DOI:10.1145/3051092

Certifying a File System Using Crash Hoare Logic: Correctness in the Presence of Crashes

By Tej Chajed, Haogang Chen, Adam Chlipala, M. Frans Kaashoek, Nickolai Zeldovich, and Daniel Ziegler

Abstract

FSCQ is the first file system with a machine-checkable proof that its implementation meets a specification, even in the presence of fail-stop crashes. FSCQ provably avoids bugs that have plagued previous file systems, such as performing disk writes without sufficient barriers or forgetting to zero out directory blocks. If a crash happens at an inopportune time, these bugs can lead to data loss. FSCQ's theorems prove that, under any sequence of crashes followed by reboots, FSCQ will recover its state correctly without losing data.

To state FSCQ's theorems, this paper introduces the Crash Hoare logic (CHL), which extends traditional Hoare logic with a crash condition, a recovery procedure, and logical address spaces for specifying disk states at different abstraction levels. CHL also reduces the proof effort for developers through proof automation. Using CHL, we developed, specified, and proved the correctness of the FSCQ file system. Although FSCQ's design is relatively simple, experiments with FSCQ as a user-level file system show that it is sufficient to run Unix applications with usable performance. FSCQ's specifications and proofs required significantly more work than the implementation, but the work was manageable even for a small team of a few researchers.

1. INTRODUCTION

This paper describes Crash Hoare logic (CHL), which allows developers to write specifications for crash-safe storage systems and also prove them correct. "Correct" means that, if a computer crashes due to a power failure or other fail-stop fault and subsequently reboots, the storage system will recover to a state consistent with its specification (e.g., POSIX[17]). For example, after recovery, either all disk writes from a file-system call will be on disk, or none will be. Using CHL we write a simple specification for a subset of POSIX and build the FSCQ *certified* file system, which comes with a machine-checkable proof that its implementation matches the specification.

Proving the correctness of a file system implementation is important, because existing file systems have a long history of bugs both in normal operation and in handling crashes.[24] Reasoning about crashes is especially challenging because it is difficult for the file-system developer to consider all possible points where a crash could occur, both while a file-system call is running and during the execution of recovery code. Often, a system may work correctly in many cases, but if a crash happens at a particular point between two specific disk writes, then a problem arises.[29, 39]

Most approaches to building crash-safe file systems fall roughly into three categories (see the SOSP paper[3] for a more in-depth discussion of related work): testing, program analysis, and model checking. Although they are effective at finding bugs in practice, none of them can guarantee the absence of crash-safety bugs in actual implementations. This paper focuses precisely on this issue: helping developers build file systems with machine-checkable proofs that they correctly recover from crashes at any point.

Researchers have used theorem provers for certifying real-world systems such as compilers,[23] small kernels,[22] kernel extensions,[35] and simple remote servers.[15] Some certification projects[1, 10, 11, 18, 28, 32] have even targeted file systems, as we do, but in each case either the file system was not complete, executable, and ready to run on a real operating system; or its proof did not consider crashes. Reasoning about crash-free executions typically involves considering the states before and after some operation. Reasoning about crashes is more complicated because crashes can expose intermediate states of an operation.

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

CHL addresses these challenges by allowing programmers to specify what invariants hold in case of crashes and

The original version of this paper is entitled "Using Crash Hoare Logic for Certifying the FSCQ File System" and was published in the *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP '15)*.[3]

by incorporating the notion of a recovery procedure that runs after a crash. CHL supports the construction of modular systems through a notion of logical address spaces. CHL also allows for a high degree of proof automation. Using CHL we specified and proved correct the FSCQ file system, which includes a simple write-ahead log and which uses asynchronous disk writes.

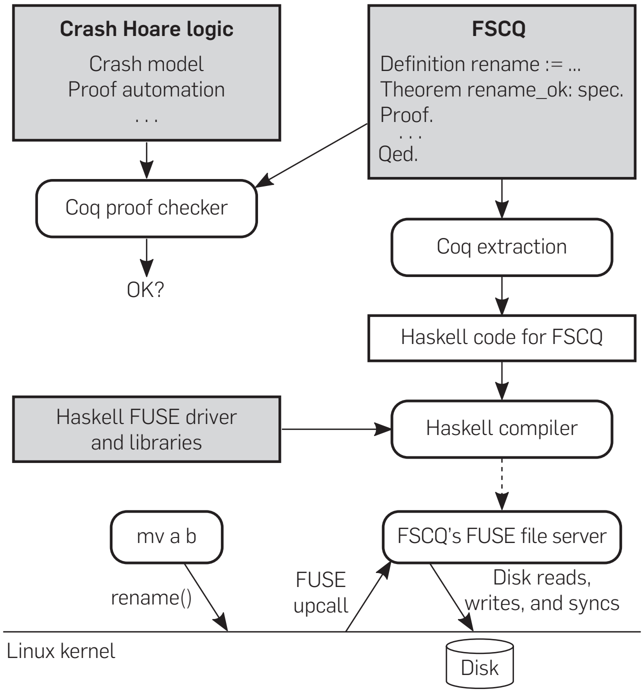
The next section of this article gives an overview of our system architecture, including implementation and proof. Then we introduce CHL, our approach to verifying storage programs that may crash. Afterward, we discuss our prototype file-system implementation FSCQ that we verified with CHL, and we evaluate it in terms of performance, correctness, and other desirable qualities.

2. SYSTEM OVERVIEW

We have implemented the CHL specification framework with the widely used Coq proof assistant,[8] which provides a single programming language for both proof and implementation. The source code is available at https://github.com/mit-pdos/fscq. Figure 1 shows the components involved in the implementation. CHL is a small specification language embedded in Coq that allows a file-system developer to write specifications that include *crash conditions* and *recovery procedures*, and to prove that implementations meet these specifications. We have stated the semantics of CHL and proven it sound in Coq.

We implemented and certified FSCQ using CHL. That is, we wrote specifications for a subset of the POSIX system calls using CHL, implemented those calls inside of Coq, and proved that the implementation of each call meets its specification. We devoted substantial effort to building reusable proof automation for CHL. However, writing specifications and proofs still took a significant amount of time, compared to the time spent writing the implementation.

Figure 1. Overview of FSCQ's implementation. Rectangular boxes denote source code; rounded boxes denote processes. Shaded boxes denote source code written by hand. The dashed line denotes the Haskell compiler producing an executable binary for FSCQ's FUSE file server.



As a standard of completeness for FSCQ, we aimed for the same features as the xv6 file system,[9] a teaching operating system that implements the Unix v6 file system with write-ahead logging. FSCQ supports fewer features than today's Unix file systems; for example, it lacks support for multiprocessors and deferred durability (i.e., `fsync`). However, it provides the core POSIX file-system calls, including support for large files using indirect blocks, nested directories, and `rename`.

Using Coq's extraction feature, we do automatic translation of the Coq code for FSCQ into a Haskell program. We run this generated implementation combined with a small (uncertified) Haskell driver as a FUSE[12] user-level file server. This implementation strategy allows us to run unmodified Unix applications but pulls in Haskell, our Haskell driver, and the Haskell FUSE library as trusted components.

3. CRASH HOARE LOGIC

Our goal is to allow developers to certify the correctness of a storage system formally—that is, to prove that it functions correctly during normal operation and that it recovers properly from any possible crashes. As mentioned in the abstract, a file system might forget to zero out the contents of newly allocated directory or indirect blocks, leading to corruption during normal operation, or it might perform disk writes without sufficient barriers, leading to disk contents that might be unrecoverable. Prior work has shown that even mature file systems in the Linux kernel have such bugs during normal operation[24] and in crash recovery.[38]

To prove that an implementation meets its specification, we must have a way for the developer to declare which behaviors are permissible under crashes. To do so, we extend *Hoare logic*,[16] where specifications are of the form $\{P\}$ `procedure` $\{Q\}$. Here, `procedure` could be a sequence of disk operations (e.g., read and write), interspersed with computation, that manipulates the persistent state on disk, like the implementation of the `rename` system call or a lower-level operation like allocating a disk block. P corresponds to the precondition that should hold before `procedure` is run and Q is the postcondition. To prove that a specification is correct, we must prove that `procedure` establishes Q, assuming P holds before invoking `procedure`. In our `rename` system call example, P might require that the file system be represented by some tree t and Q might promise that the resulting file system is represented by a modified tree t' reflecting the rename operation.

Hoare logic is insufficient to reason about crashes, because a crash may cause `procedure` to stop at any point in its execution and may leave the disk in a state where Q does not hold (e.g., in the `rename` example, the new file name has been created already, but the old file name has not yet been removed). Furthermore, if the computer reboots, it often runs a recovery procedure (such as `fsck`) before resuming normal operation. Hoare logic does not provide a notion that at any point during `procedure`'s execution, a recovery procedure may run. The rest of this section describes how CHL extends Hoare logic to handle crashes.

Traditional Hoare logic distinguishes between partial correctness, where we prove that terminating programs give correct answers, and total correctness, where we also prove termination. We use Coq's built-in termination checker to guarantee that our system calls always finish, when no crashes occur. However, we model cases where a program still runs forever, because it keeps crashing and then crashing again during recovery, ad infinitum. For that reason, our specifications can be interpreted as total correctness for crash-free executions and partial correctness for crashing executions, which makes sense, since the underlying hardware platform refuses to give the programmer a way to guarantee normal termination in the presence of crashes.

3.1. Example

Many file-system operations must update two or more disk blocks atomically. For example, when creating a file, the file system must both mark an inode as allocated as well as update the directory in which the file is created (to record the file name with the allocated inode number). To ensure correct behavior under crashes, a common approach is to use a write-ahead log. Logging guarantees that, if a file-system operation crashed while applying its changes to the disk, then after a crash, a recovery procedure can finish the job by applying the log contents. Write-ahead logging makes it possible to avoid the undesirable intermediate state where the inode is allocated but not recorded in the directory, effectively losing an inode. Many file systems, including widely used ones like Linux's ext4,[34] employ logging exactly for this reason.

The simple procedure shown in Figure 2 captures the essence of file-system calls that must update two or more blocks. The procedure performs two disk writes using a write-ahead log, which supplies the `log_begin`, `log_commit`, and `log_write` APIs. The procedure `log_write` appends a block's content to an in-memory log, instead of updating the disk block in place. The procedure `log_commit` writes the log to disk, writes a commit record, and then copies the block contents from the log to the blocks' locations on disk. If this procedure crashes and the system reboots, the recovery procedure runs. The recovery procedure looks for the commit record. If there is a commit record, it completes the operation by copying the block contents from the log into the proper locations and then cleans the log.

Figure 2. Pseudocode of `atomic_two_write`.

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

If there is no commit record, then the recovery procedure just cleans the log. If there is a crash during recovery, then after reboot the recovery procedure runs again. In principle, this may happen several times. If the recovery finishes, however, then either both blocks have been updated or neither has. Thus, in the `atomic_two_write` procedure from Figure 2, the write-ahead log guarantees that either both writes happen or neither does, no matter when and how many crashes happen.

CHL makes it possible to write specifications for procedures such as `atomic_two_write` and the write-ahead logging system, as we will explain in the rest of the section.

3.2. Crash conditions

CHL needs a way for developers to write down predicates about disk states, such as a description of the possible intermediate states where a crash could occur. To do this, CHL extends Hoare logic with crash conditions, similar in spirit to prior work on fail-stop processors[33, Section 3] and fault conditions from concurrent work,[28] but formalized precisely to allow for executable implementations and machine-checked proofs.

For modularity, CHL should allow reasoning about just one part of the disk, rather than having to specify the contents of the entire disk at all times. For example, we want to specify what happens with the two blocks that `atomic_two_write` updates and not have to say anything about the rest of the disk. To do this, CHL employs *separation logic*,[30] which is a way of combining predicates on disjoint parts of a store (in our case, the disk). The basic predicate in separation logic is a points-to relation, written as $a \mapsto v$, which means that address a has value v. Given two predicates x and y, separation logic allows CHL to produce a combined predicate $x \star y$. The $\star$ operator means that the disk can be split into two disjoint parts, where one satisfies the x predicate, and the other satisfies y.

To reason about the behavior of a procedure in the presence of crashes, CHL allows a developer to capture both the state at the end of the procedure's crash-free execution *and* the intermediate states during the procedure's execution in which a crash could occur. For example, Figure 3 shows the CHL specification for FSCQ's `disk_write`. (In our implementation of CHL, these specifications are written in Coq code; we show here an easier-to-read version.) We will describe the precise notation shortly, but for now, note that the specification has four parts: the procedure about which we are reasoning, `disk_write(a, v)`; the precondition, **disk**: $a \mapsto \langle v_0, vs\rangle \star$ *other_blocks*; the postcondition if there are no crashes, **disk**: $a \mapsto \langle v, [v_0]\oplus vs\rangle \star$ *other_blocks*; and the crash condition, **disk**: $a \mapsto \langle v_0, vs\rangle \star$

Figure 3. Specification for `disk_write`.

SPEC disk_write(a, v)
PRE **disk**: $a \mapsto \langle v_0, vs\rangle \star$ *other_blocks*
POST **disk**: $a \mapsto \langle v, [v_0]\oplus vs\rangle \star$ *other_blocks*
CRASH **disk**: $a \mapsto \langle v_0, vs\rangle \star$ *other_blocks* $\vee$
$a \mapsto \langle v, [v_0]\oplus vs\rangle) \star$ *other_blocks*

$other\_blocks \vee a \mapsto \langle v, [v_0] \oplus vs \rangle \star other\_blocks$. Moreover, note that the crash condition specifies that `disk_write` could crash in two possible states (either before making the write or after). In all three logical conditions, *other_blocks* stands for the arbitrary contents of all other disk blocks beside a, which should be preserved by this operation, even in case of a crash.

The specification in Figure 3 captures asynchronous writes. To do so, CHL models the disk as a (partial) function from a block number to a tuple, $\langle v, vs \rangle$, consisting of the last-written value v and a set of previous values vs, any of which could appear on disk after a crash. Block numbers greater than the size of the disk do not map to anything. Reading from a block returns the last-written value, since even if there are previous values that might appear after a crash, in the absence of a crash a read should return the last write. Writing to a block makes the new value the last-written value and adds the old last-written value to the set of previous values. Reading or writing a block number that does not exist causes the system to "fail" (as opposed to finishing or crashing). Finally, CHL's disk model supports a sync operation, which forces the disk to flush pending writes to persistent storage, modeled in the postcondition for sync by discarding all previous values.

Returning to Figure 3, the `disk_write` specification asserts through the precondition that the address being written, a, must be valid (i.e., within the disk's size), by stating that address a points to some value $\langle v_0, vs \rangle$ on disk. The specification's postcondition asserts that the block being modified will contain the new value $\langle v, [v_0] \oplus vs \rangle$; that is, the new last-written value is v, and v_0 is added to the set of previous values. The specification also asserts through the crash condition that `disk_write` could crash in a state that satisfies $a \mapsto \langle v_0, vs \rangle \star other\_blocks \vee a \mapsto \langle v, [v_0] \oplus vs \rangle \star other\_blocks$, that is, either the write did not happen (a still has $\langle v_0, vs \rangle$) or it did (a has $\langle v, [v_0] \oplus vs \rangle$). Finally, the specification asserts that the rest of the disk is unaffected: if other disk blocks satisfied some predicate *other_blocks* before `disk_write`, they will still satisfy the same predicate afterward.

One subtlety of CHL's crash conditions is that they describe the state of the disk *just before* the crash occurs, rather than just after. Right after a crash, CHL's disk model specifies that each block nondeterministically chooses one value from the set of possible values before the crash. For instance, the first line of Figure 3's crash condition says that the disk still "contains" all previous writes, represented by $\langle v_0, vs \rangle$, rather than a specific value that persisted across the crash, chosen out of $[v_0] \oplus vs$. This choice of representing the state before the crash rather than after the crash allows the crash condition to be similar to the pre- and postconditions. For example, in Figure 3, the state of other blocks just before a crash matches the *other_blocks* predicate, as in the pre- and postconditions. However, describing the state after the crash would require a more complex predicate (e.g., if *other_blocks* contains unsynced disk writes, the state after the crash must choose one of the possible values). Making crash conditions similar to pre- and postconditions is good for proof automation.

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

The `disk_write` specification states that blocks are written atomically; that is, after a crash, a block must contain either the last-written value or one of the previous values, and partial block writes are not allowed. This is a common assumption made by file systems and we believe it matches the behavior of many disks in practice. Using CHL, we could capture the notion of partial block writes by specifying a more complicated crash condition, but the specification shown here matches the common assumption. We leave to future work the question of how to build a certified file system without that assumption.

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

3.3. Logical address spaces

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

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

As an example of address spaces, consider the specification of `atomic_two_write`, shown in Figure 4. Rather than describe how `atomic_two_write` modifies the on-disk data structures, the specification introduces new address spaces, *start_state* and *new_state*, which correspond to the contents of the logical disk provided by the logging system. Logical address spaces allow the developer of the logging system to state a clean specification, which provides the abstraction of a simple, synchronous disk to higher layers in the file system. Developers of higher layers can then largely ignore the details of the underlying asynchronous disk.

Specifically, in the precondition, $a_1 \mapsto v_x$ applies to the address space representing the starting contents of the logical disk, and in the postcondition, $a_1 \mapsto v_1$ applies to the new contents of the logical disk. Like the physical disk, these address spaces are partial functions from addresses to values (in this case, mapping 64-bit block numbers to 4 KB block values). Unlike the physical disk, the logical disk address space provided by the logging system associates a single value with each block, rather than a set of values, because the transaction system exports a sound synchronous interface, proven correct on top of the asynchronous interface below. Note how we use a variable *others* to stand for untouched disk addresses in the logical disk, just as we did for the physical disk in Figure 3.

Crucial to such a specification are explicit connections between address spaces. In Figure 4, we use a predicate `log_rep`, whose definition we elide here, but which captures how to map a higher-level state into a set of permissible lower-level states. For this example of a logging layer, the predicate maps a "virtual" disk into a "physical" disk that includes a log. Such predicates may take additional arguments, as with the `NoTxn` argument that we use here to indicate that the logging layer is in a quiescent state, between transactions. This technique for connecting logical layers generalizes to stacks of several layers, as naturally appear in a file system.

The crash condition of `atomic_two_write`, from Figure 4, describes all of the states in which `atomic_two_write` could crash using `log_intact(`*`d1, d2`*`)`, which stands for all possible `log_rep` states that recover to transaction states `d1` or `d2`. Using `log_intact` allows us to capture all possible crash states concisely, including states that can appear deep inside any procedure that `atomic_two_write` might call (e.g., crashes inside `log_commit`).

Figure 4. Specification for `atomic_two_write`.

SPEC atomic_two_write(a_1, v_1, a_2, v_2)
PRE **disk**: log_rep(NoTxn, *start_state*)
start_state: $a_1 \mapsto v_x \star a_2 \mapsto v_y \star$ *others*
POST **disk**: log_rep(NoTxn, *new_state*)
new_state: $a_1 \mapsto v_1 \star a_2 \mapsto v_2 \star$ *others*
CRASH **disk**: log_intact(*start_state*, *new_state*)

3.4. Recovery execution semantics

Crash conditions and address spaces allow us to specify the possible states in which the computer might crash in the middle of a procedure's execution. We also need a way to reason about recovery, including crashes during recovery.

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

The specification of the `log_recover` procedure is shown in Figure 5. It says that, starting from any state matching `log_intact(`*`last_state, committed_state`*`)`, `log_recover` will either roll back the transaction to *`last_state`* or will roll forward a committed transaction to *`committed_state`*. Furthermore, the fact that `log_recover`'s crash condition implies the precondition indicates that `log_recover` is *idempotent*, meaning that it can be safely restarted after a crash to achieve the same postcondition. (Strictly speaking, this sense of idempotence differs from the mathematical notion, but follows conventions established in early work on fault-tolerant storage systems.[14])

To formalize the requirement that `log_recover` must run after a crash, CHL provides a recovery execution semantics. The recovery semantics talks about *two* procedures executing (a normal procedure and a recovery procedure) and producing either a failure, a *completed* state (after finishing the normal procedure), or a *recovered* state (after finishing the recovery procedure). This regime models the notion that the normal procedure tries to execute and reach a completed state, but if the system crashes, it starts running the recovery procedure (perhaps multiple times if there are crashes during recovery), which produces a recovered state.

Figure 6 shows how to extend the `atomic_two_write` specification to include recovery execution using the ≫ notation. The postcondition indicates that, if `atomic_two_write` finishes without crashing, both blocks were updated, and if one or more crashes occurred, with `log_recover` running after each crash, either both blocks were updated

Figure 5. Specification of `log_recover`.

SPEC log_recover()
PRE **disk**: log_intact(*last_state*, *committed_state*)
POST **disk**: log_rep(NoTxn, *last_state*) ∨
log_rep(NoTxn, *committed_state*)
CRASH **disk**: log_intact(*last_state*, *committed_state*)

or neither was. The special *ret* variable indicates whether the system reached a completed or a recovered state and in this case enables callers of `atomic_two_write` to conclude that, if `atomic_two_write` completed without crashes, it updated both blocks (i.e., updating none of the blocks is allowed only if the system crashed and recovered).

Note that distinguishing the completed and recovered states allows the specification to state stronger properties for completion than recovery. Also note that the recovery-execution version of `atomic_two_write` does not have a crash condition: if the computer crashes, it will run `log_recover` again, and the specification describes what happens when the computer eventually stops crashing and `log_recover` can run to completion.

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

3.5. Proving

In order to prove the correctness of a procedure, CHL follows the standard Hoare-logic approach of decomposing the procedure into smaller units (e.g., control-flow constructs or lower-level functions with already-proven specifications) and chaining their pre- and postconditions according to the procedure's control flow. Figure 7 shows an example of this for a simple procedure; much of this chaining is automated in CHL. The crash condition for a procedure is the disjunction (i.e., "or") of the crash conditions of all components of that procedure, as illustrated by the red arrows in Figure 7. Finally, proving the correctness of a procedure together with its recovery function requires proving that the procedure's crash condition implies the recovery precondition, and that recovery itself is idempotent.

Figure 6. Specification for `atomic_two_write` with recovery. The ≫ operator indicates the combination of a regular procedure and a recovery procedure.

SPEC atomic_two_write(a_1, v_1, a_2, v_2) ≫ log_recover()
PRE **disk**: log_rep(NoTxn, *start_state*)
start_state: $a_1 \mapsto v_x \star a_2 \mapsto v_y \star$ *others*
POST **disk**: log_rep(NoTxn, *new_state*) ∨
(*ret* = RECOVERED ∧ log_rep(NoTxn, *start_state*))
new_state: $a_1 \mapsto v_1 \star a_2 \mapsto v_2 \star$ *others*

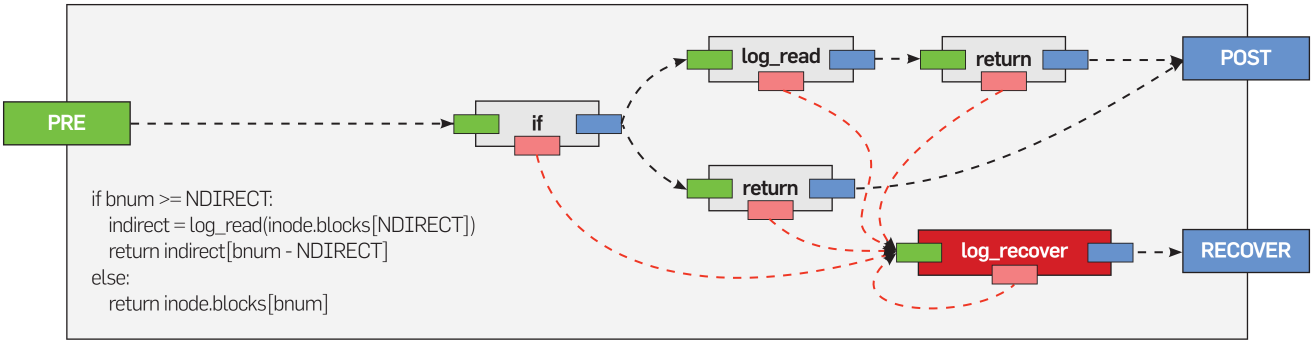
4. PROTOTYPE IMPLEMENTATION

The implementation follows the organization shown in Figure 1. FSCQ and CHL are implemented using Coq, which provides a single programming language for implementations, specifications, and proofs. Figure 8 breaks down the source code of FSCQ and CHL. Because Coq provides a single language, proofs are interleaved with source code and are difficult to separate. The development effort took several researchers about a year and a half; most of it was spent on proofs and specifications. Checking the proofs takes 11 h on an Intel i7-3667U 2.00 GHz CPU with 8 GB DRAM. The proofs are complete; we used Coq's `Print Assumptions` command to verify that FSCQ did not introduce any unproven axioms or assumptions.

CHL. CHL is implemented as a domain-specific language inside of Coq, much like a macro language (or, in the more technical language of proof assistants, we use a shallow embedding). We specified the semantics of this language and proved that it is sound. For example, we proved the standard Hoare-logic specifications for the `for` and `if` combinators. We also proved the specifications of `disk_read`, `disk_write` (whose spec is in Figure 3), and `disk_sync` manually, starting from CHL's execution and crash model. Much of the automation (e.g., the chaining of pre- and postconditions) is implemented using Ltac, Coq's domain-specific language for proof search.

FSCQ. We implemented FSCQ also inside of Coq, writing the specifications using CHL. We proved that the implementation obeys the specifications, starting from the basic operations in CHL. FSCQ's write-ahead log simplified specification and implementation tremendously, because much

Figure 7. Example control flow of a CHL procedure that looks up the address of a block in an inode, with support for indirect blocks. (The actual code in FSCQ checks for some additional error cases.) Gray boxes represent the specifications of procedures. The dark red box represents the recovery procedure. Green and pink boxes represent preconditions and crash conditions, respectively. Blue boxes represent postconditions. Dashed arrows represent logical implication.



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

Figure 8. Combined lines of code and proof for FSCQ components.

Component	Lines of code
Fixed-width words	2,709
CHL infrastructure	5,895
Proof automation	2,304
On-disk data structures	7,571
Buffer cache	662
Write-ahead logging	3,191
Bitmap allocator	441
Inodes and files	3,317
Directories	4,451
FSCQ's top-level API	1,198
Total	31,739

of the detailed reasoning about crashes is localized in the write-ahead log.

FSCQ file server. We produced running code by using Coq's extraction mechanism to generate equivalent Haskell code from our Coq implementation. We wrote a driver program in Haskell (400 lines of code) along with an efficient Haskell reimplementation of fixed-size words and disk-block operations (350 more lines of Haskell). The extracted code, together with this driver and word library, allows us to efficiently execute our certified implementation.

To allow applications to use FSCQ, we exported FSCQ as a FUSE file system, using the Haskell FUSE bindings[2] in our Haskell FSCQ driver. We mount this FUSE FSCQ file system on Linux, allowing Linux applications to use FSCQ without any modifications. Compiling the Coq and Haskell code to produce the FUSE executable, without checking proofs, takes a little under 2 min.

Limitations. Although extraction to Haskell simplifies the process of generating executable code from our Coq implementation, it adds the Haskell compiler and runtime into FSCQ's trusted computing base. In other words, a bug in the Haskell compiler or runtime could subvert any of the guarantees that we prove about FSCQ. We believe this is a reasonable trade-off, since our goal is to certify higher-level properties of the file system, and other projects have shown that it is possible to extend certification all the way to assembly.[6, 15, 22]

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

A limitation of FSCQ's write-ahead log is that it does not guarantee how much log space is available to commit a transaction; if a transaction performs too many writes, `log_commit` can return an error. Some file systems deal with this by reasoning about how many writes each transaction can generate and ensuring that the log has sufficient space before starting a transaction. We have not done this in FSCQ yet, although it should be possible to expose the number of available log entries in the log's representation invariant. Instead, we allow `log_commit` to return an error, in which case the entire transaction (e.g., system call) aborts and returns an error.

5. EVALUATION

To evaluate FSCQ, this section answers several questions:

- Is FSCQ complete enough for realistic applications, and can it achieve reasonable performance? (Section 5.1)
- What kinds of bugs do FSCQ's theorems preclude? (Section 5.2)
- Does FSCQ recover from crashes? (Section 5.3)
- How difficult is it to build and evolve the code and proofs for FSCQ? (Section 5.4)

5.1. Application performance

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

Experimental setup. We used a set of applications representing typical software development: cloning a Git repository, compiling the sources of the xv6 file system and the LFS benchmark[31] using `make`, running the LFS benchmark, and deleting all of the files to clean up at the end. We also run `mailbench`, a qmail-like mail server from the sv6 operating system.[7] This models a real mail server, where using FSCQ would ensure email is not lost even in case of crashes.

We compare FSCQ's performance to two other file systems: the Linux ext4 file system and the file system from the xv6 operating system (chosen because its design is similar to FSCQ's). We modified xv6 to use asynchronous disk writes and ported the xv6 file system to FUSE so that we can run it in the same way as FSCQ. Finally, to evaluate the overhead of FUSE, we also run the experiments on top of ext4 mounted via FUSE.

To make a meaningful comparison, we run the file systems in synchronous mode; that is, every system call commits to disk before returning. (Disk writes within a system call can be asynchronous, as long as they are synced at the end.) For FSCQ and xv6, this is the standard mode of operation. For ext4, we use the `data=journal` and `sync` options. Although this is not the default mode of operation for ext4, the focus of this evaluation is on whether FSCQ can achieve good performance for its design, not whether its simple design can match that of a sophisticated file system like ext4. To give a sense of how much performance can be obtained through further optimizations or spec changes, we measure ext4 in three additional configurations: the journal_async_commit mode, which uses checksums to commit in one disk sync instead of two ("ext4-journal-async" in our experiments); the data=ordered mode, which is incompatible with journal_async_commit ("ext4-ordered"); and the default data=ordered and async mode, which does not sync to disk on every system call ("ext4-async").

We ran all of these experiments on a quad-core Intel i7-3667U 2.00 GHz CPU with 8 GB DRAM running Linux 3.19. The file system was stored on a separate partition on an Intel SSDSCMMW180A3L flash SSD. Running the experiments on an SSD ensures that potential file-system CPU bottlenecks are not masked by a slow rotational disk. We compiled FSCQ's Haskell code using GHC 7.10.2.

Results. The results of running our experiments are shown in Figure 9. The first conclusion is that FSCQ's performance is close to that of the xv6 file system. The small gap between FSCQ and xv6 is due to the fact that FSCQ's Haskell implementation uses about four times more CPU time than xv6's. This can be reduced by generating C or assembly code instead of Haskell. Second, FUSE imposes little overhead, judging by the difference between ext4 and ext4-fuse. Third, both FSCQ and xv6 lag behind ext4. This is due to the fact that our benchmarks are bottlenecked by syncs to the SSD, and that ext4 has a more efficient logging design that defers applying the log contents until the log fills up, instead of at each commit. As a result, ext4 can commit a transaction with two disk syncs, compared to four disk syncs for FSCQ and xv6. For example, mailbench requires 10 transactions per message, and the SSD can perform a sync in about 2.8 ms. This matches the observed performance of ext4 (64 ms per message) and xv6 and FSCQ (103 and 118 ms per message, respectively).

Finally, there is room for even further optimizations: ext4's journal_async_commit commits with one disk sync instead of two, achieving almost twice the throughput in some cases; data=ordered avoids writing file data twice, achieving almost twice the throughput in other cases; and asynchronous mode achieves much higher throughput by avoiding disk syncs altogether (at the cost of not persisting data right away).

5.2. Bug discussion

To understand whether FSCQ eliminates real problems that arise in current file systems, we consider broad categories of bugs that have been found in real-world file systems[24, 38] and discuss whether FSCQ's theorems eliminate similar bugs:

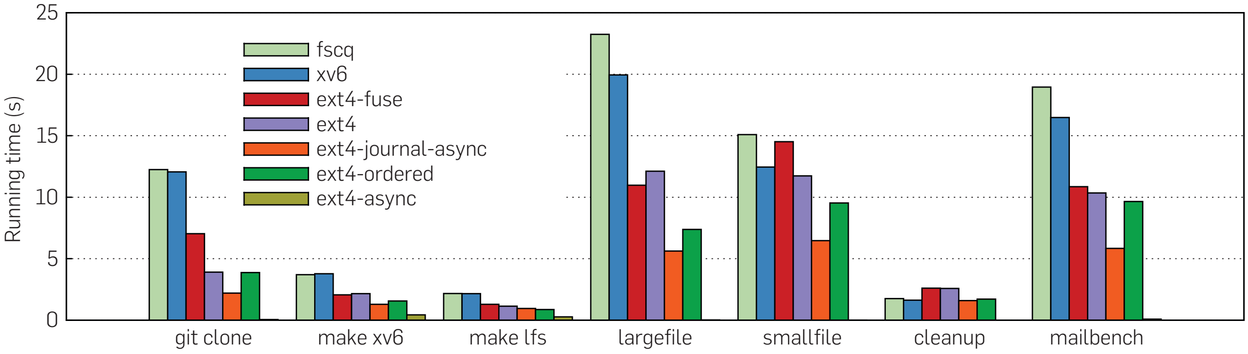
1. Violating file or directory invariants, such as all link counts adding up[36] or the absence of directory cycles.[26]
2. Improper handling of unexpected corner cases, such as running out of blocks during rename.[13]
3. Mistakes in logging and recovery logic, such as not issuing disk writes and syncs in the right order.[20]
4. Misusing the logging API, such as freeing an indirect block and clearing the pointer to it in different transactions.[19]
5. Low-level programming errors, such as integer overflows[21] or double frees.[4]
6. Resource allocation bugs, such as losing disk blocks[37] or returning ENOSPC when there is available space.[27]
7. Returning incorrect error codes.[5]
8. Bugs due to concurrent execution of system calls, such as races between two threads allocating blocks.[25]

Some categories of bugs (#1–5) are eliminated by FSCQ's theorems and proofs. For example, FSCQ's representation invariant for the entire file system enforces a tree shape for it, and the specification guarantees that it will remain a tree (without cycles) after every system call.

With regards to resource allocation (#6), FSCQ guarantees resources are never lost, but our prototype's specification does not *require* that the system be out of resources in order to return an out-of-resource error. Strengthening the specification (and proving it) would eliminate this class of bugs.

Incorrect error codes (#7) can be a problem for our FSCQ prototype in cases where we did not specify what exact code (e.g., EINVAL or ENOTDIR) should be returned. Extending the specification to include specific error codes could avoid these bugs, at the cost of more complex specifications and proofs. On the other hand, FSCQ can never have a bug where an operation fails without an error code being returned.

Figure 9. Running time for each phase of the application benchmark suite and for delivering 200 messages with mailbench.



Multiprocessor bugs (#8) are out of scope for our FSCQ prototype, because it does not support multithreading.

5.3. Crash recovery

We proved that FSCQ implements its specification, but in principle it is possible that the specification is incomplete or that our unproven code (e.g., the FUSE driver) has bugs. To do an end-to-end check, we ran two experiments. First, we ran `fsstress` from the Linux Test Project, which issues random file-system operations; this did not uncover any problems. Second, we experimentally induced crashes and verified that each resulting disk image after recovery is consistent.

In particular, we created an empty file system using `mkfs`, mounted it using FSCQ's FUSE interface, and then ran a workload on the file system. The workload creates two files, writes data to the files, creates a directory and a subdirectory under it, moves a file into each directory, moves the subdirectory to the root directory, appends more data to one of the files, and then deletes the other file. During the workload, we recorded all disk writes and syncs. After the workload completed, we unmounted the file system and constructed all possible crash states. We did this by taking a prefix of the writes up to some sync, combined with every possible subset of writes from that sync to the next sync. For the workload described above, this produced 320 distinct crash states.

For each crash state, we remounted the file system (which runs the recovery procedure) and then ran a script to examine the state of the file system, looking at directory structure, file contents, and the number of free blocks and inodes. For the above workload, this generates just 10 distinct logical states (i.e., distinct outputs from the examination script). Based on a manual inspection of each of these states, we conclude that all of them are consistent with what a POSIX application should expect. This suggests that FSCQ's specifications, as well as the unverified components, are likely to be correct.

5.4. Development effort

The final question is, how much effort is involved in developing FSCQ? One metric is the size of the FSCQ code base, reported in Figure 8; FSCQ consists of about 30,000 lines of code. In comparison, the xv6 file system is about 3000 lines of C code, so FSCQ is about 10× larger, but this includes a significant amount of CHL infrastructure, including libraries and proof machinery, which is not FSCQ-specific.

A more interesting question is how much effort is required to *modify* FSCQ, after an initial version has been developed and certified. Does adding a new feature to FSCQ require reproving everything, or is the work commensurate with the scale of the modifications required to support the new feature? To answer this question, the rest of this section presents several case studies, where we had to add a significant feature to FSCQ after the initial design was already complete.

Asynchronous disk writes. We initially developed FSCQ to operate with synchronous disk writes. Implementing asynchronous disk writes required changing about 1000 lines of code in the CHL infrastructure and changing over half of the implementations and proofs for the write-ahead log. However, layers above the log did not require any changes, since the log provided the same synchronous disk abstraction in both cases.

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

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

5.5. Evaluation summary

Although FSCQ is not as complete and high-performance as today's high-end file systems, our results demonstrate that this is largely due to FSCQ's simple design. Furthermore, the case studies in Section 5.4 indicate that one can improve FSCQ incrementally. In future work, we hope to improve FSCQ's logging to batch transactions and to log only metadata; we expect this will bring FSCQ's performance closer to that of ext4's logging. The one exception to incremental improvement is multiprocessor support, which may require global changes and is an interesting direction for future research.

6. CONCLUSION

This paper's contributions are CHL and a case study of applying CHL to build FSCQ, the first certified crash-safe file system. CHL allowed us to concisely and precisely specify the expected behavior of FSCQ. Via this verification approach, we arrive at a machine-checked proof that FSCQ avoids bugs that have a long history of causing data loss in previous file systems. For this kind of critical infrastructure, the cost of proving seems a reasonable price to pay. We hope that others will find CHL useful for constructing crash-safe storage systems.

Acknowledgments

The authors would like to thank Nathan Beckmann, Butler Lampson, Robert Morris, and the IronClad team for insightful discussions and feedback. The author would also like to thank the anonymous reviewers for their comments, and to our SOSP shepherd Herbert Bos and CACM Research Highlights editor Martin Abadi. This research was supported, in part, by NSF awards CNS-1053143 and CCF-1253229, Google, and CyberSecurity@CSAIL.

References

1. Amani, S., Hixon, A., Chen, Z., Rizkallah, C., Chubb, P., O'Connor, L., Beeren, J., Nagashima, Y., Lim, J., Sewell, T., Tuong, J., Keller, G., Murray, T., Klein, G., Heiser, G. Cogent: Verifying high-assurance file system implementations. In *Proceedings of the 21th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Atlanta, GA, Apr. 2016), 175–188.
2. Bobbio, J. et al. Haskell bindings for the FUSE library, 2014. https://github.com/m15k/hfuse.
3. Chen, H., Ziegler, D., Chajed, T., Chlipala, A., Kaashoek, M.F., Zeldovich, N. 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. Chinner, D. xfs: Fix double free in xlog_recover_commit_trans, Sept. 2014. http://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/commit/?id=88b863db97a18a04c90ebd57d84e1b7863114dcb.
5. Chinner, D. xfs: xfs_dir_fsync() returns positive errno, May 2014. https://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/commit/?id=43ec1460a2189fbee87980dd3d3e64cba2f11e1f.
6. Chlipala, A. 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)* (San Jose, CA, June 2011), 234–245.
7. Clements, A.T., Kaashoek, M.F., Zeldovich, N., Morris, R.T., Kohler, E. The scalable commutativity rule: Designing scalable software for multicore processors. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)* (Farmington, PA, Nov. 2013), 1–17.
8. Coq development team. *The Coq Proof Assistant Reference Manual, Version 8.5pl1.* INRIA, Apr. 2016. http://coq.inria.fr/distrib/current/refman/.
9. Cox, R., Kaashoek, M.F., Morris, R.T. Xv6, a simple Unix-like teaching operating system, 2014. http://pdos.csail.mit.edu/6.828/2014/xv6.html.
10. Ernst, G., Pfahler, J., Schellhorn, G., Reif, W. Inside a verified flash file system: Transactions & garbage collection. In *Proceedings of the 7th Working Conference on Verified Software: Theories, Tools and Experiments* (San Francisco, CA, July 2015).
11. Freitas, L., Woodcock, J., Butterfield, A. POSIX and the verification grand challenge: A roadmap. In *Proceedings of 13th IEEE International Conference on Engineering of Complex Computer Systems* (Belfast, Northern Ireland, Mar.–Apr. 2008), 153–162.
12. FUSE: Filesystem in userspace, 2013. http://fuse.sourceforge.net/.
13. Goldstein, A. ext4: Handle errors in ext4_rename, Mar. 2011. https://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/commit/?id=ef6078930263bfcdcfe4dddb2cd85254b4cf4f5c.
14. Gray, J. Notes on data base operating systems. In *Operating Systems: An Advanced Course*, R. Bayer, R.M. Graham, and G. Seegmuller, eds. Springer-Verlag, London, UK, 1978, 393–481.
15. Hawblitzel, C., Howell, J., Lorch, J.R., Narayan, A., Parno, B., Zhang, D., Zill, B. Ironclad Apps: End-to-end security via automated full-system verification. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)* (Broomfield, CO, Oct. 2014), 165–181.
16. Hoare, C.A.R. An axiomatic basis for computer programming. *Commun. ACM 12* 10 (Oct. 1959), 576–580.
17. IEEE (The Institute of Electrical and Electronics Engineers) and The Open Group. The Open Group base specifications issue 7, 2013 edition (POSIX.1–2008/Cor 1–2013), Apr. 2013.
18. Joshi, R., Holzmann, G.J. A mini challenge: Build a verifiable filesystem. *Formal Aspects Comput. 19*, 2 (June 2007), 269–272.
19. Kara, J. ext3: Avoid filesystem corruption after a crash under heavy delete load, July 2010. https://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/commit/?id=f25f624263445785b94f39739a6339ba9ed3275d.
20. Kara, J. jbd2: Issue cache flush after checkpointing even with internal journal, Mar. 2012. http://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/commit/?id=79feb521a44705262d15cc819a4117a447b11ea7.
21. Kara, J. ext4: Fix overflow when updating superblock backups after resize, Oct. 2014. http://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/commit/?id=9378c6768e4fca48971e7b6a9075bc006eda981d.
22. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Norrish, M., Kolanski, R., Sewell, T., Tuch, H., Winwood, S. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)* (Big Sky, MT, Oct. 2009), 207–220.
23. Leroy, X. Formal verification of a realistic compiler. *Commun. ACM 52,* 7 (July 2009), 107–115.
24. Lu, L., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H., Lu, S. A study of Linux file system evolution. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)* (San Jose, CA, Feb. 2013), 31–44.
25. Manana, F. Btrfs: Fix race between writing free space cache and trimming, Dec. 2014. http://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/commit/?id=55507ce3612365a5173dfb080a4baf45d1ef8cd1.
26. Mason, C. Btrfs: Prevent loops in the directory tree when creating snapshots, Nov. 2008. http://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/commit/?id=ea9e8b11bd1252dcbc23afefcf1a52ec6aa3c113.
27. Morton, A. [PATCH] ext2/ext3-ENOSPC bug, Mar. 2004. https://git.kernel.org/cgit/linux/kernel/git/tglx/history.git/commit/?id=5e9087ad3928c9d80cc62b583c3034f864b6d315.
28. Ntzik, G., da Rocha Pinto, P., Gardner, P. Fault-tolerant resource reasoning. In *Proceedings of the 13th Asian Symposium on Programming Languages and Systems (APLAS)* (Pohang, South Korea, Nov.–Dec. 2015).
29. Pillai, T.S., Chidambaram, V., Alagappan, R., Al-Kiswany, S., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H. All file systems are not created equal: On the complexity of crafting crash-consistent applications. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)* (Broomfield, CO, Oct. 2014), 433–448.
30. Reynolds, J.C. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science* (Copenhagen, Denmark, July 2002), 55–74.
31. Rosenblum, M., Ousterhout, J. The design and implementation of a log-structured file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP)* (Pacific Grove, CA, Oct. 1991), 1–15.
32. Schellhorn, G., Ernst, G., Pfahler, J., Haneberg, D., Reif, W. Development of a verified flash file system. In *Proceedings of the ABZ Conference* (Toulouse, France, June 2014).
33. Schlichting, R.D., Schneider, F.B. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Trans. Comput. Syst. 1*, 3 (1983), 222–238.
34. Tweedie, S.C. Journaling the Linux ext2fs filesystem. In *Proceedings of the 4th Annual LinuxExpo* (Durham, NC, May 1998).
35. Wang, X., Lazar, D., Zeldovich, N., Chlipala, A., Tatlock, Z. Jitk: A trustworthy in-kernel interpreter infrastructure. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)* (Broomfield, CO, Oct. 2014), 33–47.
36. Wong, D.J. ext4: Fix same-dir rename when inline data directory overflows, Aug. 2014. https://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/commit/?id=d80d448c6c5bdd32605b78a60fe8081d82d4da0f.
37. Xie, M. Btrfs: Fix broken free space cache after the system crashed, June 2014. https://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/commit/?id=e570fd27f2c5d7eac3876bccf99e9838d7f911a3.
38. Yang, J., Twohey, P., Engler, D., Musuvathi, M. eXplode: A lightweight, general system for finding serious storage system errors. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)* (Seattle, WA, Nov. 2006), 131–146.
39. Zheng, M., Tucek, J., Huang, D., Qin, F., Lillibridge, M., Yang, E.S., Zhao, B.W., Singh, S. Torturing databases for fun and profit. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)* (Broomfield, CO, Oct. 2014), 449–464.

Tej Chajed, Haogang Chen, Adam Chlipala, M. Frans Kaashoek, Nickolai Zeldovich, and Daniel Ziegler ({tchajed, dmz}@mit.edu, {hchen, adamc, kaashoek, nickolai}@csail.mit.edu), MIT CSAIL, Cambridge, MA.