Certifying a Crash-safe File System

by

Haogang Chen

B.S., Peking University (2007)
M.S., Peking University (2010)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2016

Author . . . .
Department of Electrical Engineering and Computer Science
August 29, 2016

Certified by . . . .
M. Frans Kaashoek
Charles Piper Professor of Computer Science and Engineering
Thesis Supervisor

Certified by . . . .
Nickolai Zeldovich
Associate Professor
Thesis Supervisor

Accepted by . . . .
Leslie A. Kolodziejski
Chair, Department Committee on Graduate Theses

Certifying a Crash-safe File System
by
Haogang Chen

Submitted to the Department of Electrical Engineering and Computer Science
on August 29, 2016, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science

Abstract

File systems are a cornerstone for storing and retrieving permanent data, yet they are complex enough to have bugs that might cause data loss, especially in the face of system crashes.

FSCQ is the first file system that (1) provides a precise specification for the core subset of POSIX file-system APIs; and the APIs include fsync and fdatasync, which allow applications to achieve high I/O performance and crash safety, and that (2) provides a machine-checked proof that its I/O-efficient implementation meets this precise specification. FSCQ's proofs avoid crash-safety bugs that have plagued file systems, such as forgetting to insert a disk-write barrier between writing the data from the log and writing the log's commit block. FSCQ's specification also allows applications to prove their own crash safety, avoiding application-level bugs such as forgetting to invoke fsync on both the file and the containing directory. As a result, applications on FSCQ can provide strong guarantees: they will not lose data under any sequence of crashes.

To state FSCQ's theorems, FSCQ introduces the Crash Hoare Logic (CHL), which extends traditional Hoare logic with a crash condition, a recovery procedure, and logical address spaces for specifying disk states at different abstraction levels. CHL also reduces the proof effort for developers through proof automation. Using CHL, the thesis developed, specified, and proved the correctness of the FSCQ file system. FSCQ introduces a *metadata-prefix* specification that captures the properties of fsync and fdatasync, based on Linux ext4's behavior. FSCQ also introduces *disk sequences* and *disk relations* to help formalize the metadata-prefix specification. The evaluation shows that FSCQ enables end-to-end verification of application crash safety, and that FSCQ's optimizations achieve I/O performance on par with that of Linux ext4.

Thesis Supervisor: M. Frans Kaashoek
Title: Charles Piper Professor of Computer Science and Engineering

Thesis Supervisor: Nickolai Zeldovich
Title: Associate Professor

Acknowledgments

I am honored to have Frans Kaashoek and Nickolai Zeldovich as my advisors. This dissertation would not be possible without their constant guidance, enthusiastic support and extraordinary patience. They demonstrate a high bar in every aspect, and their impact on my life is immeasurable. I would like to express my sincere gratitude to Adam Chlipala for his wise advice from broad directions to minuscule details.

I would like to thank my collaborators in the FSCQ project. Nickolai and Frans made substantial contributions to the design and implementation of CHL. FSCQ's hash model was independently developed by Stephanie Wang [68]. The core of the Rec library was written by Daniel Ziegler. Tej Chajed made substantial improvements to the CHL infrastructure.

This research was greatly improved by the feedback of Srivatsa Bhat, Cody Cutler, Jon Gjengset, Chris Hawblitzel, Eddie Kohler, Butler Lampson, Robert Morris, Neha Narula, Bryan Parno, Frank Wang, Xi Wang and the anonymous reviewers of HotOS 2015 and SOSP 2015.

I have had the incredible fortune to work with amazing friends and colleagues in the PDOS group at MIT. I would like to give special thanks to Xi Wang, Yangdong Mao, Taesoo Kim, Dong Zhou, Zhihao Jia, Ramesh Chandra and Austin Clements for inspiring me to work on several intriguing projects while in PDOS.

Last but not least, I thank my parents and my angel, Ze, for their unconditional love, encouragement and support.

⋆ ⋆ ⋆

This research was supported in part by NSF awards CNS-1053143 and CCF-1253229, and by the CSAIL cybersecurity initiative. The dissertation incorporates and extends work published in the following papers:

> Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Using Crash Hoare Logic for certifying the FSCQ file system. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, Monterey, California, October 2015.
>
> Haogang Chen, Daniel Ziegler, Adam Chlipala, M. Frans Kaashoek, Eddie Kohler, and Nickolai 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.

Contents

1 Introduction 11

1.1 Crash safety 11

1.2 Specification framework for crash safety 13

1.3 Specifying file-system behavior 14

1.4 Building the file system 15

1.5 Contributions 16

1.6 Outline 17

2 Related Work 19

2.1 Finding and fixing bugs in file systems 19

2.2 Formal reasoning about file systems 20

2.3 Certified systems software 21

2.4 Reasoning about failures 21

3 Background 23

3.1 File-system basics 23

3.2 Logging protocol and optimizations 24

3.3 Program correctness 26

4 Crash Hoare Logic 29

4.1 Disk model 29

4.2 Crash conditions 31

4.3 Logical address spaces 34

4.4 Recovery execution semantics 37

5 Proving specifications 39

5.1 Overview 39

5.2 Proving without recovery 41

5.3 Proving recovery specifications 43

6 Certifying FSCQ's logging system 45

6.1 Overview 45

6.2 Representation invariants 47

6.3 Logging-system specifications 49

6.4 Logging with checksums 53

6.5 Log bypass 56

7 Specification for deferred writes 59

7.1 Example application pattern 59

7.2 What should the specification be? 61

7.3 Disk sequences 63

7.4 Disk relations 65

8 Building a file system 71

8.1 Overview 71

8.2 End-to-end specification 72

8.3 Using address spaces 73

8.4 Resource allocation 74

8.5 Buffer cache 76

8.6 On-disk data structures 76

8.7 Prototype implementation 77

9 Evaluation 81

9.1 Application and I/O performance 81

9.2 Bug discussion 85

9.3 Specification correctness 86

9.4 Development effort 88

10 Conclusion and future directions 91

Figures and tables

1-1 Overview of FSCQ's implementation 15

4-1 Basic operational semantics of CHL 30
4-2 Specification for disk_write 32
4-3 Pseudocode of atomic_two_write 35
4-4 Specification for atomic_two_write 35
4-5 Part of log_rep representation invariant for the basic logging protocol . . . 36
4-6 Specification of log_recover 37
4-7 Specification for atomic_two_write with recovery 38

5-1 Example control flow of a CHL procedure 40
5-2 Specification for log_begin 41
5-3 Specification for log_write 42

6-1 Illustration of FscqLog layers and the timeline of a transaction 46
6-2 Part of Applier's representation invariant 48
6-3 Part of GroupCommit's representation invariant 48
6-4 Part of LogAPI's representation invariant 49
6-5 Call graph for log_commit and log_flush 49
6-6 Specification for LogAPI's log_commit 50
6-7 Specification for GroupCommit's group_commit 50
6-8 Specification for LogAPI's log_flush 51
6-9 Specification for Applier's applier_flush 51
6-10 Specification for Applier's applier_apply 52
6-11 Specification for DiskLog's disklog_append 52
6-12 On-disk layout of FscqLog 53
6-13 Pseudocode of DiskLog layer 54
6-14 Specification for LogAPI's log_dwrite 57
6-15 Specification for LogAPI's log_dsync 57

7-1 Pseudocode for a crash-safe application pattern 60

7-2 An illustration of FSCQ's disk-sequence abstraction 63
7-3 Specification for unlink 64
7-4 Specification for fsync on directories 65
7-5 An illustration of disk relations 66
7-6 Specification for file write that bypasses the write-ahead log 67
7-7 Specification for fdatasync 68

8-1 FSCQ components 71
8-2 FSCQ on-disk layout 72
8-3 Specification for rename with recovery 73
8-4 Specification for writing to a file through the log 74
8-5 Representation invariant for FSCQ's file layer 75
8-6 FSCQ's on-disk inode layout 77
8-7 Combined lines of code and proof for FSCQ components 78

9-1 Application performance of FSCQ running on a hard disk drive 82
9-2 Application performance of FSCQ running on an SSD drive 83
9-3 I/O performance of FSCQ 84
9-4 Bugs precluded by FSCQ's specifications 85
9-5 Lines of specification code for FSCQ system calls 88

One

Introduction

This dissertation presents a novel approach to building and certifying a file system that behaves correctly in the face of system crashes *and* achieves high I/O efficiency.

The rest of this chapter introduces crash safety, a key property for file-system correctness, explains why building a file system that is both crash-safe and I/O efficient is difficult, and proposes our solution to formalizing and certifying a crash-safe file system.

1.1 Crash safety

Applications rely on file systems to store their data, but even carefully written file systems may have bugs that cause data loss, especially in the face of system crashes. We focus on fail-stop crashes where the system halts and subsequently reboots due to power failure, software panics, etc. Because many file-system operations require multiple disk writes to complete, a crash that happens in between these updates can leave the file system's internal data structure in a partially updated state. Once the system reboots, a recovery procedure must bring the file system back to a consistent state, so that future operations can run properly. We refer to this property as *crash safety*.

Recovering from crashes is important because inconsistent state can lead to data loss or data corruption. For example, using rename to move a file across directories usually involves two steps: unlinking the file from the source directory and linking it to the destination directory. If the computer crashes in the middle, depending on the order of operations, the file could appear in both directories or neither of them. In the latter case, the data stored in the file is lost; in the former case, a subsequent attempt to remove the file from one directory can corrupt the data of the same file in the other directory (assuming the file system recycles the file's data blocks while it still appears elsewhere).

Achieving crash safety is challenging. Since crashes can happen at any time in the execution of the file system, it is difficult to reason through all possible executions to determine if the file system will operate correctly, and, if not, how to recover from the problematic scenario. Furthermore, modern hard drives usually support asynchronous I/O—that disk writes do not persist immediately and can be reordered—in order to achieve good performance. File-system developers must insert write barriers (*disk syncs*) at appropriate code locations where they believe that ordering is important; and disk sync is an expensive operation that should be avoided whenever possible. This asynchronous I/O model further complicates reasoning about crash safety.

The classic recipe for achieving crash safety is *write-ahead logging*, where a file-system call does not directly update the on-disk data structures representing the file system. Instead, it places a description of all the disk writes it wishes to make in a *log* on the disk. Once the system call has logged all of its writes, it marks them as a complete (*committed*) transaction. At that point, the entire transaction is applied to the on-disk data structures and erased from the log. If the system crashes and reboots, the recovery procedure replays each committed transaction in turn from the log, but ignores incomplete ones at the end of the log. The log makes operations atomic with respect to crashes: after recovery, either all of an operation's writes appear on the disk, or none of them does.

While conceptually simple, write-ahead logging imposes extra disk writes and expensive disk syncs. Real-world file systems usually implement sophisticated optimizations to increase disk throughput. These optimizations include deferring writing buffered data to the disk, grouping many transactions into a single I/O operation, checksumming log entries, and bypassing the write-ahead log entirely when writing to file data blocks.

These optimizations complicate both the implementation and the semantics of a file system. On one hand, it becomes increasingly hard to show that a certain optimization, or the combination of several optimizations, preserves the semantics of the original design, as evidenced by bugs that most file systems have experienced. For example, it took 6 years for ext4 developers to realize that two optimizations (data writes that bypass the log and log checksum) taken together can lead to disclosure of previously deleted data after a crash [42], which was fixed in November of 2014 by forbidding users from mounting an ext4 file system with both log bypass and log checksum.

On the other hand, some optimizations indeed relax the persistence guarantee provided by the file system, thus changing the interface semantics. For example, to support log bypass, file systems offer calls such as fsync, which give applications precise control over when to materialize changes to specific files to the disk. Unfortunately, calls such as fsync interact in subtle ways with other file-system calls, and their behaviors under crashes are poorly documented. As a result, it is not uncommon for application writers to use them incorrectly. Recent studies [10, 57, 78] have investigated several such bugs in widely used applications, including well-known database applications, and have shown that these bugs can lead to data loss.

Current approaches to building crash-safe file systems and applications fall roughly into three categories (see Chapter 2 for more details): testing, program analysis, and model checking. Although they are effective at finding bugs in practice, none of them can guarantee the absence of crash-safety bugs in actual implementations. This dissertation focuses precisely on this issue: building an I/O-efficient file system with machine-checkable proofs that it correctly recovers from crashes at any point, and providing a precise specification about the crash-safety property of file-system interfaces to applications.

1.2 Specification framework for crash safety

Theorem provers help programmers write machine-checkable proofs that a program meets its specification, a process called program certification. Researchers have used theorem provers for certifying real-world systems such as compilers [49], small kernels [47], kernel extensions [69], and simple remote servers [33], but none of these systems are capable of reasoning about file-system crashes. Reasoning about crashes is more complicated because that involves considering not only the states before and after some operation, but also all possible intermediate states when the operation crashes in the middle.

Building an infrastructure for reasoning about file-system crashes poses several challenges. Foremost among those challenges is the need for a specification framework that allows the file-system developer to state the system behavior under crashes. Second, the framework must be able to capture realistic hardware characteristics, such as asynchronous disk writes, so that the implementation of a file system can achieve good I/O performance. Third, the framework must allow modular development: developers should be able to specify and verify each component in isolation and then compose verified components. For instance, once a logging layer has been implemented, file-system developers should be able to prove crash safety in the inode layer simply by relying on the fact that logging ensures atomicity; they should not need to consider every possible crash point in the inode code. Finally, it is important that the framework allows for proofs to be automated, so that one can make changes to a specification and its implementation without having to redo all of the proofs manually.

A core contribution of this dissertation is a specification framework called *Crash Hoare Logic* (CHL) that addresses the above challenges. CHL extends classic Hoare logic [36] with *crash conditions*, which allow programmers to specify what invariants hold in case of crashes. CHL defines a realistic *disk model* that captures the notion of multiple outstanding writes in the disk controller; the model allows file-system developers to reason about all possible disk states that might result when some subset of these writes persists after a crash. CHL supports the construction of modular systems through a notion of *logical address spaces*, and uses *representation invariants* to hide lower-level details from upper layers. CHL incorporates the notion of a *recovery procedure* that runs after a crash; this enables programmers to write concise end-to-end specifications that account for both normal execution and the execution of recovery procedures. Finally, CHL allows for a high degree of proof automation by employing specifications based on *separation logic* [58].

1.3 Specifying file-system behavior

To prove a file system correct, we must have a specification of what a file system does. Somewhat surprisingly, no precise specification exists. For example, the POSIX standard is notoriously vague on what crash-safety guarantees file-system operations provide. A particular concern is the guarantees provided by fsync and fdatasync. Unfortunately, file systems provide imprecise promises on exactly what data is flushed to the disk as the result of these calls. In fact, for the Linux ext4 file system, it depends on the options that an administrator specifies when mounting the file system [57]. Because of this lack of precision, applications such as databases and mail servers, which try hard to make sequences of file creates, writes, and renames crash-safe by inserting fsyncs and fdatasyncs, may still lose data when the file systems they are running on crash at inopportune times [10, 78].

This dissertation proposes a precise *metadata-prefix* specification for fsync and fdatasync, among other POSIX file-system APIs, based on the default behavior of the Linux ext4 file system. The essence is that fdatasync(f) on a file f flushes just the data (contents) of that file, and that fsync is a superset of fdatasync: it flushes both data and metadata (attributes, directories). Furthermore, fsync flushes *all* metadata changes. That is, fsync on a directory effectively ignores its argument: fsync(d) on directory d flushes changes to other unrelated directories as well. In addition, the file system is allowed to flush any file's data to disk at any time, and it is allowed to flush metadata operations in exactly the order they were issued by the application. This specification strikes a reasonable balance between ease of use for application programmers and allowing file systems to implement optimizations that provide high I/O performance. One argument that the metadata-prefix specification is reasonable is that it matches the behavior of ext4 in its default configuration, as a side effect of having a single write-ahead log for all metadata.

The main challenge in formalizing the metadata-prefix specification and proving that a file system obeys this specification is writing down the specification for the file system's external and internal interfaces. This dissertation contributes three specification techniques that address this challenge: *disk sequences* to capture metadata ordering, a *hash model* for reasoning about log checksums, and *disk relations* to reason about how file data writes that bypass the log interact with metadata operations.

The benefit of metadata-prefix specification is twofold: First, it allows file-system developers to prove that the optimizations that they implement do not violate the metadata-prefix property. Second, it enables application developers to *prove* their own application-level crash-safety properties, based on the guarantees provided by the metadata-prefix specification. For instance, developers of a mission-critical application such as a database can prove that they are using fsync correctly and that the database does not run the risk of losing data.

1.4 Building the file system

We have implemented the CHL specification framework and built FSCQ, a certified crash-safe file system, with the widely used Coq theorem prover [16], which provides a single programming language for both proving and implementing. Figure 1-1 shows the components involved in the implementation.

CHL is a specification language embedded in Coq that allows a file-system developer to write specifications that include the notion of crash conditions and a recovery procedure, and to prove that their implementations meet these specifications. We have stated the semantics of CHL and proven it sound in Coq.

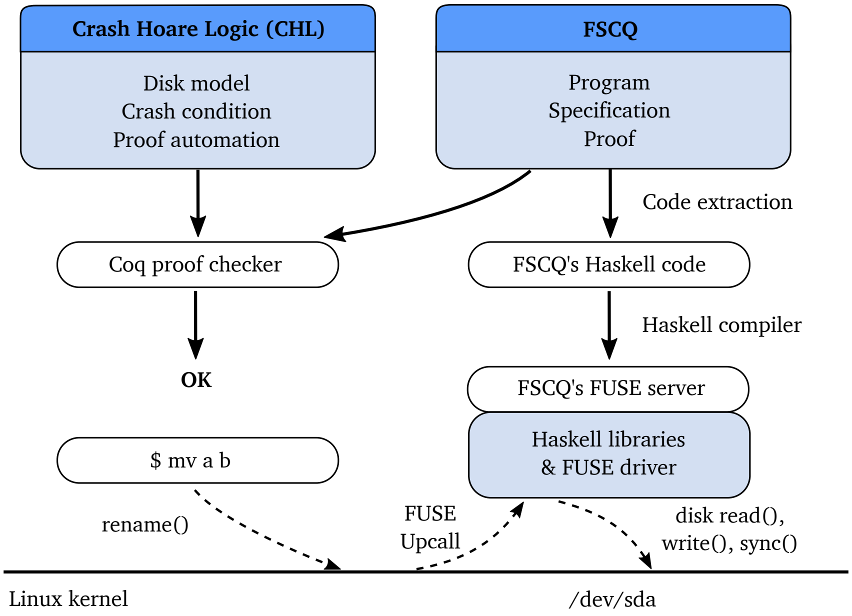


Figure 1-1: Overview of FSCQ's implementation. Shaded boxes denote source code written by hand. Solid lines denote processes. Dashed lines denote the call graph.

We implemented and certified FSCQ using CHL. That is, we wrote the metadata-prefix specifications for the core subset of the POSIX system calls using CHL, implemented those calls inside of Coq, and proved that the implementation of each call meets its specification. CHL reduces the proof burden because it automates the chaining of internal specifications. Despite the automation, writing specifications and proofs still took a significant amount of time, compared to the time spent writing the implementation.

As a target for FSCQ's completeness, we aimed for the core features as the Linux ext4 file system. FSCQ supports fewer features than today's Unix file systems; for example, it lacks support for concurrency and permissions. But, it supports the core POSIX file-system calls, including support for large files using indirect blocks, nested directories, rename and fsync, etc. FSCQ also implements standard optimizations such as deferring disk writes, group commit, log bypass for file data, log checksumming, and so on.

Using Coq's extraction feature, we extract a Haskell implementation of FSCQ. We run this implementation combined with a small (uncertified) Haskell driver as a FUSE [27] user-level file server. This implementation strategy allows us to run unmodified Unix applications but pulls in Haskell, our Haskell driver, and the Haskell FUSE library as trusted components.

To verify that FSCQ's optimizations achieve high I/O performance, we ran FSCQ using FUSE on top of Linux and ran several benchmarks. Experimental results demonstrate that FSCQ achieves *high I/O efficiency* similar to that of the Linux ext4 file system: both ext4 and FSCQ perform a similar number of I/O operations for a given benchmark.

A limitation of the FSCQ prototype is that it has high CPU overheads, because it generates an executable implementation by extracting Haskell code from Coq. This results in higher CPU consumption than in Linux ext4 and increases the overall trusted computing base. We expect that complementary techniques for producing certified assembly [2, 31, 46, 49, 74] would avoid FSCQ's CPU overheads and reduce its TCB.

1.5 Contributions

This dissertation's broad contribution is an approach to building certified file systems that are both I/O-efficient and safe under computer crashes. This dissertation makes the following intellectual contributions:

- The Crash Hoare Logic (CHL), which allows programmers to specify what invariants hold in case of crashes and which incorporates the notion of a recovery procedure that runs after a crash. CHL supports the construction of modular systems through a notion of logical address spaces. CHL also allows for a high degree of proof automation.

- A model for asynchronous disk writes, specified using CHL, that captures the notion of multiple outstanding writes in the disk controller; the model allows file-system developers to reason about all possible disk states that might result when some subset of these writes persists after a crash.

- A write-ahead log, FscqLog, certified with CHL, that provides all-or-nothing transactions on top of asynchronous disk writes, and which provides a simple synchronous disk abstraction to other layers in the file system. FscqLog also implements performance-critical optimizations such as deferring disk writes, group commit, log bypass for file data, checksummed log commit, and so on.

- The FSCQ file system, built on top of FscqLog, which is the first file system to be certified for crash safety, and is I/O efficient. It embeds design patterns that work well for constructing modular certified file systems; a certified generic object allocator that can be instantiated for disk blocks and inodes; and a certified library for laying out data structures on disk.
- A *metadata-prefix specification* of the core subset of the POSIX file-system API that captures its semantics under crashes, based on the behavior of the Linux ext4 file system. It includes a precise specification for fsync and fdatasync. The specification can help file-system developers to prove the correctness of their optimizations; and it enables application developers to build applications on top of the POSIX API and reason precisely about crash safety.
- Several specification techniques to help formalize the metadata-prefix property: *disk sequences* to capture meta-data ordering, a *hash model* for reasoning about journal checksums, and *disk relations* to reason about file-data writes that bypass the log.
- An evaluation that shows that FSCQ provides comparable I/O performance to ext4, that FSCQ can run a wide range of unmodified Unix applications with usable throughput, and that FSCQ's specifications and proofs eliminate bugs found in other file systems.
- A case study of code evolution in FSCQ, demonstrating that CHL combined with FSCQ's design allows for incremental changes to *both* the proof and the implementation. This suggests that the FSCQ file system is amenable to incremental improvements.

1.6 Outline

The rest of the dissertation elucidates how to specify, build and certify FSCQ in depth.

We first relate FSCQ with previous work in Chapter 2, and review some basic concepts in file-system design and program certification in Chapter 3.

We then introduce Crash Hoare Logic (CHL), the logic framework that forms the basis of FSCQ, in Chapter 4. We start from defining CHL's asynchronous disk model, and demonstrate how to write specifications using CHL's crash conditions, logical address spaces and recovery execution semantics. We explain how to prove CHL specifications in Chapter 5, and illustrate how CHL's proof automation can help reduce the proof burden.

In Chapter 6, we describe how to build and certify FscqLog, the logging system that provides crash safety for FSCQ; we also show how FscqLog's layered design simplifies the implementation and certification of sophisticated performance optimizations, such as group commit, log bypass and log checksums.

Chapter 6 discusses the specification for supporting deferred-write, an essential optimization that sacrifices the file system's durability guarantee to achieve better performance. We propose the metadata-prefix specification that captures the essence of fsync and fdatasync, and formalize the metadata-prefix property using novel specification techniques such as disk sequences and disk relations.

Finally, in Chapter 8 we present the details of building the complete FSCQ file system, including how to structure the file system in order to aid proving, and how to write the file system's end-to-end specification and internal specifications. We share our experiences and lessons learned in building several components of FSCQ, and point out the limitations of our prototype. Chapter 9 evaluates FSCQ's performance, correctness and development effort.

Chapter 10 concludes and explores some promising future directions.

Two

Related Work

FSCQ is the first file system with a machine-checked proof of crash safety. This section relates FSCQ to several lines of prior work.

2.1 Finding and fixing bugs in file systems

Previous papers have studied bugs in file systems [50] and in applications that make inconsistent assumptions about the underlying file systems [78]. One recent example is the 2013 Bitcoin bounty for tracking down a serious bug that corrupted financial transactions stored in a LevelDB database [24].

It is widely acknowledged that it is easy for application developers to make mistakes in ensuring crash safety for application state [57]. For instance, a change in the ext4 file-system implementation changed the observable crash behavior of the file system, as far as the application could see, leading to many applications losing data after crashes [17, 30], due to a missing fsync call to ensure that the contents of a new file are flushed to disk [10]. The ext4 developers, however, maintained that the file system never promised to uphold the earlier behavior, so this was an application bug. Similar issues crop up with different file-system options, which often lead to different crash behavior [57].

Model-checking [75–77] and fuzz-testing [39] techniques are effective at detecting file-system bugs. They enumerate possible user inputs and disk states, inject crashes, and look for cases where the file system encounters a bug or violates its invariants. These techniques find real bugs, and we use some of them to do an end-to-end check on FSCQ and its specifications. However, these techniques often cannot check all possible execution paths and thus cannot guarantee bug-free systems.

When faced with file-system inconsistencies, system administrators run tools such as fsck [5: §42] and SQCK [32] to detect and repair corruption [20]. By construction, certified file systems avoid inconsistencies caused by software bugs.

2.2 Formal reasoning about file systems

Building a correct file system has been an attractive goal for verification [26, 40]. There is a rich literature of formalizing file systems using many specification languages, including ACL2 [7], Alloy [41], Athena [4], Isabelle/HOL [70], PVS [35], SSL [28], Z [8], KIV [23], and combinations of them [25]. Most of these specifications do not model crashes. The ones that do, such as the work by Kang and Jackson [41], do not connect their specifications to executable implementations.

Prior work has largely focused on specifying the file-system API [10, 59] using trace-based specifications [34]. However, no prior specifications have addressed bypassing the log for file data writes and fdatasync. Furthermore, this trace-based approach does not work well for reasoning about the internals of a file system, which is necessary in order to prove the correctness of the file system itself. FSCQ introduces disk sequences and relations to address this challenge.

The closest effort to FSCQ is work in progress by Schellhorn, Pfähler, and others to verify a flash file system called Flashix [22, 56, 62]. They aim to produce a verified file system for raw flash, to support a POSIX-like interface, and to handle crashes. One difference from our work is that Flashix specifications are abstract state machines; in contrast, CHL specifications are written in a Hoare-logic style with pre- and postconditions. One downside of CHL is that all procedures (including internal layers inside the file system) must have explicit pre- and postconditions. However, CHL's approach has two advantages. First, developers can easily write CHL specifications that capture asynchronous I/O, such as for write-back disk caches. Second, it allows for proof automation. Another difference between FSCQ and Flashix is that FSCQ provides specifications for fsync and fdatasync, while Flashix does not.

Ntzik et al. [54] extend Views [21] framework with fault conditions, which are similar to CHL's crash conditions. Because Views deals with shared-memory concurrency, their logic models both volatile state and durable state. CHL models only durable state, and relies on its shallow embedding in Coq for volatile state. Their paper focuses on the design of the logic, illustrating it with a logging system modeled on the ARIES recovery algorithm [51]. Their aim isn't to build a complete verified system, and their logic lacks, for example, logical address spaces, which help proof automation and certifying FSCQ in a modular fashion.

Cogent [2] generates highly efficient executable code for a file system, and supports deferred writes, but lacks a specification and proof for the entire file system. Our FSCQ prototype suffers from CPU overheads due to extracting code to Haskell. FSCQ could benefit from using the DSL approach from Cogent to generate more efficient code and reduce the size of the trusted computing base.

2.3 Certified systems software

The last decade has seen tremendous progress in certifying systems software, which inspired us to work on FSCQ. The CompCert compiler [49] is formally specified and verified in Coq. As a compiler, CompCert does not deal with crashes, but we adopt CompCert's *validation* approach for proving FSCQ's cache-replacement algorithm.

The seL4 project [47] developed a formally verified microkernel using the Isabelle proof assistant. Since seL4 is a microkernel, its file system is not part of the seL4 kernel. seL4 makes *no* guarantees about the correctness of the file system. seL4 itself has no persistent state, so its specification does not make statements about crashes.

A recent paper [45] argues that file systems deserve verification too, and describes work-in-progress on BilbyFS, which uses layered domain-specific languages, but appears not to handle crashes [44]. Two other position papers [1, 12] also argue for verifying storage systems. One of them [12] summarizes our initial thinking about different ways of writing system specifications, including Hoare-style ones.

Verve [74], Bedrock [13, 14], Ironclad [33], and CertiKOS [31] have shown that proof automation can considerably reduce the burden of proving. Of these, Bedrock, Verve, and Ironclad are the most related to this work, because they support proof automation for Hoare logic. We also adopt a few Coq libraries from Bedrock.

Work on formalization of cryptographic protocols has explored how to formalize collision-resistant hash functions. For example, the RF* verification-oriented programming language [6] maintains, as auxiliary state, a mutable global dictionary from inputs to their hashes, including only the inputs already used in the current program execution, where an error is signaled if a new hash request leads to a collision in this dictionary. FSCQ builds on this idea of treating hash collisions as program non-termination, and extends it to handle crashes by reasoning about subsets of hash histories.

2.4 Reasoning about failures

Failures are a core concern in distributed systems, and TLA [48] has been used to prove that distributed-system protocols adhere to some specification in the presence of node and network failures, and to specify fault-tolerant replicated storage systems [29]. However, TLA reasons purely about designs and not about executable code. Verdi [71] reasons about distributed systems written in Coq and can extract the implementation into executable code. However, Verdi's node-failure model is a high-level description of what state is preserved across reboots, which is assumed to be correct. Extracted code must use other software (such as a file system) to preserve state across crashes, and Verdi provides no proof that this is done correctly. FSCQ and CHL address this complementary problem: reasoning about crash safety starting from a small set of assumptions (e.g., atomic block writes).

Project Zap [55, 67] and Rely [11] explore using type systems to mitigate transient faults (e.g., due to charged particles randomly flipping bits in a CPU) in cases when the system keeps executing after a fault occurs. These models consider possible faults at each step of a computation. The type system enables the programmer to reason about the results of running the program in the presence of faults, or to ensure that a program will correctly repeat its computation enough times to detect or mask faults. In contrast, CHL's model is fail-stop: every fault causes the system to crash and reboot.

Schlichting [63] describes a variant of Hoare logic with support for crashes, which models the system using a fail-stop processor and a stable storage. Every program in the fail-stop processor is a sequence of fault-tolerant actions, which combines the code with a recovery procedure. In contrast, CHL decouples recovery and normal execution by introducing explicit crash conditions.

Andronick [3] verified anti-tearing properties for smart-card software, which involves being prepared for the interruption of code at any point. This verification proceeds by instrumenting C programs to call, between any two atomic statements, a function that may nondeterministically choose to raise an uncatchable exception. In comparison, CHL handles the additional challenges of asynchronous disk writes and layered abstractions of on-disk data structures.

THREE

Background

To better understand the challenge in building and certifying a crash-safe file system, this chapter explains basic concepts of modern file systems, describes a simple logging protocol and several optimizations. Finally, we introduce Hoare logic and separation logic, the formal systems that form the basis of Crash Hoare Logic.

3.1 File-system basics

The purpose of a file system is to organize and store data. In a Unix-like file system, data is stored in a *file*, which is an array of bytes. Several files can be grouped inside a *directory*, which might also contain other directories (called *sub-directories*), forming a tree-like hierarchical structure. The file system maintains on-disk data structures to represent the tree of named directories and files, to record the identities of the blocks that hold each file's content, and to record which areas on the disk are free, as we explain next.

Buffer cache. Accessing a disk is orders of magnitude slower than accessing memory, so the file system must maintain an in-memory cache of popular blocks in its *buffer cache*. The buffer cache holds the content of a disk block in memory when the block is being read or written to. If the cache is full, the buffer cache typically evicts the least recently used block to make room for the new one. For writes, the buffer cache can choose to write the block to the underlying disk right away (*write-through*), or to delay the disk write until the block is to be evicted from the cache (*write-back* or *deferred writes*).

Files and inodes. The on-disk representation of a file is called an *inode*, which holds important metadata of a file, such as the file's size and time stamps, and a list of data-block numbers that belong to the file. File systems usually allocate a contiguous area of disk to store all inodes. Every inode is of the same size, so it is easy, given a number n, to find the n-th inode on the disk. Therefore, every inode (and file) in the file system can be internally identified by an index n; we call this an inode number, or *i-num* for short. A special inode number is reserved for the root directory.

Directories and pathnames. A directory is implemented internally as a special kind of file, whose content is a sequence of *directory entries*. Each directory entry consists of a name and an inode number that points to another file or directory. File-system users can refer to a file or a directory in the directory tree symbolically using a *pathname*, which concatenates the name of each parent directory along the path from the root.

Allocation bitmaps. File systems use allocation bitmaps to track the usage of free blocks and inodes. For example, using a block bitmap, a file system can quickly tell which blocks are free without traversing the entire directory tree.

To see how these concepts work together, consider an example that copies a file "/path/src" into "/path/dst". The file system first locates the source file in the directory tree. To do this, it finds "path" in the root directory entries (which are usually cached in the memory), follows path's inode number i_{path}, loads path's directory entries and in turn locates the name "src" and the corresponding inode number i_{src}. Then the file system reads the content of the source file by reading from the disk blocks whose addresses are stored in i_{src}. To create the destination file, the file system allocates a new inode i_{dst} and marks it as used in the inode allocation bitmap. It then allocates new disk blocks for "dst", records the allocated block numbers in i_{dst}, updates the block allocation bitmap accordingly and writes src's content into the newly allocated blocks. Finally, the file system appends a new pair ("dst", i_{dst}) into path's directory entries and writes the directory's new content back to the disk. Note that the procedure described here does not guarantee crash safety, which we will discuss next.

3.2 Logging protocol and optimizations

One of the most interesting problems in file-system design is crash recovery. Many file system operations involve multiple writes to the disk. For example, creating a file involves at least two disk writes: allocating an unused inode and linking the inode in the parent directory. If the system crashes in between the two writes, an inode might be marked as used in the inode bitmap, while does not belong to any directory.

File systems implement write-ahead logging to make sure these on-disk data structures are in a consistent state when the system crashes and restarts. The log usually resides in a fixed region of the disk and consists of two parts: a contiguous *log entries* region that stores the description of all the disk writes that an operation wishes to make, and a single *commit block* that stores the number of entries.

A basic logging protocol works as follows: (1) The logging system starts a transaction for a file-system operation. (2) For each disk write issued by the operation, the logging system appends a new address/value pair to the log entries. (3) When the logging system commits the transaction, it issues a disk sync to persist the log entries on disk, updates the commit block to reflect the length of the log, and then syncs the disk again to complete the transaction. (4) The logging system applies the changes in the log to their actual disk locations and issues a disk sync to persist the change. (5) Finally, it truncates the log by writing zero to the commit block and syncs the disk. (6) Whenever the system crashes and reboots, a *recovery procedure* runs. It reads the log based on the length in the commit block; and if the log is non-empty, it goes to step (4) to apply the log.

The correctness of the above protocol relies on an important assumption: updating the commit block is both *atomic* and *synchronous*. Atomicity ensures that the commit block contains either the old length or the new length; synchrony ensures that the disk controller does not reorder the update to the commit block with other disk writes, and is enforced by issuing two write barriers (disk syncs) before and after the update. Under this protocol, a crash midway through a transaction will result in a length of zero in the commit block; a crash after a commit (but before apply completes) will result in a non-zero length, and the log must contain exactly the same number of valid entries.

The basic logging protocol imposes disk writes and expensive write barriers. For example, an operation that writes to a single disk block would require 4 disk writes (twice for the commit block and twice for the block content) and 4 disk syncs (before and after both updates to the commit block) to complete. Sophisticated file systems implements several optimizations to reduce the number of disk operations. For example, Linux ext4 employs *deferred apply*, *group commit*, *log checksum* and *log bypass*, as we explain below.

Deferred apply. After the logging system commits a transaction, it can defer applying the log entries. Subsequent transactions can simply append more entries to the log. When the log fills up, all entries can be applied at once. This reduces the amortized number of synchronous disk write barriers from 4 to 2 per transaction.

Group commit. The logging system can accumulate the writes of multiple system calls in memory, merge them into a single transaction, and issue a single commit for the merged transaction. Group commit amortizes number of disk syncs across multiple small transactions and potentially reduces the number of disk writes by merging writes from multiple transactions.

Log checksum. If the commit block contains a checksum of all the related log entries, it is safe to omit the disk sync between writing log entries and updating the commit block.

Log checksum further reduces the number of write barriers from 2 to 1 per transaction. The recovery procedure can determine whether to commit or abort an transaction by computing the checksum of the log entries and comparing it with the checksum stored in the commit block.

Log bypass and deferred writes. With log bypass, a file system can write file content directly to file data blocks, rather than going through the logging system, while metadata (file attributes and directory entries) updates still go through the log. When combined with a write-back buffer cache, log bypass relaxes the file system's consistency guarantee: It allows each file's data to be flushed independently of other files and of the metadata, by using fsync and fdatasync. This enables high performance for applications that perform in-place data modifications. Because both group commit and log bypass defer updating the disk (for metadata and non-metadata, respectively), we refer to them using an umbrella term: *deferred writes*.

Our FSCQ prototype implements all the above optimizations with a proof of correctness. To prove the implementation, we first need a precise specification that defines what is the correct semantics for the logging system. FSCQ's specification framework—Crash Hoare Logic—is based on *Hoare logic* and *separation logic*, which we describe next.

3.3 Program correctness

Hoare logic [36] is a classic formal system for reasoning about program correctness. Separation logic [58] extends Hoare logic to support predicates on storage state.

Hoare logic A Hoare logic specification is written as a *Hoare triple*, which describes how the execution of a piece of code changes the state of the computation. A Hoare triple is defined as:

$$\{P\}\ \mathbb{C}\ \{Q\} \tag{3.1}$$

where P corresponds to the precondition that should hold before the procedure $\mathbb{C}$ is run, and Q is the postcondition. To prove that a specification is correct, we must prove that procedure $\mathbb{C}$ establishes Q, assuming P holds before invoking $\mathbb{C}$. Two basic inference rules for Hoare logic are:

$$\frac{\{P\}\ \mathbb{C}_1\ \{Q\} \qquad \{Q\}\ \mathbb{C}_2\ \{R\}}{\{P\}\ \mathbb{C}_1;\mathbb{C}_2\ \{R\}} \tag{3.2}$$

$$\frac{P_1 \Rightarrow P_2 \qquad \{P_2\}\ \mathbb{C}\ \{Q_2\} \qquad Q_2 \Rightarrow Q_1}{\{P_1\}\ \mathbb{C}\ \{Q_1\}} \tag{3.3}$$

Equation (3.2) is the *composition rule* that enables chaining the specs of two sequential procedure $\mathbb{C}_1$ and $\mathbb{C}_2$. Equation (3.3) is the *consequence rule*, which allows to strengthen the precondition or to weaken the postcondition ($\Rightarrow$ denotes logical implication).

Separation logic Separation logic makes it easy to combine predicates on disjoint parts of a storage (such as the disk). A storage is defined as a partial function from addresses to values. The domain of store d is the set of valid addresses where d is defined:

$$\text{dom } d = \{\, a \mid d(a) \text{ is defined} \,\} \tag{3.4}$$

The basic predicates in separation logic are: (1) the **emp** assertion, which asserts that the store has no valid address; (2) the *points-to* relation, written as $a \mapsto v$, which asserts that address a is *the only valid address*, and it has value v; and (3) the separating conjunction operator ($\star$), in which given two predicates P and Q, $P \star Q$ means that the store can be split into two disjoint parts, where one satisfies the P predicate, and the other satisfies Q. More formally,

$$d \models \mathbf{emp} \quad \text{iff} \quad \text{dom } d = \varnothing \tag{3.5}$$

$$d \models a \mapsto v \quad \text{iff} \quad \text{dom } d = \{a\} \ \wedge \ d(a) = v \tag{3.6}$$

$$d \models P \star Q \quad \text{iff} \quad \exists d_1 d_2,\ d = d_1 \cdot d_2 \ \wedge \ d_1 \models P \ \wedge \ d_2 \models Q \ \wedge \ d_1 \perp d_2 \tag{3.7}$$

where $d_1 \perp d_2$ means dom $d_1 \cap$ dom $d_2 = \varnothing$, and "$\cdot$" is the function-union operator.

In addition to the standard inference rules from Hoare logic (eq. (3.2), eq. (3.3)), separation logic supports a very important rule called the *frame rule*:

$$\frac{\{P\}\ \mathbb{C}\ \{Q\}}{\{P \star R\}\ \mathbb{C}\ \{Q \star R\}} \tag{3.8}$$

The frame rule states that if a procedure runs on a small part of store that satisfies P and produces outcome Q, then it can also safely run on a bigger store that satisfies $P \star R$, producing the same outcome (Q), while leaving the additional part of the store (R) untouched.

The frame rule enables *local reasoning*, a key property to support modular proofs: we can prove the specifications for code that manages different parts of the disk (e.g., the inode region and the block bitmap) separately, and then safely compose them to prove a module built on top of them (e.g., the file layer).

Separation logic also enables a high degree of *proof automation*. For instance, the following rules allow to “rewrite” or “cancel” terms on both sides of an implication:

$$\frac{}{P \iff P \star \mathbf{emp}} \qquad \frac{}{P \star Q \iff Q \star P} \qquad \frac{}{P \star Q \star R \iff P \star (Q \star R)}$$

$$\frac{P \Rightarrow Q \qquad R \Rightarrow S}{P \star R \Rightarrow Q \star S} \qquad \frac{P \Rightarrow Q}{P \star R \Rightarrow Q \star R} \tag{3.9}$$

FOUR

Crash Hoare Logic

A key goal of this dissertation is to allow developers to certify the correctness of a file system formally—that is, to prove that it functions correctly during normal operation and that it recovers properly from any possible crashes. A file system might forget to zero out the contents of newly allocated directory or indirect blocks, leading to corruption during normal operation, or it might perform disk writes without sufficient barriers, leading to disk contents that might be unrecoverable. Prior work has shown that even mature file systems in the Linux kernel have such bugs during normal operation [50] and in crash recovery [77].

To prove that an implementation meets its specification, we must have a way for the developer to state what correct behavior is under crashes. Crash Hoare Logic (CHL) extends Hoare logic with an asynchronous disk model, crash conditions, logical address spaces, and recovery execution semantics. As we will show in Chapter 7, these features allow us to state precise specifications (e.g., in the case of rename, that once recovery succeeds, either the entire rename operation took effect, or none of it did) and prove that implementations meet them. However, for the rest of this chapter, we will consider much simpler examples to explain the basics of CHL.

4.1 Disk model

CHL is specialized to proving properties of a small domain-specific language (DSL) that manipulates the state of a hypothetical disk. Following separation logic [58], the disk is defined as a partial function from integer block numbers to block contents, where block numbers greater than the size of the disk do not map to anything.

One interesting question when defining the DSL is how to model an *asynchronous disk*. A modern hard drive usually has an internal write buffer. Writes issued to the disk do not persist immediately. Instead, recent writes are cached in the internal buffer; and it is up to the hard drive to decide when to flush these outstanding writes to the disk. More importantly, the hard drive is allowed to *reorder* buffered writes to improve disk throughput. If the system does not crash, a disk read always returns the last-written value for a given address. If the system crashes and reboots, however, one might observe that a write issued later before the crash has made it to the disk, while an earlier one has not. The hard drive also offers a write-barrier operation, called *disk sync*, which forces draining the write buffer. Sync guarantees that upon its return, there will be no outstanding writes, and all previous writes are persisted on the disk. Disk syncs are very expensive and should be avoided whenever possible.

To capture this asynchronous nature, CHL describes each block's contents using a *value-set*, instead of a single block value. A value-set is a non-empty set of block values, denoted by $\langle v,\ vs\rangle$, where v is a special element representing the last value written to the address, and vs is a set of previously written values that *might* persist on the disk if the system were to crash. A disk block is said to be *synced* if its value-set has a single value (i.e., $vs = \varnothing$); otherwise the block is *unsynced*.

Reading from a block returns the last-written value v, since even if there are previous values that might appear after a crash, in the absence of a crash a read should return the last write. Writing to a block makes the new value the last-written value and adds the old last-written value v to the set of previous values vs. If the system crashes and reboots, the disk nondeterministically "chooses" any value from v and vs as a block's last-written value; and a block's previous values become empty because there will be no outstanding writes in the disk's write buffer right after reboot. A global disk sync discards all previously written values vs for each block, so that the last-written values become the only possible value that will persist after a crash. Finally, reading or writing a block number that does not exist causes the system to "fail" (as opposed to finishing or crashing). Figure 4-1 illustrates CHL's basic operational semantics for a given block.

Event	**Transition**	**Resulting state**
disk_read(a)	explicit	$a \mapsto \langle v,\ vs\rangle$ and return v
disk_write(a, v')	explicit	$a \mapsto \langle v',\ \{v\} \cup vs\rangle$
disk_sync	explicit	$a \mapsto \langle v,\ \varnothing\rangle$
sync	implicit	$a \mapsto \langle v,\ vs'\rangle$ where $vs' \subseteq vs$
crash	implicit	$a \mapsto \langle v',\ \varnothing\rangle$ where $v' \in \{v\} \cup vs$

Figure 4-1: Basic operational semantics of CHL, assuming the initial state satisfies $a \mapsto \langle v,\ vs\rangle$ for a given address a. "explicit" means that the transition is triggered by stepping the program; "implicit" indicates the transition can happen any time.

CHL's semantics allow the disk to flush its write buffer at any time in the background (shown as "sync implicit" in Figure 4-1), which might discard *some* previous values for a block. This poses a challenge to write concise predicates about the resulting disk state using the regular points-to relation. For example, even though disk_read(a) does not change the disk's state, a background disk sync might drop some values from vs silently.

If that happens, the assertion $a \mapsto \langle v,\ vs \rangle$ will become false when disk_read(a) returns.

To mitigate this issue and make implicit disk syncs transparent to the specification, CHL introduces a relaxed *subset-points-to* relation, denoted by $a \rightarrowtail \langle v,\ vs \rangle$, which asserts that address a's last-written value is v, and its previously written values are a *subset* of vs. Formally, the subset-points-to relation can be defined in terms of the regular points-to relation:

$$a \rightarrowtail \langle v,\ vs \rangle \iff \exists vs',\ a \mapsto \langle v,\ vs' \rangle \ \land\ vs' \subseteq vs \tag{4.1}$$

It is easy to see that

$$a \mapsto \langle v,\ vs \rangle \ \Rightarrow\ a \rightarrowtail \langle v,\ vs \rangle \tag{4.2}$$

$$a \mapsto \langle v,\ \varnothing \rangle \iff a \rightarrowtail \langle v,\ \varnothing \rangle \tag{4.3}$$

which means, if we replace "$\mapsto$" with "$\rightarrowtail$" from Figure 4-1, all assertions about the resulting state will still hold. We can restate both the initial state and the resulting state of implicit sync as $a \rightarrowtail \langle v,\ vs \rangle$, making it essentially a no-op with respect to the $\rightarrowtail$ relation. FSCQ always uses $\rightarrowtail$ to describe the state of the physical disk (i.e., the hard drive).

CHL's disk model assumes that blocks are written atomically; that is, after a crash, a block must contain either the last-written value or one of the previous values, and partial block writes are not allowed. This is a common assumption made by file systems, as long as each block is exactly one sector, and we believe it matches the behavior of many disks in practice (modern disks often have 4 KB sectors). We could capture the notion of partial sector writes by specifying a more complicated operational semantics for crashes, but the disk model shown here matches the common assumption about atomic sector writes. We leave to future work the question of how to build a certified file system without that assumption.

4.2 Crash conditions

Given the asynchronous disk model and the subset-points-to relation, developers can write down predicates about disk states for any crash-free execution of the file-system code in Hoare logic and separation logic. The next question is: how to specify all possible intermediate disk states when a crash interrupts the execution?

Unsurprisingly, a Hoare logic specification "$\{P\}\ \mathbb{C}\ \{Q\}$" is insufficient to reason about crashes. In our file-system settings, the procedure $\mathbb{C}$ could be the implementation of the rename system call, or a lower-level operation such as allocating a disk block. In any case, $\mathbb{C}$ consists of a sequence of primitive disk operations (e.g., disk_read, disk_write and disk_sync), interspersed with computation, that manipulates the persistent state on disk. A crash may cause $\mathbb{C}$ to stop at any point in its execution and may leave the disk in a state where Q does not hold (e.g., in the rename example, the new file name has been created already, but the old file name has not yet been removed).

To reason about the behavior of a procedure in the presence of crashes, CHL extends Hoare logic with a new *crash condition*. A CHL specification is a quadruple written as:

$$\{P\}\ \mathbb{C}\ \{Q\}\ \{R\} \tag{4.4}$$

where $\{Q\}$ is the postcondition as before, which describes the state at the end of $\mathbb{C}$'s crash-free execution; and $\{R\}$ is the crash condition that describes the intermediate states during $\mathbb{C}$'s execution in which a crash could occur.

For example, Figure 4-2 shows the CHL specification for FSCQ's disk_write primitive. Note that in our implementation of CHL, these specifications are written in Coq code; we show here an easier-to-read version. Corresponding to the quadruple in eq. (4.4), the specification has four parts: the procedure about which we are reasoning, disk_write; the precondition, $\mathbf{disk} \models F \star a \rightarrowtail \langle v_0,\ vs\rangle$; the postcondition if there are no crashes, $\mathbf{disk} \models F \star a \rightarrowtail \langle v,\ \{v_0\} \cup vs\rangle$; and the crash condition, $\mathbf{disk} \models F \star a \rightarrowtail \langle v_0,\ vs\rangle \lor F \star a \rightarrowtail \langle v,\ \{v_0\} \cup vs\rangle$.

$$\begin{array}{ll}
\text{SPEC} & \textsf{disk\_write}\ (a, v) \\
\text{PRE} & \mathbf{disk} \models F \star a \rightarrowtail \langle v_0,\ vs\rangle \\
\text{POST} & \mathbf{disk} \models F \star a \rightarrowtail \langle v,\ \{v_0\} \cup vs\rangle \\
\text{CRASH} & \mathbf{disk} \models F \star a \rightarrowtail \langle v_0,\ vs\rangle \lor \\
 & \qquad\quad\ F \star a \rightarrowtail \langle v,\ \{v_0\} \cup vs\rangle
\end{array}$$

Figure 4-2: Specification for disk_write

The disk_write specification asserts through the precondition that the address being written, a, must be valid (i.e., within the disk's size), by stating that address a points to some value-set $\langle v_0,\ vs\rangle$ on disk. The specification's postcondition asserts that the block being modified will contain the new value-set $\langle v,\ \{v_0\} \cup vs\rangle$; that is, the new last-written value is v, and v_0 is added to the set of previous values. The specification also asserts through the crash condition that disk_write could crash in two possible states: either the write did not happen (a still has $\langle v_0,\ vs\rangle$), or it did (a has $\langle v,\ \{v_0\} \cup vs\rangle$).

Moreover, the specification asserts that the rest of the disk is unaffected: if other disk blocks satisfied some predicate F before disk_write, they will still satisfy the same predicate afterwards. This illustrates the power of local reasoning in separation logic: the specification only focuses on the address that disk_write updates, and it does not have to say anything about the rest of the disk.

As we mentioned in Section 4.1, CHL's disk model allows the disk to sync in the background. Therefore the specification of disk_write uses subset-points-to ($\rightarrowtail$) instead of the regular points-to ($\mapsto$) to describe the state of the disk. The specification also assumes that the frame predicate F should be agnostic to disk syncs, a property called *sync-invariant*; otherwise F might not hold for other disk blocks in the postcondition, as background syncs can discard previous values from any disk block. The formal definition of sync-invariant is:

$$\begin{aligned} &\forall d\ d',\quad d \models F \ \wedge\ \mathrm{dom}\ d = \mathrm{dom}\ d' \ \wedge\ \forall a,\ \exists v\ vs\ vs', \\ &\qquad d(a) = \langle v,\ vs \rangle \ \wedge\ d'(a) = \langle v,\ vs' \rangle \ \wedge\ vs' \subseteq vs \quad \Rightarrow \quad d' \models F \end{aligned} \tag{4.5}$$

which states that, if a disk d satisfies some predicate F, and (partially) syncing d yields a new disk d' that also satisfies F, then the predicate F is sync-invariant. It is easy to see that if F only uses "$\rightarrowtail$" (or any combination of such formulas using separating conjunction) to describe the disk state, then it must be sync-invariant. In contrast, a predicate that contains the regular points-to relation ($\mapsto$) is *not* sync-invariant.

The specification of disk_write captures two important behaviors of real disks—that I/O can happen asynchronously and that writes can be reordered—in order to achieve good performance. CHL could model a simpler synchronous disk by specifying that a points to a single value (instead of a set of values) and changing the crash condition to say that either a points to the new value ($a \mapsto v$) or a points to the old value ($a \mapsto v_0$). This change would simplify proofs, but this model of a disk would be accurate only if the disk were running in synchronous mode with no write buffering, which achieves lower performance.

One subtlety of CHL's crash conditions is that they describe the state of the disk *just before* the crash occurs, rather than just after. Right after a crash, CHL's disk model specifies that each block nondeterministically chooses one value from the set of possible values before the crash. For instance, the first line of Figure 4-2's crash condition states that the disk still "contains" all previous writes, represented by $\langle v,\ vs \rangle$, rather than a specific value that persisted across the crash, chosen out of $\{v\} \cup vs$. This choice of representing the state before the crash rather than after the crash allows the crash condition to be similar to the pre- and postconditions. For example, in Figure 4-2, the state of other blocks just before a crash matches the F predicate, as in the pre- and postconditions. However, describing the state after the crash would require a more complex predicate (e.g., if F contains unsynced disk writes, the state after the crash must choose one of the possible values). Making crash conditions similar to pre- and postconditions is good for proof automation (as we describe in Chapter 5).

Much like other Hoare-logic-based approaches, CHL requires developers to write complete specifications for every procedure, including internal ones (e.g., allocating an object from a free bitmap). This requires stating precise preconditions and postconditions. In CHL, developers must also write crash conditions for every procedure. In practice, we have found that the crash conditions are often simpler to state than the pre- and postconditions. For example, in FSCQ, most crash conditions in layers above the log simply state that there is an active (uncommitted) transaction; only the top-level system-call code begins and commits transactions.

4.3 Logical address spaces

The above example illustrates how CHL can specify predicates about disk contents, but file systems often need to express similar predicates at other levels of abstraction as well. Consider the Unix pwrite system call. Its specification should be similar to disk_write, except that it should describe offsets and values within the file's contents, rather than block numbers and block values on disk. Expressing this specification directly in terms of disk contents is tedious. For example, describing pwrite might require saying that we allocated a new block from the bitmap allocator, grew the inode, perhaps allocated an indirect block, and modified some disk block that happens to correspond to the correct offset within the file. Writing such complex specifications is also error-prone, which can result in significant wasted effort in trying to prove an incorrect specification.

To capture such high-level abstractions in a concise manner, we observe that many of these abstractions deal with *logical address spaces*. For example, the disk is an address space from disk-block numbers to disk-block contents; the inode layer is an address space from inode numbers to inode structures; each file is a logical address space from offsets to data within that file; and a directory is a logical address space from file names to inode numbers. Building on this observation, CHL generalizes the separation logic for reasoning about the disk to similarly reason about higher-level address spaces like files, directories, or the logical disk contents in a logging system.

As an example of logical address spaces, consider the simple procedure in Figure 4-3. atomic_two_write captures the essence of file-system calls that must update two or more blocks. The procedure performs two disk writes inside of a transaction using a write-ahead log, which supplies the log_begin, log_commit, and log_write APIs. If atomic_two_write crashes and the system reboots, the recovery procedure of the logging system log_recover runs. If there is a crash during recovery, then after reboot the recovery procedure runs again. In principle, this may happen several times. Nevertheless, as long as log_recover finishes, the logging system guarantees that either both writes happen or none do, no matter when and how many crashes happen.

To simplify the reasoning, throughout the rest of this chapter, we assume the logging system uses the basic logging protocol described in Section 3.2. That is, the logging system does *not* employ deferred apply, group commit, log checksum or log bypass.

The spec of atomic_two_write, as shown in Figure 4-4, demonstrates how address

```
def atomic_two_write(a1, v1, a2, v2):
    log_begin()
    log_write(a1, v1)
    log_write(a2, v2)
    log_commit()
```

Figure 4-3: Pseudocode of atomic_two_write

spaces help to write a concise specification. Rather than describing how atomic_two_write modifies the on-disk logging data structures, the specification introduces new address spaces, *start_state* and *new_state*, which correspond to the contents of the logical disk provided by the logging system. Logical address spaces allow the developer of the logging system to state a clean specification, which provides the abstraction of a simple, synchronous interface to higher layers in the file system. Developers of higher layers can then largely ignore the details of the underlying asynchronous disk.

SPEC atomic_two_write (a_1, v_1, a_2, v_2)
PRE **disk** $\models$ log_rep(NoTxn, *start_state*)
start_state $\models F' \star a_1 \mapsto \langle v_x,\ vs_x \rangle \star a_2 \mapsto \langle v_y,\ vs_y \rangle$
POST **disk** $\models$ log_rep(NoTxn, *new_state*)
new_state $\models F' \star a_1 \mapsto \langle v_1,\ \emptyset \rangle \star a_2 \mapsto \langle v_2,\ \emptyset \rangle$
CRASH **disk** $\models$ log_intact(*start_state*, *new_state*)

Figure 4-4: Specification for atomic_two_write

Specifically, in the precondition, $a_1 \mapsto \langle v_x,\ vs_x \rangle$ applies to the address space representing the starting contents of the logical disk, and in the postcondition, $a_1 \mapsto \langle v_1,\ \emptyset \rangle$ applies to the new contents of the logical disk. Like the physical disk, these address spaces are partial functions from addresses to values (in this case, mapping 64-bit block numbers to 4 KB block values). Unlike the physical disk, writing to the logical disk through the logging system is synchronous: the new contents of the address being written to will become a single synced value $\langle v_1,\ \emptyset \rangle$, as opposed to $\langle v_1,\ \{v_x\} \cup vs_x \rangle$. This is because the logging system exports a synchronous interface, proven correct on top of the asynchronous interface underneath.

To make this specification precise, we must describe what it means for the transaction's logical disk to have value $\langle v_1,\ \emptyset \rangle$ at address a_1. We do this by connecting the transaction's logical address spaces, *start_state* and *new_state*, to their physical representation on disk. For instance, we specify where the starting state is stored on disk and how the new state is logically constructed (e.g., by applying the log contents to the starting state). We specify this connection using a *representation invariant*; in this example, the representation invariant "log_rep" is a predicate describing the physical disk contents.

log_rep takes logical address spaces as arguments and specifies how those logical address spaces are represented on disk. Several states of the logging system are possible; log_rep(NoTxn, *start_state*) means the disk has no active transaction and is in state *start_state*. In Figure 4-4, the log_rep invariant shows up twice. It is first applied to the *disk* address space, representing the physical disk contents, in the precondition. Here, it relates the starting disk contents to the *start_state* logical address space. log_rep also shows up in the postcondition, where it connects the physical disk state after atomic_two_write returns and the logical address space *new_state*. Syntactically, we use the notation "**las** $\models$ predicate" to say that the logical address space **las** matches a particular predicate; this is used both to apply a representation invariant to an address space (such as log_rep in the *disk* address space) as well as to write other predicates about an address space (such as $a_1 \mapsto \langle v_1, \emptyset \rangle$ in *new_state*).

Representation invariants can be thought of as macros, which boil down to a set of "points-to" relationships. For instance, Figure 4-5 shows part of the log_rep definition for an interesting case, namely an active transaction. It states that, in order for a transaction to be in an ActiveTxn state, the commit block must contain zero, all of the blocks in *start_state* must be on disk, and *cur_state* is the result of replaying the transaction's pending writes (i.e., log entries cached in memory) starting from *start_state*. "replay" is the one part of log_rep that does not boil down to points-to predicates: it is simply a function that takes one logical address space and produces another logical address space (by applying log entries from *pending_writes*). Note that, by using $\emptyset$ as the previous values, log_rep requires that the commit block must have been synced.

$$\begin{aligned} &\mathsf{log\_rep}(\mathsf{ActiveTxn},\ \mathit{start\_state},\ \mathit{cur\_state})\ ::= \\ &\qquad \mathsf{CommitBlock} \rightarrowtail \langle 0,\ \emptyset \rangle\ \star\ (\forall a,\ \mathit{start\_state}(a) = \langle v,\ vs \rangle\ \Rightarrow\ a \rightarrowtail \langle v,\ vs \rangle) \wedge \\ &\qquad \mathit{cur\_state} = \mathsf{replay}(\mathit{start\_state},\ \mathit{pending\_writes}) \end{aligned}$$

Figure 4-5: Part of log_rep representation invariant for the basic logging protocol

The crash condition of atomic_two_write, from Figure 4-4, describes all of the states in which atomic_two_write could crash using log_intact(*start_state*, *new_state*), which stands for all possible log_rep states that recover to transaction states *start_state* or *new_state*. Using log_intact allows us to concisely capture all possible crash states during atomic_two_write, including crashes deep inside any procedure that atomic_two_write might call (e.g., crashes inside log_commit).

4.4 Recovery execution semantics

After the computer crashes and reboots, it often runs a recovery procedure (such as log_recover) before resuming normal operation. Crash conditions and address spaces allow us to specify the possible states in which the computer might crash in the middle of a procedure's execution. But we also need a way to reason about recovery, including crashes during recovery. Hoare logic does not provide a notion that at any point during $\mathbb{C}$'s execution, a recovery procedure may run.

For example, we want to argue that a transaction provides all-or-nothing atomicity: if atomic_two_write crashes prior to invoking log_commit, none of the calls to log_write will be applied; after log_commit returns, all of them will be applied; and if atomic_two_write crashes during log_commit, either all or none of them will take effect. To achieve this property, the logging system must run log_recover after every crash to roll forward any committed transaction, including after crashes during log_recover itself.

The specification of the log_recover procedure is shown in Figure 4-6. It states that, starting from any state matching log_intact(*last_state*, *committed_state*), log_recover will either roll back the transaction to *last_state* or will roll forward a committed transaction to *committed_state*. Furthermore, the specification is *idempotent*, since the crash condition implies the precondition; this will allow for log_recover to crash and restart multiple times.

SPEC	log_recover ()
PRE	**disk** $\models$ log_intact(*last_state*, *committed_state*)
POST	**disk** $\models$ log_rep(NoTxn, *last_state*) $\vee$ log_rep(NoTxn, *committed_state*)
CRASH	**disk** $\models$ log_intact(*last_state*, *committed_state*)

Figure 4-6: Specification of log_recover

To state that log_recover must run after a crash, CHL provides a *recovery execution semantics*. In contrast to CHL's regular execution semantics, which talks about a procedure producing either a failure (accessing an invalid disk block), a crash, or a finished state, the recovery semantics talks about *two* procedures executing (a normal procedure and a recovery procedure) and producing either a failure, a *completed* state (after finishing the normal procedure), or a *recovered* state (after finishing the recovery procedure). This regime models the notion that the normal procedure tries to execute and reach a completed state, but if the system crashes, it starts running the recovery procedure (perhaps multiple times if there are crashes during recovery), which produces a recovered state.

The CHL specification with recovery execution semantics can be noted by:

$$\{P\}\ \mathbb{C}\ \bowtie\ \mathbb{R}\ \{Q\} \tag{4.6}$$

where $\bowtie$ denotes the joint execution of program $\mathbb{C}$ and $\mathbb{R}$. The specification states that, if $\mathbb{C}$ runs under condition P, and the recovery procedure $\mathbb{R}$ is executed whenever the system crashes inside $\mathbb{C}$ or $\mathbb{R}$ itself, then when either $\mathbb{C}$ or $\mathbb{R}$ terminates, condition Q will be established.

Figure 4-7 shows how to extend the atomic_two_write specification to include recovery execution using the $\bowtie$ notation. The postcondition indicates that, if atomic_two_write finishes without crashing, both blocks were updated, and if one or more crashes occurred, with log_recover running after each crash, either both blocks were updated or neither was. The special *status* variable indicates whether the system reached a completed or a recovered state and in this case enables callers of atomic_two_write to conclude that, if atomic_two_write completed without crashes, it updated both blocks (i.e., updating none of the blocks is allowed only if the system crashed and recovered).

$$\begin{array}{ll}
\text{SPEC} & \mathsf{atomic\_two\_write}\ (a_1,\ v_1,\ a_2,\ v_2) \quad \bowtie \quad \mathsf{log\_recover}\ () \\
\text{PRE} & \textbf{disk} \models \mathsf{log\_rep}(\mathsf{NoTxn},\ \mathit{start\_state}) \\
 & \textbf{start\_state} \models F' \ \star\ a_1 \mapsto \langle v_x,\ vs_x \rangle \ \star\ a_2 \mapsto \langle v_y,\ vs_y \rangle \\
\text{POST} & \textbf{disk} \models \mathsf{log\_rep}(\mathsf{NoTxn},\ \mathit{new\_state}) \ \lor \\
 & \qquad\qquad (\mathit{status} = \mathsf{Recovered} \land\ \mathsf{log\_rep}(\mathsf{NoTxn},\ \mathit{start\_state})) \\
 & \textbf{new\_state} \ \models F' \ \star\ a_1 \mapsto \langle v_1,\ \varnothing \rangle \ \star\ a_2 \mapsto \langle v_2,\ \varnothing \rangle
\end{array}$$

Figure 4-7: Specification for atomic_two_write with recovery. The $\bowtie$ operator indicates the combination of a regular procedure and a recovery procedure.

Note that distinguishing the completed and recovered states allows the specification to state stronger properties for completion than recovery. Also note that the recovery-execution version of atomic_two_write's specification does not have a crash condition, because the execution always succeeds, perhaps after running log_recover many times.

In this example, the recovery procedure $\mathbb{R}$ is just log_recover, but the recovery procedure of a system built on top of the logging system may be composed of several recovery procedures. For example, recovery in a file system consists of first reading the superblock from disk to locate the log and then running log_recover.

FIVE

Proving specifications

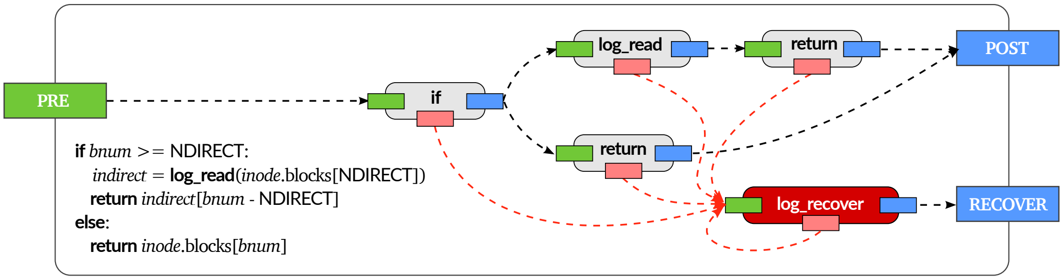
The preceding chapter explains how to write specifications using CHL. This chapter describes how to prove that an implementation meets its specification. A key challenge in the design of CHL was to reduce the proof burden on developers. In developing FSCQ, we often refactored specifications and implementations, and each time we did so we had to redo the corresponding proofs. To reduce the burden of proving, we designed CHL so that it allows for stylized proofs. As a result of this design, several proof steps can be done automatically, as we describe in this chapter.

Even with this automation, a significant amount of manual effort is still required for proving. First, CHL itself must be proven to be sound, which we have done as part of implementing CHL in Coq; developers using CHL need not redo this proof. Second, each application that uses CHL typically requires a significant amount of effort to develop its specifications and proofs, because there are many aspects that cannot be fully automated. We examine the amount of work required to build the FSCQ file system in more detail in Chapter 9.

5.1 Overview

To get some intuition for how CHL can help automate proofs, consider the control flow of the example procedure in Figure 5-1. The outer box corresponds to the top-level specification of a procedure; in this example, it is a procedure that returns the address of the *bnum*$^{\text{th}}$ block from an inode, with recovery-execution semantics. It has a precondition, a postcondition, and a recovery condition.

The arrows correspond to the procedure's control flow, and smaller rounded boxes correspond to procedures that the top-level procedure invokes (e.g., log_read). Each of these procedures has a precondition, a postcondition, and a crash condition. In the figure, after calling the if statement, the procedure can follow two different paths: it may call log_read and then return, or it may immediately return a different value. More complicated procedures may have more complicated control flows, including loops.



```
if bnum >= NDIRECT:
  indirect = log_read(inode.blocks[NDIRECT])
  return indirect[bnum - NDIRECT]
else:
  return inode.blocks[bnum]
```

Figure 5-1: Example control flow of a CHL procedure that looks up the address of a block in an inode, with support for indirect blocks. (The actual code in FSCQ checks for some additional error cases.) Rounded boxes represent the specifications of procedures. The dark red box represents the recovery procedure. Green, blue and pink boxes represent preconditions, postconditions and crash conditions, respectively. Dashed arrows represent control flow as well as logical implication.

The top-level procedure also has a recovery procedure (e.g., log_recover). The recovery procedure has a precondition, postcondition, and recovery condition. The recovery procedure may be invoked at any point after a crash. To capture this, the control flow can jump from the crash condition of a procedure to the recovery procedure. The recovery procedure can itself crash, so there is also an arrow from the recovery procedure's crash condition to its own precondition.

Proving the correctness of the top-level procedure $\mathbb{C}$ entails proving that, if $\mathbb{C}$ is executed and the precondition held before $\mathbb{C}$ started running, either 1) its postcondition holds; or 2) the recovery condition holds after recovery finishes. For the first case, we must show that the precondition of the top-level procedure implies the precondition of the first procedure invoked, that the postcondition of the first procedure called implies the precondition of the next procedure invoked in the control flow, and so on. Similarly for the second case, we must prove that the crash condition of each procedure implies the precondition of the recovery procedure, and so on.

In both cases, the logical implications follow exactly the control flow of the procedure, which allows for a high degree of automation. Our implementation of CHL automatically chains the pre- and postconditions based on the control flow of the procedure. If a precondition is trivially implied by a preceding postcondition in the control flow, then the developer does not have to prove anything. In practice this is often the case, and the developer must prove only the representation invariants (e.g., log_rep and log_intact). The rest of this chapter describes this automation in more detail, while explicitly noting what must be proven by hand by developers. The basic strategy is inspired by the Bedrock system [13] but extends the approach to handle crashes and address spaces.

5.2 Proving without recovery

The process of proving a CHL specification is to repeatedly apply inference rules on existing specification to deduce the target specification. Similar to Hoare logic and separation logic (Section 3.3), CHL supports the following *composition rule*, *consequence rule* and *frame rule*:

$$\frac{\{P_1\}\ \mathbb{C}_1\ \{P_2\}\{R_1\} \qquad \{P_2\}\ \mathbb{C}_2\ \{P_3\}\{R_2\}}{\{P_1\}\ \mathbb{C}_1; \mathbb{C}_2\ \{P_3\}\{R_1 \vee R_2\}} \tag{5.1}$$

$$\frac{P' \Rightarrow P \qquad \{P\}\ \mathbb{C}\ \{Q\}\{R\} \qquad Q \Rightarrow Q' \qquad R \Rightarrow R'}{\{P'\}\ \mathbb{C}\ \{Q'\}\{R'\}} \tag{5.2}$$

$$\frac{\{P\}\ \mathbb{C}\ \{Q\}\{R\}}{\{P \star S\}\ \mathbb{C}\ \{Q \star S\}\{R \star S\}} \tag{5.3}$$

CHL's proof automation can automatically apply the above rules whenever possible. In general, the proof strategy consists of two steps:

Phase 1: Procedure steps. The first phase of CHL's proof strategy is to use the composition rule (eq. (5.1)), and turn the theorem about $\mathbb{C}$'s specification into a series of *proof obligations* that will be proven in the next phase. Specifically, CHL considers every step in $\mathbb{C}$ (e.g., a procedure call) and reasons about what the state of the system is before and after that step. CHL assumes that each step already has a proven specification. The base primitives (e.g., disk_read and disk_write) of CHL have proven specifications provided by the implementation of CHL.

CHL starts by assuming that the initial state matches the precondition, and, for every step in $\mathbb{C}$, generates two proof obligations: (1) that the current condition (either $\mathbb{C}$'s precondition or the previous step's postcondition) implies the step's precondition, and (2) that the step's crash condition implies $\mathbb{C}$'s crash condition. At the end of $\mathbb{C}$, CHL generates a final proof obligation that the final condition implies $\mathbb{C}$'s postcondition.

SPEC	log_begin ()
PRE	**disk** $\models$ log_rep(NoTxn, *start_state*)
POST	**disk** $\models$ log_rep(ActiveTxn, *start_state*, *start_state*)
CRASH	**disk** $\models$ log_rep(NoTxn, *start_state*)

Figure 5-2: Specification for log_begin

For example, consider the atomic_two_write procedure from Figure 4-3, whose specification is shown in Figure 4-4. As the first step, CHL considers the call to log_begin and, using the specification shown in Figure 5-2, generates two proof obligations: that atomic_two_write's precondition matches the precondition of log_begin, and that log_begin's crash condition implies atomic_two_write's crash condition.

SPEC log_write (a, v)
PRE **disk** $\models$ log_rep(ActiveTxn, *start_state*, *old_state*)
old_state $\models F \star a \mapsto \langle v_0,\ vs_0 \rangle$
POST **disk** $\models$ log_rep(ActiveTxn, *start_state*, *new_state*)
new_state $\models F \star a \mapsto \langle v,\ \varnothing \rangle$
CRASH **disk** $\models$ log_rep(ActiveTxn, *start_state*, *any_state*)

Figure 5-3: Specification for log_write

When specifications involve multiple address spaces, CHL recursively matches up the address spaces starting from the built-in *disk* address space. For instance, the next step in atomic_two_write is the call to log_write, whose specification appears in Figure 5-3. By matching up the *disk* address spaces in log_begin's postcondition and log_write's precondition, CHL concludes that the address space called *start_state* in atomic_two_write is the same as the *old_state* address space in log_write. CHL then generates another proof obligation that the predicate for *start_state* in atomic_two_write implies the predicate for *old_state* in log_write.

Phase 2: Predicate implications. Some obligations generated in phase 1 are trivial, such as that the precondition of atomic_two_write implies the precondition of log_begin; since the two are identical, CHL immediately proves the implication between them using the consequence rule (eq. (5.2)) and first-order logic.

For more complicated cases, CHL relies on separation logic's frame rule (eq. (5.3)) to prove the obligations and to help carry information from precondition to postcondition. Continuing with our example, consider the proof obligation generated at atomic_two_write's first call to log_write, which requires us to prove that $F' \star a_1 \mapsto \langle v_x,\ vs_x \rangle \star a_2 \mapsto \langle v_y,\ vs_y \rangle$ implies $F \star a \mapsto \langle v_0,\ vs_0 \rangle$. Because separating conjunction ($\star$) applies only to disjoint predicates, CHL matches up $a_1 \mapsto \langle v_x,\ vs_x \rangle$ with $a \mapsto \langle v_0,\ vs_0 \rangle$ (thereby setting v_0 to v_x) and "cancels out" these terms from both sides of the implication obligation. CHL then sets the arbitrary predicate F from log_write's precondition to $F' \star a_2 \mapsto \langle v_y,\ vs_y \rangle$. This has two effects: first, it proves this particular obligation; and second, it carries over information about $a_2 \mapsto \langle v_y,\ vs_y \rangle$ into subsequent proof obligations that mention F from log_write's postcondition (such as atomic_two_write's next call to log_write).

Some implication obligations cannot be proven by CHL automatically and require developer input. This usually occurs when the developer is working with multiple levels of abstraction, where one predicate mentions a higher-level representation invariant (e.g., a directory represented by a function from file names to inode numbers) but the other predicate talks about lower-level state (e.g., a directory represented by a set of directory entries in a file).

5.3 Proving recovery specifications

So far, we described how CHL proves specifications about procedures without recovery. Proofs of specifications that involve a recovery procedure, such as Figure 4-7, are also automated in CHL: if the recovery procedure is idempotent (i.e., its crash condition implies its precondition), CHL can automatically prove the specification of procedure $\mathbb{C}$ with recovery based on the specification of $\mathbb{C}$ without recovery. The rule for proving a recovery specification is called the *recovery rule*:

$$\frac{\{P\}\ \mathbb{C}\ \{Q\}\{R\} \qquad \{R\}\ \mathbb{R}\ \{S\}\{R\}}{\{P\}\ \mathbb{C}\ \bowtie\ \mathbb{R}\ \{Q \vee S\}} \tag{5.4}$$

The recovery rule states that, if the recovery procedure $\mathbb{R}$ handles both program $\mathbb{C}$'s crash states and its own crash states, and if the outcomes of $\mathbb{C}$ and $\mathbb{R}$ are Q and S, respectively, then running $\mathbb{C}$ in combination with $\mathbb{R}$ will result in condition $Q \ \vee \ S$ when they both terminate.

For instance, since the log_recover procedure is idempotent (see Figure 4-6), CHL can automatically prove the specification shown in Figure 4-7 based on the specification from Figure 4-4.

Six

Certifying FSCQ's logging system

The previous chapters provide examples of specifications for a basic logging protocol, written in CHL. This chapter explores how to build and certify FscqLog, an I/O-efficient logging system that implements performance-critical optimizations such as deferred apply, group commit, log checksum and log bypass. These techniques, as we explained in Section 3.2, reduce the number of writes and write barriers per transaction, while still guaranteeing crash safety. These optimizations are standard to other file systems but have never been formally specified or verified until now. The FscqLog implements these optimizations internally, while providing the same synchronous disk abstraction as in previous chapters.

6.1 Overview

Verifying a single optimization is challenging enough because most system optimizations are based on a programmer's assumptions, which are rarely formalized. Verifying many together is even more challenging because of the number of simultaneously moving parts. Each additional optimization requires not only a proof that the overall logging specification still holds, but also a proof that all previous assumptions still hold. For example, group commit assumes that metadata operations can be deferred as long as their orders are preserved, and log bypass further allows deferring and re-ordering non-metadata updates; combining the two to support a general deferred-write optimization poses a new challenge: is it safe to re-order non-metadata updates against metadata updates?

The key idea behind verifying a system with many optimizations is one familiar from building the unverified equivalent: modularity. FscqLog is composed of four logical layers: LogAPI, GroupCommit, Applier, and DiskLog, shown in Figure 6-1. Each layer has a formal specification, along with a proof that its implementation meets its specification. This modular design makes FscqLog amenable to formal verification because each

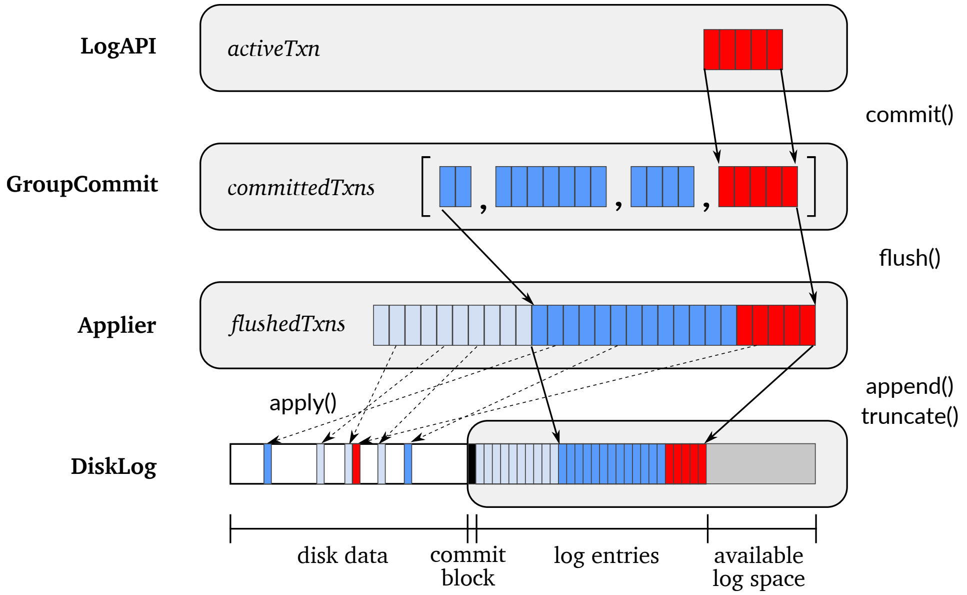


Figure 6-1: Illustration of FscqLog layers and the timeline of a transaction. Transactional writes are added to an *activeTxn* map in the LogAPI layer. When the application calls commit, *activeTxn* is buffered in a list of pending transactions in GroupCommit. When flush is called on GroupCommit, all pending transactions are appended to disk together as a single transaction in the DiskLog layer. At this point, the transactions are durable, and Applier can lazily apply and truncate the log records.

layer only tackles a simple task or a single optimization. It also allows changing the implementation of a layer without affecting the proofs in other layers.

LogAPI. The uppermost layer, LogAPI, exposes an interface with a single active transaction and allows the higher-level code (i.e., the file system) to read and write disk blocks. Writing blocks builds up an in-memory transaction, which is passed to the GroupCommit layer once the higher-level code invokes log_commit. LogAPI exposes the size of the transaction in its specification and guarantees that transactions below a certain size will be able to commit. This is necessary for proving that some system calls that potentially update many disk blocks (e.g., unlink might change all block-bitmap blocks when freeing a large file) do not fail given sufficient log space.

GroupCommit. GroupCommit accepts committed transactions from LogAPI and buffers them in an in-memory transaction list. GroupCommit exposes a flush function, which combines all buffered transaction into a single transaction and then flushes it to the on-disk log. flush allows the file system to implement the fsync system call by flushing all metadata changes from GroupCommit to disk. GroupCommit's specification also allows it to flush transactions to disk on its own at any time. GroupCommit remembers bound-aries of transactions it receives. In case the merged transaction exceeds the maximum transaction size, GroupCommit falls back to committing individual transactions in turn.

Applier. Applier manages the data part of the disk (i.e., everything but the log) by applying the log entries to the disk when the on-disk log fills up. By employing deferred apply, Applier is able to absorb repeated writes to the same address in multiple transactions. To do this, Applier keeps a copy of all flushed entries in memory when forwarding flushing requests from GroupCommit to DiskLog.

DiskLog. Finally, DiskLog lays out the log on disk. It provides three functions: append, truncate, and recover (which returns the on-disk log contents). It takes care of packing and unpacking the commit block and log entries on disk. DiskLog also exposes the size of the on-disk log to guarantee that transactions of a certain size will fit and thus be able to commit. DiskLog uses log checksum to ensure that append and truncate are crash-safe while using just one disk sync (to ensure that the entire transaction is flushed to disk, rather than to order writes within the transaction). DiskLog's recover function also uses checksums to determine whether the last call to append fully made it to disk or not, and returns the corresponding list of on-disk log entries to the caller.

Each layer (except for DiskLog) also caches copies of transaction updates in an in-memory map to serve reads efficiently. As shown in Figure 6-1, LogAPI buffers the file system's single active transaction (*activeTxn*). GroupCommit merges each committed transaction that it receives into a single in-memory transaction (*committedTxns*), which is cleared at every flush. Similarly, Applier collapses all flushed transactions into an in-memory map that is cleared every time the log is applied and truncated (*flushedTxns*). When an application calls read, the log tries to find the requested address in each of these layers, from highest to lowest, before reading from disk.

6.2 Representation invariants

As described in Section 4.3, FSCQLOG's specifications use representation invariant to relate the abstract state of a higher layer to the its lower-layer state. For example, DiskLog defines disklog_rep, which describes how logical log entries (as a list of block address and value pairs) are laid out on the log region of the disk. Applier, in turn, includes disklog_rep as part of its own applier_rep, and combines it with other points-to facts that relate the logical log entries to the data region of the disk. In its easiest form, where there is no ongoing log-apply or log-flush, the definition of applier_rep is shown in Figure 6-2.

applier_rep states that the physical disk is divided into two disjoint regions: the log region and the data region. disklog_rep describes the state of the log region, which

$$\begin{aligned}
&\mathsf{applier\_rep}\,(\,navail,\ flushed\_disk)\ ::= \\
&\qquad \mathsf{disklog\_rep}(\mathsf{Synced},\ navail,\ flushedTxns)\ \star \\
&\qquad (\forall a,\ unapplied\_disk(a) = \langle v,\ vs\rangle\ \Rightarrow\ a \rightarrowtail \langle v,\ vs\rangle) \land \\
&\qquad flushed\_disk = \mathsf{replay}\,(unapplied\_disk,\ flushedTxns)
\end{aligned}$$

Figure 6-2: Part of Applier's representation invariant

contains all flushed but not-yet-applied transactions, represented by *flushed_Txns*; and *unapplied_disk* is the on-disk state of the data region. By replaying *flushed_Txns* on *unapplied_disk*, applier_rep exports a new logical disk *flushed_disk* in its argument, which combines the on-disk data region with all flushed transactions to present a view of the persisted disk state. The other argument of applier_rep, *navail*, is the number of available log entries exported by DiskLog; it is useful for higher layers to reason about whether a commit or a flush would succeed.

Similarly, as shown in Figure 6-3, GroupCommit's representation invariant group_rep uses applier_rep to describe the disk state which currently persists on disk. On top of it, group_rep describes a sequence of logical disks, namely *disk_seq*, each representing the end state of a committed but not-yet-flushed transaction. The definition in Figure 6-3 assumes the first disk (i.e., $disk\_seq[0]$) is persisted on disk, and each subsequent disk in *disk_seq* is derived from its predecessor by applying the corresponding committed transaction from *committedTxns*. The abstraction exported by group_rep is called a *disk sequence*. In Chapter 7 we further justify the choice of the disk sequence abstraction and investigate how it helps to write a file system's specifications.

$$\begin{aligned}
&\mathsf{group\_rep}\,(\,disk\_seq)\ ::= \\
&\qquad \mathsf{applier\_rep}(navail,\ disk\_seq[0]) \land \\
&\qquad (\forall i,\ 0 < i < \|\,disk\_seq\,\| \Rightarrow \\
&\qquad\quad \|\,committedTxns[i-1]\,\| \le \mathsf{MaxLogLen} \land \\
&\qquad\quad disk\_seq[i] = \mathsf{replay}\,(disk\_seq[i-1],\ committedTxns[i-1]))
\end{aligned}$$

Figure 6-3: Part of GroupCommit's representation invariant

Figure 6-4 shows the top-level representation invariant log_rep, whose signature looks almost identical to the one shown in Figure 4-5, except that it now uses a disk sequence, as opposed to a single logical disk, to describe the transaction's starting state. "*disk_seq*.latest" denotes the last disk in the disk sequence; applying the active transaction's log entries (*activeTxn*) to the latest disk produces *cur_state*, the logical disk that represents the transaction's current view. If there is no active transaction, log_rep requires *activeTxn* to be empty.

Note that most of the above representation invariants have more sophisticated forms which show up only during crash and recovery. For simplicity we do not show them here but will explain them as we encounter them.

$$
\begin{aligned}
&\text{log\_rep}\,(\text{ActiveTxn},\ \mathit{disk\_seq},\ \mathit{cur\_state})\ ::= \\
&\qquad \text{group\_rep}(\mathit{disk\_seq}) \wedge \\
&\qquad \mathit{cur\_state} = \text{replay}\,(\mathit{disk\_seq}.\text{latest},\ \mathit{activeTxn}) \\
&\text{log\_rep}\,(\text{NoTxn},\ \mathit{disk\_seq})\ ::= \\
&\qquad \text{group\_rep}(\mathit{disk\_seq})\ \wedge\ \mathit{activeTxn} = \varnothing
\end{aligned}
$$

Figure 6-4: Part of LogAPI's representation invariant

6.3 Logging-system specifications

Given these representation invariants, we now demonstrate how to use them to write FscqLog's internal and external specifications. We consider an interesting scenario where the user of FscqLog (i.e., the file system) uses log_commit to commit a transaction and subsequently calls log_flush to make sure the change persists. The call graph used in this section is shown in Figure 6-5.

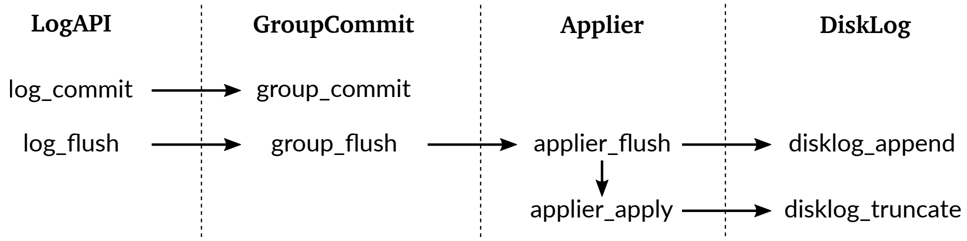


Figure 6-5: Call graph for log_commit and log_flush

Figure 6-6 shows the specification of log_commit. The spec says that log_commit transits from "active transaction" state to "no transaction" state, and it can either succeed (return true) or fail (return false). If it succeeds, the logical disk for the active transaction *cur_disk* is appended to the disk sequence *disk_seq*; if it fails, the transaction is aborted, and *disk_seq* will remain unchanged. In addition, the specification also says that if commit fails, it must be the case that the size of the transaction (total number of log entries in *activeTxn*) exceeds the logging system's limit, denoted by constant MaxLogLen. This error condition allows the caller to prove that certain operations (such as unlink) will always succeed (by showing that $\|\mathit{activeTxn}\| \leq \text{MaxLogLen}$, thus proving that the failure case in the postcondition cannot happen).

SPEC	log_commit ()
PRE	**disk** $\models$ log_rep(ActiveTxn, *disk_seq*, *cur_disk*)
POST	**disk** $\models$ (*ret* = **true** $\wedge$ log_rep(NoTxn, *disk_seq* ++ {*cur_disk*})) $\vee$ (*ret* = **false** $\wedge$ log_rep(NoTxn, *disk_seq*) $\wedge$ $\|\mathit{activeTxn}\| > \text{MaxLogLen}$)
CRASH	**disk** $\models$ log_rep(NoTxn, *disk_seq*)

Figure 6-6: Specification for LogAPI's log_commit

Internally, log_commit calls GroupCommit's group_commit(*activeTxn*), which buffers the transaction in memory. The specification of group_commit is shown in Figure 6-7. This specification looks similar to Figure 6-6, except that it expands *cur_disk*, using the fact that directly derives from LogAPI's representation invariant (Figure 6-4). As group_commit only changes the in-memory state, its crash condition is the same as its precondition.

SPEC	group_commit (*ents*)
PRE	**disk** $\models$ group_rep(*disk_seq*)
POST	**disk** $\models$ (*ret* = **true** $\wedge$ group_rep(*disk_seq* ++ {replay (*disk_seq*.latest, *ents*)} $\vee$ (*ret* = **false** $\wedge$ group_rep(*disk_seq*) $\wedge$ $\|\mathit{ents}\| > \text{MaxLogLen}$)
CRASH	**disk** $\models$ group_rep(*disk_seq*)

Figure 6-7: Specification for GroupCommit's group_commit

Next, the user calls the top-level log_flush, whose specification is shown in Figure 6-8. The specification simply says that, after log_flush returns, the disk sequence contains only the latest disk from before log_flush. This latest disk reflects all of the previously committed transactions.

One interesting aspect of log_flush's spec is the crash condition: "would_recover_any (*disk_seq*)" says that if the system were to crash, the state on disk after the crash is the state corresponding to one of the disks in *disk_seq*. This is because log_flush internally calls GroupCommit's flush procedure group_flush, which in turn calls Applier's applier_flush, with the merged transaction as the argument (i.e., applier_flush(merge(*committedTxns*))). In this case, the logging system will recover into either the first or the last disk in *disk_seq*. It is also possible that GroupCommit cannot merge the transactions (because the resulting transaction is too large to fit into the log) and falls back to flush each transactions in turn using applier_flush. If this is the case, the logging system can recover into any of the disks in *disk_seq*. Both cases are captured by would_recover_any.

SPEC log_flush ()
PRE **disk** $\models$ log_rep(NoTxn, *disk_seq*)
POST **disk** $\models$ log_rep(NoTxn, {*disk_seq*.latest})
CRASH **disk** $\models$ would_recover_any(*disk_seq*)

Figure 6-8: Specification for LogAPI's log_flush

The specification of applier_flush is shown in Figure 6-9, which also contains a success case and a failure case. However, the GroupCommit layer can conclude that the failure case will never happen. This is because (1) the code of group_flush first checks if the merged transaction is bigger than MaxLogLen and falls back to flushing individual transactions in that case; and (2) group_commit will reject a transaction whose size is bigger than MaxLogLen, so that every buffered transaction is within the size limit. The second constraint is encoded in the representation invariant shown in Figure 6-3.

The crash condition of applier_flush is a "would_recover_either" predicate, which says that the procedure will recover into either before or after the given transaction—the usual behavior of basic write-ahead logging. The "would_recover_any" predicate we have seen in Figure 6-8 is defined using would_recover_either by choosing pairwise disks from *disk_seq*.

SPEC applier_flush (*ents*)
PRE **disk** $\models$ applier_rep(*navail*, *flushed_disk*)
POST **disk** $\models$ ($ret =$ **true** $\wedge$ applier_rep($navail'$, replay(*flushed_disk*, *ents*))) $\vee$
($ret =$ **false** $\wedge$ applier_rep(*navail*, *flushed_disk*) $\wedge$
$\|ents\| >$ MaxLogLen)
CRASH **disk** $\models$ would_recover_either(*flushed_disk*, replay(*flushed_disk*, *ents*))

Figure 6-9: Specification for Applier's applier_flush

Inside applier_flush, if it sees the available log space *navail* is too small to fit the passed-in transaction (*ents*), the code will first invoke applier_apply to make room for the new transaction. applier_apply applies and truncates the on-disk log. Its specification is shown in Figure 6-10. The postcondition of applier_apply resets the available log space to its maximum value. Other than that, the specification exhibits a no-op-like behavior such that Applier is free to invoke it anytime.

Also note that the crash condition in Figure 6-10 is different from its precondition. This is because applier_apply will call disklog_truncate to clear the log, and disklog_truncate might leave the log in an *unsynced* state, in which DiskLog just overwrites the commit block but has not yet issued a write barrier to persist the change. The unsynced state is not captured by the regular form of applier_rep shown in Figure 6-2. Nevertheless, this does not invalidate the fact that applying the log is an idempotent operation and will always recover to *flushed_disk* as in the precondition. The "would_recover_before" predicate exactly captures this property.

SPEC $\quad$ applier_apply ()
PRE $\quad \textbf{disk} \models \text{applier\_rep}(\mathit{navail},\ \mathit{flushed\_disk})$
POST $\quad \textbf{disk} \models \text{applier\_rep}(\text{MaxLogLen},\ \mathit{flushed\_disk})$
CRASH $\quad \textbf{disk} \models \text{would\_recover\_before}(\mathit{flushed\_disk})$

Figure 6-10: Specification for Applier's applier_apply

Finally, applier_flush extends the log using DiskLog's disklog_append, whose specification is illustrated in Figure 6-11. Like before, FSCQLOG's implementation guarantees that the failure case in the postcondition cannot occur, because no transaction larger than MaxLogLen will be appended, and applier_apply will reset the available log space back to MaxLogLen when necessary.

The two branches in Figure 6-11's crash condition corresponds to the two cases defined by would_recover_either. In particular, the "Extended" case also describes an unsynced log state: If the system crashes while updating the commit block, it will recover to a state either before or after *new_ents* is appended. The evolution of crash conditions in each logging layer further demonstrates that representation invariants can help in hiding lower-level details from the higher layers.

One subtlety of disklog_append's specification is that it includes a frame predicate F_{disk}. This is because disklog_rep only describes the log region and does not cover the entire *disk* address space. This frame predicate allows DiskLog's upper layer (Applier) to carry its own predicate about the data region of the disk—another example of using separation logic to achieve modularity.

SPEC $\quad$ disklog_append (*new_ents*)
PRE $\quad \textbf{disk} \models F_{disk} \star \text{disklog\_rep}(\text{Synced},\ \mathit{navail},\ \mathit{old\_ents})$
POST $\quad \textbf{disk} \models (\mathit{ret} = \textbf{true} \ \land\ F_{disk} \star \text{disklog\_rep}(\text{Synced},\ \mathit{navail}',\ \mathit{old\_ents} \mathbin{+\!\!+} \mathit{new\_ents})$
$\qquad\qquad (\mathit{ret} = \textbf{false} \ \land\ F_{disk} \star \text{disklog\_rep}(\text{Synced},\ \mathit{navail},\ \mathit{old\_ents}) \land$
$\qquad\qquad\qquad \| \mathit{new\_ents} \| > \mathit{navail})$
CRASH $\quad \textbf{disk} \models F_{disk} \star \text{disklog\_rep}(\text{Synced},\ \mathit{navail},\ \mathit{old\_ents}) \ \lor$
$\qquad\qquad F_{disk} \star \text{disklog\_rep}(\text{Extended},\ \mathit{old\_ents},\ \mathit{new\_ents})$

Figure 6-11: Specification for DiskLog's disklog_append

6.4 Logging with checksums

The previous section mostly introduced specifications above the DiskLog layer that implement deferred apply and group commit. This section describes how to implement and formalize the log-checksum optimization mentioned in Section 3.2. More details about the design of FSCQ's log-checksum optimization can be found in [68].

6.4.1 On-disk layout and protocol

DiskLog is responsible for managing on-disk state of transactions. The on-disk log that DiskLog maintains internally is separated into three regions: the header (i.e., the commit block), descriptor, and data regions, shown in Figure 6-12. The header stores the number and checksum of valid blocks in the descriptor and data regions. To ensure durability after a crash, the header also stores the number of blocks at the time of the previous flush. The descriptor region stores disk-block addresses corresponding to block value updates, which are stored in the data region in the same order. Disk-block addresses from a single append call can be packed into a single descriptor block. DiskLog's specifications show that the packing and unpacking are sound.

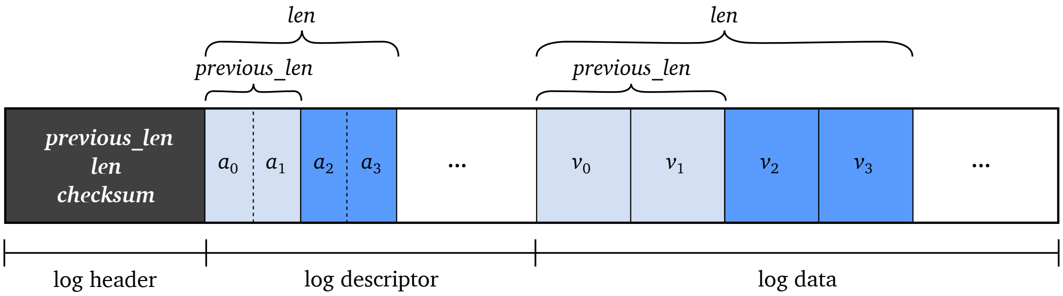


Figure 6-12: On-disk layout of FSCQLOG

To truncate the log, DiskLog simply sets the length equal to zero in the log header. To append to the log, DiskLog writes log entries and the log header together. To do this, DiskLog first writes each entry in the transaction to the descriptor and data regions. Then, DiskLog updates the *checksum*, *previous_len*, and *len* fields in the log header. The checksum is computed by hashing the stored checksum with the newly appended descriptor and data blocks. For both appending and truncating, DiskLog also stores the old length of the log in *previous_len* field. During recovery, DiskLog first tries to read the log from the disk according to the length stored in the *len* field, compute the checksum, and check it against the value stored in the *checksum* field. If they do not match, DiskLog falls back to use the length stored in the *previous_len* field, computes the checksum again, and updates the *checksum* and *len* fields to reflect the corrected state. The pseudocode for the DiskLog protocol is shown in Figure 6-13.

```
# Called by Applier layer after applying log to disk.
def disklog_truncate(txn):
    header = disk_read(CommitBlock)
    header.previous_len = header.len
    header.len = 0
    disk_write(CommitBlock, header)
    disk_sync()

# Called by Applier layer, which guarantees that there's enough space.
def disklog_append(txn):
    header = disk_read(CommitBlock)
    write_packed_addresses(LogDescStart, header.len, txn)
    pos = LogDataStart + header.len
    for (a, v) in txn.iteritems():
        disk_write(pos, v)
        header.checksum = hash(header.checksum || a || v)
        pos += 1
    header.previous_len = header.len
    header.len = header.len + len(txn)
    disk_write(CommitBlock, header)
    disk_sync()

def disklog_readlog(nr):
    checksum = hash(0)
    log = []
    for i in range(0, nr):
        a = read_packed_address(LogDescStart, i)
        v = disk_read(LogDataStart + i)
        checksum = hash(checksum || a || v)
        log.append((a, v))
    return (checksum, log)

def disklog_recover():
    header = disk_read(CommitBlock)
    (checksum, log) = disklog_readlog(header.len)
    if checksum != header.checksum:
        (checksum, log) = disklog_readlog(header.previous_len)
        header.checksum = checksum
        header.len = header.previous_len
        disk_write(CommitBlock, header)
        disk_sync()
    return log
```

Figure 6-13: Pseudocode of DiskLog layer

6.4.2 Formalizing checksums

A challenge in formalizing and verifying the checksum-based protocol lies in the probabilistic guarantee that it offers. In practice, using a strong collision-resistant hash function (such as SHA-256) ensures that the probability of a collision is negligible. Although developers assume that there are no hash collisions in practice, formalizing this assumption is difficult. Theoretically speaking, any hash function (including a collision-resistant function like SHA-256) has collisions, and as a result, two different sets of log entries may have the same checksum. Consequently, stating an axiom that a hash function has no collisions is unsound and is equivalent to assuming that true is false. On the other hand, stating an axiom that a hash function is collision-resistant requires reasoning about probabilities and the computational power of some hypothetical adversary that is attempting to find hash collisions.

An ideal hash model should allows FSCQ to state specifications in a natural way—the way that a file system developer might assume—by completely ignoring the possibility of hash collisions. Otherwise, All proofs would have to deal with probabilistic preamble like "with high probability, unless there is a hash collision, ...", and all specification would be more complex (e.g., after recovery, if there weren't hash collisions during past crashes, one will have the correct data with high probability).

Approach. FSCQLOG has a solution that is both sound and avoids reasoning about the probability of hash collisions. The key idea is to treat hash collisions as function non-termination in the formal semantics of execution. Recall from Section 4.1 that any procedure in CHL is composed of sequences of basic opcodes, such as disk_read, disk_write, etc. CHL provides a formal semantics for how each of these opcodes should execute; e.g., a disk read returns the last written value, and a crash non-deterministically chooses some set of outstanding writes to apply.

FSCQLOG introduces a new opcode to CHL, called hash, which computes the hash value of its input. CHL's formal semantics keep track of all inputs ever hashed and their corresponding hash values. If hash is presented with an input that hashes to the same result as an earlier, different input, then the hash opcode enters an infinite loop and never returns.

This formalization achieves our goals. First, it allows FSCQLOG to conclude that, if hash returns the same hash value twice, the inputs must have been equal (because otherwise hash would not have returned), without reasoning about probabilities. This allows us to write specifications about entire file-system operations, such as rename, saying that *if the operation returns*, then a transaction must have been committed. Second, this formalization is sound, because it does not prohibit the possibility of hash collisions, and instead, explicitly takes them into account (by entering an infinite loop on a collision).

At runtime, of course, the hash opcode is implemented using a collision-resistant hash function (SHA-256 in our case). This hash function does *not* enter an infinite loop when presented with a colliding input and, consequently, can differ from the formal semantics of hash. However, since we know that our hash function is collision-resistant, we know that the possibility of finding a collision is negligible, and thus the possibility that the real execution semantics will differ from the formal one is also negligible. Consequently, using a collision-resistant function for hash at runtime allows us to capture the standard assumption made by developers (that hash collisions do not happen).

Crash safety. An additional challenge that arises in FSCQLOG compared to earlier work on modeling hash collisions [6] is that the computer can crash at any point. This means that, after a crash, the list of inputs ever hashed can be different from the list of hash inputs in either the pre- or the post-condition of a procedure. However, the file-system recovery code must still be able to reason about the list of hash inputs after a crash, in order to prove that it recovers the contents of the on-disk log. FSCQLOG's solution is to prove that the list of hash inputs after a crash is a superset of the hash inputs from a procedure's precondition, which allows FSCQLOG's write-ahead log to prove its correctness.

6.5 Log bypass

Writes that bypass the log still interact with FSCQ's write-ahead log. If the file system issues a log-bypass write to block *b*, and there is an un-applied transaction that modified block *b*, it is important that this transaction does not later overwrite *b*'s contents. Thus, log-bypass writes in FSCQ go through the log abstraction (even though they are not written to the write-ahead log).

The dwrite procedure, exposed by LogAPI, GroupCommit, and Applier, checks if there was a previous logged write to the same address as the log-bypass write. In LogAPI, previous logged writes to the same address are discarded (since they have not yet committed). In GroupCommit, if there are any in-memory transaction writing to the same address, all of them are flushed to disk. In Applier, if the address appears in the on-disk log, the log is applied, so that a later log apply does not overwrite the block modified through log bypass.

Another approach could have been to discard previously committed writes to any block modified via log bypass, even in the lower layers (GroupCommit and Applier). However, this approach leads to the same problem that ext4 experienced, where new file blocks can contain data from previously deleted files after a crash [42]. By flushing previously committed writes, FSCQLOG avoids this problem.

The specification of FscqLog's top-level dwrite is shown in Figure 6-14. It differs from the specification of logged write (see Figure 5-3) in three ways. First, log_dwrite writes directly to the disk; therefore, it changes not only the state of the current logical disk (*old_state*), but also all disks from the disk sequence of the transaction's starting state (*old_disk_seq*, as they all derive from the underlying disk state). Second, much like the specification for writing to the physical disk, log_dwrite exposes an asynchronous interface, leaving the updated block in an unsynced state (i.e., the block's value-set contains more than one value). Finally, the crash condition of log_dwrite says that it could recover into any disk from either the original or the updated disk sequence. This is because log_dwrite internally might invoke log_flush to flush buffered transactions.

SPEC log_dwrite (a, v)
PRE **disk** $\models$ log_rep(ActiveTxn, *old_disk_seq*, *old_state*)
old_state $\models F \star a \mapsto \langle v_0,\ vs_0 \rangle$
$\forall$**i, old_disk_seq[i]** $\models F_i \star a \mapsto \langle v_i,\ vs_i \rangle$
POST **disk** $\models$ log_rep(ActiveTxn, *new_disk_seq*, *new_state*)
new_state $\models F \star a \mapsto \langle v,\ \{v_0\} \cup vs_0 \rangle$
$\forall$**i, new_disk_seq[i]** $\models F_i \star a \mapsto \langle v,\ \{v_i\} \cup vs_i \rangle$
CRASH **disk** $\models$ would_recover_any(*old_disk_seq*) $\vee$
would_recover_any(*new_disk_seq*)

Figure 6-14: Specification for LogAPI's log_dwrite

The caller of log_dwrite is responsible for syncing the updated block at the appropriate time. If the file system is built on top of a write-through cache, where writes immediately go to the physical disk, the caller can simply use the disk write barrier (i.e., disk_sync) to persist the change. However, FSCQ is built on top of a write-back cache (see Chapter 8), where writes are buffered in memory until the updated block is evicted from the cache.

SPEC log_dsync (a)
PRE **disk** $\models$ log_rep(ActiveTxn, *old_disk_seq*, *old_state*)
old_state $\models F \star a \mapsto \langle v_0,\ vs_0 \rangle$
$\forall$**i, old_disk_seq[i]** $\models F_i \star a \mapsto \langle v_i,\ vs_i \rangle$
POST **disk** $\models$ log_rep(ActiveTxn, *new_disk_seq*, *new_state*)
new_state $\models F \star a \mapsto \langle v_0,\ \varnothing \rangle$
$\forall$**i, new_disk_seq[i]** $\models F_i \star a \mapsto \langle v_i,\ \varnothing \rangle$
CRASH **disk** $\models$ log_rep(ActiveTxn, *old_disk_seq*, *old_state*)

Figure 6-15: Specification for LogAPI's log_dsync

To make sure that a block updated through log_dwrite is persisted on disk, the caller must first evict the block from the buffer cache and then sync the physical disk. FscqLog offers a log_dsync procedure that takes care of block syncing in the presence of deferred writes. The specification of log_dsync is shown in Figure 6-15, which is very similar to log_dwrite's specification but has a simpler crash condition. To improve performance, FSCQLOG also offers a few variants of log_dsync that allow a caller to evict a list of blocks from the buffer cache but only issue a single disk sync at the end. FSCQ uses log_dsync and its variants to implement fsync and fdatasync.

Because log-bypass writes interact subtly with logged writes, this poses challenges at the file-system level, which uses logged writes to update the metadata and log-bypass writes to change a file's data. For example, to guarantee crash safety, bypassing the log for file data requires that disk blocks are not reused until the log is flushed to disk. We discuss how to address this challenge in the next chapter.

SEVEN

Specification for deferred writes

Recall that deferred-write optimization buffers recent file-system updates in memory (Section 3.2) and allows the application to force file-system state to disk using fdatasync and fsync. Deferred writes lead to many more possible crash states: when a computer crashes, the state of the disk could reflect the changes of any system calls since the last fsync. Because applications must use fsync to achieve their own crash safety, we need a way of formalizing the deferred-write optimization at the file-system API level.

This chapter introduce a *metadata-prefix specification* that captures the properties of fsync and fdatasync, based on Linux ext4's behavior. We introduce two ideas to help in this formalization: *disk sequences* enable the specification to describe all possible disks states evolved from a sequence of deferred metadata operations, and *disk relations* allow the specification to describe what invariants must hold across all disks in the disk sequence, if we mix log-bypass writes with metadata operations.

7.1 Example application pattern

Achieving high performance and crash safety in a file-system application requires careful use of fsync and fdatasync to ensure the file system writes data to the disk. To motivate the metadata-prefix specification, we first consider a prototypical use of fsync and fdatasync. We then explain why it is important to have a precise specification and argue for the choice of our metadata-prefix specification.

Figure 7-1 shows the typical application pattern for using fsync and fdatasync, in combination with other file-system APIs, to update a file in a crash-safe manner. This pattern shows up in many real applications, such as a mail server, a text editor, a database, etc. Of course, prior research has shown that file systems provide different crash semantics [10, 57], so our example may not be crash-safe on some file systems and crash-safe on others. Nonetheless, we will explain why a developer might expect it to be crash-safe; this code also happens to be crash-safe on a file system that satisfies

```
tmpfile = "crashsafe.tmp"

def crash_safe_update(filename, data_blocks):
  f = open(tmpfile, "w")
  for block in data_blocks:
    f.write(block)
  f.close()

  fdatasync(tmpfile)
  rename(tmpfile, filename)
  fsync(dirname(filename))

def crash_safe_recover():
  unlink(tmpfile)
```

Figure 7-1: Pseudocode for an application pattern that updates the contents of a file in a crash-safe manner.

our metadata-prefix specification. The pattern assumes that the application never runs this function concurrently.

crash_safe_update(*f*, *data*) ensures that, after a crash, file *f* will have either its old contents or the new *data*; it will not have a mixture of old and new data, or partial new data, or any other intermediate state. To ensure this property, crash_safe_update first writes the new data into a temporary file. We assume that the file system enables log bypass and write-back caching, so that writing the new data to the file's data blocks does not go through the log, and the new data may not have been written to the file's data block yet.

Once crash_safe_update has finished writing data to the temporary file, it invokes fdatasync to force the file system to flush any buffered changes to the temporary file's data blocks from the write-back cache out to disk (and to issue a disk-write barrier).

After fdatasync returns, crash_safe_update replaces the original file with the new temporary file using rename. Since the file system's rename is atomic with respect to crashes, and the temporary file's contents are already on disk, if the system crashes at this point, the application will observe either the original contents (of the old file) or the new contents (of the new file). Finally, crash_safe_update uses fsync to flush its change to the directory, so that upon return, an application can be sure that the new data will survive a crash.

If the system crashes while executing crash_safe_update, it must first execute the file system's recovery code (which may replay transactions that have been committed but not applied), followed by its own recovery code. In our application pattern, the recovery code crash_safe_recover simply deletes the temporary file if one exists. This is sufficient for the pattern, since if the temporary file exists, we must have crashed in the middle of crash_safe_update and thus the original file still has its old contents.

The pattern allows for high performance because all system-call invocations, except for fsync and fdatasync, can be asynchronous. This allows the file system to defer writing directory and file modifications to disk, and thus allows it to batch updates. As a result, an efficient implementation of the file system could issue just two synchronous write barriers to disk: one for the fdatasync call and one for the fsync.

From the point of view of specifications, a clean but naïve API may be one that allows the application to encapsulate all write system calls into a single transaction, which is written to the on-disk log. While this simplifies the job of the application developer, it places a practical restriction on the amount of data that can be updated in a crash-safe manner, because the entire transaction must fit in the on-disk log. File systems have fixed-size logs that are typically small fractions of the total disk space. In fact, if the application issues a single write() that cannot fit in the log, the file system must break it up into multiple transactions. Thus, file systems in practice expose non-transactional writes and fsync/fdatasync to applications.

7.2 What should the specification be?

The POSIX specification does not well specify the behavior of fsync and fdatasync [38]. In particular, one aspect that has been the subject of disagreement is the behavior of fsync with respect to directories. To understand the challenges, consider the following possible specifications.

Suppose an application calls rename("d1/f", "d2/f"), followed by fsync("d1"). For performance reasons, a POSIX-compliant implementation might flush just the content of the specific directory (i.e., d1), allowing the file system to parallelize fsync on different files and directories. However, in our example, this would mean that the file f could be lost after a crash: it would be gone from d1, because d1 was synced to disk, but it would not yet appear in d2, because d2 was not yet synced. Thus, if the fsync specification required that just the directory itself was flushed, the file system may be able to get good performance, but it would be difficult for application developers to use such an API in the presence of crashes.

To make it easier for application developers to build crash-safe applications, the file system could provide a different specification, mirroring the traditional BSD semantics, where all metadata operations are synchronous (written to disk immediately), but file contents are asynchronous. This means that an application need not worry about calling fsync on a directory; the rename operation from the above example would be persisted to disk upon return. While this is simple to reason about, it achieves low performance for metadata-heavy workloads (such as extracting a tar file).

As can be seen from these different specifications, coming up with a specification for directory fsync requires striking a balance between achieving good performance and enabling application developers to reason about application-level crash safety.

We address this challenge by proposing a practical specification that is both easy to use at application level as well as allows for efficient file-system implementations. Specifically, the *metadata-prefix specification* says:

1. fdatasync(f) on file f flushes just the data of f. This allows for *log bypass* for file data blocks, which is important to avoid writing file data to disk twice. For example, this allows a database server to write to the disk (through a large pre-allocated file) without incurring file-system logging overheads.

2. fsync is a superset of fdatasync: it flushes both data and metadata. Furthermore, fsync flushes *all* pending metadata changes; i.e., if fsync(d) is called on directory d, it effectively ignores the argument and flushes changes to all other unrelated directories.

3. Finally, the file system is always allowed to flush any file's data, or all of the metadata operations performed up to some point in time. This allows the file system to re-order data and metadata writes to disk. After a crash, metadata updates will be consistently ordered (i.e., if the file system performed two operations, a and b, in that order, then after a crash, if b appears on disk, then so must a), but data writes can appear out-of-order.

This specification provides a clear contract between applications and a file system. Ensuring metadata ordering helps developers reason about the possible states of the directory structure after a crash: If some operation survives a crash, then all preceding operations must have also survived. For instance, in the crash_safe_update function from Figure 7-1, the developer knows that all directory changes have been flushed to disk once fsync(dirname(*filename*)) returns. Notably, this includes any possible pending changes to parent directories as well: for instance, if the application had just created the parent directory prior to calling crash_safe_update.

At the same time, the specification allows for high-performance file-system implementations: on the metadata side, it allows batching of metadata changes (until the next fsync call), and on the data side, it allows for log bypass for file data writes and allows for each file's *data* to be flushed independently of other files and of the metadata. This enables high performance for applications that perform in-place data modifications.

One limitation of the metadata-prefix specification is that it can flush unrelated changes to disk when an application invokes fsync, since the specification requires all metadata changes to be flushed together. In practice, the metadata-prefix specification captures behavior similar to that provided by the ext4 implementation (in its default configuration), largely as a side effect of ext4 having a system-wide log for all metadata.

As a result, we believe it is compatible with achieving sufficient performance for applications. Furthermore, we believe that FSCQ's techniques would be equally applicable to specifications that allow more aggressive performance optimizations as well.

In the rest of this chapter, we explore how to formalize the metadata-prefix specification using CHL.

7.3 Disk sequences

FSCQ's top-level file-system specifications also use a representation invariant to relate the state of a disk to the corresponding abstract file-system tree. For instance, the specification for the unlink system call might say that after unlink returns, the file is removed from the tree, and if unlink crashes, the file might or might not have been removed.

In the presence of deferred writes, it is difficult to describe the tree that might be on disk after a crash in the specification of unlink. Specifically, the on-disk state might have little to do with unlink itself, and instead might reflect the operations that were performed on the tree before unlink was called. For instance, the directory in which unlink is called might not have even been created yet.

To describe these crash states succinctly, FSCQ's insight is to avoid reasoning about the contents of a single tree, and instead to represent the possible on-disk state as a sequence of trees, with a designated *latest* tree. Each tree represents the state of the file system after some system call, and each system call adds a new tree to the sequence (and marks its tree as the latest).

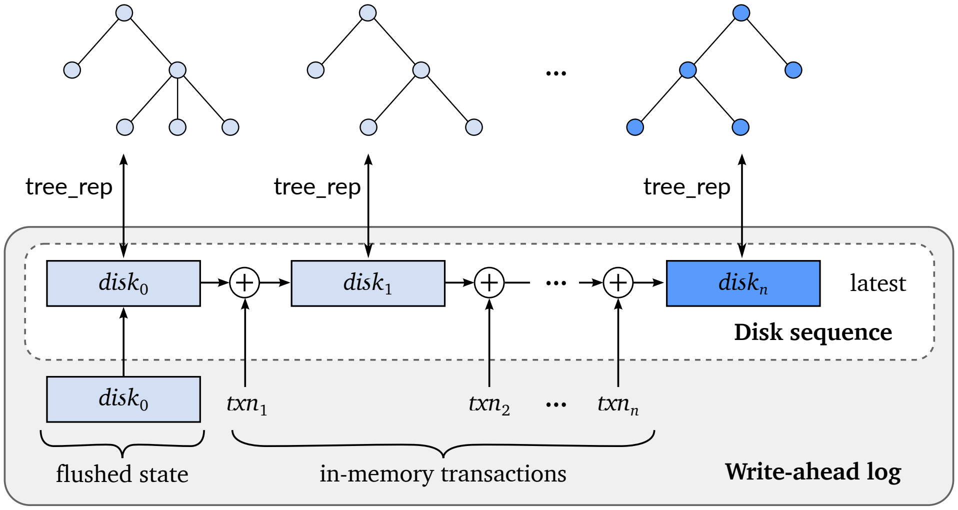


Figure 7-2: An illustration of FSCQ's disk-sequence abstraction

Figure 7-2 shows an example disk sequence. In the bottom left, the on-disk state represents the persistent state stored on disk. The list of transactions in the bottom row corresponds to the system calls that have committed in memory but whose changes have not been flushed to disk yet. Instead of reasoning about individual transactions, FSCQ reasons about disk sequences, each of which is a logical disk that would arise if one were to apply the in-memory transactions, in order, to the on-disk state. This sequence of logical disks is shown in the middle row of Figure 7-2. The top row shows the logical state of the file system (i.e., a directory tree structure) that corresponds to each of the logical disks in the disk sequence.

Disk sequences simplify specifications. For instance, consider the specification of the unlink system call, shown in Figure 7-3. The precondition describes the state of the system before unlink is called, by saying that there is some sequence of disks, called *disk_seq*, representing system calls that have been executed since the last fsync. log_rep, as defined in Figure 6-4, is a representation invariant connecting the physical state of the disk to its logical representation as a disk sequence. The postcondition adds a new disk to this sequence and states that the new disk contains a tree where the unlinked file is removed. The crash condition simply says that unlink can crash with any of the disks from the original disk sequence (corresponding to earlier system calls) or with the new disk that unlink added to this sequence.

SPEC $\quad \mathsf{unlink}\ (\mathit{cwd\_ino}, \mathit{pathname})$
PRE $\quad \mathbf{disk} \models \mathsf{log\_rep}(\mathsf{NoTxn}, \mathit{disk\_seq})$
$\qquad \mathbf{disk\_seq.latest} \models \mathsf{tree\_rep}(\mathit{old\_tree})$
POST $\quad \mathbf{disk} \models \mathsf{log\_rep}(\mathsf{NoTxn}, \mathit{disk\_seq} \mathbin{++} \{\mathit{new\_state}\})$
$\qquad \mathbf{new\_state} \models \mathsf{tree\_rep}(\mathit{new\_tree}) \ \wedge$
$\qquad\qquad \mathit{new\_tree} = \mathsf{tree\_prune}(\mathit{old\_tree},\ \mathit{cwd\_ino},\ \mathit{pathname})$
CRASH $\quad \mathbf{disk} \models \mathsf{would\_recover\_any}(\mathit{disk\_seq} \mathbin{++} \{\mathit{new\_state}\})$

Figure 7-3: Specification for unlink

The specification shown in Figure 7-3 reasons about disk sequences with a separate tree representation for each disk in the sequence. tree_rep is the representation invariant that connects the abstract file-system tree to the content on a logical disk. Specifically, tree_rep recursively maps the tree to a flat file address space, which in turn, maps to the inode and block content on the logical disk. We will discuss details of file-system construction in Chapter 8 but focus on the top-level specification for now.

Separating tree representation from disk sequences allows us to decouple the logging subsystem, which reasons about a sequence of transactions that have been committed but not flushed to disk, from the directory subsystem, which manages the file-system tree inside of a single transaction. Furthermore, note that the specification of unlink says nothing about earlier disks in the disk sequence. This is because unlink does not know, and should not care, what operations came before it. Instead, it is up to the application to keep track of what operations it invoked prior to unlink, if it wants to reason about the different trees that can arise after a crash.

Disk sequences naturally capture the metadata-prefix property, because the disk sequence is built up by applying the application’s system calls in order. As a result, we can succinctly describe the metadata-prefix property by saying that a crash during a system call can result in any of the disks from the disk sequence.

SPEC	fsync (*dir_ino*)
PRE	**disk** ⊨ log_rep(NoTxn, *disk_seq*)
	disk_seq.latest ⊨ tree_rep(*tree*) ∧ IsDir(find_inum(*tree*, *dir_ino*))
POST	**disk** ⊨ log_rep(NoTxn, {*disk_seq*.latest})
CRASH	**disk** ⊨ would_recover_any(*disk_seq*)

Figure 7-4: Specification for fsync on directories

Disk sequences also allow for a concise specification of fsync for directories, as shown in Figure 7-4. The specification directly derives from FscqLog’s log_flush specification as we have seen in Figure 6-8. It simply says that, after fsync on a directory returns, the disk sequence contains only the latest disk from before fsync. This latest tree reflects all of the system calls issued by the application up to its call to fsync. Consequently, if the application knows the corresponding tree for this latest disk, the application can conclude that, after fsync returns, this is the only tree that can arise after a later crash.

7.4 Disk relations

Writes to file data that bypass the log can cause the state of the file system after a crash to violate the order in which system calls were issued, since log-bypass writes are not ordered with respect to other updates that use the write-ahead log. For instance, if an application writes to an existing file and then renames the file, after a crash the file may have the new name but the old contents. Conversely, if the application first renames the file and then writes to it, after a crash the file may have the old name but the new contents.

Log-bypass writes pose a challenge for our disk-sequence abstraction, because writes made by an application in the latest state of the file system can affect earlier disks in the disk sequence: after a crash, the file system may end up in an earlier disk from the disk sequence, because the metadata changes might not have been fully flushed yet, but the later log-bypass writes might already appear on disk. To formalize the behavior of log-bypass writes, we need to describe how a write to a file affects not just the latest state of the file system, but every previous state in the disk sequence as well.

Log-bypass writes also pose a challenge for file-system correctness, when a block that belongs to a file in the current state of the file system used to belong to a different file, or to a directory, in an earlier state. If that were the case, and the file system issued a log-bypass write to this block, then after a crash that rolls back the log, the file system would appear to have modified the contents of a different file, or even worse, corrupted a directory. The typical approach for avoiding this problem is to ensure that data blocks are not reused until after the metadata log is applied to disk.

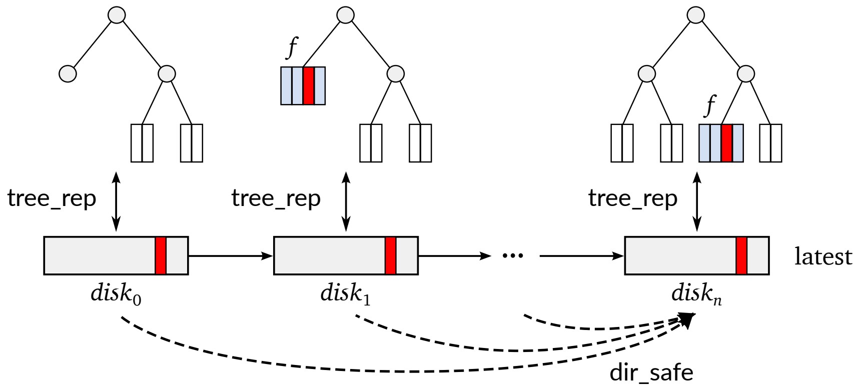


Figure 7-5: An illustration of how log-bypass writes interact with FSCQ's disk sequences, and the notion of disk relations. Dashed arrows represent the dir_safe relationship.

Figure 7-5 illustrates the problem with log-bypass writes in terms of a disk sequence. A log-bypass write (shown by dark red shading in the figure) affects every disk in the disk sequence, because under the covers, the disk sequence is simply a sequence of logical disk states that would arise if one were to apply the recent transactions to the real on-disk state. Thus, modifying the real disk state changes all of the disks in the disk sequence. The figure illustrates two subtleties with log-bypass writes. If the file being modified (f) was present elsewhere in the file-system hierarchy in a previous disk, it will also be affected by a log-bypass write. However, if the file is nowhere to be found, then the log-bypass write will have no effect on the file system's abstract tree state.

FSCQ introduces the idea of *disk relations* to address both of these challenges. The key idea is to relate two disks in a disk sequence to one another, to capture both the connection between the file-system trees at the specification level (i.e., modifying a file in the latest disk's tree will have the same effect in an earlier disk's tree) and to capture the internal consistency of the file system (i.e., that data blocks are not reused).

For example, Figure 7-6 shows the FSCQ specification for the pwrite system call. Here, pwrite's precondition requires that all of the disks in the disk sequence satisfy the dir_safe relation with respect to the latest disk state. This relation captures the internal consistency of the file system. Specifically, dir_safe(d_1, d_2) says that, if a disk block b in

```
SPEC    pwrite ($ino$, $off$, $buf$)
PRE     disk $\models$ log_rep(NoTxn, $old\_disk\_seq$) $\land$
                $\forall i$, dir_safe($old\_disk\_seq[i]$, $old\_disk\_seq$.latest)
        ∀i, old_disk_seq[i]  $\models$ tree_rep($trees[i]$)
         old_disk_seq.latest $\models$ tree_rep($tree$) $\land$
                $f$ = find_inum($tree$, $ino$) $\land$
                IsFile($f$) $\land$ $off < \| f.\text{data} \|$
POST    disk $\models$ log_rep(NoTxn, $new\_disk\_seq$) $\land$
                $\forall i$, dir_safe($new\_disk\_seq[i]$, $new\_disk\_seq$.latest)
         ∀i, new_disk_seq[i] $\models$ tree_rep($new\_trees[i]$) $\land$
                either $\exists path$, find_subtree($trees[i]$, $path$) = $\langle ino, f_i \rangle$ $\land$
                        $new\_trees[i]$ = tree_update($trees[i]$, $ino$, $f_i'$) $\land$
                        $f_i'$ = add_write($f_i$, $off$, $buf$)
                or $new\_trees[i] = trees[i]$
         new_disk_seq.latest $\models$ tree_rep($tree'$) $\land$
                $tree'$ = tree_update($tree$, $ino$, $f'$) $\land$
                $f'$ = add_write($f$, $off$, $buf$)
CRASH   disk $\models$ would_recover_any($old\_disk\_seq$) $\lor$
                would_recover_any($new\_disk\_seq$)
```

Figure 7-6: Specification for file write that bypasses the write-ahead log

disk d_2 belongs to a file with inode number i at offset o, then b in d_1 must either belong to a file with the same inode number i and offset o (but possibly with a different path name), or it must be in the free list. The bottom of Figure 7-5 shows the dir_safe relation between all older disks and the latest disk in a disk sequence.

The dir_safe relation enables pwrite to prove its postcondition. The latest state in the disk sequence simply says that the file now contains the new data. The add_write function, applied to a file, adds a pending write to that offset in a file; this models the fact that the underlying disk can now choose either this new value or some previously written values, to be at this offset after a crash. The more interesting part is what happens to earlier disks. The specification says that either a file with the same inode number will be modified, at the same offset, or that the operation may have no effect, if there is no such file. Note that the specification allows the file to have a different path name in a previous disk, corresponding to our earlier rename-and-write example.

Disk relations are also helpful in capturing other properties between disks. For instance, the pwrite specification shown in Figure 7-6 is simplified to avoid talking about inode-number reuse. While the shown specification is correct, a stronger specification (not shown) could use an inode-number-reuse relation to say that, even if an inode number was reused from an earlier disk, the disk blocks of that earlier file with the same inode number will not be affected.

One complication with log-bypass writes is the interaction between them and the resizing of a file. For example, if a file was truncated and then re-grown, the file may have the same inode number and the same length, but the underlying disk blocks are different. Thus, pwrite might or might not modify the file's blocks as they appear in an earlier disk from the disk sequence. The "either ... or" in the postcondition takes care of this problem.

```
SPEC    fdatasync ($ino$)
PRE     disk $\models$ log_rep(NoTxn, $old\_disk\_seq$) $\land$
                $\forall i$, dir_safe($old\_disk\_seq[i]$, $old\_disk\_seq$.latest)
        $\forall$i, old_disk_seq[i] $\models$ tree_rep($trees[i]$)
        old_disk_seq.latest $\models$ tree_rep($tree$) $\land$
                $f =$ find_inum($tree$, $ino$) $\land$ IsFile($f$) $\land$
                $\forall b,\ b < \|f.\text{data}\| \Rightarrow f.\text{data}[b] = \langle v_b,\ vs_b \rangle$
POST    disk $\models$ log_rep(NoTxn, $new\_disk\_seq$) $\land$
                $\forall i$, dir_safe($new\_disk\_seq[i]$, $new\_disk\_seq$.latest)
        $\forall$i, new_disk_seq[i] $\models$ tree_rep($new\_trees[i]$) $\land$
                if $\exists path$, find_subtree($trees[i]$, $path$) $= \langle ino,\ f_i \rangle$ then
                    $new\_trees[i] =$ tree_update($trees[i]$, $ino$, $f_i'$) $\land$
                    $\forall b$, if block_same($ino$, $b$, $old\_disk\_seq[i]$, $old\_disk\_seq$.latest)
                        then $f_i'.\text{data}[b] = \langle v_b,\ \emptyset \rangle$ else $f_i'.\text{data}[b] = f_i.\text{data}[b]$
                else $new\_trees[i] = trees[i]$
CRASH   disk $\models$ would_recover_any($old\_disk\_seq$)
```

Figure 7-7: Specification for fdatasync

This interaction shows up more prominently in the fdatasync specification, shown in Figure 7-7. Here, using an "or" is not sufficient, because the application wants to be sure that the blocks of the file are definitely flushed to disk. To address this problem, the fdatasync specification uses the block_same relation in its postcondition to say that, as long as the file did not shrink and re-grow since an earlier disk in the disk sequence, then the latest value of that block will be persistent on disk. Here, the precondition says that v_b was the last value written to each block b of the file (presumably using the file's add_write function, from pwrite's postcondition) but that some previously written data (corresponding to vs_b) might still be stored on disk. The postcondition uses the same notation to denote the fact that, if fdatasync succeeds, the disk cannot possibly contain any previously written data (denoted by $\emptyset$).

Disk relations fit well with our earlier specification for fsync, which flushes the write-ahead log to disk. Since fsync's postcondition, shown in Figure 7-4, says that the new disk sequence consists of just one disk, no additional safety relations are necessary, since there are no earlier disks to consider. Other system calls that add a new disk to the disk sequence, such as unlink, promise the dir_safe relation for the new disk in their postcondition. This was not shown in our earlier example in Figure 7-3 for simplicity.

EIGHT

Building a file system

This chapter describes FSCQ, the file system built on top of FSCQLOG, specified and certified using CHL. The implementation follows the organization shown in Figure 1-1 in Section 1.4. FSCQ's design closely follows the xv6 file system [18] and extends it to support log-bypass writes, fsync and fdatasync. The rest of this chapter describes FSCQ, the challenges we encountered in proving FSCQ, and the design patterns that we came up with for addressing them.

8.1 Overview

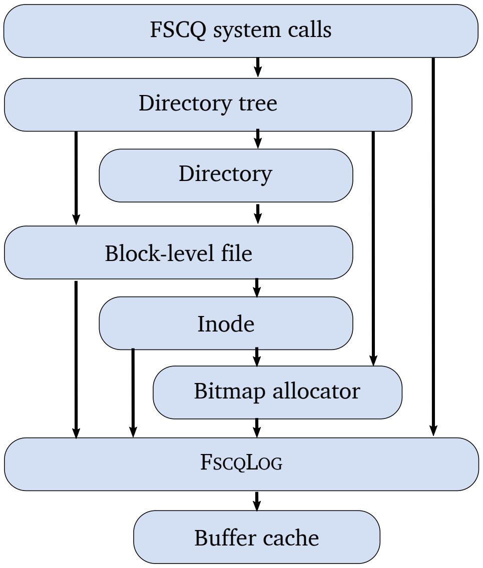


Figure 8-1: FSCQ components. Arrows represent procedure calls.

Figure 8-1 shows the overall components that make up FSCQ, with arrows showing the dependency among them. The buffer cache module at bottom provides a write-back cache of disk blocks. All disk accesses in FSCQ (including FSCQLOG) go through the cache.

Components above FscqLog provide simple implementations of standard file-system abstractions. Block allocator implements a bitmap allocator, used for both block and inode allocation. Inode implements an inode layer; the most interesting logic here is combining the direct and indirect blocks together into a single list of block addresses. Inode invokes block allocator to allocate indirect blocks and file data blocks. Block-level file exposes to higher levels an interface where each file is a list of blocks. Directory implements directories on top of block-level files. Directory tree combines directories and block-level files into a hierarchical directory-tree structure; it invokes the bitmap allocator to allocate/deallocate inodes when creating/deleting files or subdirectories. Finally, the top layer implements complete file-system calls in transactions.

Figure 8-2 shows FSCQ's disk layout. The super block contains information about where all other parts of the file system are located on disk and is initialized by mkfs. The shaded region inside the bold box is managed by FscqLog.

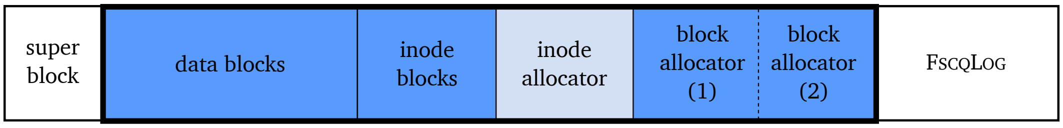


Figure 8-2: FSCQ on-disk layout

8.2 End-to-end specification

FSCQ provides a POSIX-like interface at the top level; the main differences from POSIX are (1) that FSCQ does not support hard links, (2) that FSCQ exposes a block-level file interface, as opposed to byte-level, and (3) that FSCQ does not implement file descriptors and instead requires naming open files by inode number. FSCQ relies on the FUSE driver to maintain the mapping between open file descriptors and inode numbers.

FSCQ formally specifies the crash-safety behavior for all of its APIs. For each system call, FSCQ provides two specifications: one with explicit crash conditions, and one combined with the recovery execution semantics to show the end-to-end guarantee of the system call. The former one is useful for applications built on top of FSCQ to prove their own crash-safety properties. To do that, applications supply their own recovery procedures (which usually call the file system's recovery procedure first) to FSCQ's recovery execution semantics. We have shown many such specifications in Chapter 7, including the specifications for unlink, fsync and fdatasync.

To write the end-to-end specification for a system call, we assume that the system call starts with no pending transactions and flushes the current transaction immediately

$$
\begin{array}{ll}
\text{SPEC} & \mathsf{rename}\ (\mathit{cwd\_ino}, \mathit{oldpath}, \mathit{newpath}) \quad \bowtie \quad \mathsf{fs\_recover}\ () \\
\text{PRE} & \textbf{disk} \models \mathsf{log\_rep}(\mathsf{NoTxn}, \{\mathit{start\_state}\}) \\
 & \textbf{start\_state} \models \mathsf{tree\_rep}(\mathit{old\_tree}) \ \land \\
 & \qquad\qquad \mathit{cwd\_tree} = \mathsf{find\_ino}(\mathit{old\_tree}, \ \mathit{cwd\_ino}) \ \land \ \mathsf{IsDir}(\mathit{cwd\_tree}) \\
\text{POST} & \textbf{disk} \models ((\mathit{status} = \langle \mathsf{Completed}, \ \mathsf{NoError} \rangle \ \lor \ \mathit{status} = \mathsf{Recovered}) \land \\
 & \qquad\qquad \mathsf{log\_rep}(\mathsf{NoTxn}, \ \{\mathit{new\_state}\})) \lor \\
 & \qquad\quad ((\mathit{status} = \langle \mathsf{Completed}, \ \mathsf{Error} \rangle \ \lor \ \mathit{status} = \mathsf{Recovered}) \land \\
 & \qquad\qquad \mathsf{log\_rep}(\mathsf{NoTxn}, \ \{\mathit{start\_state}\})) \\
 & \textbf{new\_state} \ \models \ \mathsf{tree\_rep}(\mathit{new\_tree}) \ \land \\
 & \qquad\qquad \langle \mathit{mover}, \ \mathit{ino} \rangle = \mathsf{find\_subtree}(\mathit{cwd\_tree}, \ \mathit{oldpath}) \land \\
 & \qquad\qquad \mathit{pruned} = \mathsf{tree\_prune}(\mathit{old\_tree}, \ \mathit{cwd\_ino}, \ \mathit{oldpath}) \land \\
 & \qquad\qquad \mathit{new\_tree} = \mathsf{tree\_graft}(\mathit{pruned}, \ \mathit{cwd\_ino}, \ \mathit{newpath}, \ \mathit{mover})
\end{array}
$$

Figure 8-3: Specification for rename with recovery

after commit; otherwise the recovered states might reveal an earlier logical disk that has nothing to do the system call in question.

Figure 8-3 shows FSCQ's end-to-end specification for its most complicated metadata operation, rename, in combination with FSCQ's recovery procedure fs_recover. rename's precondition requires that the directory tree is in a consistent state, matching the tree_rep invariant, and that the caller's current working directory inode, *cwd_ino*, corresponds to some valid path in the tree. The postcondition asserts that rename will either return an error, with the tree unchanged, or succeed, with the new tree being logically described by the functions tree_prune, tree_graft, etc. These functions operate on a logical representation of the directory-tree structure, rather than on low-level disk representations, and are defined in a few lines of code each. In case of a crash, the state will either have no effects of rename or will be as if rename had finished.

8.3 Using address spaces

Since transactions take care of crashes, the remaining challenge lies in specifying the behavior of a file system and proving that the implementation meets its specification on a reliable disk. As mentioned in Section 4.3, CHL's address spaces help express predicates about address spaces at different levels of abstraction. For example, consider the specification shown in Figure 8-4 for file_block_write, which overwrites existing blocks through the log. This specification uses separation logic in four different address spaces: the physical disk (which implements asynchronous writes and matches the log_rep predicate); the abstract disks inside the transaction, *old_state* and *new_state* (which have synchronous writes and match the files_rep predicate); the address spaces of files indexed by inode number, *old_files* and *new_files*; and finally the address spaces of file indexed by block offset, *old_f*.data and *new_f*.data. The use of separation logic within each address space allows us to concisely specify the behavior of file_block_write at all these levels of abstraction. Furthermore, CHL applies its proof-automation machinery to separation logic in *every* address space. This helps developers construct short proofs about higher-level abstractions.

SPEC	file_block_write (*inum*, *off*, *block*)
PRE	**disk** $\models$ log_rep(ActiveTxn, *disk_seq*, *old_state*)
	old_state $\models F \star$ files_rep(*old_files*)
	old_files $\models F_{files} \star inum \mapsto old\_f$
	old_f.data $\models F_{blocks} \star off \mapsto \langle v_0, vs_0 \rangle$
POST	**disk** $\models$ log_rep(ActiveTxn, *disk_seq*, *new_state*)
	new_state $\models F \star$ files_rep(*new_files*)
	new_files $\models F_{files} \star inum \mapsto new\_f$
	new_f.data $\models F_{blocks} \star off \mapsto \langle block, \varnothing \rangle \land new\_f.\mathrm{attr} = old\_f.\mathrm{attr}$
CRASH	**disk** $\models$ log_rep(ActiveTxn, *disk_seq*, *any_state*)

Figure 8-4: Specification for writing to a file through the log

8.4 Resource allocation

File systems must implement resource allocation at multiple levels of abstraction—in particular, allocating disk blocks and allocating inodes. We built and proved correct a common allocator in FSCQ. It works by storing a bitmap spanning several contiguous blocks, with bit i corresponding to whether object i is available. FSCQ instantiates this allocator for both disk-block and inode allocation, each with a separate bitmap.

Writing a naïve specification of the allocator is straightforward: freeing an object adds it to a set of free objects, and allocating returns one of these objects. The allocator's representation invariant asserts that the free set is correctly encoded using "one" bits in the on-disk bitmap. However, the caller of the allocator must prove more complex statements—for example, that any object obtained from the allocator is not already in use elsewhere. Reproving this property from first principles each time the allocator is used is labor-intensive.

To address this problem, FSCQ's allocator provides a free_objects_pred(*obj_set*) predicate that can be applied to the address space whose resources are being allocated. This predicate is defined as a set of ($\exists v,\ i \mapsto v$) predicates for each i in *obj_set*, combined using the $\star$ operator. *obj_set* is typically the allocator's set of free object IDs, so this predicate states that every free object ID points to some value.

Using free_objects_pred simplifies reasoning about resource allocation, because it can be combined with other predicates about the objects that are currently in use (e.g., disk blocks used by files), to give a complete description of the address space in question. The disjoint nature of the ⋆ operator precisely capture the idea that all objects are either available (and managed by the allocator) or are in use (and match some other predicate about the in-use objects).

$$\begin{aligned}
\mathsf{files\_rep}(\mathit{files}) \;:=\; & \exists\, \mathit{free\_blocks}_1 \;\exists\, \mathit{free\_blocks}_2 \;\exists\, \mathit{inodes}, \\
& \mathsf{allocator\_rep}(\mathit{free\_blocks}_1) \;\star \\
& \mathsf{allocator\_rep}(\mathit{free\_blocks}_2) \;\star \\
& \mathsf{inode\_rep}(\mathit{inodes}) \;\star \\
& \mathsf{files\_inuse\_rep}(\mathit{inodes}, \mathit{files}) \;\star \\
& \mathsf{free\_objects\_pred}(\mathit{free\_blocks}_1) \;\star \\
& \mathsf{free\_objects\_pred}(\mathit{free\_blocks}_2)
\end{aligned}$$

Figure 8-5: Representation invariant for FSCQ's file layer

For example, Figure 8-5 shows the representation invariant for FSCQ's file layer, which is typically applied to FscqLog's abstract disk address space, as shown in Figure 8-4. The abstract disk, according to Figure 8-5, is split up into six disjoint parts: two allocation bitmaps (represented by allocator_rep), the inode area (represented by inode_rep), file data blocks (represented by files_inuse_rep), and two free-block regions (described by free_objects_pred). The allocator's representation invariant (allocator_rep) connects the on-disk bitmap to the set of available blocks ($\mathit{free\_blocks}_1$ and $\mathit{free\_blocks}_2$). The file_inuse_rep function combines the inode state in *inodes* (containing a list of block addresses for each inode) and the logical file state *files* to produce a predicate describing the blocks currently used by all files. Finally, free_objects_pred asserts that the free blocks are disjoint from blocks used by the other predicates. By this definition, files_rep covers all dark blue regions shown in Figure 8-2.

The reason that files_rep includes two separate block allocators ($\mathit{free\_blocks}_1$ and $\mathit{free\_blocks}_2$) is to avoid undesired block reuse. As mentioned in Section 7.4, log bypass for file-data writes requires that disk blocks are not reused until the log is flushed to disk. To implement this, FSCQ uses two disk-block allocators. When a disk block is freed, it goes into one of the allocators. When FSCQ needs a new disk block, it allocates from the other one. When FSCQ flushes the log to disk, it swaps the roles of the two allocators, since it is now safe to reuse blocks freed prior to the log flush. This trick simplified FSCQ's proofs since we did not need to implement and prove a separate list of pending blocks that cannot yet be reused.

The same pattern applies to allocating inodes as well. The only difference is that, in files_rep, the predicate describing the actual bitmap, allocator_rep, and the predicate describing the available objects, free_objects_pred, were both applied to the same address space (the abstract disk). In the case of inodes, the two predicates are applied to different address spaces: the bitmap predicate is applied to the abstract disk, but free_objects_pred is applied to the inode address space.

8.5 Buffer cache

FSCQ implements a write-back buffer cache: writes are buffered in memory until being evicted. Eviction happens when reads and writes bring other blocks into the buffer cache. The buffer cache exposes an asynchronous interface resembling that of the physical disk and also uses the subset-points-to relation in its specifications. Both reads and writes bring the accessed block into the cache and can cause another block to be evicted if the cache is full. Similar to background disk syncs, implicit eviction does not change the cache's observable state, allowing concise specifications for reads and writes.

The buffer cache provides a cache_sync operation that allows the caller to explicitly evict a block from the cache, and, if the evicted block is dirty, the buffer cache will write the block to disk and issue a write barrier to persist the change. The postcondition of cache_sync guarantees that the evicted block contains a single possible value. For better performance, the buffer cache also offers a few variants of cache_sync that allow to evict several blocks, but only issuing a single disk-write barriers (see Section 6.5).

There is another interesting aspect in the design of our buffer cache: how it implements replacement policies. We wanted the flexibility to use different replacement algorithms, but proving the correctness of each algorithm posed a nontrivial burden. Instead, we borrowed the *validation* approach from CompCert [49]: rather than proving that the replacement algorithm always works, FSCQ checks if the result is safe (i.e., is a currently cached block) before evicting that block. If the replacement algorithm malfunctions, FSCQ evicts the first block in the buffer cache. This allows FSCQ to implement replacement algorithms in unverified code while still guaranteeing overall correctness.

8.6 On-disk data structures

Another common task in a file system is to lay out data structures in disk blocks. For example, this shows up when storing several inodes in a block; storing directory entries in a file; storing addresses in the indirect block; and even storing individual bits in the allocator bitmap blocks. To factor out this pattern, we built the Rec library for packing and unpacking data structures into bit-level representations. We often use this library to pack multiple fields of a data structure into a single bit vector (e.g., the bit-level

```
Definition inode_type : Rec.type := Rec.RecF ([
  ("len",  Rec.WordF 64);      (* number of blocks *)
  ("attrs", iattr_type);       (* file attributes, another record type *)
  ("indptr", Rec.WordF 64);    (* indirect pointer *)
  ("blocks", Rec.ArrayF 9 (Rec.WordF 64))]). (* direct block pointers *)
```

Figure 8-6: FSCQ's on-disk inode layout

representation of an inode) and then to pack several of these bit-vectors into one disk block.

For example, Figure 8-6 shows FSCQ's on-disk inode structure, in Coq syntax. The first field is len, storing the number of blocks in the inode, as a 64-bit integer (Rec.WordF indicates a word field). The other fields are the file's attributes (such as the modification time), the indirect-block pointer indptr, and a list of 9 direct block addresses, blocks.

The library proves basic theorems, such as the fact that accesses to different fields are commutative, that reading a field returns the last write, and that packing and unpacking are inverses of each other. As a result, code using these records does not have to prove low-level facts about layout in general.

One additional pattern that shows up is the need to treat several contiguous blocks as a single list of objects. For example, FSCQ has several contiguous blocks storing inodes. It is helpful to reason about on-disk inodes in terms of a single list containing all inodes in these blocks. A similar pattern shows up for log descriptors, directory entries and even for the bits in the allocator bitmaps. FSCQ provides a RecArray library that captures this common pattern.

8.7 Prototype implementation

The implementation follows the organization shown in Figure 1-1 in Section 1.4. FSCQ and CHL are implemented using Coq, which provides a single programming language for implementation, stating specifications, and proving them. Figure 8-7 breaks down the source code of FSCQ and CHL. Because Coq provides a single language, proofs are interleaved with source code and are difficult to separate. The development effort took several researchers about two years; most of it was spent on proofs and specifications. Checking the proofs takes about 8 hours on an Intel i7-980X 3.33 GHz CPU with 24 GB DRAM. The proofs are complete; we used Coq's Print Assumptions command to verify that FSCQ did not introduce any unproven axioms or assumptions.

CHL. CHL is implemented as a domain-specific language inside of Coq, much like a macro language (i.e., using a shallow embedding). We specified the semantics of this language and proved that it is sound. For example, we proved the standard Hoare-logic

Component	Lines of code
FSCQ and CHL infrastructure	21,360
Hashing semantics	459
General data structures	3,863
Buffer cache	2,362
Write-ahead log	10,391
Inodes and files	3,212
Directories	5,698
FSCQ's top-level API	1,997
Total	49,342

Figure 8-7: Combined lines of code and proof for FSCQ components

specifications for the for and if combinators. We also proved the specifications of disk_read, disk_write (whose specifications is in Figure 4-2 in Section 4.2), and disk_sync manually, starting from CHL's execution and disk model. Much of the automation (e.g., the chaining of pre- and postconditions) is implemented using Ltac, Coq's domain-specific language for proof search.

FSCQ. We implemented FSCQ also inside of Coq, writing the specifications using CHL. We proved that the implementation obeys the specifications, starting from the basic operations in CHL. FSCQLOG simplified FSCQ's specification and implementation tremendously, because much of the detailed reasoning about crashes is localized in FSCQLOG.

FSCQ file server. We produced running code by using Coq's extraction mechanism to generate equivalent Haskell code from our Coq implementation. We wrote a driver program in Haskell (550 lines of code) along with an efficient Haskell reimplementation of fixed-size words, disk-block operations, and a buffer-cache replacement policy (470 more lines of Haskell). The extracted code, together with this driver and word library, allows us to efficiently execute our certified implementation.

To allow applications to use FSCQ, we exported FSCQ as a FUSE file system, using the Haskell FUSE bindings [9] in our Haskell FSCQ driver. We mount this FUSE FSCQ file system on Linux, allowing Linux applications to use FSCQ without any modifications. Compiling the Coq and Haskell code to produce the FUSE executable, without checking proofs, takes a little under two minutes.

Limitations. Although extraction to Haskell simplifies the process of generating executable code from our Coq implementation, it incurs high CPU overhead and adds the

Haskell compiler and runtime into FSCQ's trusted computing base. In other words, a bug in the Haskell compiler or runtime could subvert any of the guarantees that we prove about FSCQ. We believe this is a reasonable trade-off, since our goal is to certify higher-level properties of the file system, and other projects have shown that it is possible to extend certification all the way to assembly [13, 33, 47].

Another limitation of the FSCQ prototype lies in dealing with in-memory state in Coq, which is a functional language. CHL's execution model provides a mutable disk but gives no primitives for accessing mutable memory. We address this by explicitly passing an in-memory state variable through all FSCQ functions. This contains the current buffer-cache state (a map from address to cached block value) as well as the current transaction state in several FSCQLOG layers. In the future, we want to support multiprocessors where several threads share a mutable buffer cache, and we will address this limitation.

NINE

Evaluation

This chapter answers the following questions about FSCQ:

- Is FSCQ complete enough for realistic applications, and can it achieve reasonable performance? (Section 9.1)
- What kinds of bugs do FSCQ's theorems preclude? (Section 9.2)
- Are FSCQ's specifications correct and useful? Does FSCQ recover from crashes? (Section 9.3)
- How difficult is it to build and evolve the code and proofs for FSCQ? (Section 9.4)

9.1 Application and I/O performance

FSCQ is complete enough that we can use FSCQ for software development, running a mail server, etc. For example, we have used FSCQ with the GNU coreutils (ls, grep, etc.), editors (vim and emacs), software development tools (git, gcc, make, and so on), and running a qmail-like mail server. Applications that, for instance, use extended attributes or create very large files do not work on FSCQ, but there is no fundamental reason why they could not be made to work.

Experimental setup. To validate that FSCQ's design indeed achieves usable performance and its I/O efficiency goal, we run two benchmarks to stress both data and metadata aspects of FSCQ. We run mailbench, a qmail-like mail server from the sv6 operating system [15]. This models a real mail server, where using FSCQ would ensure email is not lost even in case of crashes. We also used a modified LFS largefile benchmark [60] to evaluate FSCQ's I/O performance. mailbench performs many metadata operations by manipulating small files, and our modified largefile performs many data writes to an existing large file followed by fdatasync calls.

We compare FSCQ's performance to Linux ext4 file system in two configurations: the default mode (async,data=ordered), which enables log-bypass writes for file data, but does not use log checksum; and the logged mode (journal_async_commit,data=journal), which uses write-ahead log for both file data and metadata, and uses checksums to commit in one disk sync instead of two. We run ext4 in two modes because FSCQ implements both log bypass and log checksum, while the two optimizations are incompatible in ext4 [42]. To match the two configurations, we also run FSCQ in two similar modes: one with log bypass and the other without log bypass. Log checksum is enabled for FSCQ in both modes. To make fair comparison, we also run ext4 inside FUSE with a modified fusexmp_fh driver, which serializes and forwards all file-system requests to a native ext4 partition.

We run all of these experiments on a quad-core Intel(R) Core(TM) i7-980X 3.33 GHz CPU with 24 GB memory running Linux 3.11. Unless specified otherwise, the file system was stored on a separate partition on a Hitachi HDS721010CLA332 hard disk drive. We compiled FSCQ's Haskell code using GHC 8.0.1.

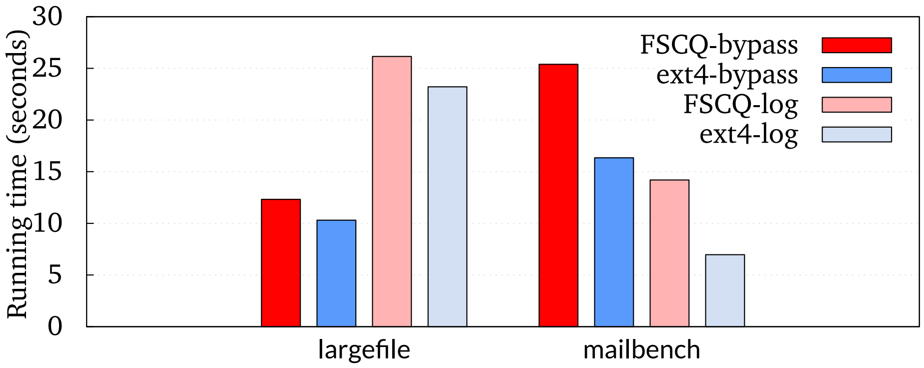


Figure 9-1: Running time for largefile and for delivering 200 messages in mailbench on a hard disk drive. "ext4-bypass" runs ext4 in the default configuration (async,data=ordered); "ext4-log" runs ext4 in logged configuration (journal_async_commit,data=journal)

Application performance. The results of running our experiments are shown in Figure 9-1. The first conclusion is that for largefile, the logged configuration is about 2× slower than the bypass configuration; this is true for both FSCQ and ext4. This is because without log bypass, every file data update will result in at least two disk writes. Therefore, log bypass is an essential optimization for applications that frequently overwrite large chunks of existing data.

Second, the result of mailbench shows the opposite: For ext4, the logged configuration performs 2.3× better than the log-bypass configuration. This is because mailbench manipulates many small files, and log-bypass writes have an adverse effect when running on a rotational hard disk drive due to the seek time. In contrast, updating small files through the log turns most random accesses into sequential accesses, effectively eliminates disk seeks [60]. In addition, ext4's log-bypass configuration does not use log checksum, requiring two write barriers to commit a transaction, while the logged configuration needs only a single write barrier per metadata transaction.

Third, for largefile, FSCQ's performance is close to that of the ext4 file system in both configurations: FSCQ is about 20% slower than ext4 in the bypass mode and about 13% slower in the logged mode, respectively. The small performance gap is due to the fact that FSCQ's Haskell implementation uses about 4× more CPU time than ext4's. We hope to adopt ideas for better generation of certified assembly code in future work.

Forth, for mailbench running in the log-bypass configuration, FSCQ is about 55% slower than ext4. Besides FSCQ's higher CPU overhead, this is because mailbench creates many small files and appends to them. An appending operation requires writing to the file's data block twice: once from the logging system to initialize the newly allocated block (which is turned into more disk writes) and once through the log-bypass write to store the actual data. Due to FSCQ's log-bypass design, this results in more writes and syncs than ext4 running in the bypass configuration, as shown in Figure 9-3. We discuss FSCQ's I/O performance below.

Finally, for mailbench running in the logged configuration, FSCQ is about 2× slower than ext4. This is also due to two reasons: 1) FSCQ has a higher CPU overhead; and 2) FSCQ's log layout is not yet optimized for reducing the seek time. FSCQLOG's header, descriptor and data regions start at fixed disk locations (see Section 6.4.1), thus flushing a transaction involves writing to several non-contiguous addresses. In contrast, ext4's log design ensures that flushing a transaction can be done using sequential disk writes. In fact, FSCQ issues fewer number of disk I/Os than ext4 in this case, as shown in Figure 9-3. We expect the performance difference would go away after reducing FSCQ's CPU overhead and running the same benchmark on an SSD.

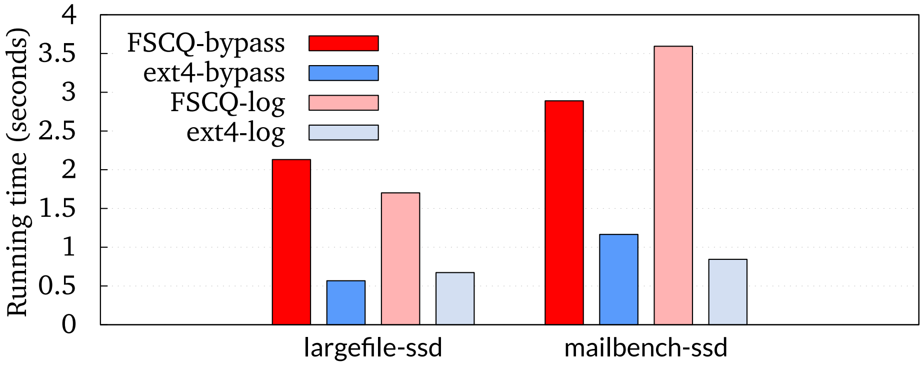


Figure 9-2: Application performance of FSCQ running on an SSD drive

We also run the same set of benchmarks on an OCZ-VERTEX3 SSD drive. The result in shown in Figure 9-2. FSCQ's performance is 2× to 4× slower than ext4 in each configuration, although the absolute performance of both file systems is better than running on a hard disk drive. This is because SSD drives have no seeks and exhibit lower latency than a rotational hard disk drive. As the benchmark becomes CPU-bounded, FSCQ suffers more performance penalty from Haskell's CPU overhead.

I/O performance. A primary goal of FSCQ was to achieve good I/O performance by supporting group commit, log bypass, log checksum, and other optimizations. To validate that FSCQ's design indeed achieves its I/O efficiency goal, and to further understand the end-to-end performance results reported in Figure 9-1, we use the Linux blktrace support to trace the disk operations performed by each file system.

Figure 9-3 shows the results for this experiment, reporting the number of disk writes and disk write barriers (syncs) issued by each of the file systems per application-level operation (delivering a mail message in mailbench and writing a 4KB block in a large file for largefile). We draw several conclusions.

	largefile		**mailbench**	
	writes	syncs	writes	syncs
FSCQ-bypass	1.0	1.0	50.0	9.8
ext4-bypass	1.0	1.0	38.0	7.5
FSCQ-log	4.0	1.0	28.8	3.5
ext4-log	5.0	1.0	38.3	3.7

Figure 9-3: I/O performance of FSCQ compared to Linux ext4. Each cell reports the number of writes and barriers, respectively, per application-level operation.

First, FSCQ indeed achieves good I/O efficiency, issuing a similar number of write barriers and disk writes to ext4 in both configurations. For largefile, FSCQ and ext4 have the same number of write barriers (1.0 per application-level block write) in both configurations.

Second, for largefile, both FSCQ and ext4's log-bypass configurations write each block to disk just once, whereas the logged configurations perform 4 to 5 disk writes for each application-level block write. This is because every application-level write turns into an on-disk transaction, which later must be applied separately. For logged configurations, FSCQ requires fewer number of disk writes to complete each operation because FSCQ supports fewer features than ext4 and consequently performs less bookkeeping for metadata. For example, when writing to a file, ext4 also updates the modification time in the file's inode, while FSCQ does not do so for simplicity.

Third, for mailbench running on FSCQ and ext4, the number of disk barriers issued in the logged configurations is about half of that in the bypass configurations. This is because both FSCQ and ext4's logged configuration enables log checksum, which commits each transaction with a single write barrier. In contrast, ext4's default configuration (ext4-bypass) does not use log checksum, requiring two write barriers to commit a transaction. The effect of log-checksum optimization is more prominent here because mailbench's operations are metadata-intensive, thus committing more transactions than largefile does.

Finally, for mailbench under log-bypass configurations, FSCQ requires 30% more writes and syncs than ext4. This is because every time mailbench writes to a newly created file, FSCQ has to flush all cached transactions to disk. More specifically, when growing a file, FSCQ first issues a transactional write to initialize the newly allocated block, commits the file-grow transaction and then issues another log-bypass write to the same block to store the actual data. Because the second write hits the previously cached transactional write, GroupCommit has to flush all cached transactions to preserve the ordering (see Section 6.5). In contrast, ext4 optimizes this scenario by discarding previously committed write from the transaction cache. This optimization is incompatible with log checksum [42], which is always enabled in our FSCQ prototype.

To summarize, FSCQ indeed achieves usable application performance, and its I/O performance is on par with Linux ext4. The evaluation also indicates that there is still room for further optimizations, such as reducing the CPU overhead, adopting a better log layout to minimize seek time and optimizing for file-grow operations.

9.2 Bug discussion

We answer the question of whether FSCQ's theorems prevent real bugs in two ways. First, we present a case study of different kinds of bugs that have been discovered in the Linux ext4 file system and argue that FSCQ prevents them. Second, we describe our own experience in developing FSCQ and point out specific bugs that were caught in the process of proving its correctness.

Bug category and example	**Possible in FSCQ?**	**Prevented by FSCQ?**
Logging logic; write/barrier ordering [19, 42, 65]	Yes	Yes
Misuse of logging API [61, 66]	Yes	Yes
Bugs in recovery protocol [37, 52]	Yes	Yes
Improper corner-case handling [73]	Yes	Yes
Low-level bugs [42, 53, 72]	Some (memory safe)	Yes
Concurrency [43, 64]	No	—

Figure 9-4: Representative bugs found in Linux ext4 and whether FSCQ's specifications preclude them

ext4 bugs case study. We looked through the git logs for the Linux ext4 file system starting from 2013 and categorized the bugs fixed in those commits. Figure 9-4 shows the resulting categories along with representative bugs from each category. For instance, this table includes the bug that was mentioned in Chapter 1, where ext4 would disclose previously deleted file data after a crash [42]. The figure also shows whether each bug category could have occurred in the implementation of FSCQ; for instance, some bugs arise due to concurrent execution of system calls, which is impossible in FSCQ by design (i.e., FSCQ is not sophisticated enough to have such a bug). The figure also shows whether the theorems of FSCQ prevent those bugs.

We draw three conclusions from this case study. First, FSCQ is sophisticated enough that its implementation could have had many of the bugs that were fixed in ext4, making verification important. Second, FSCQ's theorems preclude every bug category that was possible in its implementation. This suggests that FSCQ's theorems are effective at preventing real bugs. Finally, the one category where FSCQ is not sophisticated enough to have bugs is concurrency: FSCQ is a single-threaded file system. Verifying a concurrent file system is an open problem and is future work.

Development experience. While proving the correctness of FSCQ, we ran into several cases where we were unable to prove a correctness theorem and discovered an underlying implementation issue as a result. For instance, when mknod was invoked on an existing pathname, it would delete the old file. This was allowed by the specification (which in itself could have arguably been a bug), but more importantly, it failed to de-allocate the old file's blocks. This violated the dir_safe relation, and we were unable to prove that log bypass would be safe after mknod. The previous example is also related to log bypass: while trying to prove that it is safe to bypass the log for modifying a file data block, we realized that there could be a pending non-bypass write to that same block in the write-ahead log. This forced us to change the system's design for handling log-bypass writes, as described in Section 6.5. These examples show that proofs are good at bringing out corner cases that are easy to overlook during development and testing.

9.3 Specification correctness

To demonstrate that FSCQ's specifications for its system calls are meaningful, we performed the following experiments.

fsstress. We ran fsstress from the Linux Test Project to check if it finds any bugs in FSCQ. When we first ran fsstress, it caused our FUSE file server to crash. However, after some investigation, we discovered that this was due to a bug in our Haskell FUSE bindings that sit between FSCQ and the Linux FUSE interface. The bug was due to the developer thinking that some corner case could not be triggered and calling the error function in Haskell to panic if that case ever executed. As it turns out, fsstress found a way to trigger that corner case. After fixing this bug, fsstress ran without problems and did not discover any bugs in FSCQ's proven code.

Enumerating crash states. We implemented the crash_safe_update pattern whose pseudocode was shown in Figure 7-1. Our specific implementation writes and syncs some data to a temporary file using fdatasync, then performs an atomic rename of the temporary file to a destination file using fsync on the directory. We ran the pattern on FSCQ while monitoring all of the disk writes and barriers issued by FSCQ. We then computed all possible subsets and re-orderings of FSCQ's disk writes, subject to its barriers, to produce every possible state in which FSCQ could have crashed. Finally, we re-mounted the resulting disk with FSCQ and examined the file-system state after FSCQ performed its recovery. This experiment produced 182 possible disks after a crash but only three distinct file-system states after FSCQ executed its recovery code: neither file existed, the temporary file existed with no contents, or the destination file existed with the written contents. All of these states are safe, since either the destination file didn't exist or it contained the correct data (the empty temporary file could be removed during recovery).

Certifying an application. The above experiments suggest that FSCQ specifications capture the right properties, because the implementation appears not to have bugs. However, to increase our confidence that the specifications themselves are correct (and not just the implementation), we wrote a formal specification for the crash_safe_update pattern and proved its correctness based on the specification of FSCQ.

Proving the correctness of crash_safe_update led us to discover several cases where the FSCQ specification was too weak. For example, the read specification originally forgot to mention that the data returned by the system call is related to the contents of the file. Another example is the fsync system call, which forgot to promise that the dir_safe relation holds on the new disk sequence. This also uncovered many cases where the specification was not as convenient to use as it could have been. None of these issues required changing the FSCQ implementation, and we were able to re-prove the correctness of FSCQ after fixing the specification.

Proving crash_safe_update also led us to discover a number of corner cases in its code. For example, we discovered that crash_safe_update cannot perform a safe update on a file with the same file name as the temporary file that it uses. After fixing the specification to take into account these corner cases, we were able to prove the correctness of crash_safe_update when running on top of FSCQ.

Specification conciseness. Generally speaking, a shorter specification is easier to inspect and understand and thus less likely to have flaws. FSCQ's specification allows the file system to perform many complex optimizations that are visible to the application, especially after a crash. One concern might be that the specification for such a complex interface is also complicated. To understand whether FSCQ's formalization techniques are effective at succinctly describing FSCQ's allowed behavior, we look at the lines of code needed to specify the pre, post, and crash conditions of each system call.

	PRE	POST	CRASH	Total
read	4	3	2	9
write	5	13	4	22
create	4	11	2	17
unlink	4	11	2	17
rename	4	16	2	22
stat	3	3	2	8
truncate	4	10	7	21
open	5	4	2	11
fdatasync	4	11	2	17
fsync	3	3	2	8
Total	42	89	29	160

Figure 9-5: Lines of specification code for FSCQ system calls

The result in Figure 9-5 shows that FSCQ's top-level specification has 160 lines in total. The average number of lines for each system call is about 15. The most complicated specifications are rename and write, whose easier-to-read version are shown in Figure 8-3 and Figure 7-6, respectively. This is unsurprising because rename is the most complicated system call that manipulates the metadata; and the specification for log-bypass write has to reason about disk relations. Nevertheless, their specifications consist of 22 lines each. This suggests that FSCQ's specifications are concise and amenable to human inspection.

9.4 Development effort

The final question is, how much effort is involved in developing FSCQ? One metric is the size of the FSCQ code base, reported in Figure 8-7; FSCQ consists of about 50,000 lines of code, which includes specifications, proofs and a significant amount of CHL infrastructure, including libraries and proof machinery, which is not FSCQ-specific. In comparison, Linux ext4 has about 60,000 lines of unverified C code.

Although the code footprint of FSCQ and ext4 are similar, ext4 offers many features that are absent in FSCQ, such as permissions, extended attributes and sparse files. Therefore, a more interesting question is how much effort is required to *modify* FSCQ, after an initial version has been developed and certified. Does adding a new feature to FSCQ require reproving everything, or is the work commensurate with the scale of the modifications required to support the new feature? To answer this question, the rest of this section presents several case studies, where we had to add a significant feature to FSCQ after the initial design was already complete.

Asynchronous disk writes. We initially developed FSCQ and FscqLog to operate with synchronous disk writes. Implementing asynchronous disk writes required changing about 1,000 lines of code in the CHL infrastructure and changing over half of the implementations and proofs for FscqLog. However, layers above FscqLog did not require any changes, since FscqLog provided the same synchronous disk operations in both cases.

Indirect blocks. Initially, FSCQ supported only direct blocks. Adding indirect blocks required changing about 1,500 lines of code and proof in the Inode layer, including infrastructure changes for reasoning about on-disk objects that span multiple disk blocks (the inode and its indirect block). We made almost no changes to code above the inode layer; the only exception was block-level files, in which we had to fix about 50 lines of proof due to a hard-coded constant bound for the maximum number of blocks per file.

Buffer cache. We added a buffer cache to FSCQ after we had already built FscqLog and several layers above it. Since Coq is a pure functional language, keeping buffer-cache state required passing the current buffer-cache object to and from all functions. Incorporating the buffer cache required changing about 300 lines of code and proof in FscqLog, to pass around the buffer-cache state, to access disk via the buffer cache and to reason about disk state in terms of buffer-cache invariants. We also had to make similar straightforward changes to about 600 lines of code and proof for components above FscqLog.

Optimizing log layout. FscqLog's initial design used one disk block to store the length of the on-disk log and another block to store a commit bit, indicating whether log recovery should replay the log contents after a crash. Once we introduced asynchronous writes, storing these fields separately necessitated an additional disk sync between writing the length field and writing the commit bit. To avoid this sync, we modified the logging protocol slightly: the length field was now *also* the commit bit, and the log is applied on recovery iff the length is nonzero. Implementing this change required modifying about 50 lines of code and about 100 lines of proof.

Log checksum. We developed the log-checksum optimization in parallel with other logging-system optimizations. This involved about 1300 lines of change to the CHL infrastructure to reason about the new hash opcode and its crash semantics. Integrating checksums into FSCQLOG changed about 800 lines of code and proof in DiskLog to handle the new commit block layout and about 800 lines of proofs in DiskLog and Applier to reason about checksum-based recovery. We also had to mechanically change about 500 lines above FSCQLOG to include the notion of hash state in all internal specifications.

Group commit. We added group commit to FSCQLOG after the entire file system was proven. We realized that GroupCommit could be implemented as an intermediate layer between LogAPI and Applier. Originally, LogAPI called Applier's applier_flush upon commit. We changed it to call GroupCommit's group_commit instead, which buffers the transaction in memory and calls applier_flush with a single combined transaction when possible. GroupCommit itself contains about 1800 lines of code. Because of the layering, there was little change to both LogAPI and Applier. We also modified about 100 lines of specifications above FSCQLOG to adapt to the new disk-sequence abstraction; these changes were mechanical.

TEN

Conclusion and future directions

Sophisticated file systems have a long history of subtle bugs that lead to data loss or unintended data disclosure. FSCQ is the first file system that achieves high I/O performance (on par with Linux ext4) and has a machine-checked proof that its implementation meets a formal specification, under any sequence of crashes.

To achieve this goal, this dissertation contributes Crash Hoare Logic (CHL), a logic framework that allows us to concisely and precisely specify the expected behavior of FSCQ. Because of CHL's proof automation, the burden of proving that FSCQ meets its specification is manageable. FSCQ also provides the first precise specification of fsync and fdatasync, called the metadata-prefix specification. To help formalize the specification and certify optimizations, this dissertation introduces several specification techniques, such as disk sequences, disk relations, and a hash model. The benefit of the verification approach is that FSCQ provably avoids bugs that have a long history of causing data loss in previous file systems, while achieving good I/O performance. FSCQ's specification also enables applications built on top of it to prove their own crash-safety. We hope that others will find FSCQ, CHL and our specification techniques useful for certifying other crash-safe storage systems.

FSCQ provides strong properties, but it is not a finished product. Fruitful areas of future research include:

Extracting to native code. Although FSCQ achieves good I/O performance, the extracted Haskell code incurs significant CPU overhead and adds the Haskell compiler and runtime into FSCQ's trusted computing base (TCB). We would like to generate certified executable code for FSCQ, which would enable FSCQ to implement efficient low-level optimizations, such as utilizing fast machine-level bit-operations as well as eliminate the Haskell runtime from the TCB. Some recent work such as COGENT [2] is working towards this direction. We hope to incorporate similar ideas to improve FSCQ's CPU performance.

Certifying crash-safe applications. The example application pattern used in our evaluation, the crash-safe file update, is simple; but proving it correct on top of FSCQ still requires non-trivial effort. One problem is that FSCQ's current tree-based top-level specification is not amenable to proof automation. We would like to investigate how to certify a complete application, such as a mail server or a key-value store, using better specification and proof-automation techniques.

Supporting concurrency. All practical file systems run in multi-user environments and exploit concurrency to achieve good performance. FSCQ currently does not support concurrent system calls and does not model any shared in-memory state. We need to reason about two main forms of concurrency: (1) I/O concurrency, where computation of one process overlaps with the I/O of another process; and (2) parallelism, where computations run on different cores and access shared in-memory state truly concurrently (e.g., reading and writing to a shared buffer cache). Verification of concurrent programs is an open problem in general. Certifying a file system with limited concurrency might be a good concrete problem to advance the state of the art of concurrent verification.

Bibliography

[1] Ramnatthan Alagappan, Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi 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] Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O'Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, Joseph Tuong, Gabriele Keller, Toby Murray, Gerwin Klein, and Gernot Heiser. COGENT: Verifying high-assurance file system implementations. In *Proceedings of the 21th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 175–188, Atlanta, GA, April 2016.

[3] June Andronick. Formally proved anti-tearing properties of embedded C code. In *Proceedings of the 2nd IEEE International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, pages 129–136, Paphos, Cyprus, November 2006.

[4] Konstantine Arkoudas, Karen Zee, Viktor Kuncak, and Martin Rinard. Verifying a file system implementation. In *Proceedings of the 6th International Conference on Formal Engineering Methods*, Seattle, WA, November 2004.

[5] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, May 2014.

[6] Gilles Barthe, Cédric Fournet, Benjamin Grégoire, Pierre-Yves Strub, Nikhil Swamy, and Santiago Zanella-Béguelin. Probabilistic relational verification for cryptographic implementations. In *Proceedings of the 41st ACM Symposium on Principles of Programming Languages (POPL)*, San Diego, CA, January 2014.

[7] William R. Bevier and Richard M. Cohen. An executable model of the Synergy file system. Technical Report 121, Computational Logic, Inc., October 1996.

[8] William R. Bevier, Richard M. Cohen, and Jeff Turner. A specification for the Synergy file system. Technical Report 120, Computational Logic, Inc., September 1995.

[9] J. Bobbio et al. Haskell bindings for the FUSE library, 2014. https://github.com/m15k/hfuse.

[10] James Bornholt, Antoine Kaufmann, Jialin Li, Arvind Krishnamurthy, Emina Torlak, and Xi Wang. Specifying and checking file system crash-consistency models. In *Proceedings of the 21th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 83–98, Atlanta, GA, April 2016.

[11] Michael Carbin, Sasa Misailovic, and Martin C. Rinard. Verifying quantitative reliability for programs that execute on unreliable hardware. In *Proceedings of the 2013 Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Indianapolis, IN, October 2013.

[12] Haogang Chen, Daniel Ziegler, Adam Chlipala, M. Frans Kaashoek, Eddie Kohler, and Nickolai 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.

[13] Adam Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 234–245, San Jose, CA, June 2011.

[14] Adam Chlipala. The Bedrock structured programming system: Combining generative metaprogramming and Hoare logic in an extensible program verifier. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 391–402, Boston, MA, September 2013.

[15] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie 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, November 2013.

[16] Coq development team. *The Coq Proof Assistant Reference Manual, Version 8.5pl1*. INRIA, April 2016. http://coq.inria.fr/distrib/current/refman/.

[17] Jonathan Corbet. ext4 and data loss. http://lwn.net/Articles/322823/, March 2009.

[18] Russ Cox, M. Frans Kaashoek, and Robert T. Morris. Xv6, a simple Unix-like teaching operating system, 2014. http://pdos.csail.mit.edu/6.828/2014/xv6.html.

[19] Lukas Czerner. [PATCH] ext4: Fix data corruption caused by unwritten and delayed extents. https://lwn.net/Articles/645722, April 2015.

[20] Brian Demsky and Martin Rinard. Automatic detection and repair of errors in data structures. In *Proceedings of the 18th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 78–95, Vancouver, Canada, October 2003.

[21] Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew Parkinson, and Hongseok Yang. Views: Compositional reasoning for concurrent programs. In *Proceedings of the 40th ACM Symposium on Principles of Programming Languages (POPL)*, pages 287–300, Rome, Italy, January 2013.

[22] 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.

[23] Gidon Ernst, Gerhard Schellhorn, Dominik Haneberg, Jorg Pfähler, and Wolfgang 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.

[24] Robert Escriva. Claiming Bitcoin's bug bounty, November 2013. http://hackingdistributed.com/2013/11/27/bitcoin-leveldb/.

[25] Miguel Alexandre Ferreira and Jose Nuno 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*, August 2009.

[26] Leo Freitas, Jim Woodcock, and Andrew Butterfield. POSIX and the verification grand challenge: A roadmap. In *Proceedings of 13th IEEE International Conference on Engineering of Complex Computer Systems*, pages 153–162, March–April 2008.

[27] FUSE: Filesystem in userspace, 2013. http://fuse.sourceforge.net/.

[28] Philippa Gardner, Gian Ntzik, and Adam Wright. Local reasoning for the POSIX file system. In *Proceedings of the 23rd European Symposium on Programming*, pages 169–188, Grenoble, France, 2014.

[29] Roxana Geambasu, Andrew Birrell, and John MacCormick. Experiences with formal specification of fault-tolerant storage systems. In *Proceedings of the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Anchorage, AK, June 2008.

[30] Bogdan Gribincea et al. Ext4 data loss. https://bugs.launchpad.net/ubuntu/+source/linux/+bug/317781, January 2009.

[31] Ronghui Gu, Jeremie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. Deep specifications and certified abstraction layers. In *Proceedings of the 42nd ACM Symposium on Principles of Programming Languages (POPL)*, Mumbai, India, January 2015.

[32] Haryadi S. Gunawi, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. SQCK: A declarative file system checker. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 131–146, San Diego, CA, December 2008.

[33] Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. Ironclad Apps: End-to-end security via automated full-system verification. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 165–181, Broomfield, CO, October 2014.

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

[35] Wim H. Hesselink and M.I. Lali. Formalizing a hierarchical file system. In *Proceedings of the 14th BCS-FACS Refinement Workshop*, pages 67–85, December 2009.

[36] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[37] Ben Hutchings. [PATCH 3.2 027/115] jbd2: fix fs corruption possibility in jbd2_journal_destroy() on umount path. https://lkml.org/lkml/2016/4/26/1230, April 2016.

[38] 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), April 2013.

[39] Dave Jones. Trinity: A Linux system call fuzz tester, 2014. http://codemonkey.org.uk/projects/trinity/.

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

[41] Eunsuk Kang and Daniel Jackson. Formal modeling and analysis of a Flash filesystem in Alloy. In *Proceedings of the 1st Int'l Conference of Abstract State Machines, B and Z*, pages 294–308, London, UK, September 2008.

[42] Jan Kara. [PATCH] ext4: Forbid journal_async_commit in data=ordered mode. http://permalink.gmane.org/gmane.comp.file-systems.ext4/46977, November 2014.

[43] Jan Kara. ext4: fix crashes in dioread_nolock mode. http://permalink.gmane.org/gmane.linux.kernel.commits.head/575311, February 2016.

[44] Gabi Keller. Trustworthy file systems, 2014. http://www.ssrg.nicta.com.au/projects/TS/filesystems.pml.

[45] Gabriele Keller, Toby Murray, Sidney Amani, Liam O'Connor, Zilin Chen, Leonid Ryzhyk, Gerwin Klein, and Gernot Heiser. File systems deserve verification too. In *Proceedings of the 7th Workshop on Programming Languages and Operating Systems*, Farmington, PA, November 2013.

[46] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems*, 32(1):2:1–70, February 2014.

[47] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Michael Norrish, Rafal Kolanski, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 207–220, Big Sky, MT, October 2009.

[48] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[49] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, July 2009.

[50] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. A study of Linux file system evolution. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)*, pages 31–44, San Jose, CA, February 2013.

[51] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter 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, March 1992.

[52] Kamal Mostafa. [PATCH 3.13 075/103] jbd2: fix descriptor block size handling errors with journal_csum. https://lkml.org/lkml/2014/9/30/747, September 2014.

[53] Kamal Mostafa. ext4: fix null pointer dereference when journal restart fails. https://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/commit/?id=9d506594069355d1fb2de3f9104667312ff08ed3, June 2016.

[54] Gian Ntzik, Pedro da Rocha Pinto, and Philippa Gardner. Fault-tolerant resource reasoning. In *Proceedings of the 13th Asian Symposium on Programming Languages and Systems (APLAS)*, Pohang, South Korea, November–December 2015.

[55] Frances Perry, Lester Mackey, George A. Reis, Jay Ligatti, David I. August, and David Walker. Fault-tolerant typed assembly language. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, San Diego, CA, June 2007.

[56] J. Pfähler, G. Ernst, G. Schellhorn, D. Haneberg, and W. Reif. Crash-safe refinement for a verified flash file system. Technical Report 2014-02, University of Augsburg, 2014.

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

[58] John 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.

[59] Tom Ridge, David Sheets, Thomas Tuerk, Andrea Giugliano, Anil Madhavapeddy, and Peter 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)*, Monterey, CA, October 2015.

[60] 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, October 1991.

[61] Eric Sandeen. [PATCH] ext4: fix unjournaled inode bitmap modification. http://permalink.gmane.org/gmane.comp.file-systems.ext4/35119, October 2012.

[62] Gerhard Schellhorn, Gidon Ernst, Jorg Pfähler, Dominik Haneberg, and Wolfgang Reif. Development of a verified flash file system. In *Proceedings of the ABZ Conference*, June 2014.

[63] Richard D. Schlichting and Fred B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems*, 1(3):222–238, 1983.

[64] Theodore Ts'o. ext4: fix race between truncate and __ext4_journalled_writepage(). https://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/commit/?id=bdf96838aea6a265f2ae6cbcfb12a778c84a0b8e, June 2015.

[65] Theodore Ts'o. [PATCH] ext4, jbd2: add req_fua flag when recording an error flag. http://permalink.gmane.org/gmane.comp.file-systems.ext4/49323, July 2015.

[66] Theodore Ts'o. [PATCH] ext4: use private version of page_zero_new_buffers() for data=journal mode. https://lkml.org/lkml/2015/10/9/1, October 2015.

[67] David Walker, Lester Mackey, Jay Ligatti, George A. Reis, and David I. August. Static typing for a faulty Lambda calculus. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, Portland, OR, September 2006.

[68] Stephanie Wang. Certifying checksum-based logging in the RapidFSCQ crash-safe filesystem. Master's thesis, Massachusetts Institute of Technology, June 2016.

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

[70] Markus Wenzel. Some aspects of Unix file-system security, August 2014. http://isabelle.in.tum.de/library/HOL/HOL-Unix/Unix.html.

[71] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. Verdi: A framework for implementing and formally verifying distributed systems. In *Proceedings of the 2015 ACM SIGPLAN Conference*

on Programming Language Design and Implementation (PLDI), pages 357–368, Portland, OR, June 2015.

[72] Darrick J. Wong. jbd2: Fix endian mixing problems in the checksumming code. http://lists.openwall.net/linux-ext4/2013/07/17/1, July 2013.

[73] Darrick J. Wong. [PATCH] ext4: fix same-dir rename when inline data directory overflows. http://permalink.gmane.org/gmane.comp.file-systems.ext4/45594, August 2014.

[74] Jean Yang and Chris Hawblitzel. Safe to the last instruction: Automated verification of a type-safe operating system. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 99–110, Toronto, Canada, June 2010.

[75] Junfeng Yang, Can Sar, Paul Twohey, Cristian Cadar, and Dawson 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.

[76] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal 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, December 2004.

[77] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal 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, November 2006.

[78] Mai Zheng, Joseph Tucek, Dachuan Huang, Feng Qin, Mark Lillibridge, Elizabeth S. Yang, Bill W. Zhao, and Shashank 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, October 2014.