Verifying an I/O-Concurrent File System

by

Tej Chajed

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

Master of Science in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2017

Author .....
Department of Electrical Engineering and Computer Science
January 31, 2017

Certified by .....
M. Frans Kaashoek
Charles Piper Professor
Thesis Supervisor

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

Accepted by .....
Professor Leslie A. Kolodziejski
Chair, Committee on Graduate Students

Verifying an I/O-Concurrent File System

by

Tej Chajed

Submitted to the Department of Electrical Engineering and Computer Science
on January 31, 2017, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science

Abstract

Systems software is a good target for verification due to its prevalent usage and its complexity, which can lead to tricky bugs that are hard to test for. One source of complexity in systems software is concurrency, but thus far verification techniques have struggled to enable large-scale verification of concurrent systems. This thesis contributes a verified file system, CIO-FSCQ, with I/O concurrency: if a file system call experiences a miss in the buffer cache and starts a disk I/O, the file system overlaps the I/O with the execution of another file system call.

CIO-FSCQ re-uses the implementation, specifications, and proofs of an existing verified sequential file, FSCQ, and turns it into an I/O-concurrent file system. This re-use is enabled by CIO-FSCQ's *optimistic system calls*. An optimistic system call runs sequentially if all the data it needs is in the buffer cache. If some data is not in the cache, CIO-FSCQ issues I/Os to retrieve the data from disk and returns an error code. In the miss case, a system call wrapper reverts any partial changes and yields the processor so that another system call can run in parallel with the I/O. CIO-FSCQ retries the system call later, at which point the data is likely in the buffer cache. A *directory-isolation protocol* guarantees that FSCQ's specifications and proofs can be re-used even if optimistic system calls are retried. An evaluation of CIO-FSCQ shows that it speeds up a simple file-system workload by overlapping disk I/O with computation, and that the effort of building and verifying CIO-FSCQ is small compared to the effort of verifying FSCQ.

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

Thesis Supervisor: Nickolai Zeldovich
Title: Associate Professor

Acknowledgments

This thesis wouldn't have been possible without the help and support of many people. First, I thank my parents for instilling an appreciation for the importance of education, and my whole family for their eternal support, of myriad forms I couldn't possibly enumerate. Thanks to everyone who gave feedback on the thesis: Jon Gjengset, Robert Morris, Nickolai Zeldovich, and Frans Kaashoek. Thanks to Adam Chlipala, who mentored us through all the PL. Special thanks go to Nickolai, for staying on top of the technical details enough to always be ready to bounce ideas off of, and for valuable benchmarking guidance. And most of all thanks to Frans, for his faith and support when there was no file system, and patience and support while writing the thesis.

This research was supported in part by NSF award 1563763.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Verification of software correctness has recently made great progress and is becoming realistic for systems [4, 15–17, 24, 26, 39, 40]. Verified systems come with a specification and a machine-checked proof that their implementation meets their specification. Verification eliminates large classes of bugs that plague software. Unfortunately, these successes have largely involved sequential systems. While many approaches have been proposed for verifying concurrent programs, there are few examples of applying these approaches to complete systems.

This thesis focuses on adding I/O concurrency to a file-system implementation, taking advantage of the idle CPU to execute system calls while disk operations complete in the background. The rest of this chapter explains I/O concurrency within a file system and describes a novel approach to take a verified sequential file system (FSCQ) and make it I/O concurrent while re-using the sequential implementation, specifications, and proofs. The chapter concludes with a summary of the main contributions and with a roadmap for the rest of the thesis.

1.1 Problem and goal

Today's hardware offers many opportunities to exploit concurrency. A modern computer has several processors and I/O devices. Exploiting these opportunities is challenging, because writing correct concurrent code is difficult. The programmer must

avoid tricky race conditions that can result in subtle bugs. Research in verification has resulted in many frameworks to prove the absence of such subtle bugs [10, 12, 13, 32, 34, 35]. These frameworks strive to apply fine-grained notions of concurrency, but tend to focus on small example programs rather than large systems. In this thesis we want to explore how to add simple but effective forms of concurrency to existing sequential systems, re-using sequential specifications and their proofs as much as possible. Here we do not tackle multiprocessor concurrency, focusing on parallelism between a single processor and an I/O device, specifically a disk. We isolate concurrent reasoning such that the bulk of the verification re-uses the large verification effort present in FSCQ.

As a starting point of this exploration, this thesis targets a limited form of concurrency: I/O concurrency. I/O concurrency allows I/O to devices to proceed in parallel with computation on a processor. Consider two applications sharing a single processor that have both issued system calls to the file system. Without any concurrency, the system calls would simply execute sequentially. With I/O concurrency, if the first application's data isn't in memory, the file system issues a disk read to fetch it but does not wait for its completion. Instead, it starts executing the second application's system call using the otherwise idle processor. The file system resumes the first application's system call at some point after the I/O completes. Since I/O operations typically take a long time, I/O concurrency can lead to a large improvement in performance: while one system call is blocked waiting for I/O, another can compute. Without I/O concurrency the processor would be idle for large periods of time.

Our goal is to add I/O concurrency to the sequential FSCQ while re-using its implementation, specifications, and proofs as much as possible. This problem is challenging since FSCQ does not support several threads of computation and has no support to coordinate accesses of several threads to shared data structures in the file system such as the buffer cache. To understand the challenge in more depth, consider two threads, one of them executing a rename operation. Suppose the rename locates the source file, avoiding I/O by relying on cached data, and unlinks it, then goes on

to locate the destination directory. If the destination directory isn't in the cache, then ideally the file system would start the disk I/O and switch to the other thread. However, now the other thread could observe an inconsistent state: the file doesn't exist, in either in the source or the destination directory. Such a state would never be observable in a sequential file system without I/O concurrency.

The problem here is that even on a uniprocessor, I/O concurrency leads to interleaving and interference between threads. At first glance changing FSCQ to support I/O concurrency seems to require much work: the top-level specifications must be changed to capture threads interleaving, the implementation must avoid problematic interference, and the proofs must be changed to verify that the new implementation meets the new specification. How can we reduce the burden of this work?

1.2 Approach: optimistic systems calls

The approach we take is to execute system calls *optimistically*, hoping to read all necessary data from the buffer cache. If that is the case, the system call behaves like a regular sequential system call: it runs from the start to the end without other system calls interleaving. If the system call misses in the buffer cache, the file system issues a disk read and rolls back any writes the system call has made since it started. At some point later, the file system restarts the system call (*e.g.*, after the disk I/O completes) from the beginning. Now it is likely that the data needed for the system call is in the buffer cache, and the system call executes to completion. This approach works well since system calls tend to each read only a few blocks and thus need few retries, and the buffer cache is large enough to hold blocks for several system calls.

Figure 1-1 illustrates the approach by showing an example execution in terms of system calls. An ongoing rename misses in the cache at (1). It initiates reading from disk but does not wait for the result, instead rolling back its partial state and restarting at (2). Other threads run, then rename restarts. This time it hits in the cache at (3), blocking if the disk fetch is not yet complete. A subsequent read requires the same restart if the cache misses. If all cache reads succeed, then the

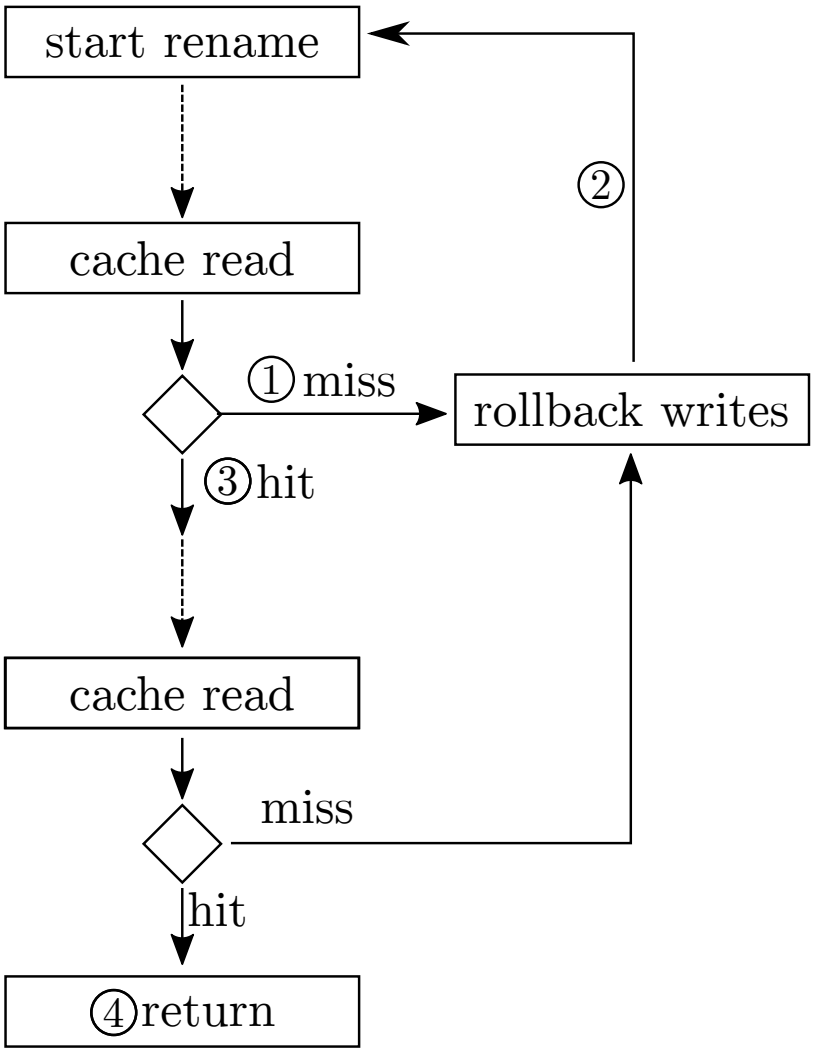


Figure 1-1: Execution flowchart of the CIO-FSCQ rename system call, showing a cache miss at ①, rollback and restart at ②, and subsequent cache hit ③ at the same point. When all cache reads hit, rename can run to ④ and complete without I/O.

function proceeds to (4) and returns. It is possible that multiple system calls miss in the cache and initiate disk reads, which can then run in parallel. The disk hardware might then pipeline requests and complete them faster.

To be able to revert changes inexpensively, an optimistic system call buffers its changes in memory and commits them only when the system call completes. If the system call must abort because one of its reads misses in the cache, the system deletes the buffered changes, rolling the file system back to its state before the system call started.

This approach is inspired by the `EAGAIN` error code that certain Unix systems calls can return [21]. For example, modern UNIXes support asynchronous read system calls and return `EAGAIN` when the data isn't available yet. A key difference, however, is that we return an error code from within all optimistic system calls, not just the read-only ones. This pattern is safe since when we retry optimistic system calls, any writes are rolled back before other threads run.

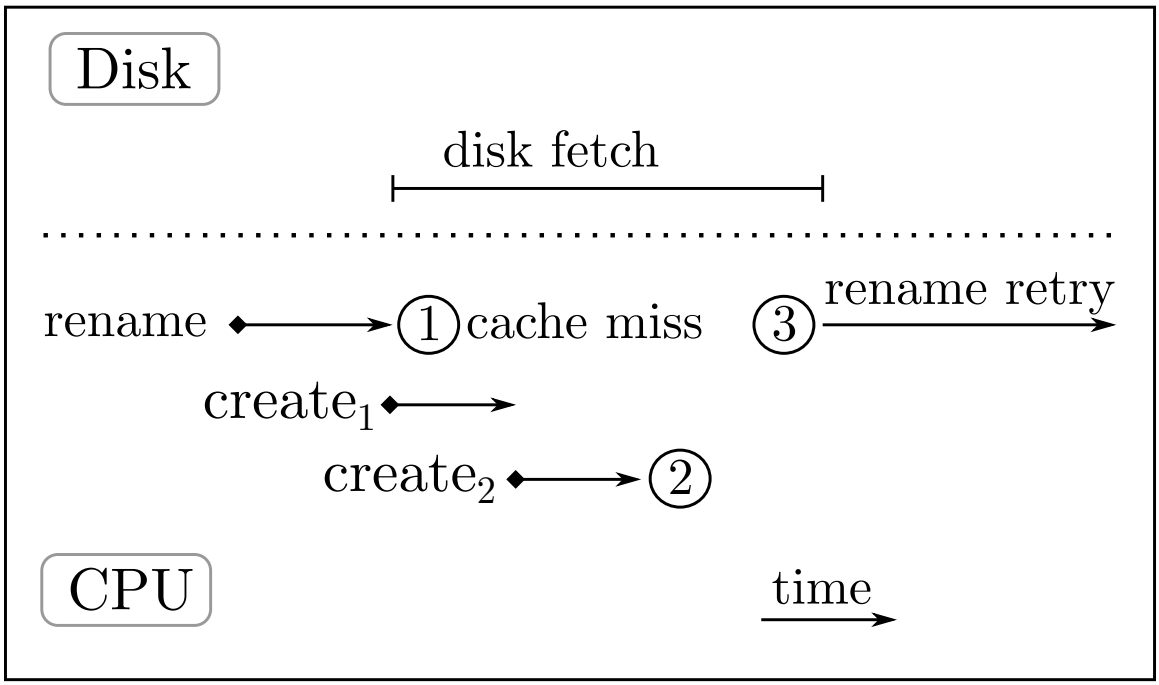


Figure 1-2: An example execution showing I/O concurrency: the create_1 and create_2 system calls run in parallel with the disk I/O for the rename.

Figure 1-2 show how optimistic system calls achieve I/O concurrency and how

they can improve performance compared to a non-I/O concurrent file system. As in the previous example, an ongoing rename misses on a read in the cache at (1). While fetching from disk, rather than let the CPU sit idle, the file system runs two system calls from another application that modify data in the cache: create_1 and create_2 without needing I/O. These two system calls execute concurrently with the disk I/O for the rename. At (3) the disk read completes and the rename is re-scheduled; this time it executes to completion without missing in the cache.

The approach using optimistic system calls allows us to reuse much of the specification, implementation, and proofs of the sequential FSCQ. With the above approach, switching between threads happens only when the optimistic system call misses in the cache. Since the system state is rolled back, the switch logically appears before the system call has started. Inside the file system the optimistic call runs sequentially without interleaving with other threads, and can thus be implemented and verified by re-using the sequential FSCQ.

1.3 Design overview

The above approach reduces the specification, implementation, and proof work of turning FSCQ into CIO-FSCQ to developing the following components:

- **Cooperative Concurrency Logic (CCL).** A concurrent logic to define the operations in I/O-concurrent programs (*e.g.*, starting a read, yielding the processor, and collecting the result of the read later) and a way of writing specifications for programs that use those operations.

- **CIO-Cache.** A concurrent cache which schedules I/O when reads miss, returns an error code to the system call when a read misses, and blocks threads when on a retry the I/O hasn't completed. We insert this buffer cache between FSCQ and the disk, and it provides the same disk interface FSCQ expects, so that we can insert it without any modifications to FSCQ.

- **System call wrapper.** A wrapper around optimistic system calls that handles

errors by rolling back, yielding the processor to other threads, and retrying afterward.

- **Sequential to concurrent translator.** CIO-translator translates FSCQ's sequential system calls to automatically produce optimistic system calls using CIO-Cache. These sequential calls are programs written using FSCQ's Crash Hoare Logic (CHL). The translator re-writes these sequential programs to programs in CCL using CIO-Cache rather than accessing the disk directly. A general proof of correctness for the translator guarantees that verified sequential code is translated to an optimistic system call with an analogous specification, providing a verified implementation of much of the concurrent system call.

- **A proof of correctness assuming a file-system protocol.** The remaining proof is that the wrapped optimistic system calls are correct, based on proofs for the optimistic system calls alone. This is difficult to prove because when a thread retries, other threads have run in the meantime, while to use the sequential specification for the retry, its precondition must still hold. FSCQ's specifications in particular impose some restrictions in each precondition beyond the file-system invariants, notably that the current working directory exist for each system call (a requirement that the VFS layer in Linux guarantees).

  To reason about the behavior of other threads, as in other concurrent logics, CCL requires users to define a *protocol* that defines a restriction on threads so that we can prove the correctness of interleaved executions. We can use this protocol to satisfy FSCQ's restrictions on each precondition even when other threads run. The most interesting protocol under which we prove correctness is a protocol that restricts threads to operate on different directories, a common usage pattern for file systems in practice. Under this assumption, we can prove that threads that modify their own directories can retry FSCQ's system calls after other threads run. The top-level specification states that as long as all threads follow the protocol, each operation is linearizable: it appears to occur atomically after other threads run.

These components are mostly independent of the specifics of FSCQ, except for the last component, which involves defining a protocol and showing that it preserves FSCQ's preconditions. Chapter 3 describes each of these components in detail.

While building and verifying the above components requires substantial effort, they represent much less work than writing an I/O-concurrent file system from scratch. The extra verification work is mostly in reasoning about concurrency inside the cache and at the level of retrying systems calls.

1.4 CIO-FSCQ prototype

We built a prototype of CIO-FSCQ within the Coq proof assistant [6]. Most of the implementation uses Gallina, the purely functional language of Coq, with a strategy for modeling external I/O and concurrency that borrows from FSCQ. We use Coq's built-in extraction feature combined with a trusted interpreter in Haskell to produce runnable implementations of each system call. As in FSCQ, these implementations can be run using the standard UNIX system call interface from unmodified applications with FUSE, a library that forwards file-system calls to a userspace file-system implementation.

There are three main caveats to the prototype of CIO-FSCQ. First, CIO-Cache currently does not provide persistence; data is not written from the cache to the disk, so the file system loses data when unmounted and re-mounted. This is only a limitation of the prototype, which could provide an `unmount` operation to empty the cache and prove that `unmount` does not change the abstract disk or directory tree.

Second, CCL does not model crashes. Handling crashes requires some careful changes to the design. In particular, the protocol would likely have to be extended with a crash invariant, since a crash could occur while other threads are running. Translation of crash persistence could be accomplished by moving writeback to the end of a system call; without I/O concurrency for flushing data to disk this change would be relatively straightforward. Due to the lack of crashes in the semantics, we also do not model asynchronous writes with a disk write barrier in CCL, making

writes instantly persistent instead. Reasoning about crashes and asynchronous writes would be an interesting direction of future work, and would test how CHL generalizes to a concurrent setting.

Third, CIO-FSCQ inherits the large CPU overhead of FSCQ due to following the same approach to executing the Coq code. The performance of the extracted Haskell, combined with inefficiency within the Gallina implementation (*e.g.*, deserialization of entire data structures for a single record), means both FSCQ and CIO-FSCQ can be bottlenecked by the CPU rather than I/O. For this reason, we can only demonstrate an improvement in CIO-FSCQ compared to FSCQ on slow I/O devices.

1.5 Thesis contributions

The main result of this thesis is the first verified file system with I/O concurrency. More specifically, the contributions of the thesis are as follows:

1. An approach based on optimistic system calls for adding I/O concurrency to an existing verified file system, as a means of obtaining concurrency with lower verification effort.

2. A design consisting of five components to realize the goal of a verified I/O concurrent file-system.

3. A prototype of the design in Coq, including a runtime platform in Haskell to execute system calls through the standard file-system interface.

4. An evaluation of the prototype demonstrating I/O concurrency. On a parallel workload of two threads, CIO-FSCQ is 26% faster overall and 3× faster for the CPU-bound thread that blocks on the other thread's disk I/O in FSCQ.

1.6 Thesis outline

Here we give a brief overview of the rest of this thesis. In Chapter 2, we discuss related work from the perspectives of both verification and systems techniques. Next, in

Chapter 3 we present the design of CIO-FSCQ. The design walks through each component mentioned above in detail. Chapter 4 discusses the implementation, including the runtime that makes the Coq implementation executable, and how programs interact with the file system with the UNIX system call interface. In Chapter 5 we present an evaluation of CIO-FSCQ showing that it achieves better performance than FSCQ in a concurrent workload, and demonstrates I/O concurrency by executing system calls while disk reads complete in the background. In Chapter 6 we discuss some limitations with the current design that are good opportunities for future work. Finally, in Chapter 7 we conclude.

Chapter 2

Related Work

As far as we know CIO-FSCQ is the first verified I/O-concurrent file systems. This thesis builds on previous research on I/O concurrency in systems and verification of concurrent software. There are several existing verified file systems [1, 3, 23, 33], which are all sequential. While we re-use some aspects of FSCQ in order to extend it, the design of CIO-FSCQ largely involved techniques orthogonal to those introduced by existing verified file systems.

2.1 I/O concurrency in systems

I/O concurrency dates back to the early days of operating systems (*e.g.*, Stretch [5], CTSS [7], THE [9]). These systems use I/O concurrency to provide time sharing: when one user is thinking, the operating system runs the process of another user. Similarly, when one user's process blocks on I/O, the operating system runs another user's process. CIO-FSCQ uses the same idea inside the file system to talk to the disk.

Previous file systems have used a more complicated plan for issuing I/O efficiently. The file system uses multiple threads: threads take a lock on inodes or files before issuing I/O. The locks prevent other file-system threads from observing partial results and avoids the race described in Section 1.1. Reasoning about locking is complicated. Optimistic system calls avoid the need for locks, simplifying reasoning, but at the

cost of potential retries.

The idea of returning a signal indicating a cache miss from optimistic system calls was inspired by UNIX's non-blocking I/O (*e.g.*, the `O_NONBLOCK` option to POSIX's `read` function [21]), which returns the `EAGAIN` error code when data is not yet available. The intention is that the caller can retry an I/O operation until it completes. Non-blocking I/O in UNIX, however, is targeted for reading from network connections in high-performance network servers in the style of Flash [30]. UNIX file systems do not support non-blocking I/O via returning `EAGAIN` because file systems have no support for roll back. Instead, file systems efficiently support concurrency by using multiple kernel file-system threads, requiring careful synchronization in file-system code.

Work from NetApp [8] on Waffinity incrementally parallelized an existing legacy code base. While this general approach resembles the spirit of CIO-FSCQ adding concurrency to FSCQ, the techniques used differ significantly. In particular, Waffinity identifies operations that are safe to run in parallel and uses locks to ensure that only safe operations can be executed concurrently. For example, reads to disjoint regions of a file are allowed to proceed in parallel, whereas operations that modify metadata acquire a reader-writer lock. Such a structure requires less intrusive modification than modifying the entire codebase, but would be difficult to verify. In contrast, CIO-FSCQ does not examine the internal structure of FSCQ (and cannot even assume that disjoint regions of a file do not overlap in some way) and adds concurrency in an always-correct manner. Furthermore, we prove that the approach is correct, adding assurance that the concurrency introduced is safe under only our assumptions about file-system usage.

2.2 Concurrent verification

Research in verification has two main themes that are relevant to this thesis: research on verified concurrent systems and research on designing logics for verifying concurrent software.

2.2.1 Verified systems

Most certified systems software today is not concurrent (*e.g.*, seL4 [24], XMHF [36], CertiKOS [14] until recent work, IronClad [16], CompCert [26], and FSCQ [3]). We discuss here a few systems that have demonstrated verified concurrent reasoning.

Microsoft Research developed CIVL [18], a language and verifier within BOOGIE, and used the system to verify a concurrent garbage collector. The proof is organized into several layers of abstraction. The proof includes a top-level functional specification, showing that the garbage collector operates atomically as far as threads observe. The verification relies on proving atomicity of operations in each abstraction layer. The annotation burden appears to be quite high, with invariants that required deep knowledge of how the code works and why it is correct. Boogie is based on Z3, an SMT solver; as with other work using automated theorem proving, scaling proves a challenge: interference checking in CIVL can degenerate to an unscalable quadratic check between all pairs of actions if isolation is not carefully established.

Microsoft Research has also attempted an ambitious goal of verifying the HyperV hypervisor using VCC [25]. This project appears to have stopped with around 20% of the code verified. The approach taken was to verify existing code with annotations and specifications for internal functions, which appears to be difficult compared to writing the software with verification in mind from the beginning.

Recently the authors of CertiKOS extended their verified operating system to support concurrency [15]. The accomplishment is impressive but required heavyweight techniques and many abstraction layers. It is unclear how to apply the deep specification approach used in CertiKOS to other systems. The goals of CIO-FSCQ differ in that we focus on a limited form of concurrency (CertiKOS is verified against a model of a multicore x86 processor) and aim for much lower verification effort by re-using the sequential verification for the bulk of the system.

Verified distributed systems include projects such as Verdi [39], used to verify linearizability of the Raft consensus protocol [28], and IronFleet [17]. Distributed system verification faces some of the same challenges as concurrent verification: pro-

cesses on different machines interleave execution in a similar manner to threads on a single machine. However, distributed systems restrict interactions between the processes to message passing (sending and receiving requests over the network), whereas application using a file system interact through shared memory (*e.g.*, a shared buffer cache).

2.2.2 Logics

While there are few examples of verified concurrent systems software, there are many examples of logics for verifying concurrent programs. A few classic proposals [20, 22, 29] introduced several ideas that continue to be influential, much as Hoare logic [19] and the more recent separation logic [31] continue to form the basis for much sequential verification even today.

In concurrent verification logics, there is a great deal of modern work [10–13, 27, 32, 34, 35]. These formalisms address a variety of problems, but all fall short of verifying realistic programs. Many are mechanized within a proof assistant and include proofs of example programs, but these programs are generally of theoretical interest as verification challenges (*e.g.*, proving the Trieber lock-free stack correct). Extending a logic to include verification of larger, realistic programs introduces engineering challenges that themselves have a large design space.

There are several directions that concurrent logics push against. One is the definition of concurrency itself: the most basic setting for concurrent execution is verifying several programs all running concurrently from start to finish. This is a fine theoretical setting and includes many of the challenges of concurrent verification, but does not reflect how concurrent software is written and executes. Modeling and reasoning about dynamic threads and message passing introduces further complexity. In our setting of the file system, the execution model looks like a client-server architecture: each system call is a handler and is called concurrently by the outside world (the Linux VFS layer for a standard file system). Orthogonal to how concurrency arises is the granularity of thread interactions: these can range from cooperative semantics, the easiest to reason about, to true multicore concurrency, where execution of instructions

is simultaneous and issues such as data races and weak memory models arise. Finally, modular verification is a challenging target even for a logic, independent of implementations: it is desirable for the logic to allow verifying programs in isolation and re-using this proof when the program is composed with concurrent threads or used as a primitive in a larger program. Concurrency makes reasoning about abstraction layers much more challenging than sequential verification.

There are some common themes in verification frameworks highlighted by the Views metatheory [10]: concurrent verification is about threads reasoning *locally*, assuming some *limited interference* from other threads, while simultaneously manipulating state according to some *protocol*. Furthermore, concurrent reasoning seems to always require some form of *abstract state* (sometimes described as virtual or ghost state), to capture properties about thread interactions that are not present at runtime. Finally, some facts are *stable* under interference from other threads. Especially for fine-grained concurrency, proving stability under the protocol is a common proof obligation that some frameworks ameliorate.

We've experimented with concurrency by implementing parts of concurrent separation logic (CSL) [2] and local rely guarantee (LRG) [12], both of which incorporate separation logic, an idea we found especially useful in the sequential file system. CSL provides local reasoning by separating memory into disjoint resources and specifying a protocol where threads lock resources (acquiring them) and then release them. LRG more generally allows threads to be proven with respect to any rely condition (specifying interference) and guarantee (specifying the local rules) — threads with compatible rely/guarantee conditions can be composed. These two approaches are in some ways opposite extremes: in CSL composition is trivial but sharing resources is almost expressly forbidden, whereas in rely-guarantee sharing is expressive but composition requires the conditions to line up correctly. The CCL concurrency framework is a simple variant of rely-guarantee reasoning, with a single protocol rather than independent rely-guarantee conditions.

Chapter 3

Design

CIO-FSCQ consists of five components, as organized in Figure 3-1: a logic, CCL (Section 3.1); a cache, CIO-Cache, written using CCL (Section 3.2); a translator from verified sequential code to verified optimistic system calls (Section 3.3); a wrapper for optimistic system calls producing the implementation of the CIO-FSCQ file system (Section 3.4); and finally verification of this implementation against a protocol (Section 3.5).

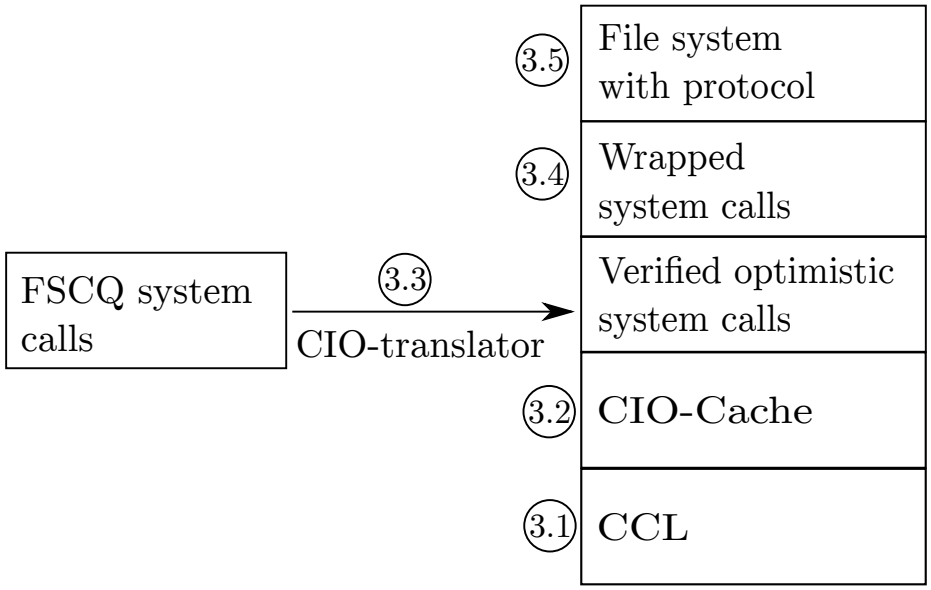


Figure 3-1: Components of CIO-FSCQ. Circled references are section numbers.

function signature	description
`Get<T>(var: variable T) : T`	Return the value of variable `var`
`Set<T>(var: variable T, v: T)`	Set variable `var` to `v`
`GhostUpdate(update: S -> S)`	Update ghost state `s` to `update(s)`

Table 3.1: Primitive memory and ghost state operations provided by CCL. Ghost state collectively has type `S` throughout. The type `variable T` refers to a variable holding a `T`; the set of variables and types is fixed and determined separately.

3.1 Cooperative Concurrency Logic

At the base of CIO-FSCQ is Cooperative Concurrency Logic (CCL). CCL consists of three parts: a small programming language for disk operations and cooperative multithreading, a semantics to define the meaning of the operations in the programming language, and a specification language that allows the programmer to state Hoare-style pre- and post-conditions. To be able to prove that an implementation meets its specification, CCL supports defining a protocol to put restrictions on the execution of all threads. The following four subsections explain each part in more detail.

3.1.1 Programs

Programs manipulate a disk, a memory, and *ghost state*. The memory is typed: each location in memory has an associated type, and programs are guaranteed to use the memory in a well-typed manner. The ghost state behaves much like the memory, but it cannot influence program behavior. Programs manipulate ghost state to ease verification by explicitly indicating how the abstract state evolves. At runtime, ghost state is neither stored in memory nor updated. Memory and ghost state are manipulated with the operations in Table 3.1.

Programs can retrieve a memory cell with `Get`; the type parameter determines what the expected return type is. The type `variable T` is a pointer into the memory; by construction, using a dependent type in Gallina, pointers only type-check if they point to valid variables, with the correct type. Similarly when memory is updated with `Set`, the value necessarily has the correct type. Ghost state is updated atomically and

```
Definition AcquireLock :=
  l ← Get(mLock);
  if l == Unlocked then
    tid ← GetTID();
    Set(mLock, Locked);
    GhostUpdate (λ _ ⇒ Owned tid);
    Ret
  else
    Yield();
    AcquireLock()

Definition ReleaseLock :=
  Set(mLock, Unlocked);
  GhostUpdate(λ _ ⇒ Free)
```

Figure 3-2: Code for acquiring and releasing lock.

with access to the entire ghost state using `GhostUpdate`. The ghost state of type `S` is represented with the same type as the memory, but with a different set of typed pointers. While ghost state can be updated based on both its current value and outside information, including variables by referencing them in a closure passed to `GhostUpdate`, there is no operation that can read ghost state and use it to affect program execution, so that at runtime it is safe to not execute ghost updates.

As an example of using the typed memory and ghost updates, consider using a lock. Suppose the program had pre-allocated a lock in memory, referred to as `mLock`. The type of the lock will be a simple boolean, represented as `Inductive LockFlag := Locked | Unlocked`. However, the ghost state for the lock will be a more expressive type `Inductive LockState := Owned TID | Free`, also capturing the owning thread. For simplicity of presentation, suppose that the entire ghost state consists of only a `S = LockState`, the state for `mLock`. We give code to acquire and release this lock, with the appropriate ghost updates, in Figure 3-2. Such a lock is not needed in CIO-FSCQ, but we verified a similar example while developing CCL. In a larger program, verification can take advantage of the fact that this implementation tracks not just the binary state of the lock but also the owning thread, despite the memory state not capturing this information.

function signature	description
`BeginRead(a: addr)`	Schedule a disk fetch of address `a`
`WaitForRead(a: addr) : block`	Get data from a scheduled read of address `a`
`Write(a: addr, v: block)`	Write `v` to address `a`

Table 3.2: Disk operations provided by CCL.

The disk model consists of the representation of the disk itself and the operations that act on the disk in CCL. The operations are given in Table 3.2. Addresses are written as `addr` and refer to block indices on the disk: we use natural numbers to represent these indices. Disk blocks are fixed to be 4KB of binary data. Writes are straightforward: they update the block at an address. We assume writes are visible immediately since real disk hardware guarantees this: writes go into a buffer within the disk and reads check this buffer. Reads are asynchronous to support I/O concurrency: a disk read is split into a call to `BeginRead` to signal the intention to read a block and `WaitForRead` to block until it completes and retrieve the result. The disk representation tracks in-flight reads, which enable the model to treat issuing `WaitForRead` without a prior `BeginRead` as an error.

To give some intuition for asynchronous reads, we give some examples of correct and incorrect code in Figure 3-3. `SynchronousRead` and `Copy` assume are no concurrent reads for the same address pending. `IncorrectBegin` is always incorrect, since the second read is guaranteed to fail; this is a conservative model of the physical disk, which permits concurrent reads. `IncorrectDataRace` always fails since writes require that there are no pending reads to the address being written, and is considered an error by the semantics since in practice CIO-Cache avoid all data races.

FSCQ uses checksum logging for efficiency. Checksums logging records a hash of data in the log to track whether the data on disk matches what is expected, which is useful to detect when a crash interrupted writing the log to disk. However, verification using hashes requires some technique to avoid considering the possibility of a hash collision; collisions always thwart the intention of using hashes but in practice are highly unlikely. The approach taken in FSCQ is to add hashing as a primitive to

```
Definition SynchronousRead(a: addr) : block :=
  BeginRead(a);
  v ← WaitForRead(a);
  Ret v

Definition Copy(a: addr, a': addr) :=
  v ← SynchronousRead(a);
  Write(a', v)

Definition IncorrectBegin(a: addr) : block :=
  BeginRead(a);
  (* cannot have two pending reads for the same address *)
  BeginRead(a);
  v ← WaitForRead(a);
  Ret v

Definition IncorrectDataRace(a: addr, v: block) :=
  BeginRead(a);
  (* cannot write to an address with a pending read *)
  Write(a, v)
```

Figure 3-3: Some short usage examples for disk code. See main text for discussion of how the correct examples work and why the incorrect programs trigger errors.

function signature	description
`Hash(sz: nat, buf: bytes sz) : bytes 32`	Compute the hash of the data in `buf`, looping infinitely if this results in a hash collision.

Table 3.3: Hash operation provided by CCL.

function signature	description
`GetTID() : TID`	Get the thread identifier for the current thread
`Yield()`	Let other threads execute

Table 3.4: Cooperative concurrency operations provided by CCL.

the programming language and model hash collisions within a given execution as an infinite loop [38]: conceptually, the system tracks all values hashed, and if two different values hash to the same result, the program does not terminate and thus the partial correctness specifications say nothing about it. At runtime collisions are not detected, but finding a collision is unlikely so the proof gives reasonable confidence in the behavior of the program. In order to translate FSCQ's programs accurately, we add hashes to CCL with the same semantics as in FSCQ. For completeness, the signature of the hash operation is given in Table 3.3.

Finally, CCL provides two operations for cooperative concurrency, given in Table 3.4. Threads can access a unique thread identifier with `GetTID`, and can invoke `Yield` to let other threads execute before returning control to the current thread. For I/O concurrency, threads should issue `BeginRead` and then yield until the disk read completes; this desirable pattern is shown in Figure 3-4. While the `AsyncRead` does not have a way of guaranteeing after yielding that the disk read is complete and that `WaitForRead` will not block, the runtime for CCL programs (described in more detail in Section 4.3) makes this pattern efficient. In particular, threads are scheduled such that pending I/O issued before a yield has completed before control returns to the thread. In the `SynchronousRead` example of Figure 3-3, there is no yield between starting the I/O and requesting the data; threads are cooperatively scheduled so the runtime cannot let another thread run while waiting for the disk.

```
Definition AsyncRead(a: addr) : block :=
  BeginRead(a);
  Yield();
  v ← WaitForRead(a);
  Ret v
```

Figure 3-4: Example of issuing a read asynchronously. For this to work, threads must coordinate so that other threads do not finish the pending read or write to `a` during the yield.

3.1.2 Execution Semantics

In CCL, a program is a sequence of the above operations. CCL uses the standard monadic combinators `Ret` and `Bind` [37] for sequencing operations. The type of programs producing values of type `A` is `cprog A`. The `Bind` combinator enables programs to include arbitrary code within Gallina while referencing the above I/O operations. We use this *shallow embedding* into Gallina so that programs do not need to model control flow and local memory (only I/O operations and shared memory), getting these features for free from Gallina.

To describe the disk operations' semantics, we return to the disk model by describing how disk state is represented. A disk is at its most basic a set of blocks, which we fix at 4KB. For verification purposes it is convenient to abstract away the fact that a disk is an array of blocks and represent it as a mapping from addresses to blocks. Furthermore, CCL abstracts away the size of the disk and allows it to map only some of the addresses. This basic disk representation is a partial map from addresses (we use `addr` to refer to an address, but the concrete representation in the prototype is an unbounded natural number) to blocks (these are chunks of 4KB worth of data). For convenience of verifying that asynchronous reads are used correctly, CCL augments this state with some additional per-address information. Specifically, the semantics track if any thread is reading each address. The semantics makes it an error to issue `BeginRead` when someone else is reading, to issue `WaitForRead` without someone having started the read, and to attempt to write to an address while another thread is reading it. These semantics are restrictive compared to real hardware (concurrent

reads are not races) but our usage in CIO-FSCQ follows these more restrictive rules since the cache tracks pending reads and avoids concurrent reads and writes to the same address.

More formally, the semantics is expressed a big-step relation: it relates initial states and programs to final states and return values. A special final state identifies error executions, which include, for example, attempts to write out-of-bounds addresses. All verified programs prove that they never cause an error if their preconditions are satisfied. The semantics can be understood by understanding how we model each primitive procedure above; the semantics of `Bind` simply chains together procedures, producing an error if any intermediate computation does so. There are some cases where there is no execution from some state: this is called stuck execution and is produced in particular by an infinite loop. CCL specifications are always written in a *partial correctness* style, which only talks about behavior when a program terminates. Non-terminating programs are rare in CIO-FSCQ, but the retry loop used for each system call is a notable example (see Section 3.4, and in particular the loop in Figure 3-13).

The execution states of programs include the disk, hashed values, memory, and ghost state. Furthermore, states track both an "initial" and current ghost state: the initial ghost state gives the ghost state from the last yield point, which is needed to ensure yields only succeed if the program has followed the protocol since its last yield point. The hashed values are used to detect hash collisions in the semantics. We will refer to the tuple of state with $\{d; h; m; s_0; s\}$, with the variables consistently representing disk, hashed values, memory, initial ghost state, and current ghost state.

The disk state is represented as a partial map from addresses to a combination of a block and a bit to track pending reads: in Gallina it has the type `addr -> option (block * bool)`, where the boolean is true if there is a pending read at that address. Addresses outside of the range of the disk can simply map to `None`.

Figure 3-5 presents the semantics of the primitive operations in CCL. We defer discussion of `Yield` and its semantics to Section 3.1.3. As we described for the API, programs have a fixed set of variables and ghost state collectively has type `S`. The

$$
\begin{aligned}
Get(v : variable(T)) &\mapsto \{d; h; m; s_0; s\} + get(m, v) \\
Set(v : variable(T), val : T) &\mapsto \{d; h; set(m, v, val); s_0; s\} \\
GhostUpdate(update : S \to S) &\mapsto \{d; h; m; s_0; update(s)\} \\
BeginRead(a : addr) &\mapsto \{d[a \mapsto (b_0, true)]; h; m; s_0; s\} \\
&\quad \text{if } d[a] = (b_0, false) \\
WaitForRead(a : addr, b : block) &\mapsto \{d[a \mapsto (b_0, false)]; h; m; s_0; s\} + b_0 \\
&\quad \text{if } d[a] = (b_0, true) \\
Write(a : addr, b : block) &\mapsto \{d[a \mapsto (b, false); h; m; s_0; s]\} \\
&\quad \text{if } d[a] = (b_0, false) \\
Hash(sz : nat, buf : bytes(sz)) &\mapsto \{d; h \cup \{buf\}; m; s_0; s\} + \text{hash}(buf) \\
&\quad \text{if } buf \text{ does not collide with } h \\
GetTID() &\mapsto \{d; h; m; s_0; s\} + tid
\end{aligned}
$$

Figure 3-5: Execution semantics for each primitive operation. The presentation above gives the transitions for each operation. We write $p \mapsto \sigma' + r$ when program p steps to a new state σ' and returns value r, omitting the return value if it is of type `unit`. The starting state $\{d; h; m; s_0; s\}$ is left implicit, as is the current thread ID tid (as referenced by the rule for `GetTID()`).

$$
\begin{aligned}
BeginRead(a) &\mapsto error \\
&\quad \text{if } d[a] \text{ undef} \vee \\
&\quad\quad d[a] = (b_0, true) \\
WaitForRead(a) &\mapsto error \\
&\quad \text{if } d[a] \text{ undef} \vee \\
&\quad\quad d[a] = (b_0, false) \\
Write(a, b) &\mapsto error \\
&\quad \text{if } d[a] \text{ undef} \vee \\
&\quad\quad d[a] = (b_0, true)
\end{aligned}
$$

Figure 3-6: Execution semantics for error cases of primitive operations. The starting state $\{d; h; m; s_0; s\}$ is left implicit.

type `variable(T)` above internally depends on the set of memory variables, such that well-typed variable references always point to valid memory addresses.

In addition to the above rules, CCL distinguishes between normal executions and those that result in an error. Explicitly executing to an error lets specifications talk about the absence of error executions. The cases where each primitive operation results in an error are given in Figure 3-6. When the semantics chains operations via `Bind`, errors halt the entire execution. The error rules for `BeginRead`, `WaitForRead`, and `Write` ensure that every usage of these operations can always execute, either to a next state or to an error; without this property, there would be states where these primitives' execution would get stuck and appear indistinguishable from an infinite loop. Note that hashes cause an infinite loop in the semantics when two colliding values are hashed: specifications intentionally ignore this situation, which is in practice unlikely for good hash functions (that is, encountering a collision at runtime is unlikely, even if some collision must exist).

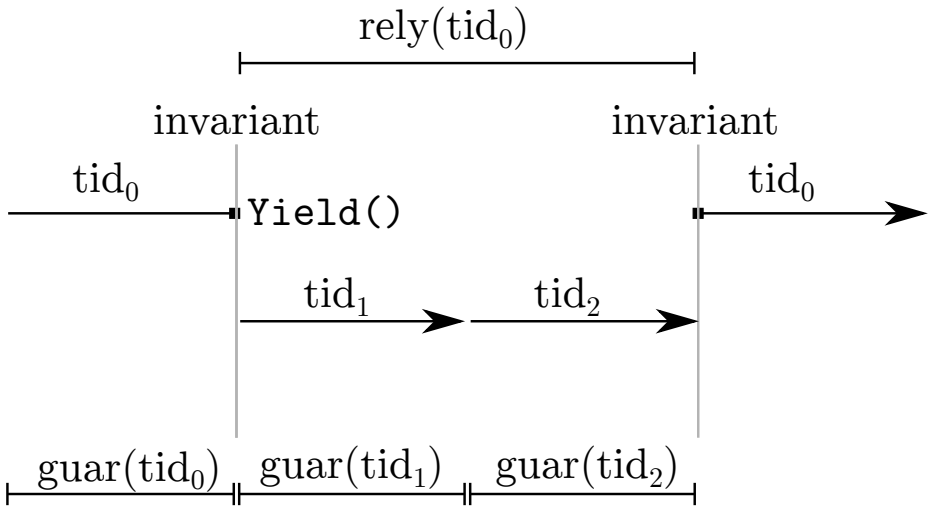


Figure 3-7: Example execution showing where the protocol applies in a valid execution. The thread tid_0 issues a `Yield`, letting threads tid_1 and tid_2 run. It must guarantee that it has respected $guar(tid_0)$ and the invariant, and in turn knows that $rely(tid_0)$ and the invariant hold.

3.1.3 Protocols

The execution semantics takes as a parameter a *protocol*. The protocol states what guarantees thread provide across yields. For example in CIO-FSCQ, the most interesting protocol we defined is that each thread operates in its own directory. This protocol, which we describe in detail in Section 3.5, allows us to re-use FSCQ's specifications after retrying even when other threads run, since they cannot interfere with each system call's precondition.

A protocol consists of two parts: an *invariant*, which governs what holds at yield points, and a *guarantee*, which governs allowed transitions between yield points. An example execution showing what the protocol governs is given in Figure 3-7. Invariants are the most basic rule threads follow. For the directory isolation protocol, the invariant merely links the disk and memory state to a directory tree abstraction stored as ghost state. The guarantee gives a relational rule, enabling protocols to express notions such as read-only or append-only. The guarantee is parameterized by a thread ID, to enable distinguishing threads and giving threads special privileges. The directory isolation protocol has a global set of permissions denoting which threads own which directories. The isolation part of the protocol is expressed as a guarantee condi-

$$
\begin{aligned}
Yield() \mapsto \{d'; h'; m'; s'; s'\} & \\
\quad \text{whenever } & invariant(d, m, s) \wedge guar(tid, s_0, s) \wedge \\
& invariant(d', m', s') \wedge rely(tid, s, s') \\
& \wedge h \subseteq h' \\
Yield(a) \mapsto error & \\
\quad \text{if } & \neg invariant(d, m, s) \vee \\
& \neg guar(s_0, s)
\end{aligned}
$$

Figure 3-8: Execution semantics for the `Yield` operation, starting in state $\{d; h; m; s_0; s\}$.

tion: between yields, each thread can only change directories that are world-writable or that it has exclusive ownership of.

While the guarantee condition is in principle more general than an invariant, for properties that can be expressed as an invariant it is often convenient to use the invariant rather than the guarantee condition. In addition, separating the protocol into these two parts allows us to write the invariant over the full system state while the guarantee condition is expressed in terms of only the ghost state. When, for example, memory variables are relevant to a guarantee condition, they can be mirrored in the ghost state and required to match via the invariant. The directory isolation protocol requires that an invariant covering the disk and memory to establish the directory tree abstraction but has a guarantee condition written solely in terms of this abstraction.

The protocol is used to determine the semantics of `Yield`. The formal semantics are given in Figure 3-8. In the figure, we use $rely(tid, s, s')$ for the rely condition, which is defined as an arbitrary number of steps between s and s' that respect the guarantee condition, for threads other than tid. As the rule shows, the yield instruction requires that the invariant holds at call time, so that other threads observe a consistent state; if it is violated, yielding results in an error. In turn, the semantics promise that when the yield completes, the invariant will still hold. In addition, the yield requires each thread to prove that between its last yield and the current state, it respected the

guarantee condition; if the thread does not do so, yield again results in an error. In turn, the semantics guarantees that other threads did so as well, using *rely* to express the behavior of other threads in terms of the guarantee condition.

The semantics of yield abstract over what other threads may do, promising only that they follow the protocol. As long as all running threads follow the protocol, this abstraction is valid. CCL is cooperative in that threads are guaranteed to execute sequentially until they choose to yield; this shows up in the semantics as the protocol appearing only in the rule for yields, whereas otherwise the behavior of other threads does not influence program behavior.

Hashing is incorporated into the yield semantics by promising that the set of hashed values only increases. This is sufficient for any use of hashing: more hashed values only increases the number of inputs that will trigger a hash collision. In addition, the semantics guarantee that programs can only increase the set of hashed values, so other threads are guaranteed to satisfy this condition.

3.1.4 Specifications

Specifications in CCL are written in a standard Hoare triple style with a partial-correctness interpretation, as mentioned above: a program is associated with a precondition and postcondition, and a proof of a specification expresses that if the program is run in a state satisfying the precondition and terminates, the final state will satisfy the postcondition. Furthermore, we define correctness such that any verified program (regardless of specification) does not result in an execution error if the precondition is satisfied.

3.2 CIO-Cache

We have written a program in CCL we call CIO-Cache, a concurrent buffer cache with support for transactional writes. The cache provides four operations, listed in Table 3.5. Of particular note is `CacheRead`, whose pseudocode implementation we give in Figure 3-9. `CacheRead` is unusual for a read operation in that it may fail in

```
Definition CacheRead(a: addr) : option value :=
  c ← Get(mCache);
  match cache_get c a with
  | Present v ⇒ Ret (Some v)
  | Missing ⇒ BeginRead(a);
              Set(mCache, set_pending c a));
              Ret None
  | Pending ⇒ v ← WaitForRead(a);
              Set(mCache, fill_val c a v);
              Ret (Some v)
  end.
```

Figure 3-9: Pseudocode for `CacheRead`. Note that a miss followed by a read for the same address will trigger the pending case and read the value from disk. For I/O concurrency, the caller should yield after a miss but before reading again.

the case of a cache miss, returning `None`. This signals to the caller that the read missed in the cache and that the cache has issued an I/O to read the data from disk; the caller is expected to yield (to allow I/O concurrency) and retry the read. Upon retrying, the cache will call `WaitForRead` on the pending disk read and block, this time succeeding.

Writes in CIO-Cache are transactional: `CacheWrite` buffers the write separately from the cache until either a commit (`CacheCommit`), which makes the writes part of the cache, or an abort (`CacheAbort`), which discards all the buffered writes since the last commit. The specifications for `CacheCommit` and `CacheAbort` are shown in Figure 3.2. While somewhat verbose, these specifications capture the transactional behavior of the cache on its state, namely `vdisk s` and `vdisk_committed s`. The current implementation requires the specifications to assert that other parts of the memory and ghost state are not modified; see Section 4.1 for a discussion of this modularity issue.

CIO-Cache uses some part of the memory to hold the cache, and has some associated ghost state we call `vdisk_committed` and `vdisk`, which represent the committed and current state of the virtual disk exposed by the cache respectively. The virtual disk is a simple abstraction with a 4KB block per address. When cache reads succeed,

```
Theorem CacheCommit_ok:
  SPEC tid ⊢
  {<
   PRE d m s₀ s:
     CacheInvariant(d, m, s)
   POST d' m' s₀' s':
       CacheInvariant(d', m', s') ∧
       vdisk s' = vdisk s ∧
       (* the committed disk is updated *)
       vdisk_committed s' = vdisk s ∧
       modified cache_vars m m' ∧
       modified cache_ghost s s' ∧
       s₀' = s₀
  >} CacheCommit.

Theorem CacheAbort_ok:
  SPEC tid ⊢
  {<
   PRE d m s₀ s:
     CacheInvariant(d, m, s)
   POST d' m' s₀' s':
       CacheInvariant(d', m', s') ∧
       (* the current disk is reverted *)
       vdisk s' = vdisk_committed s ∧
       vdisk_committed s' = vdisk_committed s ∧
       modified cache_vars m m' ∧
       modified cache_ghost s s' ∧
       s₀' = s₀
  >} CacheAbort.
```

Figure 3-10: Specifications for `CacheCommit` and `CacheAbort`.

function signature	description
`CacheRead(a: addr) : option value`	Read address `a` from the cache; if not present in the cache, schedule a disk read and return `None`
`CacheWrite(a: addr, v: block)`	Write `v` to address `a`
`CacheCommit()`	Commit writes since the last abort or commit
`CacheAbort()`	Abort writes since the last abort or commit

Table 3.5: Operations provided by CIO-Cache. The type `option T` is either `Some t` with `t: T`, or is `None`.

they return immediately, but they may miss and return an error instead. Internally, the cache starts reading from disk with `BeginRead` when a read misses, recording that a read is pending. When a second read for the same address finds the pending marker, it completes the pending read with `WaitForRead`, returning a result and filling the cache entry. The specifications for each cache operation specify behavior in terms of modifications to these ghost variables. Each operation also takes care to update the ghost variables to reflect changes to the abstraction.

Recall that the semantics of programs are governed by a protocol, as explained in Section 3.1.3. The protocol specifically governs behavior when a program yields. However, the cache does not include internal yields, leaving these to the caller (for example, if `CacheRead` misses, it does not yield after scheduling the disk read). As such, each specification requires and preserves some invariants internal to the cache but does not require the protocol's invariant. This is an appropriate specification: optimistic system calls will not guarantee global invariants while running on the cache. The cache also explicitly states that it does not modify variables other than those controlled by the cache, so that file-system code can reason about its own memory.

3.3 CIO-translator

CIO-Cache exposes read and write operations that resemble sequential code operating on a synchronous disk. We formalize this intuition by translating sequential programs written using Crash Hoare Logic (CHL) from FSCQ to the cache operations using CIO-translator. The translated code as a whole returns an error if any read misses. When `CacheRead` fails and returns `None` (signaling a cache miss), the translator handles this by in turn by returning `None` early in the translated code. Thus programs of type `prog A` are translated into concurrent programs of type `cprog (option A)`.

In addition to this straightforward compilation, the translator also translates sequential CHL specifications regarding the disk to concurrent specifications regarding the virtual disk ghost variable. Simplified code for the specification translation is shown in Figure 3-11. In this specification translation we drop the CHL crash condition, since CCL does not model crashes. We must also thread the cache assertions that only cache variables are modified, so users of the specification can rely on the translated code not modifying other variables. The translator is parameterized over the memory variables, ghost state and a global protocol in essentially the same was as the cache.

The goal of the translator is to preserve sequential specifications: verified code when translated should satisfy the translated specification. We show a statement of this correctness property in Figure 3-12. The intuition behind the proof of this statement is that the translated, isolated code behaves in the same way as the original code. We prove this via simulation: every execution of the compiled code has an equivalent execution of the sequential code. We state the property in three parts, covering the three possible outcomes of running the compiled code: the result may be a successful run, the cache may have missed at some point, or the code may have failed due to an out-of-bounds write. For successful runs, we show that the resulting virtual disk is the same as the disk from some execution of the sequential code (this is the optimistic case). When a read misses in the cache, we merely show that the code followed the cache protocol, since the system call wrapper will abort any

```
Definition OptimisticSpec A (spec: SeqSpec A) : ConcurrentSpec (option A) :=
  (* the crash condition in spec is unused *)
  let (seq_pre, seq_post, _seq_crash) := spec in
  λ d m s₀ s ⇒
  {| precondition :=
       CacheInvariant d m s ∧
       seq_pre (vdisk s)
     postcondition := λ d' m' s₀' s' r ⇒
       CacheInvariant d' m' s' ∧
       match r with
       | Some r ⇒ seq_post r (vdisk s')
       | None ⇒ ⊤
       end ∧
       modified cache_vars m m' ∧
       modified cache_ghost s s' ∧
       s₀' = s₀ |}
```

Figure 3-11: Simplified code for the translation from a sequential specification to an appropriate specification for its optimistic, translated version. The concurrent specification is a function from the initial state so that the postcondition can refer to the initial state; sequential specifications do not do this.

partial updates anyway and restore the committed virtual disk. When the concurrent code fails, we show the sequential code would have failed as well — when using the simulation in the context of verified programs we will rule this possibility out.

To use this simulation argument to argue specifications are translated correctly, we prove that if the sequential code satisfies some specification `spec`, the translated, isolated code satisfies a translated specification `spec'`, where `spec'` must refer to the virtual disk whenever `spec` refers to the physical disk. In this proof, we rule out the

```
Definition translate (p: prog A) : cprog (option A) := ...

Theorem translator_correct : ∀ A (p: prog A) (spec: SeqSpec A),
  prog_ok p spec →
  cprog_ok (translate p) (OptimisticSpec spec).
```

Figure 3-12: The function signature for the translator and its correctness property. Translation preserves specifications, after they are translated by `OptimisticSpec`.

case of the concurrent code failing by using the fact that the sequential code would also fail and the assumption in both cases that the precondition of the sequential code was satisfied. If the optimistic system call misses in the cache, we guarantee some consistency properties of the variables (this includes the fact that the committed disk does not change), but cannot guarantee the specification's postcondition.

3.4 Wrapped system calls

We designed a generic wrapper that turns an FSCQ call into an optimistic system call. In the error case, the wrapper rolls back the disk so other threads see a clean state between system calls, and then yields to let other threads run concurrently. After yield returns, the wrapper retries the system call. The retry is likely to succeed, but if it fails the wrapper tries again.

Turning FSCQ systems into optimistic systems calls requires dealing with FSCQ's `memstate`, which is passed to and returned from every FSCQ system call. This in-memory state is important to thread through FSCQ system calls for correctness. The concurrent system calls cannot take this approach: when a system call yields, it must observe the new memory state produced by other system calls that ran in the meantime. In wrapping FSCQ system calls, CIO-FSCQ moves this memory state into a CCL memory variable, updating it between system calls to make it visible to other threads. The wrapped optimistic system calls are collectively the file-system implementation in CIO-FSCQ.

For verification purposes, CIO-FSCQ programs manipulate *ghost state* representing the abstractions involved in the code. In the case of a file system, the main ghost state is a directory tree. Each top-level system call requires an appropriate call to `GhostUpdate` to update this abstract directory tree (other than read-only operations). FSCQ already includes a representation of directory trees as an inductive, recursive datatype, which its top-level specifications refer to. We store a copy of this directory tree in ghost state for verification purposes. To update it in CIO-FSCQ, we add a call to `GhostUpdate` in each system call with an appropriate update function

```
Definition file_get_attr(inum) :=
  mem_state ← Get(mMemState);
  r ← optimistic_file_get_attr(mem_state, inum);
  match r with
  | Some (attr, mem_state') ⇒ Set(mMemState, mem_state');
                              CacheCommit();
                              GhostUpdate(id);
                              Ret attr
  | None ⇒ CacheAbort();
           Yield();
           file_get_attr(inum)
  end.
```

Figure 3-13: Example of wrapping the automatically generated `optimistic_file_get_attr` to include modifications to mutable memory and a retry loop. Note that the structure of this code is identical for every system call, although other system calls will modify the abstract directory tree where `file_get_attr` calls `GhostUpdate(id)`.

implementing that operation, this time as a functional program on a directory tree. This code is largely copied from the specifications of FSCQ, which are already written in this style of functional updates.

An example of a complete system call built from its optimistic version is show in Figure 3-13. The figure shows `file_get_attr`, which is used to implement the `stat` system call. This is a read-only system call so no update to ghost state is needed; the code redundantly calls `GhostUpdate` to show where the abstract state would be updated if necessary. The implementation uses a generic wrapper function rather than duplicating this pattern for each system call.

3.5 File-system protocols

The compiled file-system operations retry when they fail, but only after yielding to other threads. This yield is clearly safe since after rolling back the disk, the state is identical to the initial state. However, starting the system call again must satisfy the system call precondition to prove anything about the result; that is, the precondition

should be *stable* under interference. We do so by specifying a file-system protocol and proving each precondition remains true under interference allowed by the protocol. For example, if threads operate in disjoint directories, it is safe to retry creating a file since no other thread will remove the containing directory in the meantime, a fact we prove to verify the `create` system call. We also prove each system call obeys the protocol.

We wrote two protocols for the file system and proved specifications for the CIO-FSCQ system calls under each: the first one is a read-only file system, supporting only read-only system calls; the second one, which we call *directory isolation*, partitions the file systems into subtrees and assigns each subtree to a different thread. Both protocols share a common abstract structure, built on top of the cache. As mentioned above, FSCQ's mutable memory `memstate` is stored in a CCL variable. The ghost state includes the abstract directory tree. The invariant in the protocol connects this `memstate` and the cache's virtual disk to the abstract directory tree, re-using a predicate in all of FSCQ's top-level system call specifications.

The read-only file system has a simple guarantee condition, requiring that the directory abstraction does not change. The memory and disk state may still evolve (*e.g.*, the cache may change), but must represent the same logical tree. Under this protocol, the file system is relatively easy to verify since the abstract state does not evolve.

The directory-isolation protocol allows controlled modification to the file system. It reflects a common usage pattern for file systems: for example, users often operate in disjoint directories. When a given user runs several programs concurrently, they tend to operate on disjoint sets of files rather than using synchronization or relying on thread safety of the file system implementation. The directory-isolation protocol captures this common practice by defining ownership of directories. Each directory in the file system can be either shared, owned by a specific thread, or read-only. In addition, the protocol states some consistency properties for ownership: children of a node in the tree must be at least as restrictive as their parents. This avoids, for example, stating that a directory is read-only but that a file in it is shared. The

```
Theorem file_get_attr_ok: ∀ inum,
  SPEC tid ⊢
  {< pathname attr data,
   PRE d m s₀ s:
     Invariant(d, m, s) ∧
     find_subtree pathname (directory_tree s) = Some (File inum attr data) ∧
     Owner pathname = Owned tid ∧
     Guarantee(tid, s₀, s)
   POST d' m' s₀' s' r:
       Invariant(d', m', s') ∧
       Rely(tid, s, s') ∧
       r = attr ∧
       Guarantee(tid, s₀', s')
  >} file_get_attr inum.
```

Figure 3-14: Simplified specification for `file_get_attr`. The specification shows linearizability: the `Rely(tid, s, s')` indicates other threads may have run, following the protocol. However, the ownership requirement in the precondition guarantees that the file does not change and thus the returned attributes correspond to the original (and current) file.

semantics of the protocol, to be usable within CCL, are defined as a per-thread guarantee condition: for each file or directory in the file system, if some thread does not have permission to access it (it is either read-only or owned by a different thread), then that file or directory must remain unchanged.

For the directory isolation protocol, system calls that are read-only have straightforward specifications: the return values are guaranteed to be consistent with the original state, since the precondition and protocol together guarantee the files and directories being accessed are read-only to other threads. An example specification for `file_get_attr_ok` (which is read-only) is shown in Figure 3-14. System calls that modify the directory tree have *linearizable specifications*: first other threads run, modifying parts of the directory tree, then the system call modifies the tree atomically. Again, while other threads execute, the precondition and protocol together guarantee that the paths being accessed do not change.

Chapter 4

Implementation

4.1 Modularity for memory and ghost variables

CCL as described requires all memory variables and their types to be given ahead of time for each program. This is inconvenient for writing code written in several abstraction layers, where not all the variables required are known until the highest-level layer. For example, the CIO-Cache uses some memory variables, but the file system using the cache requires some additional ones. To work around this requirement, the cache is parameterized over a set of memory variables, with the requirement that users include variables for the cache in this set. When the file system uses the cache, it follows this requirement by including the appropriate variables. Furthermore, the cache's memory variables are all internal implementation details (in fact, only the cache's `vdisk` and `vdisk_committed` ghost variables are relevant to users), yet users must know about these variables to include them in the global memory.

A similar problem arises in defining the global protocol. CCL programs are verified with a particular global protocol. The cache is instead verified against a generic, caller-specified protocol; since it does not yield internally, the global protocol does not affect its semantics.

	Lines of code
Common components	
FSCQ Coq implementation and proof	72,000
Shared Haskell runtime	400
I/O concurrency-specific	
Coq implementation and proof	11,000
Concurrent Haskell runtime	730

Table 4.1: Lines of code in CIO-FSCQ.

4.2 Coq implementation and proofs

CIO-FSCQ is implemented largely in Coq. The number of lines of code in the implementation is summarized in Figure 4.1. The implementation has around 83,000 lines of Coq code, including implementation, specifications, and proof scripts. Of these, 11,000 are specific to the I/O concurrent FSCQ. The remaining 72,000 lines are for a recent version of the sequential FSCQ, which is needed for both its implementation and full proof. The concurrent Haskell runtime is comparable in size to the FSCQ runtime, which is 570 lines of code (much of which is duplicated in the concurrent runtime due to insufficient factoring). The extracted code from CIO-FSCQ comprises about 36,000 lines of automatically generated Haskell code.

4.3 Haskell runtime

Our implementation in Coq specifies and models I/O, but has no way of actually executing with a physical disk, or scheduling threads between yields. Following FSCQ, we run the file system by first using the native extraction feature of Coq to produce an analogous Haskell version of each system call. The Haskell program is actually a datatype, with constructors for the I/O operations, memory interactions, and concurrency primitives (in CCL, these are just `GetTID` and `Yield`), as well as sequencing in the form of `Bind` constructors. An interpreter written in Haskell takes this program and runs it within the Haskell `IO` monad.

The Haskell interpreter interacts with the disk through a file with the file-system

image. It executes reads and writes by issuing reads and writes to offsets within this file. It is also possible to use the block device (*e.g.*, `/dev/sdb1`) for an external drive; Linux then translates reads and writes to this file into driver operations, using only VFS code rather than another file system.

To allow overlap of computation and I/O, `BeginRead` uses Haskell's lightweight threads from `Control.Concurrent` to create a new background thread for the I/O operation, reading from the physical disk. When the program invokes `WaitForRead`, it must block until the background thread has completed reading the data. To do so, `BeginRead` creates a Haskell `MVar` that it eventually fills with the result, and `WaitForRead` reads this `MVar`, a blocking operation in Haskell. The runtime keeps track of the `MVar`s for pending reads in a map keyed by address. Since the runtime runs verified code, it can assume that code calls these methods in sequence and thus that when a thread invokes `WaitForRead`, there is an associated `MVar` and the runtime has started the I/O.

The cooperative concurrency is straightforward to implement within Haskell's concurrent runtime: the interpreter spawns a thread per system call, assigning it a unique thread id, and coordinates access to the CPU with a global lock. To execute a `Yield`, the interpreter releases the lock.

Scheduling when a thread resumes is important to achieve I/O concurrency: CIO-FSCQ system calls follow a pattern of initiating I/O, then yielding before expecting the results. This pattern allows for I/O concurrency as long as other threads run in the meantime, but the scheduler might not achieve this if threads were re-scheduled too soon after initiating I/O. The interpreter uses a simple heuristic to encourage good thread scheduling: when a thread yields, before attempting to resume by acquiring the global lock, the interpreter waits for the thread's pending I/O to finish. This heuristic assumes threads yield after starting I/O because they need the results after resuming, and might be inefficient for programs that yield for other reasons. Note that the decision to wait affects liveness but CIO-FSCQ makes no promises about liveness.

The Haskell interpreter also implements memory operations. The Haskell type

system cannot directly express the heterogeneous list we use in Coq, so the runtime represents the memory as a `Data.Map` from integers (list indices) to `Any`, the special Haskell primitive for representing a value of any type. Each `Get` then escapes the Haskell type system to coerce this value to the right type. This is safe because the program type checks within Coq, which has a stronger type system that can express the heterogeneous list. The runtime initializes all the variables in the memory map with default values.

Chapter 5

Evaluation

We evaluate CIO-FSCQ to answer two questions:

1. What is the effort involved in building and verifying CIO-FSCQ? (Section 5.1)
2. Does CIO-FSCQ successfully employ I/O concurrency to improve performance? (Section 5.2)

5.1 Effort

To answer the first question, we examine the lines of code in CIO-FSCQ, reported in Table 4.1. FSCQ is about 83,000 lines of code (*i.e.*, specification, implementation, and proofs), to which CIO-FSCQ adds I/O concurrency with 11,000 lines of code. Based on these numbers, it is clear that the design of CIO-FSCQ succeeds in leaving much of the file-system implementation and verification to the existing sequential file system.

In addition to few total lines of code, CIO-FSCQ can leverage incremental changes to FSCQ easily, because CIO-FSCQ's design is mostly agnostic to FSCQ. For example, CIO-FSCQ incorporates a newer version of FSCQ that implements a new logging protocol to get higher performance but CIO-FSCQ started with an older version of FSCQ. As FSCQ becomes more sophisticated, incorporating changes would require no change to CCL, the cache, or the translator, or to any of their proofs. There are

about 3,100 lines of code involved in translating FSCQ's specifications, describing the directory isolation protocol, and proving the system calls correct under this protocol, but much of this code is boilerplate. For example, proving even a read-only file system correct, which requires almost no reasoning about FSCQ's specifications, still takes 1,300 lines of code. Furthermore, changes to FSCQ that maintain the top-level specifications about the directory tree can be incorporated with no modification to the I/O concurrency-specific code.

5.2 I/O concurrency performance

To answer the second question, we measured the performance of CIO-FSCQ on a concurrent workload and compared it to two baseline measurements. The first is running FSCQ, which executes system calls sequentially. The second is Seq-CIO-FSCQ, a configuration for CIO-FSCQ where the interpreter does not release and re-acquire the global lock during a yield, so that programs run sequentially. This baseline includes the overhead of using the concurrent cache and retrying optimistic system calls, but cannot take advantage of I/O concurrency.

To evaluate I/O concurrency, we constructed a small benchmark that exercises the disk in parallel with operations that can run from the cache. We initialize the disk with a large file (10MB) and a small file (4KB). The benchmark consists of two processes: the "large read" process reads the large file in 4KB chunks, while the "small reads" process reads the small file repeatedly 2500 times, each time in one read system call. The benchmark starts these two processes at the same time so that in a system that supports I/O concurrency they can run in parallel: when the large-read process misses in the cache, the small-read process can make progress.

The setup for running the benchmark consists of a Linux machine with a 2.83GHz Intel Core 2 CPU with 4 cores. In all cases the file system is mounted on an external USB drive and accessed directly through its block device in `/dev`. The USB drive achieves relatively slow read throughput (5.9 MB/s for random 4KB reads and 11.6 MB/s for sequential 4K reads), leading to enough I/O delay that CIO-FSCQ improves

Task	time (sec)
10MB disk read	1.9
Small reads (FSCQ)	0.4
Small reads (CIO-FSCQ)	0.6

Table 5.1: Characteristics of the benchmark when run sequentially.

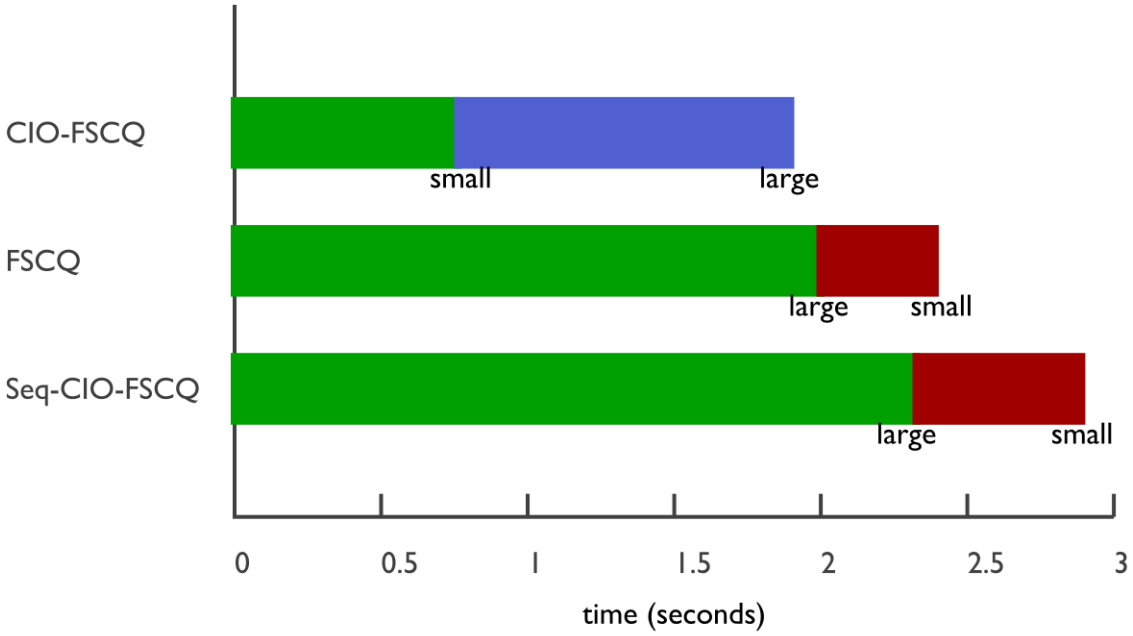
throughput. CPU overhead in FSCQ is high enough that fast I/O (*e.g.*, through an SSD) masks the speedup from I/O concurrency. All runs start with a freshly initialized file system, so the large read's system calls all miss in the cache. The small reads hit in the cache after the first iteration.

Table 5.1 shows the performance of each process run individually on FSCQ. FSCQ is faster for the small reads than CIO-FSCQ, because CIO-FSCQ pays CPU overhead for its cache and retrying system calls when they miss.

The results of running the benchmark are shown in Figure 5-1. We show the completion time for both the large-read and small-reads process. The benchmark's completion time is the time at which the slowest process finishes.

Compared to the sequential systems, CIO-FSCQ completes the benchmark quicker (in 1.9s), because during each system call of the large read, small reads can begin and finish, with little impact on the performance of the large read. In contrast, on FSCQ there is no I/O concurrency and the benchmark completes in 2.5s; this total running time is slightly larger than the sum of the completion time of the small-read process (0.4s) and the large-read process (2.0s).

Figure 5-1 also shows that with CIO-FSCQ the process that runs small-reads finishes sooner than with the sequential systems. The small-read finishes in about 0.8s, with the large reads running in parallel. The reason is that every time when the large-read process misses in the buffer cache and is waiting for a disk read to complete, CIO-FSCQ processes a system call from the small-reads process. Some CPU time is consumed by the large read in CIO-FSCQ: the small reads take 0.8s rather than 0.6s when run alongside the large read compared to alone. FSCQ could achieve an early completion time for the small-read process if the small reads were all scheduled before the large read went to disk, but FSCQ has no way of predicting when a system call



System	large read	small reads	overall (sec)
CIO-FSCQ	1.9	0.8	1.9
FSCQ	2.0	2.4	2.4
Seq-CIO-FSCQ	2.3	2.9	2.9

Figure 5-1: Completion times for large read/small reads concurrent benchmark. Each time is the average of ten runs; standard deviations were less than 70ms in all cases.

will miss in the cache.

Comparing FSCQ and Seq-CIO-FSCQ shows the overhead of adding the cache and retrying system calls to finish cache reads (there are around 2500 disk reads in this benchmark, each of which triggers a retry). This overhead amounts to less than 0.5s for the total completion time of the benchmark.

In summary, the results in this section demonstrate that CIO-FSCQ is able to exploit I/O concurrency with a modest effort for verification. To be able to benefit from I/O concurrency with faster devices, however, we must improve CIO-FSCQ's CPU's performance.

Chapter 6

Future work

6.1 Design changes

There are several limitations in the design of CIO-FSCQ which form good directions for future work.

More concurrency. CIO-FSCQ models only I/O concurrency, allowing only a single thread to access the CPU at any given time. While this can mask I/O latency, it leaves CPU resources idle on multicore machines. In general allowing simultaneous access to the shared memory would break the correctness of FSCQ. However, it should be possible to safely run read-only file-system calls in parallel, with a guarantee in the FSCQ specifications both that the `memstate` is not updated and that the disk is never written.

To extend CIO-FSCQ to allow read parallelism would first require modeling multicore execution of programs. Instead of programs interfering only at yield points, they would need to interleave at arbitrary points. Multicore execution also introduces the possibility for data races on memory (the semantics already includes races on the disk). Since interference is now present at every program step, the protocol would need to govern every step rather than only behavior at yield time. Finally, the interpreter would have to guarantee that the program steps are observed atomically to other threads, as modeled in the semantics; this is relatively easy in the current

execution semantics since threads only interact at yield time, so a global lock provides the correct guarantees.

For more fine-grained concurrency and read-write parallelism, we will likely need a different approach, since threads in the file-system will need to coordinate to maintain consistency of internal data structures like allocators, the in-memory log, and inodes.

Modeling crashes. FSCQ is distinguished by its support for crash-safety in its verification. CIO-FSCQ does not model crashes or even provide persistence. We believe that crashes could be incorporated into CCL and faithfully translated to a modified cache, as follows.

FSCQ writes are asynchronous, with a `Sync` operation that flushes any pending writes to the disk. The asynchrony manifests itself at crash time: any data written after the latest `Sync` might not be persistent and could be lost following a crash. To correctly imitate this behavior in a concurrent setting, optimistic system calls should have I/O behavior identical to their sequential counterparts. As long as the cache tracks the sequence of writes and syncs by the translated code, if the optimistic system call completes without needing to read from disk, its writes can be committed by synchronously reproducing the tracked sequence of writes and syncs.

A further optimization is to allow I/O concurrency between syncs and read-only operations, hiding the disk writes from other threads by keeping the old values in the cache. Any crash during this process would be equivalent to a sequential crash of the syncing thread.

6.2 Implementation limitations

The current prototype of CIO-FSCQ has a few limitations independent of its design that we hope to address in future work.

The execution semantics of CCL consider a single thread a time, modeling other threads abstractly using the protocol. Since verified code follows the protocol, we believe this is a sound model of threads cooperatively interleaving. However, the se-

mantics still abstracts over details not present during execution: ghost state is explicitly manipulated, and yields should simply run other threads. An extension to CCL that would improve trust in the semantics to faithfully model execution would be a lower-level operational semantics for execution of a collection of threads, interleaved at yield points. We could connect the existing, *instrumented* and thread-local semantics to the lower-level and global semantics with a proof that guarantees verified code in the instrumented semantics is also correct in the thread-local semantics, as long as each thread is launched with its precondition satisfied. Such a proof would both model threads interleaving and make precise the notion that ghost state is unneeded at execution time by not including it in the lower-level semantics.

Even without modeling crashes and including crash invariants in specifications, CIO-FSCQ could support durability at an implementation level by flushing writes from the cache, especially when `unmount` is called to cleanly shut down the file system. CIO-FSCQ with `unmount` could be proven correct with a specification that asserts the file-system is unchanged while the memory is reset to default values. A related task is to support cache eviction, especially of dirty writes (which must also persist data).

As mentioned in Section 4.1, CCL programs require all memory variables to be declared in advance. The cache has an ad-hoc scheme to work around this for the purpose of leaving the memory undetermined until the translation and file system code. However, this scheme is unsatisfactory in several ways. First, as mentioned, it fails to hide implementation details from callers. Second, the cache is provided with the whole memory even though it uses only a portion of it: for this reason its specifications all explicitly mention that non-cache variables are unmodified. Finally, the scheme the cache uses is general in many ways, and could be used by applications verified on top of the file-system, but is implemented as a specific pattern for the cache.

One concern with retrying optimistic system calls is that it might take many tries before all the data is in cache, or that before all the data is available some of it is evicted from the cache. Generally this does not pose a problem: most system calls

only read a few blocks from disk. However, reads can potentially be large, and POSIX places no limits on the maximum size of a read. The current prototype of CIO-FSCQ makes no special provision for large reads. Since data is not evicted from the cache, eventually all system calls will finish, but large reads require an inefficient pattern of retries that each read one additional block of data. One way to ameliorate this problem is to prefetch more data whenever issuing I/O, *e.g.*, heuristically reading eight contiguous blocks whenever a miss occurs. The file system could even execute sufficiently large reads without yielding, with a proof that in this case restarting the system call is unnecessary.

Chapter 7

Conclusion

This thesis contributes an approach to verifying I/O concurrency and CIO-FSCQ, a verified I/O-concurrent file system based on that approach. The design of CIO-FSCQ aims to re-use the verification effort of a sequential file system, as a means of lowering proof burden, while still achieving concurrency between a single system call running within the file-system and disk I/O. CIO-FSCQ is based on optimistic system calls, which attempt to run using only a cache and abort rather than wait for disk I/O to complete; a generic translator produces optimistic system calls from FSCQ while preserving specifications. The concurrent file system includes a protocol — directory isolation — that guarantees FSCQ system calls are used correctly and its specifications can be used when optimistic system calls are re-tried until they find all data in the cache. The directory-isolated file system has a linearizable version of FSCQ's sequential specifications and a proof based only on these top-level specifications.

We implemented a prototype of CIO-FSCQ in Coq with a Haskell runtime to execute system calls through the standard file-system interface. An evaluation of the prototype shows that on a benchmark with potential to overlap disk I/O and computation, CIO-FSCQ improves performance.

Bibliography

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

[2] S. Brookes. A semantics for concurrent separation logic. *Theoretical Computer Science*, 375(1–3), May 2007. Festschrift for John C. Reynolds's 70th Birthday.

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

[4] A. 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.

[5] E. F. Codd, E. S. Lowry, E. McDonough, and C. A. Scalzi. Multiprogramming STRETCH: Feasibility considerations. *Commun. ACM*, 2(11):13–17, Nov. 1959. ISSN 0001-0782. doi: 10.1145/368481.368502. URL http://doi.acm.org/10.1145/368481.368502.

[6] Coq development team. *The Coq Proof Assistant Reference Manual, Version 8.6*. INRIA, Apr. 2016. http://coq.inria.fr/distrib/current/refman/.

[7] F. J. Corbató, M. Merwin-Daggett, and R. C. Daley. An experimental time-sharing system. In *Proceedings of the May 1-3, 1962, Spring Joint Computer Conference*, AIEE-IRE '62 (Spring), pages 335–344, New York, NY, USA, 1962. ACM. doi: 10.1145/1460833.1460871. URL http://doi.acm.org/10.1145/1460833.1460871.

[8] M. Curtis-Maury, V. Devadas, V. Fang, and A. Kulkarni. To Waffinity and beyond: A scalable architecture for incremental parallelization of file system code. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 419–434, Berkeley, CA, USA, 2016. USENIX

Association. ISBN 978-1-931971-33-1. URL http://dl.acm.org/citation.cfm?id=3026877.3026910.

[9] E. W. Dijkstra. The structure of the "THE"-multiprogramming system. In *Proceedings of the First ACM Symposium on Operating System Principles*, SOSP '67, pages 10.1–10.6, New York, NY, USA, 1967. ACM. doi: 10.1145/800001.811672. URL http://doi.acm.org/10.1145/800001.811672.

[10] T. Dinsdale-Young, L. Birkedal, P. Gardner, M. Parkinson, and H. 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, Jan. 2013.

[11] M. Dodds, X. Feng, M. Parkinson, and V. Vafeiadis. Deny-guarantee reasoning. In *Proceedings of the 18th European Symposium on Programming Languages and Systems: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, ESOP '09, pages 363–377, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-00589-3. doi: 10.1007/978-3-642-00590-9_26. URL http://dx.doi.org/10.1007/978-3-642-00590-9_26.

[12] X. Feng. Local rely-guarantee reasoning. In *Proceedings of the 36th ACM Symposium on Principles of Programming Languages (POPL)*, Savannah, GA, Jan. 2009.

[13] X. Feng, R. Ferreira, and Z. Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In *European Symposium on Programming*, pages 173–188. Springer, 2007.

[14] R. Gu, J. Koenig, T. Ramananandro, Z. Shao, X. Wu, S.-C. Weng, H. Zhang, and Y. Guo. Deep specifications and certified abstraction layers. In *Proceedings of the 42nd ACM Symposium on Principles of Programming Languages (POPL)*, Mumbai, India, Jan. 2015.

[15] R. Gu, Z. Shao, H. Chen, X. N. Wu, J. Kim, V. Sjöberg, and D. Costanzo. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 653–669, GA, 2016. USENIX Association. ISBN 978-1-931971-33-1. URL https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu.

[16] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. 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, Oct. 2014.

[17] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill. IronFleet: Proving practical distributed systems correct. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, Monterey, CA, Oct. 2015.

[18] C. Hawblitzel, E. Petrank, S. Qadeer, and S. Tasiran. Automated and modular refinement reasoning for concurrent programs. Technical Report MSR-TR-2015-8, Microsoft Research, Feb. 2015.

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

[20] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 26(1):100–106, Jan. 1983.

[21] IEEE (The Institute of Electrical and Electronics Engineers) and The Open Group. The Open Group base specifications issue 7, 2016 edition (POSIX.1-2008/Cor 1-2016), Sept. 2016.

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

[23] G. Keller, T. Murray, S. Amani, L. O'Connor, Z. Chen, L. Ryzhyk, G. Klein, and G. Heiser. File systems deserve verification too. In *Proceedings of the 7th Workshop on Programming Languages and Operating Systems*, Farmington, PA, Nov. 2013.

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

[25] D. Leinenbach and T. Santen. Verifying the Microsoft Hyper-V hypervisor with VCC. In *Proceedings of the 2nd World Congress on Formal Methods*, pages 806–809, 2009.

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

[27] P. W. OHearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, Apr. 2007. ISSN 0304-3975. doi: 10.1016/j.tcs.2006.12.035. URL http://dx.doi.org/10.1016/j.tcs.2006.12.035.

[28] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Annual Technical Conference*, pages 305–319, Philadelphia, PA, June 2014.

[29] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6(4):319–340, 1976.

[30] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable web server. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '99, pages 15–15, Berkeley, CA, USA, 1999. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1268708.1268723.

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

[32] I. Sergey, A. Nanevski, and A. Banerjee. Mechanized verification of fine-grained concurrent programs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 77–87, 2015.

[33] H. Sigurbjarnarson, J. Bornholt, E. Torlak, and X. Wang. Push-button verification of file systems via crash refinement. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 1–16, GA, 2016. USENIX Association. ISBN 978-1-931971-33-1. URL https://www.usenix.org/conference/osdi16/technical-sessions/presentation/sigurbjarnarson.

[34] K. Svendsen and L. Birkedal. Impredicative concurrent abstract predicates. In *Proceedings of the 23rd European Symposium on Programming Languages and Systems*, pages 149–168, 2014.

[35] A. Turon, V. Vafeiadis, and D. Dreyer. GPS: Navigating weak memory with ghosts, protocols, and separation. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 691–707, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2585-1. doi: 10.1145/2660193.2660243. URL http://doi.acm.org/10.1145/2660193.2660243.

[36] A. Vasudevan, S. Chaki, L. Jia, J. McCune, J. Newsome, and A. Datta. Design, implementation and verification of an extensible and modular hypervisor framework. In *Proceedings of the 34th IEEE Symposium on Security and Privacy*, San Francisco, CA, May 2013.

[37] P. Wadler. Monads for functional programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 24–52, London, UK, UK, 1995. Springer-Verlag. ISBN 3-540-59451-5. URL http://dl.acm.org/citation.cfm?id=647698.734146.

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

[39] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. 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.

[40] J. Yang and C. 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.