Push-Button Verification of File Systems via Crash Refinement

Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang, *University of Washington*

https://www.usenix.org/conference/osdi16/technical-sessions/presentation/sigurbjarnarson

This paper is included in the Proceedings of the
12th USENIX Symposium on Operating Systems Design
and Implementation (OSDI '16).

November 2–4, 2016 • Savannah, GA, USA

ISBN 978-1-931971-33-1

Push-Button Verification of File Systems via Crash Refinement

Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, Xi Wang
University of Washington

Abstract

The file system is an essential operating system component for persisting data on storage devices. Writing bug-free file systems is non-trivial, as they must correctly implement and maintain complex on-disk data structures even in the presence of system crashes and reorderings of disk operations.

This paper presents Yggdrasil, a toolkit for writing file systems with *push-button* verification: Yggdrasil requires no manual annotations or proofs about the implementation code, and it produces a counterexample if there is a bug. Yggdrasil achieves this automation through a novel definition of file system correctness called *crash refinement*, which requires the set of possible disk states produced by an implementation (including states produced by crashes) to be a subset of those allowed by the specification. Crash refinement is amenable to fully automated satisfiability modulo theories (SMT) reasoning, and enables developers to implement file systems in a modular way for verification.

With Yggdrasil, we have implemented and verified the Yxv6 journaling file system, the Ycp file copy utility, and the Ylog persistent log. Our experience shows that the ease of proof and counterexample-based debugging support make Yggdrasil practical for building reliable storage applications.

1 Introduction

File systems are a vital operating system service for user applications to manage and persist data. Their correctness is critical to system reliability; file system corruption can damage files and even render the disk unable to mount [12, 13]. Correctly implementing a file system is difficult [23], due to the need to maintain complex on-disk data structures that must remain consistent in the face of power failures and system crashes. Many bugs have been found in commonly used file systems, and have led to serious data losses [27, 35, 39, 45, 52, 54]. Such bugs are likely to continue proliferating due to the complexity of modern storage stacks [1, 8].

Yggdrasil is a toolkit that helps programmers write file systems and formally verify their correctness in a *push-button* fashion. Yggdrasil asks programmers for three inputs: a specification of the expected behavior, an implementation, and consistency invariants indicating whether a file system image is in a consistent state. It then performs verification to check if the implementation meets the specification. If there is a bug, Yggdrasil produces a counterexample to help identify and fix the cause. If the verification passes, Yggdrasil produces an executable file system. It requires *no* manual annotations or proofs about the implementation code.

A key challenge for push-button file system verification is to minimize the proof burden. One approach to verified file systems is to ask programmers to construct a proof of implementation correctness using an interactive theorem prover such as Coq [11] or Isabelle [33]. Pioneering work in this direction includes COGENT [2, 34], Flashix [16, 47], and FSCQ [7], which are impressive engineering achievements. However, writing proofs requires both a high degree of expertise and a significant time investment. For instance, Amani et al. reported that verifying two operations of the BilbyFs file system took 9.25 person months, writing 13,000 lines of proof for 1,350 lines of code [2]. Verifying the FSCQ file system took Chen et al. 1.5 years; the code size is 10× that of xv6, an unverified file system with similar features [7]. To free programmers from such a proof burden, Yggdrasil provides fully automated reasoning.

Conceptually, showing that a file system is correct involves exploring its behavior along all execution paths and against all possible disk states. In practice, such exhaustive exploration is intractable: file systems operate on massive inputs (e.g., entire disks); their code often has many execution paths; and non-determinism adds even more complexity, since one needs to reason about crashes at arbitrary points during execution and reorderings of writes due to the disk cache. Existing file-system automated reasoning tools [52–54] therefore focus on bug finding rather than verification.

Yggdrasil scales up automated reasoning for verifying file systems with the idea of *crash refinement*, a new definition of file system correctness. Crash refinement captures the notion that even in the presence of non-determinism, such as system crashes and reordering of writes, any disk state produced by a correct implementation must also be producible by the specification (see §3 for a formal definition). This definition is amenable to efficient satisfiability modulo theories (SMT) reasoning, an extension of boolean satisfiability. Yggdrasil formulates file system verification as an SMT problem and invokes a state-of-the-art SMT solver (Z3 [15]) to fully automate the proof process.

SMT reasoning is not, by itself, a push-button solution; building verified file systems also requires careful design. Crash refinement enables programmers to implement file systems by *stacking layers of abstraction*: if an implementation is a crash refinement of an (often much simpler) specification, they are indistinguishable to higher layers. The higher layers can use lower specifications without reasoning about the implementation details. This modular design allows Yggdrasil to verify a file system by exhausting all execution paths within a layer while avoiding path explosion between layers.

In addition, crash refinement enables transparent switching between different implementations that satisfy the same specification. Programmers can use simple data structures for verification, and then refine them to more efficient versions with the same correctness guarantees. Separating logical and physical concerns in this fashion allows Yggdrasil to verify complex, high-performance on-disk data structures.

We have used Yggdrasil to implement and verify Yxv6+sync, a journaling file system that resembles xv6 [14] and FSCQ [7], and Yxv6+group_commit, an optimized variant with relaxed crash consistency [5, 37]. To demonstrate Yggdrasil on a broader set of applications, we have built Ycp, a file copy utility on top of Yxv6; and Ylog, which resembles the persistent log from the Arrakis operating system [36]. We have also built general-purpose "peephole optimizers" [28] for file system code (e.g., removing superfluous disk flushes). We believe that the ease of verification makes Yggdrasil attractive for building verified storage applications.

We have been using the Yxv6 file system, which runs on top of FUSE [17], to self-host Yggdrasil's daily development on Linux. It has passed `fsstress` from the Linux Test Project [26] and the SibylFS POSIX conformance tests [42] (except for incomplete features, such as hard links and extended attributes). We have found its performance to be reasonable: within 10× of ext4's default configuration and 3–150× faster than FSCQ. Yggdrasil focuses on single-threaded systems; verifying concurrent implementations is beyond the scope of this paper.

This paper makes the following contributions:

- a formalization of file system crash refinement that is amenable to fully automated SMT reasoning;
- the Yggdrasil toolkit for building verified file systems through crash refinement; and
- a case study of building the Yxv6 file system and several other storage programs using Yggdrasil.

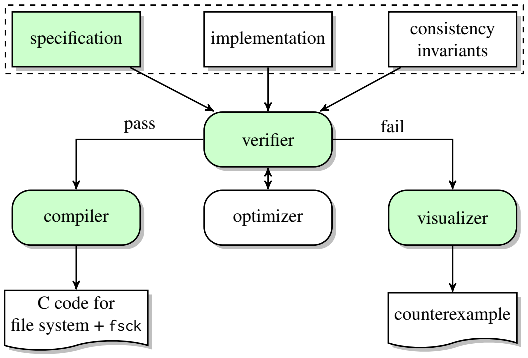


Figure 1: The Yggdrasil development flow. Rectangular boxes (within the dashed frame) denote input written by programmers; rounded boxes denote Yggdrasil's components; and curved boxes denote output. Shaded boxes are trusted to be correct and the rest are untrusted.

The rest of the paper is organized as follows. §2 gives a walkthrough of Yggdrasil's usage. §3 presents formal definitions and the main components. §4 describes the Yxv6 file system and §5 describes other storage applications built using Yggdrasil. §6 discusses Yggdrasil's limitations and our experience. §7 provides implementation details. §8 evaluates correctness and performance. §9 relates Yggdrasil to prior work. §10 concludes.

2 Overview

Figure 1 shows the Yggdrasil development flow. Programmers write the specification, implementation, and consistency invariants all in the same language (a subset of Python in our current prototype; see §3.2). If there is any bug in the implementation or consistency invariants, the verifier generates a counterexample to visualize it. For better run-time performance, Yggdrasil optionally performs optimizations (either built-in or written by developers) and re-verifies the code. Once the verification passes, Yggdrasil emits C code, which is then compiled and linked using a C compiler to produce an executable file system, as well as an `fsck` checker.

This section gives an overview of each of these steps, using a toy file system called YminLFS as a running example. We will show how to specify, implement, verify, and debug it; how to optimize its performance; and how to get a running file system mounted via FUSE [17].

YminLFS is a log-structured file system [44]. It is kept minimal for demonstration purposes: there are no segments, subdirectories, or garbage collection, and files are zero-sized (no read, write, or unlink). But its core functionality is still tricky to implement correctly due to nondeterminism and corner cases like overflows. In fact, the verifier caught two bugs in our initial implementation. The development of YminLFS took one of the authors less than four hours, as detailed next.

2.1 Specification

In Yggdrasil, a file system specification consists of three parts: an abstract data structure representing the logical layout, a set of operations over this data structure to define the intended behavior, and an equivalence predicate that defines whether a given implementation satisfies the specification.

Abstract data structure. We start by specifying the abstract data structure for YminLFS:

```
class FSSpec(BaseSpec):
  def __init__(self):
    self._childmap  = Map((InoT, NameT), InoT)
    self._parentmap = Map(InoT, InoT)
    self._mtimemap  = Map(InoT, U64T)
    self._modemap   = Map(InoT, U64T)
    self._sizemap   = Map(InoT, U64T)
```

The state of the data structure is described by five *abstract maps*, created by calling the Map constructor with *abstract types* specifying the map's domain and range. The childmap maps a directory inode number and a name to a child inode number; parentmap maps an inode number back to its parent directory's inode number; and the remaining maps store inode metadata (mtime, mode, and size). Both InoT and U64T are 64-bit integer types, and NameT is a string type.

The FSSpec data structure itself places only weak constraints on the logical layout of YminLFS. For example, it does not rule out layouts in which an inode d contains an inode f according to the childmap, but f is not contained in d according to the parentmap. The FSSpec specification disallows such invalid layouts with a *well-formedness invariant*:

```
def invariant(self):
 ino, name = InoT(), NameT()
 return ForAll([ino, name], Implies(
  self._childmap[(ino, name)] > 0,
  self._parentmap[self._childmap[(ino, name)]] == ino))
```

The invariant says that the parent and child mappings of valid (positive) inode numbers agree with each other. Both ForAll and Implies are built-in logical operators.

File system operations. Given our logical layout, we can now specify the desired behavior of file system operations. Read-only operations, such as lookup and stat, are easy to define:

```
def lookup(self, parent, name):
  ino = self._childmap[(parent, name)]
  return ino if ino > 0 else -errno.ENOENT

def stat(self, ino):
  return Stat(size=self._sizemap[ino],
              mode=self._modemap[ino],
              mtime=self._mtimemap[ino])
```

Operations that modify the file system are more complex, as they involve updating the state of the abstract maps. For example, to add a new file to a given directory, mknod needs to update all abstract maps as follows:

```
def mknod(self, parent, name, mtime, mode):
  # Name must not exist in parent.
  if self._childmap[(parent, name)] > 0:
    return -errno.EEXIST

  # The new ino must be valid & not already exist.
  ino = InoT()
  assertion(ino > 0)
  assertion(Not(self._parentmap[ino] > 0))

  with self.transaction():
    # Update the directory structure.
    self._childmap[(parent, name)] = ino
    self._parentmap[ino] = parent
    # Initialize inode metadata.
    self._mtimemap[ino] = mtime
    self._modemap[ino]  = mode
    self._sizemap[ino]  = 0

  return ino
```

The InoT() constructor returns an abstract inode number, which is constrained to be valid (i.e., positive) and not present in any directory. The changes to the file system are wrapped in a transaction to ensure that they happen atomically or not at all (if the system crashes).

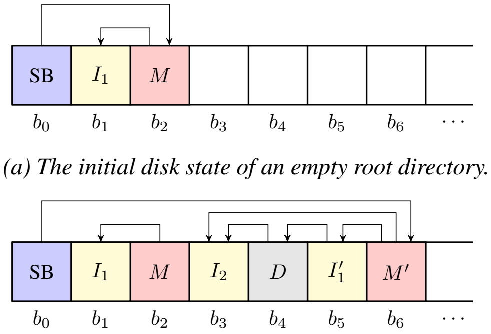
State equivalence predicate. The last part of our YminLFS specification defines what it means for a given file system state to be correct:

```
def equivalence(self, impl):
  ino, name = InoT(), NameT()
  return ForAll([ino, name], And(
    self.lookup(ino, name) == impl.lookup(ino, name),
    Implies(self.lookup(ino, name) > 0,
      self.stat(self.lookup(ino, name)) ==
      impl.stat(impl.lookup(ino, name)))))
```

In particular, we require a correct implementation to contain the same files as the abstract data structure, and each file to have the same metadata as its abstract counterpart.

Putting it all together. With our toy specification completed, we now highlight two key features of the Yggdrasil specification approach. First, Yggdrasil specifications are free of implementation details and are therefore reusable. The FSSpec data structure does not mandate any particular on-disk layout, nor does it force the implementation to be, for example, a log-structured file system. In fact, our Yxv6 journaling file system is built on top of an extension of this specification (see §4).

Second, Yggdrasil specifications are both succinct and expressive. For example, the specification of mknod provides two deep properties in just a few lines of code: crash-free functional correctness (i.e., a file will be created with the correct metadata if there is no crash); and crash safety (i.e., file creation is all-or-nothing even in the face of crashes).



(a) The initial disk state of an empty root directory.

(b) The disk state after adding one file.

Figure 2: YminLFS's on-disk layout. SB is the superblock; I denotes an inode block; M denotes an inode mapping block; D denotes a data block; arrows denote pointers.

2.2 Implementation

To implement a file system in Yggdrasil, the programmer needs to choose a *disk model*, write the code for each specified operation, and write the *consistency invariants* for the on-disk layout. We describe the disk model next, followed by a brief overview of the implementation and consistency invariants for YminLFS. We omit full implementation details (200 lines of Python) for space reasons.

Disk model. Yggdrasil provides several disk models: YminLFS (as well as Yxv6) uses the asynchronous disk model; we will use a synchronous one in §5. The asynchronous disk model specifies a block device that has an unbounded volatile cache and allows arbitrary reordering. Its interface includes the following operations:

- d.write(a, v): write a data block v to disk address a;
- d.read(a): return a data block at disk address a; and
- d.flush$()$: flush the disk cache.

This disk model is trusted to be a correct specification of the underlying physical disk, as we discuss in §4.2. Unless otherwise specified, we assume 64-bit block addresses and 4 KB blocks. We also assume that a single block read/write is atomic, similar to prior work [7, 37].

A log-structured file system. YminLFS is implemented as a log-structured file system that works in a copy-on-write fashion. In particular, it does not overwrite existing blocks (except for the superblock in block zero); it has no garbage collection; and it simply fails when it runs out of blocks, inodes, or directory entries. Its interface provides a mkfs operation for initializing the disk, as well as the operations for reading and modifying the file system state that we specified in §2.1.

The mkfs operation initializes the disk as shown in Figure 2a. The effect of the operation is to create a file system with a single empty root directory. This involves writing three blocks: the superblock, an inode I_1 for the root directory, and an inode mapping M that stores the mapping from inode numbers to block numbers. After initialization, M has one entry, $1 \mapsto b_1$, and I_1 points to no data blocks, as the root directory is empty. The superblock points to M, and it stores two additional counters: the next available inode number i (which is initialized to 2 since the root is 1) and the next available block number b (which is initialized to 3).

To add a file to the root directory, mknod changes the disk state from Figure 2a to Figure 2b, as follows:

1. add an inode block I_2 for the new file;
2. add a data block D for the root directory, which now has one entry that maps the name of the new file to its inode number 2;
3. add an inode block I'_1 for the updated root directory, which points to its data block D;
4. add an inode mapping block M', which has two entries: $1 \mapsto b_5$ and $2 \mapsto b_3$;
5. finally, update the superblock SB to point to the latest inode mapping M'.

Since the disk can reorder these updates, mknod must issue disk flushes to be crash-safe. For example, if there is no flush between the last two writes (steps 4 and 5), the disk can reorder them; if the system crashes in between the reordered writes, the superblock will point to garbage data in b_6, resulting in corrupted YminLFS state. For now, we assume a naïve but correct implementation of mknod that inserts five flushes, one after each write. In §2.4, we will use the Yggdrasil optimizer to remove the first three flushes.

Consistency invariants. A consistency invariant for a file system implementation is analogous to the well-formedness invariant for its specification—it is a predicate that determines whether a given disk state corresponds to a valid file-system image. Yggdrasil uses consistency invariants for two purposes: push-button verification and run-time checking in the style of fsck [20, 30]. For verification, Yggdrasil checks that the invariant holds for the initial file system state right after mkfs; in addition, it assumes the consistency invariant as part of the precondition for each operation, and checks that the invariant holds as part of the postcondition. Once the implementation is verified, Yggdrasil can optionally generate an fsck-like checker from these invariants (though the checker cannot repair corrupted file systems). Such a checker is useful even for a bug-free file system, as hardware failures and bugs in other parts of the system can damage the file system [40].

The YminLFS consistency invariant constrains three components of the on-disk layout (Figure 2): the superblock *SB*, the inode mapping block M, and the root directory data block D. The superblock constraint requires the next available inode number i to be greater than 1, the next available block number b to be greater than 2, and the pointer to M to be both positive and smaller than b. The inode mapping constraint ensures that M maps each inode number in range $(0, i)$ to a block number in range $(0, b)$. Finally, the root directory constraint requires D to map file names to inode numbers in range $(0, i)$. These three constraints are all Yggdrasil needs to verify YminLFS (see §2.3).

2.3 Verification

To verify that the YminLFS implementation (§2.2) satisfies the `FSSpec` specification (§2.1), Yggdrasil uses the Z3 solver [15] to prove a two-part crash refinement theorem (§3). The first part of the theorem deals with crash-free executions. It requires the implementation and specification to behave alike in the absence of crashes: if both YminLFS and `FSSpec` start in equivalent and consistent states, they end up in equivalent and consistent states. The verifier defines equivalence using the specification's `equivalent` predicate (§2.1), and consistency using the implementation's consistency invariants (§2.2).

The second part of the theorem deals with crashing executions. It requires the implementation to exhibit no more crash states (disk states after a crash) than the specification: each possible state of the YminLFS implementation (including states caused by crashes and reordered writes) must be equivalent to *some* crash state of `FSSpec`.

Counterexamples. If there is any bug in the implementation or consistency invariants, the verifier will generate a *counterexample* to help programmers understand the bug. A counterexample consists of a concrete trace of the implementation that violates the crash refinement theorem. As an example, consider the potential missing flush bug described in §2.2. If we remove the flush between the last two writes in the implementation of `mknod`, Yggdrasil outputs the following counterexample:

```
# Pending writes
lfs.py:167 mknod write(new_imap_blkno, imap)

# Synchronized writes
lfs.py:148 mknod write(new_blkno, new_ino)
lfs.py:154 mknod write(new_parentdata, parentdata)
lfs.py:160 mknod write(new_parentblkno, parentinode)
lfs.py:170 mknod write(SUPERBLOCK, sb)

# Crash point
[..]
lfs.py:171 mknod flush()
```

The output describes the bug by showing the point at which the system crashes and the list of writes pending in the cache (along with their source code locations). In this example, the write of the new inode mapping block (step 4 above) is still pending, but the write to update the superblock to point to that block (step 5) has reached the disk, corrupting YminLFS's state.

The visualization of "pending" and "synchronized" writes in the counterexample is specific to the asynchronous disk model; one can extend Yggdrasil with new disk models and customized visualizations.

Our initial YminLFS implementation contained two other bugs: one in the `lookup` logic and one in the data layout. Neither of the bugs appeared during testing runs. Both bugs were found by the verifier in a matter of seconds, and we quickly localized and fixed them by examining the resulting counterexamples.

Proofs. If the Yggdrasil verifier finds no counterexamples to the crash refinement theorem, then none exist, and we have obtained a *proof* of correctness. In particular, the crash refinement theorem holds for all disks with up to 2^{64} blocks, and for every trace of file system operations, regardless of its length. After we fixed the bugs in our initial YminLFS implementation, the verifier proved its correctness in under 30 seconds.

It is worth noting that the theorem holds if the file system is the only user of the disk. For instance, it does *not* hold if an adversary corrupted the file system image by directly modifying the disk. To address this issue, one can run `fsck` generated by Yggdrasil, which guarantees to detect any such inconsistencies.

2.4 Optimizations and compilation

As described in §2.2, YminLFS's `mknod` implementation uses five disk flushes. Yggdrasil provides a greedy optimizer that tries to remove every disk flush and re-verify the code. Running the optimizer on the `mknod` code removes three out of the five flushes within three minutes, while still guaranteeing correctness.

The optimized and verified YminLFS implementation, which is in Python, is executable but slow. Yggdrasil invokes the Cython compiler [3] to generate C code from Python for better performance. It also provides a small bridge to connect the generated C code to FUSE [17]. The result is a single-threaded user-space file system.

2.5 Summary

We have demonstrated how to specify, implement, debug, verify, optimize, and execute the YminLFS file system using Yggdrasil. Compared to previous file system verification work, push-button verification eases the proof burden and enables automated features such as visualizing bugs and optimizing code.

Since there is no need to manually prove or annotate implementation code when using Yggdrasil, the verification effort is spent mainly on writing the specification and coming up with consistency invariants about the on-disk data format. We find the counterexample visualizer useful for finding bugs in these two parts.

The trusted computing base (TCB) includes the file system specification, Yggdrasil's verifier, visualizer, and compiler (but not the optimizer), their dependencies (i.e., the Z3 solver, Python, and gcc), as well as FUSE and the Linux kernel. See §6 for discussion on limitations.

3 The Yggdrasil architecture

In Yggdrasil, the core notion of correctness is crash refinement. This section gives a formal definition of crash refinement, and describes how Yggdrasil's components use this definition to support verification, counterexample visualization, and optimization.

3.1 Reasoning about systems with crashes

In Yggdrasil, programmers write both specifications and implementations (referred to as "systems" in this section) as state machines: each system comprises a state and a set of operations that transition the state. A transition can occur only if the system is in a consistent state, as determined by its consistency invariant $\mathcal{I}$. This invariant is a predicate over the system's state, indicating whether it is consistent or corrupted; see §2.2 for an example.

Consider a specification F_0 and an implementation F_1. Our goal is to show that F_1 is correct with respect to F_0. Since both systems are state machines, a strawman definition of correctness is that they transition in lock step (i.e., bisimulation): starting from equivalent consistent states, if the same operation is invoked on both systems, they will transition to equivalent consistent states (where equivalence between states is defined by a system-specific predicate). However, this bisimulation-based definition is too strong for systems that interact with external storage, as it does not account for nondeterminism from disk reorderings, crashes, or recovery.

To address this shortcoming, we introduce *crash refinement* as a new definition of correctness. At a high level, crash refinement says that F_1 is correct with respect to F_0 if, starting from equivalent consistent states and invoking the same operation on both systems, *any* state produced by F_1 is equivalent to *some* state produced by F_0. To formalize this intuition, we define the behavior of a system in the presence of crashes, formalize crash refinement for individual operations, and extend the resulting definition to entire systems.

System operations. We model the behavior of a system operation with a function f that takes three inputs:

- its current state s;
- an external input $\boldsymbol{x}$, such as data to write; and
- a crash schedule $\boldsymbol{b}$, which is a set of boolean values denoting the occurrence of crash events.

Applying f to these inputs, written as $f(s, \boldsymbol{x}, \boldsymbol{b})$, produces the next state of the system.

As a concrete example, consider a single disk write operation that writes value v to disk address a. The external input to the write operation's function f_w is the pair (a, v). The state s is the disk content before the write; $s(a)$ gives the old value at the address a. The asynchronous disk model in Yggdrasil generates a pair of boolean values $(on, sync)$ as the crash schedule. The on value indicates whether the write operation completed successfully by storing its data into the volatile cache. The $sync$ value indicates whether the write's effect has been synchronized from the volatile cache to stable storage. After executing the write operation, the disk is updated to contain v at the address a only if both on and $sync$ are true, and left unchanged otherwise (e.g., the system crashed before completing the write, or before synchronizing it to stable storage):

$$f_w(s, \boldsymbol{x}, \boldsymbol{b}) = s[a \mapsto \texttt{if } on \wedge sync \texttt{ then } v \texttt{ else } s(a)],$$
$$\text{where } \boldsymbol{x} = (a, v) \text{ and } \boldsymbol{b} = (on, sync).$$

Crash refinement. To define crash refinement for a given schedule, we start from a special case where write operations always complete and their effects are synchronized to disk. That is, the crash schedule is the constant vector $\boldsymbol{true}$. Let $s_0 \sim s_1$ denote that s_0 and s_1 are equivalent states according to a user-defined equivalence relation (as in §2.1). We write $s_0 \sim_{\mathcal{I}_0, \mathcal{I}_1} s_1$ to say that s_0 and s_1 are equivalent and consistent according to their respective system invariants $\mathcal{I}_0$ and $\mathcal{I}_1$:

$$s_0 \sim_{\mathcal{I}_0, \mathcal{I}_1} s_1 \triangleq \mathcal{I}_0(s_0) \wedge \mathcal{I}_1(s_1) \wedge s_0 \sim s_1.$$

With a crash-free schedule $\boldsymbol{true}$, two functions f_0 and f_1 are equivalent if they produce equivalent and consistent output states when given the same external input $\boldsymbol{x}$, as well as equivalent and consistent starting states:

Definition 1 (Crash-free equivalence)**.** Given two functions f_0 and f_1 with their system consistency invariants $\mathcal{I}_0$ and $\mathcal{I}_1$, respectively, we say f_0 and f_1 are *crash-free equivalent* if the following holds:

$$\forall s_0, s_1, \boldsymbol{x}.\ (s_0 \sim_{\mathcal{I}_0, \mathcal{I}_1} s_1) \Rightarrow (s_0' \sim_{\mathcal{I}_0, \mathcal{I}_1} s_1')$$
$$\text{where } s_0' = f_0(s_0, \boldsymbol{x}, \boldsymbol{true}) \text{ and } s_1' = f_1(s_1, \boldsymbol{x}, \boldsymbol{true}).$$

Next, we allow for the possibility of crashes. We say that f_1 is correct with respect to f_0 if, for any crash schedule, the state produced by f_1 with that schedule is equivalent to a state produced by f_0 with *some* schedule:

Definition 2 (Crash refinement without recovery)**.** Function f_1 is a *crash refinement (without recovery)* of f_0 if (1) f_0 and f_1 are crash-free equivalent and (2) the following holds:

$$\forall s_0, s_1, \boldsymbol{x}, \boldsymbol{b}_1.\ \exists \boldsymbol{b}_0.\ (s_0 \sim_{\mathcal{I}_0, \mathcal{I}_1} s_1) \Rightarrow (s_0' \sim_{\mathcal{I}_0, \mathcal{I}_1} s_1')$$
$$\text{where } s_0' = f_0(s_0, \boldsymbol{x}, \boldsymbol{b}_0) \text{ and } s_1' = f_1(s_1, \boldsymbol{x}, \boldsymbol{b}_1).$$

Finally, we consider the possibility that the system may run a *recovery function* upon reboot. A recovery function r is a system operation (as defined above) that takes no external input (as it is executed when the system starts). It should also be *idempotent*: even if the system crashes during recovery and re-runs the recovery function many times, the resulting state should be the same once the recovery is complete.

Definition 3 (Recovery idempotence)**.** A recovery function r is idempotent if the following holds:

$$\forall s, \boldsymbol{b}.\ r(s, \boldsymbol{true}) = r(r(s, \boldsymbol{b}), \boldsymbol{true}).$$

Note that this definition accounts for multiple crash-reboot cycles during recovery, by repeated application of the idempotence definition on each intermediate crash state $r(s, \boldsymbol{b})$, $r(r(s, \boldsymbol{b}), \boldsymbol{b'})$, ..., where $\boldsymbol{b}$, $\boldsymbol{b'}$, ... are the schedules for each crash during recovery.

Definition 4 (Crash refinement with recovery)**.** Given two functions f_0 and f_1, their system consistency invariants $\mathcal{I}_0$ and $\mathcal{I}_1$, respectively, and a recovery function r, f_1 with r is a *crash refinement* of f_0 if (1) f_0 and f_1 are crash-free equivalent; (2) r is idempotent; and (3) the following holds:

$$\forall s_0, s_1, \boldsymbol{x}, \boldsymbol{b}_1.\ \exists \boldsymbol{b}_0.\ (s_0 \sim_{\mathcal{I}_0,\mathcal{I}_1} s_1) \Rightarrow (s_0' \sim_{\mathcal{I}_0,\mathcal{I}_1} s_1')$$
$$\text{where } s_0' = f_0(s_0, \boldsymbol{x}, \boldsymbol{b}_0) \text{ and } s_1' = r(f_1(s_1, \boldsymbol{x}, \boldsymbol{b}_1), \boldsymbol{true}).$$

Furthermore, systems may run background operations that do not change the externally visible state of a system (i.e., no-ops), such as garbage collection.

Definition 5 (No-op)**.** Function f with a recovery function r is a *no-op* if (1) r is idempotent, and (2) the following holds:

$$\forall s_0, s_1, \boldsymbol{x}, \boldsymbol{b}_1.\ (s_0 \sim_{\mathcal{I}_0,\mathcal{I}_1} s_1) \Rightarrow (s_0 \sim_{\mathcal{I}_0,\mathcal{I}_1} s_1')$$
$$\text{where } s_1' = r(f(s_1, \boldsymbol{x}, \boldsymbol{b}_1), \boldsymbol{true}).$$

With per-function crash refinement and no-ops, we can now define crash refinement for entire systems.

Definition 6 (System crash refinement)**.** Given two systems F_0 and F_1, and a recovery function r, F_1 is a *crash refinement* of F_0 if every function in F_1 with r is either a crash refinement of the corresponding function in F_0 or a no-op.

The rest of this section will describe Yggdrasil's components based on the definition of crash refinement.

3.2 The verifier

Given two file systems, F_0 and F_1, Yggdrasil's verifier checks that F_1 is a crash refinement of F_0 according to Definition 6. To do so, the verifier performs symbolic execution [6, 24] for each operation $f_i \in F_i$ to obtain an SMT encoding of the operation's output, $f_i(s_i, \boldsymbol{x}, \boldsymbol{b_i})$, when applied to a symbolic input $\boldsymbol{x}$ (represented as a bitvector), symbolic disk state s_i (represented as an uninterpreted function over bitvectors), and symbolic crash schedule $\boldsymbol{b_i}$ (represented as booleans). It then invokes the Z3 solver to check the validity of either the no-op identity (Definition 5) if f_1 is a no-op, or else the per-function crash refinement formula (Definition 4) for the corresponding functions $f_0 \in F_0$ and $f_1 \in F_1$.

To capture *all* execution paths in the SMT encoding of $f_i(s_i, \boldsymbol{x}, \boldsymbol{b_i})$, the verifier adopts a "self-finitizing" symbolic execution scheme [49], which simply unrolls loops and recursion *without* bounding the depth. Since this scheme will fail to terminate on non-finite code, the verifier requires file systems to be implemented in a finite way: for instance, loops must be bounded [50]. In our experience (further discussed in §4), the finiteness requirement does not add much programming burden.

To prove the validity of the per-function crash refinement formula, the verifier uses Z3 to check if the formula's negation is unsatisfiable. If so, the result is a proof that f_1 is a crash refinement of f_0. Otherwise, Z3 produces a *model* of the formula's negation, which represents a concrete counterexample to crash refinement: disk states s_0 and s_1, an input $\boldsymbol{x}$, and a crash schedule $\boldsymbol{b_1}$, such that $s_0 \sim_{\mathcal{I}_0,\mathcal{I}_1} s_1$ but there is no crash schedule $\boldsymbol{b_0}$ that satisfies $f_0(s_0, \boldsymbol{x}, \boldsymbol{b_0}) \sim_{\mathcal{I}_0,\mathcal{I}_1} f_1(s_1, \boldsymbol{x}, \boldsymbol{b_1})$.

Checking the satisfiability of the negated crash refinement formula in Definition 4 requires reasoning about quantifiers. In general, such queries are undecidable. In our case, the problem is decidable because the quantifiers range over finite domains, and the formula is expressed in a decidable combination of decidable theories (i.e., equality with uninterpreted functions and fixed-width bitvectors) [51]. Moreover, Z3 can solve this problem in practice because the crash schedule $\boldsymbol{b_0}$, which is a set of boolean variables, is the only universally quantified variable in the negated formula. As many file system specifications have simple semantics, the crash schedule $\boldsymbol{b_0}$ has few boolean variables—often only one (e.g., the transaction in §2.1)—which makes the reasoning efficient.

The verifier's symbolic execution engine supports all regular Python code with concrete (i.e., non-symbolic) values. For symbolic values, it supports booleans, fixed-width integers, maps, and lists of concrete length, as well as regular control flow including conditionals and loops, but no exceptions or coroutines. It does not support symbolic execution into C library code.

3.3 The counterexample visualizer

To make counterexamples to validity easier to understand, Yggdrasil provides a visualizer for the asynchronous disk model. Given a counterexample model of the formula in Definition 4, the visualizer produces concrete disk event traces (e.g., see §2.3) as follows. First, it uses the crash schedule $\boldsymbol{b_1}$ to identify the boolean variable *on* that indicates where the system crashed, and relates that location to the implementation source code with a stack trace. Second, it evaluates the boolean *sync* variables that indicate whether a write is synchronized to disk, and prints out the pending writes with their corresponding source locations to help identify unintended reorderings. Yggdrasil also allows programmers to supply their own plugin visualizer for data structures specific to their file system images. We found this facility useful when developing YminLFS and Yxv6.

3.4 The optimizer

The Yggdrasil optimizer improves the run-time performance of implementation code. Yggdrasil treats the optimizer as untrusted and re-verifies the optimized code it generates. This simple design, made possible by push-button verification, allows programmers to plug in custom optimizations without the burden of supplying a correctness proof. We provide one built-in optimization that greedily removes disk flush operations (see §2.4), implemented by rewriting the Python abstract syntax tree.

4 The Yxv6 file system

The section describes the design, implementation, and verification of the Yxv6 journaling file system. At a high level, verifying the correctness of Yxv6 requires Yggdrasil to obtain an SMT encoding of both the specification and implementation through symbolic execution, and to invoke an SMT solver to prove the crash refinement theorem. A simple approach, used by YminLFS in §2, is to directly prove crash refinement between the entire file system specification and implementation. However, the complexity of Yxv6 makes such a proof intractable for state-of-the-art SMT solvers. To address this issue, Yxv6 employs a modular design enabled by crash refinement to scale up SMT reasoning.

4.1 Design overview

Yxv6 uses crash refinement to achieve scalable SMT reasoning in three steps. First, to reduce the size of SMT encodings, Yxv6 stacks five layers of abstraction, each consisting of a specification and implementation, starting with an asynchronous disk specification (§4.2). We use Yggdrasil to prove crash refinement theorems for each layer, showing that each correctly implements its specification. Upper layers then use the specifications of lower layers, rather than their implementations, in order to accelerate verification. This layered approach effectively bounds the reasoning to a single layer at a time.

Second, many file system operations touch only a small part of the disk. To allow the SMT solver to exploit this locality, Yxv6 explicitly uses multiple separate disks rather than one. For example, by storing the free bitmap on a separate disk, the SMT solver can easily infer that updating it does not affect the rest of the file system. We then prove crash refinement from this multi-disk system to a more space-efficient file system that uses only a single disk (§4.3). The result of these first two steps is Yxv6+sync, a *synchronous* file system that commits a transaction for each system call (by forcing the log to disk), similar to xv6 [14] and FSCQ [7].

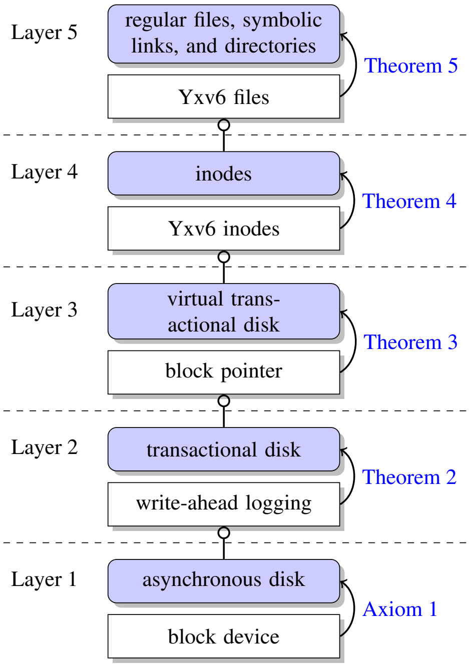


Figure 3: The stack of layers of Yxv6. Within each layer, a shaded box represents the specification; a (white) box represents the implementation; and the implementation is a crash refinement of its specification, denoted using an arrow. Each implementation (except for the lowest layer) builds on top of a specification from the layer below, denoted using a circle.

Finally, for better run-time performance, we implement an optimized variant of Yxv6+sync that groups multiple system calls into one transaction [19] and commits only when the log is full or upon fsync. We prove the resulting file system, called Yxv6+group_commit, is a crash refinement of Yxv6+sync with a more relaxed crash consistency model (§4.4).

4.2 Stacking layers of abstraction

Figure 3 shows the five abstraction layers of Yxv6. Each layer consists of a specification and an implementation that is written using a lower-level specification. We describe each of these layers in turn.

Layer 1: Asynchronous disk. The lowest layer of the stack is a specification of an asynchronous disk. This specification comprises the asynchronous disk model we used in §2.2 to implement YminLFS. Since the implementation of a physical block device is opaque, we assume the specification correctly models the block device (i.e., the specification is more conservative and allows more behavior than real hardware), as follows:

Axiom 1. A block device is a crash refinement of the asynchronous disk specification.

Layer 2: Transactional disk. The next layer introduces the abstraction of a transactional disk, which manages multiple separate data disks, and offers the following operations:

- d.begin_tx() starts a transaction;
- d.commit_tx() commits a transaction;
- d.write_tx(j, a, v) adds to the current transaction a write of value v to address a on disk j; and
- d.read(j, a) returns the value at address a on disk j.

The specification says that operations executed within the same transaction are atomic (i.e., all-or-nothing) and sequential (i.e., transactions cannot be reordered).

The implementation uses the standard write-ahead logging technique [19, 31]. It uses one asynchronous disk (from layer 1) for the log, and a set of asynchronous disks for data. Using a single transactional disk to manage multiple data disks allows higher layers to separate writes within a transaction (e.g., updates to data and inode blocks will not interfere), which helps scale SMT reasoning; §4.3 refines the multiple disks to one.

The implementation is parameterized by the transaction size limit k (i.e., the maximum number of writes in one transaction). The log disk uses a fixed number of blocks, determined by k, as a header to store log entry addresses, and the remaining blocks to store log entry data. The first entry in the first header block is a counter of log entries; the consistency invariant for the transactional disk layer says that this counter is always zero after recovery. The Yxv6+sync file system sets $k = 10$, while Yxv6+group_commit sets $k = 511$. For each of these settings, we prove the following theorem:

Theorem 2. The write-ahead logging implementation is a crash refinement of the transactional disk specification.

Layer 3: Virtual transactional disk. The specification of the virtual transactional disk is similar to that of the transactional disk, but instead uses 64-bit *virtual disk addresses* [22]. Each virtual address can be mapped to a physical disk address or unmapped later; reads and writes are valid for mapped addresses only. We will use this abstraction to implement inodes in the upper layer.

The virtual transactional disk implementation uses the standard block pointers approach. It uses one transactional disk managing at least three data disks: one to store the free block bitmap, another to store direct block pointers, and the third to store both data and singly indirect block pointers (higher layers will add additional disks). The free block bitmap disk stores only one bit in each of its blocks, which simplifies SMT reasoning but wastes disk space; §4.3 will refine it to a more space-efficient version.

The implementation relies on two consistency invariants: (1) the mapping from virtual disk addresses to physical disk addresses is injective (i.e., each physical address is mapped at most once), and (2) if a virtual disk address is mapped to physical address a, the a^{th} bit in the block bitmap must be marked as used. We use these invariants to prove the following theorem:

Theorem 3. The block pointer implementation is a crash refinement of the virtual transactional disk specification.

Layer 4: Inodes. The fourth layer introduces the abstraction of inodes. Each inode is uniquely identified using a 32-bit inode number. The specification maps an inode number to 2^{32} blocks, and to a set of metadata such as size, mtime, and mode.

The implementation is straightforward thanks to the virtual transactional disk specification. It simply splits the 64-bit virtual disk address space into 2^{32} ranges, and each inode takes one range, which has 2^{32} "virtual" blocks, similar to NVMFS/DFS [22]. Inode metadata resides on a separate disk managed by the virtual transactional disk (which now has four data disks). There are no consistency invariants in this layer. We prove the following theorem:

Theorem 4. The Yxv6 inode implementation is a crash refinement of the inode specification.

Layer 5: File system. The top layer of the file system is an extended version of FSSpec given in §2, with regular files, directories, and symbolic links.

The implementation builds on top of the inode specification, using a separate inode bitmap disk and another for orphan inodes. Both are managed by the virtual transactional disk (which now has six data disks plus the log disk, giving a total of seven disks). There are two consistency invariants: (1) if an inode is not marked as used in the inode bitmap disk, its size must be zero in the metadata; and (2) if an inode has n blocks, no "virtual" block larger than n is mapped. Using these invariants, we prove the final crash refinement theorem:

Theorem 5. The Yxv6 implementation of files is a crash refinement of the specification of regular files, symbolic links, and directories.

Finitization. The Yggdrasil verifier requires Yxv6 operations to be finite, as mentioned in §3.2. Most file system operations satisfy this requirement, as they use only a small number of disk reads and writes. For example, moving a file involves updating only the source and destination directories. However, there are two exceptions.

First, search-related procedures, such as finding a free bit in a bitmap, may need to read many blocks. We choose *not* to verify the bit-finding algorithm, but instead adopt the idea of validation [38, 46, 48] to implement such search algorithms. The validator, which we do verify, simply checks that an index returned by the search is indeed marked free in the bitmap and if not, fails the operation with an error code. We use similar

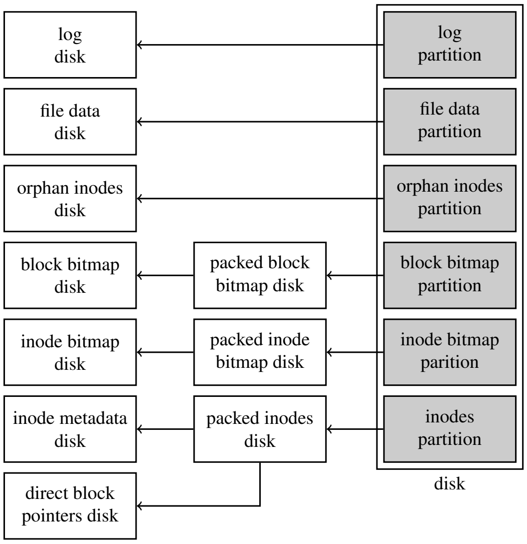


Figure 4: The refinement of disk layout of the Yxv6 file system, from multiple disks to a single disk. The arrows $A \leftarrow B$ denote that B is a crash refinement of A.

strategies for directory entry lookup. This approach allows us to treat search procedures as a black box, absolving the SMT solver from the need to reason about the many paths through the algorithm.

The second case is unlinking a file, as freeing all its data blocks needs to write potentially many blocks. To finitize this operation, our implementation simply moves the inode of the file into a special orphan inodes disk, which is a finite operation, and relies on a separate garbage collector to reclaim the data blocks at a later time. We further prove that reclamation is a no-op (as per the definition in §3.1), as freeing a block referenced by the orphan inodes disk does not affect the externally visible state of the file system. We will summarize the trade-offs of validation in §4.5.

4.3 Refining disk layouts

Theorem 5 gives a file system that runs on seven disks: the write-ahead log, the file data, the block and inode bitmaps for managing free space, the inode metadata, the direct block pointers, and the orphan inodes. Using separate disks scales SMT reasoning, but it has two downsides. First, the two bitmaps use only one bit per block and the inode metadata disk stores one inode per block, wasting space. Second, requiring seven disks makes the file system difficult to use. We now prove with crash refinement that it is correct to pack these disks into one disk (Figure 4) similar to the xv6 file system [14].

Intuitively, it is correct to pack multiple blocks that store data sparsely into one with a dense representation, because the packed disk has the same or fewer possible disk states. For instance, bitmap disks used in §4.2 store one bit per block; the n-th bit of the bitmap is stored in the lowest bit of block n. On the other hand, a *packed* bitmap disk stores $4\,\text{KB} \times 8 = 2^{15}$ bits per block, and the n-th bit is stored in bit $n \bmod 2^{15}$ of block $n/2^{15}$. Clearly, using the packed bitmap is a crash refinement of the sparse one. The same holds for using packed inodes. Similarly, a single disk with multiple non-overlapping partitions exhibits fewer states than multiple disks; for example, a flush on a single disk will flush all the partitions, but not for multiple disks. Combining these packing steps, we prove the following theorem:

Theorem 6. The Yxv6 implementation using seven non-overlapping partitions of one asynchronous disk, with packed bitmaps and inodes, is a crash refinement of that using seven asynchronous disks.

4.4 Refining crash consistency models

Theorem 6 gives a *synchronous* file system that commits a transaction for each system call. This file system, which we call Yxv6+sync, incurs a slowdown as it flushes the disk frequently (see §8 for performance evaluation). The Yxv6+group_commit file system implements a more relaxed crash consistency model [5, 37]. Unlike Yxv6+sync, its write-ahead logging implementation groups multiple transactions together [19].

Intuitively, doing a single combined transaction produces fewer possible disk states compared to two separate transactions, as in the latter scheme the system can crash in between the two and expose the intermediate state. We prove the following theorem:

Theorem 7. Yxv6+group_commit is a crash refinement of Yxv6+sync.

4.5 Summary of design trade-offs

Unlike conventional journaling file systems, the first Yxv6 design in §4.2 uses multiple disks. To decide the number of disks, we adopt a simple guideline: whenever a part of the disk is *logically* separate from the rest of the file system, such as the log or the free bitmap, we assign a separate disk for that part. In our experience, this is effective in scaling up SMT reasoning.

Yxv6's final on-disk layout closely resembles that of the xv6 and FSCQ file systems. One notable difference is that Yxv6 uses an orphan inodes partition to manage files that are still open but have been unlinked, similarly to the orphan inode list [21] in ext3 and ext4. This design ensures correct atomicity behavior of `unlink` and `rename`, especially when running with FUSE, which xv6 and FSCQ do not guarantee.

Another difference to FSCQ is that Yxv6 uses validation instead of verification in managing free blocks and inodes. Although the resulting allocator is safe, it does not guarantee that block or inode allocation will succeed when there is enough space, treating such failures as a quality-of-service issue.

5 Beyond file systems

Although we designed Yggdrasil for writing verified file systems, the idea of crash refinement generalizes to applications that use disks in other ways. This section describes two examples: Ycp, a file copy utility; and Ylog, a persistent log data structure.

The Ycp file copy utility. Like the Unix cp utility, Ycp copies the contents of one file to another. Unlike cp, it has a formal specification: if the copy operation succeeds, the file system is updated so that the target file contains the same data as the source file; if it fails due to a system crash or an invalid target (e.g., a directory or a symbolic link), the file system is unchanged.

The implementation of Ycp uses the Yxv6 file system specification (Figure 3). It follows a common atomicity pattern: (1) create a temporary file, (2) write the source data to it, and (3) rename it to atomically create the target file. There is no consistency invariant as Ycp uses file system operations and is independent of disk layout.

We verify that the implementation of Ycp is a crash refinement of its specification using Yggdrasil. This shows that Yggdrasil and Yxv6's specification are useful for reasoning about application-level correctness.

The Ylog persistent log. Ylog is a verified implementation of the persistent log from the Arrakis operating system [36]. The Arrakis log is designed to provide an efficient storage API with strong atomicity and persistence guarantees. The core logging operation is a multi-block append, which extends an on-disk log with entries that can span multiple blocks. This append operation must appear to be both atomic and immediately persistent, even in the presence of crashes.

The Arrakis persistent log was originally designed to run on top of an LSI Logic MegaRAID SAS-3 3108 RAID controller with a battery-backed cache. We therefore chose to implement Ylog on top of a *synchronous* disk model, which does not reorder writes and matches the behavior of the RAID controller. Ylog uses the same on-disk layout as Arrakis: the first block (i.e., superblock) contains metadata, such as the number of entries and a pointer to the end of the log, followed by blocks that contain the data of each entry.

When comparing Ylog's implementation with that of Arrakis, we discovered two bugs in the Arrakis persistent log: its crash recovery logic was *not* idempotent, and the log could end up with garbage data if the system crashed again during recovery. The bugs were reported to and confirmed by the Arrakis developers.

6 Discussion

This section discusses the limitations of Yggdrasil, as well as our experience using and designing the toolkit.

component	specification	implementation	consistency inv
Yxv6	250	1,500	5
YminLFS	25	150	5
Ycp	15	45	0
Ylog	35	60	0
infrastructure	–	1,500	–
FUSE stub	–	250	–

Figure 5: Lines of code for the Yggdrasil toolkit and storage systems built using it, excluding blank lines and comments.

Limitations. Yggdrasil reasons about single-threaded code, so file systems written using Yggdrasil do not support concurrency. Cython [3], Yggdrasil's Python-to-C compiler, is unverified, although we have not yet encountered any bugs in the development.

Yggdrasil relies on SMT solvers for automated reasoning, and is limited to first-order logic. It is less expressive than interactive theorem provers such as Coq or Isabelle, although our experience shows that it is sufficient for writing and verifying file systems like Yxv6 based on crash refinement.

Since the Z3 solver is at the core of Yggdrasil, its correctness is critical. To understand this risk, we ran the Yxv6 verification using every buildable snapshot of the Z3 Git repository over the past three years, a total of 1,417 versions. We also used two other SMT solvers, Boolector [32] and MathSAT 5 [9], for cross-checking. We did not observe any inconsistent results.

The Yxv6 file system lacks several modern file system features, such as extents and delayed allocation in ext4. Compared to hand-written file system checkers, its fsck tool is generated by Yggdrasil and guaranteed to detect any violations of consistency invariants, but it cannot repair corrupted file systems.

Lessons learned. Bitvector operations and reasoning about non-determinism (e.g., crashes) are common in file system implementations. These characteristics motivated us to formulate file system verification as an SMT problem, exploiting the fully automated decision procedures for the theories of bitvectors and uninterpreted functions. In addition, using SMT enables Yggdrasil to produce and visualize counterexamples; we find this ability useful for tracking subtle file system bugs during development, especially corner cases such as overflows and missing flushes [27].

In earlier development of Yggdrasil, we struggled to find a disk representation for scalable SMT reasoning. We explored several approaches, such as a lazy list of symbolic blocks (e.g., EXE [53]) and the theory of arrays, all resulting in a verification bottleneck.

Yggdrasil represents a disk using uninterpreted functions that map a block address and an in-block offset to a 64-bit integer. This two-level map helped to scale up verification. Mapping to 64-bit integers also allowed

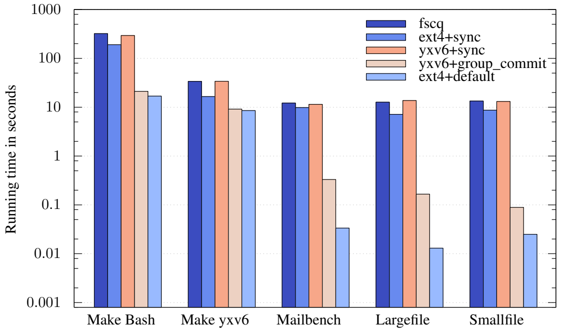


Figure 6: Performance of file systems on an SSD, in seconds (log scale; lower is better).

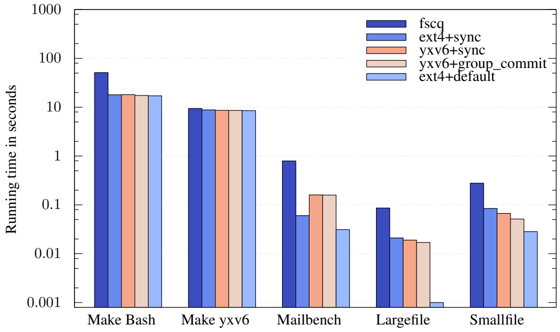


Figure 7: Performance of file systems on a RAM disk, in seconds (log scale; lower is better).

Yggdrasil to generate efficient C code. The idea of separating logical and physical data representations using crash refinement further reduced the verification time by orders of magnitude. As we will show in §8, verifying Yxv6+sync's theorems took less than a minute, thanks to Z3's efficient decision procedures, whereas Coq took 11 hours to check the proofs of FSCQ [7] (which has similar features to Yxv6+sync).

Crash refinement requires programmers to design a system as a state machine and implement each operation in a finite way. File systems fit well into this paradigm. We have used crash refinement in several contexts: to stack layers of abstraction, to pack multiple blocks or disks, and to relax crash consistency models. Crash refinement does not require advanced knowledge of program logics (e.g., separation logic [41] in FSCQ), and is amenable to automated SMT reasoning.

7 Implementation

Figure 5 lists the code size of the file systems and other storage applications built using Yggdrasil, the common infrastructure code, and the FUSE boilerplate. In total, they consist of about 4,000 lines of Python code.

8 Evaluation

This section uses Yxv6 as a representative example to evaluate file systems built using Yggdrasil. We aim to answer the following questions:

- Does Yxv6 provide end-to-end correctness?
- What is the run-time performance?
- What is the verification performance?

Unless otherwise noted, all experiments were conducted on a 4.0 GHz quad-core Intel i7-4790K CPU running Linux 4.4.0.

Correctness. We tested the correctness of Yxv6 as follows. First, we ran it on existing benchmarks. Both Yxv6+sync and Yxv6+group_commit passed the `fsstress` tests from the Linux Test Project [26]; they also passed the SibylFS POSIX conformance tests [42], except for incomplete features such as hard links or extended attributes. Second, we have been using Yxv6 to self-host Yggdrasil's development since early March, including the writing of this paper; our experience is that it is reliable for daily use. Third, we applied the disk block enumerator from the Ferrite toolkit [5] (similar to the Block Order Breaker [37]) to cross-check that the file system state was consistent after a crash and recovery.

To test the correctness of Yxv6's `fsck`, we manually corrupted file system images by overwriting them with random bytes; Yxv6's `fsck` was able to detect corruption in all these cases.

Run-time performance. To understand the run-time performance of Yxv6, we ran a set of five benchmarks similar to those used in FSCQ [7]: compiling the source code of `bash` and Yxv6, running a mail server from the sv6 operating system [10], and the LFS benchmark [44].

We compare the two Yxv6 variants against the verified file system FSCQ and the ext4 file system in two configurations: its default configuration (i.e., `data=ordered`), and with `data=journal+sync` options, which together are similar to Yxv6+sync. Although Yxv6's implementation is closest to xv6, we excluded xv6's performance numbers as it crashed frequently on three benchmarks and did not pass the `fsstress` tests.

Figure 6 shows the on-disk performance with all the file systems running on a Samsung 850 PRO SSD. The y-axis shows total running time in seconds (log scale). We see that Yxv6+sync performs similarly to FSCQ and to ext4's slower configuration. Yxv6+group_commit, which groups several operations into a single transaction, outperforms those file systems by 3–150× and is on average within 10× of ext4's default configuration.

To understand the CPU overhead, we repeated the experiments using a RAM disk, as shown in Figure 7. The two variants of Yxv6 have similar performance numbers. They both outperform FSCQ, and are close in performance to ext4 (except for the `largefile` benchmark). We believe the reason is that Yxv6 benefits from Yggdrasil's Python-to-C compiler, while FSCQ's performance is affected by its use of Haskell code extracted from Coq.

Verification performance. As we mentioned in §6, the total verification time for Yxv6+sync is under a minute on a single core. It achieved this verification performance due to Z3's efficient SMT solving and the use of crash refinement in the file system.

Verifying Yxv6+group_commit took a longer time, because it is parameterized to use larger transactions (see §4.2). It finished within 1.6 hours using 24 cores (Intel Xeon 2.2 GHz), approximately 36 hours on a single core.

9 Related work

Verified file system implementations. Developers looking to build and verify file systems have primarily turned to interactive theorem provers such as Coq [11] and Isabelle [33]. Our approach is most similar to FSCQ [7], a verified crash-safe file system developed in Coq. Their proof shows that after reboot, FSCQ's recovery routines will correctly recover the file system state without data loss. These theorems are stated in *crash Hoare logic*, which extends Hoare logic with support for crash conditions and recovery procedures. Our approach also bears similarities to Flashix [16, 47], another verified crash-safe file system. The Flashix proof consists of several refinements from the POSIX specification layer down to an implementation which can be extracted to Scala. These refinements are proved in the KIV interactive theorem prover in terms of abstract state machines.

Compared to these examples, Yggdrasil's push-button verification substantially lowers the proof burden. Yggdrasil can verify the Yxv6 implementation given only the specifications and five consistency invariants. This ease of verification, together with richer debugging support, also helped us implement several optimizations in Yxv6 that make its performance 3–150× faster than FSCQ and within 10× of ext4.

COGENT [2] takes a different approach to building verified file systems, defining a new restricted language together with a certified compiler to C code. The COGENT language rules out several common sources of errors, such as memory safety and memory leaks, reducing the verification proof burden. We believe Yggdrasil and COGENT to be complimentary: on one hand, COGENT provides certified extraction to C code which could replace Yggdrasil's unverified extraction from Python; on the other hand, Yggdrasil's crash refinement strategy could help COGENT to produce more automated proofs.

File system specifications and crash consistency. Several projects have developed formal specifications of file systems. SibylFS [42] is an effort to formalize POSIX interfaces and test implementation conformance. But because POSIX file system interfaces underspecify allowed crash behavior, so does the SibylFS formalization. Commuter [10] formalizes the commutativity of POSIX interface calls to study scalability, but as with SibylFS, the formalization does not consider crashes.

Modern file systems adopt various crash recovery strategies, including write-ahead logging (or journaling) [19, 31], log-structured file systems [44], copy-on-write (or shadowing) [4, 43], and soft updates [18, 29]. This diversity complicates reasoning about application-level crash safety. Pillai et al. [37] and Zheng et al. [55] surveyed the crash safety of real-world applications, finding many crash-safety bugs despite extensive engineering effort to tolerate and recover from crashes. Bornholt et al. [5] formalized the crash guarantees of modern file systems as *crash-consistency models*, to help application writers provide crash safety. A formally verified file system can provide these models as an artifact of the verification process. Yggdrasil's crash refinement strategy helps to abstract low-level implementation details out of these application-facing models.

Bug-finding tools. Rather than building a new verified file system, several existing projects focus on finding bugs in existing file systems. FiSC [54] and eXplode [52] use model checking to find consistency bugs. ELEVEN82 [25] is a bug-finding tool for "recoverability bugs," where a system can crash in such a way that even after recovery, the file system is left in a state not reachable by any crash-free execution. Yggdrasil is complementary to these tools: ELEVEN82's automata-based bug detection allows it to explore complex optimizations, while Yggdrasil provides proofs not only of crash safety but of functional correctness.

10 Conclusion

Yggdrasil presents a new approach for building file systems with the aid of push-button verification. It guarantees correctness through a definition of file system crash refinement that is amenable to efficient SMT solving. It introduces several techniques to scale up automated verification, including the stack of abstractions and the separation of data representations. We believe that this is a promising direction since it provides a strong correctness guarantee with a low proof burden. All of Yggdrasil's source code is publicly available at `http://locore.cs.washington.edu/yggdrasil/`.

Acknowledgments

We thank Helga Gudmundsdottir, Niel Lebeck, Hank Levy, Haohui Mai, Qiao Zhang, the anonymous reviewers, and our shepherd, Petros Maniatis, for their feedback. This work was supported in part by DARPA under contract FA8750-16-2-0032 and by a gift from the VMware University Research Fund.

References

[1] R. Alagappan, V. Chidambaram, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Beyond storage APIs: Provable semantics for storage stacks. In *Proceedings of the 15th Workshop on Hot Topics in Operating Systems (HotOS)*, Kartause Ittingen, Switzerland, May 2015.

[2] S. Amani, A. Hixon, Z. Chen, C. Rizkallah, P. Chubb, L. O'Connor, J. Beeren, Y. Nagashima, J. Lim, T. Sewell, J. Tuong, G. Keller, T. Murray, G. Klein, and G. Heiser. COGENT: Verifying high-assurance file system implementations. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 175–188, Atlanta, GA, Apr. 2016.

[3] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. Seljebotn, and K. Smith. Cython: The best of both worlds. *Computing in Science Engineering*, 13(2): 31–39, Mar.–Apr. 2011. http://cython.org/.

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

[5] J. Bornholt, A. Kaufmann, J. Li, A. Krishnamurthy, E. Torlak, and X. Wang. Specifying and checking file system crash-consistency models. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 83–98, Atlanta, GA, Apr. 2016.

[6] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 209–224, San Diego, CA, Dec. 2008.

[7] H. Chen, D. Ziegler, T. Chajed, A. Chlipala, M. F. Kaashoek, and N. Zeldovich. Using Crash Hoare Logic for certifying the FSCQ file system. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, Monterey, CA, Oct. 2015.

[8] H. Chen, D. Ziegler, A. Chlipala, M. F. Kaashoek, E. Kohler, and N. Zeldovich. Specifying crash safety for storage systems. In *Proceedings of the 15th Workshop on Hot Topics in Operating Systems (HotOS)*, Kartause Ittingen, Switzerland, May 2015.

[9] A. Cimatti, A. Griggio, B. Schaafsma, and R. Sebastiani. The MathSAT5 SMT solver. In *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 93–107, Rome, Italy, Mar. 2013.

[10] A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–17, Farmington, PA, Nov. 2013.

[11] Coq development team. *The Coq Proof Assistant Reference Manual, Version 8.5pl2*. INRIA, July 2016. http://coq.inria.fr/distrib/current/refman/.

[12] J. Corbet. Thoughts on the ext4 panic, Oct. 2012. https://lwn.net/Articles/521803/.

[13] J. Corbet. A tale of two data-corruption bugs, May 2015. https://lwn.net/Articles/645720/.

[14] R. Cox, M. F. Kaashoek, and R. T. Morris. Xv6, a simple Unix-like teaching operating system, 2016. http://pdos.csail.mit.edu/6.828/xv6.

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

[16] G. Ernst, J. Pfähler, G. Schellhorn, and W. Reif. Inside a verified flash file system: Transactions & garbage collection. In *Proceedings of the 7th Working Conference on Verified Software: Theories, Tools and Experiments*, San Francisco, CA, July 2015.

[17] FUSE. Filesystem in userspace, 2016. https://github.com/libfuse/libfuse.

[18] G. R. Ganger and Y. N. Patt. Metadata update performance in file systems. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation (OSDI)*, pages 49–60, Monterey, CA, Nov. 1994.

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

[20] V. Henson. The many faces of fsck, Sept. 2007. https://lwn.net/Articles/248180/.

[21] V. Henson, Z. Brown, T. Ts'o, and A. van de Ven. Reducing fsck time for ext2 file systems. In *Proceedings of the Linux Symposium*, pages 395–408, Ottawa, Canada, June 2006.

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

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

[24] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[25] E. Koskinen and J. Yang. Reducing crash recoverability to reachability. In *Proceedings of the 43rd ACM Symposium on Principles of Programming Languages (POPL)*, pages 97–108, St. Petersburg, FL, Jan. 2016.

[26] LTP. Linux Test Project, 2016. http://linux-test-project.github.io/.

[27] L. Lu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and S. Lu. A study of Linux file system evolution. *ACM Transactions on Storage*, 10(1): 31–44, Jan. 2014.

[28] W. M. McKeeman. Peephole optimization. *Communications of the ACM*, 8:443–444, July 1965.

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

[30] M. K. McKusick and T. J. Kowalski. Fsck—the UNIX file system check program. In *UNIX System Manager's Manual (SMM), 4.4 Berkeley Software Distribution*. University of California, Berkeley, Oct. 1996.

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

[32] A. Niemetz, M. Preiner, and A. Biere. Boolector 2.0. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 9:53–58, 2015.

[33] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer-Verlag, Feb. 2016.

[34] L. O'Connor, Z. Chen, C. Rizkallah, S. Amani, J. Lim, T. Murray, Y. Nagashima, T. Sewell, and G. Klein. Refinement through restraint: Bringing down the cost of verification. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 89–102, Nara, Japan, Sept. 2016.

[35] N. Palix, G. Thomas, S. Saha, C. Calvès, J. L. Lawall, and G. Muller. Faults in Linux: Ten years later. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 305–318, Newport Beach, CA, Mar. 2011.

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

[37] T. S. Pillai, V. Chidambaram, R. Alagappan, S. Al-Kiswany, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. All file systems are not created equal: On the complexity of crafting crash-consistent applications. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 433–448, Broomfield, CO, Oct. 2014.

[38] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *Proceedings of the 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 151–166, Lisbon, Portugal, Mar.–Apr. 1998.

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

[40] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. IRON file systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, pages 206–220, Brighton, UK, Oct. 2005.

[41] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, Copenhagen, Denmark, July 2002.

[42] T. Ridge, D. Sheets, T. Tuerk, A. Giugliano, A. Madhavapeddy, and P. Sewell. SibylFS: formal specification and oracle-based testing for POSIX and real-world file systems. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, pages 38–53, Monterey, CA, Oct. 2015.

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

[44] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–15, Pacific Grove, CA, Oct. 1991.

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

[46] H. Samet. Proving the correctness of heuristically optimized code. *Communications of the ACM*, 21 (7):570–582, July 1978.

[47] G. Schellhorn, G. Ernst, J. Pfähler, D. Haneberg, and W. Reif. Development of a verified flash file system. In *Proceedings of the ABZ Conference*, June 2014.

[48] T. Sewell, M. Myreen, and G. Klein. Translation validation for a verified OS kernel. In *Proceedings of the 2013 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 471–482, Seattle, WA, June 2013.

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

[50] L. Torvalds. Re: [patch] measurements, numbers about `CONFIG_OPTIMIZE_INLINING=y` impact, Jan. 2009. https://lkml.org/lkml/2009/1/9/497.

[51] C. M. Wintersteiger, Y. Hamadi, and L. de Moura. Efficiently solving quantified bit-vector formulas. In *Proceedings of the 10th Conference on Formal Methods in Computer-Aided Design*, pages 239–246, Lugano, Switzerland, Oct. 2010.

[52] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using model checking to find serious file system errors. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 273–287, San Francisco, CA, Dec. 2004.

[53] J. Yang, C. Sar, P. Twohey, C. Cadar, and D. Engler. Automatically generating malicious disks using symbolic execution. In *Proceedings of the 27th IEEE Symposium on Security and Privacy*, pages 243–257, Oakland, CA, May 2006.

[54] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. EXPLODE: A lightweight, general system for finding serious storage system errors. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 131–146, Seattle, WA, Nov. 2006.

[55] M. Zheng, J. Tucek, D. Huang, F. Qin, M. Lillibridge, E. S. Yang, B. W. Zhao, and S. Singh. Torturing databases for fun and profit. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 449–464, Broomfield, CO, Oct. 2014.