 1986.

[55] Jacob Van Geffen, Xi Wang, Emina Torlak, and James Bornholt. Synthesis-Aided Crash Consistency for Storage Systems. In Karim Ali and Guido Salvaneschi, editors, *37th European Conference on Object-Oriented Programming (ECOOP 2023)*, volume 263 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 35:1–35:26, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[56] Jian Xu, Juno Kim, Amir Saman Memaripour, and Steven Swanson. Finding and fixing performance pathologies in persistent memory software stacks. In Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck, editors, *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, pages 427–439. ACM, 2019.

[57] Jian Xu and Steven Swanson. NOVA: A log-structured file system for hybrid Volatile/Non-volatile main memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 323–338, Santa Clara, CA, February 2016. USENIX Association.

[58] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. An empirical guide to the behavior and use of scalable persistent memory. In Sam H. Noh and Brent Welch, editors, *18th USENIX Conference on File and Storage Technologies, FAST 2020, Santa Clara, CA, USA, February 24-27, 2020*, pages 169–182. USENIX Association, 2020.

[59] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using model checking to find serious file system errors. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, OSDI '04, page 273–287, USA, 2004. USENIX Association.

[60] Diyu Zhou, Vojtech Aschenbrenner, Tao Lyu, Jian Zhang, Sudarsun Kannan, and Sanidhya Kashyap. Enabling high-performance and secure userspace nvm file systems with the trio architecture. In *Proceedings of the 29th Symposium on Operating Systems Principles*,