Performance Optimization of the VDFS Verified File System

by

Alexander V. Konradi

S.B., Massachusetts Institute of Technology, 2016

Submitted to the Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2017

Author . . . .

Department of Electrical Engineering and Computer Science

June 14, 2017

Certified by . . . .

M. Frans Kaashoek

Charles Piper Professor

Thesis Supervisor

Certified by . . . .

Nickolai Zeldovich

Associate Professor

Thesis Supervisor

Accepted by . . . .

Christopher J. Terman

Chairman, Masters of Engineering Thesis Committee

Performance Optimization of the VDFS Verified File System

by

Alexander V. Konradi

Submitted to the Department of Electrical Engineering and Computer Science
on June 14, 2017, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

Formal verification of software has become a powerful tool for creating software systems and proving their correctness. While such systems provide strong guarantees about their behavior, they frequently exhibit poor performance relative to their unverified counterparts. Verified file systems are not excepted, and their poor performance limits their utility. These limitations, however, are not intrinsic to verification techniques, but are the result of designing for proofs, not performance.

This thesis proposes a design for large files and in-memory caches that are amenable both to a high-performance implementation and proofs of correctness. It then describes their usage in VDFS, a verified high-performance file system with deferred durability guarantees. The evaluation of VDFS' performance shows that these additions measurably improve performance over previous verified file systems, and make VDFS competitive with unverified file system implementations. This thesis contributes proven implementation techniques for large files and in-memory caches that can be applied to increase performance of verified systems.

Thesis Supervisor: M. Frans Kaashoek
Title: Charles Piper Professor

Thesis Supervisor: Nickolai Zeldovich
Title: Associate Professor

Acknowledgments

There are many people who made this work possible, and without whom I would never have finished this thesis. I don't know that I can thank them enough, but I can certainly try.

First, Frans and Nickolai. Thank you for your constant guidance, especially when unsolicited, and for your unending patience in refining the work comprising (and the text of) this thesis. I can't tell you how much you've taught me over the last year, not just about formal verification or computer systems, but about myself as well. I know that wherever I go, the lessons you taught me will not be forgotten.

Thank you to Tej, for your help when I was first starting to work on VDFS. Without you, I would still be a systems programmer doing proofs, and would never have made it past my first theorem. Thank you for helping to introduce me to the wide world of formal verification, and for always being willing to discuss VDFS. Thank you too to Daniel, for pushing my comfort boundaries around FSCQ, and for always having a plan to follow. Though VDFS still extracts Haskell code, I look forward to seeing the additional performance gains that low-level extracted code can provide.

Finally, thank you to all the people who didn't contribute directly to this work, but who helped me keep going when the way ahead was bleak. Thank you to my girlfriend, Rachel, for putting up with me describing the minutiae of VDFS time after time. I don't think I would have finished this thesis without your encouragement. Thank you also to PDOS and the G9 crowd, for being a source of stability throughout all the ups and downs. I'll miss all of you more than you know as I leave MIT for the wider world.

Contents

List of Figures

Chapter 1

Introduction

In the world of software engineering, it is a generally accepted truism that, despite programmers' best efforts to the contrary, most software has bugs. While this is not a new phenomenon, it has received more attention in recent times as software systems have grown larger and more complex. New techniques and practices have been developed and incorporated into the standard practice to combat the growth of such errors, including automated testing, symbolic execution, and mandatory code audits. Like the software they seek to improve, these systems are imperfect: automated testing typically covers only a fraction of possible inputs, symbolic execution is impractical at large scales, and code audits require experienced developers who can still make mistakes.

More recently, as the computing power available to individuals has increased, the usage of formal software verification has become practical as another tool for fighting bugs. Instead of running code through a battery of test cases, formal verification seeks to rigorously define the expected behavior of code, and prove with mathematical certainty that its implementation meets this specification. While not widely used, formal verification has yielded some impressive results, including verified kernels (CertiKOS [1], seL4 [2]), optimizing compilers (CompCert [3]), and file systems (FSCQ [4], Yggdrasil [5]). While correct, these systems often have significant asterisks attached, including missing functionality or lackluster performance relative to their unverified counterparts. The goal of this thesis is to remove some of the limitations from, and improve the performance of, the FSCQ file system.

1.1 Background: FSCQ

FSCQ [4, 6] is a verified file system written using the Coq [7] proof assistant. As should be expected from a verified file system, FSCQ provides an implementation of a significant subset of the POSIX file system operations [8], and includes proven top-level specifications about each system call. These proofs are fully mechanized, allowing anyone with access to the source code to run Coq's proof checker and verify their validity.

Like most verified systems, FSCQ's proofs only address correctness, not performance. While an earlier prototype of FSCQ [4] supported only synchronous disk operations, the version that VDFS builds on fully implements deferred durability guarantees. Instead of being written to disk immediately, each operation is allowed to be persisted asynchronously, in-order, and is only guaranteed to be durable after a `sync()` operation. This deferred durability guarantee is similar to that provide by ext4 [9], a popular Linux file system. Like ext4, FSCQ exploits the asynchrony allowed by deferred durability to provide high I/O efficiency. However, due to high CPU overhead, FSCQ's actual execution speed is much slower than ext4, which limits the workloads it can handle.

1.2 Goals

This thesis seeks to address some of the functionality and performance issues of FSCQ, both by increasing the maximum supported file size by several orders of magnitude, and by addressing several performance issues through the addition of in-memory caches. This modified version of FSCQ with support for large files and caches will be referred to as VDFS.

1.2.1 Large files

The general expectation of file systems is, somewhat obviously, that they are capable of reliably storing and retrieving files. While important to the proof, the actual maximum size of the file is of little concern for research systems—both FSCQ and Yggdrasil only support files of up to 2MB. Unlike FSCQ, VDFS aims to target applications with more stringent requirements, like high-performance databases, which are written with deferred durability guarantees in mind and take full advantage of the speed increases they offer. Such systems, however, typically expect to be able to allocate much larger files on disk, on the order of hundreds of megabytes or gigabytes in size. VDFS addresses this limitation in FSCQ by extending the maximum file size to 513GB.

In FSCQ, VDFS, and other file systems, files are represented as indexed nodes, or *inodes*, where each inode logically represents a single file. Since each inode is relatively small, instead of storing all block numbers for a file in the inode, the inode stores a few direct block numbers, as well as a number of indirect blocks. Each indirect block contains several more block numbers, either for direct or additional indirect blocks. While FSCQ's inodes are limited to several direct block addresses and a single indirect block, VDFS follows the approach of most Unix file systems and extends this scheme by adding several levels of indirect blocks to each inode.

Unlike a typical file system, VDFS implements and proves specifications for operations on general n-indirect blocks. While it may seem counter-intuitive, this helps to simplify the implementation and reduce the proof burden. By instantiating n with indirection level 3, VDFS uses the generic code and proofs to succinctly implement and specify operations over concrete inodes with singly-, doubly-, and triply-indirect blocks. Though the generic approach requires reasoning about all indirection levels simultaneously, it is significantly more conducive to verification. While specialized implementations for each of the three indirection levels would require proofs over three cases, the general approach allows each function to be defined once, and allows proofs to be performed inductively with only a base and an inductive case for each.

In addition to the on-disk representation, FSCQ's code makes certain assumptions about the maximum size of files. In particular, the `fdatasync` operation in FSCQ, which ensures that all changes to the data blocks for a file are persisted to disk, runs in time proportional to the length of the file. This is acceptable for files which contain a maximum of approximately 512 blocks, but is less so for files whose maximum size is more than 2^{27} blocks.

To circumvent this potential performance issue, VDFS, in addition to caching dirty blocks in-memory before writing them out to disk, keeps a list of dirty blocks for each file. This allows it to implement `fdatasync` with a running time proportional to the number of dirty blocks in the file, thus avoiding unnecessarily penalizing operations on large files.

1.2.2 In-memory caches

While support for larger files increases the applicability of VDFS, it does not address FSCQ's overall slow execution. Some of the performance penalty seen by FSCQ can be attributed to its extraction to Haskell, a functional language that Coq supports natively. Much of it, however, can be attributed to naïve implementations of various operations in FSCQ that, while slow, are easy to prove correct. Though some of the performance issues can be addressed by small code changes—iterating forwards over lists instead of backwards, for instance—many require larger modifications. This thesis extends FSCQ with several in-memory caches, which are designed to prevent unnecessary searching and duplication of work.

Like the on-disk state of the file system, each of these caches has an invariant over its contents that must be maintained throughout execution. These invariants typically state that the cache reflects some convenient transformation of the on-disk state, such that the elements of the cache can be used in place of the results of disk operations. This requires that every operation that updates the disk also update the cache, and that the cache invariant still hold afterwards. While this adds several additional proof obligations, it is necessary for correctness. To aid in the proofs of invariant maintenance, these caches share several common properties.

The first property is that no cache contents are required for correctness: if a cache look-up fails, the result is the same as if the cache didn't exist. This ensures that it is always safe to remove any or all elements from the cache, and that an empty cache satisfies its invariant. In addition, it allows code that modifies state to update cached data or remove it, allowing both write-through and invalidation implementations since both satisfy the invariant.

The second property is that the caches are treated as local and opaque. Caching is implemented as a separate layer on top of existing code, thus decoupling the responsibility of cache maintenance from operation implementation. This allows transparent addition of caching layers without affecting other layers or invariants.

Finally, the cache invariants specify absolute consistency between present elements and the on-disk state. Instead of treating cached elements as hints which must first be verified to ensure correctness, the invariants ensure that cached results can be used to satisfy requests without any additional checks.

1.3 Outline and Contributions

Chapter 3 describes FSCQ, upon which VDFS is built, and provides background context for this work. The contributions of this thesis are presented as follows:

- Chapter 4 describes a design and representation for large files and in-memory caches that is conducive to verification and performance.

- Chapter 5 details how these optimizations are implemented in VDFS, the first high-performance verified deferred-durability file system.

- Chapter 6 contains an evaluation of VDFS' performance relative to other file systems on a variety of disk configurations that demonstrates that VDFS is significantly faster than FSCQ and competitive with unverified file systems.

Chapter 2

Related work

The work described in this thesis builds directly on the FSCQ file system developed in Haogang Chen's doctoral thesis [6], and constitutes a set of optimizations applied to improve its performance and usability. By implementing common file system techniques in a verified setting, VDFS provides increased performance over FSCQ and other previous verified file systems.

2.1 File-system verification

While formal verification of programs is not new, there has been significant recent progress in the area of verified file systems. Cogent [10] is a verified file system that supports deferred durability and produces highly efficient executable code, but lacks a specification and proof of the entire file system. FSCQ [6] is another verified file system; while it implements deferred durability and provides a top-level specification of correctness, its run-time performance is poor. Yggdrasil [5], the most recent result, uses a state-of-the-art SMT solver to automatically verify that a file system meets its specifications without manual proofs. It produces fast executable code, though it is unable to provide a top-level proof for its example deferred-durability file system. While the VDFS implementation builds on top of FSCQ, it could benefit from the use of an SMT solver to reduce manual proof effort as Yggdrassil does. VDFS also uses the same extraction system as FSCQ; though it would increase the proof burden, VDFS could likely reduce execution overheads by using a lower-level or domain specific language like Cogent.

2.2 Indirect blocks

The use of indirect blocks to store file system data is not a new one; they were used in the file system for the original Unix time-sharing system [11], which was released in 1974. While verified file systems like FSCQ and Yggdrasil make use of indirect blocks, and Cogent uses doubly indirect blocks, none of them, nor other prior work on file system verification [12–20] use VDFS' technique of generalizing the implementation for all indirection levels.

2.3 In-memory caching

Modern file systems like ext4 perform extensive caching of on-disk information in-memory [21] with the intent of reducing file system I/O operations. While previous verified file systems, including FSCQ, cache entire blocks in memory, they do not use VDFS' approach of retaining deserialized structures in memory for faster lookups. Whereas some file systems preload caches predictively [22], VDFS implements caches lazily, loading data only when necessary.

Chapter 3

Background

One of the main contributions of FSCQ is its development and use of Crash Hoare Logic (CHL) [4] to reason about the flow of programs and their possible crash states. In CHL, theorems about execution are stated as quadruples of the form $\{\{pre, post, crash, prog\}\}$, where *post* is condition that holds after executing *prog*, if the condition *pre* held before execution. *crash* is, then, the condition that holds on any state produced by a crash during the execution of *prog*. By matching pre- and post-conditions together, these CHL quadruples can be chained to produce proofs about larger programs. Indeed, this is exactly how FSCQ's correctness proofs work—they chain together these execution theorems to produce specifications for entire file system operations.

To describe the pre-, post-, and crash conditions, FSCQ uses predicates over the on-disk contents. Since on-disk contents are cumbersome to reason about directly, FSCQ makes use of multiple logical address spaces to capture program and disk state at various levels of abstraction. The directory-level representation of the file system, for example, is separate from the block-level address space of the physical disk, even though the directories have a physical representation on disk. The connections between these address spaces are represented as invariants which are then maintained by the program execution. FSCQ makes these levels of abstraction concrete by splitting the actual implementation into corresponding layers, each with their own functions and specifications. These layers are described below, and their relationships are de-

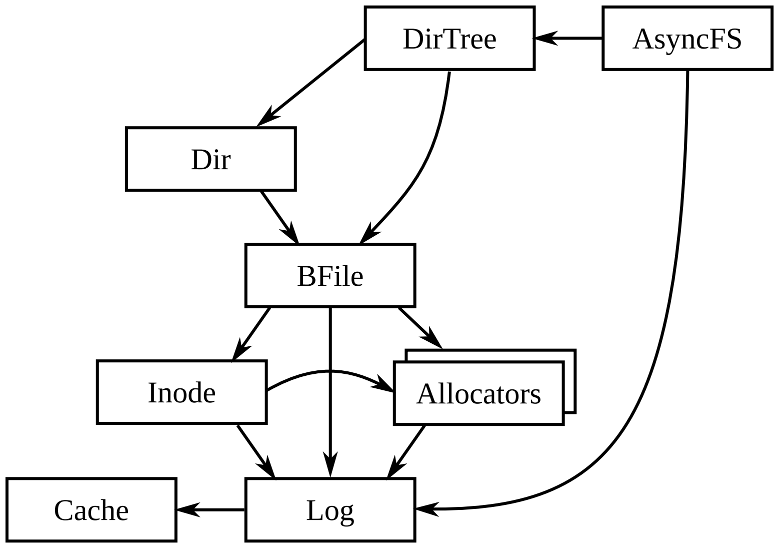


Figure 3-1: FSCQ is built from individual layers to reduce complexity.

picted visually in Figure 3-1.

3.1 Construction of FSCQ

At the lowest level of abstraction is FSCQ's block cache layer, which implements a bounded-size in-memory cache of disk blocks. While not necessary for correctness, this provides huge performance benefits by substituting most disk reads and writes with pure memory operations. Built on top of the cache is the log layer, which uses write-ahead logging to present a "reliable disk" abstraction. Where the physical disk is capable of reordering writes, the reliable disk wraps operations in transactions and ensures that each transaction fails or succeeds atomically. This is done using standard write-ahead logging techniques and a recovery procedure that restores the physical disk to the last logged state after a crash. By making transactions atomic, the log layer allows the layers above to mostly avoid reasoning about crashes, as all operations above the log are transactional.

The inode layer builds upon the reliable disk abstraction to transform the individual block-level view of the reliable disk into inodes. Each inode is individually addressable and contains both a set of attributes and a fixed-length list of block numbers. The block file layer uses these attributes and block number lists to implement "b-files". Each b-file stores a variable-length lists of blocks and some associated metadata. Building on the b-file abstraction, the directory layer represents directories as specialized b-files whose blocks are used to store variable-length lists of $(name, inode)$ pairs, which represent directory contents. The directory tree layer uses the directory representation to present a high-level view of the file system as a tree of named directories and files. At the top level, the AsyncFS layer wraps directory tree operations inside transactions to implement the final file system interface that is exported to callers. For convenience, FSCQ includes a generic allocator that is instantiated separately to track allocated inodes and blocks.

3.2 Separation logic

Since each level of abstraction introduces additional logical address spaces, it is necessary to have a convenient system for reasoning about their contents. In FSCQ, this system takes the form of predicates applied to the respective address spaces. By defining these predicates using separation logic [23], FSCQ's specifications make it easy to reason about changes to disjoint parts of the disk.

The most basic separation logic predicate is the points-to relation, written $a \mapsto v$; it specifies that in any address space to which it is applied, the address a maps uniquely to the value v. This allows predicates to impose conditions on address spaces, and to describe connections between them. The inode invariant, for example, is a specification over two address spaces: a concrete array of inode records, and a list of logical inodes. For a given inode number i, it states that if $i \mapsto I$ in the inode record list, then $i \mapsto INODE(I)$ in the logical inode space, where $INODE(I)$ represents the transformation from on-disk to abstract inode state. By maintaining this invariant, the inode layer ensures that $INODE(I)$ is a correct abstraction of I.

In addition to the points-to relation, separation logic provides a combining operator $*$; if the predicate $p*q$ holds over an address space, the space can be separated into two disjoint address spaces, one of which satisfies p and the other q. This allows invariants to concisely state facts about parts of address spaces and ignore the rest.

For example, the specification for a write of v to block a can specify that for all predicates F and values v_0, if $F * a \mapsto v_0$ holds on the disk before the write, then $F * a \mapsto v$ holds after. By using separation logic, this operation can effectively ignore all disk blocks except a, thus trivially guaranteeing that F is maintained on the rest of the disk.

3.3 Deferred durability

In order to provide strong guarantees about the durability of data on disk after a system call, a file system must ensure that the results of each operation are persisted to disk by waiting for all pending writes to complete before returning. To ensure that each system call is durable upon completion, the file system must flush all disk writes from the cache and executes a disk `SYNC` operation before returning. While this model provides strong guarantees about consistency, it significantly hampers performance. In a synchronous logging file system like the FSCQ prototype, each system call must write every block twice: first to the log, and then to the actual location on disk; and sequential operations that modify the same block still incur this overhead.

To provide improved performance, most modern file systems provide weaker guarantees about the durability of data, such that disk operations can be deferred until convenient. FSCQ implements deferred durability; where the synchronous prototype of FSCQ ensures that each operation is persisted immediately, FSCQ guarantees only that a prefix of the operations are persisted. This allows it to buffer, but not reorder, the results of system calls. FSCQ takes advantage of the weaker guarantees to coalesce multiple transactions in-memory before writing them to disk, and to collapse duplicate block updates between transactions. In addition, FSCQ avoids writing file data blocks to the log when possible, instead implementing log-bypass writes that update the actual file blocks directly. FSCQ implements the standard `sync()` system call to allow applications to force updates to individual files and directories to be persisted.

To facilitate reasoning about deferred durability, FSCQ's log layer exposes not just the current disk, but a sequence of disks. The latest disk represents the contents that would be returned by a read call, but not necessarily the data persisted on disk. Because the block cache can defer writes, and because the disk itself buffers and reorders writes, the actual on-disk data can differ significantly from the contents of the current logical disk. On a crash, the disk "chooses" the contents for each block such that for each block number, the on-disk contents are in the corresponding location of some disk in the disk sequence. This models the ability of the disk controller to flush blocks asynchronously; the selected contents for each block correspond exactly to the last contents that were flushed. The specification for the `sync()` operation simply guarantees that, upon completion, for each disk in the disk sequence, the value for flushed blocks is exactly the last value written to disk. This ensures that while the disk can "choose" any values from the disk sequence, the only possible value for flushed blocks is exactly the last flushed value.

3.4 Extraction

One of FSCQ's primary contributions is that, unlike previous work, it is a complete file system with both proven specifications and executable code. Because FSCQ's code and specifications are implemented in Gallina, Coq's code extraction facilities are used to transcribe the executable functions into Haskell. This extracted code is then compiled against the Haskell standard library with a small trusted interpreter into a single binary that, when executed, implements a FUSE server. By using the FUSE API in the Linux kernel, FSCQ allows applications to read and write files using the standard POSIX system call interface. VDFS retains FSCQ's extraction code almost unmodified, though it substitutes some native Haskell functions for their extracted counterparts when the unverified substitute provides substantial performance improvements. By combining the trusted interpreter and FUSE implementation with verified code, VDFS can provide a verified file system implementation to unmodified applications that use the standard system call interface.

Chapter 4

Design

The changes that VDFS introduces over FSCQ can be partitioned into two groups: support for large files, and in-memory caches. With regards to design, VDFS adds support for large files by generalizing the indirect block representation of FSCQ for arbitrary levels of indirection. Likewise, the in-memory caches, while diverse in their function, share similar structure that makes them easy to reason about.

4.1 Indirect blocks

The inode layer builds upon the log layer to transform the block-level view of the disk into inodes. Each inode is individually addressable and contains both a set of attributes and a variable-length list of block numbers. The block file layer uses inodes to implement b-files, storing metadata in the inode attributes and data blocks in the inode's list of blocks.

To ensure inode retrieval and update operations run quickly, the inodes are packed on disk as fixed-size entries in a single contiguous region. While this allows constant-time access to individual inodes, it complicates inode block storage, as each inode can only contain a fixed number of block addresses. One possible solution is to simply make each inode large enough to store the addresses of all of its blocks. While usable, this results in either enormous space inefficiency or an extremely small maximum file size. Instead, the scheme adopted by many file systems, and by FSCQ, is to

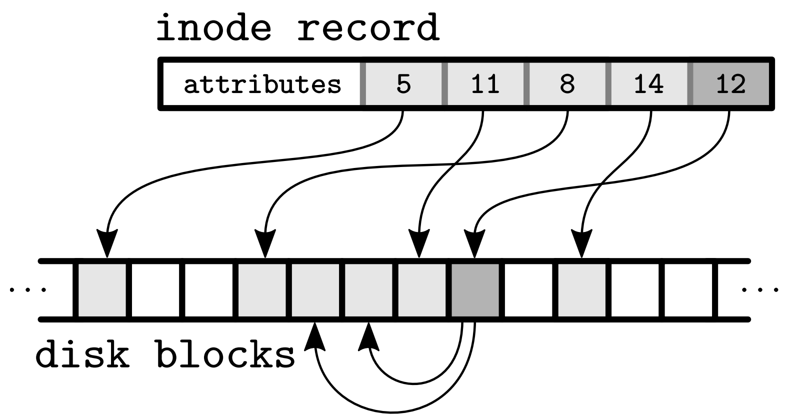


Figure 4-1: Each inode contains attributes, direct block references, and a single indirect block address.

store a number of block addresses in each inode, followed by an optional indirect block address. If an indirect block address is present, then the block it addresses is assumed to contain an array of block addresses that belong to the inode. The full list of addressed blocks for an inode can be computed by simply prepending the direct block addresses—those contained in the inode—to the list of addresses in the indirect block. This layout is depicted graphically in Figure 4-1. Each inode in FSCQ has space for 10 direct blocks and a single indirect block; with 4KByte blocks and 64-bit block addresses, this implies a maximum file size of $(10+512) \cdot 4\text{KByte} = 2.039\text{MByte}$.

Since one of its stated goals is to support much larger files than FSCQ, VDFS must use a different scheme for representing inodes on disk. One possible solution is simply to make all direct blocks into indirect blocks, so that each inode is capable of storing $(11 \cdot 512) \cdot 4\text{KB} = 22.5\text{MB}$. While this is an incremental improvement on FSCQ, it is still unrealistic, and introduces additional overhead for small files as well. Instead, VDFS adds additional levels of indirection to each inode, allowing it to store files several orders of magnitude larger than FSCQ. While the use of doubly- and triply-indirect blocks is common practice, VDFS takes advantage of the common structure and generalizes the approach to n levels of indirection. Though VDFS' implementation uses only three levels of indirection, this generalization allows for a simple recursive implementation of all operations and correspondingly straightforward proofs of correctness.

4.1.1 Representation

VDFS' representation for indirect blocks connects a block number b and the list l of data block numbers referenced through b:

- if the block with address b has indirection level 0, then its contents are exactly the serialized form of l;
- if the block with address b has indirection level $(n+1)$, then it contains some list of blocks $[b'_0, b'_1, ..., b'_{N-1}]$, where N is the number of addresses that can be stored in a block, each b'_i is the address of a block with indirection level n that references a list of data blocks l'_i, and the top-level l is exactly the in-order concatenation of the l'_is.

Then b can be treated as the root of a tree of indirect blocks, where l corresponds logically to the collective contents of its leaves as depicted in Figure 4-2. Furthermore, since every indirect block contains a full list of children, every indirect tree is complete. This requires that the number of blocks referenced by a tree with indirection level n is $\sum_{i=0}^{n} N^{i+1}$, where N is the number of addresses that can be stored in a block.

Requiring that every block number be valid and unique would waste space as every file, no matter how small its actual contents, would use the same number of blocks on disk. Instead, VDFS representation invariant, depicted in Figure 4-3, has an additional rule:

- if any block number b is zero, the actual contents of the block at address 0 are ignored and are substituted with a list of 0's as the children of b.

As a direct corollary, if l is the list of blocks with block number 0 at the root, then the elements of l are 0. Since VDFS' block allocator never allocates the 0 block, this maintains the invariant that every indirect block tree is logically complete without wasting disk space on unused blocks. As Figure 4-3 shows, this representation invariant is defined inductively over the level of indirection.

The representation invariant in Figure 4-3 has a number of convenient properties. First, it is recursive, such that a block with root b and indirection level n is the root

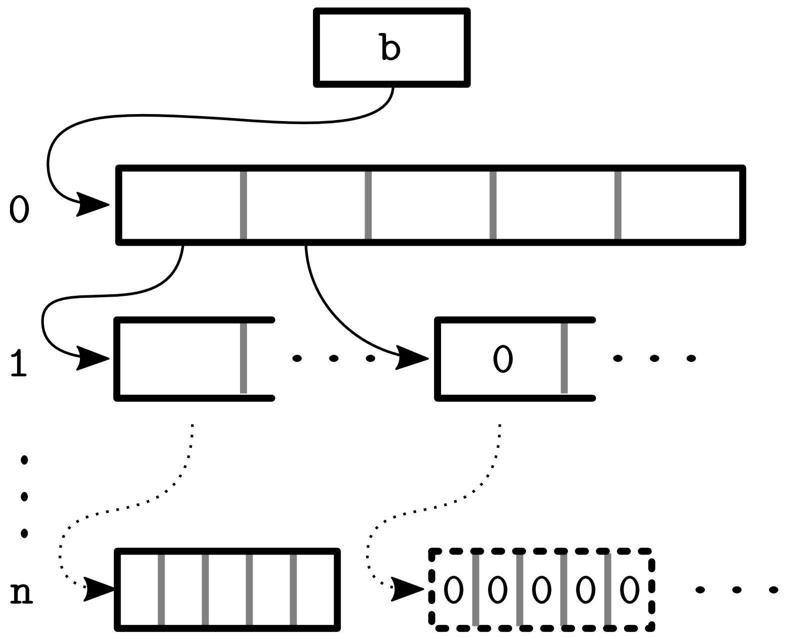


Figure 4-2: A depiction of the logical contents of an indirect block tree, with 5 addresses per block. Dotted blocks are purely logical and are not stored on disk.

of its own tree, regardless of whether the address b itself was obtained from an inode or from the contents of an $(n+1)$-indirect block. This allows programs and proofs to reason about indirect block trees while remaining oblivious to the contents of the rest of the disk, which is exactly the case that separation logic covers. In addition, it does not require blocks to store any information about their level of indirection. Instead, the level of indirection is prescribed externally, either by a root for its children, or by some higher-level representation invariant. This allows proofs to treat every block with a non-zero indirection level identically, which makes recursion easy. Finally, this representation allows for efficient space usage. Since 0-rooted trees take up no space on disk, and every block address is required to be either 0 or valid, then for any indirection level, the number of valid blocks required to store a tree on-disk is logarithmic in the number of non-zero lowest-level block numbers.

```
Definition indrep_n_helper b iblocks :=
  if (b =? 0) then
    [[ iblocks = repeat $0 NIndirect ]]
  else
    b |-> serialize (iblocks).

(* indlvl = 0 if ibn is the address of an indirect block,
   indlvl = 1 for doubly indirect, etc. *)

Fixpoint indrep_n_tree indlvl b l :=
  match indlvl with
  | 0 => indrep_n_helper b l
  | S indlvl' =>
    exists iblocks l_part,
      indrep_n_helper b iblocks * [[ l = concat l_part ]] *
      listmatch (fun b' l' => indrep_n_tree indlvl' b' l') iblocks l_part
  end.
```

Figure 4-3: The recursive indirect block representation invariant specifies how a tree rooted at b is connected to its blocks l.

4.2 In-memory caching

In order to implement caching without compromising correctness, each cache must have its own representation invariant over the contents of the cache and the logical representation of the cached data that must be maintained throughout execution. To facilitate this, each cache is implemented within an additional layer of VDFS that provides the same guarantees as the layer below while fulfilling requests from the cache where possible. This allows the cache invariant to reason about the logical abstraction exported by the layer below, instead of the on-disk state, and to remain compatible with the layer above by exporting the same functions and specifications.

While VDFS' caches support different parts of the file system, they share three key properties. The first is that each cache, like the layer it wraps, only reasons about the state of the current transaction. Since the representation for the layers above the log are stated relative to state of the current transaction, this allows the cache invariant to reference only a single disk state. While this makes the invariant succinct, it introduces complications for aborted transactions, which will be discussed in chapter 5.

The second property is that for every element in the cache, the cached data represents a function of the logical state of the layer below. This allows caches to maintain correctness relative to the abstract state of the wrapped layer instead of reasoning about the actual on-disk state. The third and final property is that each cache is only guaranteed to be partially correct, such that any look-up in a valid cache is allowed to fail. This has several implications that simplify proofs and implementation which will be discussed further below.

4.2.1 Partial correctness

The partial correctness property of each cache has several implications that aid in proofs. The first is that any element in the cache can be invalidated at any time without affecting the correctness of the cache. While cache limits are not implemented in VDFS, this ensures that implementation code is always allowed to evict elements at-will in order to maintain strict bounds on the cache size.

Starting with any valid valid cache and removing all of its elements results in the empty cache; thus, as a corollary, the empty cache is always valid. Alternatively, since no look-up an empty cache will succeed, and all cache look-ups are allowed to fail, an empty cache satisfies the partial correctness property. This allows the various caching layers to initialize the in-memory state with empty caches, and to fill them opportunistically. In addition, it allows the current cache to be dropped at-will and replaced with an empty cache. While wholesale cache invalidation is generally undesirable, it allows VDFS to restore the caches to a valid state after a crash without consulting the on-disk contents.

Each cache guarantees that if a look-up succeeds, the retrieved contents are exactly some transformation of the logical state of the wrapped layer. This can be trivially maintained by ensuring that every write to the on-disk state also updates the cache, thus implementing the cache with a write-through strategy. For some operations, however, the transformation maintained by the cache is expensive to update. Since elements in the cache can be dropped without affecting correctness, this expensive update can be avoided by simply invaliding the respective cache entry. This allows each cache to be updated on a best-effort basis, where every disk write corresponds to either a cache write or invalidation.

Chapter 5

Implementation

The implementation of VDFS is based on the FSCQ file system with some key modifications. The code and proofs for the logging, block cache, and top-level file system interface is retained almost unmodified, as are the directory and directory tree layers, with the insertion of caching layers in between. The block and inode bitmap allocators are extended with in-memory caches, as is the inode layer. Finally, the inode implementation is extended with support for doubly- and triply-indirect blocks, and the block file data synchronization operation is completely re-implemented.

5.1 Indirect blocks

VDFS' n-indirect blocks are implemented as a representation invariant over indirect trees, and several functions that manipulate indirect block trees. Like the representation invariant, these functions are written recursively to match the indirect block tree structure. The `indget` function, depicted in Figure 5-1, exemplifies this structure. Given an address b and an offset, it returns the block number at the given offset in the leaves of the tree rooted at b. When the indirection level is 0, it simply indexes into the contents of block b and returns the value there. When the indirection level is nonzero, it indexes into the top level block, retrieving the root of a smaller indirect block tree, and recurses. Though VDFS ensures that calls to `indget` never traverse 0 blocks, this is only guaranteed by invariants above the indirect block abstraction.

```
Fixpoint indget indlvl b off :=
  If (b =? 0) {
    Ret 0
  } else {
    let divisor := NIndirect ^ indlvl in
    b' <- IndRec.get b (off / divisor);
    match indlvl with
    | 0 => Ret b'
    | S indlvl' => indget indlvl' b' (off mod divisor)
    end
  }.
```

Figure 5-1: The `indget` function retrieves $l[off]$, where b is the root of an indirect tree with leaves l.

Thus, to maintain correctness in all cases, `indget` also handles the case of a 0 root; since the leaves of its tree are identically 0, the value at any index is also 0.

The specification for `indget` is equally simple, and simply declares that if b is the root of an indirect block tree with leaves l, then for all values off such that $off < \text{length}(l)$, the return value is $l[off]$. Since both the representation invariant and implementation are defined as recursive functions over decreasing indirection levels, the proof for this specification is naturally recursive. Indeed, the proof is simple; most of the effort is proving equivalence between nested and flat list representations for l.

While not all functions are as simple as `indget`, they and their corresponding proofs follow a similar recursive structure. The proofs of correctness are more involved, though the large part of the proof work is showing equivalence of list operations. These list operation proofs are then used to demonstrate that modifications to inner blocks in an indirect block tree produce the correct lowest-level block list.

The most complex operation implemented over indirect block trees is clearing ranges of blocks. Given a root b of an indirect block tree and a range $[r, r + len)$, the `indclear` function ensures that, upon completion, the leaves of b's tree are the same as before except that every block in the given range is now 0. While this could be implemented simply by setting the contents of bottom-level blocks to 0 in appropriate ranges, such a naïve approach wastes space unnecessarily.

For example, consider the case where the given range covers the entire indirect block tree, so that afterwards the tree rooted at b has leaves that are identically 0.

```
Definition get (ir : irec) off :=
  If (off < NDirect) {
    Ret (IRBlocks ir) [off]           (* select from indirect blocks *)
  } else {
    let off := off - NDirect in       (* offset relative to indirect
                                         blocks                     *)
    If (off < NIndirect) {
      indget 0 (IRIndPtr ir) off      (* retrieve through indirect
                                         block                      *)
    } else {
      let off := off - NIndirect in
      If (off < (NIndirect ^ 2)) {
        indget 1 (IRDindPtr ir) off   (* retrieve through doubly
                                         indirect block             *)
      } else {
        let off := off - NIndirect ^ 2 in
        indget 2 (IRTindPtr ir) off   (* retrieve through triply-
                                         indirect block             *)
      }
    }
  }.
```

Figure 5-2: The inode `get` function either indexes into the direct blocks or retrieves from an indirect block, depending on the offset.

Since this has the same contents as the tree rooted at 0, all internal indirect blocks of the tree rooted at b are redundant, and so consume unnecessary space. More generally, since the representation invariant guarantees that any 0 entry in a tree has leaves that are identically 0, any block with identically 0 contents can be replaced in its parent by a 0 entry. Thus, the efficient implementation of `indclear` used in VDFS both clears entries in blocks and opportunistically replaces them with 0 references in their parent indirect block when possible.

While these operations over indirect block trees are written and specified for all levels of indirection, they are used to implement singly-, doubly, and triply-indirect blocks in VDFS. The `get` function that the inode layer exports is implemented using the `indget` function with constant indirection levels as depicted in Figure 5-2. Likewise, the inode `shrink` function uses `indclear` to free blocks whose contents are no longer necessary. Since the proofs for the indirect block manipulation functions are valid for all levels of indirection, they can be used in proofs of specifications for `get`, `shrink`, and the other functions exported by the inode layer.

5.1.1 Syncing large files

Since VDFS supports the `sync()` system call, it must guarantee that, when called with a file as an argument, all data blocks for the file are persisted to disk upon returning. FSCQ implements this by iterating over each block in file and flushing it from the cache, then executing a `SYNC` disk operation. Since the sync operation doesn't complete until all pending writes are persisted, and since all blocks in the file were written, the entire contents of the file are persisted to disk after the sync. While this is easy to prove correct, it has a running time that is linear in the length of the file. If the maximum file size is small, as it is in FSCQ, the total execution time has a small upper bound so the linearity is acceptable.

For larger files, however, this scheme presents a significant problem, especially if only a few of the blocks have been modified since the last disk sync. Enumerating all the blocks of a large file requires retrieving the contents of every indirect block either from the in-memory cache or directly from disk. This wastes space in the cache and requires additional unnecessary disk operations. VDFS solves this by keeping track of a set of dirty blocks per file, and only flushing the appropriate set from the cache on a `sync()`. This makes the running time linear in the number of blocks dirtied since the last sync instead of in the length of the file.

While this optimization is intuitively correct, the disk sequence abstraction in FSCQ does not provide enough information to prove it so. As part of its transactional interface, the log layer exposes the sequence of disks in the current transaction but does not expose information about how the disks are related. This makes it easy for higher layers to reason about the state of the current transaction, but not about the state of the physical disk or of previous disks in the disk sequence. Since the `sync()` system call must ensure that *all* blocks in a file are synced on disk, information about the state of each block must be maintained for both the current and all previous disks in the disk sequence.

To supply this information, VDFS' log layer exposes a new abstract memory state that indicates whether a block's contents are persistent in all previous disk states. The invariant maintained by the log layer is that, for a disk sequence ds and associated synced map sm, $\forall a$, if $a \mapsto \texttt{true}$ in sm, then $\forall d \in ds, a \mapsto v$ in d implies that v is synced. Similar to the caches, this invariant is one-way: addresses that map to `false` in a synced map don't guarantee anything about the corresponding values in the disk sequence. Since logged writes only affect the latest disk in the disk sequence, and since they always write synced blocks, logged writes preserve the synced map. The only operations that alter the synced map are a direct write to an address, which changes $a \mapsto b$ to $a \mapsto \texttt{false}$ for any b, and an explicit sync of an address, which changes $a \mapsto b$ to $a \mapsto \texttt{true}$.

Since all writes to non-data blocks in VDFS are logged writes, the only operations that modify the synced map are those in the block file layer. By using this "synced map", the block file layer can state and maintain the invariant that for every block in every file, the block number is either in the relevant list of dirty blocks or is synced in all previous disks. Since syncing an already-synced block is a no-op, the block file layer can use this invariant to prove that syncing only the dirty blocks for a file equivalent to syncing all the blocks.

5.2 In-memory caches

The implementation of FSCQ includes a generic serialization module that is used to store "records" on disk. The record array module builds on this to implement serialization of sequences of arbitrary homogeneous objects, and is used in both the inode and directory layer to store and load inodes and file entries to and from the disk, and in the block and inode allocators to store individual bits. Because FSCQ's executable code is compiled Haskell extracted from the verified Gallina source, serialization and de-serialization are much slower than if they were implemented directly in C. VDFS provides improved performance over FSCQ while retaining the serialization code by caching de-serialized records which can be used to satisfy read requests.

5.2.1 Inode cache

The inode cache is implemented as part of the inode layer, and stores de-serialized inode contents in memory. Since inode contents are used to service every file operation (block numbers for reads and writes, and attributes for metadata operations), this provides a measurable performance increase over repeated serialization and de-serialization. The cache is implemented as a key value store that maps each inode number to either the de-serialized inode contents, or the special value "none". The correctness invariant for the cache is stated such that for any inode number that does not map to "none", the value in the cache is exactly the de-serialized version of the inode on the current disk in the disk sequence. This corresponds to the partial correctness guarantee mentioned previously, where "none" entries are invalid. Since any contents retrieved by a look-up are guaranteed to be valid, the cache can be used to fulfill most read operations without any disk operations or de-serialization.

5.2.2 Directory entry cache

FSCQ's directory layer is built on top of the block file layer and implements directories as lists of $(name, inum)$ pairs, where $name$ is a fixed-length string of bits and $inum$ is the corresponding inode number. Since the list of entries is unsorted, directory look-ups are implemented as linear searches through the list. VDFS introduces a new directory caching layer that stores name-to-inode mappings for individual directories in memory. Like the inode cache, it is implemented as a map from directory inode numbers to optional values.

The invariant requires that for each directory inode number in the cache, the corresponding value is a map from names to inode numbers that exactly corresponds to the on-disk contents of the directory. Thus while no directory is required to be in the cache, every cached map of directory contents must contain exactly the same contents as the directory's logical state. This ensures that if some directory contents d are in the cache, the directory caching layer can provide a correct response for a look-up of any name n: if n is in d, n is in the directory on disk; if n is not in d,

n must not be in the on-disk directory. To maintain the invariant, when a directory is loaded into the cache, it must be loaded in its entirety. While this requires that VDFS traverses the entire directory list on the first look-up, the cost is amortized over all later look-ups.

In addition to directory contents, the cache also contains an unverified “hint” field that is used when adding a file to the directory. The hint is unverified in that it must be checked instead of assumed correct, and so requires no additional invariants or proofs. The hint is set for a directory cache entry by the last successful `unlink` operation, and its value is the index of the removed file entry. It is then checked by the next `link` operation when attempting to add a file to the directory. If the hinted directory entry index does not contain a file entry, the new file entry is added there; otherwise the `link` call falls back to a linear search through the directory to find an empty entry.

5.2.3 Caching bitmap allocator

While FSCQ uses separate bitmap allocators for tracking both inodes and data blocks on disk, both are specialized versions of the same generic bitmap allocator. VDFS' caching bitmap allocator exposes the same functions and specifications, and is used to implement both the inode and block allocator. Unlike the inode and directory caches, which service look-ups and updates for a single key, VDFS' bitmap allocator is allowed to return any valid element for a response. This difference of function requires that the bitmap allocator use a different implementation than the key-value map used in the inode and directory entry caches.

Instead, the bitmap allocator maintains an optional list, which has value either “none” or “some f”, where f is a list containing free blocks. The bitmap allocator implements partial correctness by requiring in its invariant that if a list is present, the list must contain exactly the blocks marked free in the bitmap. By maintaining this complete list, the bitmap allocator is able to check in constant time whether an available block exists. If so, the block is marked as allocated in the bitmap, removed from the cache, and returned to the caller. If the list is empty, the invariant implies that no blocks are available, which allows the allocator to return a response without consulting any on-disk state.

While the non-caching bitmap allocator never allocates the 0 block, the block is never marked as allocated on disk; the guarantee is ensured by special-casing in the `allocate` function. Thus the bit for the 0 block always indicates “free”, and it remains in the logical list of free blocks exposed by the non-caching block allocator. To emulate this behavior, the caching allocator also does not allocate the 0 block, though it does so by ensuring that it is never in the list of free blocks. To maintain the property that the cache invariant is a function of the logical state, the invariant requires that the cached list, if present, is a permutation of the result of removing the 0 block from the abstract free list. This ensures that any block returned by the caching allocator is valid without requiring special-casing in the cache-aware `allocate` function. Instead, this special casing is implemented as a linear list traversal that occurs when the cache is first filled from disk. While not free, the cost is amortized over all later allocations and frees.

5.2.4 Cache rollback

All of these cache invariants require only that the cache contents be correct relative to the latest disk in the disk sequence. Since every logged write produces a new disk in the disk sequence, the caches’ correctness is maintained by ensuring that the contents are updated or invalidated for each write. Each one of these logged writes is part of a transaction that is started and ended by the top-level AsyncFS layer. When all operations within the transaction succeed, the transaction is committed and the last disk in the disk sequence becomes durable. If an operation executed during a transaction fails (because a directory is out of space or nonexistent, the disk is full, etc.), the transaction is aborted and the disk sequence is reverted to the last disk before the start of the transaction.

Though this ensures that file system operations are atomic, it breaks the representation invariant for the caches, as they could have been modified by operations preceding the one that failed. To restore consistency between the caches and on-disk state after a transaction abort, VDFS rolls back all changes in the cache since the last commit.

Every AsyncFS operation that is not strictly read-only (since read-only transactions never fail) simply saves a copy of each cache at the start of each transaction. On commit, the saved copies are ignored and the updated cached copies are retained; on an abort, the updated caches are dropped and the saved copies are substituted in their place. Because VDFS' code is purely functional, the caches are not mutable, so the saved copies are guaranteed to remain unchanged. Since an abort restores the pre-transaction disk and the pre-transaction caches, the cache invariants trivially hold for both post-commit and post-abort states, thus maintaining correctness.

Chapter 6

Evaluation

The evaluation of VDFS seeks to answer the following questions:

- Does VDFS handle more workloads than FSCQ?
- Does VDFS improve over current verified file systems?
- Does VDFS achieve comparable performance to ext4?
- How much improvement do each of VDFS' optimizations provide?

6.1 Method

To measure performance of VDFS, two microbenchmarks and three application workloads are used. For context, the benchmarks are run on VDFS, FSCQ [4], and Yxv6 [5], which represent recent state of the art in machine-checkable verified file systems, and on ext4, which is a highly-optimized but unverified Linux file system. The Yxv6 file system is run in two modes: the verified synchronous mode where all system calls immediately persist their changes, and the asynchronous mode where system calls are deferred in memory, called `group_commit` by Yggdrasil, which is denoted Yxv6*. This second mode, however, does not have a top-level file-system specification that describes how changes are deferred [24, 25]; as a result, it provides no meaningful proof of crash safety. ext4 is also run in two modes: one with `data=ordered` (denoted here as ext4) and the other with `data=journal,journal_async_commit` mount options (denoted ext4/J). Because using both can lead to consistency errors on a crash [26], ext4 prohibits the use of `journal_async_commit` in `data=ordered` mode.

All experiments were run on a Dell PowerEdge R430 server with two Intel Xeon E5-2660v3 CPUs and 64GB of RAM. To ensure that VDFS is applicable across multiple disk configurations, the benchmarks were run on a diverse set of drives. One is a 7200rpm WDC WD2003FYYS rotational disk, which is denoted as HDD. Two SSDs are used both because of demonstrated performance differences and because it is not clear which, if either of them, provide strong crash safety guarantees [27]. One is an inexpensive Samsung 850 SSD, which is denoted SSD1, and the other is an expensive high-performance Intel S3700 SSD, denoted SSD2. Finally, to simulate a "maximally performant" disk, a virtual RAM drive, denoted RAM, is used as well.

6.2 Microbenchmarks

The microbenchmarks are intended to measure performance of deferred writes for small file operations and large file writes, inspired by LFS [28]. The `smallfile` benchmark creates 1,000 files; for each, it creates the file, writes 100 bytes to it, and `fsync`s it; throughput is measured as the number of files created per second. The `largefile` benchmark overwrites a 1 GByte file, calling `fsync` every 10 MBytes; throughput is measured as the average disk write speed, in MB/s.

The results of these benchmarks are shown in Figure 6-1, and lead to several conclusions. First, VDFS achieves good performance, significantly improving on all prior verified file systems, due to its I/O and CPU optimizations. VDFS is also more complete: no prior verified file system is even capable of running the large-file benchmark because they lack doubly indirect blocks, and Yxv6 is not capable of supporting more than 256 files in a directory.

Second, VDFS' performance is close to that of ext4 for `smallfile` on HDD, and even beats ext4 on SSD1. This is because VDFS is as efficient as ext4 in terms of disk barriers, but ext4 writes out one additional block to its journal (to initially zero out the new file), which VDFS combines with the subsequent data write. VDFS also achieves performance close to that of ext4 for `largefile` on HDD and SSD1. However, on SSD2 and RAM, VDFS's performance lags behind that of ext4 due to the CPU overhead of Haskell.

6.3 Applications

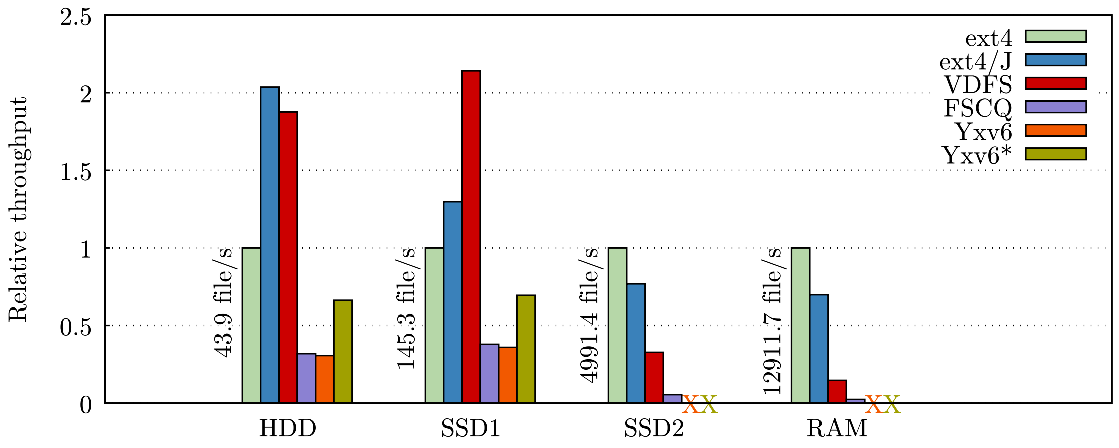
Figure 6-2 shows the performance for three applications. mailbench is a qmail-like mail server [29], which has been modified to call `fsync` and `fdatasync` to ensure messages are stored durably in the spool and in the user mailbox. "dev. mix" is measuring the result of running `git clone` on the xv6 source code repository [30] followed by running `make` on it. The TPCC benchmark measures the performance of executing a TPC-C-like [31] workload against a SQLite database.

The results reinforce the conclusions drawn from the microbenchmarks. VDFS significantly outperforms other verified file systems, and is able to run applications that others cannot. VDFS' performance on HDD and SSD1 is comparable to ext4, but VDFS' Haskell overhead becomes much more significant with SSD2 and RAM in particular.

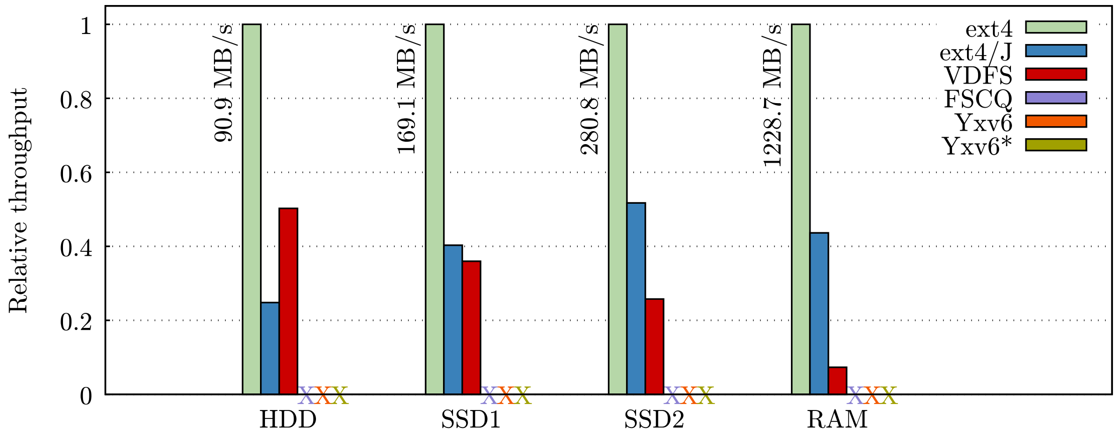
6.4 Impact of optimizations

To examine how much impact each of this thesis' optimizations had on performance, the same application and microbenchmark suite were run with each of the optimizations turned off. The results of the the microbenchmark comparisons are presented in Figure 6-3, and for the application benchmarks in Figure 6-4. The optimizations implemented in VDFS are grouped into three sets:

- the per-file dirty block list optimization,
- the free items caches in the block and inode allocators,



(a) Smallfile



(b) Largefile

Figure 6-1: Performance for Linux ext4, VDFS, FSCQ, Yggdrasil for microbenchmarks. Both measure throughput; higher is better. Benchmarks that didn't complete are marked with "X."

- and the look-up caches used for directory and inode contents.

The results presented here are for the full VDFS system, and for the same system but with each set of optimizations disabled individually. The label "VDFS " is used for results with all optimizations enabled, "VDFS-D" for VDFS with the dirty block lists disabled, "VDFS-A" for VDFS with the allocator caches disabled, and "VDFS-L" for VDFS with the look-up caches disabled.

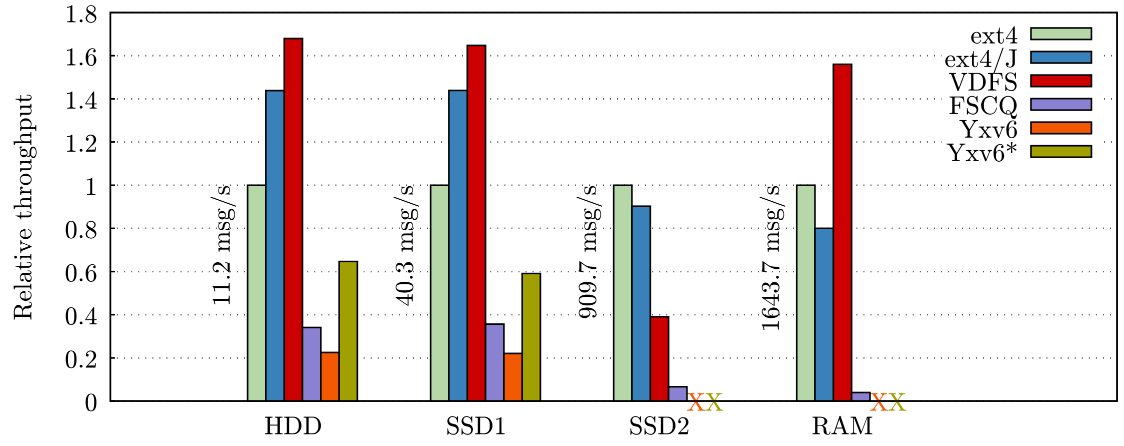
The per-file dirty list significantly increases the performance of the largefile and mailbench benchmarks, which both write small sections of large files and then `sync()`

them to disk. Since the list of dirty blocks is kept directly in memory, VDFS avoids the need to iterate over entire file block lists.

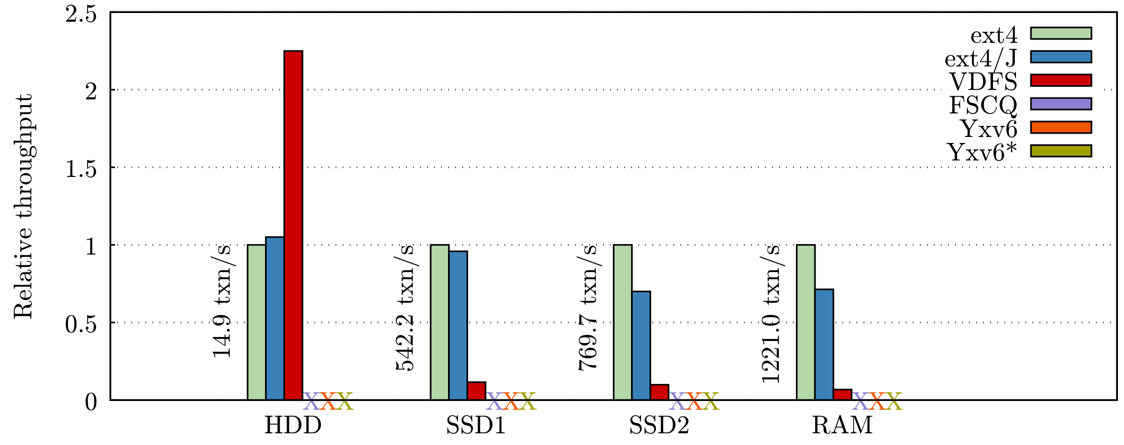
Likewise, the cached lists of free elements for the block and inode allocators significantly increase performance on workloads that allocate many inodes or file data blocks. This includes all of the benchmarks except largefile, which simply overwrites data in-place and so performs no allocations at all.

Lastly, the effects of the look-up caches can be seen in the mailbench benchmark, which writes files then looks them up by name, and the smallfile benchmark, which uses the directory entry index hint to avoid searching for an available directory entry.

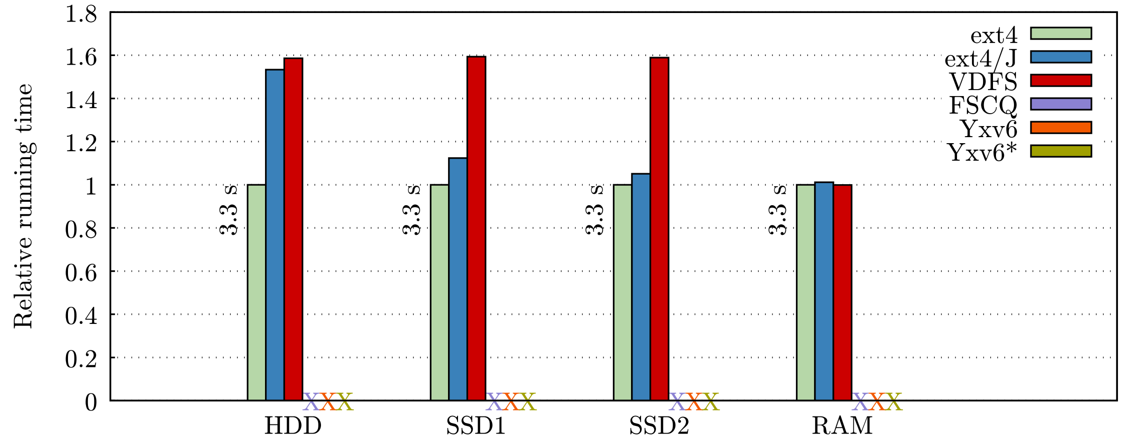
Across all the benchmarks, the difference in execution time between VDFS and VDFS-D, VDFS-A, and VDFS-L demonstrates the impact of the particular optimizations. It is worth noting that the relative impact of the optimizations on running time and throughput is correlated with the speed of the block device upon which the tests are run. Since faster block devices take less time to perform I/O, more time is spent executing file system code on the CPU. As expected, the trend of larger performance differences for faster I/O shows that these optimizations significantly reduce VDFS' CPU utilization.



(a) Mailbench

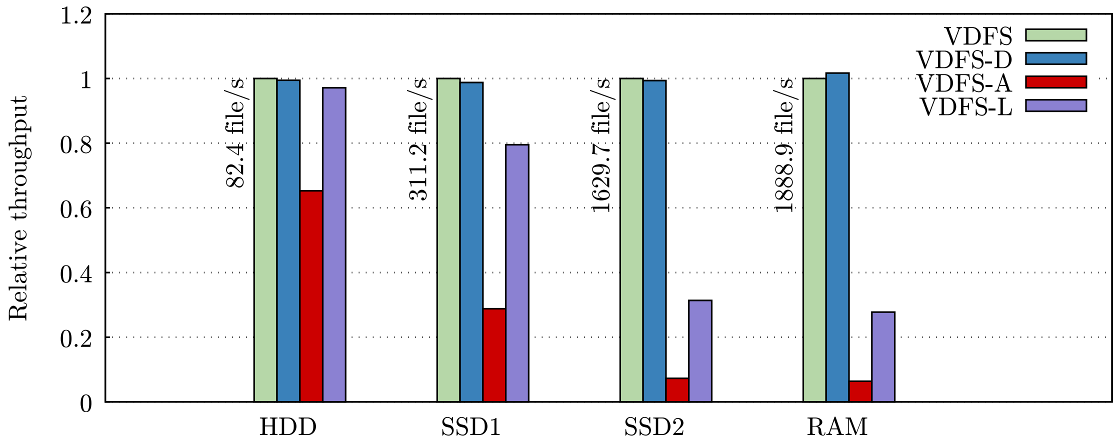


(b) TPCC

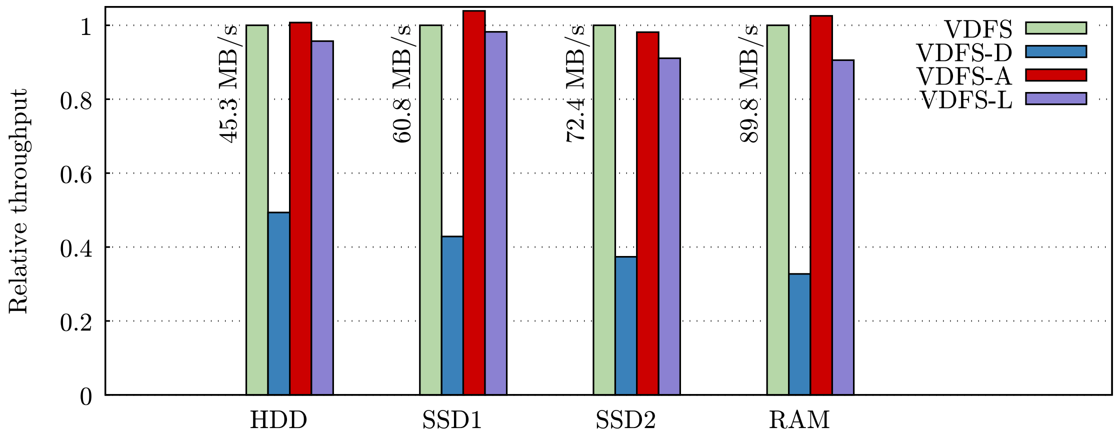


(c) Dev. mix

Figure 6-2: Performance for Linux ext4, VDFS, FSCQ, Yggdrasil for application workloads. mailbench and TPCC measure throughput; higher is better. Dev. mix measures run time; lower is better. Benchmarks that didn't complete are marked with "X."

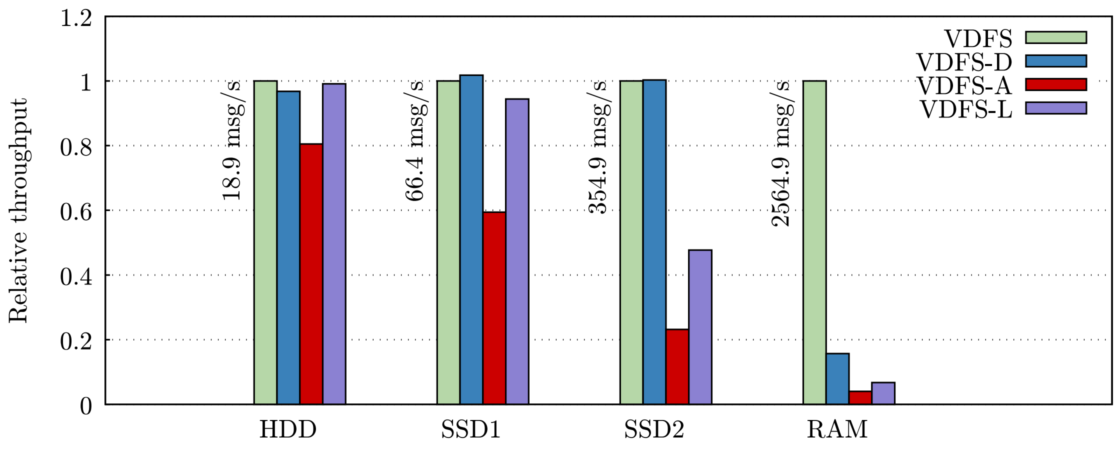


(a) Smallfile

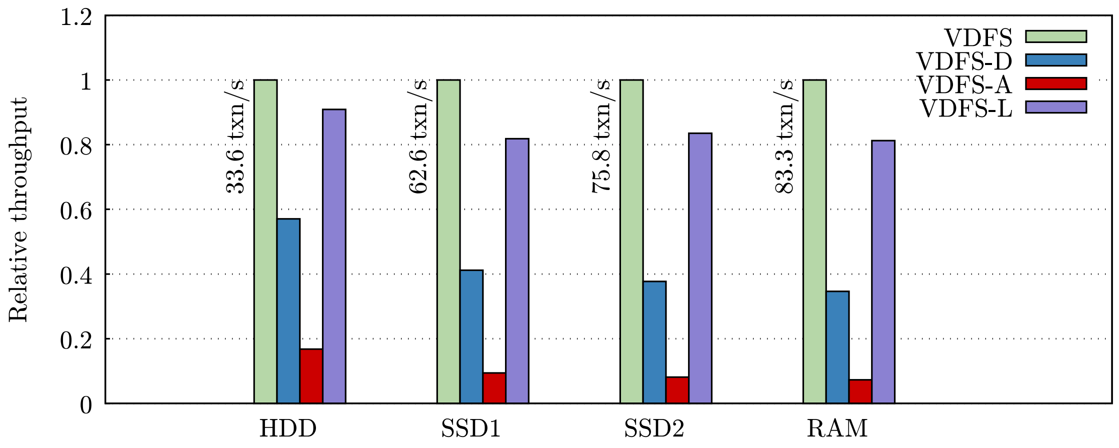


(b) Largefile

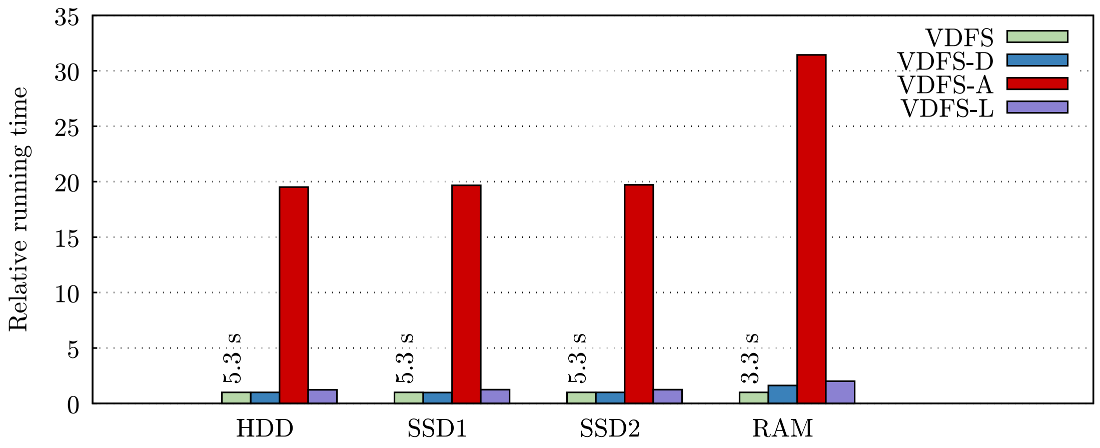
Figure 6-3: Microbenchmark performance for VDFS with optimizations selectively disabled. Both measure throughput; higher is better.



(a) Mailbench



(b) TPCC



(c) Dev. mix

Figure 6-4: Application workload performance for VDFS with optimizations selectively disabled. mailbench and TPCC measure throughput; higher is better. Dev. mix measures run time; lower is better.

Chapter 7

Conclusion

As in many systems, there is a tension when building a verified system between designing for performance and for ease of verification. FSCQ is a verified file system that takes the latter route—it is written in a functional style that is amenable to automated reasoning and proofs. While convenient for verification, this causes FSCQ to exhibit poor performance that prevents its use in real-world applications.

This thesis presents the first high-performance verified deferred-durability file system. By building on FSCQ, VDFS demonstrates that the use of a high-level language for building verified programs allows for a high-performance implementation while maintaining correctness guarantees.

The thesis first describes an indirect block representation and the design of in-memory caches that are amenable to both verification and high performance. It then describes how these optimizations were applied to FSCQ to produce VDFS, a verified file system that implements deferred-durability guarantees. Lastly, this thesis presents an evaluation of VDFS against FSCQ and other verified and non-verified file systems that demonstrates its performance on both microbenchmarks and realistic application workloads.

While VDFS provides improved performance over state-of-the-art verified file systems, its execution is still CPU-bound on sufficiently fast disks. This is due in large part to its use of Haskell as an intermediate extraction language. While convenient, this extraction to Haskell produces inefficiencies that unverified file systems avoid by using C or other low-level programming languages.

Though it demonstrates high performance for serial workloads, VDFS is unable to make use of multiple cores, which are common in modern machines. While unverified scalable file systems are used in practice, defining the specification for a concurrent general-purpose file system remains an open problem. Though significant progress has been made in certifying concurrent systems [1, 32], the development of techniques for verifying general concurrent systems also remains open.

Despite these unsolved problems, recent work on verified systems has made real strides towards matching the performance of unverified counterparts. This thesis alone demonstrates that high performance can be achieved for a verified file system without losing correctness. Hopefully the design and implementation of its optimizations will be useful for future verified systems work.

Bibliography

[1] R. Gu, Z. Shao, H. Chen, X. N. Wu, J. Kim, V. Sjöberg, and D. Costanzo, "CertiKOS: An extensible architecture for building certified concurrent OS kernels," in *Proceedings of the 12th Symposium on Operating Systems Design and Implementation (OSDI)*, (Savannah, GA), pp. 653–669, Nov. 2016.

[2] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, M. Norrish, R. Kolanski, T. Sewell, H. Tuch, and S. Winwood, "seL4: Formal verification of an OS kernel," in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, (Big Sky, MT), pp. 207–220, Oct. 2009.

[3] X. Leroy, "Formal verification of a realistic compiler," *Communications of the ACM*, vol. 52, pp. 107–115, July 2009.

[4] 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), pp. 18–37, Oct. 2015.

[5] H. Sigurbjarnarson, J. Bornholt, E. Torlak, and X. Wang, "Push-button verification of file systems via crash refinement," in *Proceedings of the 12th Symposium on Operating Systems Design and Implementation (OSDI)*, (Savannah, GA), pp. 1–16, Nov. 2016.

[6] H. Chen, *Certifying a Crash-safe File System.* PhD thesis, Massachusetts Institute of Technology, Sept. 2016.

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

[8] IEEE (The Institute of Electrical and Electronics Engineers) and The Open Group, "The Open Group base specifications issue 7, 2013 edition (POSIX.1-2008/Cor 1-2013)," Apr. 2013.

[9] S. C. Tweedie, "Journaling the Linux ext2fs filesystem," in *Proceedings of the 4th Annual LinuxExpo*, (Durham, NC), May 1998.

[10] 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)*, (Atlanta, GA), pp. 175–188, Apr. 2016.

[11] D. M. Ritchie and K. Thompson, "The unix time-sharing system," *Commun. ACM*, vol. 17, pp. 365–375, July 1974.

[12] S. Amani and T. Murray, "Specifying a realistic file system," in *Proceedings of the Workshop on Models for Formal Analysis of Real Systems*, (Suva, Fiji), pp. 1–9, Nov. 2015.

[13] K. Arkoudas, K. Zee, V. Kuncak, and M. Rinard, "Verifying a file system implementation," in *Proceedings of the 6th International Conference on Formal Engineering Methods*, (Seattle, WA), Nov. 2004.

[14] W. R. Bevier and R. M. Cohen, "An executable model of the Synergy file system," Tech. Rep. 121, Computational Logic, Inc., Oct. 1996.

[15] W. R. Bevier, R. M. Cohen, and J. Turner, "A specification for the Synergy file system," Tech. Rep. 120, Computational Logic, Inc., Sept. 1995.

[16] G. Ernst, G. Schellhorn, D. Haneberg, J. Pfähler, and W. Reif, “Verification of a virtual filesystem switch,” in *Proceedings of the 5th Working Conference on Verified Software: Theories, Tools and Experiments*, (Menlo Park, CA), May 2013.

[17] M. A. Ferreira and J. N. Oliveira, “An integrated formal methods tool-chain and its application to verifying a file system model,” in *Proceedings of the 12th Brazilian Symposium on Formal Methods*, Aug. 2009.

[18] L. Freitas, J. Woodcock, and A. Butterfield, “POSIX and the verification grand challenge: A roadmap,” in *Proceedings of 13th IEEE International Conference on Engineering of Complex Computer Systems*, pp. 153–162, Mar.–Apr. 2008.

[19] P. Gardner, G. Ntzik, and A. Wright, “Local reasoning for the POSIX file system,” in *Proceedings of the 23rd European Symposium on Programming*, (Grenoble, France), pp. 169–188, 2014.

[20] W. H. Hesselink and M. Lali, “Formalizing a hierarchical file system,” in *Proceedings of the 14th BCS-FACS Refinement Workshop*, pp. 67–85, Dec. 2009.

[21] A. K. KV, M. Cao, J. R. Santos, and A. Dilger, “Ext4 block and inode allocator improvements,” in *Linux Symposium*, vol. 1, 2008.

[22] J. Griffioen and R. Appleton, “The design, implementation, and evaluation of a predictive caching file system,” Tech. Rep. CS-264-96, Department of Computer Science, University of Kentucky, 1996.

[23] 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*, (Copenhagen, Denmark), pp. 55–74, July 2002.

[24] X. Wang, “Personal communication,” Mar. 2017.

[25] H. Sigurbjarnarson, J. Bornholt, E. Torlak, and X. Wang, “The Yggdrasil toolkit,” 2017. `https://github.com/locore/yggdrasil/`.

[26] J. Kara, "[PATCH] ext4: Forbid journal_async_commit in data=ordered mode." `http://permalink.gmane.org/gmane.comp.file-systems.ext4/46977`, Nov. 2014.

[27] M. Zheng, J. Tucek, F. Qin, and M. Lillibridge, "Understanding the robustness of SSDs under power fault," in *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)*, (San Jose, CA), pp. 271–284, Feb. 2013.

[28] 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)*, (Pacific Grove, CA), pp. 1–15, Oct. 1991.

[29] 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)*, (Farmington, PA), pp. 1–17, Nov. 2013.

[30] R. Cox, M. F. Kaashoek, and R. T. Morris, "Xv6, a simple Unix-like teaching operating system," 2014. `http://pdos.csail.mit.edu/6.828/2014/xv6.html`.

[31] A. Pavlo, "Python implementation of tpc-c," 2017. `https://github.com/apavlo/py-tpcc`.

[32] T. Chajed, "Verifying an i/o-concurrent file system," Master's thesis, Massachusetts Institute of Technology, 2017.