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]
[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*

Abstract

RefFS is the first concurrent file system that guarantees both liveness and safety, backed by a machine-checkable proof in Coq. Unlike earlier concurrent file systems, RefFS provably avoids termination bugs such as livelocks and deadlocks, through its innovative introduction and use of the *dynamically layered definite releases* specification. This specification enables handling of *blocking scenarios*, facilitates *modular* reasoning for *nested blocking*, and effectively eliminates the possibility of circular blocking.

The approaches underlying the aforementioned specification are integrated into a framework called MVT[1]. This framework serves as a guide for developers in verifying concurrent file systems. By extending the specification, we further validated the correctness of the locking scheme for 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 could potentially lead to deadlocks of the entire OS[2]. Overall, RefFS achieves better 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 machine-checkable proof of both safety and liveness properties. Ensuring liveness means the operations of RefFS are guaranteed to terminate under the condition that no thread would be starved due to unfair scheduling. The proof rules out a wide range of bugs that are common in concurrent file systems [68], such as deadlocks, livelocks and infinite loops caused by other reasons (e.g., overflow).

Proving the absence of termination bugs is important because they are too subtle to be correctly handled by developers. For instance [3], a task does not directly deadlock with another task by forming a simple ABBA[3] deadlock pattern, but instead, it may deadlock through a complex circular dependency chain involving multiple tasks. A wide range of other termination bugs [52] continue to be reported, indicating potentially more undiscovered termination bugs that pose a threat to our software system. Once triggered, these bugs can lead to serious consequences, such as system hangs [12].

Testing and program analysis techniques (see §2 for more details) have been used to detect (a subclass of) termination bugs. Although effective in practice, they cannot cover all possible cases to guarantee a system's liveness. Formal verification is a promising approach. Researchers have made tremendous progress in concurrent file system verification [17, 104] and liveness verification [42, 55, 77]. However, as we discuss below, none of existing efforts are capable of modularly verifying the liveness of concurrent file systems.

In principle, proving liveness requires a well-foundedness argument [1, 59], i.e., within a finite number of steps, something must happen. For a sequential program, we may use a well-founded metric as a specification to measure its progress [44, 66]. For instance, we can define the metric by calculating the total steps of the program. With each step, the metric must decrease but cannot infinitely decrease. Hence, the program must terminate after running out the metric.

Unfortunately, defining a specification that can account for the progress of a concurrent file system still faces the following challenges. First, the specification should support the reasoning of *blocking scenarios*. For a thread that is blocked in a busy waiting loop (e.g., to acquire a lock), its progress cannot be achieved by its own steps but relies on the steps of *other* threads (e.g., the thread owning the lock). Note that we aim for general busy waiting loops. This should be distinguished with efforts [8, 46] that only recognize lock primitives but miss blocking scenarios from ad-hoc synchronizations, which are also quite common in file systems (see §2).

Second, it is important that the specification allows modular reasoning of *nested blocking*. Consider the following case. An `unlink` operation owns `parent` and requests for `child`. So thread $\mathtt{t}_1$ that tries to acquire `parent` is blocked by `unlink`, which is nestedly blocked by another thread $\mathtt{t}_2$ that owns `child`. Naively, the well-founded metric for $\mathtt{t}_1$'s progress on acquiring `parent` may not only include `unlink`'s steps to release `parent` but also $\mathtt{t}_2$'s steps to release `child`. The latter creates progress for the former, thus contributes to $\mathtt{t}_1$'s progress *indirectly*. This issue becomes more pronounced with more levels of nested blocking and more threads involved, which makes the reasoning not modular.

Last but not least, the specification should help rule out *circular nested blocking* (i.e., deadlock). While an intuitive approach might specify a static order for nested blocking, the complexity of file systems introduces *diverse* and *dynamic* locking orders that are extremely challenging to formally cap-

[1] Modular Verification of Termination
[2] Confirmed by Linux maintainers including Linus Torvalds.
[3] One acquires locks in the order of AB while another in the order of BA.

ture. Specifically, file systems exhibit a wide range of nested blocking scenarios, each with its own locking order and concurrently executed by multiple threads. It is essential to establish a total order that justifies the absence of cycles. However, these nested blocking scenarios exhibit not static but *dynamic* orders, complicating the construction of such a total order. For instance, two inodes that are with parent-child relation may swap their positions as a file system evolves, therefore exhibiting different orders in the parent-child nested blocking of `unlink`. In addition, `rename` alone contains multiple nested blocking cases, some of which establish non-deterministic orders between unrelated inodes (e.g., source and target directories may be two arbitrary directories and their locking order is dynamic).

Although state-of-the-art efforts have provided insights into the first two challenges, few offer an approach for both. In addition, none can be applied for the diverse and dynamic blocking scenarios in file systems.

To meet such challenges, this paper first proposes the *dynamically layered definite releases* (*DLDR*) specification, which can account for the progress of RefFS. *Definite release* specifies that an acquired lock (e.g., per-inode lock) will always be released, which assists to reason about lock-based blocking by establishing a *rely-guarantee* style protocol between threads that wait for the lock and the thread that owns the lock. Definite release can be specified on concrete (lock) state without assuming locks as primitives, therefore the idea is applicable to ad-hoc synchronizations as well.

The key to modular reasoning is to assign definite releases *layers* that represent their orders in nested blocking. The acquisition of a lock only waits for the definite release of the lock (direct steps). How the definite release is fulfilled (indirect steps) is decoupled and reasoned separately using layers.

Because nested blocking exhibits dynamic orders, the layers are *dynamically* specified according to the file system state. For the parent-child order in `unlink`/`rmdir`, we first propose *hierarchical layers*, which assign an inode's layer by computing the length of path from root. Further we extend the hierarchical layers with the notion of *temporary dependency* represented as ghost state to model the non-deterministic nested blocking order in `rename`. Intuitively, temporary dependency adds a *temporary, logical* edge to the file system tree and on this extended tree, an inode's layer is now computed according to the *longest path* from the root. This layering strategy introduces the needed dependency while obeying existing dependencies to avoid unsound circular dependencies.

To demonstrate the generality of *DLDR*, we extend it to account for directory locking in Linux VFS, which has more nested blocking scenarios. Specifically, we make the extension following the Linux directory locking documentation [25] (assuming it faithfully reflects the implementation). With the formal specification, we give an intuitive and formal deadlock-freedom proof of the locking scheme. We found that a recent patch [51] merged in the mainstream introduced a serious flaw in the scheme, which could lead to deadlocks in running code and got confirmed by the Linux maintainers including Linus Torvalds.

Next, we present the MVT framework, whose program logic is based on the methodologies of *DLDR* and proved sound. MVT is built on existing efforts [64, 104] for verifying termination and functional correctness of concurrent file systems but makes the following new contributions. MVT supports specifying (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.

After that, we successfully apply *DLDR* to verify RefFS in MVT. Compared to existing verified concurrent file system, RefFS for the first time supports highly concurrent path traversal using reference counting (refcount) [26]. This allows operations to bypass each other, thus achieves better performance. Users are not bothered with the more fine-grained behaviors because the refined abstraction of RefFS hides refcounts, locks and internal data structures to offer atomic directory lookups. The abstraction preserves the termination behaviors of RefFS as well, thus layering a solid foundation to reason about applications on top of RefFS [14].

In summary, this paper makes the following contributions:

- A study of termination bugs that motivates this work (§2).
- Dynamically layered definite releases specification for the progress of RefFS and an extension of it for directory locking in Linux (§3).
- The MVT framework, which allows a programmer to specify and verify both termination and functional correctness of concurrent file systems (§4).
- RefFS, the first modularly verified concurrent file system with termination guarantees (§5).
- An evaluation reporting the performance of RefFS and our verification experience (§6).

Our prototypes of MVT and RefFS still have some 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 recovery procedure will restore the state to re-execute the code. But recovery procedure needs to be reasoned on the crashed state. To support crash safety, one may follow the techniques in DaisyNFS [15–17]. MVT does not consider termination problems due to interrupts/exceptions. Supporting them requires considering intermediate states in programs , which we leave as future work. RefFS has simplified read access to use exclusive locks. Nevertheless, we can use read-write locks in RefFS and reason with their implementations in MVT.

2 Motivation

2.1 Studies of Termination Bugs

A prior study [68] on file system patches reveals that up to 40% of concurrency bugs are due to deadlocks. A recent survey [12] on security vulnerabilities of file systems shows that about 7% of CVEs are related to non-termination. Meanwhile, the deadlock-related semantic bugs are hard to diagnose [70], hurting the system for years without being fixed.

Although these efforts have shed light on the significance of avoiding termination bugs, they have not thoroughly examined these bugs from a verification perspective. This raises several important questions: (1) How can we 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 focus our verification efforts in fruitful directions. Therefore, we perform a comprehensive study of termination bugs in Linux file systems (from 2020 to June 2023). We collect 205 bugs in total (by reading commit messages of patches [52]) and make the following observations.

Bug classification. The termination bugs are first classified based on whether they are concurrency bugs. Within the category of non-concurrent bugs (18%), the non-termination, shown as infinite loops, is mainly due to logic mistakes (e.g., missing checks [85]) and generic errors (e.g., inappropriate truncation [40] and overflow [95]). In the concurrent category of termination bugs, the majority of them are attributed to deadlocks (78% of total). Deadlocks occur when a thread becomes blocked, waiting for a specific action that never happens, leading to a situation where none of the involved threads can make progress. A smaller subset of these bugs are caused by livelocks (3% of total). In cases of livelock, a thread is constantly delayed or postponed, resulting in an inability to make progress and ultimately leading to a lack of overall system advancement.

Next, we look into the most dominant class of bugs, i.e., deadlocks, to understand the underlying factors contributing to their prevalence.

Ad-hoc synchronization. A significant portion (40%) of deadlocks involve ad-hoc synchronizations where threads are waiting for specific events such as transaction completion [7], flushing of dirty inodes [6], or other custom synchronization points [78, 90]. These ad-hoc synchronizations can be challenging to detect and analyze, as they often lack specific patterns. Deadlock analysis tools and techniques commonly focus on well-known and structured synchronization patterns, such as lock acquisitions and releases, which may cause them to miss these ad-hoc synchronization-related deadlocks.

Nested blocking. A majority (83%) of deadlocks, excluding AA deadlocks[4], involve nested blocking. Many of those bugs [98, 99] even arise from combinations of diverse nested blocking scenarios (including ad-hoc synchronizations), which can make them even more challenging to identify and resolve. To prevent such bugs, it is necessary to examine not only the local blocking order but also the absence of cyclic dependencies globally within the system. This calls for a comprehensive analysis of the entire codebase, considering the interactions and potential dependencies among multiple threads and resources. However, existing code comments or documentation often fail to present a clear and detailed global order of dependencies, hampering the ability to detect and prevent such deadlocks effectively.

Dynamic order. Dynamic locking order is not uncommon in file systems, particularly in interfaces like object removal, rename, and link creation. For instance, object removal (`unlink`/`rmdir`) acquires inode locks in a parent-child order, with the definition 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 within file systems, such as the forest structure in BTRFS or various list implementations, also exhibit dynamic locking orders. These scenarios further contribute to the complexity of the code. Reasoning about diverse and dynamic nested blocking scenarios in file systems can be extremely intricate. Even for experts in the domain, mistakes can still occur [51], highlighting the difficulty of effectively managing these complexities.

To summarize, the combination of ad-hoc synchronization, nested blocking, and dynamic order in concurrent file systems has contributed to deadlocks becoming the dominant class of termination bugs. These factors, individually and collectively, present significant challenges for verification

2.2 Limitation of Existing Efforts

There are a number of program-analysis based techniques that aim for deadlock [49,92], livelock [9] and infinite loop [11,13] 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 [34,48] can also reveal termination bugs. However, they, as well as dynamic analysis tools cannot avoid false negatives.

For instance, Linux kernel has a runtime validator to ensure the locking correctness [27]. Users needs to inform the validator of the hierarchy (a fixed order) between lock objects. It has the following drawbacks. First, it does not recognize ad-hoc synchronizations. Second, it does not provide a general principle on how to handle dynamic locking order, whose hierarchy cannot be predetermined. Third, the annotations provided by developers may be wrong or insufficient, which give rise to false positives [96].

[4] A thread deadlocks when request the lock owned by itself

Some efforts support deadlock-freedom verification [61, 94]. They track dependencies between blocking primitives to prevent circular blocking. Some [10, 60] also have limited support for dynamically-changeable lock orders. They only consider situations where lock order changes have *local* effects. For instance, a tree mutation with all related nodes protected by locks is not visible to concurrent threads. Thus, it is easier to locally check the absence of circular lock dependencies before and after the lock order change. However, in file systems, lock order changes may result from concurrent threads, which requires nontrivial concurrency reasoning (see §3.3 for details). In addition, these efforts still cannot be applied for ad-hoc synchronizations.

There are also some theoretical work [28, 64] that can 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 §3.2), thus cannot modularly verify file systems. TaDA Live [28] introduces layers to capture lock orders. But it only supports layers defined on current state, which are insufficient to capture the dynamic layers in file systems. Also, TaDA Live and LiLi have neither a mechanical framework nor an executable implementation.

3 Proving the Termination of RefFS

In Linux filesystems, directory locking refers to the locking scheme used for directory operations. It is important because only after VFS has performed directory locking can kernel file systems take over. Hence, any bugs within can endanger the whole OS. However, ensuring its correctness is challenging because it is a major source of diverse and dynamic nested locking scenarios. In this section, we tackle the challenges by first introducing the dynamically layered definite release specification for proving the termination of RefFS. Note that the directory locking behavior of RefFS is fine-grained and similar to that of VFS. Then, we extend the specification to validate the locking correctness of VFS.

3.1 Definite Release

RefFS uses fine-grained locks (i.e., per-inode locks) for synchronization. A thread t acquiring the lock may be blocked by another thread holding the lock. To prove t's termination, we need to show that t will no longer be blocked. Consider the code snippet in Fig. 1a. This simplified version is incorrect because it omits reference counting; we restore it in §5. 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, and a fair lock, such as a ticket lock, is used to ensure termination. For a ticket lock (Fig. 1b), 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. Then the question is why the lock(cur) statement would terminate.

```
// 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.

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

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

(b) ticket lock.

Figure 1: **Single locking in path traversal.**

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 in RefFS, we propose a domain-specific specification called *definite release*.

Definition 1 (Definite Release)

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

Definite release is inspired from 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 cannot support layered reasoning, thus cannot modularly handle nested locking (§3.2).

More formally, definite release ($\mathcal{D}$) is in the form of a state transition: "t owns the lock" $\leadsto$ "t releases the lock". Here, the two state assertions can specify concrete lock state (not just abstract state of lock primitives). For instance, the definite release of ticket locks can be formalized as $(\texttt{owner} = \mathsf{t}.\texttt{i}) \leadsto (\texttt{owner} = \mathsf{t}.\texttt{i} + 1)$, where $\mathsf{t}.\texttt{i}$ means the local variable i of thread t. For other ad-hoc synchronizations, e.g., waiting for count to be zero, we can specify the waited action using the state of count similarly. Intuitively, the notation $\leadsto$ captures that the state transition *eventually* happens, whose formal details we present in §4.

Rely-guarantee style reasoning. Definite release establishes a protocol that helps reason about the termination of locks in two aspects. First, the release of a lock after acquired should be **guaranteed** by all threads. For now, we require that the definite release is fulfilled without relying on other threads so that the definite release guarantee is like an "axiom" that everyone can trustfully rely on. For example, in Fig. 1, once a thread t acquires the lock of cur, given that the lookup will not be blocked and can terminate, t will indeed release the lock on its own.

Second, one can **rely** on other threads releasing the lock to prove the termination of lock statement. Still in Fig. 1, a blocked thread t that spins inside the while loop of lock knows that the lock holder will release the lock by increasing owner, and t only needs to wait for a finite number of threads

```
// Pre: no lock owned
lock(parent);
child=lookup(parent,name);
lock(child);
// Perform checks and
// do unlink/rmdir
...
unlock(parent);
```

Figure 2: **Code snippet for nested locking in `unlink/rmdir`**. Code for error handling omitted.

releasing the lock. So t can eventually acquire the lock.

Note that circular reasoning is usually unsound in proving termination. For instance, in a deadlock example, one may guarantee to always release the lock under the assumption that others release their locks. But our reasoning is not circular because the guarantee of definite release does *not* make assumptions on other threads.

3.2 Hierarchical Layers

File systems also require nested locking to accomplish an operation. Consider the code for removing an inode (i.e., `unlink` and `rmdir`) in Fig. 2. The code acquires the lock of the `parent` inode to look up for the `child`, and then acquires the lock of the `child` to check whether the operation can be performed. The lock of `parent` is released after the operation. Can we use definite releases to reason about this example?

Unfortunately, in nested locking, the release of the first lock may be blocked by the acquisition of the second lock, which violates the rule that definite release should be fulfilled on its own. Hence, we fail to apply the same reasoning. We first define *lock dependency* to simplify the illustration.

Definition 2 (Lock Dependency)

For nested locking `lock(A);lock(B)`, the definite release of lock `A` depends on the definite release of lock `B`, which we call a **lock dependency** from `A` to `B`.

Intuition on termination. Fig. 3a shows the possible lock dependencies in file systems, which constitute a *dependency graph*. Let us only consider the parent-child nested locking. There is a lock dependency from each parent to its children, as shown by the arrows. The dependencies may further extend to the parent's descendants. For example, `root` has lock dependency on `B`, which has lock dependency on `C`. If the *dependency chain* ever contains a cycle, the code is at risk of deadlocks. The good news is that the dependency graph is in the form of the file system tree, so the dependency chain ends up on a leaf inode. The definite release of leaf inode will not be blocked and can be fulfilled on its own. Thus, the definite releases of other inodes in the chain are guaranteed one by one in the leaf-to-root direction. Consequently, the definite release of all inodes can be guaranteed, with which the parent-child nested locking can terminate.

Discussion. If we stick to the approach in §3.1 (LiLi's approach), we cannot use definite releases but have to specify actions that can definitely happen on their own. For instance, to resolve the blocking on `root`, we may have to follow the *dependency chain* (e.g., `root` → `B` → `C` in Fig. 3a) to find the thread that will not be blocked and rely on its lock release (e.g., `C`) to create progress. The reasoning using this fine-grained action intertwines the internal waiting queue of not only `root` but also other locks (e.g., `B` and `C`) as well as the dependencies between locks, which is unnecessarily complex. The evaluation in the second paragraph of §6.2 shows that LiLi's approach may cost 7 times the proof effort (measured in Coq lines) than our modular approach presented below.

Hierarchical layers for definite releases. It is not harmful for definite releases to have dependencies *as long as* there are no circular dependencies. To formalize this intuition, we need a specification representing the dependencies between definite releases. We choose to associate a layer with each definite release and only allow a definite release to depend on those with *higher* layers. For file systems, we propose *hierarchical layers*. The layer of an inode's definite release (an inode's layer in short) is computed by the length of the path from `root`. E.g., in Fig. 3a, `root` is assigned layer-0, and after each path, the layer is increased by one.

More formally, hierarchical layers can be represented as below. Definite release $\mathcal{D}$ takes the inode number as an argument and specifies that for any thread, if the inode is locked by t, the inode will eventually be unlocked. The $locked(inum, \mathrm{t})$ and $unlocked(inum)$ predicates hide internal lock implementations. We define a *layer function* L (implemented by HL), which takes a definite release and a state FS to return a layer if defined. The assertion *reachFromRoot* asserts that the *inum* is reachable from root by following the path in current state FS. Then the layer is the length of the path (with type *Nat*) or undefined otherwise. For now, we assume state FS does not change. Because state FS represents the file tree (enforced by invariants), the hierarchical layers ensure that there are no circular dependencies for any state FS.

$$\mathcal{D}(inum) \overset{\text{def}}{=} \forall \mathrm{t}, locked(inum, \mathrm{t}) \leadsto unlocked(inum)$$

$$L(\mathcal{D}\ inum, FS) \overset{\text{def}}{=} HL(inum, FS)$$

$$HL(inum, FS) \overset{\text{def}}{=} \begin{cases} length(path) & \text{if } \exists path, \\ & \quad reachFromRoot\ inum\ path\ FS \\ \text{undefined} & \text{otherwise} \end{cases}$$

Reasoning with hierarchical layers. The part that `lock` statements can **rely** on definite releases to terminate does not change. We can prove the termination of each lock independently by considering the lock's internal waiting queue.

How the definite releases should be **guaranteed** alters: the fulfillment of definite releases can be blocked and depend on the fulfillment of *higher-layer* definite releases to remove the blockage, i.e., a parent can depend on its children. For Fig. 2, when the release of `parent` is blocked at `lock(child)`, we need to show that the layer of `parent` is lower than `child`, which is true from hierarchical layers. After the termination of `lock(child)` proved with the definite release of `child`, the definite release of `parent` is ensured. Although definite re-

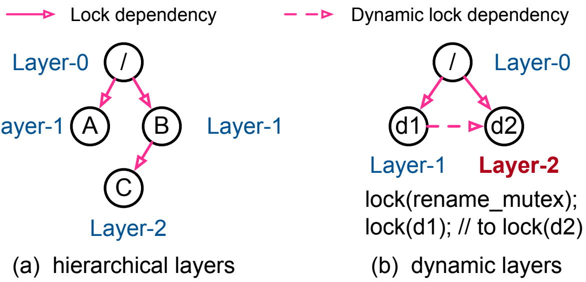


Figure 3: **Lock dependency graphs and layers.**

```
lock_rename(d1,d2):
if(d1==d2){
  lock(d1);return;}
lock(rename_mutex);
if(ancestor(d1,d2)){
  lock(d1);
  lock(d2);
  return;
}
if(ancestor(d2,d1)){
  lock(d2);
  lock(d1);
  return;
}
lock(d1);
lock(d2);
return;
```

Figure 4: **Nested locking in `lock_rename`.**

lease is no longer a fact on its own, it can be soundly deduced because hierarchical layers ensure no circular dependencies.

3.3 Dynamic Layers

Hierarchical layers capture the parent-child lock dependencies and explain the termination of parent-child nested locking. In file systems, the `rename` operation also exhibits nested locking. The `rename` needs to acquire the locks of two directories to ensure atomicity of critical section. Fig. 4 resembles the implementation of Linux VFS. If two directories are equal, we only need to acquire one lock. To simplify the corner cases from concurrent cross-directory `renames`, VFS requires cross-directory `renames` to first acquire a per-filesystem lock (i.e., `rename_mutex` in the code). Nevertheless, to not conflict with the parent-child order, if one directory (e.g., `d1`) is the ancestor to another (e.g., `d2`), we should first acquire the ancestor (e.g., `d1`). Otherwise, the locking order does not matter, so the code chooses a default order (`d1` before `d2`).

In contrast to the parent-child lock dependency, which can be inferred for any given file tree, the lock dependency in `lock_rename` is not known beforehand. It exists temporarily during execution and then becomes unknown again. Specifically, prior to acquiring `rename_mutex`, the lock dependency between `d1` and `d2` is *not stable* since other threads can dynamically alter it. Once `rename_mutex` is acquired, the precise lock dependency becomes known. In Fig. 3b, we represent this temporarily introduced lock dependency with a dashed arrow. Furthermore, after the code acquires the respective locks for `d1` and `d2`, the lock dependency between them can be observed to disappear. This is because their definite releases will no longer depend on each other.

Intuition on termination. During `lock_rename`, the tree-shaped dependency graph is initially expanded by adding a dashed arrow that does not conflict with the existing parent-child dependencies. As the execution progresses, this introduced dependency eventually disappears. It is important to note that the existing lock dependencies remain unaffected, and all definite releases are appropriately ordered, without any circular dependencies. This property guarantees that a definite release can be fulfilled once the preceding definite releases in the dependency chain have been fulfilled. Hence, despite the dynamically changing lock dependency in `lock_rename`, the code terminates, given all definite releases are fulfilled.

Dynamic layers. We propose the *dynamic layers* to capture the lock dependency in `lock_rename`. The layer for each inode is calculated by the length of the *longest* path from the root in the dependency graph. E.g., in Fig. 3b, starting from the initial hierarchical layers, after the introduced lock dependency, the longest path from `root` to `d2` is `root-d1-d2`, so the layer for `d2` is updated to layer-2.

To encode dynamic layers, we introduce *temporary dependency*, a *ghost state* which tracks the lock dependency introduced by `lock_rename`, and extend the file system state FS to a full state S. Specifically, temporary dependency ($TDep$) is an option type of a pair of inums, i.e., either `None` or `(inum1,inum2)` saying there is a lock dependency from `inum1` to `inum2`. There is only ever one $TDep$ value in the entire system, with $TDep$ set to `None` when the `rename_mutex` is not held and $TDep$ set to a value determined by the thread that holds `rename_mutex`. The layer function L is now implemented by DL. If $inum$ does exist in the file tree, evidenced by the existence of its parent, ($par = parent(inum, FS)$) and happens to be the second item in $TDep$ (written $TDep.inum2$ for simplicity), its layer is one plus the larger layer between the hierarchical layer (reuse the definition in §3.2) of its parent and $TDep.inum1$. In other cases, the layer is computed from its parent (root inode has layer zero) or undefined otherwise.

$$
\begin{aligned}
&L(\mathcal{D}\ inum, S) \overset{\text{def}}{=} DL(inum, S)\\
&DL(inum, S) \overset{\text{def}}{=} (\text{assume } S = (FS, TDep))\\
&\begin{cases}
\begin{array}{ll}
max\{HL(TDep.inum1, FS), & \text{if } \exists par, par = parent(inum, FS)\\
\quad HL(par, FS)\} + 1 & \quad \wedge inum = TDep.inum2\\
DL(par, S) + 1 & \text{if } \exists par, par = parent(inum, FS)\\
 & \quad \wedge inum \neq TDep.inum2\\
0 & \text{if } inum = root(FS)\\
\text{undefined} & \text{otherwise}
\end{array}
\end{cases}
\end{aligned}
$$

Note that this definition has focused on inode locks. For the per-filesystem lock `rename_mutex`, we also specify a definite release ($\mathcal{D}_{fs}$), whose layer ($L(\mathcal{D}_{fs}, S)$) is a default minimum value (`min`) so it can depend on all other definite releases.

Reasoning with dynamic layers. The temporarily introduced lock dependency is sound because it does not conflict with the existing lock dependencies. Consequently, when considering the state changes and resulting layer changes, it is important to ensure that all modified layers adhere to the existing lock dependencies. This aspect affects our reasoning as follows:

We should keep in mind that the fulfillment of definite releases $\mathcal{D}_1$ can rely on the fulfillment of *higher-layer* definite releases $\mathcal{D}_2$. Therefore, in addition to proving that the definite releases are fulfilled in the correct order, it becomes necessary to further demonstrate that during the dependency from $\mathcal{D}_1$ to $\mathcal{D}_2$, this dependency relation (i.e., the relative layer relation between $\mathcal{D}_1$ and $\mathcal{D}_2$) remains unchanged, regardless of any state changes and layer changes of $\mathcal{D}_1$ and $\mathcal{D}_2$.

Let's review the reasoning of Fig. 2. The layers of `parent` and `child` may change due to state changes. But we know the layer of `parent` is *stably lower* than `child`, so the code can rely on the definite release of `child` to prove the termination of `lock(child)`.

For Fig. 4, the reasoning process should take into account the update of temporary dependency. The temporary dependency can be modified at the discretion of the proof author. However, it is crucial that the proof author does not abuse this freedom, as doing so would lead to a failed proof. Before line 15, we set temporary dependency to (`d1`,`d2`) in order to establish a lock dependency from `d1` to `d2`. We set it to `None` after line 15. It is worth noting that the full code of `rename` may acquire additional locks after `lock_rename`, potentially requiring further updates to the temporary dependency. The details are deferred to §5 for further explanation.

Other layer changes. Apart from the dynamic layer change in `lock_rename`, inode deletion or insertion can also impact the dependency graph by breaking or adding an edge. We can interpret a `rename` operation as atomically breaking an edge and then adding an edge. It is important to note that these changes in the dependency graph require holding the related locks and cannot arbitrarily occur. By carefully analyzing these cases, we observe that the layer changes align with the existing lock dependencies. Consequently, they do not introduce circular dependencies. Therefore, we can conclude that all definite releases are guaranteed to be fulfilled.

3.4 Directory Locking in Linux VFS

To understand whether dynamic layers scale to more directory locking orders in VFS, we extend dynamic layers to cover all nested locking scenarios as mentioned in the Linux documentation [25]. The extra lock dependencies in VFS that have not been discussed yet are the following: (1) the *dir-to-non-dir* dependency is from any directory to any non-directory, e.g., exposed in link creation, and (2) the *inode-pointer* dependency is from a non-directory to a non-directory with larger address, e.g., shown in a rename when source and target are non-directories.

To capture these dependencies, the layers are defined differently for directories and non-directories. A layer could either be (`Dir`, *nat*) for a directory with its dynamic layer, or (`NonDir`, *addr*) for a non-directory with its address. Comparison rules are: (1) (`Dir`, *nat*) < (`NonDir`, *addr*) for the *dir-to-non-dir* dependency; (2) (`NonDir`, $addr_1$) < (`NonDir`, $addr_2$) iff $addr_1 < addr_2$ for the *inode-pointer* dependency and (3) (`Dir`, n_1) < (`Dir`, n_2) iff $n_1 < n_2$. These rules give a total order of layers because directories are always acquired before non-directories and the two groups are ordered by dynamic layers and inode pointers, respectively.

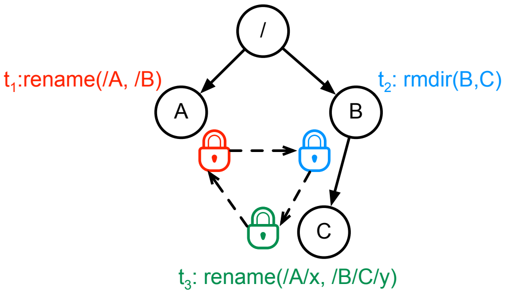


Figure 5: **A deadlock bug in Linux VFS.**

A total order means for a given state, if the code do acquire locks in this order, deadlocks will not happen. But because layers are state-dependent, we still need to analyze whether state updates will bring circular dependencies. We have shown the reasoning for `rename` in §3.3, the most challenging case in file systems. Proof of other cases is provided in Appendix.

Doing the proofs formally uncovered a flaw in current version of the locking scheme. The relevant part is the locking rules for `rename`. For a `rename`, the code used to (1) lock the source when it is a non-directory, (2) lock the target if it exists and (3) (if we need to lock both) lock them following the order as mentioned above, i.e., directory before non-directory or non-directories in inode pointer order. However, a recent patch [51] decides to also lock source when it is a directory (for the purpose of updating its pointer to the parent). For a non-cross-directory rename, this introduces a new and dynamic order between directories (i.e., source and target) that is not protected by `rename_mutex`. The patch takes it for granted that locking source and target (and also source directory and target directory) in inode pointer order would be enough to establish a linear order between directories. However, the problem is that inode pointer order is not transitive with parent-child order as shown in Fig. 5.

Specifically, assume the inode pointer order is `C` < `A` < `B` and three operations all have finished pathname lookups. t_3 owns `rename_mutex` and `C` and requests for `A`. t_1 owns `root` and `A` and requests for `B`. t_1 owns `B` and requests for `C`. Now, we have a deadlock. The maintainers confirmed it and issued a series of patches to fix it [89].

We found this flaw when we failed to define the dynamic layer 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.

3.5 Discussion about Support for Delay

In addition to blocking, *delay* is another factor that can impact system progress. Infinite delays can result in livelocks [54]. However, it is important to note that not all delays are detrimental. For instance, although the acquisition of unfair locks, e.g., test-and-set locks, may be delayed, they can ensure a whole-system progress, i.e., there exists *some* thread that can make progress. To address this issue, a previous effort [64] proposed the concept of token transfer. This idea allows for beneficial delays while preventing infinite delays without whole-system progress. 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. This mechanism can be compatible with our approaches and integrated accordingly. However, for the sake of simplicity in presentation, we will omit these specific details related to supporting delay.

4 The MVT Framework

4.1 Overview

MVT allows for verifying the functional correctness and termination of file systems and borrows following ideas: First, MVT expresses correctness with *termination-preserving refinement* [66], which says observable events (e.g., output and termination events) from the implementation can also be produced from the abstraction. Hence, functional correctness and termination of implementation is ensured given the abstraction is correct in these aspects. Second, MVT supports *compositional* concurrency reasoning with *rely and guarantee conditions* [30, 47]. Rely conditions specify the interference of environmental threads. Each thread should make transitions that satisfy the guarantee conditions. By ensuring that the rely conditions of each thread are implied by the guarantee conditions of its environmental threads, we can verify each thread locally and compose their proofs soundly. Third, MVT specifies *definite actions* [64] to provide general logic rules for blocking in while loops, e.g., ad-hoc synchronizations like `while(y!=1){}` $\|$ `y=1`.

However, definite actions are not modular as discussed in §3. MVT extends definite actions to support layered reasoning. The reasoning principles for layered definite releases directly apply to layered definite actions. Fig. 6 shows the workflow of MVT. Users should specify the specification (§4.2) for the implementation and then follow the inference rules to perform the Hoare-style verification (§4.3). The framework soundness ensures the proof implies the termination-preserving refinement. In addition, MVT supports the verification of C language following an existing framework [104], and code proofs are mechanized in Coq to ensure reliability.

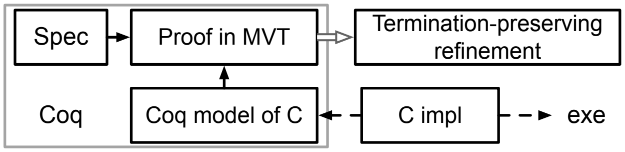


Figure 6: **The workflow of applying MVT.**

4.2 Specification in MVT

The specifications include the abstraction, rely-guarantee conditions, invariants, definite actions and the layer function.

State. We give the state model of MVT. State includes not only the low-level concrete state modified by the implementation but also several auxiliary parts. Because MVT establishes the refinement between the implementation and abstraction, the state contains each thread's *abstract operation* remained to be refined as well as the high-level *abstract state* and an *abstract stack*, which are modified by the abstract operation. MVT uses *tokens* as a local state to ensure termination (see §4.3). Users could also introduce some *ghost state*, which exists in the abstract model, to assist verification.

Abstraction. The abstraction includes the abstract representation of the concrete state and abstract operations on it. An abstraction can hide implementation details, which is easier to check and less error-prone. Highly concurrent traversals that allow thread bypassing are not linearizable (atomic) [104]. MVT provides a specification language which allows to write non-atomic abstract operations. The language has standard commands such as `while` and `if`, but is more suitable for expressing abstract operations: (1) the language's state includes the abstract state and an *abstract stack*, which maps variables to *abstract values*, e.g., lists; the language supports (2) user-supplied primitives that model atomic transitions of abstract state and (3) atomic block $\langle C \rangle$ where C executes atomically (see §5.1 for an example).

Rely/guarantee conditions (R/G) and invariants (I). R and G define the allowed state transitions and are checked in each step: we check that (1) the current thread's state transitions satisfy G, and (2) each state assertion should stay true even when environmental threads' transitions R change the state. I define the boundaries of R/G and should stay true under all transitions. *R*/*G* and *I* define the concurrency protocol (more details in [30, 104]).

Definite action and layer function. A *definite action* describes a state transition, written $P \rightsquigarrow Q$, which means once assertion P is true, (1) Q should eventually be established, and (2) P should be preserved by both environment and current thread until Q holds. The second condition ensures that the only way to make P false is to fulfill the definite action. A definite action is called *enabled* if P holds. A *layer function* L takes a definite action and a state S to return a layer if defined.

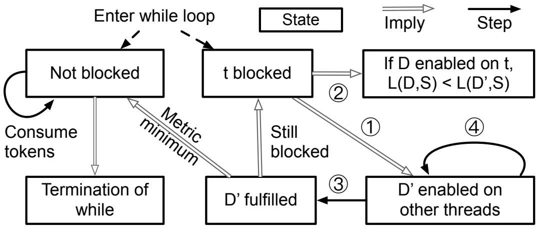


Figure 7: **Logic for termination of while loops.**

4.3 Verification in MVT

The specifications defined by users are decorated into the Hoare triples as the judgement $L,\mathcal{D},R,G,I \vdash \{P \wedge Aop\}C\{Q \wedge AopEnd\}$. The judgement means, for a method C, starting from the precondition P with abstract operation Aop unexecuted, C must terminate to reach the postcondition Q with Aop executed, represented by $AopEnd$. MVT provides a top-level rule, called OBJECT rule, which checks the well-formedness of specifications and asks to prove the judgement for each method of the object. Proving the OBJECT rule gives the termination of the implementation and abstraction, and a termination-preserving refinement between them. To verify each method, users follow inference rules of C statements to step through the program C.

Termination of while loops. Most rules for C statements are standard and similar to previous efforts [30, 104]. Here we mainly introduce the WHILE rule. The termination reasoning of WHILE rule is twofold (see Fig. 7 and the rule below). If the while loop is not blocked, we follow previous efforts [44, 64] to require each round of the loop to consume resources called *tokens*. The number of tokens should be specified before each **while** loop, which strictly decreases, so the loop must terminate after exhausting all tokens. E.g., we can use the length of `path` to specify the number N of tokens for the while loop in Fig. 1a. This while loop is not blocked because, after N rounds, the loop must terminate. The WHILE rule requires establishing the termination of the loop body, which may face blocking and is proved inside with separate rules.

If a thread t is blocked, we prove the following. ① **Blocked condition**: t must be waiting for definite actions $\mathcal{D}'$ to happen, which are enabled on other threads. ② **Dependency condition**: if t has enabled definite actions $\mathcal{D}$, layers of $\mathcal{D}$ must be *stably lower* than $\mathcal{D}'$ until the waited definite action is fulfilled. ③ **Metric decrease**: whenever an definite action in $\mathcal{D}'$ is fulfilled, a well-founded *metric* (i.e., the metric cannot infinitely decrease) will decrease. ④ **Metric non-increase**: the metric never increases.

Suppose enabled definite actions are eventually fulfilled (explained later how this assumption is discharged), conditions ①, ③ and ④ ensure that progress will be created so that the metric will eventually decrease to its minimum, which implies that t is no longer blocked. Then we can use tokens to ensure the termination of the loop.

For instance, when a thread is blocked in the while loop of `lock(cur)` in Fig. 1b, it waits for the definite release of `cur`. The metric can be specified as `(t.i-owner)`, which decreases whenever an environment thread increase `owner`. When the metric decreases to zero, the loop terminates.

② is crucial to avoid circular dependencies. It says, no matter how state S is changed by other threads which introduce dynamic layer changes, the lock dependencies from $\mathcal{D}$ to $\mathcal{D}'$ remains stable, enforced by layer relation $L(\mathcal{D},S) < L(\mathcal{D}',S)$.

Lock abstraction. We can prove the termination of lock implementations with the WHILE rule. But MVT provides an abstraction for locks to ease the burden. A lock `L` is abstracted as an integer. But the abstract operation of acquiring the lock is *not total* but *partial* in the sense it should be blocked when the lock is unavailable. So we may not use `L=t` as the abstract operation for acquiring locks. We follow a recent effort [65] to use `await(L==0){L=t}` as the abstract operation [5], where `L=t` executes atomically when `L==0` holds.

The lock abstraction may also face blocking, and the rule for locks is similar to the WHILE rule with only the metric non-increase condition different. It is now **metric non-increase for the lock**, which says the metric never increases under transitions where the thread keeps being blocked (i.e., the lock owned by others). This reflects the guarantees of fair locks. When `L==0` holds multiple times, the current thread will eventually become the first to the lock. So the condition only covers the cases when `L` is not 0 or t (current thread).

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 should preserve an enabled definite action of some thread except the thread itself could fulfill the definite action. **Postcondition restriction**: the postcondition implies that there are no enabled definite actions.

An enabled definite action is forced to be fulfilled due to the following reasons. (1) When the program terminates, it must be fulfilled according to **postcondition restriction**. (2) 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 **dependency condition**. Intuitively, this will not introduce unsound circular dependencies, so those higher-layer definite actions can indeed be fulfilled (3) The thread is proved to terminate, thus fulfills the definite action itself due to **well-formedness**.

Update of auxiliary state. MVT provides rules for users to update auxiliary states. Abstract state and abstract stack are updated by the abstract operation, which should simulate the concrete operation. Tokens cannot be increased, and the user-specified ghost state can be updated according to the user's will.

Soundness. We have proved soundness of the program logic

[5] We may need wrappers around the `await` statement depending on different locks and different scheduling fairness, which may make it non-atomic. The details have been presented in previous work [65] and are omitted here.

```
// Omit error handling
// Omit definition of Inode
struct inodelock{
  Inode *inode;
  int refcount;
  lock lk;
}

getilock(ilock):
⟨ilock->refcount++⟩

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

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

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

Figure 8: **Reference counting and traversal in RefFS.** Error handling omitted.

(theorem 1) on paper. We show the logic rules establish termination-preserving simulations [66] between the implementation and the abstraction, which ensures the refinement. The full formal soundness proof on paper will be reported separately and its Coq proof is left as future work.

Theorem 1 (Termination Preserving Refinement)

Given the implementation and abstraction, if there exist *rely*/*guarantee* conditions, an *invariant*, *definite actions* and a *layer function*, such that for each operation of the implementation and corresponding abstract operation, the judgment holds w.r.t. the pre-/post-conditions by applying inference rules, then the abstraction is a termination-preserving refinement of the implementation.

5 Design and Verification of RefFS

RefFS is a concurrent in-memory file system running on FUSE. The high-level FUSE API provides the path arguments for all interfaces so that RefFS can implement and verify the path traversal. The verified interfaces include the ones that manipulate the file system structure, e.g., `mkdir`/`mknod`, `rmdir`/`unlink` and `rename`, and those that perform input and output to files, e.g., `open`, `read`, `write` and `release`, which cover commonly used operations.

5.1 Implementation and Abstraction

RefFS reuses most code of a previously verified concurrent file system, AtomFS [104], e.g., the internal functions that operate on directories and files. But RefFS uses reference counting (refcounting) for traversal, which is more fine-grained and provides better performance than lock coupling in AtomFS. Also, the rename implementation and FD-based interfaces of RefFS are different from AtomFS and explained below.

Refcounting. In Fig. 8, the struct `inodelock` has a `refcount` field that counts the references and protects the struct from being freed when `refcount>0`. An inode's `refcount` is initialized to 1, marking that there is one reference in its parent's directory entry. A thread hoping to access the inode can first invoke `getilock` to increase `refcount` by one. To drop the reference, it calls `putilock`, which will free the struct when `refcount` becomes zero. For simplicity, we use the atomic block $\langle C \rangle$ to represent that the code is executed atomically, which is achieved with locks.

The right side of Fig. 8 shows the simplified `traversal` function. It first increases the refcount of `cur` and invokes `lookup` for each name of the path. `lookup` could directly request for `par`'s lock because it holds its reference. After it has found the `child` in `par`, it increases the `refcount` of `child` for later access and releases the `par`'s lock and reference. This allows concurrent threads to bypass each other during path traversal.

The correctness of reference counting lies in: (1) **reference increase**: a thread can increase the references of an inode if it already has a reference in hand (except `root`), e.g., `getilock(child)` of `lookup` in Fig. 8 is due to owning the directory entry of `child` in `par`; (2) **reference decrease**: a thread can decrease a reference only if it owns it; and (3) **reference counting**: the value of `refcount` equals all references combined. (1) and (2) are enforced through rely/guarantee conditions. (3) is formalized as an invariant.

Rename implementation. In Fig. 9, RefFS's `rename` first traverses down the common path of `src` and `dst` (see the `traversal` in Fig. 8). After getting the references of the last common ancestor, we decide whether this is a cross-directory rename by comparing the `src` and `dst`. If they are not equal, the code will acquire `rename_mutex`. The code then traverses the *remaining* path to get the references of source and target directories. Holding the `rename_mutex` ensures the relative position between the two directories is not changed by other renames, so we can use the path arguments to know whether they are ancestors to each other, e.g., `src` being a proper prefix of `dst` means `sdir` is an ancestor to `ddir`. We will first acquire the lock of the ancestor. Otherwise, we will acquire them in default order. The following operation may need to acquire the lock of the inode with name `dn` in `ddir`.

FD-based interfaces. A file descriptor (FD) allows an operation to access an inode directly. RefFS uses the inode number (inum) as the FD. RefFS's `open` follows the path to locate the target inode. It will increase the reference of the inode and return the inum as FD so that `read` and `write` operations can directly access the inode. `release` will drop the reference.

Non-atomic abstraction. The abstract file system, written as `AFS`, is a mapping of inode number (inum) to an *abstract inode*. An *abstract inode* is either a list of bytes for files or a mapping of name to inum for directories. Refcounting allows the operations to bypass each other, which will lead to non-linearizable behavior [104]. Therefore, the abstract operation consists of a series of atomic directory lookups and an atomic

```
  rename(src,sn,dst,dn):
1 ... /*Traverse common
2 path of src and dst*/
3 rel=pathrel(src,dst);
4 if(rel!=0){
5   lock(rename_mutex);}
6 ... /*Traverse to get src
7 and dst directories*/
8 if (rel==0){
9   lock(sdir);
10 } else if(rel==1){
11   lock(sdir);
12   lock(ddir);
13 } else {
14   lock(ddir);
15   lock(sdir); }
16 ...
17 //If dn exists in ddir
18 lock(dchild);
19 ...
```

Figure 9: **Highlighting lock acquisitions of RefFS's rename.** pathrel(src,dst) return 0 if src equals dst, return 1 if src is a proper prefix of dst, otherwise return -1.

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

Figure 10: **Abstract operation MKDIR of RefFS.**

critical section [76, 79]. Fig. 10 shows the abstract operation MKDIR for RefFS. In this code, lookup and do_mkdir are primitives that atomically transfer AFS to AFS′ and return. In addition, we can group statements operating on local variables, e.g., the loop body is grouped into an atomic block.

5.2 Verification of RefFS

During the proofs, we should be careful with the update of ghost state. The most important ghost state is temporal dependency (*TDep*, introduced in §3.3). For Fig. 9, TDep is set to (sdir, ddir) before line 12 or (ddir, sdir) before line 15 to allow the lock dependency between them. Then TDep is updated to (sdir,dchild) before line 18 to establish the lock dependency from source directory to target and reset to None after line 18.

The termination proofs include the checks in 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 preserve the fact until the thread releases the lock itself. The postcondition would specify the thread does not own any lock, so definite releases must be 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 (written $\mathcal{D}'$). And the metric is defined as 0 when L is available and 1 otherwise. The following holds. (1) Blocked condition: some thread t' must hold L, so $\mathcal{D}'$ is enabled on t'. (2) Dependency condition: the dynamic layers ensure that for any lock that t owns, its layer must be stably lower than L. (3) Metric decrease: when L is released, the metric decreases. (4) Metric non-increase for the lock: when L is owned by others, the metric stays 1 and never increases.

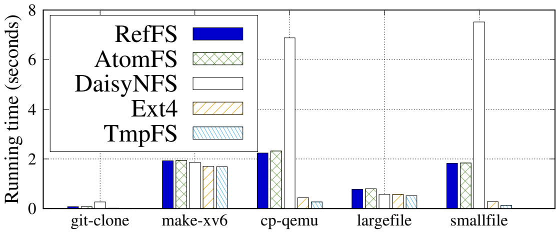


Figure 11: **Applications.** Largefile operates on a big file with 10MB. Smallfile operates on 10K files with 1KB size.

6 Evaluation

This section empirically answers several questions:

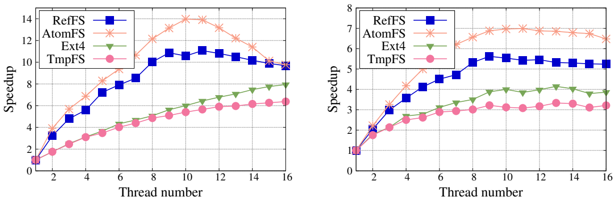
- Can RefFS provide good performance for real-world applications?
- Does reference-counting perform better compared with lock-coupling?
- What is the development effort and how modular is MVT's proof?
- Can MVT help eliminate bugs in practice?

6.1 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 with a widely-used disk file system (ext4 [87]), a verified concurrent file system (AtomFS [104]), a verified concurrent NFS server (DaisyNFS [17]), and an in-memory file system (tmpfs). All the evaluated file systems use in-memory storage, e.g., emulated persistent memory (i.e., /dev/pmem0 in Linux) or in-memory disk in DaisyNFS.

Application performance. RefFS is complete enough to run many kinds of realistic software, including Vim [88] and GCC [33]. To evaluate the application performance, we select two microbenchmarks and three application workloads: LFS microbenchmark [71, 80], cloning the git repository of xv6-public, compiling the sources of the xv6 file system with a makefile and copying source code of qemu. These workloads are also used by prior verified file systems [19, 104]. The application workloads only use a single core.

In Fig. 11, RefFS achieves similar results as AtomFS, and better performance than DaisyNFS in most cases due to DaisyNFS's network I/O overhead. The worse performance of RefFS compared with tmpfs and ext4 is mainly due to the lack of fine-grained optimizations, e.g., highly optimized path traversals and optimized structures for data and metadata.



(a) Scalability on Fileserver.

(b) Scalability on Webproxy.

Figure 12: **Scalability of RefFS.**

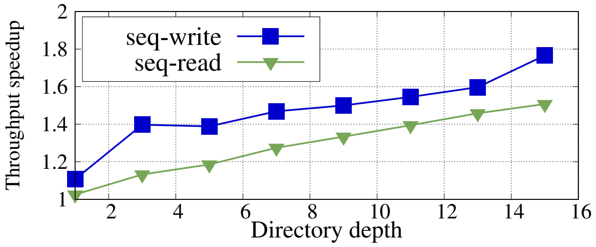


Figure 13: **Speedup of RefFS over AtomFS.**

Running RefFS with FUSE also introduces overhead. These issues can be overcome in future work.

File system scalability. We adopt two commonly used workloads in Filebench [32], Fileserver and Webproxy, to measure the scalability of RefFS. We evaluate with 16 cores and increase the thread number used in the workloads. We do not evaluate DaisyNFS here because its in-memory disk can only support about 400MB of space, while the scalability tests consume more than 3GB in many cases.

The speedup results are in Fig. 12. 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 for each thread number (e.g., 1.08–1.43x higher in Fileserver). RefFS's performance is worse than ext4 and tmpfs, as expected.

Other benefits of reference counting. Besides more parallelism in traversals, 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. 13, 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 code 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**

6.2 Verification Evaluation

Verification effort. MVT reuses the code from CRL-H [104] framework, including the support for C language and concurrency reasoning. The extension is about 3K LOC, mainly devoted to the logic for termination and the model of non-atomic abstract operations. Table 1 shows the lines of code for verifying RefFS. RefFS also reuses AtomFS's internal functions inside the critical section and their proofs, except now we also verify their termination.

Modularity of termination proofs. Definite release with dynamic layers allows to verify each lock separately and reuse the termination proofs for all lock statements. As evidence for modularity, we also use the non-layered definite actions to verify the termination of a lock statement in RefFS as discussed in §3.2, which needs 3K LOC. By contrast, the termination proof for a lock statement with our layered approach (see §5.2) is less than 0.4K LOC.

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. The termination assumes a fair scheduler and a sequentially consistent hardware model. Despite that, we test RefFS with xfstests, a comprehensive file system testing suite, which reports no bugs.

6.3 Bug Discussion

We discuss whether the verification can find bugs in §2. Non-concurrent termination bugs such as logic and low level programming errors [58,86] will fail the proof because we cannot define a well-founded metric that decreases for each round of the loop. In AA-deadlocks [22,57,72], when thread requests for `A` again, the layer of waited action (definite release of `A` by other threads) is not higher than that of enabled action (definite release of `A` by current thread), which will trigger a layer violation. In ABBA-deadlocks [4,20,35,102], proof authors either fail to define the layers, which cannot account for the dependencies of AB and BA at the same time, or define the wrong layer specification and later find the layer cannot meet the required dependencies in code.

For deadlocks that involve dynamic orders [2,5,53,97], MVT 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 synchronizations [21,50,69], MVT's general notion of definite actions can specify them and their reasoning is similar to the bug types above.

The above categories include 90% of bugs surveyed in §2, which MVT can eliminate. A small number of livelock bugs show a pattern where the thread is constantly delayed in infinite loops. This pattern does not appear in RefFS, thus is not primarily considered. But the mechanism for verifying them is briefly discussed in §3.5. Unsupported bugs mainly involve interrupts/exceptions [36,37] and crashes [23].

7 Related Work

Starting from the seminal work of seL4 [56], these years have witnessed tremendous progress on the verification of systems, including operating systems [38,74,83], distributed systems [42,81,93], file systems [18,19,45,82] and many others [24,31,41,62,63,73,75,84,91,100,101,103]. Yet, few of them provide guarantees of systems' liveness. VSync [77] proposes await model checking to automatically verify the termination of lock primitives. CCAL [39] has been used to verify the termination of an MCS lock [55] by organizing the implementation into layers. Ironfleet [42] verifies liveness of distributed systems with a blend of TLA and Hoare style automated verification. However, these efforts do not propose a specification that could be used for the diverse and dynamic nested blocking in file systems.

Reference counting, a widely used technique in Linux, has also caused many severe bugs [43]. Various methods (e.g., invariant-based [29,67] and anti-pattern based [43]) have been proposed to detect these bugs. Although effective in practice, they still suffer from false positives and false negatives. Our work for the first time verifies the correctness of refcounting by showing the implementation using it can refine an abstraction where its details are hidden.

8 Conclusion

This paper has presented MVT 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.

References

[1] Bowen Alpern and Fred B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2:117–126, 2005.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

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

[7] 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.

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

[9] 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.

[10] 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.

[11] Jacob Burnim, Nicholas Jalbert, Christos Stergiou, and Koushik Sen. Looper: Lightweight detection of infinite loops at runtime. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 161–169. IEEE, 2009.

[12] 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*, pages 8–15, 2019.

[13] Michael Carbin, Sasa Misailovic, Michael Kling, and Martin C Rinard. Detecting and escaping infinite loops with jolt. In *European Conference on Object-Oriented Programming*, pages 609–633. Springer, 2011.

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

[15] 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*, pages 243–258, 2019.

[16] 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.

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

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

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

[20] 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.

[21] 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.

[22] 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.

[23] Dave Chinner. xfs: log worker needs to start before intent/unlink recovery. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=a9a4bc8c76d747aa40b30e2dfc176c781f353a08, 2022.

[24] 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*, pages 851–865, 2021.

[25] 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.

[26] 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.

[27] 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.

[28] Emanuele D'Osualdo, Julian Sutherland, Azadeh Farzan, and Philippa Gardner. Tada live: Compositional reasoning for termination of fine-grained concurrent programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 43(4):1–134, 2021.

[29] 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.

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

[31] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 287–305. ACM, 2017.

[32] Filebench. Filebench, 2019.

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

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

[35] 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.

[36] Andreas Gruenbacher. gfs2: Fix mmap + page fault deadlocks for buffered i/o. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=00bfe02f479688a67a29019d1228f1470e26f014, 2021.

[37] Andreas Gruenbacher. gfs2: Disable page faults during lockless buffered reads. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=52f3f033a5dbd023307520af1ff551cadfd7f037, 2022.

[38] 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.

[39] 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*, pages 646–661. ACM, 2018.

[40] 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.

[41] Travis Hance, Andrea Lattuada, Chris Hawblitzel, Jon Howell, Rob Johnson, and Bryan Parno. Storage systems are distributed systems (so verify them that way!). In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 99–115, 2020.

[42] 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*, pages 1–17. ACM, 2015.

[43] Liang He, Purui Su, Chao Zhang, Y. Cai, and Jinxin Ma. One simple api can cause hundreds of bugs an analysis of refcounting bugs in all modern linux kernels. *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023.

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

[45] 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.

[46] 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):1–33, 2022.

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

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

[49] 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.

[50] 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.

[51] Jan Kara. fs: Lock moved directories. https://github.com/torvalds/linux/commit/28eceeda130f5058074dd007d9c59d2e8bc5af2e, 2023.

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

[53] 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.

[54] Jaegeuk Kim. f2fs: should avoid inode eviction in synchronous path. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=b0f3b87fb3abc42c81d76c6c5795f26dbdb2f04b, 2020.

[55] 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.

[56] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220. ACM, 2009.

[57] 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.

[58] 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.

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

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

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

[62] Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. A secure and formally verified linux kvm hypervisor. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1782–1799. IEEE, 2021.

[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 2016)*, pages 385–399, 2016.

[65] Hongjin Liang and Xinyu Feng. Progress of concurrent objects with partial methods. *Proc. ACM Program. Lang.*, 2(POPL), dec 2018.

[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 23rd EACSL Annual Conference on Computer Science Logic and the 29th Annual ACM/IEEE Symposium on Logic in Computer Science (CSL-LICS 2014)*, pages 65:1–65:10, 2014.

[67] Jian Liu, Lin Yi, Weiteng Chen, Chenyu Song, Zhiyun Qian, and Qiuping Yi. Linkrid: Vetting imbalance reference counting in linux kernel with symbolic execution. In *USENIX Security Symposium*, 2022.

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

[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, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. Scaling symbolic evaluation for automated verification of systems code with serval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 225–242, 2019.

[74] 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*, pages 252–269. ACM, 2017.

[75] Luke Nelson, Jacob Van Geffen, Emina Torlak, and Xi Wang. Specification and verification in the field: Applying formal methods to bpf just-in-time compilers in the linux kernel. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, pages 41–61, 2020.

[76] Gian Ntzik. *Reasoning about POSIX file systems*. PhD thesis, Imperial College London, 2016.

[77] Jonas Oberhauser, Rafael Lourenco de Lima Chehab, Diogo Behrens, Ming Fu, Antonio Paolillo, Lilith Oberhauser, Koustubha Bhat, Yuzhong Wen, Haibo Chen, Jaeho Kim, et al. 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*, pages 530–545, 2021.

[78] 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.

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

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

[81] 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.

[82] 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.

[83] 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.

[84] 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*, pages 866–881, 2021.

[85] 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.

[86] 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.

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

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

[89] Al Viro. Patches: rename deadlock fixes. https://git.kernel.org/pub/scm/linux/kernel/git/viro/vfs.git/commit/?h=work.rename, 2023.

[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] Xi Wang, David Lazar, Nickolai Zeldovich, Adam Chlipala, and Zachary Tatlock. Jitk: A trustworthy in-kernel interpreter infrastructure. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 33–47, 2014.

[92] 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.

[93] 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 *ACM SIGPLAN Notices*, volume 50, pages 357–368. ACM, 2015.

[94] Amy Williams, William Thies, and Michael D Ernst. Static deadlock detection for java libraries. In *European conference on object-oriented programming*, pages 602–629. Springer, 2005.

[95] 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.

[96] 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.

[97] 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.

[98] Chunguang Xu. ext4: fix a possible abba deadlock due to busy pa. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=8c80fb312d7abf8bcd66cca1d843a80318a2c522, 2021.

[99] Sidong Yang. btrfs: qgroup: fix deadlock between rescan worker and remove qgroup. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=d4aef1e122d8bbdc15ce3bd0bc813d6b44a7d63a, 2022.

[100] Jianan Yao, Runzhou Tao, Ronghui Gu, and Jason Nieh. Duoai: Fast, automated inference of inductive invariants for verifying distributed protocols. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 485–501, 2022.

[101] Jianan Yao, Runzhou Tao, Ronghui Gu, Jason Nieh, Suman Jana, and Gabriel Ryan. Distai: Data-driven automated invariant learning for distributed protocols. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 405–421, 2021.

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

[103] Arseniy Zaostrovnykh, Solal Pirelli, Rishabh Iyer, Matteo Rizzo, Luis Pedrosa, Katerina Argyraki, and George Candea. Verifying software network functions with no verification expertise. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 275–290, 2019.

[104] 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*, pages 259–274, 2019.