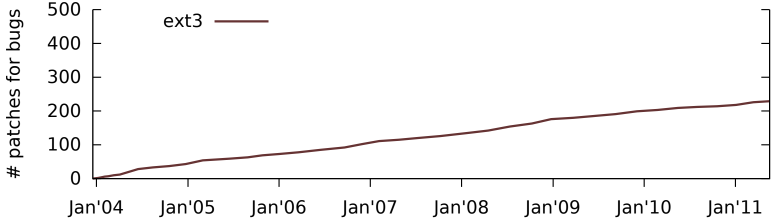
Certifying a file system: Correctness in the presence of crashes

Tej Chajed, Haogang Chen, Stephanie Wang, Daniel Ziegler, Adam Chlipala, Frans Kaashoek, and Nickolai Zeldovich

MIT CSAIL

File systems are complex and have bugs

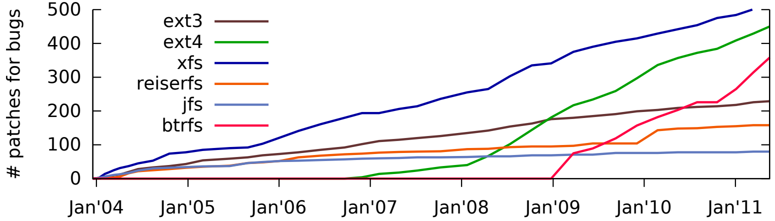
File systems are complex (e.g., Linux ext4 is ∼60,000 lines of code) and have many bugs:



Cumulative number of patches for file-system bugs in Linux; data from [Lu et al., FAST’13]

File systems are complex and have bugs

File systems are complex (e.g., Linux ext4 is ∼60,000 lines of code) and have many bugs:

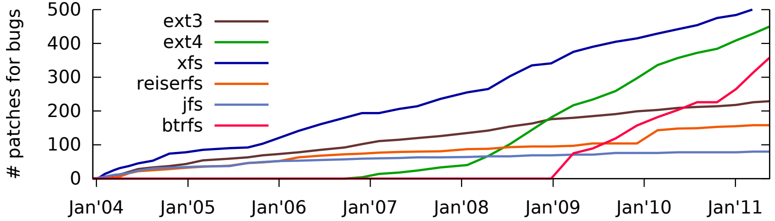


Cumulative number of patches for file-system bugs in Linux; data from [Lu et al., FAST'13]

New file systems (and bugs) are introduced over time

File systems are complex and have bugs

File systems are complex (e.g., Linux ext4 is ∼60,000 lines of code) and have many bugs:



Cumulative number of patches for file-system bugs in Linux; data from [Lu et al., FAST'13]

New file systems (and bugs) are introduced over time

Some bugs are serious: **security exploits**, **data loss**, etc.

Much research in avoiding bugs in file systems

Most research is on finding bugs:

- Crash injection (e.g., EXPLODE [OSDI'06])
- Symbolic execution (e.g., EXE [Oakland'06])
- Design modeling (e.g., in Alloy [ABZ'08])

Some elimination of bugs by proving:

- FS without directories [Arkoudas et al. 2004]
- BilbyFS [Keller et al. 2014]
- Flashix [Ernst et al. 2015]

Much research in avoiding bugs in file systems

Most research is on finding bugs:

- Crash injection (e.g., EXPLODE [OSDI'06])
- Symbolic execution (e.g., EXE [Oakland'06])
- Design modeling (e.g., in Alloy [ABZ'08])

} Reduce # bugs

Some elimination of bugs by proving:

- FS without directories [Arkoudas et al. 2004]
- BilbyFS [Keller et al. 2014]
- Flashix [Ernst et al. 2015]

Much research in avoiding bugs in file systems

Most research is on finding bugs:

- Crash injection (e.g., EXPLODE [OSDI'06])
- Symbolic execution (e.g., EXE [Oakland'06])
- Design modeling (e.g., in Alloy [ABZ'08])

$\Big\}$ Reduce # bugs

Some elimination of bugs by proving:

- FS without directories [Arkoudas et al. 2004]
- BilbyFS [Keller et al. 2014]
- Flashix [Ernst et al. 2015]

$\Big\}$ Incomplete + no crashes

File system must preserve data after crash

Crashes occur due to power failures, hardware failures, or software bugs

Difficult because crashes expose many different partially-updated states

```
commit 353b67d8ced4dc53281c88150ad295e24bc4b4c5
--- a/fs/jbd/checkpoint.c
+++ b/fs/jbd/checkpoint.c
@@ -504,7 +503,25 @@ int cleanup_journal_tail(journal_t *journal)
                spin_unlock(&journal->j_state_lock);
                return 1;
        }
+       spin_unlock(&journal->j_state_lock);
+
+       /*
+        * We need to make sure that any blocks that were recently written out
+        * --- perhaps by log_do_checkpoint() --- are flushed out before we
+        * drop the transactions from the journal. It's unlikely this will be
+        * necessary, especially with an appropriately sized journal, but we
+        * need this to guarantee correctness.  Fortunately
+        * cleanup_journal_tail() doesn't get called all that often.
+        */
+       if (journal->j_flags & JFS_BARRIER)
+               blkdev_issue_flush(journal->j_fs_dev, GFP_KERNEL, NULL);

+       spin_lock(&journal->j_state_lock);
+       if (!tid_gt(first_tid, journal->j_tail_sequence)) {
+               spin_unlock(&journal->j_state_lock);
+               /* Someone else cleaned up journal so return 0 */
+               return 0;
+       }
        /* OK, update the superblock to recover the freed space.
         * Physical blocks come first: have we wrapped beyond the end of
         * the log?  */
```

File system must preserve security after crash

Mistakes in crash handling can also lead to data disclosure

- Two optimizations in Linux ext4: direct data write and log checksum
- Subtle interaction: new file can contain other users' data after crash
- Bug introduced in 2008, fixed in 2014 (six years later!)

```
Author: Jan Kara <jack@suse.cz>
Date:   Tue Nov 25 20:19:17 2014 -0500

    ext4: forbid journal_async_commit in data=ordered mode

    Option journal_async_commit breaks gurantees of data=ordered mode as it
    sends only a single cache flush after writing a transaction commit
    block. Thus even though the transaction including the commit block is
    fully stored on persistent storage, file data may still linger in drives
    caches and will be lost on power failure. Since all checksums match on
    journal recovery, we replay the transaction thus possibly exposing stale
    user data.

    [...]
```

Goal: certify a complete file system under crashes

- A file system with a **machine-checkable proof**
- that its implementation meets **its specification**
- under **normal execution**
- and under any sequence of **crashes**
- including **crashes during recovery**

Contributions

CHL: Crash Hoare Logic for persistent storage

- Crash condition and recovery semantics
- CHL automates parts of proof effort
- Proofs mechanically checked by Coq

FSCQ: the first certified crash-safe file system

- Basic Unix-like file system (not parallel)
- Simple specification for a subset of POSIX
  - E.g., no hard links or symbolic links
- About 1.5 years of work, including learning Coq

FSCQ runs standard Unix programs: `mv`, `git`, `make`, ...

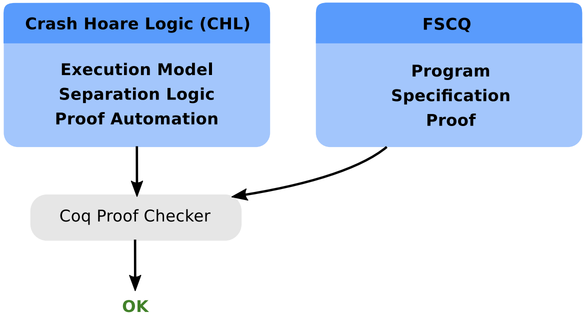
Crash Hoare Logic (CHL)

Execution Model
Separation Logic
Proof Automation

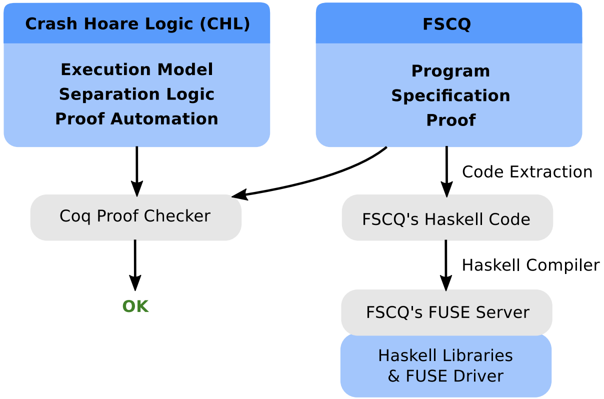
FSCQ

Program
Specification
Proof

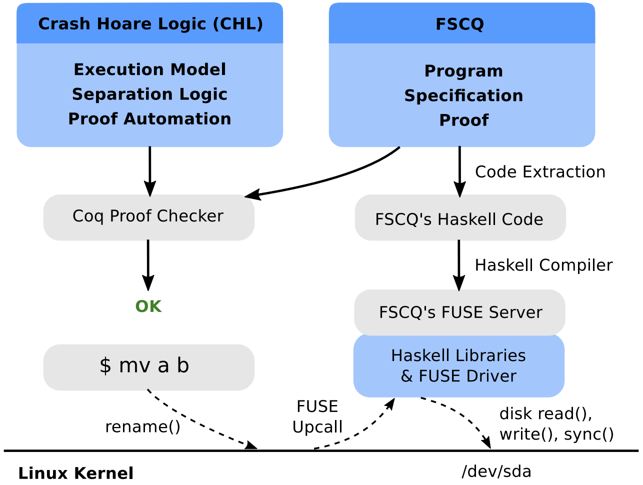
FSCQ runs standard Unix programs: `mv`, `git`, `make`, ...



FSCQ runs standard Unix programs: `mv`, `git`, `make`, ...



FSCQ runs standard Unix programs: `mv`, `git`, `make`, ...



TCB includes Coq's extractor, Haskell compiler and runtime, ...

How to specify what is “correct”?

Need a specification of “correct” behavior before we can prove anything

Look it up in the POSIX standard?

How to specify what is "correct"?

Need a specification of "correct" behavior before we can prove anything

Look it up in the POSIX standard?

> *[...] a power failure [...] can cause data to be lost. The data may be associated with a file that is still open, with one that has been closed, with a directory, or with any other internal system data structures associated with permanent storage. This data can be lost, in whole or part, so that only careful inspection of file contents could determine that an update did not occur.*
>
> IEEE Std 1003.1, 2013 Edition

POSIX is vague about crash behavior

- POSIX's goal was to specify "common-denominator" behavior
- File system implementations have different interpretations
- Leads to bugs in higher-level applications [Pillai et al. OSDI'14]

A starting point: “correct” is transactional

Run every file-system call inside a transaction

```
def create(d, name):
    log_begin()
    newfile = allocate_inode()
    newfile.init()
    d.add(name, newfile)
    log_commit()
```

A starting point: “correct” is transactional

Run every file-system call inside a transaction

```
def create(d, name):
    log_begin()
    newfile = allocate_inode()
    newfile.init()
    d.add(name, newfile)
    log_commit()
```

`log_begin` and `log_commit` implement a write-ahead log on disk

After crash, replay any committed transaction in the write-ahead log

A starting point: “correct” is transactional

Run every file-system call inside a transaction

```
def create(d, name):
    log_begin()
    newfile = allocate_inode()
    newfile.init()
    d.add(name, newfile)
    log_commit()
```

`log_begin` and `log_commit` implement a write-ahead log on disk

After crash, replay any committed transaction in the write-ahead log

Q: How to formally specify both normal-case and crash behavior?

Q: How to specify that it’s safe to crash during recovery itself?

Approach: Hoare Logic specifications

{pre} code {post}

SPEC	disk_write(a, v)
PRE	$a \mapsto v_0$
POST	$a \mapsto v$

CHL extends Hoare Logic with crash conditions

{pre} `code` {post}
{crash}

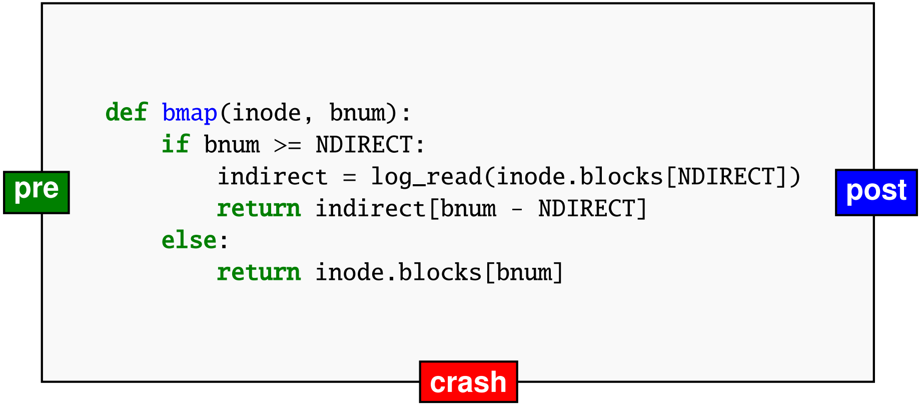
SPEC	disk_write(a, v)
PRE	$a \mapsto v_0$
POST	$a \mapsto v$
CRASH	$a \mapsto v_0 \lor a \mapsto v$

CHL's disk model matches what most other file systems assume:

- writing a single block is an atomic operation
- no data corruption

Specifications for `disk_write`, `disk_read`, and `disk_sync` represent our disk model

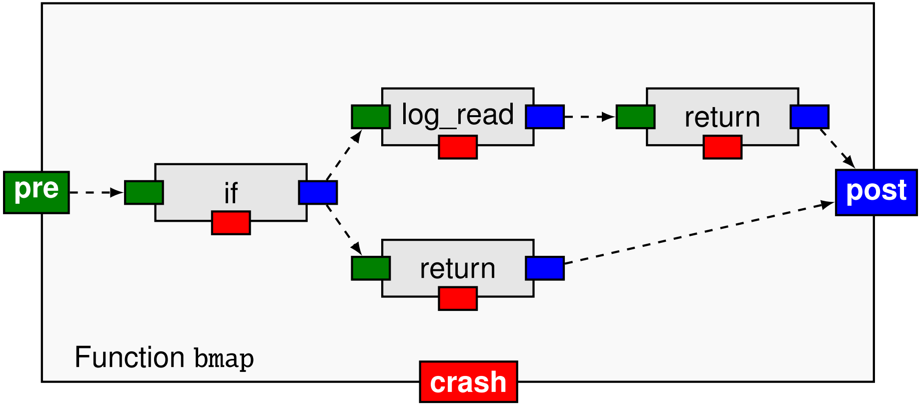
Certifying larger procedures



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

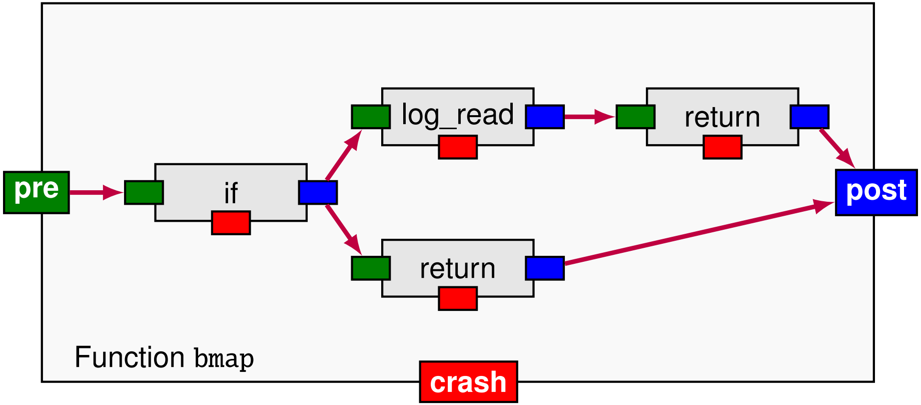
Certifying larger procedures

Need pre/post/crash conditions for each called procedure



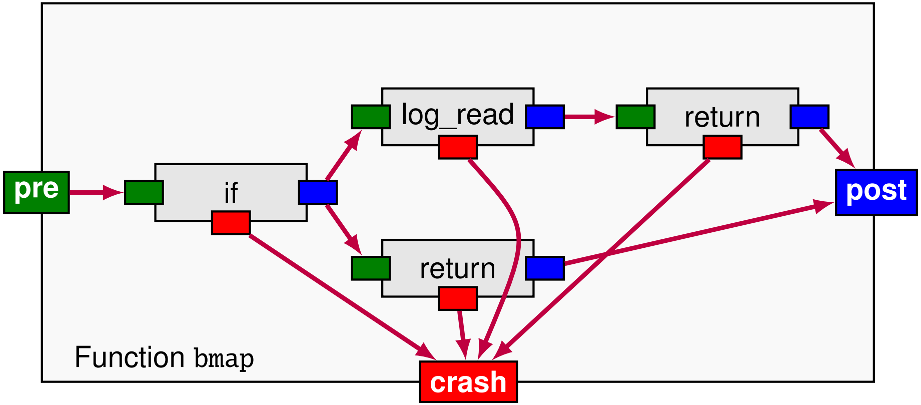
Certifying larger procedures

CHL’s proof automation chains pre- and postconditions



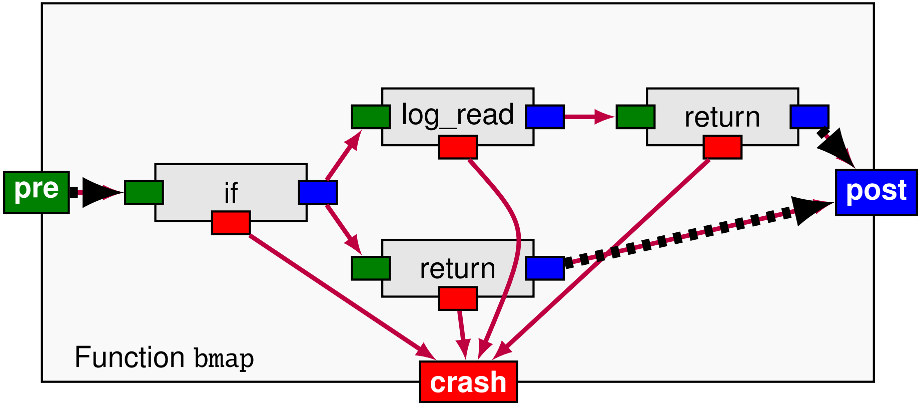
Certifying larger procedures

CHL's proof automation combines crash conditions



Certifying larger procedures

Remaining proof effort: changing representation invariants



Common pattern: representation invariant

SPEC	log_write(a, v)
PRE	**disk**: log_rep(ActiveTxn, *start_state*, *old_state*)
	old_state: $a \mapsto v_0$
POST	**disk**: log_rep(ActiveTxn, *start_state*, *new_state*)
	new_state: $a \mapsto v$
CRASH	**disk**: log_rep(ActiveTxn, *start_state*, *any*)

`log_rep` is a representation invariant

- Connects logical transaction state to an on-disk representation
- Describes the log's on-disk layout using many $\mapsto$ primitives

Specifying an entire system call (simplified)

SPEC create(*dnum*, *fn*)
PRE **disk**: log_rep(NoTxn, *start_state*)
start_state: dir_rep(*tree*) $\wedge$
$\exists$ *path*, *tree*[*path*].inode = *dnum* $\wedge$
fn $\notin$ *tree*[*path*]

Specifying an entire system call (simplified)

SPEC create(*dnum*, *fn*)

PRE **disk**: log_rep(NoTxn, *start_state*)

start_state: dir_rep(*tree*) $\wedge$

$\exists$ *path*, *tree*[*path*].inode = *dnum* $\wedge$

fn $\notin$ *tree*[*path*]

POST **disk**: log_rep(NoTxn, *new_state*)

new_state: dir_rep(*new_tree*) $\wedge$

new_tree = *tree*.update(*path*, *fn*, empty_file)

Specifying an entire system call (simplified)

SPEC create(*dnum*, *fn*)

PRE **disk**: log_rep(NoTxn, *start_state*)

start_state: dir_rep(*tree*) $\wedge$
$\exists$ *path*, *tree*[*path*].inode = *dnum* $\wedge$
fn $\notin$ *tree*[*path*]

POST **disk**: log_rep(NoTxn, *new_state*)

new_state: dir_rep(*new_tree*) $\wedge$
new_tree = *tree*.update(*path*, *fn*, empty_file)

CRASH **disk**: log_rep(NoTxn, *start_state*) $\vee$
log_rep(NoTxn, *new_state*) $\vee$
$\exists$ *s*, log_rep(ActiveTxn, *start_state*, *s*) $\vee$
log_rep(CommittedTxn, *start_state*, *new_state*) $\vee \ldots$

Specifying log recovery

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

`log_recover` is idempotent

- Crash condition implies pre condition
- $\Rightarrow$ OK to run `log_recover` *again* after a crash

CHL's recovery semantics

`create` is atomic, if `log_recover` runs after every crash:

SPEC	create(*dnum*, *fn*)
ON CRASH	log_recover()
PRE	**disk**: log_rep(NoTxn, *start_state*)
	start_state: dir_rep(*tree*) $\wedge$
	$\exists$ *path*, *tree*[*path*].inode = *dnum* $\wedge$
	fn $\notin$ *tree*[*path*]
POST	**disk**: log_rep(NoTxn, *new_state*)
	new_state: dir_rep(*new_tree*) $\wedge$
	new_tree = *tree*.update(*path*, *fn*, empty_file)
RECOVER	**disk**: log_rep(NoTxn, *start_state*) $\vee$
	log_rep(NoTxn, *new_state*)

CHL summary

Key ideas: crash conditions and recovery semantics

CHL benefit: enables precise failure specifications

- Allows for automatic chaining of pre/post/crash conditions
- Reduces proof burden

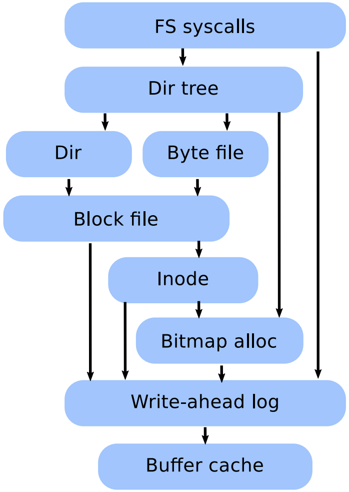
CHL cost: must write crash condition for every function, loop, etc.

- Crash conditions are often simple (above logging layer)

FSCQ: building a file system on top of CHL

File system design is close to v6 Unix,
plus logging, minus symbolic links

Implementation aims to reduce proof effort



Evaluation

What bugs do FSCQ's theorems eliminate?

How much development effort is required for FSCQ?

How well does FSCQ perform?

FSCQ's theorems eliminate many bugs

One data point: once theorems proven, no implementation bugs

- Did find some mistakes in spec, as a result of end-to-end checks
- E.g., forgot to specify that extending a file should zero-fill

FSCQ's theorems eliminate many bugs

One data point: once theorems proven, no implementation bugs

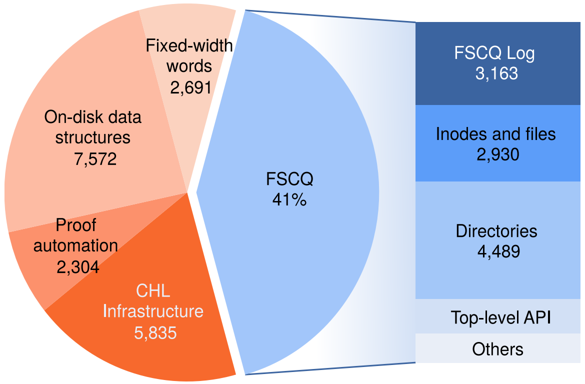
- Did find some mistakes in spec, as a result of end-to-end checks
- E.g., forgot to specify that extending a file should zero-fill

Common classes of bugs found in Linux file systems:

Bug class	**Eliminated in FSCQ?**
Violating file or directory invariants	Yes
Improper handling of corner cases	Yes
Returning incorrect error codes	Some
Resource-allocation bugs	Some
Mistakes in logging and recovery logic	Yes
Misusing the logging API	Yes
Bugs due to concurrent execution	No concurrency
Low-level programming errors	Yes

Implementing CHL and FSCQ in Coq

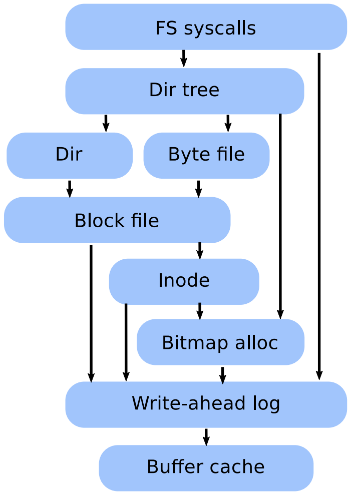
Total of ~30,000 lines of **verified** code, specs, and proofs
Comparison: xv6 file system is ~3,000 lines of code



Change effort proportional to scope of change

- Reordering disk writes:
  ~1,000 lines in FSCQLOG
- Indirect blocks:
  ~1,500 lines in inode layer
- Buffer cache:
  ~300 lines in FSCQLOG,
  ~600 lines in rest of FSCQ
- Optimize log layout:
  ~150 lines in FSCQLOG

Modest incremental effort, partially due to CHL's proof automation and FSCQ's internal layers



Performance comparison

File-system-intensive workload

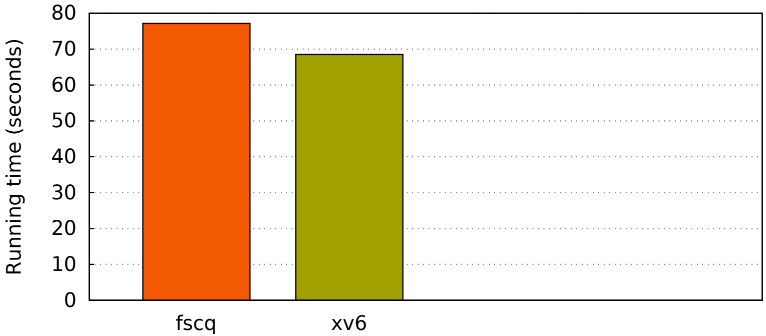
- Software development: git, make
- LFS benchmark
- `mailbench`: qmail-like mail server

Compare with other (non-certified) file systems

- xv6 (similar design, written in C)
- ext4 (widely used on Linux), in non-default *synchronous* mode to match FSCQ's guarantees

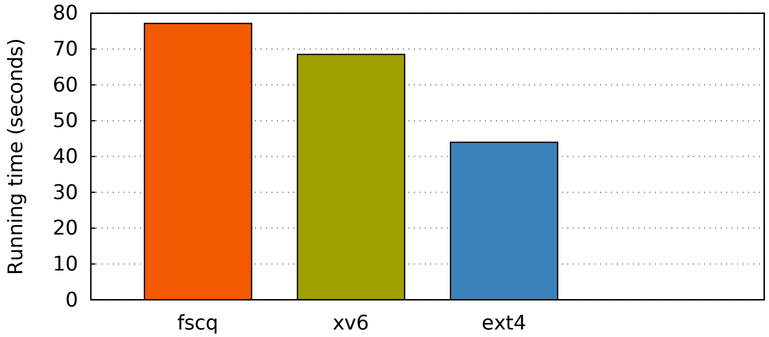
Running on an SSD on a laptop

Running time for benchmark workload



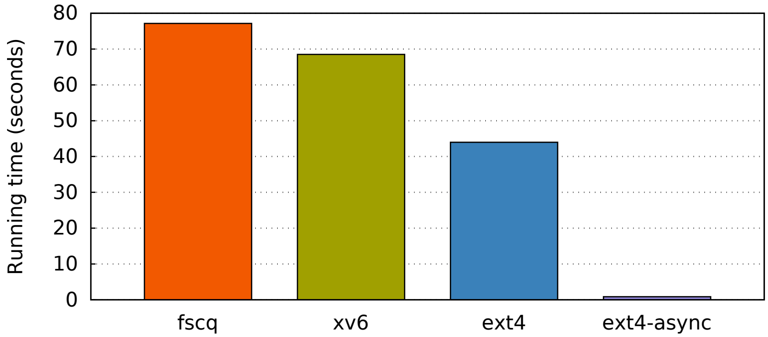
- FSCQ slower than xv6 due to overhead of extracted Haskell

Running time for benchmark workload



- FSCQ slower than xv6 due to overhead of extracted Haskell
- FSCQ slower than ext4 due to simple write-ahead logging design

Opportunity: change semantics to defer durability



- FSCQ slower than xv6 due to overhead of extracted Haskell
- FSCQ slower than ext4 due to simple write-ahead logging design
- Deferred durability (ext4's default mode) allows for big improvement

Directions for future research

Certifying deferred durability (`fsync`)

- Preliminary results: much of the effort is in coming up with a good specification
- `FSCQ` on par with ext4-async in terms of disk I/O (took ∼1 year of extra work)

Certifying a parallel (multi-core) file system

Certifying applications with CHL (mail server, key-value store, ...)

Reducing TCB size and generating efficient executable code

Eventual goal: provable end-to-end app security

Formal verification turned a corner

- May soon be practical for engineering secure systems

Can we prove end-to-end security for a complete application?

- E.g., messages typed by Alice should appear only on Bob's phone

Many challenges at the intersection of systems and verification

- Hardware support for isolating unverified code
- Composing different types of specifications and proofs
- Verification of software updates

Eventual goal: provable end-to-end app security

Formal verification turned a corner

- May soon be practical for engineering secure systems

Can we prove end-to-end security for a complete application?

- E.g., messages typed by Alice should appear only on Bob's phone

Many challenges at the intersection of systems and verification

- Hardware support for isolating unverified code
- Composing different types of specifications and proofs
- Verification of software updates

Big opportunity to address fundamental security problems

Conclusions

CHL helps specify and prove crash safety

- Crash conditions
- Recovery execution semantics

FSCQ: first certified crash-safe file system

- Usable performance
- 1.5 years of effort, including learning Coq and building CHL

Many open problems and potential for fundamental contributions

`https://github.com/mit-pdos/fscq-impl`