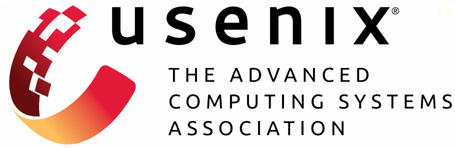


Using Dynamically Layered Definite Releases for Verifying the RefFS File System

Mo Zou, Dong Du, and Mingkai Dong, *Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University; Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China;* Haibo Chen, *Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University; Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China; Huawei Technologies Co. Ltd*

https://www.usenix.org/conference/osdi24/presentation/zou

This paper is included in the Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation.

July 10–12, 2024 • Santa Clara, CA, USA

978-1-939133-40-3

Open access to the Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation is sponsored by

Using Dynamically Layered Definite Releases for Verifying the RefFS File System

Mo Zou[1,2], Dong Du[1,2], Mingkai Dong[1,2], Haibo Chen[1,2,3]
[1]*Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University*
[2]*Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China*
[3]*Huawei Technologies Co. Ltd*

Abstract

RefFS is the first concurrent file system that guarantees both liveness and safety, backed by a machine-checkable proof. Unlike earlier concurrent file systems, RefFS provably avoids termination bugs such as livelocks and deadlocks, through the *dynamically layered definite releases* specification. This specification enables handling of general *blocking scenarios* (including ad-hoc synchronization), facilitates *modular* reasoning for *nested blocking*, and eliminates the possibility of circular blocking.

The methodology underlying the aforementioned specification is integrated into a framework called MoLi (Modular Liveness Verification). This framework helps developers verify concurrent file systems. We further validate the correctness of the locking scheme for the Linux Virtual File System (VFS). Remarkably, even without conducting code proofs, we uncovered a critical flaw in a recent version of the locking scheme, which may lead to deadlocks of the entire OS (confirmed by Linux maintainers). RefFS achieves better overall performance than AtomFS, a state-of-the-art, verified concurrent file system without the liveness guarantee.

1 Introduction

This paper presents RefFS, a concurrent file system with a mechanized proof of both safety and liveness properties. Liveness means that each operation of RefFS provably terminates under the assumption of fair scheduling. The proof rules out a wide range of bugs that occur in concurrent file systems [67], such as deadlocks, livelocks, and infinite loops.

Proving the absence of termination bugs is important, because they are too subtle to be correctly handled by developers. For instance [5], a task might not deadlock with another task via a direct ABBA[1] pattern, but instead through a complex circular dependency chain involving multiple tasks. A wide range of other termination bugs [54] occur, posing a threat to the software system. Once triggered, these bugs can lead to serious consequences, such as a system hang [16].

Testing and program analysis techniques (see §2 for more detail) have been used to detect (a subclass of) termination bugs. Although effective in practice, they cannot cover all possible cases. Formal verification is a promising approach. Researchers have made tremendous progress in concurrent file system verification [20, 101] and liveness verification [43, 56, 74]. Yet, modular liveness verification (focusing on one operation or one line of code at a time) of concurrent file systems remains an open problem.

In principle, proving liveness requires a well-foundedness argument [2, 60], i.e., within a finite number of steps, some progress event happens. For a sequential program, a well-founded metric such as the remaining steps of the program measures its progress [46, 66]. With each step, the metric must decrease, but not infinitely. Hence, the program provably terminates after running out the metric.

Unfortunately, proving liveness for a concurrent file system still faces the following challenges. First, the approach should support general *blocking scenarios*. A thread that is blocked in a busy waiting loop (e.g., to acquire a lock) cannot achieve progress by its own steps, but relies on the steps of *other* threads (e.g., the thread owning the lock). We aim for general busy waiting loops. This should be distinguished from work [12, 48] that supports only lock primitives, ignoring ad-hoc synchronization, which is common in file systems (see §2).

Furthermore, it is important to support modular reasoning, even though *nested blocking* causes progress dependencies between threads. Consider the following case. An `unlink` operation owns `parent` and requests `child`. Thread t_1 that tries to acquire `parent` is blocked by `unlink`, which itself is blocked by another thread t_2 that owns `child`. A metric for t_1's progress towards acquiring `parent` would include not only `unlink`'s steps to release `parent`, but also t_2's steps to release `child`. The latter contributes to t_1's progress *indirectly*. Explicitly considering such indirect steps hampers modularity. This issue becomes more pronounced with more threads chained by nested blocking.

Last but not least, the approach should prevent *circular nested blocking*. While an intuitive approach might specify a static order for nested blocking, the complexity of file systems introduces *diverse* and *dynamic* blocking order. On the one hand, file systems exhibit diverse nested blocking scenarios, with different blocking orders and concurrently executed by multiple threads. On the other hand, these nested blocking scenarios exhibit dynamic order, i.e., the exact rank of some specific blocking event cannot be known statically. For instance, the parent-child nested blocking of `unlink` refers to a dynamic set of inode pairs as a file system evolves. Two inodes with parent-child relation may swap their positions, ex-

[1] One acquires locks in the order of AB while another in the order of BA.

hibiting opposite orders in `unlink`. In `rename`, the blocking order of old and new parents is also unknown in advance. To avoid circular dependency, formally capturing the blocking order is essential but extremely challenging.

To meet such challenges, this paper makes the following contributions:

- A methodology for proving liveness, based on an acyclic waits-for graph. A vertex refers to an action waited by a blocked thread. The blocking thread must definitely fulfill the action. A directed edge represents a waits-for dependency between actions. To avoid circular dependencies, the waits-for graph must be acyclic.

- The dynamically layered definite releases (*DLDR*) specification, which supports modular liveness reasoning about concurrent file systems, with the following key ideas. (1) *Definite release* specifies that an acquired lock will always be released. (2) It proves termination of ad-hoc busy waiting lock loop based on a metric-decreasing idea. (3) We assign each definite release a layer, and allow a definite release to wait for only a higher-numbered one. Acquiring a lock waits only for the definite release of the lock (direct steps); layers decouple dependencies between definite releases (indirect steps), achieving modular reasoning. (4) The layering has two components: one follows the file system hierarchy (a parent may wait for a child), which is acyclic by definition; a *temporary dependency* models the non-deterministic blocking order in rename, and ensures acyclicity by construction.

- A protocol-level proof of Linux VFS's locking scheme based on an extension of the *DLDR* specification. The proof follows the Linux directory locking documentation [29]. We found a serious deadlock flaw; it was confirmed and fixed (we prove the fix correct).

- The MoLi framework for verifying termination and functional correctness of concurrent file systems. MoLi supports (1) definite releases with dynamic layers to achieve modular termination reasoning, and (2) non-atomic abstract operations to model non-atomic implementations. The framework is mechanized in Coq to ensure the reliability of the verification. Currently, MoLi does not support crash safety. MoLi's soundness has been formally proved on paper (a mechanized Coq proof is left as future work).

- The RefFS file system, the first modularly-verified concurrent file system with termination guarantees. RefFS supports highly concurrent path traversal using reference counting (refcount) [30]. Users are not bothered by the more fine-grained behaviors because the abstraction of RefFS hides refcounts, locks, and internal data structures, and exhibits atomic directory lookups.

The rest of this paper is organized as follows. §2 motivates this work with a study of termination bugs. §3 explains the MoLi methodology. §4 introduces the *DLDR* specification and its extension to directory locking in Linux. §5 describes the MoLi framework. §6 presents RefFS's design and verification. §7 evaluates RefFS and MoLi. §8 relates MoLi to previous work. §9 concludes.

2 Motivation

2.1 Study of Termination Bugs

Termination bugs cover a wide range, from non-concurrent ones (such as infinite loops) to concurrent ones (such as deadlocks and livelocks). A recent survey [16] on security vulnerabilities in file systems shows that about 7% of CVEs are related to non-termination. A prior study [67] on file system patches reveals that up to 40% of concurrency bugs are due to deadlocks. Deadlock-related semantic bugs are hard to diagnose, e.g., misuse of the `GFP_KERNEL` flag [70], and may hurt the system for years.

From a verification perspective, this raises several important questions: (1) How can one classify these bugs based on the challenges they pose for verification? (2) What are the primary classes of termination bugs? (3) What makes these bugs dominant and challenging to avoid? Answering these questions can help focus our verification efforts. Therefore, we performed a comprehensive study of termination bugs in Linux file systems (from 2020 to 2023). We collected 213 bugs in total by reading commit messages of patches [54]. We make the following observations.

Bug classification. Termination bugs can be classified based on whether they are concurrency bugs or not. 18% of termination bugs are non-concurrent. They include infinite loops or recursion, due mainly to logic mistakes (e.g., missing checks [82]) or generic errors (e.g., inappropriate truncation [42] or overflow [94]). 82% of termination bugs are concurrent. Within concurrent bugs, 95% of them are deadlocks and 5% are livelocks. Deadlock occurs when a thread becomes blocked, waiting for a specific action that never happens, and none of the involved threads can make progress. In livelock, a thread is constantly delayed, resulting in an inability to make progress.

Let us now look into deadlocks to understand the underlying factors contributing to their prevalence.

Ad-hoc synchronization. 46% of deadlocks (all percentages hereafter are relative to deadlocks) involve ad-hoc synchronization, where a thread is waiting for a specific event, such as transaction completion [11], flushing of dirty inodes [9], or other custom synchronization points [76, 90]. Deadlock analysis tools commonly focus on well-known and structured synchronization patterns, e.g., lock acquisition and release, but ad-hoc synchronization often lacks such patterns, which makes it challenging to analyze and detect.

Nested blocking. 21% of deadlocks are of type AA, where

a thread is blocked by itself [25, 72]. The remaining 79% involve nested blocking. Nested blocking occurs when a task that is blocking another task, gets blocked itself. For instance, nested acquisition of locks may cause nested blocking. 42% of deadlocks involve *both* nested blocking and ad-hoc synchronization [1, 24]. 23% of deadlocks involve at least three concurrent tasks [4, 7]. To prevent such bugs, it is necessary to examine, not only the local blocking order, but also the absence of global, circular dependencies within the system. However, existing approaches often fail to present a global order of dependencies, hampering the ability to detect such deadlocks effectively.

Dynamic order. 8% of deadlocks exhibit a dynamic blocking order, where the exact order between two blocking events cannot be predetermined. For instance, object removal (`unlink`/`rmdir`) acquires inode locks in a parent-child order, with the definitions of "parent" and "child" based on the current state of the file tree. As the file tree evolves, the set of inode pairs that satisfy this parent-child relationship dynamically changes. Similarly, many other data structures, such as the forest structure in BTRFS and various list implementations, also exhibit dynamic blocking order. These scenarios further contribute to the complexity of the issue. Even for experts in the domain, mistakes still occur [10, 86, 87], highlighting the difficulty of effectively managing these complexities.

To summarize, this paper focuses primarily on addressing the deadlock-related challenges in file systems, i.e., ad-hoc synchronization (§4.1), nested blocking (§4.2), and dynamic order (§4.3). We briefly discuss the support for livelocks (§4.5) and non-concurrent bugs (§5.3).

2.2 Limitations of Previous Work

There are a number of program-analysis-based techniques that aim at detecting deadlock [51, 91], livelock [13] or infinite loop [15, 17] respectively. Although effective in practice, their common problem is false positives. Programmers still have to manually confirm or reproduce the bug.

Various fuzzing-based testing tools [38, 50] can also reveal termination bugs. However, they and previous program-analysis tools cannot avoid false negatives.

For instance, the Linux kernel has a runtime validator to check locking correctness [31]. Users inform the validator of the hierarchy (a fixed order) between lock objects. This has the following drawbacks. First, the validator does not recognize ad-hoc synchronization. Second, it does not provide a general principle on how to handle dynamic locking order, whose hierarchy cannot be predetermined. Third, the annotations by developers may be wrong or insufficient, giving rise to both false positives [95] and false negatives.

Some efforts support deadlock-freedom (DF) verification [12, 48, 62, 93]. They track dependencies between blocking primitives to prevent circular blocking. This approach treats deadlock-freedom as a safety property, and does not prevent livelocks or non-termination of critical sections. They often assume known lock primitives, and do not apply to ad-hoc synchronization.

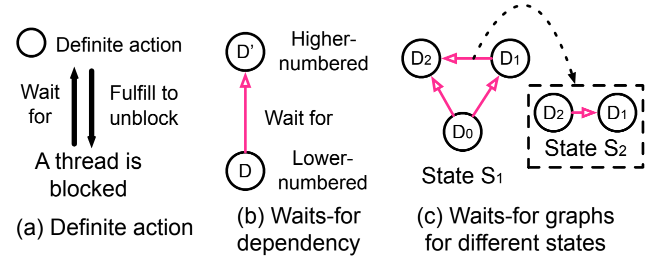


Figure 1: **The MoLi methodology for proving termination.** Waits-for graph consists of definite actions (vertices) and waits-for dependencies (edges). In (c), the dependencies between $\mathcal{D}_1$ and $\mathcal{D}_2$ are different at different states.

Some of these DF efforts [14, 61] also have limited support for dynamic lock orders. They consider only situations where a lock order change has *local* effects, i.e., it suffices to locally check some ordering constraints of the current operation to ensure acyclicity, without considering concurrent operations. For instance, for a rotation of a balanced tree, it suffices to check only the relations between the moved nodes, to ensure a global tree-based partial order. However, in a file system, it is necessary to check the absence of circular dependencies globally after the order change, which requires nontrivial concurrency reasoning (see §4.3 for more detail).

There is theoretical work [32, 64] to help reason about general blocking scenarios. LiLi [64] proposes a program logic to support the verification of starvation freedom and deadlock freedom. But LiLi does not support layered reasoning (see §4.2), and thus does not allow modular reasoning over nested blocking (i.e., verifying each line of code independently). TaDA Live [32] introduces layers to capture blocking orders, but it does not support layer changes caused by concurrent operations. Neither TaDA Live nor LiLi has a mechanical framework or an executable implementation.

3 The MoLi Methodology

Our approach to proving liveness is to exhibit an acyclic waits-for graph. A vertex is an action that a blocked thread is waiting for. A directed edge represents a waits-for dependency between actions. The waits-for graph shrinks as follows (we will discuss its growth in §4.5). Vertices with out-degree zero (not waiting for anything) must be unblocked to be removed from the graph. New vertices become leaves and are unblocked. Consequently, all waited actions eventually happen, creating progress for the blocked threads. Below, we discuss how this approach handles the challenges of file systems.

To support general blocking scenarios, MoLi borrows ideas from previous work [64]: a thread t is blocked when another

thread does not fulfill the unblocking action that t is waiting for (e.g., releasing the lock or finishing the transaction); thus, the specification should describe actions that one thread will definitely fulfill (Fig. 1a). All such definite actions should be specified by proof authors, and constitute the vertices of the waits-for graph (Fig. 1c).

In nested blocking, a definite action, e.g., releasing lock $\mathtt{L}_1$ by thread t, gets blocked and waits for another definite action, e.g., releasing lock $\mathtt{L}_2$ by another thread. We represent this waits-for dependency as a directed edge. To capture acyclicity, MoLi requires proof authors to layer vertices so that a vertex only waits for a *higher-numbered* vertex (Fig. 1b). For instance, proof authors can define the layers as the reverse topological order of the waits-for graph.

To capture the dynamic order of nested blocking, MoLi allows to define layers *dynamically* according to the system state. For instance (Fig. 1c), definite action $\mathcal{D}_1$ waits for $\mathcal{D}_2$ in one state, while this dependency may be the opposite in a different state. The layers reflect the waits-for dependencies of the current state.

To ensure acyclicity despite layer changes, MoLi enforces a *dependency condition* on layers: for any ongoing dependency between definite actions, their layer relation represents this dependency and stays unchanged despite state changes. For instance, as long as definite action $\mathcal{D}_1$ waits for $\mathcal{D}_2$, the layer of $\mathcal{D}_1$ must remain less than $\mathcal{D}_2$ despite state changes. Because all ongoing dependencies are immune to state changes, the system is never in danger of circular dependency.

We now explain in more detail how to apply this methodology to a concurrent file system.

4 Dynamically Layered Definite Releases

Directory locking refers to the locking scheme used for directory operations. Ensuring its correctness is important yet challenging. We tackle the challenges by introducing the dynamically layered definite releases specification (§4.1-4.3). Then, we apply the specification to the Linux Virtual File System (VFS) (§4.4) and discuss how to support delay (§4.5).

4.1 Definite Release

The rule for concurrent access in a Linux FS is to protect each inode with its own associated, fine-grained lock. A thread t acquiring a lock may be blocked by another thread holding the lock. To prove t's termination, we need to show that t will eventually become unblocked. Consider the code snippet in Fig. 2a[2]. The code traverses from the `cur` directory and looks up the name `path[i]` to find the `next` inode. The `lookup` is protected by holding the lock on `cur`, and the lock is released afterward. Assume all threads only execute this piece of code.

[2] This simplified version is incorrect because it omits reference counting; we will fix this in §6

```
// Pre: no lock owned
while(path[i]!=NULL)
  lock(cur);
  next=lookup(cur,path[i]);
  if (next==NULL) {
    unlock(cur); return;}
  unlock(cur);
  cur=next; i++;
}
```

(a) traversal loop.

```
def lock(cur):
  int i;
  i=getAndInc(cur.next);
  while(i!=cur.owner){}

def unlock(cur):
  cur.owner=cur.owner+1;
```

(b) ticket lock.

Figure 2: **Single locking in path traversal.** Termination of `lock(cur)` relies on other threads releasing the lock by increasing `cur.owner`.

A fair lock, such as a ticket lock (Fig. 2b), is used to ensure termination; every thread lines up for the lock by getting a ticket. A thread acquires the lock if its ticket equals `owner`, and releases the lock by increasing the `owner`. The question is why the `lock(cur)` statement would terminate.

Blocking is caused by the absence of environmental behavior, e.g., not releasing the lock. To prove termination, the specification should describe the certainty of some state transition. For lock-based blocking, we propose a domain-specific specification called *definite release*.

Definition 1 (Definite Release)

> **Definite release** says, for some thread t and lock, if t owns the lock, t will eventually release the lock.

Definite release is inspired by the *definite action* notion proposed by LiLi [64], which can characterize an action that will definitely happen. However, one key difference is that definite action in LiLi does not support layered reasoning, and thus cannot modularly handle nested locking (see §4.2).

Specifically, definite release ($\mathcal{D}$) specifies a state transition: "t owns the lock" $\leadsto$ "t releases the lock", where, informally, the notation $\leadsto$ states that the state transition *eventually* happens. Definite release supports busy-waiting lock loops (not just lock primitives). For instance, definite release of a ticket lock can be formalized as $(\mathtt{owner} = \mathrm{t}.\mathtt{i}) \leadsto (\mathtt{owner} = \mathrm{t}.\mathtt{i}+1)$, where t.`i` means the local variable `i` of thread t.

Rely-guarantee style reasoning. Definite release establishes a protocol that helps reason about the termination of locks as follows. First, once acquired, the release of a lock must be **guaranteed** by each thread. Second, to prove the termination of a lock, a thread may **rely** on some other thread releasing the lock. However, the above protocol does not avoid circular reasoning, which is usually unsound in proving termination. For instance, in a deadlock, each thread relies on the other to release, but this is circular and will never happen. LiLi's approach fixes this by requiring that each thread fulfills a definite release *without* waiting itself for any definite release.

Definite release achieves thread-modular verification of Fig. 2, focusing on one thread at a time, rather than explicitly considering steps of concurrent threads. **Rely**: if thread t is

```
1 // Pre: no lock owned          5 // Perform checks and
2 lock(parent);                  6 // do unlink/rmdir
3 child=lookup(parent,name);     7 ...
4 lock(child);                   8 unlock(parent);
```

Figure 3: **Code snippet for nested locking in `unlink/rmdir`.** The release of `parent` may be blocked by the acquisition of `child`. Code for error handling omitted.

blocked, spinning inside the **while** loop of `lock`, t relies on definite release, i.e., the increase of `owner` by lock holder; t can acquire the lock, because it needs to wait for a finite number of definite releases only. **Guarantee**: once t acquires the lock of `cur`, assuming that `lookup` terminates, t guarantees to fulfill the release without waiting for any definite release.

4.2 Hierarchical Layering

Some file system operations require nested locking (holding one lock and requesting another). For instance, consider the algorithm for removing an inode (in `unlink` or `rmdir`) in Fig. 3. The algorithm acquires `parent` to look up `child`. The lock on `parent` does not protect its children. To check whether `child` can be removed, we acquire `child`. When the operation finishes, it releases `parent`.

Unfortunately, in such nested locking, the release of the first lock may be blocked waiting for the release of the second lock. Hence, the guarantee of the first lock's definite release no longer holds.

If we stick to LiLi's approach in §4.1, we cannot prove the first lock by using its definite release, but have to specify more fine-grained actions that can definitely happen without waiting for any actions. For instance, acquiring `parent` may have to wait for (1) the release of `child` if the thread that owns `parent` is blocked requesting `child`, and (2) the release of `parent` if the thread that owns `parent` is not blocked. As a result, the proof for `lock(parent)` explicitly considers how `lock(child)` would be unblocked first, which is not modular. A modular approach would verify each line of code separately. The evaluation in §7.2 shows that LiLi's approach may cost 7 times the proof effort (measured in the lines of Coq) than our modular approach presented below.

We define *lock dependency* as below.

Definition 2 (Lock Dependency)

For nested locking in order of `A` and `B`, the definite release of lock `A` may depend on the definite release of lock `B`. This defines the **lock dependency** relation from `A` to `B`.

Intuition on termination. Lock dependency coincides with waits-for dependency (§3). Fig. 4a shows the possible lock dependencies in a file system as a *waits-for graph*. There is a lock dependency from a parent to its child, as shown by the directed edge. The dependencies extend transitively to the parent's descendants. For example, `root` has a lock dependency

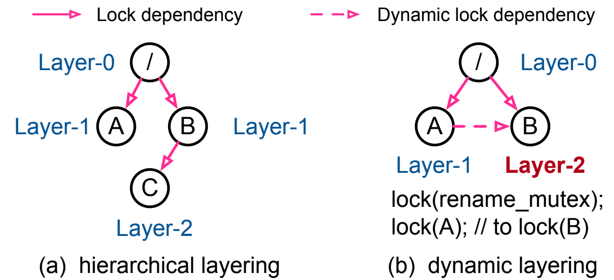


Figure 4: **Waits-for graphs and layerings.** Hierarchical layering accounts for parent-child lock dependencies. Dynamic layering captures the dynamic lock dependencies in `rename`.

on `B`, which has a lock dependency on `C`. If the waits-for graph ever contains a cycle, the system is deadlocked.

The good news is that this waits-for graph so far follows the file system tree, so the dependency chain ends at the leaf inodes. The definite release of a leaf inode does not wait for anything. The definite releases of other inodes in the chain occur one by one in the leaf-to-root direction. Consequently, the definite release of all inodes must happen, ensuring that the parent-child nested locking terminates.

Hierarchical layering for definite releases. It is not harmful for definite releases to have dependencies *as long as* the dependencies are not circular. To formalize this intuition, we choose to assign a layer to each definite release, and allow a definite release to wait only for a *higher-numbered* one. In a file system, we use *hierarchical layering* — the layer of an inode's definite release (an inode's layer in short) is the length of the path from `root`. For instance, in Fig. 4a, `root` is assigned 0, and after each hop, the layer increases by one.

More formally, hierarchical layering can be represented as below. A *layer function* $\mathcal{HL}$ takes the inode number `inum` and file system state `FS` to return `inum`'s layer under `FS`. If `inum` is reachable from the root, distance computes the length of path (with type *Nat*) from the root to `inum` under `FS`. The layer is undefined otherwise.

$$\mathcal{HL}(\texttt{inum},\texttt{FS}) \stackrel{\text{def}}{=} \text{distance}(\text{root}(\texttt{FS}),\texttt{inum},\texttt{FS}) \quad \text{if defined}$$

For a well-formed `FS`, each inode is reachable from the root following a unique path, i.e., each inode has a uniquely determined layer. Hence, inodes are totally ordered, and dependencies are not circular for any well-formed `FS` (assume `FS` does not change for now). Users specify the well-formedness of `FS` as invariants (e.g., each child has only one parent), and prove that invariants always hold.

Modular reasoning with layering. Each thread should still **guarantee** to release an acquired lock. But its fulfillment can wait for a *higher-numbered* definite release. Hierarchical layering ensures the wait is not circular. A thread proves the termination of a `lock` statement independently by only **relying** on definite release of the lock.

```
def lock_rename(d1,d2):          6    lock(d1);
1 if(d1==d2){                    7    return;
2   lock(d1);return;}            8 }
3 lock(rename_mutex);            9 lock(d1); TDep:=(d1,d2)
4 if(ancestor(d2,d1)){          10 lock(d2); TDep:=None
5   lock(d2);                   11 return;
```

Figure 5: **Nested locking in `lock_rename`.** The locking order is dynamic, e.g., `lock_rename(d1,d2)` and `lock_rename(d2,d1)` may acquire `d1` and `d2` in an opposite order. Code in gray boxes captures the dynamic order using ghost state. These locks are released when `rename` exits.

Layering decouples the dependency between definite releases and enables modular reasoning for Fig. 3. When t is blocked inside `lock(parent)`, t relies on definite release of `parent`, and proves termination by considering only `parent`'s internal waiting queue, without looking into other code. t proves `lock(child)` similarly. After t acquires `parent`, its definite release requires to prove `parent`'s layer is less than `child`, which is true by definition of layers.

4.3 Dynamic Layering

The `rename` operation moves an inode (more precisely, the subtree with the inode as root) from its old parent to a new parent. It needs to acquire the locks of both directories to ensure atomicity. The function `lock_rename` in Fig. 5 is a simplified version of the implementation of Linux VFS (ignore the code in gray boxes for now). If the two directories are the same, it only needs to acquire one lock. To avoid concurrent issues, VFS requires a cross-directory `rename` to acquire the global per-filesystem lock `rename_mutex`. Holding `rename_mutex` prevents another cross-directory `rename` from changing the ancestry relationship. Furthermore, to not conflict with the parent-child order, if one directory is the ancestor to another, it acquires the ancestor first. If not, the locking order does not matter, so the code chooses a default order, i.e., old parent first (`d1` before `d2`).

Dynamic lock dependency. In contrast to the parent-child lock dependency, the lock dependency in `lock_rename` cannot be predetermined. Specifically, prior to acquiring `rename_mutex`, the lock dependency between `d1` and `d2` is *not stable* since other threads might dynamically alter it. Once `rename_mutex` is acquired, the lock dependency becomes fixed. If `d2` is an ancestor to `d1`, the lock dependency is from `d2` to `d1` as shown in lines 5-6. Otherwise, it is from `d1` to `d2` as shown in lines 9-10. Furthermore, after lines 6 and 10 where the code has acquired the respective locks for `d1` and `d2`, we can remove the dependency between them, because one of them no longer waits for the other.

Intuition on termination. The waits-for graph remains acyclic during `lock_rename`, which ensures termination. Before `lock_rename` executes, the graph is tree-shaped. Then there is a dynamic lock dependency between `d1` and `d2`. According to the locking rules in `lock_rename`, this dynamic lock dependency is not from a child to its ancestor, and thus does not form a cycle with existing parent-child dependencies. Finally, removing the dynamic lock dependency will not introduce a cycle. The acyclicity of the waits-for graph ensures that definite releases happen in order, despite dynamically-changing lock dependencies. The code terminates, because all definite releases are guaranteed to be satisfied.

Dynamic layering. We propose *dynamic layering* to capture the lock dependency in `lock_rename`. The layer for each inode is defined as the length of the *longest* path from the root in the waits-for graph. For instance, after the introduced dynamic lock dependency (drawn as a dashed arrow) in Fig. 4b, the longest path from `root` to `B` is `root`-`A`-`B`, so the layer for `B` is now 2.

To encode dynamic layers, we leverage *ghost state* [75], a commonly used verification technique. Ghost state is added by users in the abstract model to assist the proof, which does not influence (or exist in) the concrete program. We introduce a *temporary dependency* `TDep`, a globally unique value that tracks the lock dependency introduced by `lock_rename`. `TDep` is an option type of a pair of inums, i.e., either `None` or $(\texttt{inum}_1, \texttt{inum}_2)$ representing there is a lock dependency from $\texttt{inum}_1$ to $\texttt{inum}_2$.

Dynamic layering $\mathcal{DL}$ is defined below. $\mathcal{DL}$ takes `inum` to return its layer under *full state* `S`, where `S` includes file system state `FS` and ghost state `TDep`. The root inode has layer 0. If there is no dynamic dependency to `inum`, i.e., `inum` does not equal the second item in `TDep` (written $\texttt{TDep.inum}_2$ for simplicity), `inum`'s layer is one plus its parent's layer. If `inum` equals $\texttt{TDep.inum}_2$, its layer is one plus the larger layer between the hierarchical layers (as defined in §4.2) of its parent and $\texttt{TDep.inum}_1$. Otherwise, the layer is undefined.

$$
\begin{aligned}
&\texttt{S} = (\texttt{FS}, \texttt{TDep}) \\
&\mathcal{DL}(\texttt{inum}, \texttt{S}) \stackrel{\text{def}}{=} \\
&\begin{cases}
0 & \text{if } \texttt{inum} = \textsf{root}(\texttt{FS}) \\
\mathcal{DL}(\texttt{par}, \texttt{S}) + 1 & \text{if } \exists \texttt{par}, \textsf{parent}(\texttt{par}, \texttt{inum}, \texttt{FS}) \\
& \qquad \land \texttt{inum} \neq \texttt{TDep.inum}_2 \\
\max\{\mathcal{HL}(\texttt{TDep.inum}_1, \texttt{FS}), & \text{if } \exists \texttt{par}, \textsf{parent}(\texttt{par}, \texttt{inum}, \texttt{FS}) \\
\qquad \mathcal{HL}(\texttt{par}, \texttt{FS})\} + 1 & \qquad \land \texttt{inum} = \texttt{TDep.inum}_2 \\
\text{undefined} & \text{otherwise}
\end{cases}
\end{aligned}
$$

This definition focuses on inode locks. For the per-filesystem lock `rename_mutex`, we also specify a definite release, whose layer is a special minimum value so it can depend on all other definite releases.

The code in gray boxes (Fig. 5) presents feasible updates to `TDep`. Only the thread that holds `rename_mutex` can set `TDep`, and `TDep` is `None` when no thread holds `rename_mutex`. `TDep` is set to (`d1`,`d2`) after `lock(d1)` in line 9 to represent the upcoming lock dependency from `d1` to `d2`. Note that we do not need to set `TDep` after `lock(d2)` in line 5 because the

lock dependency from d2 to d1 is already present in the waits-for graph due to transitive parent-child dependencies. We set TDep to None after line 10 to remove the lock dependency.

Reasoning with dynamic layering. Each lock statement still enjoys a modular proof by relying on its definite release. The guarantee of definite release can wait for a higher-numbered definite release. Now, taking state changes into consideration, we must further prove that the dependency relation (i.e., layer relation) stays unchanged during the wait. We call this the *dependency condition*.

The dependency condition is vital to acyclicity. Note that a circular dependency can happen only if a state change introduces a direct or indirect dependency opposite to an existing dependency. However, the dependency condition forbids this, by requiring the dependency relation to be stable.

Let's prove the dependency condition for Fig. 3. When a thread has acquired parent, and gets blocked by lock(child) at line 4, parent's definite release waits for that of child. The dependency condition requires to show that parent is lower-numbered than child, even in the presence of concurrent operations modifying file-system state. This is the case because (1) the current thread holding parent prevents a concurrent unlink/rmdir/rename from removing child and (2) according to the definition of $\mathcal{DL}$, child's layer is greater than or equals parent's layer $+1$.

4.4 Directory Locking in Linux VFS

To understand whether dynamic layering scales to more directory locking orders in VFS, we extend dynamic layering to cover all the nested locking scenarios mentioned in the Linux documentation [29]. The extra lock dependencies that have not been discussed yet are the following: (1) the *dir-to-non-dir* dependency, from a directory to a non-directory, and (2) the *inode-pointer* (i.e., inode address) dependency, from a non-directory to a non-directory with larger inode pointer. For instance, (1) link creation locks the parent and then the non-directory source, or (2) rename locks the source and the target when they are non-directories.

To capture these dependencies, a layer could either be (Dir, *nat*) for a directory with its dynamic layer ($\mathcal{DL}$ defined in §4.3), or (NonDir, *addr*) for a non-directory with its inode address. Comparison rules are:

- (Dir, *nat*) < (NonDir, *addr*) for the *dir-to-non-dir* dependency;
- (NonDir, $addr_1$) < (NonDir, $addr_2$) iff $addr_1 < addr_2$ for the *inode-pointer* dependency;
- (Dir, n_1) < (Dir, n_2) iff $n_1 < n_2$.

These rules give a total order of layers, because a directory is always acquired before a non-directory and the two groups are ordered by dynamic layers and inode pointers, respectively.

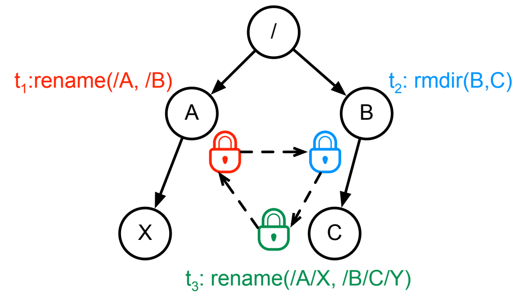


Figure 6: **A deadlock bug in Linux VFS.** Locking order according to increasing inode pointer order is (C, A, B), which conflicts with parent-child order (B, C).

When layers do not change, deadlocks will not happen if the code acquires locks in this total order (this can be easily verified). When considering layer changes, one proves the dependency condition to prevent circular dependency. We have shown the reasoning for the most challenging case, i.e., rename in §4.3. Proof of all other cases is provided in [99].

Doing the proofs helped us uncover a flaw in a recent version of the locking scheme (commit 28ecee [53]). Apart from locking the two parents, rename may also need to lock the source inode (under the old parent) to be renamed, and the target inode (under the new parent) if it already exists. The locking rules before commit 28ecee used to be:

- locking the source if it is a non-directory;
- locking the target if it exists;
- (if one need to lock both) locking them following the order mentioned above, i.e., directory before non-directory, and non-directories in inode pointer order.

However, commit 28ecee additionally locks the source even when it is a directory, for the purpose of updating its pointer to the new parent. For a non-cross-directory rename, this introduces a new, dynamic order between source and target subdirectories that is not protected by rename_mutex. Commit 28ecee takes it for granted that locking source and target subdirectories (and also two parents) in inode pointer order would be enough to establish a total order between directories.

However, the problem is that inode pointer order is not transitive with parent-child order, as shown in Fig. 6. Specifically, assume the inode pointer order is C < A < B (all are directories) and assume three operations have finished pathname lookups. t_3 owns rename_mutex and C, and requests A. t_1 owns root and A, and requests B. t_1 owns B, and requests C. Now, there is a deadlock. The maintainer confirmed this [98].

The fix [87] is to acquire the source subdirectory only in the cross-directory rename case, because a non-cross-directory rename does not change the parent of the source. The locking of source and target subdirectories is now protected under rename_mutex, and we can capture this order by constructing a dynamic layering (see [99] for details).

We found this flaw when we failed to define the dynamic layering specification for the scheme. Indeed, having a formal specification that effectively captures lock dependencies is crucial. Such a specification not only aids in conducting formal proofs, but also enhances our fundamental understanding of the system. Interestingly, even without engaging in code proofs, the specification itself can help uncover practical bugs and vulnerabilities.

4.5 Discussion about Support for Delay

A thread will not terminate if it is infinitely delayed in an infinite loop. For example, when a thread requests an unfair lock, e.g., test-and-set lock, other threads repeatedly preempt the lock first. Delays are benign as long as there is whole-system progress, e.g., the thread that preempts the lock can make progress. To allow benign delays, while preventing infinite delay without whole-system progress, previous work [64] proposed the concept of token transfer. The basic premise is that each thread is assigned a finite number of tokens. When a thread causes delays for other threads, it should either show progress itself or consume its tokens.

Let us go back to the discussion in §3. The waits-for graph grows due to normal execution or delay; either case has a decreasing metric that shows progress. When a thread normally executes, although it may get blocked by a **while** loop, its code size decreases. We assume a fixed but arbitrary number of threads to ensure the waits-for graph does not grow infinitely. In the delay case, a thread is made to execute more steps and may get blocked, but the delaying thread shows progress or consumes its tokens.

The mechanism for delay is necessary for proving the absence of livelock, where a thread is constantly delayed in infinite loops. This pattern does not appear in RefFS. For the sake of simplicity in presentation, we will omit these details.

5 The MoLi Framework

5.1 Overview

The MoLi framework verifies the functional correctness and termination of file systems based upon the following ideas.

- MoLi expresses correctness with *termination-preserving refinement* [66], which means that all observable events (e.g., output and termination events) from the implementation (i.e., concrete data structures and operations) must also be produced from the abstraction (i.e., logical layouts and abstract operations over them). Hence, functional correctness and termination of implementation is ensured, provided the abstraction is correct in these aspects. Termination-preserving refinement also helps build proofs in a layered way, i.e., low-level code can be replaced by its abstraction in a higher-layer proof. For instance, the proof of applications can use the abstraction of file systems.

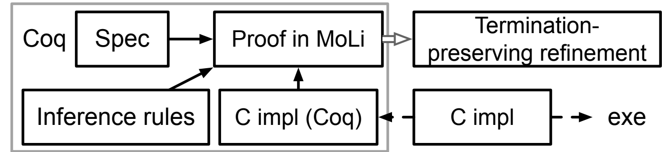


Figure 7: **The MoLi workflow.** Users provide a specification. MoLi helps users develop code proofs with inference rules.

- MoLi supports *compositional* concurrency reasoning with *rely and guarantee conditions* [35, 49]. A rely condition specifies the interference that a thread will accept from the other threads. A guarantee condition specifies the transitions that a thread will make. If the rely condition of a thread is implied by the guarantee of all other threads, and each thread is individually correct, then the concurrent system is correct.

MoLi supports modular liveness reasoning with layered definite actions. Fig. 7 shows the workflow of MoLi. MoLi is a framework built on Coq. MoLi supports the verification of C language (the Coq model of C language follows an existing framework [101]). Users specify the specification (§5.2), and then follow the inference rules provided by MoLi to manually perform the Hoare-style verification (§5.3). The framework is sound, which ensures that the proof implies the termination-preserving refinement.

5.2 Specification in MoLi

A specification includes the abstraction, rely-guarantee conditions, invariants, definite actions, and the layer function, as defined next.

Abstraction. The abstraction includes the abstract representation of the concrete state and abstract operations (*Aop*) on it. An abstraction can hide implementation details, which is easier to check and less error-prone than the concrete implementation. MoLi provides a specification language, which allows users to write non-atomic abstract operations. The language has standard commands such as **while** and **if**, and is suitable for expressing abstract operations: (1) the language's state includes the abstract state and a local *abstract stack*, which maps variables to *abstract values*, e.g., lists; (2) the language supports user-supplied primitives that model atomic transitions of abstract state; (3) it also supports atomic block $\langle C \rangle$ where C executes atomically (see §6.1 for an example).

State. In MoLi, state includes not only the concrete state accessed by the implementation, but also several auxiliary parts, i.e., the specfication language's state, tokens, and ghost state. MoLi uses *tokens* as local state to ensure termination (as explained in §5.3). Users may also introduce *ghost state*, which exists only in the abstract model, to assist verification.

Rely/guarantee conditions (R/G) and invariants (I). R and G

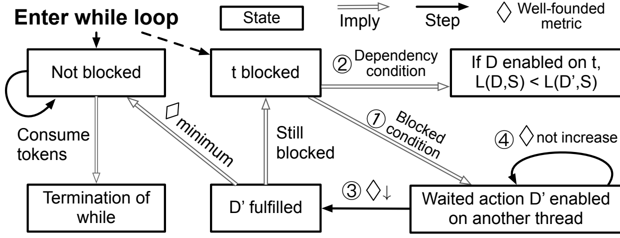


Figure 8: **Logic for termination of a while loop.** Users specify the number of *tokens* and a *well-founded metric* $\Diamond$. In the non-blocking case, tokens strictly decrease for each iteration. In the blocking case, $\Diamond$ strictly decreases whenever a waited definite action ($\mathcal{D}'$) executes. When $\Diamond$ decreases to its minimum, the thread is no longer blocked. The dependency condition enforces an enabled definite action ($\mathcal{D}$) waits only for a higher-numbered one ($\mathcal{D}'$).

define the allowed state transitions and are checked at each step: one checks that (1) the current thread's state transitions satisfy G, (2) the pre-/post-conditions of a command stay true, even when a concurrent thread changes the state, as long as this change is allowed by R, and (3) I specifies an invariant on state, which must remain true under all transitions. R/G and I define the concurrency protocol. Previous efforts [35, 101] provide more detail.

Definite action and layer function. A *definite action* describes a state transition, written $P \rightsquigarrow Q$, which means that, once assertion P is true, (1) Q will eventually hold, and (2) P is preserved by both current and environmental thread until Q holds. The second condition ensures that P may not become false until the definite action is fulfilled. A definite action is called **enabled** when P holds. A *layer function* $\mathcal{L}$ takes a definite action and a state S to return a layer if defined.

5.3 Verification in MoLi

The judgement $\mathcal{L},\mathcal{D},R,G,I \vdash \{P \wedge Aop\}C\{Q \wedge \texttt{skip}\}$ means (1) starting from the precondition P, the operation C must terminate to reach the postcondition Q, and meanwhile (2) an abstract operation terminates (from Aop to `skip`), simulating the concrete operation C. Here, both P and Q imply the invariant I and specify the consistency relation between the concrete and abstract state. MoLi provides inference rules to help users prove the judgement holds. A top-level rule, called the OBJECT rule, proves the well-formedness of the specification and the judgement for each method of the object. The OBJECT rule establishes termination of the implementation and abstraction, and a termination-preserving refinement between them. To verify each method, users follow inference rules of C language statements to step through the program.

Most rules for C language, e.g., **sequence** and **if** rules, are standard and similar to previous work [35, 101]. We explain the WHILE rule (see Fig. 8 and the simplified rule below).

Termination of a* while *loop. The WHILE rule requires establishing the judgement of the loop body (see the first line of the rule). Assuming the loop body can terminate, we check whether the loop is blocked by other threads. Based on that, the logic uses different strategies to ensure termination, i.e., consumption of tokens in the non-blocking case and decrease of metric by executing definite actions in the blocking case.

$$\frac{\begin{array}{l}\mathcal{L},\mathcal{D},R,G,I \vdash \{P \wedge B\}C\{P\} \\ \text{if not blocked, consume tokens} \\ \text{if blocked, prove conditions ①-④} \\ \text{(see ①-④ in the text and Fig. 8)}\end{array}}{\mathcal{L},\mathcal{D},R,G,I \vdash \{P\}\textbf{while } (B)\{C\}\{P \wedge \neg B\}} \text{ (WHILE)}$$

If the **while** loop is not blocked, we must show the **while** loop can iterate for only a finite number of rounds. Following previous efforts [46, 64], we require that each iteration consumes resources called *tokens*. Users specify the number of tokens before the **while** loop. The loop terminates after exhausting its tokens. For instance, the traversal **while** loop in Fig. 2a can iterate only a finite number of rounds, bounded by the length of `path`, which specifies the number of tokens.

In contrast, if a thread t is blocked, its termination relies on definite actions of other threads. Users define a well-found *metric* (i.e., that cannot infinitely decrease) for the **while** loop. Whenever a definite action happens, the metric must decrease. When the metric decreases to its minimum, the thread is no longer blocked. Specifically, users prove the following.

① **Blocked condition**: if t is blocked, t must be waiting for some definite action $\mathcal{D}'$ (to be specified) to happen, which is enabled on another thread.

② **Dependency condition**: if t has an enabled definite action $\mathcal{D}$, its layer must stay *less* than that of $\mathcal{D}'$ until $\mathcal{D}'$ is fulfilled.

③ **Metric decrease**: whenever $\mathcal{D}'$ is fulfilled, the metric decreases.

④ **Metric non-increase**: the metric never increases.

Let us prove the termination of ticket lock in Fig. 2b under the above conditions. Suppose a thread t is blocked in the **while** loop of `lock(cur)`. A precondition is that t does not own `cur`. Then we can prove ① t waits for the definite release of `cur`, which is enabled on another thread. If t has enabled definite actions, e.g., t has acquired other locks, the layer function must have captured these lock dependencies. With the layer function, users prove ② the layers of owned locks are stably less than `cur` until t's ticket equals `cur.owner`. One can specify the well-founded metric as `t.i-cur.owner`, which measures the distance from t's ticket to the current owner. ③ The metric decreases whenever an environment thread increases `cur.owner`, and ④ never increases. When the metric decreases to zero, the loop terminates.

Lock abstraction. We can prove the termination of lock implementations with the WHILE rule. But MoLi follows recent work [65], and provides a specialized inference rule for fair locks (e.g., ticket lock) to ease the burden. The abstract state of a lock is an integer `L`, whose value is either `0` when available or `t` when owned by thread `t`.

The reasoning of a lock operation is similar to the WHILE rule, only with a different metric non-increase condition: the metric never increases under transitions where the lock keeps being unavailable. We allow the metric to increase when the lock is available, because a fair lock guarantees that, if the lock becomes available for a finite number of times, a thread will eventually become the first to the lock.

Definiteness of definite actions. The WHILE rule assumes that a definite action will be fulfilled once enabled. To meet this assumption, the OBJECT rule checks two things.

- **Well-formedness**: all steps preserve an enabled definite action of some thread except that the thread itself may fulfill the definite action.
- **Postcondition restriction**: the postcondition of an operation implies no enabled definite actions.

MoLi requires an enabled definite action to be fulfilled because the following conditions hold.

- When an operation terminates, it must be fulfilled according to the **postcondition restriction**.
- Whenever the thread is blocked in a **while** loop, the proof of **while** loop ensures the termination by only relying on higher-layer definite actions according to the **dependency condition**; intuitively, this will not introduce unsound circular dependencies, so those higher-layer definite actions can indeed be fulfilled.
- The thread is proved to terminate, and thus fulfills the definite action itself due to **well-formedness**.

Soundness. After users prove each operation by following the inference rules, the framework constructs an *overall termination metric* for each thread, and shows that the overall metric strictly decreases. The overall metric combines several metrics that users have provided in their proofs.

- The totally ordered layering of definite actions ensures that the waits-for graph shrinks until the thread's waited definite action is not blocked.
- The well-founded metric for a **while** loop measures the progress from the execution of the waited definite action, until the **while** loop is not blocked.
- The **while** loop tokens count down the iterations.

We have formally proved the following main theorem (see [100] for the full pen-and-paper proof).

```
// Error handling omitted
// Def of Inode omitted
struct inodelock{
  Inode *inode;
  int refcount;
  lock lk;
}

def getilock(ilock):
  ⟨ilock->refcount++⟩

def putilock(ilock):
  ⟨ilock->refcount--;
  if(ilock->refcount==0){
    free(ilock);
  }⟩

def traversal(cur,path):
  local i=0, ret;
  getilock(cur);
  while(path[i]){
    ret=lookup(cur,path[i]);
    cur=ret;i++;}
  return cur;

def lookup(par,name):
  local child;
  lock(par);
  child=find(par,name);
  getilock(child);
  unlock(par);
  putilock(par);
  return child;
```

Figure 9: **Reference counting and traversal in RefFS.** By holding a refcount in hand, `lookup` requests `par` without worrying `par` has been freed.

Theorem 1 (Main Theorem)

Given the implementation and abstraction, if there exist *rely*/*guarantee* conditions, an *invariant*, *definite actions* and a *layer function*, such that for each operation C of the implementation and corresponding abstract operation Aop, the judgment $(\mathcal{L}, \mathcal{D}, R, G, I \vdash \{P \wedge Aop\} C \{Q \wedge \texttt{skip}\})$ holds w.r.t. the pre-/post-conditions by applying inference rules, then the implementation ensures termination and is a termination-preserving refinement of the abstraction.

6 Design and Verification of RefFS

RefFS is a concurrent in-memory file system running on FUSE. We verify its interfaces that manipulate the file system structure (e.g., `mkdir`/`mknod`, `rmdir`/`unlink` and `rename`), and that perform input and output to files (e.g., `open`, `read`, `write` and `close`). This covers most common operations.

6.1 Implementation and Abstraction

RefFS reuses code from a previously verified concurrent file system, AtomFS [101], e.g., the internal functions that operate on directories and files. However, RefFS uses reference counting (refcounting) for traversal, which is more fine-grained and provides better performance than lock coupling used by AtomFS. Consequently, the rename implementation, file-descriptor-based interfaces and abstraction of RefFS are different from AtomFS, as explained below. We also prove liveness guarantee of RefFS.

Refcounting. In Fig. 9, struct `inodelock` wraps over an inode with a lock for protecting the struct, and a reference count for resource reclamation. The `refcount` field counts the references to the struct. `getilock` increases `refcount`

by one. putilock decreases refcount by one and frees the struct when refcount becomes zero. In other words, refcount can prevent use-after-free as long as a thread still holds a refcount that other threads may not decrease (i.e., refcount>0). We use the atomic block notation $\langle C \rangle$ to represent that command C executes atomically. This is achieved with locks (not shown in the code).

To correctly implement refcounting, RefFS takes care of the following aspects.

① **Initialization**: when allocated, an inode's refcount is initialized to 1, marking that there is one reference in its parent's directory entry; this refcount is decreased when the inode is removed from its parent (the root's initial reference cannot be decreased).

② **Reference increase**: a thread can increase the refcount of an inode when it already holds its refcount (a thread can directly increase the refcount of root).

③ **Reference decrease**: a thread can decrease a refcount that belongs to the thread, e.g., the thread has increased the refcount previously.

④ **Reference counting**: each thread decreases the refcount that it no longer needs, and the value of refcount equals the number of all references combined.

③ is the key to stablizing refcount>0 and preventing use-after-free, because if a thread has increased a refcount, other threads may not arbitrarily decrease it. The above refcounting protocols are formalized as rely/guarantee conditions and invariants, and proved for RefFS.

Let us see how the traversal function in Fig. 9 obeys and leverages the above protocols. The precondition specifies that either cur is root, or the current thread holds a reference to cur, so that traversal can increase the refcount of cur (②). The code invokes lookup for each item of the path. lookup can request par's lock without worrying that par is freed, because the code holds par's refcount (③). After having found the child in par, the thread holds a reference to child, due to owning the directory entry of child in par. Therefore, it can increase the refcount of child (②). It then releases par's lock and refcount (③ and ④).

Refcounting provides the following performance benefits.

- During path traversal, there are intervals where an operation has searched the parent directory and is about to lookup the child directory. The operation does not need to nestedly lock parent and child to protect the child from being freed during the interval. Instead, traversal prevents use-after-free by holding a refcount of child. This creates more parallelism because concurrent operations can bypass each other during path traversal.
- Some operations use a file descriptor (FD) to directly access an inode, and leverage refcounting to prevent use-after-free. For instance, open increases the refcount of the target inode, and returns the inum as FD, thereafter, read and write operations can directly access the inode.

```
def rename(src,sn,dst,dn):
1 ... //Traverse common
  path of src and dst
2 rel=pathrel(src,dst);
3 if(rel!=0){
4   lock(rename_mutex);}
5 ... //Traverse to get
  src and dst directories
6 if (rel==0){
7   lock(sdir);
8 } else if(rel==1){
9   lock(sdir);
10   lock(ddir);
11 } else {
12   lock(ddir);
13    TDep:=(ddir,sdir)
14   lock(sdir); }
15 ...
16 //If dn exists in ddir
17  TDep:=(sdir,dchild)
18 lock(dchild);
19  TDep:=None
20 ...
```

Figure 10: **Highlighting lock acquisitions of RefFS rename.** pathrel(src,dst) returns 0 if src equals dst, returns 1 if src is a proper prefix of dst, and returns -1 in other cases. Depending on the result, the code decides whether to acquire rename_mutex after traversing the common path of src and dst. Code in gray boxes updates ghost state.

Refcounting brings challenges for liveness reasoning because we need to consider more intricate lock dependencies of FS operations. Specifically, when using lock coupling (concurrent operations cannot bypass each other), an operation that starts the traversal first would not be blocked. This is not the case with refcounting: an operation may be blocked by another operation that bypasses it, or by a file-descriptor-based operation. Nevertheless, MoLi's modular verification approach effectively manages the complexities.

Rename implementation. In Fig. 10 (ignore the code in gray boxes for now), src/dst is the path to the old/new parent, and sn/dn is the name of source/target. rename first traverses the common path of src and dst (using traversal in Fig. 9). After getting the reference of their least common ancestor, the algorithm decides whether this is a cross-directory rename by comparing src and dst. If they are not equal (or cross-directory), the code acquires rename_mutex. The code then traverses the *remaining* path to get the references of the old and new parents. Holding rename_mutex ensures that the relative position between the two directories will not be changed by concurrent renames. Therefore, one may use the path arguments to know whether they are ancestors to each other, e.g., if src is a proper prefix of dst, this means that sdir is an ancestor to ddir. The code acquires the lock of the ancestor first. Otherwise, it will acquire them in a default order. If the target (i.e., dchild) already exists, the code acquires its lock.

Non-atomic abstraction. The abstraction of RefFS consists of abstract file system state and abstract operations. The abstract file system, called AFS, is a mapping of inode number (inum) to an *abstract inode*. An *abstract inode* is either a file (a list of bytes) or a directory (a mapping of name to inum).

For an operation that needs path traversal, the correspond-

```
def MKDIR(path,n):                       4   if(tmp==NULL){
1 local cur=root,tmp,i=0;                5     return -1;}
2 while(path[i]){                        6   cur=tmp;i++⟩}
3  ⟨tmp=lookup(cur,path[i]);             7 ret do_mkdir(cur,n);
```

Figure 11: **Abstract operation `MKDIR` of RefFS.** `MKDIR` consists of a series of atomic directory lookups (line 3-6) and an atomic critical section (line 7).

ing abstract operation hides as much implementation detail as possible (e.g., hiding refcount, locks and internal data structure), and guarantees atomic directory lookups and an atomic fulfillment of the operation after locating the target inodes. Fig. 11 shows the abstract operation `MKDIR`. `lookup` and `do_mkdir` primitives model atomic transitions of the abstract file system. We simplify the abstract operation by grouping the loop body (lines 3-6) into an atomic block.

6.2 Verification of RefFS

We use ghost state, the temporary dependency `TDep`, to assist the proof, as introduced in §4.3. In Fig. 10, code in gray boxes updates it. `TDep` is set to `(ddir, sdir)` before line 14 to represent the upcoming lock dependency between them. `TDep` may be updated to `(sdir,dchild)` before line 18 to establish the lock dependency from `sdir` to `dchild`. `TDep` is reset to `None` after line 18.

The termination proofs include the checks in the OBJECT rule and the termination proof of locks and while loops. Since while loops are not blocked in RefFS, their proofs are trivial.

Definiteness check. The well-formedness of definite releases holds because once a thread holds a lock, all steps keep the fact true until the thread releases the lock itself. The postcondition of each operation specifies that the thread does not own any lock, so it must be the case that definite releases are fulfilled before the thread reaches the end.

Proof for locks. When thread t is blocked in `lock(L)`, t waits for the definite release of `L` (specified as $\mathcal{D}'$). The metric is 0 when `L` is available and 1 otherwise. The following holds. (1) Blocked condition: some thread t' holds `L`, so $\mathcal{D}'$ is enabled on t'. (2) Dependency condition: the dynamic layering ensures that for any lock that t owns, its layer is stably lower than `L`. (3) Metric decrease: when `L` is released, the metric decreases. (4) Metric non-increase for the lock: when `L` is not available, the metric stays 1 and never increases.

7 Evaluation

This section empirically answers several questions:

- Can RefFS provide good performance for real-world applications, and does reference-counting perform better compared with lock-coupling (§7.1)?

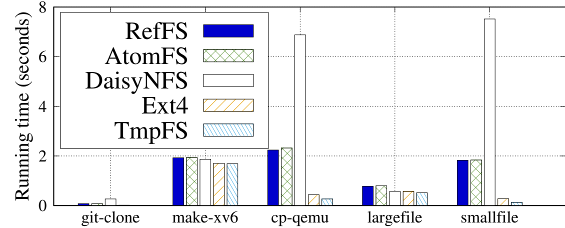


Figure 12: **Applications.** The figure shows the running time of different applications, i.e., git, make and cp. Largefile operates on a big file with 10MB. Smallfile operates on 10K files with 1KB size.

- Compared with previous work, how modular are the termination proofs using MoLi (§7.2)?
- How much is the verification effort (§7.3)?
- Can MoLi help eliminate bugs in practice (§7.4)?

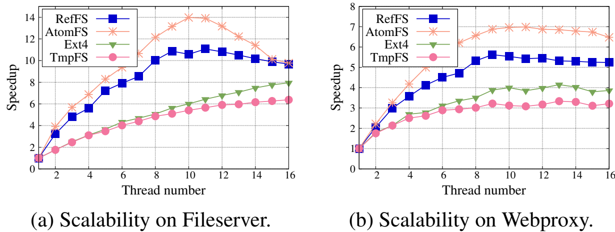
7.1 Performance Evaluation

Experimental setup. We run all of the experiments on a server machine (AWS EC2 i3.metal instance) with 72 cores (2.3GHz), 512GB DRAM, and a local 15,200GB SSD (8 disks) running Linux 5.15.8. We limit our experiments to one 36-core socket to avoid variability. We compare the performance of RefFS to the widely-used disk file system ext4 [84], the verified concurrent file system AtomFS [101], the verified concurrent NFS server DaisyNFS [20], and an in-memory file system tmpfs. All the file systems use in-memory storage.

Application performance. RefFS is complete enough to run many kinds of realistic software, including Vim [85] and GCC [37]. To evaluate application performance, we select two microbenchmarks and three application workloads: LFS microbenchmark [71, 77], cloning the git repository of xv6-public, compiling the sources of the xv6 file system with a makefile and copying the source code of qemu. These workloads are also used by previous verified file systems [22, 101]. The application workloads use only a single core.

In Fig. 12, RefFS achieves similar results to AtomFS, and better performance than DaisyNFS in most cases, due to the network I/O overhead of DaisyNFS. The worse performance of RefFS compared with tmpfs and ext4 is due mainly to the lack of fine-grained optimizations, e.g., highly optimized path traversals and optimized structures for data and metadata. Running RefFS with FUSE also introduces overhead. These issues can be overcome by more engineering.

File system scalability. We adopt two commonly used workloads in Filebench [36], Fileserver and Webproxy, to measure the scalability of RefFS. We evaluate with 16 cores and increase the number of threads used in the workloads. We do not evaluate DaisyNFS because its in-memory disk can only



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

Figure 13: **Scalability of RefFS.** The overall scalability of RefFS is similar to AtomFS.

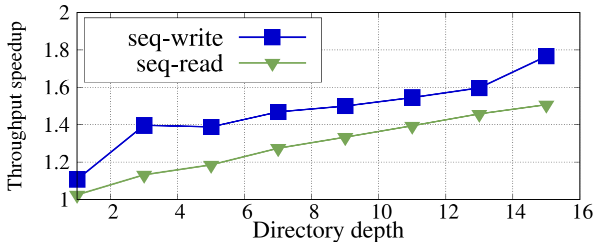


Figure 14: **Speedup of RefFS over AtomFS.** RefFS achieves higher speedup over AtomFS as the directory depth increases in LargeFile benchmarks.

support about 400MB of space, while the scalability tests consume more than 3GB in many cases.

The speedup results are in Fig. 13. RefFS can scale up to 9 cores. AtomFS shows a similar scalability, but the actual throughput (not shown in figure) of RefFS is better than AtomFS at all loads (1.08–1.43x higher in Fileserver and 1.03–1.32x higher in Webproxy). RefFS's performance is worse than ext4 and tmpfs, as expected.

Other benefits of reference counting. AtomFS uses lock coupling to traverse the path even for read and write operations. In RefFS, reference counting allows read and write to directly access the inode because `open` has increased the inode's reference. To show the benefit of this, we evaluate RefFS and AtomFS using LargeFile benchmarks under different depths of directories. In Fig. 14, with the depth increase, the speedup of RefFS over AtomFS becomes higher in both seq-write and seq-read tests in LargeFile.

Table 1: Lines of Coq proof for verifying RefFS.

Component	LOC	Component	LOC
Abstraction and aops	.1K	Invariant	.7K
Rely/guarantee	.4K	Code	.4K
Layered definite releases	.1K	Proof	32K
Total			**33.7K**

7.2 Modularity of Termination Proofs

Dynamically layered definite releases allow to verify each lock separately, and reuse the termination proofs for all each lock statement. To evaluate the benefits, we also use the non-layered definite actions (LiLi's approach in §4.1) to verify the termination of a lock statement in RefFS. The crucial difference is that the non-layered approach has to reason globally about the dependency chain.

Suppose there is a lock dependency chain of `root` → `B` → `C` as shown in Fig. 4a, where each thread in the chain owns a lock and requests the next lock except the last thread. Now suppose t wants to acquire `root`'s lock. To prove t's progress, LiLi's non-layered approach needs to prove how the chain gets shorter until t can acquire `root`. The following complexities exist.

In LiLi's approach, one must specify actions that can definitely happen on their own. In this case, t first waits for the lock release of `C`. Defining the action needs following the dependency chain, which requires a proof that the chain is free of cycles. Proving this fact in a general situation requires establishing global invariants, adding to the proof burden. The resulting definite action is very fine-grained and less intuitive.

Furthermore, to show the progress created by the fine-grained definite action, one should define a decreasing metric. However, the definition of the metric is unavoidably complex. Defining it as the length of the dependency chain does not work. Because after lock `C` is released, the thread that owns `B` acquires `C`, and may go on requesting other locks, thus getting involved in a even longer dependency chain. As a result, defining this metric requires considering a thread's local progress (e.g., its remaining steps), and the length of the dependency chain, with the former prioritized before the latter (we omit further detail). This poses a significant proof burden in the metric-decrease and metric-non-increase proofs.

By contrast, the specification and termination proof for a lock statement with our layered approach (see §6.2) focus only on the lock. The extra proof burden is the dependency condition, which requires to show any owned locks are lower-numbered than the lock to acquire. This proof is usually trivial with dynamic layering. Code proofs in Coq have confirmed the analysis—the non-layered approach needs 3K LOC while the layered approach requires less than 0.4K LOC.

7.3 Verification Evaluation

Verification effort. MoLi reuses the code from CRL-H [101] framework, including the support for C language and concurrency reasoning. MoLi's extension mainly devotes to the logic for termination and the model of non-atomic abstract operations, which is about 3K LOC. Table 1 shows the lines of proof for verifying RefFS. RefFS also reuses AtomFS's internal functions inside the critical section and their proofs, except that now we also verify termination.

Trusted computing base and tests. Our work has some trusted parts. The abstraction of RefFS is trusted. VFS, FUSE, C compiler, C implementation of a lock and memory allocator of glibc are trusted. Termination proof assumes a fair scheduler and a sequentially consistent hardware model. We also test RefFS with xfstests, a comprehensive file system testing suite, which reports no bugs.

Limitations. Our prototypes of MoLi and RefFS still have limitations. Currently, RefFS is an in-memory file system and does not consider crashes. In general, the reasoning for termination is orthogonal to crashes because the recovery procedure will restore the state to re-execute the code. Hence, the termination proof still applies to the non-crash setting. When a crash happens in the middle of a program, what MoLi does not consider is the termination of the recovery procedure, whose precondition is the crashed state. To support crash safety, one may combine the techniques in DaisyNFS [18–20].

MoLi does not support reasoning about termination in the presence of interrupts or exceptions. Similar to crash safety, supporting them requires considering intermediate states.

RefFS has simplified read access to use exclusive locks. Nevertheless, we can use read-write locks in RefFS and reason with their implementations in MoLi.

7.4 Bug Discussion

We discuss whether MoLi can help find practical bugs. Non-concurrent termination bugs such as logic and low level programming errors [59, 83] will fail the proof, because one cannot define a well-founded metric that decreases for each iteration. In AA-deadlocks [25,58,72], when a thread requests for a lock again, the layer of the waited action (definite release of the lock by other threads) is not larger than that of enabled action (definite release of the lock by the current thread), which will violate the dependency condition. In deadlocks caused by nested blocking [6, 23, 39, 97], proof authors either fail to define the layers, or define the wrong layers, later finding that the layers cannot pass the dependency condition proof.

For deadlocks with dynamic orders [3, 8, 55, 96], MoLi allows defining state-dependent dynamic layers to precisely represent such orders. Therefore, such bugs can be discovered during proofs. For deadlocks that involve ad-hoc synchronization [24,52,69], MoLi's general notion of definite actions can specify and verify them, similar to the bug types above.

These bug patterns also exist in other domains, e.g., memory management [27] and network [45] subsystems in an OS, database and web applications [68]. Therefore, we believe MoLi is also applicable to these domains.

8 Related Work

Starting from the seminal work of seL4 [57], these years have witnessed tremendous progress on the verification of systems, including operating systems [40, 73, 80], distributed systems [43, 78, 92], file systems [21, 22, 47, 79] and many others [26, 28, 63, 81, 88, 89]. Yet, only few of them guarantee systems' liveness, and none of the proposed frameworks could be used for concurrent file systems.

VSync [74] relies on a special *await loops* shown in lock implementations, and proposes *await model checking* to automatically verify the termination of lock primitives, even under weak memory models. It does not support general while loops. VSync relies on a specific client library for correctness; the client library may still not cover all corner cases, especially for large-scale systems such as file systems.

CCAL [41] has been used to verify the termination of an MCS lock [56] by organizing the implementation into layers. However, it does not provide a program logic to guide the proofs, so it is unclear how CCAL can be used to prove the termination of concurrent file systems.

Ironfleet [43] verifies distributed systems with a blend of TLA and Hoare style automated verification. However, this methodology does not have the power for concurrency and termination verification in file systems. Ironfleet's reduction approach for concurrency verification relies on implementation being atomic, but file systems, e.g., RefFS, are not atomic. On termination, Ironfleet does not consider blocking and dynamic lock dependencies as shown in file systems.

Reference counting, a widely used technique in Linux, has also caused many severe bugs [44]. Various methods (e.g., invariant-based [33,34] and anti-pattern based [44]) have been proposed to detect these bugs. Although effective in practice, they still suffer from false positives and false negatives. Our work verifies the correctness of refcounting by showing the implementation using it can refine an abstraction where its details are hidden.

9 Conclusion

This paper has presented MoLi for verifying concurrent file systems. It supports the dynamically layered definite releases specification, with which we verify RefFS, the first modularly verified concurrent file system with termination guarantee. We also for the first time formally prove the correctness of directory locking scheme in Linux VFS. The formal specification has helped us uncover real Linux bugs in practice.

Acknowledgments

We sincerely thank the anonymous reviewers for their valuable comments. We are especially grateful to our shepherd Marc Shapiro, whose suggestions significantly improved this paper. We thank Xinyu Feng and Hongjin Liang for discussions on the metatheory of MoLi. This work is supported by the National Natural Science Foundation of China (No. 61925206 and 62132014). Haibo Chen (haibochen@sjtu.edu.cn) is the corresponding author.

References

[1] Paulo Alcantara. cifs: fix potential deadlock in cache_refresh_path(). https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=9fb0db40513e27537fde63287aea920b60557a69, 2023.

[2] Bowen Alpern and Fred B. Schneider. Recognizing safety and liveness. *Distrib. Comput.*, 2(3):117–126, sep 1987.

[3] Josef Bacik. btrfs: drop path before adding new uuid tree entry. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=9771a5cf937129307d9f58922d60484d58ababe7, 2020.

[4] Josef Bacik. btrfs: fix lockdep splat in add_missing_dev. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=fccc0007b8dc952c6bc0805cdf842eb8ea06a639, 2020.

[5] Josef Bacik. btrfs: fix potential deadlock in the search ioctl. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=a48b73eca4ceb9b8a4b97f290a065335dbcd8a04, 2020.

[6] Josef Bacik. btrfs: move the chunk_mutex in btrfs_read_chunk_tree. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=01d01caf19ff7c537527d352d169c4368375c0a1, 2020.

[7] Josef Bacik. btrfs: open device without device_list_mutex. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=18c850fdc5a801bad4977b0f1723761d42267e45, 2020.

[8] Josef Bacik. btrfs: unlock to current level in btrfs_next_old_leaf. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=0e46318df8a120ba5f1e15210c32cfab33b09f40, 2020.

[9] Josef Bacik. btrfs: exclude mmaps while doing remap. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=8c99516a8cdd15fe6b64a12297a5c7f52dcee9a5, 2021.

[10] Josef Bacik. btrfs: fix lockdep splat with reloc root extent buffers. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=b40130b23ca4a08c5785d5a3559805916bddba3c, 2022.

[11] Josef Bacik. btrfs: unlock locked extent area if we have contention. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=9e769bd7e5db5e3bd76e7c67004c261f7fcaa8f1, 2022.

[12] Stephanie Balzer, Bernardo Toninho, and Frank Pfenning. Manifest deadlock-freedom for shared session types. In *ESOP*, pages 611–639, 2019.

[13] Johann Blieberger, Bernd Burgstaller, and Robert Mittermayr. Static detection of livelocks in Ada multitasking programs. In *Proceedings of the 12th International Conference on Reliable Software Technologies*, Ada-Europe'07, page 69–83, Berlin, Heidelberg, 2007. Springer-Verlag.

[14] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: Preventing data races and deadlocks. *SIGPLAN Not.*, 37(11):211–230, nov 2002.

[15] Jacob Burnim, Nicholas Jalbert, Christos Stergiou, and Koushik Sen. Looper: Lightweight detection of infinite loops at runtime. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, page 161–169, USA, 2009. IEEE Computer Society.

[16] Miao Cai, Hao Huang, and Jian Huang. Understanding security vulnerabilities in file systems. In *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '19, page 8–15, New York, NY, USA, 2019. Association for Computing Machinery.

[17] Michael Carbin, Sasa Misailovic, Michael Kling, and Martin C. Rinard. Detecting and escaping infinite loops with jolt. In *Proceedings of the 25th European Conference on Object-Oriented Programming*, ECOOP'11, page 609–633, Berlin, Heidelberg, 2011. Springer-Verlag.

[18] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying concurrent, crash-safe systems with perennial. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*,

SOSP '19, page 243–258, New York, NY, USA, 2019. Association for Computing Machinery.

[19] Tej Chajed, Joseph Tassarotti, Mark Theng, Ralf Jung, M. Frans Kaashoek, and Nickolai Zeldovich. Gojournal: a verified, concurrent, crash-safe journaling system. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 423–439. USENIX Association, July 2021.

[20] 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, 2022.

[21] Haogang Chen, Tej Chajed, Alex Konradi, Stephanie Wang, Atalay undefinedleri, 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.

[22] 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.

[23] Zhihao Cheng. ubifs: Fix deadlock in concurrent bulk-read and writepage. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=f5de5b83303e61b1f3fb09bd77ce3ac2d7a475f2, 2020.

[24] Zhihao Cheng. ubifs: Fix deadlock in concurrent rename whiteout and inode writeback. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=afd427048047e8efdedab30e8888044e2be5aa9c, 2021.

[25] Zhihao Cheng. ubifs: Fix aa deadlock when setting xattr for encrypted file. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=a0c51565730729f0df2ee886e34b4da6d359a10b, 2022.

[26] Rafael Lourenco de Lima Chehab, Antonio Paolillo, Diogo Behrens, Ming Fu, Hermann Härtig, and Haibo Chen. Clof: A compositional lock framework for multi-level numa systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 851–865, New York, NY, USA, 2021. Association for Computing Machinery.

[27] Hugh Dickins. mm: lock newly mapped vma with corrected ordering. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=1c7873e3364570ec89343ff4877e0f27a7b21a61, 2023.

[28] Haoran Ding, Zhaoguo Wang, Zhuohao Shen, Rong Chen, and Haibo Chen. Automated verification of idempotence for stateful serverless applications. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 887–910, Boston, MA, July 2023. USENIX Association.

[29] Linux documentation. Kernel subsystem documentation » filesystems in the linux kernel » directory locking. https://www.kernel.org/doc/html/latest/filesystems/directory-locking.html, 2023. Referenced December 2023.

[30] Linux documentation. Kernel subsystem documentation » filesystems in the linux kernel » pathname lookup. https://www.kernel.org/doc/html/latest/filesystems/path-lookup.html, 2023. Referenced December 2023.

[31] Linux documentation. Locking in the kernel » runtime locking correctness validator. https://www.kernel.org/doc/html/latest/locking/lockdep-design.html, 2023. Referenced April 2023.

[32] Emanuele D'Osualdo, Julian Sutherland, Azadeh Farzan, and Philippa Gardner. Tada live: Compositional reasoning for termination of fine-grained concurrent programs. *ACM Trans. Program. Lang. Syst.*, 43(4), nov 2021.

[33] Michael Emmi, Ranjit Jhala, Eddie Kohler, and Rupak Majumdar. Verifying reference counting implementations. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, 2009.

[34] Michael Emmi, Ranjit Jhala, Eddie Kohler, and Rupak Majumdar. Verifying reference counting implementations. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009,*, TACAS '09, page 352–367, Berlin, Heidelberg, 2009. Springer-Verlag.

[35] Xinyu Feng. Local rely-guarantee reasoning. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, page 315–327, New York, NY, USA, 2009. Association for Computing Machinery.

[36] Filebench. Filebench, 2019.

[37] GNU. Gcc, the gnu compiler collection. https://www.gnu.org/software/gcc/, 2019. Referenced April 2019.

[38] Google. syzkaller - kernel fuzzer, 2023.

[39] Andreas Gruenbacher. gfs2: Fix deadlock between gfs2_{create_inode,inode_lookup} and delete_work_func. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=dd0ecf544125639e54056d851e4887dbb94b6d2f, 2020.

[40] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. Certikos: An extensible architecture for building certified concurrent OS kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 653–669, Savannah, GA, 2016. USENIX Association.

[41] Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan (Newman) Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. Certified concurrent abstraction layers. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, page 646–661, New York, NY, USA, 2018. Association for Computing Machinery.

[42] Chunhai Guo. erofs: avoid infinite loop in z_erofs_do_read_page() when reading beyond eof. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=8191213a5835b0317c5e4d0d337ae1ae00c75253, 2023.

[43] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. Ironfleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 1–17, New York, NY, USA, 2015. Association for Computing Machinery.

[44] Liang He, Purui Su, Chao Zhang, Yan Cai, and Jinxin Ma. One simple api can cause hundreds of bugs an analysis of refcounting bugs in all modern linux kernels. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 52–65, New York, NY, USA, 2023. Association for Computing Machinery.

[45] Allison Henderson. net:rds: Fix possible deadlock in rds_message_put. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=f1acf1ac84d2ae97b7889b87223c1064df850069, 2024.

[46] Jan Hoffmann, Michael Marmar, and Zhong Shao. Quantitative reasoning for proving lock-freedom. In *Proceedings of the 2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '13, page 124–133, USA, 2013. IEEE Computer Society.

[47] Atalay Ileri, Tej Chajed, Adam Chlipala, Frans Kaashoek, and Nickolai Zeldovich. Proving confidentiality in a file system using disksec. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 323–338, Carlsbad, CA, 2018. USENIX Association.

[48] Jules Jacobs, Stephanie Balzer, and Robbert Krebbers. Connectivity graphs: a method for proving deadlock freedom based on separation logic. *Proc. ACM Program. Lang.*, 6(POPL), jan 2022.

[49] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, oct 1983.

[50] D. Jones. Trinity: A linux system call fuzz tester, 2019.

[51] Horatiu Jula, Daniel M Tralamazza, Cristian Zamfir, George Candea, et al. Deadlock immunity: Enabling systems to defend against deadlocks. In *OSDI*, volume 8, pages 295–308, 2008.

[52] Jan Kara. ext4: fix deadlock with fs freezing and ea inodes. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=46e294efc355c48d1dd4d58501aa56dac461792a, 2020.

[53] Jan Kara. fs: Lock moved directories. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=28eceeda130f5058074dd007d9c59d2e8bc5af2e, 2023.

[54] Linux kernel stable tree. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/log/, 2024.

[55] Hyeong-Jun Kim. f2fs: compress: fix potential deadlock of compress file. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=7377e853967ba45bf409e3b5536624d2cbc99f21, 2021.

[56] Jieung Kim, Vilhelm Sjöberg, Ronghui Gu, and Zhong Shao. Safety and liveness of mcs lock—layer by layer. In *Asian Symposium on Programming Languages and Systems*, pages 273–297. Springer, 2017.

[57] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 207–220, New York, NY, USA, 2009. Association for Computing Machinery.

[58] Konstantin Komarov. fs/ntfs3: Changing locking in ntfs_rename. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=0ad9dfcb8d3fd6ef91983ccb93fafbf9e3115796, 2022.

[59] Greg Kurz. fuse: Fix infinite loop in sget_fc(). https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=e4a9ccdd1c03b3dc58214874399d24331ea0a3ab, 2021.

[60] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3:125–143, 1977.

[61] K. Rustan M. Leino and Peter Müller. A basis for verifying multi-threaded programs. In *European Symposium on Programming*, 2009.

[62] K. Rustan M. Leino, Peter Müller, and Jan Smans. Deadlock-free channels and locks. In *ESOP*, pages 407–426, 2010.

[63] Xupeng Li, Xuheng Li, Christoffer Dall, Ronghui Gu, Jason Nieh, Yousuf Sait, and Gareth Stockwell. Design and verification of the arm confidential compute architecture. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 465–484, 2022.

[64] Hongjin Liang and Xinyu Feng. A program logic for concurrent objects under fair scheduling. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, page 385–399, New York, NY, USA, 2016. Association for Computing Machinery.

[65] Hongjin Liang and Xinyu Feng. Progress of concurrent objects with partial methods. *Proc. ACM Program. Lang.*, 2(POPL), dec 2017.

[66] Hongjin Liang, Xinyu Feng, and Zhong Shao. 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)*, CSL-LICS '14, New York, NY, USA, 2014. Association for Computing Machinery.

[67] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. A study of linux file system evolution. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies*, FAST'13, page 31–44, USA, 2013. USENIX Association.

[68] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, page 329–339, New York, NY, USA, 2008. Association for Computing Machinery.

[69] Filipe Manana. btrfs: fix deadlock when cloning inline extent and low on free metadata space. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=3d45f221ce627d13e2e6ef3274f06750c84a6542, 2020.

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

[71] mit pdos. mit-pdos/fscq: Fscq is a certified file system written and proven in coq. https://github.com/mit-pdos/fscq, 2019. Referenced April 2019.

[72] Trond Myklebust. Nfs: Don't deadlock when cookie hashes collide. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=648a4548d622c4ae965058db1a6b5b95c062789a, 2022.

[73] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. Hyperkernel: Push-button verification of an os kernel. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 252–269, New York, NY, USA, 2017. Association for Computing Machinery.

[74] Jonas Oberhauser, Rafael Lourenco de Lima Chehab, Diogo Behrens, Ming Fu, Antonio Paolillo, Lilith Oberhauser, Koustubha Bhat, Yuzhong Wen, Haibo Chen, Jaeho Kim, and Viktor Vafeiadis. Vsync: push-button verification and optimization for synchronization primitives on weak memory models. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 530–545, New York, NY, USA, 2021. Association for Computing Machinery.

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

[76] Bob Peterson. gfs2: fix a deadlock on withdraw-during-mount. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=865cc3e9cc0b1d4b81c10d53174bced76decf888, 2021.

[77] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP '91, page 1–15, New York, NY, USA, 1991. Association for Computing Machinery.

[78] Upamanyu Sharma, Ralf Jung, Joseph Tassarotti, Frans Kaashoek, and Nickolai Zeldovich. Grove: A separation-logic library for verifying distributed systems. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 113–129, New York, NY, USA, 2023. Association for Computing Machinery.

[79] 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, 2016. USENIX Association.

[80] Helgi Sigurbjarnarson, Luke Nelson, Bruno Castro-Karney, James Bornholt, Emina Torlak, and Xi Wang. Nickel: A framework for design and verification of information flow control systems. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 287–305, Carlsbad, CA, 2018. USENIX Association.

[81] Runzhou Tao, Jianan Yao, Xupeng Li, Shih-Wei Li, Jason Nieh, and Ronghui Gu. Formal verification of a multiprocessor hypervisor on arm relaxed memory hardware. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 866–881, New York, NY, USA, 2021. Association for Computing Machinery.

[82] Theodore Ts'o. ext4: add error checking to ext4_ext_replay_set_iblocks(). https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=1fd95c05d8f742abfe906620780aee4dbe1a2db0, 2021.

[83] Theodore Ts'o. ext4: add error checking to ext4_ext_replay_set_iblocks(). https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=1fd95c05d8f742abfe906620780aee4dbe1a2db0, 2021.

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

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

[86] Al Viro. rename(): avoid a deadlock in the case of parents having no common ancestor. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=a8b0026847b8c43445c921ad2c85521c92eb175f, 2023.

[87] Al Viro. rename(): fix the locking of subdirectories. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=22e111ed6c83dcde3037fc81176012721bc34c0b, 2023.

[88] Jiawei Wang, Diogo Behrens, Ming Fu, Lilith Oberhauser, Jonas Oberhauser, Jitang Lei, Geng Chen, Hermann Härtig, and Haibo Chen. BBQ: A block-based bounded queue for exchanging data and profiling. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 249–262, Carlsbad, CA, July 2022. USENIX Association.

[89] Jiawei Wang, Bohdan Trach, Ming Fu, Diogo Behrens, Jonathan Schwender, Yutao Liu, Jitang Lei, Viktor

Vafeiadis, Hermann Härtig, and Haibo Chen. BWoS: Formally verified block-based work stealing for parallel processing. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 833–850, Boston, MA, July 2023. USENIX Association.

[90] Wengang Wang. ocfs2: fix deadlock between setattr and dio_end_io_write. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=90bd070aae6c4fb5d302f9c4b9c88be60c8197ec, 2021.

[91] Yin Wang, Terence Kelly, Manjunath Kudlur, Stéphane Lafortune, and Scott A Mahlke. Gadara: Dynamic deadlock avoidance for multithreaded programs. In *OSDI*, volume 8, pages 281–294, 2008.

[92] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. Verdi: a framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, page 357–368, New York, NY, USA, 2015. Association for Computing Machinery.

[93] Amy Williams, William Thies, and Michael D. Ernst. Static deadlock detection for java libraries. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, ECOOP'05, page 602–629, Berlin, Heidelberg, 2005. Springer-Verlag.

[94] Darrick J. Wong. xfs: fix s_maxbytes computation on 32-bit kernels. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=932befe39ddea29cf47f4f1dc080d3dba668f0ca, 2020.

[95] Darrick J. Wong. xfs: more lockdep whackamole with kmem_alloc*. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=6dcde60efd946e38fac8d276a6ca47492103e856, 2020.

[96] Darrick J. Wong. xfs: fix an abba deadlock in xfs_rename. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=6da1b4b1ab36d80a3994fd4811c8381de10af604, 2021.

[97] Chao Yu. f2fs: compress: fix potential deadlock. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=3afae09ffea5e08f523823be99a784675995d6bb, 2021.

[98] Mo Zou. Re: [PATCH] Documentation: fs: fix directory locking proofs. https://lore.kernel.org/linux-fsdevel/CAHfrynPiUWiB0Vg3-pTi_yC6cER0wYMmCo_V8HZyWAD5Q_m+jQ@mail.gmail.com/, 2023.

[99] Mo Zou. Directory locking proof for Linux VFS. https://ipads.se.sjtu.edu.cn/pub/projects/reffs, 2024.

[100] Mo Zou. The soundness proof of MoLi. https://ipads.se.sjtu.edu.cn/pub/projects/reffs#soundness_proof, 2024.

[101] Mo Zou, Haoran Ding, Dong Du, Ming Fu, Ronghui Gu, and Haibo Chen. Using concurrent relational logic with helpers for verifying the atomfs file system. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 259–274, New York, NY, USA, 2019. Association for Computing Machinery.