Using Concurrent Relational Logic with Helpers for Verifying the AtomFS File System

Mo Zou[†‡], Haoran Ding[†‡], Dong Du[†‡], Ming Fu[§], Ronghui Gu[◇], Haibo Chen[†‡*]

† Shanghai Key Laboratory of Scalable Computing and Systems, Shanghai Jiao Tong University

‡ Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University

§ Huawei Technologies Co. Ltd

◇ Columbia University

Abstract

Concurrent file systems are pervasive but hard to correctly implement and formally verify due to nondeterministic interleavings. This paper presents AtomFS, the first formally-verified, fine-grained, concurrent file system, which provides linearizable interfaces to applications. The standard way to prove linearizability requires modeling linearization point of each operation—the moment when its effect becomes visible atomically to other threads. We observe that *path inter-dependency*, where one operation (like rename) breaks the path integrity of other operations, makes the linearization point external and thus poses a significant challenge to prove linearizability.

To overcome the above challenge, this paper presents Concurrent Relational Logic with Helpers (CRL-H), a framework for building verified concurrent file systems. CRL-H is made powerful through two key contributions: (1) extending prior approaches using fixed linearization points with a *helper* mechanism where one operation of the thread can *logically* help other threads linearize their operations; (2) combining relational specifications and rely/guarantee conditions for relational and compositional reasoning. We have successfully applied CRL-H to verify the linearizability of AtomFS directly on C code. All the proofs are mechanized in Coq. Evaluations show that AtomFS speeds up file system workloads by utilizing fine-grained, multicore concurrency.

ACM Reference Format:

Mo Zou, Haoran Ding, Dong Du, Ming Fu, Ronghui Gu, Haibo Chen. 2019. Using Concurrent Relational Logic with Helpers for Verifying the AtomFS File System. In *ACM SIGOPS 27th Symposium on Operating Systems Principles (SOSP '19), October 27–30, 2019, Huntsville, ON, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3341301.3359644

1 Introduction

Applications rely on file systems to store their data, but even carefully-written file systems can have bugs, especially concurrency bugs [9]. Formal methods are the only known way to guarantee that a system is free of programming errors [46]. Recent efforts (e.g., FSCQ [16], COGENT [5], Yggdrasil [66], CIO-FSCQ [11]) have demonstrated the feasibility of verifying file systems, but none of them verify concurrent file systems that run on multicore hardware.

This paper presents AtomFS, the first verified concurrent file system with fine-grained concurrency. AtomFS leverages fine-grained locking (i.e., per-inode lock) but can still provide linearizable (or atomic[1]) interfaces, which are helpful to applications for two reasons. First, existing applications rely on some interfaces implemented atomically by file systems. For example, 10 of 11 applications (e.g., databases, key-value stores) expect atomicity of file system updates [63]. Second, the linearizability of file system interfaces helps to reason about the correctness of concurrent applications [12].

Proving the linearizability of a concurrent file system raises three major challenges. First, to prove linearizability, the most intuitive approach is to find an instant between the invocation and the return of the implementation at which the effect of the operation takes place. The instant is known as the *linearization point* (LP) of the operation. However, it is difficult to apply this idea to verify the linearizability of file systems. Consider two interleaved operations rename(/a, /e) and mkdir(/a/b/c) (Figure 1). Here, the fixed LP can only be located in the critical section between path *lookup* and *return* as the inode modification happens in the section. Suppose the rename completes earlier than mkdir that has already traversed through /a and both of them succeed. As rename takes effect earlier, the sequential history generated through the temporal order of fixed LPs

[*] Mo Zou, Haorang Ding and Dong Du are supported by SOSP 2019 student scholarships from the ACM Special Interest Group in Operating Systems.

[1] "atomic" represents that the operations appear to take effect at a single discrete point in time. We will formally define atomicity and discuss its relation with linearizability in §2

SOSP '19, October 27–30, 2019, Huntsville, ON, Canada

© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-6873-5/19/10…$15.00

https://doi.org/10.1145/3341301.3359644

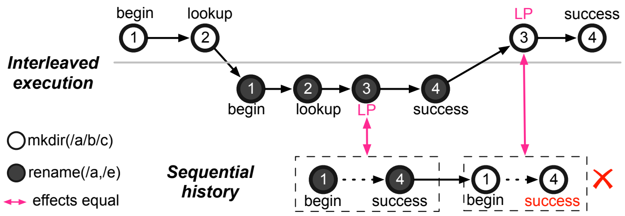


Figure 1. Fixed LP approach fails in the verification of linearizability. For the legal, interleaved execution, the sequential history generated by following temporal order of fixed LPs is (rename(/a, /e), success) → (mkdir(/a/b/c), success), which is illegal as the mkdir should return failure.

is (rename(/a, /e), success) → [2](mkdir(/a/b/c), success), which is illegal under sequential execution. To obtain sequential, legal history, the rename should help mkdir commit its effect before it takes effect itself, which causes the LP of mkdir external (i.e., not located in its own code). We call this phenomenon *path inter-dependency*, which widely exists in concurrent file systems running on Linux, FreeBSD, and xv6. These practical file systems use fine-grained locks to achieve scalability on multicores but lead to external LPs, which make it tough to prove the linearizability of file systems.

Second, a verification framework for concurrent file systems is desired but still missing. There has been some prior work that provides compositional reasoning for concurrent systems. For example, CSPEC [12] uses movers to reduce interleavings that developers must consider. CertiKOS [34] eases the burden of verifying concurrent OS kernels by organizing the system into layers. However, the concurrent objects these frameworks verify do not have external LPs, and it is unclear how to extend them to verify a concurrent file system with external LPs. Though there has been a theoretical effort [48] investigating how to verify linearizability with external LPs, it relies on pen-and-paper proofs for key theorems and does not consider the unique feature of *path inter-dependency* exhibited by concurrent file systems.

Third, it is unknown how to design a linearizable, fine-grained, concurrent file system, and what invariants (or properties) are necessary to achieve linearizability in rigorous proofs. It takes non-trivial efforts to capture all the invariants and precisely specify these invariants. Proving these invariants to hold at any time needs enormous work, especially under arbitrary concurrent interference. In contrast, prior file system verification assumes atomic operations and sequential execution of file systems.

To tackle these challenges, we first propose a helper mechanism to overcome challenges of external LPs. Specifically, rename may serve as the helper to logically help other threads linearize their operations if rename is to break their path integrity. Our helper mechanism extends prior work [48] that supports reasoning about helping in lock-free programming with *ghost state* and *linearize-before relation*, so that we can handle file system-specific challenges and verify the linearizability of file system interfaces.

Second, this paper presents Concurrent Relational Logic with Helpers (CRL-H) for verifying concurrent file systems. Compared with existing frameworks, CRL-H is distinguished by (1) combining relational[3] specifications and rely/guarantee conditions for relational and compositional reasoning; (2) implementing the helper mechanism to deal with external LPs without breaking the existing features of thread-local verification; and (3) mechanizing all the proofs in the Coq proof assistant [24] to guarantee the correctness of our verification procedure.

Furthermore, we successfully build AtomFS and apply CRL-H to verify the linearizability of AtomFS. The design of AtomFS follows the *non-bypassable criterion* to avoid operation bypasses and leverages lock-coupling [38], which is a known method for fine-grained concurrency synchronization. We capture several global invariants of AtomFS, which are essential for the proofs. In addition, CRL-H models a subset of the C programming language, which allows us to build AtomFS directly using C with low performance loss.

Contributions. In summary, this paper makes the following contributions:

- The identification of the *path inter-dependency* phenomenon and the resulting challenges of external LPs (§3).
- The Concurrent Relational Logic with Helpers (CRL-H), which allows programmers to specify and verify the linearizability of concurrent file systems (§4).
- The first formally-verified, fine-grained, concurrent file system, called AtomFS, where the linearizability proofs of AtomFS are mechanized in Coq. AtomFS currently does not consider crashes (§5 and §6).
- An evaluation showing that AtomFS can speed up unmodified applications by utilizing multicore concurrency, and a report of our development experience (§7 and §8).

2 Related Work

Concurrency bugs in file systems. Concurrency bugs are the second most common, which account for about 20% of all the bugs, in Linux file systems [53]. Atomicity violations and deadlock dominate among the concurrency bugs, which may result in serious consequence. Several approaches [43, 56, 77] are proposed to detect concurrency bugs for file systems. Although effective in practice, these approaches can not promise to detect all the bugs in the file systems.

Formal verification of file systems. In recent years, there has been significant progress on formal verification of file systems [5, 11, 14, 16, 41, 66]. FSCQ projects contribute a line

[2] We use → to informally represent orders between threads/operations.

[3] The terminology "relational" follows from [48] and specifically denotes the relationship between abstract-concrete state pairs.

of useful work [11, 14, 16, 41]. FSCQ, DFSCQ, and SFSCQ are all crash-safe, sequential file systems, verified by extending traditional Hoare Logic with a crash condition. CIO-FSCQ reuses the proofs of FSCQ and provides I/O concurrency. However, all these file systems cannot utilize the benefits of multicore concurrency.

Concurrent file system specification. Ntzik et al. [59, 60] propose a specification of concurrent POSIX file systems and reason about several client examples based on the specification. Modeling path traversals is a major challenge in the specification. Each directory lookup can be implemented atomically with the protection of a fine-grained lock. However, it is hard to capture the concurrent behavior of the overall traversal. As POSIX is silent on this matter, they choose to model the file system operations as sequential and parallel combinations of atomic actions, which exposes unnecessary details to the client reasoning. We observe the helping pattern among file system operations, and it allows us to verify the linearizability of file system interfaces.

Verified concurrent systems. There are several concurrent systems [12, 13, 15, 17, 35, 36, 47, 75, 76] which have been formally verified. CCAL [35] has been used to verify CertiKOS's concurrent scheduling queue [34] and MCS lock implementation [45]. Armada [13] verifies a concurrent crash-safe mail server and has a notion of recovery helping to justify the crash recovery procedure. Although similar in concept, recovery helping is used to logically fulfill the operation that has passed its LP prior to a crash and is not for handling external LPs. In summary, these verified systems do not have external LPs and their verification approaches have no notion of concurrency helping to address the challenge.

Atomicity and linearizability. Linearizability [39] is a correctness condition for concurrent objects, e.g., concurrent file systems. In this work, we formally define the atomicity of an object as follows. For each operation of an object, assume there is an atomic specification (e.g., group all statements and execute atomically). For all possible clients, if every observable event trace generated from invoking the implementation can also be produced by invoking the corresponding atomic specification (i.e., the implementation has the same effect as an atomic specification), then the object is atomic. This property is also formalized as *contextual refinement*. Several researchers have proved that contextual refinement is equivalent to linearizability when the specification is atomic [22, 30, 48]. Therefore, atomicity defined in this work is equivalent to linearizability, and we may use them interchangeably.

Verifying linearizability. Verifying linearizability of concurrent data structures [18, 23, 26, 31, 44, 48, 70, 71] has been extensively studied, including the most challenging ones with external linearization points. A large class of lock-free algorithms (e.g., elimination backoff stack [37]) let each thread maintain a descriptor recording all the information required to fulfill its intended operation. When a thread A detects conflicts with another thread B, A may access B's descriptor and fulfill B's intended behavior. So the LP of B resides in the code of A. Previous work [31, 48, 70] has formalized helping to support reasoning about external LPs in these algorithms. However, different features in file systems (more details in §3) make the helping pattern in previous work not directly applicable to file systems. The helper mechanism of CRL-H extends prior work [48] with some domain-specific notions, such as linearize-before relation and roll-back mechanism, which enable CRL-H to verify concurrent file systems.

3 Proving File System Atomicity with Helpers

File system atomicity requires the fine-grained implementation of a file system operation to have the same effect as an atomic specification. For a sequential file system, atomicity is one of the inherent characteristics, while concurrent file systems with fine-grained locking impose complex interleaving that impedes verifying the atomicity of file system operations. In this section, we first introduce how to prove linearizability through an intuitive approach—fixed *linearization point*s, which are statically located in the implementation. Then we define the *path inter-dependency* phenomenon, which leads to external LPs and frustrates the fixed LP approach. After that, we present the challenges of applying an existing effort—helping to handle external LPs in file systems and propose a general approach called *helpers* to address the unique helping pattern in file systems. We end this section with a discussion about how to prove helpers correct.

3.1 Proving Linearizability Using Fixed LPs

Linearizability ensures that each execution history of concurrent objects is consistent with a sequential, legal history. It allows overlapping operations executed concurrently to take effect in any order but requires preserving the real-time order of non-overlapping operations. Simple Hoare Logic applies well in sequential verification by specifying and verifying the code with given pre- and post-conditions, but it is unable to verify the correctness of concurrent file systems, which requires to specify and verify atomicity under concurrent settings. Hence, there have been numerous approaches proposed for verifying linearizability using a variety of techniques [20, 21, 42, 49, 52, 58, 62].

Among these approaches, an intuitive one is to establish forward simulation relations [54] between concrete object implementation and corresponding atomic specification, which can derive atomicity by showing the concrete object implementation contextually refines its corresponding atomic specification. We adopt a similar technique to prove atomicity for concurrent file systems. In the forward simulation relation, abstract state (*abstraction*) is introduced to represent the logical layout of concrete data structures and

```
 1 /*error and corner cases
 2 handling omitted*/
 3 def ins(path, name)
 4     cur=root;
 5     //traverse path from root
 6     ...
 7     lock(cur);
 8     node=init();
 9     insert(cur, name, node);
10     ▶ intuitive LP: INS ◀
11     unlock(cur);
12     return success;
13
14 def del(path, name)
15     cur=root;
16     //traverse path from root
17     ...
18     lock(cur);
19     node=lookup(cur, name)
20     lock(node);
21     delete(cur, name);
22     ▶ intuitive LP: DEL ◀
23     unlock(cur);
24     free(node)
25     return success;
26
27 def rename(src,sn,dst,dn)
28     //traverse path from root
29     //get src directory (sdir)
30     //and dst directory (ddir)
31     ...
32     lock(sdir);
33     lock(ddir);
34     dnode=lookup(ddir,dn);
35     snode=lookup(sdir,sn);
36     lock(dnode);
37     lock(snode);
38     delete(ddir,dn);
39     delete(sdir,sn);
40     insert(ddir,dn,snode);
41     ▶ intuitive LP: RENAME ◀
42     unlock(snode);
43     unlock(sdir);
44     unlock(ddir);
45     free(dnode)
46     return success;
```

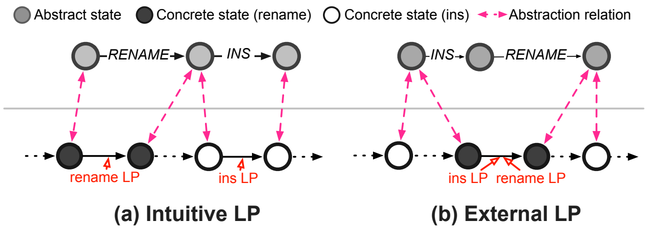
Figure 2. Pseudocode of a simplified file system. The detailed optimizations, error handling, path traversal code, and stat implementation are omitted for simplicity.

a set of *abstract operations* over the abstract state is used to specify the corresponding concrete operations. The core of a simulation proof is an *abstraction relation*, which relates the abstract state to the concrete state and is defined as an invariant over the relational (i.e., abstract-concrete) states.

We use directory operations, a major source of inter-thread concurrency, to illustrate how LPs can be used to prove atomicity in file systems. To simplify the exposition, we only consider six POSIX interfaces, mknod(path), mkdir(path), rmdir(path), unlink(path), rename(source, target) and stat(path), and merge mknod/mkdir into ins and unlink/rmdir into del. The implementations are shown in Figure 2, which represents a simplified modern file system with fine-grained locks.

An intuitive selection of LPs in the simplified file system is marked in the code (i.e., lines 10, 22 and 41 in Figure 2). Take ins as an example. Its LP is put at line 10 (i.e., before unlock). This is because the ins will update an inode in the critical section (lines 7–11) which is protected by a per-inode lock. Thus, the effect of ins takes place at the LP by atomically executing the abstract operation.

As shown in the simulation graph (Figure 3(a)), atomic operations INS and RENAME represent the abstract operations for ins and rename respectively. The *abstraction relation* always maintains the consistency between states at two levels. At the concrete level, each step defined by the semantics of programming language will make a transition, while each abstract operation makes a transition at the abstract level when the concrete code passes its LP. So the *abstraction relation* also establishes the correspondence between concrete operations and atomic operations, which further implies that ins and rename are atomic. As shown in Figure 4(a), by following the temporal order of LPs, we can construct a sequential history of abstract operations and results, which is consistent with the concrete execution and should be legal.



(a) Intuitive LP (b) External LP

Figure 3. Simulation diagrams. The central line is the division of the abstract level (above) and concrete level (below). Figure(a) shows the simulation diagram when an operation's LP resides in its own implementation. In Figure(b), the LP of ins is external and resides in rename's code.

3.2 Path Inter-dependency

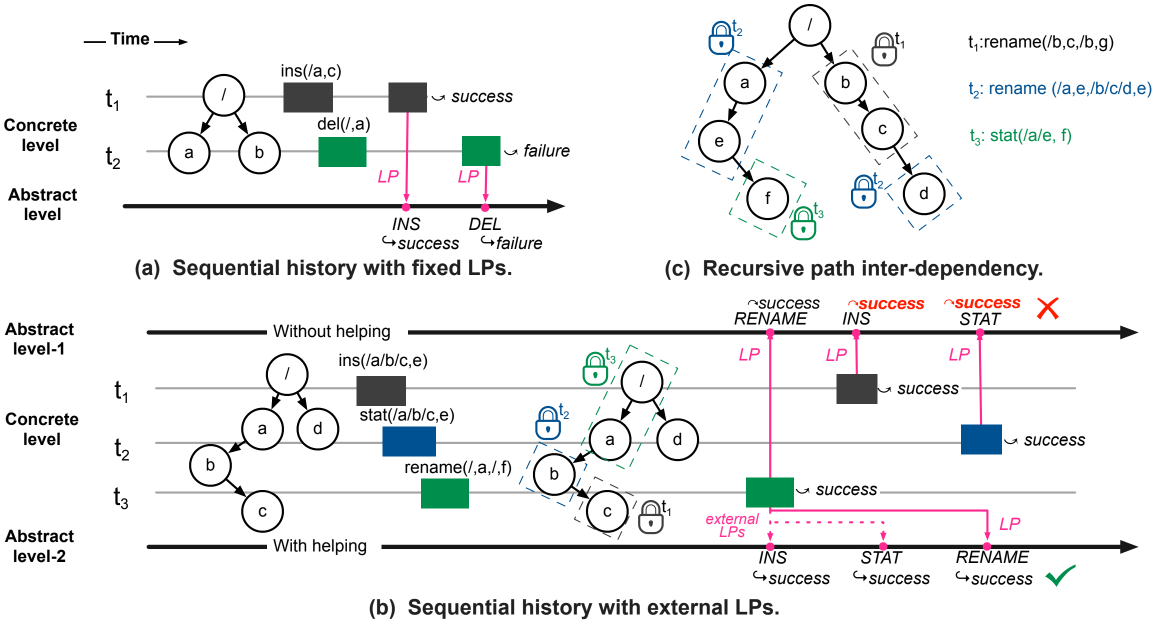
Unfortunately, identifying static LPs located in the implementation code cannot handle all the cases in concurrent file systems. As Figure 1 and 4(b) indicate, fixed LPs we choose are incorrect, and in the above legal interleavings we are not able to obtain a sequential, legal history by serializing the concurrent operations in terms of the temporal order of LPs. Digging into these cases, we find that existing implementations of concurrent file systems often leverage fine-grained locking to allow concurrent operations on separate inodes. So the path already traversed by an operation can be modified by another concurrent operation. Here we introduce a new concept, *path inter-dependency*, to illustrate the relationship between these concrete operations.

Definition 1 (*path inter-dependency*)

> One **concrete operation** A is said to have *path inter-dependency* on an interleaving operation B if B modifies A's traversed path.

In definition 1, the **abstract operation** of A relies on the integrity of the path as specified in the argument to execute successfully. Thus, if a concrete operation has *path inter-dependency* on a rename, its abstract operation should happen before RENAME. It gives the reason why the fixed LP is not sufficient—although the LP is when the effect is published, the linearization is determined when the *path inter-dependency* phenomenon happens, not based on when the LP executes.

To account for the *path inter-dependency*, when a rename is about to pass its LP and release its effects, it should first check whether it modifies other interleaved operations' traversed paths, and (if yes) help to commit their effects before itself. Thus the LPs of those operations should reside in the code of rename. As shown in Figure 3(b), at the LP of rename, the abstract level will first execute ins's abstract operation INS, then execute RENAME. We call this kind of LP an **external linearization point** [25], where the LP of one



(a) Sequential history with fixed LPs.

(c) Recursive path inter-dependency.

(b) Sequential history with external LPs.

Figure 4. Cases of interleaving operations. In Figure (a), ins completes earlier than del, and the execution can be linearized with fixed LPs. Figure (b) presents a more complicated case, where the operations can only be linearized with external LPs in rename. Figure (c) shows concurrent renames, which introduces recursive dependencies among operations and complicates helping.

operation resides in the code of another operation. According to POSIX, rename is the only file system interface that may break other threads' path integrity and lead to external LPs. However, the general concept of *path inter-dependency* makes it applicable to new file system interfaces [67] that might introduce path integrity violations.

Generality. We have done an investigation to see how many file systems allow *path inter-dependency* relations among operations. Specifically, we measure nine file systems, including six Linux (5.2.8) file systems (i.e., ext2 [10], ext4 [69], Btrfs [55], XFS [68], ReiserFS [2] and Tmpfs [3]), two FreeBSD (12.0) file systems (i.e., UFS [1] and ZFS [7]) and one teaching file system (i.e., xv6fs [19]) of xv6 (#b81891). First, we log at the begin and end of the critical section of most used, path-based file system operations (i.e., rename, create, unlink, mkdir, and rmdir). Then, we concurrently execute test cases of rename and op, where rename will modify op's path. It suggests that op has *path inter-dependency* on rename if rename finishes while meantime op is in the critical section. We have tested all combinations of rename + op (op is one of the five operations). As a result, all combinations show the *path inter-dependency* phenomenon in all the considered file systems. Thus, we argue that the *path inter-dependency* phenomenon is a generic and fundamental phenomenon which causes LPs of file systems to be external.

3.3 Challenges in Applying Helping

As introduced in §2, previous work [48] has proposed helping to support reasoning about external LPs in lock-free algorithms. Specifically, helping allows one thread to help another thread (e.g., thread t) commit its effect with a primitive lin(t). Here "lin(t)" means executing thread t's abstract operation atomically. Applying the helping mechanism in Figure 4(b), we can change line 41 of rename to

$$\langle \mathrm{LP} : \mathrm{lin}(t_1); \mathrm{lin}(t_2); \mathrm{RENAME}\rangle$$ [4]

Now when rename passes its LP, it will execute the abstract operations of t_1, t_2, and then its own.

However, there are three aspects that make existing helping not directly applicable to address the external LPs in file systems. First, helping in those lock-free algorithms is made *explicit* with a global array of per-thread information. Filling in a thread's structure fulfills the helping process while helping in file systems is *implicit*. There is no global information such as thread identifiers and threads' intended operations, which are necessary to perform helping. Second, only one thread is helped at a time in those lock-free algorithms (e.g., a push only helps one pop in elimination stack), but a rename may help an *unbounded* set of threads and should carefully decide the helping order. Different helping order may lead to different results. For example, in Figure 4(b), helping t_2 : stat first would lead to the failure of stat, which is different from the concrete execution. Moreover, we need to consider the problem of *recursive path inter-dependency*, in which two operations have dependency through more than one *path inter-dependency* relations.

[4] We use angle brackets to represent that the statements in it are executed atomically and suppose we know ins and stat's thread identifiers.

***Recursive* path inter-dependency.** Suppose three threads hold locks as shown in Figure 4(c), which may be possible after some execution. Now t_1 : rename is about to pass the LP, which will help execute t_2 : rename as t_2 has *path inter-dependency* on t_1. And releasing the effect of t_2 : rename will further break t_3's path integrity. Thus, t_3 : stat should also be helped and be ordered before t_2 : rename. The relation between t_3 and t_1 is called recursive *path inter-dependency*, which might happen due to interleaving renames. If there are multiple renames, the chains could extend recursively. Thus, the helping set should contain not only path-interdependent operations but also operations with recursive dependencies.

3.4 The Helper Mechanism for File Systems

We propose the helper mechanism for file systems, which is equipped with *ghost state* and a *linearize-before relation* to handle the above challenges. We introduce a new primitive linothers to implement the helper mechanism for file systems.

Ghost state. Ghost state [61] (or auxiliary state) is a widely used technique in verification, which is introduced not in the concrete programs but in the abstract model to prove the program's correctness. They do not affect the system behavior, and updates to ghost state can be grouped with a program statement into an atomic block.

In the file system, global information about other threads is missing, so ghost state is introduced by the helper mechanism to record the information. The information should include at least threads' identifiers and abstract operations, so we have a way to help the threads execute their abstract operations. For example, prior work on helping [48, 70] introduces a thread pool as ghost state, which is a mapping of thread identifiers to their abstract operations.

Linearize-before relation. The helper needs to find a set of threads which should be helped (*helping set*) and decide the relative order among these threads (*helping order*). We introduce *linearize-before relation* for the purpose, which refers to the relationship between an operation A and B if A should be ordered before B in the sequential history. To define the linearize-before relation, we need the ghost state to contain other information, which can be used to judge whether two operations have logical dependencies. For instance, the ghost state can record execution histories of each thread, such as the inodes they have traversed through. So the linearize-before relation can use the information to know which threads will be influenced by a rename. Also, the linearize-before relation should include recursive dependencies as in Figure 4(c). We will present the linearize-before relation we define for AtomFS in §5.2.

Linothers. We introduce a new primitive linothers(t) which combines the *ghost state* and *linearize-before relation* to implement the helper mechanism. The new primitive can be used to address the external LP issue and file system-specific challenges (§3.3) by updating the LP of rename (line 41 in Figure 2) to:

$$\langle \mathsf{LP} : \mathsf{linothers(t); RENAME} \rangle$$

```
ThreadPool tp : Tid->Descriptor;
def linothers(t):
    #find all linearize-before relations in thead pool
    lbset=linearizeBeforeSet(tp)
    #find all threads that should be linearized before t
    helpset=helpSet(t,lbset)
    #totorder is a list containing all elements of helpset
    #and totorder obeys all linearize-before relations
    totorder=totalOrder(helpset,lbset)
    #linearize all threads in totorder
    for tid in totorder :
        lin(tid)
```

Figure 5. Pseudocode of linothers.

We present the pseudocode of linothers in Figure 5. First, we use the linearize-before relation (implemented as linearizeBeforeSet) to find a set of thread pairs (lbset). The linearize-before relation should take the ghost state (e.g., a thread pool) as an argument. Then we find all threads that t should help using lbset. We put all threads to be helped in helpset and find a helping order that obeys all linearize-before requirements.

3.5 Proving Helpers Correct

Helpers allow us to reason about the *path inter-dependency* among concrete operations in file systems with extensions like ghost state and linearize-before relation. However, we could still fail to finish the simulation proof that establishes atomicity because helpers have been performed incorrectly (with wrong helping set or helping order) or the implementation is not atomic. Extra proof work is required to prove helpers correct in the simulation proof.

Besides the differences in §3.3, helping in file systems is also different from the lock-free algorithms in the following ways. Traditional helping aims to guarantee the wait-free property of lock-free algorithms, so threads that make progress must help less fortunate threads to complete their operations, and the effect of a helped operation is published immediately when helping happens. However, helping in file systems is a logical concept used to coordinate concurrent operations at the abstract level. When helping happens in file systems, only abstract operations execute, which is before when concrete operations could take effect. So the abstract state might be a few transitions ahead of the concrete state. This brings about new proof patterns that do not exist in the previous reasoning. For example, the *abstraction relation* that connects the abstract and concrete states is not straightforward when the two states are different. Also, the simulation proof requires that the return values match between the two levels, which obligates us to show that the concrete operation will compute the same result as abstract operation. The reasoning about such future executions is closely related to how helpers are performed. We will elaborate more on how to prove helpers correct as we explain the framework (§4) and the proofs of AtomFS (§5).

4 Concurrent Relational Logic with Helpers

4.1 Overview

Concurrent Relational Logic with Helpers (**CRL-H**) builds on prior work [48, 49] that combines relational specifications [21] and local rely-guarantee reasoning [27] to reason about concurrency, and thus achieves both expressive specification and compositional verification. However, the combination is insufficient to verify file systems with *path inter-dependency*, which might cause external LPs. CRL-H adopts the helper mechanism (§3) to overcome the challenges. Below we briefly introduce our combination of the local rely-guarantee reasoning and the relational specifications.

- **Local rely-guarantee reasoning** (LRG) is an extension of rely-guarantee reasoning [42] and introduces rely conditions, guarantee conditions and invariants in the specification. The rely condition (R) specifies a thread's expectations of *shared state* transitions made by other threads, while the guarantee condition (G) specifies the *shared state* transitions made by the thread itself. The boundary between shared and local states is described through invariants. Using local rely-guarantee, CRL-H can achieve compositional verification by proving the rely condition of each thread is implied by the guarantee conditions of others.
- In addition, CRL-H uses **relational specifications** for specifying the relation between concrete state and abstract state, which are more expressive than Hoare logic [40] and can prove the simulation relation for verifying atomicity. To do so, CRL-H introduces an abstract object (or abstraction), to specify the logical layout of concrete data structures and a set of abstract operations (Aops for short) to specify corresponding concrete operations, and adopts the abstraction relation [54] to prove that a concrete implementation refines its abstract specification.

CRL-H extends the combination approach with *helper metadata* and a *roll-back mechanism* to support helpers. Helper metadata is ghost state introduced by CRL-H to provide global information for *users* to decide the helping set and helping order, and reflects the semantics of helper process (e.g., which Aops are helped and their effects). Although an intuitive way to specify the consistency relation between two levels is to require that each corresponding state (e.g., inodes) should have the same content, it does not work in file systems as the helper will execute helped operations at the abstract level in advance and thus make the content not match. We propose a roll-back mechanism to establish the consistency relation by rolling back a list of applied effects at the abstract level. CRL-H also builds a logic that allows us to prove our implementation meets the specification using inference rules. The soundness of CRL-H ensures that the logic implies linearizability.

```
Definition Inum := Z.
Definition Root := Inum.
Definition FName := string.
Definition Links := Map.t FName Inum.

Inductive Inode :=
| File : list byte -> Inode
| Dir  : Links -> Inode.

Definition Imap := Map.t Inum Inode.
Definition AFS := prod Imap Root.
Inductive Args := (* omitted *).
Inductive Ret := Success | Failure (* others omitted *).
Definition Aop := AFS -> Args -> AFS -> Ret -> Prop.

Definition mkdirSpec : Aop :=
  fun afs args afs' ret => (mkdirCond afs args /\
  afs' = absMkdir(afs, args) /\ ret = Success) \/
  (~ mkdirCond afs args /\ afs' = afs /\ ret = Failure).
(* mkdirCond and absMkdir omitted for space reasons *)
```

Figure 6. Abstraction for concurrent file systems.

4.2 Specifying Concurrent File Systems

To specify concurrent file systems with CRL-H, we need to formally define abstract operations over the file system abstraction, rely/guarantee conditions (R/G), and invariants for specifying interactions among threads.

Abstraction with map spec. As shown in Figure 6, CRL-H provides a default abstraction for concurrent file systems which can be easily extended by users for specific needs. The abstract-level file system (AFS) is defined as the root inode number (Root) plus a map from inode numbers (Inum) to Inode. Inode could be either a directory (Dir), which maps file names to inode numbers, or files, which contain a list of bytes. Abstract operations (Aops) are defined as the atomic transitions on AFS, e.g., mkdirSpec in the figure.

Local reasoning is an essential technique to reduce the proof burden, where a thread could focus on the part of the memory that it owns and operates on. So the model of file system should support local reasoning by allowing to be specified as the focused inodes and the rest. We choose to model a file system with a *map spec*, which splits the description of a set of memory blocks for inodes and the corresponding shape properties (e.g., tree shape) into separate parts. It gives us more flexibility to spell out complex invariants when focusing on only parts of inodes in the file system tree. So we can represent a file system state as the union of focused inodes and a map that includes the rest of the inodes. We can enforce the shape properties of all inodes with an invariant. Instead, modeling the file system as a tree would force us to specify its layout and shape information globally, thus making it hard to focus on separate inodes because the shape properties would break.

Invariants. CRL-H supports local reasoning by logically splitting states into shared states and local state. It provides inference rules to reason about operations on local state like verifying sequential programs and reason about operations on shared states through a clearly defined protocol, which is specified using R/G and invariants. The shared states are formalized as triples of the concrete state (C memory), the abstract state and ghost state. We use invariants, defined as relational specifications, to specify the layout of shared states and constrain their well-formedness. For instance, the invariant (I) would be defined as:

$$\begin{aligned}\mathrm{I(cfs, afs, ghost)} = {} & \mathrm{cinv(cfs)} * \mathrm{ainv(afs)} \\ & * \mathrm{ginv(ghost)} \wedge \mathrm{acrel(cfs, afs)}\ldots\end{aligned}$$

Here we use separating conjunction $*$ from separation logic [64] to specify disjointness of file system states. The cinv describes the data layout of the file system at the concrete level (cfs) and should be instantiated according to the concrete implementation. The abstract-level file system layout is described by ainv, which is specialized for the AFS abstraction (afs). ainv includes a well-formedness constraint on afs, e.g., afs always forms a tree. The ginv specifies the layout of the ghost state. Some other constraints might also be required for accomplishing the proofs. For instance, an abstraction relation (acrel) is used to relate the abstract- and concrete-level states, i.e., afs and cfs. The abstraction relation will be further explained in §4.4.

Rely and guarantee conditions. CRL-H uses rely and guarantee conditions to specify the allowed transitions on shared states made by environmental threads and itself. To define the guarantee conditions of thread t, we need to find all possible transitions that can be performed on shared states by t. t's rely conditions are simply defined as the union of guarantee conditions of all other threads. Both rely and guarantee conditions specify the transitions over shared states consisting of concrete state, abstract state, and ghost state. This relational reasoning with relational specifications helps us to prove atomicity, which is stronger than functional correctness proved by pre- and post-conditions over program states. The concrete code execution (C code) modifies the concrete state (C memory) while executing abstract atomic operations updates abstract state. Meanwhile, we can record the necessary information in ghost state. However, we are not allowed to arbitrarily modify abstract state and ghost state because the modifications should not violate the invariants.

4.3 Helping with Helper Metadata

Helper metadata. CRL-H instantiates the ghost state proposed in the helper mechanism as *helper metadata*, which contains a Helplist and a ThreadPool, which maps thread IDs to pairs of Aopstate and Descriptor.

An Aopstate could be either (aop, args) which means the operation aop needs to be executed with args, or (end, ret) which means the operation has been finished with return value ret. The Descriptor is provided for users to record all the auxiliary information about a thread. For example, in AtomFS, we use per-inode locks to traverse a path. Thus, we add a field, LockPath, in the Descriptor, which records the acquired locks (including those that have been released) of the current operation. We will explain more about how to use the Descriptor for verifying file systems in §5.

In addition, Helplist records the execution order (at abstract level) of *helped threads*, and will be used in relating concrete state and abstract state (§4.4). Here, *helped threads* means their abstract operations are done by other threads but not themselves, while *unhelped threads* means their abstract operations are pending for execution. The helper metadata is intended to provide global information to decide helping set and order.

Helper process. As introduced in §3.4, we use a primitive—linothers to perform helping. First, the linothers will be executed at rename's LP. Second, the user of CRL-H is responsible for invoking the linothers for helping, which is achieved by applying an *implication-rule* (introduced in §4.5) provided by CRL-H. Third, if the AopState of the current rename is (aop, args) which means the rename is not helped by others, we should perform helping. The user should provide an ordered list of thread IDs that should be helped , and CRL-H will perform these threads' Aops on the abstract file system and append them at the end of Helplist. We will introduce how a user can generate the helping set and helping order in §5.2. If the AopState is (end, ret), which means the rename has been helped by other operations, we do not need to perform helping. The AopState should be carefully maintained, by initializing it to (aop, args) when an operation begins, setting it to (end, ret) when an operation is helped, and clearing the entry when an operation passes its LP.

4.4 Abstraction Relation with Roll-back Mechanism

Simulation-based proofs require us to define an abstraction relation that relates the abstract file system tree to the concrete file system tree. An intuitive way to relate the two trees is to require that each corresponding inode should have the same content. However, there are two issues that the intuitive relation cannot handle. First, concrete-level transitions inside the critical section could expose intermediate states, which do not match the abstract state. Second, as helpers will execute abstract operations, the abstract state could be a few transitions ahead of the concrete state, which makes the two states different. To solve the two issues, CRL-H introduces *relaxed consistency mapping* and a *roll-back mechanism* to help users in writing abstraction relation.

Relaxed consistency mapping. We classify the status of a inode according to whether the inode has been locked. We require concrete and abstract inodes be consistent with each other only when the status is unlocked. If an inode is locked, it means the inode might be arbitrarily modified, so we do not have to restrict its content, and the consistency of contents between two levels is allowed to be broken inside the critical section. When exiting critical sections, the consistency between the two levels should be reestablished.

Roll-back mechanism. The roll-back mechanism allows us to establish the abstraction relation by rolling back the effects of helped Aops on abstract state. To achieve this, CRL-H leverages the helper metadata. For instance, we could add a field, named Effect, in each thread's Descriptor. If a thread's Aop is helped, Effect would records which abstract inodes are modified and how they are modified by the Aop. With the metadata, the consistency relation between an abstract inode (Ino) and concrete inode (ino) with the same inode number (inum) could be written as:

$$\exists \mathsf{effects}, \mathsf{Ino}'.\ \ \mathsf{Ino}' = \mathsf{rollback}(\mathsf{Ino}, \mathsf{effects}) \wedge \mathsf{match}(\mathsf{ino}, \mathsf{Ino}')$$
$$\textit{where}\ \mathsf{effects} = \mathsf{search}(\mathsf{inum}, \mathsf{ThreadPool}, \mathsf{Helplist})$$

The search function will search through all entries in the thread pool to find related effects on the inum, and the selected effects are arranged in the reverse order of Helplist for the rollback function (i.e., first roll back the effects applied last). We choose to roll back the effects on the abstract inode instead of applying them on the concrete inode because the abstract state is easier to manipulate. Then the consistency relation between Ino and ino can be written concisely.

4.5 Proving with Inference Rules

Inference rules are provided to assist proving that a file system implementation meets its specification. The judgement of CRL-H is of the form $R;G;I \vdash \{P * (\mathsf{aop}, \mathsf{args})\}C\{Q * (\mathsf{end}, \mathsf{ret})\}$, where R, G, I and P/Q represent rely, guarantee, invariant and pre-/post-conditions. CRL-H provides two kinds of inference rules, *C-statement-rules* and *implication-rules*. Users can use *C-statement-rules* to step through the C code (modeled in Coq) and use *implication-rules* to update the abstraction and ghost state. The *C-statement-rules* are mostly standard and similar to rules of LRG [27], e.g., SequenceRule, IfRule, WhileRule, AtomRule, etc. For instance, the AtomRule is used to step through a primitive statement of the language (e.g., an assignment). The *implication-rules* adopt the technique from [50] and represent the updates to abstraction and ghost state as assertion-level changes. So auxiliary commands (e.g., linothers) are removed in the verified program. For example, users can invoke an LinothersRule for helping. CRL-H will perform helping on states as described in §4.3. Then the current precondition P will be updated to P', which reflects the states after helping. Our approach to verifying fine-grained concurrency requires reasoning about three aspects that do not have direct analogs in reasoning about sequential file systems:

- **Shared data protocol.** CRL-H requires specifying a protocol on how threads operate on shared data concurrently. The protocol is defined through invariants and R/G. Therefore, a user has to prove that the invariants hold at any time during program execution, while guarantee conditions should capture each atomic transition.
- **Assertion stability.** Thread-local reasoning about shared data from a thread's perspective should manifest allowed modifications that can be made by other threads. Thus, every thread-local assertion about shared data should be *stable*, which means if the assertion holds on the pre-state, it should hold on the post-state after concurrent interference. For example, because rename can help threads execute their Aops, a stable assertion could be written as $(\mathsf{Helped} \wedge \mathsf{P}_{\mathsf{helped}}) \vee (\mathsf{Unhelped} \wedge \mathsf{P}_{\mathsf{unhelped}})$, which means if the current thread is unhelped, it will satisfy $\mathsf{P}_{\mathsf{unhelped}}$ and if helped, it should satisfy the $\mathsf{P}_{\mathsf{helped}}$ assertion.
- **Abstraction transition at linearization point.** The AopState in the helper metadata describes the status of an abstract operation, which is initiated as (aop, args) when the operation begins and updated to (end, ret) when the operation passes its LP. This is enforced by describing the AopState in the pre-/post-conditions. In addition, We should make the abstract transition using an *implication-rule* at a proper time (i.e., at the LP or external LP).

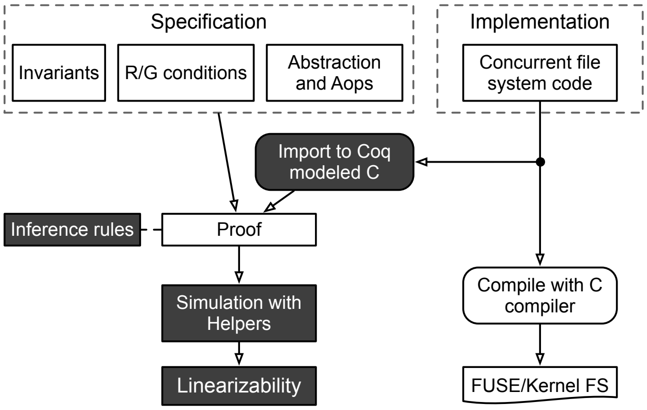


Figure 7. Development flow of applying CRL-H to verify concurrent file systems. Rectangular boxes denote source code; rounded boxes denote processes. Shaded boxes denote CRL-H framework components. The curved box denotes executable binary of the file system.

4.6 Defining Correctness

CRL-H intuitively relates the concrete file system code with its abstract-level specification. As shown in Figure 7, after proving the implementation meets the specification using the inference rules, the soundness of the framework proves our logic implies linearizability. Following [48], we prove an *atomicity* theorem as follows.

The proof for the theorem is constructed in the following steps. First, a rely guarantee-based simulation between the concrete code and the abstract operation is defined, and the simulation is proved to imply contextual refinement between the two sides. Then we prove the logic indeed establishes such a simulation. Thus the logic establishes contextual refinement. Finally, contextual refinement is proved to be equivalent to linearizability by prior work [30].

Theorem 1 (File System Atomicity)

Given the implementation and abstraction of a file system, if there exist *rely*/*guarantee* conditions and an *invariant*, such that for each operation of the implementation and corresponding abstract operation, we can prove the judgment hold w.r.t. the pre-/post-conditions by applying inference rules, then the file system implementation is atomic.

5 Proving AtomFS

AtomFS is a concurrent in-memory file system running on FUSE. We first introduce the *non-bypassable criterion* and how AtomFS uses lock coupling to meet the criterion (§5.1), then we show how to formalize the helpers (§5.2), the invariants of AtomFS (§5.3) and how we handle file descriptor-based interfaces (§5.4).

5.1 Non-Bypassable Criterion

We present the design considerations of AtomFS before digging into its proofs. The key feature of AtomFS is leveraging lock coupling in path traversals because lock coupling can ensure "non-bypass" among concurrent operations. The case in Figure 8 motivates our choice.

Two threads are executing (rename,del) and ins respectively. One possible interleaving would be first ins traverses to b and halts. Then rename finishes and del starts. Suppose ins does not always hold the lock and del can *bypass* ins by first holding the lock of c. Here, "A bypasses B" means the inode accessed by A in the tree is a descendant of the inode accessed by B. Then the execution would generate results as shown in the concrete level. However, the results cannot be matched by any sequential history (or the interleaved execution is **non-linearizable**).

If we analyze the case using helpers, rename should first help ins finish its abstract operation successfully, so a following del should fail in the abstract level. However, if del can bypass ins in the concrete execution, it can delete b's link to c, which causes ins to fail and generates non-linearizable behavior. We introduce the following *non-bypassable criterion* to ensure linearizability among path-based operations.

The *non-bypassable criterion* can be expressed as: an unhelped thread A cannot bypass a helped thread B (unhelped/helped thread defined in §4.3). Intuitively, when B is helped by some rename and therefore linearized, its effect is published based on the current abstract file system state. This generates the obligation to show the concrete state should allow B to compute the same result when it takes effect in the concrete level. However, if an unhelped thread A bypasses B and could modify the concrete state B will use, B may compute an inconsistent result with its abstract level. Thus, the criterion can guide us in handling such corner cases and ensuring linearizability. Here, the *non-bypassable criterion* only applies to path-based operations.

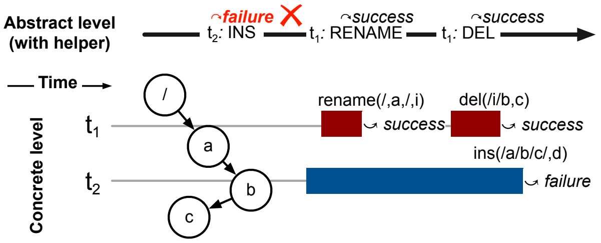


Figure 8. Non-bypassable criterion violation. The del bypasses ins and first acquires the lock of c.

Lock coupling. Following the criterion, AtomFS leverages lock coupling [38] for the path traversal process. Specifically, AtomFS always acquires the next inode's lock before it releases the current inode's lock. So our design satisfies the *non-bypassable criterion* and forbids such corner cases. In the proofs of AtomFS, we propose the non-bypassable invariants (§5.3) to check the bypass will not happen in AtomFS.

Linux VFS study. Although some existing file systems do not use lock coupling to forbid the occurrence of the bypass, they still need an approach (usually much more complicated than lock coupling) to obey the *non-bypassable criterion.* For example, Linux VFS adopts *traversal retry* [51] to allow operations to bypass each other when traversing the path. Specifically, an operation will check whether a rename operation has executed during the path traversal through a *revalidation process.* If it finds a rename has executed, it will redo the path traversal. As a result, rename does not need to help those threads that have not finished their path lookups. For the threads that are helped in the critical section, the lock will protect them from being bypassed. So the non-bypassable criterion is still obeyed. Compared with our approach, the *traversal retry* will increase the difficulty of file systems, which makes its proofs not as intuitive as lock coupling.

5.2 Formalizing Helpers

To formalize helpers, we introduce how helper metadata and linearize-before relation are defined and used in AtomFS.

Helper metadata. In AtomFS, the per-thread Descriptor of helper metadata is instantiated as three fields, including LockPath, Effect and FutLockPath. Here we focus on the first field, LockPath, which is a sequence of inode numbers that the current thread has locked through from the root inode. It could be either a single path for operations like mkdir or a pair of paths for rename (i.e., SrcPath and DestPath). For instance, in Figure 4(b), the SrcPath of t$_2$ is (root, a, e) and the DestPath is (root, b, c, d), while the LockPath of t$_3$ is (root, a, e, f).

Invariant name	States specified	Description	Provider
Abstract-concrete-relation	Abstract FS and concrete FS	Concrete inode and abstract inode shoud be related using roll-back mechanism.	AtomFS
Helped-non-bypassable	Ghost state	A helped Op cannot bypass another Op helped before it.	AtomFS
Unhelped-non-bypassable	Ghost state	An unhelped Op cannot bypass a helped Op.	AtomFS
GoodAFS	Abstract FS	The abstract FS has tree properties like root reachability.	CRL-H
Last-locked-lockpath	Concrete FS and ghost state	The last inode in the LockPath of thread t is locked by t in concrete FS.	AtomFS
Helplist-consistency	Ghost state	An abstract operation is helped iff its thread ID is in the Helplist.	AtomFS
Future-lockpath-validness	Abstract FS and ghost state	Thread t will acquire locks indicated by FutLockPath.	AtomFS
Lockpath-wellformed	Ghost state	The LockPathPrefix relation is acyclic.	AtomFS

Table 1. Invariants in AtomFS. Using CRL-H to verify AtomFS requires specifying several invariants. Each invariant (with *invariant name*) specifies restrictions between *states* (i.e., abstract FS, concrete FS and ghost state) as explained in *description*.

Linearize-before relation. As AtomFS adopts lock coupling, we can use LockPath to define the linearize-before relation between threads. We define the SrcPrefix relation to account for *path inter-dependency* and LockPathPrefix relation for recursive *path inter-dependency*. The SrcPrefix relation means the LockPath of a thread contains the SrcPath of a rename. The LockPathPrefix relation means that thread t_2's (any) LockPath is the prefix of thread t_1's (any) LockPath. For example, in Figure 4(b) (assuming t_2 is ready to pass its LP), thread t_3 has SrcPrefix relation on t_2 as its LockPath (i.e., (root, a, e, f)) contains t_2's SrcPath (i.e., (root, a, e)).

Then we can decide the helping set for a rename **recursively**. First, we find all the threads which have the SrcPrefix relation on the rename, and put the threads in a set named HelpSet (**Step-1: Init**). Then, we recursively pick one thread in the HelpSet, find all the new threads having LockPathPrefix relation on it, and put these new threads into the HelpSet (**Step-2: Recursive search**). We repeat the second step until we cannot add any new threads into the HelpSet. The helping order is an order of all threads in the HelpSet that satisfies all LockPathPrefix relations. The whole process happens atomically at the LP of a rename.

Here, LockPathPrefix and SrcPrefix relations may decide a linearize-before relation between two commutative operations, which means our relations are stricter than the ideal linearize-before relations. However, using stricter relations in the proofs does not comprise on correctness if the stricter relations can find a helping order. Specifically, we need to be careful with rename's path traversals to ensure deadlock-freedom. A rename will first traverse to the last common inode of source and destination path (with hand-over-hand locking), and only releases the lock of the inode after acquiring the lock of source and destination directory. Then we can ensure the LockPaths of threads do not form cycles.

5.3 Invariants in AtomFS

As shown in the Table 1, we list eight major invariants used to specify the global properties of AtomFS. Here, we introduce the two most important and interesting invariants.

Abstract-concrete relation. The abstract-concrete relation employs the roll-back mechanism to relate the abstract state and concrete state. We use the Effect in helper metadata to record the effects of a helped operation. Effect may consist of a set of micro-operations. For instance, OPins denotes an inode insertion operation and OPcreate denotes an inode creation operation. Take INS as an example. It inserts a child inode number cinum into a parent inode with inode number pinum. Its Effect is represented as (OPins : (pinum, name, cinum), OPcreat : cinum). The effects are recorded in the inode granularity, so we can search for all the effects on an inode and arrange the effects in the reverse order of the Helplist. Then the relation can be established by the roll-back mechanism.

Non-bypassable invariants. Non-bypassable invariants (i.e., *helped-non-bypassable* and *unhelped-non-bypassable* in the table) are used to meet the *non-bypassable criterion*. We use LockPath and future lockpath (FutLockPath) in the Descriptor, to formalize the non-bypassable invariants. The FutLockPath is initiated when an operation is helped, which records the inode locks it will acquire. For instance, if an ins(/a/b/c/, d) is helped and the LockPath is (root, a), then the FutLockPath is (b, c) (if the path lookups will succeed). For a helped operation op, all the operations behind op can be seen as possible *bypassers*. Here, the "behind" is defined through the LockPathPrefix relation. Now, non-bypassable invariants can be formalized as an originally behind operation (i.e., a *bypasser*) locking an inode which is in the FutLockPath of op.

5.4 Tuning FD-based Interfaces for Linearizability

Besides path-based interfaces, file systems also provide file descriptors (FDs) for locating inodes. However, adopting the FD-based interfaces will lead to non-linearizability in AtomFS. Consider the example in Figure 9. A rename helps ins, and a readdir starts execution after rename finishes. readdir bypasses ins and finds directory c empty. If we analyze the case using helpers, there is no sequential, legal history of the concrete execution. As discussed in *the non-bypassable criterion* (§5.1), readdir runs into inconsistency with ins because the state of c at the two levels are different when computations of readdir and READDIR happen. More generally, every FD-based interface can bypass a helped, path-based interface. In some cases, this would

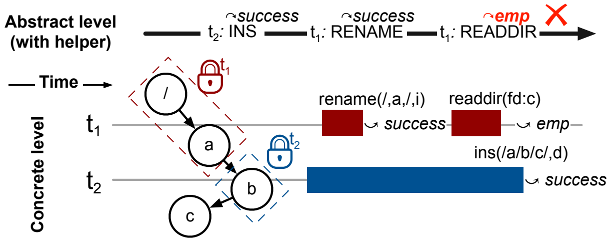


Figure 9. File descriptor-based interfaces. FD-based interface readdir bypasses a helped operation ins, which causes non-linearizablity.

cause different abstract and concrete states and introduce non-linearizable behavior. Therefore, AtomFS chooses to traverse a path to locate the target inode for those FD-based interfaces (e.g., read, write, readdir), which is possible because the high-level FUSE API provides the path arguments for all interfaces. AtomFS relies on VFS and FUSE to maintain the mapping from a file descriptor to the path of an inode so that AtomFS could support unmodified applications and still ensure linearizability. Surprisingly, POSIX semantics like reading/writing to an unlinked file is also supported because FUSE will create a temporary file for reads/writes after the file has been removed. However, it also makes VFS and FUSE as the trusted computing base of AtomFS.

Discussion about support for FDs FD-based interfaces scale much better than doing a pathname resolution for every read and write. To support FDs, several modifications are needed for AtomFS. First, AtomFS can adopt VFS's traversal retry mechanism (§5.1). So in Figure 9, rename will make ins redo the path lookups. Second, the del operation should not free the memory of an opened inode. We may introduce a reference count for each inode to represent that the inode is in use, so later FD-based accesses are still allowed. In this way, for operations that have not finished traversals when a rename happens, they will redo path lookups and helping is not required. For operations that are already in the critical section when helped, a bypass by an FD-based operation is harmless because the states they operate on will not interfere. Also, these FD-based operations have no *path inter-dependency* on renames, and therefore do not need to be helped. They are linearized when they pass their LPs.

6 Prototype Implementation

Coq provides a single language for specifications, implementations, and proofs, which allows us to build the prototypes of CRL-H and AtomFS. The development took several researchers about a year and a half, which includes learning the theories that underlie our framework, building the framework, and developing proofs for AtomFS.

CRL-H. CRL-H is built on an existing open source project, μC/OS-II [76] and several theoretical work [27, 48–50]. CRL-H follows the logic proposed by Liang et al. [48], which adopts RGSim [49] (a rely-guarantee based simulation) as the meta-theory and provides a logic based on LRG [27]. There are several major differences between CRL-H and Liang's framework. First, the framework models a simple language and only has paper proofs. Second, the framework also supports future-dependent LPs, which makes the state model and logic complicated and not easy to implement in Coq. CRL-H provides a simplified logic, which is enough to reason about external LPs in file systems. Third, to perform helping, the framework introduces auxiliary commands (e.g., lin(t)) and corresponding semantics of the commands. CRL-H instead directly models helping as assertion-level changes [50], which eliminates all auxiliary commands.

There are nearly 100k lines of code for CRL-H, about half of which are reused from μC/OS-II [76]. The reused parts include supports for C language (the memory model, semantics, part of C inference rules and auxiliary lemmas for inference rules), separation logic automation and libraries for maps, integers, maths, etc. Many original definitions, lemmas, and proofs are fixed to fit into the new state model. New implementations mainly devote to supporting LRG reasoning of our logic.

AtomFS. AtomFS is a concurrent in-memory file system. It employs a hash table followed by linked lists for directory lookups and a fixed-size array of indexes for file data storage. It provides POSIX-like interfaces at the top level. We have evaluated AtomFS using xfstests [4], a comprehensive file system testing suite, and the results show AtomFS can pass 418 cases out of 451 (test cases for tmpfs). All failed cases are caused by lacked functionality supports instead of bugs, i.e., current prototype of AtomFS does not support *hard/symbolic link*, *permission*, etc. Also, AtomFS does not consider crash safety yet.

Table 2 shows the lines of code for specifications, implementations, and proofs of AtomFS. Unlike previous work [12, 16] which generates executable code from Coq to Haskell, CRL-H has already modeled a subset of the C language, thus allowing the verified AtomFS (written in C language) to be compiled to binary without extraction. We run the verified implementation with a small unverified FUSE [32] driver as a user-level file system. AtomFS can run unmodified applications; however, it also includes VFS, the FUSE driver and library as trusted components. Besides FUSE and VFS, our trusted computing base includes the C compiler, the C implementation of a lock[5] and the memory allocator of glibc. Currently, for some internal functions (190 LOC) inside the critical section, we have only verified their sequential specifications because these internal functions will only modify lock-protected states and will not interfere with environments. We leave completing the formal proofs of these internal functions as future work.

Limitations. First, AtomFS does not support crash safety. Prior work [6] has proposed to decouple the in-memory file

[5] Locks have well-known linearizable implementations [45].

Component	Lines of code/proof
Abstraction and Aops	344
Invariants	1397
R-G conditions	451
Verified code	673
Proof	60,324
Total	63,099

Table 2. Lines of specifications, implementations, and proofs for AtomFS.

system, which presents simple interfaces to the clients, from the on-disk file system, which considers crashes. We follow the same design strategies in this work. In the prototype implementation, we aim to build an in-memory concurrent file system and prove its atomicity without considering crashes. To prove crash safety for concurrent file systems, we may need to extend CRL-H with crash conditions [16] to specify and verify the recovery procedure.

Second, as discussed in §5.4, AtomFS needs to traverse a whole path even for FD-based interfaces like read and write because AtomFS wants to ensure all interfaces are atomic. Although we can only prove the atomicity for path-based interfaces, mixed granularity of interfaces is hard to use for applications. For instance, the verified CMAIL [12] relies on atomic interfaces of file systems (e.g., read and link). AtomFS can adopt VFS's traversal retry mechanism to locate inodes with file descriptors and still ensure linearizability.

Third, it is hard to ensure our linearizability guarantees convey to the applications as VFS and FUSE do a lot of tracking and caching before passing the control to AtomFS. Or they could directly serve some read-only operations (e.g., read) from the cache without entering AtomFS. Therefore, the functional correctness relies on that the cache coherence protocols of VFS and FUSE are correct. To prove that the syscalls provided by VFS are also atomic, we need to reason about the syscall-level code, which would be easier because challenges like external LPs are hidden in the atomic specifications of AtomFS. In common cases, a syscall is linearized at the atomic invocation of AtomFS's interface. A read-only syscall could also be linearized when the cache hits. We leave verifying the syscall-level atomicity as our future work.

7 Performance Evaluation

We now focus on the performance evaluation that aims to answer the following questions:

- Can AtomFS provide reasonable performance for practical applications? (§7.2)
- Can AtomFS achieve good scalability on a multicore machine? (§7.3)

7.1 Experimental Setup

We run all of the experiments on a server with an Intel Xeon 2.30GHz CPU with 16 physical cores and 62GB DRAM running Linux 5.2.8. We compare the performance of AtomFS with a mature and widely-used disk file system (ext4 [69]), a formally verified high-performance file system (DFSCQ [14]) and an in-memory file system (tmpfs). All the evaluated file systems use Ramdisk (i.e., /dev/ram0 in Linux) as the storage.

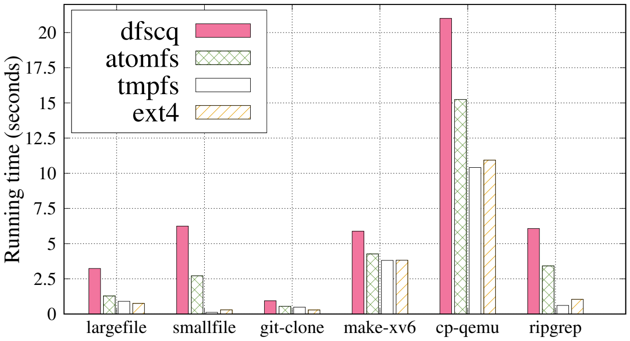


Figure 10. Application workloads. The figure shows the running time of different application workloads (i.e., git, compile, cp, ripgrep). Largefile benchmark will operate on a big file with 10MB, and the smallfile operates on 10K files with 1KB size.

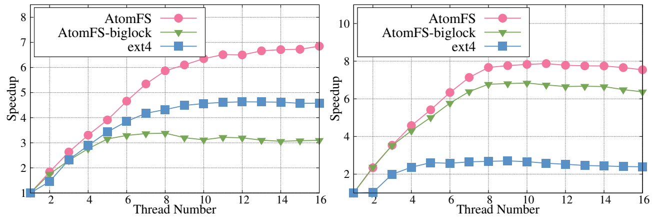
7.2 Application Performance

AtomFS is complete enough to run many kinds of realistic software, including Vim [74] and GCC [33]. To evaluate the application performance, we select two microbenchmarks and four application workloads: LFS microbenchmark [57, 65], cloning the git repository of xv6-public, compiling the sources of the xv6 file system with a makefile, copying source code of qemu and searching a string with a tool called ripgrep. The application workloads only use a single core.

The results are shown in Figure 10. Compared with AtomFS, DFSCQ needs more running time for all applications (from 1.38x to 2.52x). The main reason is that AtomFS is implemented in C instead of Haskell used in DFSCQ, thus can avoid the Haskell overhead. The performance of AtomFS is worse than tmpfs and ext4 for two reasons. First, AtomFS is implemented with FUSE, which introduces higher overhead than in-kernel file systems. Second, AtomFS uses simplified data structures to manage data and metadata. Both issues can be overcome in future work, and the results reveal that AtomFS can achieve reasonable performance for practical applications.

7.3 Multicore Scalability

To evaluate the scalability of AtomFS with lock-coupling and per-inode locks, we use the two most commonly used workloads in Filebench [29] (i.e., Fileserver and Webproxy). We implement a coarse-grained version of AtomFS using big-lock to precisely measure the speedup achieved by lock-coupling. In the big-lock version, all file system operations first acquire a big-lock and do not release the lock until the operations finish. We evaluate the big-lock version of AtomFS,



(a) Scalability on Fileserver. **(b) Scalability on Webproxy.**

Figure 11. Scalability of AtomFS. The overall scalability of AtomFS is better than the big-lock version of AtomFS and ext4.

AtomFS, and ext4 with 16 cores and gradually increase the thread number used in the benchmark. The speedup results are in Figure 11.

We can mainly draw three conclusions. First, the AtomFS does not bypass the VFS-level path lookups. The VFS-level traversals can provide good scalability, as the big-lock version of AtomFS still scales when the thread number increases to 8. Second, lock-coupling contributes to the scalability, which is confirmed by the better scalability and performance of AtomFS over the big-lock version of AtomFS. The throughput of AtomFS is 1.46x higher with 16 threads in Fileserver than the big-lock version (Figure 11(a)). This is because Fileserver concurrently handles more different directories and files (i.e., 526 different directories and about 10000 files). The performance of lock-coupling gains less improvement (1.16x higher throughput with 16 threads) in Webproxy (Figure 11(b)), as Webproxy involves only two directories, which cannot leverage the benefit of multicore concurrency. Third, AtomFS achieves somewhat worse performance than ext4 (6.39x/5.83x lower throughput in Fileserver/Webproxy with 16 cores). This is because the lock-coupling traverse of directory and file operations becomes the major bottleneck as the cores increase.

8 Experience

Theory choices. Concurrent separation logic (CSL) [8, 72] and rely-guarantee (RG) [27, 28, 42, 73] are two mainstream approaches to verifying concurrent software. We initially adopted CSL to verify our implementation. CSL has the notion of ownership, where a thread can only access the portions it owns. A thread can do local reasoning by acquiring the ownership of shared resources. However, transferring the ownership of shared states would break its connection with other shared resources, e.g., locking an inode will transfer the inode into private states but will also lose the relation between the inode and other inodes.

Experience with LRG. Our experience with LRG reasoning suggests that LRG is powerful for fine-grained reasoning. The G condition should capture all transitions on shared states allowed by the program, which could be a big disjunction of lots of cases for some programs. However, for AtomFS, all accesses to shared states are performed in the critical section. Therefore, by weakening the guarantee conditions, we can effectively merge all concrete-level operations into three kinds of transitions, Lock, Unlock and Lockedtrans. The Lock/Unlock captures the atomic operation of acquiring/releasing an inode lock. The Lockedtrans specifies that a thread can make arbitrary modifications to an inode that is locked by the thread, which captures all possible transitions inside the critical section. Since a thread's guarantee conditions serve as the rely conditions of other threads, merging the guarantee conditions reduces the cases to reason about when proving stability. In some cases, we need to strengthen rely conditions (i.e., other threads' guarantee conditions) to precisely capture environmental interference and pass the stability check. For example, when specifying the INS transition, we should not only specify the changes in the abstract state but also require that the lock of the father inode should be held. This prevents arbitrary insertions in the abstract inode, which violates the invariants.

One downside of LRG is that the proof rules have many side conditions. For each atomic statement, we have to check that the transition satisfies the G, the invariants hold after the transition, and the assertion is stable under environmental interference. Also, it is hard to achieve full automation in LRG because we need to manually specify the assertions to represent updates to the abstraction and ghost state.

9 Conclusion

This paper has presented CRL-H, a framework for specifying, implementing, and verifying concurrent file systems with atomic interfaces through the helper mechanism. CRL-H allows developers to precisely specify the expected behavior of file systems through relational specifications and enables developers to verify the file systems with thread-local reasoning. We have applied CRL-H to specify and verify AtomFS, the first formally verified, fine-grained, concurrent file system. Our experience shows that the proof burden is manageable, and the AtomFS can achieve acceptable performance.

Acknowledgments

We sincerely thank our shepherd Don Porter, the PDOS group and the anonymous reviewers for their insightful comments. We thank Xinyu Feng and Hongjin Liang for their advices on CRL-H and proofs of AtomFS. We also thank Mingkai Dong for his initial effort in helping implement the AtomFS file system. This work is supported in part by the National Key Research & Development Program (No. 2016YFB1000104), and the National Natural Science Foundation of China (No. 61772335). Haibo Chen (haibochen@sjtu.edu.cn) is the corresponding author.

All of CRL-H and AtomFS's source code will be publicly available at https://ipads.se.sjtu.edu.cn/projects/atomfs.

References

[1] 2019. FreeBSD Handbook. https://www.freebsd.org/doc/handbook/ Referenced April 2019.

[2] 2019. Reiser4 FS Wiki. https://reiser4.wiki.kernel.org/index.php/Main_Page Referenced April 2019.

[3] 2019. tmpfs - Linux manual page. http://man7.org/linux/man-pages/man5/tmpfs.5.html Referenced April 2019.

[4] 2019. Toward better testing. https://lwn.net/Articles/591985/ Referenced August 2019.

[5] 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. 2016. 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)*. ACM, New York, NY, USA, 175–188. https://doi.org/10.1145/2872362.2872404

[6] Srivatsa S Bhat, Rasha Eqbal, Austin T Clements, M Frans Kaashoek, and Nickolai Zeldovich. 2017. Scaling a file system to many cores using an operation log. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 69–86.

[7] Jeff Bonwick and Bill Moore. 2007. ZFS: The last word in file systems.

[8] Stephen Brookes. 2007. A semantics for concurrent separation logic. *Theoretical Computer Science* 375, 1-3 (2007), 227–270.

[9] Miao Cai, Hao Huang, and Jian Huang. 2019. Understanding Security Vulnerabilities in File Systems. In *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '19)*. ACM, New York, NY, USA, 8–15. https://doi.org/10.1145/3343737.3343753

[10] Remy Card. 1995. Design and implementation of the second extended filesystem. In *Proceedings of the First Dutch International Symposium on Linux, 1995*.

[11] Tej Chajed. 2017. *Verifying an i/o-concurrent file system*. Master's thesis. Massachusetts Institute of Technology.

[12] Tej Chajed, Frans Kaashoek, Butler Lampson, and Nickolai Zeldovich. 2018. Verifying concurrent software using movers in {CSPEC}. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 306–322.

[13] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. 2019. Verifying concurrent crash-safe systems with Armada. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP 2019)*. Huntsville, Ontario, Canada.

[14] Haogang Chen, Tej Chajed, Alex Konradi, Stephanie Wang, Atalay İleri, Adam Chlipala, M Frans Kaashoek, and Nickolai Zeldovich. 2017. Verifying a high-performance crash-safe file system using a tree specification. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 270–286.

[15] Hao Chen, Xiongnan Newman Wu, Zhong Shao, Joshua Lockerman, and Ronghui Gu. 2016. Toward compositional verification of interruptible OS kernels and device drivers. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 431–447.

[16] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M Frans Kaashoek, and Nickolai Zeldovich. 2015. Using Crash Hoare logic for certifying the FSCQ file system. In *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 18–37.

[17] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. 2009. VCC: A practical system for verifying concurrent C. In *International Conference on Theorem Proving in Higher Order Logics*. Springer, 23–42.

[18] Robert Colvin, Lindsay Groves, Victor Luchangco, and Mark Moir. 2006. Formal verification of a lazy concurrent list-based set algorithm. In *International Conference on Computer Aided Verification*. Springer, 475–488.

[19] Russ Cox, M Frans Kaashoek, and Robert Morris. 2011. Xv6, a simple Unix-like teaching operating system.

[20] W-P De Roever, Frank de Boer, Ulrich Hanneman, Jozef Hooman, Yassine Lakhnech, Mannes Poel, and Job Zwiers. 2001. *Concurrency verification: Introduction to compositional and non-compositional methods*. Vol. 54. Cambridge University Press.

[21] Willem-Paul De Roever, Kai Engelhardt, and Karl-Heinz Buth. 1998. *Data refinement: model-oriented proof methods and their comparison*. Vol. 47. Cambridge University Press.

[22] John Derrick, Gerhard Schellhorn, and Heike Wehrheim. 2011. Mechanically verified proof obligations for linearizability. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 33, 1 (2011), 4.

[23] John Derrick, Gerhard Schellhorn, and Heike Wehrheim. 2011. Verifying linearisability with potential linearisation points. In *International Symposium on Formal Methods*. Springer, 323–337.

[24] Coq development team. 2019. The Coq Proof Assistant. http://coq.inria.fr/

[25] Brijesh Dongol and John Derrick. 2015. Verifying linearisability: A comparative survey. *ACM Computing Surveys (CSUR)* 48, 2 (2015), 19.

[26] Cezara Drăgoi, Ashutosh Gupta, and Thomas A Henzinger. 2013. Automatic linearizability proofs of concurrent objects with cooperating updates. In *International Conference on Computer Aided Verification*. Springer, 174–190.

[27] Xinyu Feng. 2009. Local rely-guarantee reasoning. In *ACM SIGPLAN Notices*, Vol. 44. ACM, 315–327.

[28] Xinyu Feng, Rodrigo Ferreira, and Zhong Shao. 2007. On the relationship between concurrent separation logic and assume-guarantee reasoning. In *European Symposium on Programming*. Springer, 173–188.

[29] Filebench. 2019. Filebench. https://github.com/filebench/filebench

[30] Ivana Filipović, Peter OHearn, Noam Rinetzky, and Hongseok Yang. 2010. Abstraction for concurrent objects. *Theoretical Computer Science* 411, 51-52 (2010), 4379–4398.

[31] Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2018. ReLoC: A mechanised relational logic for fine-grained concurrency. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*. ACM, 442–451.

[32] FUSE. 2019. The reference implementation of the Linux FUSE (Filesystem in Userspace) interface. https://github.com/libfuse/libfuse

[33] GNU. 2019. GCC, the GNU Compiler Collection. https://www.gnu.org/software/gcc/. Referenced April 2019.

[34] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 653–669. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu

[35] Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan Newman Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. 2018. Certified concurrent abstraction layers. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 646–661.

[36] Chris Hawblitzel, Erez Petrank, Shaz Qadeer, and Serdar Tasiran. 2015. Automated and modular refinement reasoning for concurrent programs. In *International Conference on Computer Aided Verification*. Springer, 449–465.

[37] Danny Hendler, Nir Shavit, and Lena Yerushalmi. 2004. A scalable lock-free stack algorithm. In *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*. ACM, 206–215.

[38] Maurice Herlihy and Nir Shavit. 2011. *The art of multiprocessor programming*. Morgan Kaufmann.

[39] Maurice P Herlihy and Jeannette M Wing. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12, 3 (1990), 463–492.

[40] Charles Antony Richard Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (1969), 576–580.

[41] Atalay Ileri, Tej Chajed, Adam Chlipala, Frans Kaashoek, and Nickolai Zeldovich. 2018. Proving confidentiality in a file system using DiskSec. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 323–338. https://www.usenix.org/conference/osdi18/presentation/ileri

[42] Cliff B. Jones. 1983. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 5, 4 (1983), 596–619.

[43] D. Jones. 2019. Trinity: A Linux system call fuzz tester. http://codemonkey.org.uk/projects/trinity/

[44] Artem Khyzha, Mike Dodds, Alexey Gotsman, and Matthew Parkinson. 2017. Proving linearizability using partial orders. In *European Symposium on Programming*. Springer, 639–667.

[45] Jieung Kim, Vilhelm Sjöberg, Ronghui Gu, and Zhong Shao. 2017. Safety and Liveness of MCS Lock—Layer by Layer. In *Asian Symposium on Programming Languages and Systems*. Springer, 273–297.

[46] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. 2009. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 207–220.

[47] Mohsen Lesani, Christian J Bell, and Adam Chlipala. 2016. Chapar: certified causally consistent distributed key-value stores. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 357–370.

[48] Hongjin Liang and Xinyu Feng. 2013. Modular verification of linearizability with non-fixed linearization points. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 459–470.

[49] Hongjin Liang, Xinyu Feng, and Ming Fu. 2012. A rely-guarantee-based simulation for verifying concurrent program transformations. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 455–468.

[50] Hongjin Liang, Xinyu Feng, and Zhong Shao. 2014. Compositional verification of termination-preserving refinement of concurrent programs. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. ACM, 65.

[51] Linux. 2019. Pathname lookup; The Linux Kernel documentation. https://www.kernel.org/doc/html/latest/filesystems/path-lookup.html. Referenced April 2019.

[52] Richard J Lipton. 1975. Reduction: A method of proving properties of parallel programs. *Commun. ACM* 18, 12 (1975), 717–721.

[53] Lanyue Lu, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, and Shan Lu. 2013. A study of Linux file system evolution. In *Presented as part of the 11th {USENIX} Conference on File and Storage Technologies ({FAST} 13)*. 31–44.

[54] Nancy Lynch and Frits Vaandrager. 1995. Forward and backward simulations. *Information and Computation* 121, 2 (1995), 214–233.

[55] Chris Mason. 2007. The btrfs filesystem. *The Orcale cooperation* (2007).

[56] Changwoo Min, Sanidhya Kashyap, Byoungyoung Lee, Chengyu Song, and Taesoo Kim. 2015. Cross-checking semantic correctness: The case of finding file system bugs. In *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 361–377.

[57] mit pdos. 2019. mit-pdos/fscq: FSCQ is a certified file system written and proven in Coq. https://github.com/mit-pdos/fscq. Referenced April 2019.

[58] Ben C Moszkowski. 1997. Compositional reasoning using interval temporal logic and tempura. In *International Symposium on Compositionality*. Springer, 439–464.

[59] Gian Ntzik. 2016. *Reasoning about POSIX file systems*. Ph.D. Dissertation. Imperial College London.

[60] Gian Ntzik, Pedro da Rocha Pinto, Julian Sutherland, and Philippa Gardner. 2018. A concurrent specification of POSIX file systems. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[61] Susan Owicki and David Gries. 1976. Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM* 19, 5 (1976), 279–285.

[62] Peter O'Hearn, John Reynolds, and Hongseok Yang. 2001. Local reasoning about programs that alter data structures. In *International Workshop on Computer Science Logic*. Springer, 1–19.

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

[64] John C Reynolds. 2002. Separation logic: A logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. IEEE, 55–74.

[65] Mendel Rosenblum and John K Ousterhout. 1991. The design and implementation of a log-structured file system. In *ACM SIGOPS Operating Systems Review*, Vol. 25. ACM, 1–15.

[66] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. 2016. Push-Button Verification of File Systems via Crash Refinement. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 1–16. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/sigurbjarnarson

[67] Kuei Sun, Matthew Lakier, Angela Demke Brown, and Ashvin Goel. 2018. Breaking Apart the {VFS} for Managing File Systems. In *10th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*.

[68] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. 1996. Scalability in the XFS File System.. In *USENIX Annual Technical Conference*, Vol. 15.

[69] Theodore Ts'o and Stephen Tweedie. 2002. Future directions for the ext2/3 filesystem. In *Proceedings of the USENIX annual technical conference (FREENIX track)*.

[70] Aaron J Turon, Jacob Thamsborg, Amal Ahmed, Lars Birkedal, and Derek Dreyer. 2013. Logical relations for fine-grained concurrency. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 343–356.

[71] Viktor Vafeiadis. 2008. *Modular fine-grained concurrency verification*. Technical Report. University of Cambridge, Computer Laboratory.

[72] Viktor Vafeiadis. 2011. Concurrent separation logic and operational semantics. *Electronic Notes in Theoretical Computer Science* 276 (2011), 335–351.

[73] Viktor Vafeiadis and Matthew Parkinson. 2007. A marriage of rely/guarantee and separation logic. In *International Conference on Concurrency Theory*. Springer, 256–271.

[74] vim. 2019. welcome home: vim online. https://www.vim.org. Referenced April 2019.

[75] James R Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D Ernst, and Thomas Anderson. 2015. Verdi: a framework for implementing and formally verifying distributed systems. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 357–368.

[76] Fengwei Xu, Ming Fu, Xinyu Feng, Xiaoran Zhang, Hui Zhang, and Zhaohui Li. 2016. A practical verification framework for preemptive OS kernels. In *International Conference on Computer Aided Verification*. Springer, 59–79.

[77] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. 2006. Using model checking to find serious file system errors. *ACM Transactions on Computer Systems (TOCS)* 24, 4 (2006), 393–423.