Specifying crash safety for storage systems

Specifying Crash Safety for Storage Systems

Haogang Chen, Daniel Ziegler, Adam Chlipala, M. Frans Kaashoek, Eddie Kohler†, Nickolai Zeldovich
MIT CSAIL and †Harvard University

1 Introduction

Software that is provably correct has been a long-time goal of computer science. Until recently this goal was realized for only small programs, but over the last decade several large systems have been built that have provable correctness properties. Examples include CompCert [21], seL4 [20], IronClad [13], CertiKOS [12], Bedrock [4, 5], Termite [32], Click's dataplane [8], and Jitk [35]. One aspect not covered by these systems is reasoning about failures—power failures, hardware faults, or software bugs—which is well-known to be tricky in systems code.

An important example of where failures matter is a file system, because developers often make subtle mistakes in recovery code, and recovery code is complex and not frequently executed. Even if such bugs are rare, they can still be costly, since they can lead to complete data loss [41]. Proving the absence of such bugs in critical software, such as a file system, is an appealing proposition.

To explore what it takes to certify a file system, we are in the process of building one such file system and its machine-checked proof. Our goal is to certify a simple Unix-like file system with logging. We want to prove that our file system's implementation is correct: it matches the specification, even if there are crashes.

There are many aspects involved in implementing a certified file system, but the design choice on which everything else depends is the question of how to write specifications. We found that writing specification is surprisingly tricky. This paper summarizes what we have learned from exploring several specification strategies, including one approach that has worked well for us.

Writing a precise specification assumes that you know what the specification is. Unfortunately, POSIX is not well specified: file systems have many different interpretations of what POSIX means, in particular under failures. Furthermore, file systems allow operators to configure them to have different behaviors under failure, which can result in surprising results for application writers [29]. In this paper, we adopt a simple compromise: each system call runs atomically. If there is a failure, either the system call happened completely or not all. The system call never leaves the file system in an intermediate state.

There are many ways to write a specification for atomic systems calls, like there are many ways to design and implement an API, and it was initially unclear to us which one was the right choice. What criteria determine if a specification is a good one? Our three goals were (1) to prevent real bugs, (2) to enable proof automation, and (3) to allow for modularity. The rest of this paper explores this question by examining different approaches to specify the behavior of a file system under crashes. We report on what approaches we discarded and describe why an approach based on Hoare logic with an extensions for crash predicates and recovery semantics works well.

2 File system bugs

We consider four broad categories of bugs in real-world file systems [22, 40] that the specifications must handle.

Sequential bugs. Many bugs arise even without concurrency or failures: (1) Low-level bugs, such as integer overflows [19] or double-frees [2]; (2) Violating directory invariants, such as link counts adding up [37], or lack of directory cycles [24]; (3) Improper handling of corner cases, such as writing to a large offset in a sparse file [33], or running out of blocks during `rename` [11]; (4) Returning incorrect error codes [3]; (5) Resource allocation bugs, such as losing disk blocks [38] or returning `ENOSPC` when space is available [26].

Concurrency bugs. Bugs due to concurrent system calls, such as a race when two threads allocate blocks [23].

Recovery. Bugs in the transaction logic that implements recovery after a crash, such as not issuing disk writes and flushes in the right order [18].

Misusing transactions. Finally, file systems developers sometimes misuse transactions, such as forgetting to allocate data blocks as part of the same transaction that updates the inode size [25], or freeing an indirect block and clearing the pointer to it in different transactions [17].

3 Specification styles

To prove the correctness of a file system, we need to formalize the specification of what it means for a file system to be correct. This section describes several approaches for writing down specifications, using a running example of a file-system create operation.

Traces. One approach for reasoning about the file-system interface is to use execution traces, as introduced by Herlihy and Wing [14]. Traces have been used for reasoning about system-call commutativity [6] and for verifying distributed systems [36]. This led us to believe they would be a good fit for capturing the concurrency in a file system, such as multiple threads, crashes, etc.

```
op1:  create("/tmp", "foo")
      crash
op2:  open("/tmp/foo")
op2:  return OK, fd = 5
```

(a) valid

```
op1:  create("/tmp", "foo")
      crash
op2:  open("/tmp/foo")
op2:  return OK, fd = 5
      crash
op2:  open("/tmp/foo")
op2:  return Error, ENOENT
```

(b) invalid

Figure 1: Trace examples. Crashes are highlighted in red.

In the traces approach, the specification is a predicate for a trace that determines whether a trace—consisting of operation invocations and responses to them—is valid. Figure 1(a) shows an example valid trace involving the `create` system call (for now, imagine that instead of `crash`, op1 returns OK). The specification itself is written in some formal language, such as Coq [7] in the case of Verdi [36]. A proof in this trace approach has to demonstrate that any sequence of operations generated by an implementation is legal according to the specification.

Behavioral. Behavioral specifications prove the equivalence of two implementations of some function f. One of the implementations (the "specification") is typically written in a high-level pseudo-code-like language, which incorporates non-determinism and higher-level abstractions (e.g., data structures like sets and lists, rather than raw bytes of disk/memory). The specification code indirectly describes how f modifies the state. This approach is used by seL4 [20], and in the context of file systems by BilbyFS [1] and VFS [9].

```
function CREATE(dirpath, filename)
    if * then CRASH
    dir ← LOOKUP(dirpath)
    if filename ∈ dir then
        ret ← EEXIST
    else
        f ← new FILE
        dir ← dir ∪ {filename ↦ f}
        ret ← OK
    if * then CRASH
    return ret
```

Figure 2: Behavioral specification for an atomic `create`.

Figure 2 shows a sketch of a specification for file creation, inspired by specifications from BilbyFS and VFS; for now, ignore the crash statements. Here, `dir` is a dictionary mapping directory entries to files, which avoids having to reason about the low-level layout of a directory on disk. Of course, the proof has to demonstrate that the implementation follows this abstract specification.

Hoare logic. Instead of specifying that the real implementation of f is equivalent to some abstract implementation of f, Hoare logic [10, 15] reasons about what predicates hold before and after f runs. Hoare specifications are written as `{pre} f {post}`, meaning that, if precondition `pre` holds before function f runs, then postcondition `post` will be true afterwards. Unlike the approaches we sketcher earlier, the specification is exactly the theorem that needs to be proved about the implementation: for any state that matches `pre`, the result after running f's implementation will match `post`. Hoare logic is used in Verve [39], IronClad [13], and Bedrock [4].

```
{ rep(tree) }
  create(dirpath, filename)
{ ret = Err ∧ rep(tree)   ∨
  ret = OK ∧ rep(tree')
  where  tree' = tree_upd( tree, dirpath, filename) }
  AFTER RECOVERY USING  log_recover
{ rep(tree) ∨ rep(tree')
  where  tree' = tree_upd( tree, dirpath, filename) }
```

Figure 3: Hoare-style specification for `create` using crash predicates and recovery execution semantics.

For example, Figure 3 shows the Hoare specification for `create`; for now, ignore the "after recovery" portion. Here, `rep(tree)` is a predicate that describes the state of the disk corresponding to a logical tree structure `tree` (e.g., using separation logic [30] to precisely describe the on-disk state) and `tree_upd` is a function that adds a new entry to this logical tree.

Domain-specific languages. An approach commonly used in the programming languages community is to introduce a domain-specific language (DSL) to represent the high-level interface exported by some module (e.g., a file system). Formalizing the specification involves writing down precise semantics for this language, including a logical model of the state, how a program executes in this language, and how each primitive operation in this language (e.g., `create`) affects the logical state; these are typically written down as small-step semantics [21]. This approach is used by the CompCert compiler [21] and the CertiKOS operating system [12].

In this approach, an implementation is a compiler that takes a program written in one language (e.g., the system call interface) and translates it into a lower-level language (e.g., another DSL that supports reading and writing disk blocks). Proving the correctness of an implementation entails establishing a correspondence between the high-level and low-level DSLs, and proving that a translated program running in the low-level DSL behaves exactly as the original program interpreted according to the high-level DSL's semantics.

For example, Figure 4 shows a sketch of a specification for a DSL that includes a `create` operation. The semantics of a DSL is essentially a *simple reference implementation* as an interpreter. Here we write the interpreter in OCaml notation, though implementations com-

```
let rec interpreter ((program : dsl_program),
                     (state : tree)) : retCode * tree =
  match program with
    Create (dirpath, filename) ->
      nondeterministic_choice:
         (Err, state)
      || (OK, tree_upd (state, dirpath, filename))
  | (* ...other operations here... *)
  | Sequence (prog1, prog2) ->
    let (ret1, state1) = interpreter (prog1, state) in
    interpreter (prog2 (ret1), state1)
```

Figure 4: Sketch of a DSL-style specification for `create`.

monly use logic programs in the style of Prolog (otherwise known as operational semantics). The example recursive interpreter here makes crucial use of an imaginary `nondeterministic_choice` construct in OCaml, which allows us to say that some piece of code goes down one of several control-flow paths unpredictably. The particular use of nondeterminism here is to say that any system call may fail (here, return `Err`) for any reason; but we also specify which *state change* occurs in both the error case (here, no change) and the success case (here, returning `OK`, accompanied by an updated file-system state).

Other approaches. The list above is not exhaustive in its coverage of ways to specify and verify the correctness of a system. One notable alternative, used by model checking [8] and synthesis [32], is to express specifications as a series of assertions, written in the same programming language as the program being verified. Verification boils down to ensuring that the assertions are never triggered.

4 Certifying Crash Safety

In order to reason about crashes using any of the specifications from §3, we have to address three key questions: (1) How to specify the failure model at the low level?; (2) How to capture semantics in case of crashes at the high level?; and (3) How to reason about recovery logic that runs after a crash before the system resumes operation (e.g., log recovery)?

Traces. Modeling both low- and high-level crashes with traces is, at some level, straightforward. We introduce a new event that can show up in a trace, `crash`, which signifies that the machine crashed, and specifications still reason about which traces (now including `crash`) are valid. Figure 1(a) shows an example of a valid trace of disk block operations involving a `crash`; here, `open` is allowed to return either OK or `ENOENT` because the file might or might not have been created when the crash occurred.

One difficulty with traces comes from reasoning about state, which is not made explicit. For example, Figure 1(b) shows a trace that is not valid: once a system crashes in the middle of `create`, it commits to some state—either the create happened or not—and should not change after future crashes. Capturing this can be tedious without an explicit notion of state, and we made several such mistakes in our trace-based file-system specification attempts.

Reasoning about recovery functions with traces boils down to treating the `crash` event as an implicit invocation of some recovery code. The recovery code could finish, represented by another event in the trace, or the machine could crash before recovery finishes, represented by another `crash` event.

Behavioral. There are several ways to reason about crashes in the behavioral approach. One plan is to introduce a `crash` statement that represents the system crashing. A specification can describe possible crash points by inserting non-deterministic calls to `crash`. At the low level, a specification can capture the notion of atomic sector writes by introducing non-deterministic `crash` calls before and after updating the disk state in the disk write function. At the high level, a specification can capture the notion of an atomic system call by non-deterministically calling `crash` just at the beginning and end of an operation, as shown in Figure 2. Reasoning about recovery boils down to associating some function with the `crash` statement, which can then be thought of as a `goto` statement to that function.

Inserting `crash` statements is a good fit for specifying operations that are atomic with respect to crashes. However, it is awkward to use the `crash` statement to specify more complex crash scenarios, where the crash state does not correspond to any point in the pseudocode. For instance, consider the specification in Figure 2, where the new file is initialized before the directory entry is added. There is no easy way to add a `crash` statement to model a possible crash scenario where the directory entry has been added but the new file has not been initialized, which can happen with asynchronous disk writes.

An alternative way to model crashes in behaviors, explored by Pfähler et al [28], is to reason about logical sets of all possible states after a crash. The specification can then non-deterministically select one of the states after a crash. Reasoning about recovery can probably be done in a similar way to the `crash` statement, although Pfähler, Schellhorn, et al have only shown that this works for recovering in-memory state, and it is not clear if this works for recovering the on-disk contents [34].

Hoare logic. To reason about crashes in Hoare logic, we came up with the notion of a *crash predicate*, which describes all possible states in which a program might crash. For example, consider a function `write_pair` that writes to two disk blocks. Figure 5 shows our crash-predicate-style specification for this function, assuming that the two blocks are distinct, and that each block write is itself atomic. The crash predicate enables us to specify both low-level failure model (in the crash condition of the

disk write function) and high-level crash semantics (by describing the states in which the system might crash).

$$
\begin{array}{ll}
 & \{\ \text{disk}[a_0] = x_0 \wedge \text{disk}[a_1] = x_1\ \} \\
 & \quad \texttt{write\_pair}(a_0, v_0, a_1, v_1) \\
 & \{\ \text{disk}[a_0] = v_0 \wedge \text{disk}[a_1] = v_1\ \} \\
\text{AFTER CRASH:} & \{\ \text{disk}[a_0] = x_0 \wedge \text{disk}[a_1] = x_1 \vee \\
 & \quad \text{disk}[a_0] = v_0 \wedge \text{disk}[a_1] = x_1 \vee \\
 & \quad \text{disk}[a_0] = v_0 \wedge \text{disk}[a_1] = v_1\ \}
\end{array}
$$

Figure 5: Hoare logic specification with crash predicates for a function that writes to two disk blocks.

To reason about recovery, we came up with a new *recovery execution semantics* that captures both the notion of crashes and of jumps to recovery code. This enables us to write specifications such as the one shown in Figure 3, which says that the entire `create` system call is atomic, after the recovery function runs (and cleans up non-atomic intermediate states).

One downside of this approach above is that it requires modifying the Hoare logic and the execution semantics to model crashes. This would be a non-trivial change for an existing system like Dafny [31], which has been built around traditional Hoare logic and execution semantics.

An alternative approach is to model crashes as nondeterministic calls to a `crash` function, much as we suggested earlier for behavioral specifications. In this plan, the crash predicate effectively becomes the precondition of the `crash` function. However, this approach does not let the developer specify *different* crash predicates for different functions, and does not directly allow reasoning about recovery.

Domain-specific languages. To model crashes in the DSL approach, we added crashes to the execution semantics of each language, much as described above in Hoare logic. One downside of DSLs is the fact that the notion of crashes has to be explicitly incorporated into each DSL (e.g., low-level disk, top-level file-system API, intermediate inode and directory APIs, etc). For instance, to add crashes to Figure 4, we had to add an extra nondeterministic choice at the top level of the interpreter, to either crash or run the usual code for the first instruction of the program. Such a change captures crash phenomena like losing some pending writes.

Another complexity is that it can be difficult to reason about recovery logic, because some intermediate states might not be representable in the top-level DSL. For instance, suppose that `create` crashes after allocating an inode but before adding it to a directory. There may be no way to describe this state in the top-level DSL, if it reasons about the state as a complete tree. As a result, we found no easy way to reason about programs without considering the recovery function, and found this to be a significant source of complexity. This motivated us to split the crash predicate from the recovery execution semantics in our design, as described above.

Bug class	Traces	Behavior	Hoare	DSL
Sequential bugs	N	Y	Y	Y
Concurrency	Y	N	N	N
Recovery	N	N	Y	Y
Mis-using txns	Y	Y	Y	Y

Figure 6: Evaluating whether a proof approach is a good match for reasoning about a bug category.

5 Evaluation

This section qualitatively evaluates how well the different specification approaches achieve our three goals: preventing real bugs, enabling proof automation, and allowing for modularity. Although it's difficult to give hard evidence for any of these results, our results are based on our experience trying to implement a file system using several of these approaches, as well as some speculation, since we did not build a complete system with each approach.

5.1 Bugs

Figure 6 summarizes our qualitative experience by indicating which specification approaches are a good match for reasoning about the classes of bugs we introduced in §2. We find that traces are a natural fit for reasoning about concurrency and about prefix properties (e.g., using transactions). No other approach is as good of a match for reasoning about concurrency. On the other hand, traces do not help much for sequential bugs and transaction internals, because they are not a good match for reasoning about state. Behavioral specs are a good fit for sequential bugs and for ensuring the transactional API is used correctly. However, they have trouble reasoning about asynchronous disk writes, which arise in the internals of a logging system or recovering on-disk state. The challenge comes from out-of-order writes, which result in crash states that do not correspond to any sequential execution of a prefix of the behavioral spec. Hoare logic and DSLs can handle most bug classes except for concurrency. (Many concurrent extensions of Hoare logic have been developed [16, 27], but the formal-methods community has yet to converge on the one appropriate base strategy, to the extent that it has for the foundation of sequential program verification that we have adopted.)

Additionally, we found that significant care is needed when writing specifications; even if a specification approach is a good match, it is easy to write an incomplete specification that does not eliminate the possibility of some bugs. For example, in order to ensure that proper error codes are returned, the specification has to precisely define what error code is appropriate in every situation.

One way to ensure that a specification is good enough is to try to use it in the next level up (e.g., building an application on top of a file system) and prove application-level properties using the underlying spec. For example, our initial specification for file writes forgot to mention that file attributes (e.g., inode type) were preserved. We discovered this while proving a stronger property (that the entire file system formed a tree).

We found it cumbersome to write liveness specifications. For instance, our current specs do not exclude bugs where the file system returns an out-of-space error despite there being available resources. Instead, our specification allows all writes to return `ENOSPC`. Similarly, it is easy to write specifications that could lose resources (e.g., losing track of a disk block if a crash occurs at an inopportune time). Separation logic [30] helps avoid such bugs.

5.2 Proof effort and automation

We found that for traces and behavioral specifications, proofs are mostly manual, which mirrors the experience reported by seL4 developers in their proof effort [20].

The Hoare-style approach is more amenable to automation, because it has clear, mechanical rules for how to reason about programs (e.g., if a program invokes the disk write function, the proof system can mechanically determine what predicate holds after the disk write returns, using the specification of the disk write function). Most systems based on Hoare logic provide significant automation [4, 5, 13, 31], and we were able to extend this style of automated reasoning to cover crashes. The two places where manual effort is required is in defining loop invariants for every single `for` loop, and in proving representation invariants (e.g., that the entire directory tree is still intact after a directory entry is modified).

DSLs require more proof effort than Hoare logic, because each DSL introduces its own execution model. This requires proving that the different execution models are equivalent, which takes effort (as observed by the CompCert authors, backward simulation proofs that are necessary to do so are tedious [21]).

5.3 Modularity

We do not have sufficient experience with traces or behavioral specifications to say how well they handle modularity. We expect modularity to be somewhat tricky for traces, and a reasonable fit for behavioral specs.

With Hoare logic, we believe that introducing crash predicates and recovery execution semantics was a significant improvement for modularity. The alternative, using a global crash invariant, forces the developer to commit to a crash invariant early on, and makes it difficult to develop modules of the file system (e.g., logging, bitmaps, inodes, directories) in isolation. A further concern is that the global crash invariant may have to involve the application using the file system (e.g., a mail server), if the invariant has to connect the on-disk state (e.g., which files are on disk) to application-visible state (e.g., which SMTP messages were acknowledged), which again makes it difficult to develop modules (and applications) in isolation.

We found that DSLs provide strong modularity, but also that the hard modularity boundaries can be a burden. For example, we tried to define separate languages for reasoning about inodes, directories, and data blocks. However, the strict static partitioning between these languages, and between the disk blocks used by them, was at odds with the needs of a file system, where the same disk blocks can store either file data or directory entries.

5.4 File system

As an end-to-end evaluation, we are building a certified file system using the Hoare logic approach with crash predicates and recovery semantics. This approach has been working out well for us. Our specification eliminates the possibility of most bugs that we considered in this paper, except for multi-thread concurrency, which we do not know how to handle. We find that the Hoare approach leads to relatively short proofs, that the proof effort was modest, and that the proofs and specifications are modular. For example, we were able to change the design of our logging subsystem without changing any higher-level code or specifications, and we were able to add indirect blocks to our file system with similarly minimal effort.

6 Summary

Specifying crash-safe storage systems is challenging. We experimented with a number of specification approaches in developing a certified file system. Although the approaches share many similarities, we had to discard several of them because the approach was not a good match for our goals: it either failed to capture real bugs, was not amenable to proof automation, or did not allow for modularity. After trial and error, we found that Hoare logic augmented with crash predicates and recovery execution semantics achieves our goals. An interesting direction for future work may be integrating the best parts of each approach, such as integrating a trace-based approach to concurrency with Hoare logic. We hope the qualitative discussion and evaluation in this paper helps inform future work in reasoning about crashes.

Acknowledgments

Thanks to Butler Lampson, Robert Morris, and the IronClad team for insightful discussions, and to anonymous reviewers for helpful feedback. This work was supported by NSF awards CNS-1053143, CNS-1413920, and CCF-1253229.

REFERENCES

[1] S. Amani, Z. Chen, G. Keller, T. Murray, L. Ryzhyk, G. Klein, and G. Heiser. File system verification & synthesis, 2013. http://www.ssrg.nicta.com.au/Events/summer/13/Amani.pdf.

[2] D. Chinner. 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.

[3] D. Chinner. xfs: xfs_dir_fsync() returns positive errno, May 2014. https://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/commit/?id=43ec1460a2189fbee87980dd3d3e64cba2f11e1f.

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

[6] A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–17, Farmington, PA, Nov. 2013.

[7] Coq development team. *Coq Reference Manual, Version 8.4pl5*. INRIA, Oct. 2014. http://coq.inria.fr/distrib/current/refman/.

[8] M. Dobrescu and K. Argyraki. Software dataplane verification. In *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI)*, Seattle, WA, Apr. 2014.

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

[10] R. W. Floyd. Assigning meanings to programs. In *Proceedings of the American Mathematical Society Symposia on Applied Mathematics*, volume 19, pages 19–31, 1967.

[11] A. Goldstein. ext4: handle errors in ext4_rename, Mar. 2011. https://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/commit/?id=ef6078930263bfcdcfe4dddb2cd85254b4cf4f5c.

[12] 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.

[13] 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.

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

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

[16] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4), 1983.

[17] J. Kara. 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.

[18] J. Kara. 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.

[19] J. Kara. 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.

[20] 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.

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

[22] L. Lu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and S. Lu. A study of Linux file system evolution. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)*, pages 31–44, San Jose, CA, Feb. 2013.

[23] F. Manana. 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.

[24] C. Mason. 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.

[25] D. Monakhov. ext4: fix transaction issues for ext4_fallocate and ext_zero_range, Aug. 2014. http://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/commit/?id=c174e6d6979a04b7b77b93f244396be4b81f8bfb.

[26] A. Morton. [PATCH] ext2/ext3 -ENOSPC bug, Mar. 2004. https://git.kernel.org/cgit/linux/kernel/git/tglx/history.git/commit/?id=5e9087ad3928c9d80cc62b583c3034f864b6d315.

[27] P. W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007.

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

[29] T. S. Pillai, V. Chidambaram, R. Alagappan, S. Al-Kiswany, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. All file systems are not created equal: On the complexity of crafting crash-consistent applications. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 433–448, Broomfield, CO, Oct. 2014.

[30] 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.

[31] K. Rustan and M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the LPAR-16*, 2010.

[32] L. Ryzhyk, A. Walker, J. Keys, A. Legg, A. Raghunath, M. Stumm, and M. Vij. User-guided device driver synthesis. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, Broomfield, CO, Oct. 2014.

[33] E. Sandeen. ext4: fix async i/o writes beyond 4GB to a sparse file, Feb. 2010. https://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/commit/?id=a1de02dccf906faba2ee2d99cac56799bda3b96a.

[34] G. Schellhorn, G. Ernst, J. Pfähler, D. Haneberg, and W. Reif. Development of a verified flash file system. In *Proceedings of the ABZ Conference*, June 2014.

[35] X. Wang, D. Lazar, N. Zeldovich, A. Chlipala, and Z. Tatlock. Jitk: A trustworthy in-kernel interpreter infrastructure. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 33–47, Broomfield, CO, Oct. 2014.

[36] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson. Verdi: A framework for formally verifying distributed system implementations. In *Proceedings of the 2015 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Portland, OR, June 2015.

[37] D. J. Wong. 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.

[38] M. Xie. 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.

[39] 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.

[40] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. EXPLODE: A lightweight, general system for finding serious storage system errors. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 131–146, Seattle, WA, Nov. 2006.

[41] M. Zheng, J. Tucek, D. Huang, F. Qin, M. Lillibridge, E. S. Yang, B. W. Zhao, and S. Singh. Torturing databases for fun and profit. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 449–464, Broomfield, CO, Oct. 2014.