COGENT: Verifying High-Assurance File System Implementations

Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb,
Liam O'Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, Joseph Tuong,
Gabriele Keller, Toby Murray, Gerwin Klein, Gernot Heiser

Data61 (formerly NICTA) and UNSW, Australia
first.last@data61.csiro.au

Abstract

We present an approach to writing and formally verifying high-assurance file-system code in a restricted language called COGENT, supported by a certifying compiler that produces C code, high-level specification of COGENT, and translation correctness proofs. The language is strongly typed and guarantees absence of a number of common file system implementation errors. We show how verification effort is drastically reduced for proving higher-level properties of the file system implementation by reasoning about the generated formal specification rather than its low-level C code. We use the framework to write two Linux file systems, and compare their performance with their native C implementations.

Categories and Subject Descriptors D.4.5 [*Operating Systems*]: Reliability—Verification; D.2.4 [*Software Engineering*]: Software / Program Verification—Formal methods; D.3.2 [*Programming Languages*]: Language Classification—Applicative (functional) languages

Keywords file systems; verification; domain-specific languages; co-generation; Isabelle/HOL

1. Introduction

Operating systems (OS) code is critical for the dependability and security of computer systems. In monolithic systems, most OS services are part of the kernel and are included in the trusted computing base (TCB) of any application. While microkernels can help reduce the TCB [10, 17, 29, 47], in practice we need more, because most applications depend on the correct operation of a significant number of OS services.

ASPLOS '16, April 02 – 06, 2016, Atlanta, GA, USA.

ACM 978-1-4503-4091-5/16/04. . . $15.00.
http://dx.doi.org/10.1145/10.1145/2872362.2872404

File systems, which constitute the largest fraction of code in Linux after device drivers, have among the highest defect density of Linux kernel code [38]. Furthermore, new file systems are frequently added for new media types or for new performance and reliability trade-offs. The lack of support for correctly writing new file systems is a standard bottleneck.

Formal verification in the style of seL4 [26] or Ironclad [15] is the only approach to fully solve this problem. While formal verification for file systems has seen a surge in activity in recent years [9, 11, 18], it has however not yet reached that same level of detail and assurance. In the most advanced and deepest of these verifications, Chen et al. [8] formally shows crash-resilience of the FSCQ file system implemented in the Coq prover [6]. Even this verification still relies on generating Haskell code from the Coq implementation, and executing that generated code with a full Haskell run-time at user level. This proof crosses an important barrier for formal verification: it removes human intervention from the link between the code that runs and the model that is verified. However, the trusted code base is still huge: the run-time is larger than the file system, the code generation step from Coq is unverified, and the semantics of the target language (Haskell) is complex and informal. While certainly an impressive result, for high-assurance systems on potentially resource-constrained devices, this is not yet good enough.

As motivated in [23], we design a new language called COGENT to bridge the gap between verifiable formal model and low-level code. In separate work [36], we define the formal semantics of COGENT, describe the COGENT compiler in detail which generates C code and an Isabelle/HOL specification, and show how the seven main stages of automatic compiler-produced proof work that connect the C and the Isabelle/HOL specification. In this paper, we show how to reason about COGENT programs, how COGENT is used to implement two Linux file systems, and we analyse how COGENT reduces the effort of reasoning about a low-level C implementation in order to more productively reason about high-level functional specifications — similar in style to Chen et al's verification. To this end, we verified two

high-level functional correctness properties of one of the two Linux file systems. These are properties that would form key building blocks in, for instance, a proof of crash resilience.

COGENT goes further than just bridging the gap from model to code. It also provides tools for file system programmers *without any formal verification expertise* to *provably* avoid common errors: The language is type safe, i.e. the type system is strictly enforced. The compiler and type system in turn enforces basics like memory safety, but also the absence of any undefined behaviour on the C level, null pointer dereferences, buffer overflows, memory leaks, and pointer mismanagement in error handling. Mismanagement of pointers in error-handling code is a wide-spread problem in Linux file systems specifically [41], and Saha et al. [42] shows that file systems have among the highest density of error-handling code in Linux. In the verification of seL4, a significant fraction of the overall effort went into proving absence of such errors [25] – COGENT's *linear type system* [49] provides this for free and assists the programmer with memory allocation handling. Memory safety and memory leaks are problems with file systems, but also more generally in systems code [43]. The infamous Heartbleed bug for instance, was a buffer overflow [16], the recent "goto-fail" defect in Apple's SSL/TLS implementation was an error-handling problem obscured by `goto`s in an `if`-cascade [37], and the recent memory leak in Android Lollipop also was part of error-handling code [30]. All of these problems are prevented on the language level by COGENT, and are enforced by its certifying compiler. Few other languages come with type-system guarantees that address problems like memory leaks without significant overhead or large language run-times (e.g. Rust), and none come with a formal semantics and a machine-checked proof of their guarantees.

COGENT compiles to straight C code that can be compiled by gcc or CompCert [27, 28]. The generated C code also falls into the fragment understood by Sewell et al.'s gcc translation validation tool [46], providing a pathway to extend the verification all the way to binary level if needed. We designed COGENT with the code patterns and common errors in Linux file systems in mind. As detailed in separate work [36], its semantics is sequential (allowing asynchronous I/O, but not full concurrency), restricted to total functions, and contains no built-in loops or recursion. This simplifies reasoning, both for the compiler and on top of the language. Iterators, external abstract functions, and types that rely on sharing, are implemented in a formally modelled foreign function interface, supported by a custom template-style C extension. In our file system implementation, a small library of abstract data types (ADTs) and iterators is sufficient, and the foreign-function interface is powerful enough to provide interoperability with an existing red-black tree implementation in C.

In summary, we make the following contributions:

1. We demonstrate the use of COGENT for implementing actual file systems (Section 3).
2. We present proofs of high-level correctness properties of a file system implemented in COGENT and discuss how COGENT facilitates such proofs (Section 4).
3. We discuss the effort involved in implementing file systems with verified properties in COGENT (Section 5).
4. We evaluate the COGENT-implemented file systems against their native C counterparts (Section 5).

2. Overview of Our Approach

2.1 COGENT

COGENT is tailored to writing systems code, with a specific focus on file systems. It is intentionally more restrictive than general-purpose functional languages, to enable the use of efficient and powerful reasoning techniques on its high-level semantics, to avoid the need for an elaborate language run-time, and to enable the COGENT compiler to output an automatic proof that the C code it generates correctly implements the behaviour of the COGENT program [36]. In this paper, we further demonstrate how to prove functional correctness of the COGENT program.

COGENT manages to avoid a garbage collector, unlike most existing high-level languages such as Java, Haskell, and OCaml, by implementing a *linear type system* [49], which ensures that each linearly typed object is used exactly once. The type system not only provides memory safety, but also makes memory leaks compile-time errors. Traditional pure functional languages, such as Haskell, that disallow side effects, favour frequent copies of dynamically allocated data structures and make it hard to reason about the performance of the generated code. COGENT's linear type system increases performance by allowing the compiled code to modify data structures in-place. The type system supports parametric polymorphism. While doing so might not be an obvious choice for a language geared towards systems programming, it is particularly important in the context of COGENT, which relies on external ADTs to implement data structures that cannot be implemented with linear types.

COGENT also supports higher-order functions. In contrast to general-purpose functional languages, however, all functions in COGENT have to be defined on the top-level. This restriction is necessary to avoid the run-time overhead and heap allocations necessary to implement closures otherwise.

As an example, Figure 1 depicts a snippet of COGENT from our ext2 implementation, specifically the `ext2_inode_get()` function. This function looks up an inode on disk, given its inode number `inum`. Lines 3–4 declare the type of this function. It takes 3 arguments: an `ExState`, which is a reference to an ADT embodying the outside environment of the file system; an `FsState`, a reference to a COGENT structure that holds the file system state, and a `U32`, an unsigned 32-bit integer that in this case is the

```
1  type RR c a b = (c, <Success a | Error b>)
2
3  ext2_inode_get : (ExState, FsState, U32) ->
4        RR (ExState, FsState) (VfsInode) (U32)
5  ext2_inode_get (ex, state, inum) =
6      let ((ex, state), res) =
7        ext2_inode_get_buf (ex, state, inum)
8      in res
9      | Success (buf_blk, offset) ->
10       -- read the inode, from the offset
11       let ((ex, state), res) =
12         deserialise_Inode (ex, state,
13                            buf_blk, offset,
14                            inum)
15         !buf_blk
16       in res
17       | Success inode ->
18           let ex =
19             osbuffer_destroy (ex, buf_blk)
20           in ((ex, state), Success inode)
21       | Error () ->
22           let ex =
23             osbuffer_destroy (ex, buf_blk)
24           in ((ex, state), Error eIO)
25     | Error (err) -> ((ex, state), Error err)
26
27 osbuffer_destroy:(ExState, OsBuffer)->ExState
```

Figure 1. Looking up an inode Ext2, in COGENT.

number of the inode to be looked up. It returns a result of type `RR` (defined on line 1), which is a pair. Its first component `c` is mandatory data that must always be returned, while the second component is a *tagged union* value which signals whether the function succeeded (`Success a`) or not (`Error b`), where `a` and `b` are result values for the respective cases. `ext2_inode_get()` (line 4) always returns both the `ExState` and `FsState`; when successful it also returns a `VfsInode` reference to the looked-up inode; otherwise it instead returns also a `U32` error code.

`ext2_inode_get()` first calls (line 6–7) the COGENT function `ext2_inode_get_buf()`, which, after more calculation, internally calls an ADT function to read the corresponding block from disk. We match (lines 8, 9, 25) on the result `res` to make a case distinction: for `Success` (line 9), the result contains a reference `buf_blk` to a buffer, plus the offset `offset` of the inode in that buffer. We use these to deserialise the inode from the buffer into a `VfsInode` structure.

In this situation, linear types are unnecessarily restrictive. Passing the linear value `buf_blk` to `deserialise_Inode()` in line 12 would consume it, so `deserialise_Inode()` would have to return it as an additional result for it to be used again later on in the program. This is overly complicated, given that `deserialise_Inode()` never modifies the buffer. The "!" operator (in "`!buf_blk`", line 15) allows us temporarily to escape the use-once restriction of the linear type system: it indicates that within the right-hand side of `=` here, `buf_blk` is used read-only and can therefore be referenced multiple times without consuming it. The type system prevents `deserialise_Inode()` from modifying the buffer that is holding the inode being deserialised. Moreover, it ensures that no references to buffer or parts of it can be returned, as this would result in aliasing.

For the `Error` case (line 25), the result includes an error code that is simply propagated to the caller along with the return values that were mandatory. The `Success`/`Error` case distinction is repeated following the attempt to deserialise the inode from the buffer. In either case, the buffer must be released, by calling the ADT function `osbuffer_destroy()`, whose type signature (line 27) suggests it consumes the buffer.

Note that in this function, COGENT's linear type system would flag an error if the buffer `buf_blk` was never released. `ext2_inode_get()` will only type-check if all of the `Error` cases are handled. Also observe that the mandatory `ExState` and `FsState` values are threaded through the function. The `ExState` value encompasses all external state, and so includes for instance Linux's buffer cache. Because COGENT is a pure functional language, it makes side effects explicit. This means this `ExState` is passed to `osbuffer_destroy()` explicitly, which then returns a new `ExState` reference that encompasses the external state of the world after the buffer has been released. This does not mean that large structures are copied at run-time. In practice, COGENT's linear type system allows the `ExState` update in `osbuffer_destroy()` to be performed in-place. The same is true for the other functions that each consume and return an `ExState` or `FsState`.

It is obvious from Figure 1 that COGENT's current surface syntax for error handling is unnecessarily verbose. The textual overhead for the regular error handling pattern can easily be reduced by additional syntactic sugar in a future language iteration without touching the verification infrastructure – we merely focused on verification first.

2.2 COGENT File Systems

We implement two file systems in COGENT, to drive the development of the language and as case studies for automatic proof generation. The first is an almost feature-complete revision 1 ext2 implementation: it passes the Posix File System Test Suite [35], except for the ACL and symlink tests (as we have not implemented those features). Its performance is comparable to Linux's native ext2 implementation. For contrast, the second is a new flash file system called BilbyFs [23] whose design strikes a balance between the simplicity of JFFS2 and the performance of UBIFS, the two most popular Linux flash file systems.

The COGENT implementations of ext2 and BilbyFs share a common library of ADTs that includes fixed-length arrays for words and structures, simple iterators for implementing `for`-loops, and COGENT stubs for accessing a range of Linux APIs such as the buffer cache and its native red-black tree implementation, plus iterators for each ADT. The in-

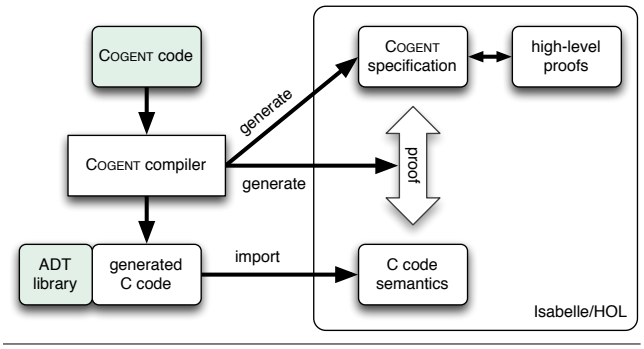

Figure 2. Code/Proof Co-Generation Process.

terfaces exposed by these ADTs are carefully designed to ensure compatibility with COGENT's linear type system.

For full functional correctness, these ADTs need to be verified separately. We have done so for a smaller instance (the `WordArray` ADT) to validate the cross-language semantics, but have not verified for instance the red-black tree implementation, although it is certainly feasible to do so. For example, a verification of a red-black tree implementation (although in a higher-order logic and thus not usable in our framework) is part of the Isabelle/HOL [34] standard library.

The ext2 implementation showcases COGENT's ability to enable the re-engineering of existing file systems, and thus its potential to provide an incremental upgrade path to increase the reliability of existing systems code. BilbyFs, on the other hand, provides a glimpse of how to design and engineer new file systems that are not only performant, but amenable to being verified as correct against a high-level specification of file system correctness. To this end, BilbyFs pursues aggressive modular decomposition, whereas the structure of the ext2 implementation mirrors that of its native Linux counterpart. Section 3 presents the two file systems in more detail.

2.3 Automatic Code/Proof Co-Generation

Details of the code/proof co-generation form the topic of separate work [36]. We provide a brief summary below.

Figure 2 shows the code and proof generation process: the COGENT compiler generates C code, which is then compiled by some C compiler, linking against the ADT library, to produce binary files. Besides the C code, the COGENT compiler also generates a formal specification in Isabelle/HOL that precisely encodes the semantics of the source COGENT program, and a refinement proof that this semantics is correctly implemented by the C code. The specification can be used to prove higher-level properties about the COGENT program.

The generated C code can be compiled by standard compilers such as gcc, and also CompCert. We assign the generated C language subset a semantics, which is from Norrish's *C Parser* [50], originally developed for the seL4 verification. Since Norrish's C Parser also supplies the C semantics for Sewell et al.'s gcc translation validation tool [46], we can in future extend the compiler-generated proofs for the C code to the binary level.

The refinement proofs state that every behaviour exhibited by the C code can also be exhibited by the COGENT code and, furthermore, that the C code is always well-defined, including that e.g. the generated C code never dereferences a null pointer, and never causes signed overflow. It also implies that the generated C code is type-safe and memory-safe, meaning the code will never try to dereference an invalid pointer, or try to dereference two aliasing pointers of incompatible types. In conjunction with the COGENT typing proofs, generated by the COGENT compiler for the input program, we get additional guarantees that the generated code handles all error cases, is free of memory leaks, and never *double-free*s a pointer.

These proofs do *not* guarantee, for instance, that the implementation always behaves as a proper file system — e.g. that data written to disk will always be able to be read back. Providing that kind of additional guarantee requires proving the COGENT code functionally correct against a high-level specification of file system functionality.

In later sections, we show how to reason on top of the COGENT formal specification in Isabelle/HOL, leveraging COGENT's purely functional semantics. Reasoning here is simpler than reasoning on C code directly because the COGENT semantics is represented as pure functions in the logic, making it possible to reason equationally about it via Isabelle's powerful rewriting engine. We return to this point in Section 4.

If desired, one could further formalise different logics on top of our generated COGENT specification to further simplify reasoning about different domain-specific properties. For example, a Crash Hoare Logic [8] to simplify reasoning about crash safety.

3. File Systems in COGENT

As mentioned in Section 2, we have implemented two file systems in COGENT: an ext2 implementation, and a new raw flash file system called BilbyFs. Both file system implementations sit below Linux's virtual file system switch (VFS) module, using C stub code to provide the top-level entry points expected by the VFS. These C stubs call directly into the top-level COGENT functions that implement each file system, using locking to prevent two COGENT functions from executing concurrently (because our current verification technology does not support concurrency).

An ADT provides a common interface to the VFS within COGENT, while each file system uses a separate ADT for interfacing with the physical storage medium, as these differ between the two. Both ADTs capture all relevant properties of the underlying media, and should be fully re-usable for other file systems targeting the same media.

3.1 ext2

ext2 is a well-known file system for block devices. While on large block devices it has long been supplanted by journaling file systems, which provide better reliability guarantees in the event of a crash, it remains popular for smaller block devices like SD cards and USB flash memory.

The COGENT ext2 implementation follows ext2fs of Linux 3.14.12; essentially we transliterated the Linux implementation into COGENT. This approach tests COGENT's ability to let systems programmers (re)implement their code, and in the process gain (proof-) guaranteed type- and memory-safety, as well as absence of undefined behaviour, missing error cases and memory leaks (as discussed in Section 1).

In its current state, our ext2 implementation has a number of limitations compared to Linux ext2fs. Each of these is straightforward to remove, except the lock that forces the code to run without concurrency. It emulates an early version (revision 1) of ext2, with 1k blocks and 128byte inodes. It also does not yet support read-ahead or direct-IO, and uses a simpler block allocation algorithm than Linux, so the order of blocks on disk is different, leading to different seek times, and different on-disk fragmentation. The implementation also currently elides extended attributes, quotas, reserved blocks and symlinks; however, none of these features are exercised by the benchmarks presented in Section 5.

3.2 BilbyFs

BilbyFs is meant to serve as a case study in how to design new file systems that are fully verifiable, meaning that not only is their generated C code proved to always implement their COGENT semantics, but that their COGENT semantics can also be verified to satisfy high-level correctness properties, such as full functional correctness against an abstract specification of file system correctness.

Unlike ext2, BilbyFs is not a block device file system. Instead it is a file system for raw flash devices, like those on embedded devices. BilbyFs' design draws inspiration from the two most popular raw flash file systems, JFFS2, which was merged into Linux in 2001; and the newer UBIFS, which was merged in 2008. BilbyFs' design balances the simplicity of JFFS2 with the run-time performance properties of UBIFS. Like each of these file systems, BilbyFs is a log-structured file system. Like JFFS2 and UBIFS, it provides crash-tolerance by structuring flash updates in atomic transactions, and discarding incomplete transactions when re-mounting following a crash.

Like UBIFS, BilbyFs writes data to the flash asynchronously, allowing otherwise small writes to be batched into large transactions to improve metadata packing and throughput [19]. Like JFFS2, however, BilbyFs eschews storing the flash *index*, which records the on-flash location of each file system object, on the flash. Instead it maintains the index in memory. This means that the index must be reconstructed at mount time, and limits the maximum medium size that can be supported.

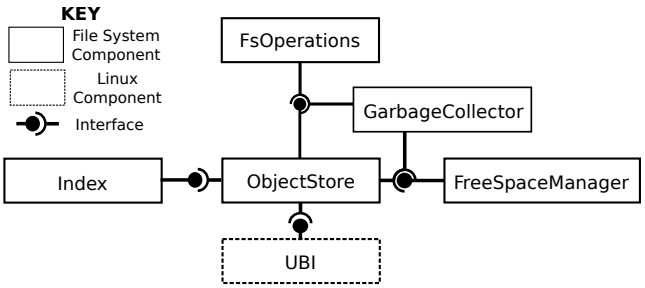

Figure 3. The modular design of BilbyFs.

As mentioned, BilbyFs' design is rooted in aggressive modular decomposition, to reduce the cost of completing a manual proof of full functional correctness of its COGENT code, building on top of the automatic correctness proofs generated by the COGENT compiler. Recent research [32] provides evidence suggesting that the effort required for manual verification scales quadratically with the size of the statement to be proved. This indicates that increasing modularity will reduce the overall verification effort.

BilbyFs' modular design is depicted in Figure 3. At the bottom level, BilbyFs interfaces with Linux's UBI component via a COGENT ADT. It uses UBI to read and write the flash, allowing UBI to handle wear levelling and manage logical erase blocks as it does for UBIFS. The in-memory Index component, one level up, implemented in COGENT, tracks the on-flash address of each file system object. The ObjectStore uses this Index to provide an abstract interface for reading and writing generic objects on flash. To do so it also makes use of the FreeSpaceManager, which in turn is used by the GarbageCollector to free erase blocks no longer in use. Finally, the FsOperations component implements the top-level file system operations and objects, like inodes directory entries and data blocks. This decomposition ensures that the key file system logic is confined to the FsOperations component, while the physical representation of objects on flash is handled by the ObjectStore.

3.3 Reusable Abstract Data Types

As mentioned, the two file systems share a common ADT library (7 ADTs in total). It includes the `WordArray` type mentioned in Section 2, for arrays of primitive words; a separate polymorphic `Array` type for (linear) heap values; common utility functions like iterators for implementing `for`-loops with early exit and accumulators, and (inline) functions for manipulating machine words; a heapsort implementation; and polymorphic linked lists. It also includes stubs for accessing existing kernel APIs, including the buffer cache (e.g. `osbuffer_destroy()` from Figure 1), the page cache, a native red-black tree implementation, checksum functions, time and date functions, and the VFS.

Some care is needed when designing ADTs that respect and enforce the constraints of COGENT's linear type system. For example, the function for accessing an element of the general polymorphic `Array` type must make sure that the element cannot be accessed a second time, inadvertently giving two writable references to a single value, which violates the constraints of the linear type system. This is why we have a separate `WordArray` type for strings of (non-linear) machine words. When accessing array-elements read-only, no such removal is needed: the type system guarantees that the aliasing is safe in such cases.

While designing and implementing ADTs thus requires some skill, it should be a relatively rare activity because ADTs are freely reusable. Given that our current ADT library is shared between the two relatively different file systems, we expect that it should be completely reusable for other file system implementations as well.

4. Formal Verification

Recall from Section 2.3 that the COGENT compiler generates C code as well as a high-level Isabelle/HOL specification from the input COGENT program. In this section, we describe how this high-level specification facilitates further, formal reasoning at much reduced effort compared to traditional functional correctness verification as typified by e.g. seL4 [26]. To this end, we describe two BilbyFs high-level functional correctness properties that we proved.

4.1 Top-Level Correctness Specification

These proofs show that the `sync()` and `iget()` operations of BilbyFs are *functionally correct*, meaning that they behave correctly in accordance with a *top-level*, abstract specification for these operations. This abstract specification is written directly in Isabelle's higher-order logic. It is short enough that a human can audit it to ensure that it accurately captures the intended behaviour of these operations. A complete description of our top-level specifications is available in separate work [3]. The top-level specifications for `sync()` and `iget()` are depicted in Figure 4. The total line count of both specifications including all their dependencies is 239 lines of Isabelle/HOL.

`sync()` and `iget()` each implement the corresponding functions expected of the Linux VFS layer. The `sync()` operation synchronises the current in-memory state of the file system to physical storage. The in-memory state may differ from the physical state because, as mentioned in Section 3.2, BilbyFs buffers pending writes in memory to improve metadata packing and throughput. The top-level *abstract file system* (AFS) specification for `sync()`, afs_sync, operates over the abstract file system state *afs*, which tracks the state of the physical storage medium (med), the pending in-memory medium updates (updates), and whether the file-system is currently read-only (is_readonly). The specification says that `sync()` first checks whether the file system is read-only, in which case an appropriate Error code is returned with the file system state unchanged (lines 2 and 3). Otherwise, it applies the in-memory updates to the physical medium.

Its specification is sufficiently nondeterministic to capture the behaviour of a correct file system under the situation when the in-memory updates are only partially applied, perhaps because of a flash device failure part-way through. For this reason, the specification allows any number of updates n (line 5) to succeed, between 0 and the total number of updates currently in-memory (i.e. length (updates *afs*)). It then (lines 8 and 9) applies the first n updates *toapply* to the physical medium med *afs* of file system state *afs*, and remembers the updates that remain to be applied *rem*. If all updates were applied, it returns Successfully, yielding the new file system state (lines 10 and 11). Otherwise (lines 12 to 14), it returns an appropriate error code, selected nondeterministically because the specification abstracts away from the precise reason *why* the failure might have occurred. In case of an I/O error (eIO), the file system is also put into read-only mode.

The `iget()` operation looks-up inodes on the physical medium. It takes a inode number *inum* and a VFS inode structure *vnode*. It first (line 2) checks whether an inode with the given inode number exists in the file system. To do so it must consult both the in-memory and on-medium state, computing via the expression updated_afs *afs* what the file system state *would be* if it were synchronised to the medium. If the inode number is not present (lines 11 and 12) an appropriate error code is returned. Otherwise, the `iget()` specification reads the inode with that number from the medium (line 4) and then returns appropriately based on whether the inode read succeeded (lines 6 and 7) or produced an error (lines 8 an 9). In the case of Success, the inode must be converted to a VFS inode structure for interoperating with the Linux VFS. Observe that the `iget()` specification does not return an updated *afs* structure: thus its type signature automatically captures that it can never modify the abstract file system state.

Note that the `iget()` specification intentionally elides interaction with the Linux inode cache. These are managed by a trivial amount of C code that sits between the Linux VFS layer and the BilbyFs implementation as produced by the COGENT compiler. Incorporating them into the COGENT implementation of BilbyFs (and thus into e.g., the `iget()` specification) would add little value, as these caches would be just large unverified ADTs and opaque to COGENT and the verification.

4.2 Functional Correctness Proof

We prove the correctness of the BilbyFs `sync()` and `iget()` operations, against their top-level specifications of Figure 4. The proof follows the modular decomposition of the BilbyFs implementation from Figure 3 (see Section 3.2). For both

```
afs_sync afs ≡
 if is_readonly afs then
  return (afs, Error eRoFs)
 else do
  n ← select{0..length (updates afs)};
  let updates = updates afs;
    (toapply, rem) = (take n updates, drop n updates);
    afs = (afs(|med := apply_updates toapply (med afs),
                updates := rem|));
  in if rem = [] then
   return (afs, Success ())
  else do
   e ← select {eIO, eNoMem, eNoSpc, eOverflow};
   return (afs(|is_readonly := (e = eIO)|), Error e)
  od
 od
```

```
afs_iget afs inum vnode ≡
 (if inum ∈ dom (updated_afs afs) then
  do
   r ← read_afs_inode afs inum;
   case r of
    Success inode ⇒
     return (inode2vnode inode, Success ())
   | Error e ⇒
     return (vnode, Error e)
  od
 else
  return (vnode, Error eNoEnt))
```

Figure 4. Top-level specifications for `sync()` and `iget()`, against which their BilbyFs COGENT implementations are verified.

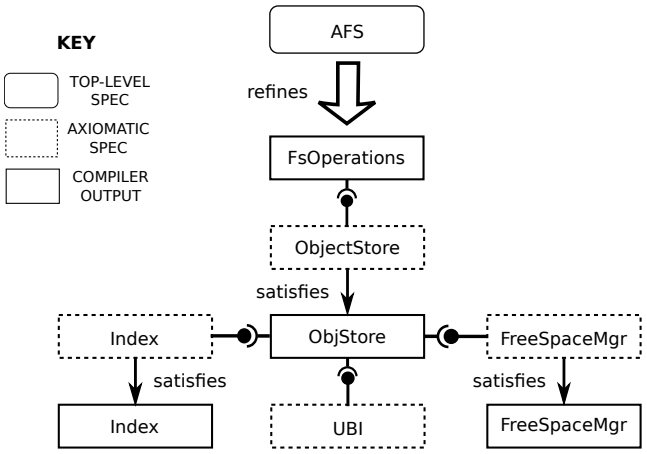

Figure 5. Modular functional correctness proof of BilbyFs.

operations, we prove the full component chain up to the UBI component.

Figure 5 depicts the structure of this proof. At the top, we prove that the code for the VFS-facing component of BilbyFs, called FsOperations, correctly *refines* its high-level specification as given in the abstract file system specification (AFS) of Figure 4. To complete this proof we formally define how to map the abstract file system state (*afs* of Figure 4) to the concrete file system state as implemented in COGENT. For instance, the abstract in-memory list of updates (updates *afs* of Figure 4) corresponds to an in-memory buffer where pending medium-writes are stored. The mapping from the former to the latter requires parsing the physical buffer contents (a list of bytes). In addition, the physical medium state (med *afs*) corresponds to the state of the UBI component of BilbyFs, which provides the interface to the flash storage. Similar to the in-memory buffer, mapping the state of the UBI component to its abstract representation in the AFS requires logically mimicking the file system `mount` operation, to parse the entire physical medium. In this sense, our proofs deal directly with the raw bytes stored in-memory and on-flash by the file system.

The FsOperations proof relies on formal assumptions about the ObjectStore. Those assumptions form an *axiomatic* specification of the ObjectStore, which serves as a compact representation of its correctness that abstracts away from its implementation details.

For `sync()`, the proof proceeds by establishing that the ObjectStore implementation indeed satisfies these axioms. This proof in turn makes use of axiomatic specifications for the components on which the ObjectStore relies, the Index and FreeSpaceManager, whose COGENT implementations we also prove satisfy their axioms. The proof bottoms out at the UBI component which, being entirely abstract, is captured by an axiomatic specification only. The validity of the entire functional correctness proof then rests on the validity of the assumptions encoded in the UBI specification. We return to these assumptions later in Section 4.4.

4.3 Ease of Proof

While they establish a similar property, these proofs are far simpler than e.g. the comparable functional correctness proofs of seL4 [26]. Just as with seL4, the functional correctness proof here relies on establishing global invariants of the abstract specification and its implementation. For BilbyFs, the invariants include e.g. the absence of link cycles, dangling links and the correctness of link counts, as well as the consistency of information that is duplicated in the file system for efficiency.

Importantly, unlike with seL4, none of the invariants have to include that in-memory objects do not overlap, or that object-pointers are correctly aligned and do point to valid objects. All of these details are handled automatically by

COGENT's type system and are justified by the C code correctness proofs generated by the COGENT compiler.

Even better, when proving that the file system correctly maintains its invariants, we get to reason over pure, functional specifications of the COGENT code. Because they are pure functions, these specifications do not deal with mutable state (as e.g., the seL4 ones do). Thus when proving that they satisfy their invariants, or refine the top-level AFS, we need not resort to cumbersome machinery like separation logic [40]: each function can be reasoned about by simply unfolding its definition, and the presence of separation between objects x and y follows trivially from x and y being separate variables.

Functional programmers have long recognised, and advocated for, the benefits afforded by reasoning over pure functions. COGENT puts these benefits directly into the hands of verification engineers without the need for a large, untrusted run-time system.

4.4 Assumptions and Invariants

As mentioned above, the BilbyFs proofs rest on the assumptions made about the Linux UBI component, as encoded in its axiomatic specification. At present, these assumptions are a little more restrictive than could be expected of a flash memory in practice. In particular, the axiom for the ubi_write function, which takes a buffer and writes it to the flash, states that either the entire write succeeds, or it fails leaving the flash unchanged. In practice, this write may be spread across multiple flash pages, each of which may succeed or fail and for which failure may leave the flash page only partially-written, or even corrupted [48]. With additional work, these assumptions can be made fully realistic.

The proofs assume that the file system invariant holds before invoking `sync()` and `iget()`, and we prove that these operations maintain it. The invariant talks about the contents of erase-blocks and wbuf, the in-memory buffer that stores pending updates. It asserts that the contents of erase-blocks and wbuf must form a valid log, i.e., data can be parsed as a sequence of valid transactions. After parsing, the logical representation of a transaction is a list of file system objects such as inode, directory entry and data blocks. The invariant also says that each transaction has a unique transaction number that indicates the order in which transactions must be applied when mounting the file system.

5. Evaluation

We evaluate COGENT from two perspectives: its ease-of-use as a systems language and verification target, and how well file systems implemented in COGENT perform.

5.1 Experience with COGENT

5.1.1 COGENT as a systems implementation language

In order to shed light on COGENT's usability as a systems programming language, we briefly describe the experience of developing our ext2 and BilbyFs implementations. In both cases, we started from a C implementation. In the case of ext2, this was Linux's ext2fs implementation; for BilbyFs it was our own implementation of the file system that was used to prototype its modular design. The two file systems were written by separate developers, but in the case of BilbyFs the same developer wrote both its C and COGENT implementations. Both developers were already familiar with functional programming, the ext2 developer is an undergraduate student.

Naturally, the COGENT language evolved in the process – at the time of the initial implementations, the language had linear types but no polymorphism and limited support for loops. The developers jointly wrote the shared ADT library, and the ext2 developer spent considerable time assisting with COGENT toolchain design and development. Unfortunately, this makes it infeasible to give accurate effort estimates for how long each file system would have taken to write had the language and toolchain been stable, as they are now.

Having to adopt COGENT's functional style was not a major barrier for either developer; indeed one reported that COGENT's use of nested `let`-expressions for sequencing and error handling aided his understanding of the potential control paths of his code. While both had to get used to the linear type system, both reported that this happened quite quickly and that the linear type system generally did not impose much of a burden when writing ordinary COGENT code. Both developers noted the usefulness of COGENT's linear types for tracking memory allocation and catching memory leaks.

Where linear types did cause friction was mostly when having to design the shared ADT interfaces to respect the constraints of the type system, as mentioned earlier in Section 3.3. Another place where linear types caused friction was when implementing the `rename()` operation in COGENT. `rename()` takes two directory arguments, the source and target directory of the file to be renamed. In case of renaming a file without changing its directory, these two arguments are identical. As COGENT does not allow such aliasing, we need two versions of this operation, leading to about 150 lines of essentially replicated code.

Both developers reported that the strong type system provided by COGENT decreased the time they usually would have spent debugging, which is to be expected. Logic bugs, which cannot be captured by the static semantics, can remain in COGENT code and are harder to debug because of lack of tool support. The developers, however, found comparatively few bugs in the COGENT code; the vast majority of bugs were in the C wrapper code or, to a lesser degree, in the C ADT implementations. Both developers reported that they felt that the COGENT support for a template-style C extension increased productivity in writing ADT implementations. Another reported benefit of COGENT's pluggable

ADTs was the ability to switch-in different ADT implementations to aid debugging.

System	native C	COGENT	generated C
ext2	4,077	2,789	12,066
BilbyFs	4,021	4,643	18,182

Table 1. Implementation source lines of code, native vs. COGENT, measured with sloccount. Generated line counts include ADTs.

Table 1 shows the source code sizes of the two systems. For the native ext2 system (i.e. the Linux code) we exclude code that implements features our COGENT implementation does not currently support. We can see that for the ext2 system, the COGENT implementation is about 2/3 the size of C. BilbyFs' COGENT implementation is larger than ext2's relative to their respective native C implementations. This is because BilbyFs makes much heavier use of the various ADTs from the common ADT library, some of which present fairly verbose client interfaces in their current implementation.

The current COGENT language can be viewed as a core language: we intentionally kept it minimal, without much syntactic sugar or support for common usage patterns. Rather than speculating about the optimal set of language shortcuts, we implemented the two case studies in the core language. This was obviously a source of frustration for both developers, but it gave us a better understanding of which additions are actually worthwhile. For example, code sequences as depicted in Figure 1, where the error handling code simply frees all the memory allocated in the sequence so far before returning an error value, are common. The programmer should not have to write such boilerplate code, especially because the type system provides sufficient information for the compiler to identify the objects that have to be destroyed before returning. Language support for this and other common patterns can be added fairly easily: as the compiler will just desugar these patterns into core language constructs, they do not affect the verification or code generation at all, and are orthogonal to the issues discussed here.

The blowout in size of the generated C code in Table 1 is mostly a result of normalisation steps applied by the COGENT compiler. Most of this is easily optimised away by the C compiler. However, we found that gcc's optimiser does an unsatisfactory job of optimising operations on large structs, resulting in unnecessary copy operations left in the code. This could be addressed by producing more optimised COGENT output for such cases.

It is also quite difficult to analyse performance of COGENT code, as links between assembly output and COGENT source code tend to be convoluted.

5.1.2 COGENT as a verification target

COGENT was carefully designed to provide a usable verification target for higher-level reasoning on top of its functional semantics. Specifically, we took care to ensure its semantics could be directly encoded as a pure, side-effect-free function in Isabelle's higher-order logic.

Recall from Section 2.3 that the COGENT compiler proves refinement, which means that reasoning about the generated specification is sound with respect to the generated C code, in that any property proved of the former is guaranteed to hold for the latter.

The linear type system is critical here, as we can only give COGENT a purely functional semantics because the linear types limit aliasing. Pure functions are naturally easy to reason about, especially by exploiting powerful yet sound automated tools like Isabelle's rewriting engine.

Compared to the seL4 project [26], the proofs generated by the COGENT compiler roughly correspond to the second refinement step from the intermediate executable specification to C code (except that the COGENT code is higher-level). Klein at al. [26] reports that one third of the total seL4 verification effort went into that step (not counting re-usable libraries and frameworks), so we can confidently predict that our co-generation automates at least a third of the overall effort of producing a completely verified file system. The savings are likely to be higher, as COGENT already proves a number of properties like correct pointer typing that in seL4 needed complex invariants established in the first refinement step, from abstract to executable specification [25].

The verification found six defects in the already tested and benchmarked BilbyFs implementation. Three of these occurred in serialisation functions, and three in the `sync()` implementation itself. Serialisation and de-serialisation are mechanical and tedious to write, which makes them prime candidates for further language and proof generation support.

The effort for verifying the complete file system component chain for the functions `sync()` and `iget()` in BilbyFs was roughly 9.25 person months, and produced roughly 13,000 lines of proof for the 1,350 lines of COGENT code. 4.5 person months, ≈4,000 lines of proof, of these were spent on serialisation/de-serialisation functions (≈850 lines of COGENT code), which, as mentioned, could be further automated. An additional ≈1,500 of the ≈13,000 lines of proof are libraries. The `sync()`-specific proof size is just about 5,700 lines and took 3.75 person months for ≈300 lines of COGENT code. The `iget()` proofs took 1 person month for ≈1,800 lines of proof and ≈200 lines of COGENT code.

This compares favourably with traditional C-level verification as for instance in seL4, which spent 12 person years with 200k lines of proof for 8,700 source lines of C code. Roughly 1.65 person months per 100 C source lines in seL4 are reduced to ≈0.69 person months per 100 COGENT source lines in COGENT.

5.2 COGENT performance

The evaluation platform for the ext2 file system is a four-core i7-6700 running at 3.1 GHz, with a Samsung HD501JL 7200RPM 500G SATA disk, running Linux kernel 4.3.0-

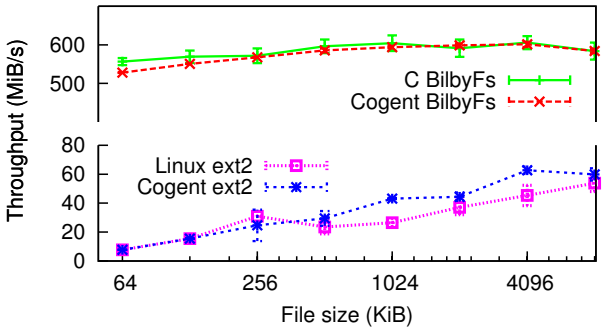

Figure 6. IOZone throughput for random 4 KiB writes.

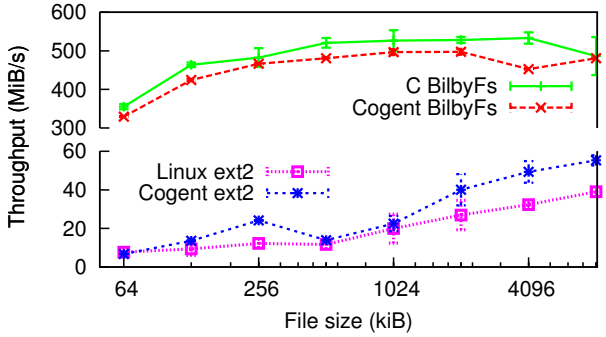

Figure 7. IOZone throughput for sequential 4 KiB writes.

1 from the Debian 8.0 distribution. To reduce run-to-run variability, we disable all but one core. For the BilbyFs evaluations we use a Mirabox with 1 GiB of NAND flash, a Marvell Armada 370 single-core 1.2 GHz ARMv7 processor and 1 GiB of DDR3 memory, running Linux kernel 3.17 from the Debian 6a distribution.

5.2.1 I/O microbenchmarks

We use the IOZone file system microbenchmarks [20] to evaluate basic performance. We use the default settings for an automated run, but include the cost of 'flush' at the end of each write for ext2.

Figure 6 shows performance on random writes. For BilbyFs, we do not include the cost of 'flush' at the end of each write since it completely hides the overhead of the COGENT implementation. BilbyFs shows a 5% throughput degradation in the worst case (64KiB files) for the COGENT version. The CPU load is around 20% compared to 15% on the C version. This is mostly the effect of redundant memory copy operations in the generated C code when passing structs on the stack.

On the other hand COGENT ext2 shows a modest improvement in throughput. CPU usage for COGENT and native Linux are the same at around 10%. We used `blktrace` to investigate; it appears that the COGENT implementation is slightly slower, which means that disk I/O operations hit the disk more often, instead of being merged in the I/O queue. We speculate that the on-disk firmware does a better job of scheduling disk writes than the Linux CFQ I/O scheduler.

Figure 7 shows sequential write performance. Again, COGENT outperforms Linux native for ext2, with very similar CPU load (around 10%). Indirect blocks have to be allocated at 512 KiB and a double-indirect block at 1024 KiB, causing the dips at these points. Again, tracing the I/O patterns show more, and more frequent, disk I/Os for the COGENT implementation. BilbyFs shows a throughput degradation of about 10% with CPU usage of 20% compared to the 15% of the C version. This degradation is explained by the same reasons presented for Figure 6.

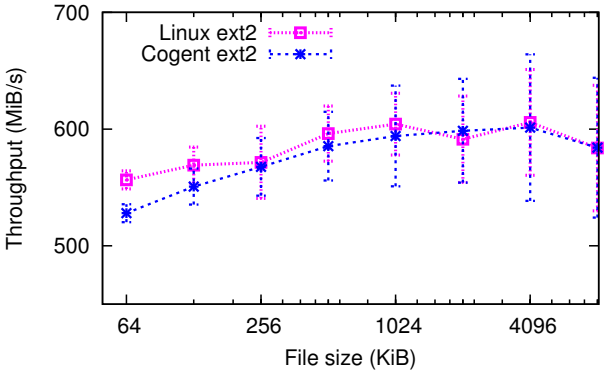

Figure 8. Random write performance on RAM disk.

In order to identify overheads resulting from the use of COGENT, without disk artifacts perturbing the results, we re-run the ext2fs benchmarks on a RAM disk, created as follows:

```
1 modprobe rd rd_size=1048576
2 dd if=/dev/zero of=/dev/ram0 bs=1M
3 mkfs -t ext2 -O none -r 0 -I 128 -b 1024 \
4   /dev/ram0
```

Figure 8 shows that without physical I/O, COGENT is slightly slower than native Linux, as expected. This confirms that the performance differences observed in the IO benchmarks are indeed the result of disk artifacts. The ($\pm 8\%$) error bars (showing standard deviation on ten runs) are because of CPU contention with other system activity; they are larger for COGENT because its slightly longer running time gives more opportunity for such contention.

5.2.2 File-system macro benchmarks

For a macro benchmark that exposes present limitations of COGENT, we run *postmark* [22], a benchmark that emulates a busy mail server by creating and deleting many small files. For ext2 tests, we configure the benchmark to start with 50,000 files of size 10,000 bytes each, and run on a RAM disk, so that I/O latency will not mask COGENT overheads. BilbyFs results are on a RAM disk that emulates the MTD

System	Total time sec	creation files/sec	read rate kB/sec
C ext2	10	5025	248
COGENT ext2	21	2393	118
C BilbyFs	6	33375	431
COGENT BilbyFs	10	20025	259

Table 2. Postmark run summary, CPU usage is 100% in all cases.

interface, this time on the same i7 machine that we use for the ext2 tests. BilbyFs' file creation times are much faster than ext2's, so we increased the initial number of files to 200,000.

Table 2 shows the results, each of the values is the mode of ten runs. Typically five or six of the ten runs have essentially the same performance, the rest is worse as benchmarking overlaps with unavoidable system activity. We can see significant degradation of the COGENT implementations, a factor around two for ext2 on the RAM disk, and 1.5 times for BilbyFs.

Profiling COGENT ext2 performance shows that most of the time is spent in converting from in-buffer directory entries to COGENT's internal data type. A better implementation of the directory handling code could reduce this problem.

BilbyFs' bottleneck is in a function that summarises information about newly created files for the log. The same function shows as a bottleneck in both C and COGENT versions, but in the COGENT version it takes about three times as long.

The underlying reason for these results is that the COGENT compiler is at present overly reliant on the C compiler's optimiser. In particular, it passes many structs on the stack, which result in much extra copying, which the C compiler fails to optimise away. Further work is needed to generate code that is more in line with the C compiler's optimisation capabilities.

6. Related Work

Functional verification of file systems belongs to systems verification in general. Klein [24] gives an overview of the early work in this area. The more recent achievements are the comprehensive verification of the seL4 microkernel [25, 26], the verification stack of the Verisoft project [1, 2], the increase of verification productivity in CertiKOS [12, 13], and the full end-to-end application verification in Ironclad [15], which builds on a modified verified Verve kernel [51].

We share the vision for end-to-end systems verification with Ironclad, with verified file systems as a major component, but we are not ready to sacrifice performance as they do. Although our case studies interface with Linux for meaningful evaluation, COGENT is designed to integrate with the verification stack of the seL4 kernel, which will allow a verified COGENT file system to run in protected isolation from legacy applications for a truly minimal TCB.

Verified File Systems Early Z specifications of file systems are those by Morgan and Sufrin [33] for UNIX, and Bevier et al [7] for a custom file system. Arkoudas et al [5] verify two key operations on the block-level of a file system, but the result remains partial and the authors even argue that system components such as file systems will probably always remain beyond the reach of full correctness proofs.

Joshi and Holzman [21] proposed a file system verification challenge, which our work fits into. The aim of this paper was not to provide a full file system verification, but rather to show that high-level file system properties are possible to prove down to the low-level implementation at the same proof productivity as reasoning about higher-level specifications. There is plenty of previous work that shows proofs about such high-level specifications. Our work closes the gap to code. For instance, Hesselink and Lali [18] manage to prove a file refinement stack that goes down to a formal model, but assumes an infinite storage device and other simplifications, and does not end in code. The Event B refinement proof by Damchoom et al [9] similarly does not end in code. In theory, Event B can generate code from low-level models, but neither of these verifications are close enough to achieve usable file system implementations, let alone high performance.

The most realistic high-level Flash file system verification work to date is conducted using the KIV tool [39] and goes from the Flash device layer up to a Linux VFS implementation [11, 44, 45]. The verification work is still in progress and the current code generation from low-level models targets Scala running on a Java Virtual Machine, which implies run-time overheads and dependency on a large language runtime. It may be fruitful to investigate a COGENT backend for this work.

Maric and Sprenger [31] investigate the issue of crash tolerance in file systems, and previously Andronick [4] formally analysed similar issues for tearing in smart cards with persistent storage. COGENT does not provide any special handling for crash tolerance, but the generated specifications are detailed enough to facilitate reasoning about it. Chen et al [8] take cash tolerance to the level of a complete file system in a proof that includes functional correctness of an implementation in Coq. This Coq implementation is roughly on the same level as the Isabelle/HOL specifications generated out of COGENT. Their work is a strong argument that COGENT is hitting the right level of abstraction.

Another stream of work in the literature focuses on more automatic techniques such as model checking and static analysis [14, 41, 52]. While in theory these techniques could be used to provide similar guarantees as COGENT, this has not yet been achieved in practice. Instead of providing guarantees, such analyses are more useful as tools for efficiently

finding defects in existing implementations. They also do not provide a path to further higher-level reasoning.

7. Conclusions

We have presented an approach to prove the functional correctness of low-level file system code. We build on COGENT, a certifying compiler that produces C code, Isabelle/HOL specifications, and translation correctness proofs, and we prove properties about the generated specifications in Isabelle/HOL. Composed with the compiler correctness proof, we obtain proofs about the C code.

We find that COGENT not only allows non-experts in formal verification to write provably-safe, fully realistic file system code, it is also the key step to lowering the effort and complexity for the full mechanical verification of file systems against high-level formal specifications.

Our evaluation shows that on a real disk our present file system implementations in COGENT have almost identical throughput with their C counterparts, with slightly more CPU usage. Furthermore, we find that this degradation can, in many cases, be avoided by improving the COGENT compiler output, such that it is less dependent on the optimiser of the underlying C compiler.

COGENT is a sequential language. Although it allows asynchronous disk access, obvious future work is full concurrency. While not a trivial transition, we believe the linear type system of COGENT will help: it lets the compiler keep track of memory locations and side effects. Moreover, the high-level functional semantics can already be executed fully concurrently — sequentialisation happens through the explicit states that the programmer can see and pass through the program in COGENT, making interaction points explicit. Given additional suitable synchronisation primitives, if those states become more fine-grained, more concurrency becomes available.

Acknowledgements

We would like to thank Andrew Baumann and Leonid Ryzhyk for insightful feedback on earlier drafts of this paper.

NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

This material is based on research sponsored by Air Force Research Laboratory and the Defense Advanced Research Projects Agency (DARPA) under agreement number FA8750-12-9-0179. The U.S. Government is authorised to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory, the Defense Advanced Research Projects Agency or the U.S. Government.

References

[1] ALKASSAR, E., HILLEBRAND, M., LEINENBACH, D., SCHIRMER, N., STAROSTIN, A., AND TSYBAN, A. Balancing the load — leveraging a semantics stack for systems verification. *Journal of Automated Reasoning: Special Issue on Operating System Verification 42, Numbers 2–4* (2009), 389–454.

[2] ALKASSAR, E., PAUL, W., STAROSTIN, A., AND TSYBAN, A. Pervasive verification of an OS microkernel: Inline assembly, memory consumption, concurrent devices. In *Proceedings of Verified Software: Theories, Tools and Experiments 2010* (Edinburgh, UK, Aug. 2010), vol. 6217 of *Lecture Notes in Computer Science*, pp. 71–85.

[3] AMANI, S., AND MURRAY, T. Specifying a realistic file system. In *Workshop on Models for Formal Analysis of Real Systems* (Suva, Fiji, Nov. 2015), pp. 1–9.

[4] ANDRONICK, J. Formally Proved Anti-tearing Properties of Embedded C Code. In *Proceedings of Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation* (Paphos, Cyprus, Nov. 2006), pp. 129–136. Invited Speaker.

[5] ARKOUDAS, K., ZEE, K., KUNCAK, V., AND RINARD, M. C. Verifying a file system implementation. In *Proceedings of the 6th International Conference on Formal Engineering Methods* (Seattle, WA, USA, Nov. 2004), vol. 3308 of *Lecture Notes in Computer Science*, pp. 373–390.

[6] BERTOT, Y., AND CASTÉRAN, P. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. 2004.

[7] BEVIER, W., COHEN, R., AND TURNER, J. A specification for the Synergy file system. Tech. Rep. Technical Report 120, Computational Logic Inc., Austin, Texas, USA, Sept. 1995.

[8] CHEN, H., ZIEGLER, D., CHAJED, T., CHLIPALA, A., KAASHOEK, M. F., AND ZELDOVICH, N. Using Crash Hoare logic for certifying the FSCQ file system. In *Proceedings of the 25rd ACM Symposium on Operating Systems Principles* (Monterey, CA, Oct. 2015), pp. 18–37.

[9] DAMCHOOM, K., BUTLER, M., AND ABRIAL, J.-R. Modelling and proof of a tree-structured file system in Event-B and Rodin. In *Proceedings of the 10th International Conference on Formal Engineering Methods* (Kitakyushu-City, Japan, Oct. 2008), vol. 5256 of *Lecture Notes in Computer Science*, pp. 25–44.

[10] ELPHINSTONE, K., AND HEISER, G. From L3 to seL4 – what have we learnt in 20 years of L4 microkernels? In *ACM Symposium on Operating Systems Principles* (Farmington, PA, USA, Nov. 2013), pp. 133–150.

[11] ERNST, G., SCHELLHORN, G., HANEBERG, D., PFÄHLER, J., AND REIF, W. Verification of a virtual filesystem switch. In *Proceedings of Verified Software: Theories, Tools and Experiments 2013* (Menlo Park, CA, USA, May 2013), vol. 8164 of *Lecture Notes in Computer Science*, pp. 242–261.

[12] GU, L., VAYNBERG, A., FORD, B., SHAO, Z., AND COSTANZO, D. CertiKOS: A certified kernel for secure cloud

computing. In *Proceedings of the 2nd Asia-Pacific Workshop on Systems (APSys)* (2011).

[13] GU, R., KOENIG, J., RAMANANANDRO, T., SHAO, Z., WU, X. N., WENG, S., ZHANG, H., AND GUO, Y. Deep specifications and certified abstraction layers. In *Proceedings of the 42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2015), pp. 595–608.

[14] GUNAWI, H. S., RUBIO-GONZÁLEZ, C., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEA, R. H., AND LIBLIT, B. EIO: error handling is occasionally correct. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies* (Feb. 2008), pp. 207–222.

[15] HAWBLITZEL, C., HOWELL, J., LORCH, J. R., NARAYAN, A., PARNO, B., ZHANG, D., AND ZILL, B. Ironclad apps: End-to-end security via automated full-system verification. In *USENIX Symposium on Operating Systems Design and Implementation* (Broomfield, CO, US, Oct. 2014), pp. 165–181.

[16] The Heartbleed bug. http://heartbleed.com, https://www.openssl.org/news/secadv_20140407.txt, Apr. 2014. Accessed March 2015.

[17] HERDER, J. N., BOS, H., GRAS, B., HOMBURG, P., AND TANENBAUM, A. S. MINIX 3: A highly reliable, self-repairing operating system. *ACM Operating Systems Review 40*, 3 (July 2006), 80–89.

[18] HESSELINK, W. H., AND LALI, M. I. Formalizing a hierarchical file system. In *Proceedings of the 14th BCS-FACS Refinement Workshop* (Eindhoven, The Netherlands, Nov. 2009), vol. 259 of *Electronic Notes in Theoretical Computer Science*, pp. 67–85.

[19] HUNTER, A. A brief introduction to the design of UBIFS, 03 2008.

[20] IOzone filesystem benchmark. http://www.iozone.org/.

[21] JOSHI, R., AND HOLZMANN, G. J. A mini challenge: build a verifiable filesystem. *Formal Aspects of Computing 19*, 2 (2007), 269–272.

[22] KATCHER, J. PostMark: A new file system benchmark. Tech. Rep. TR-3022, NetApp, October 1997.

[23] KELLER, G., MURRAY, T., AMANI, S., O'CONNOR-DAVIS, L., CHEN, Z., RYZHYK, L., KLEIN, G., AND HEISER, G. File systems deserve verification too! In *Workshop on Programming Languages and Operating Systems (PLOS)* (Farmington, Pennsylvania, USA, Nov. 2013), pp. 1–7.

[24] KLEIN, G. Operating system verification — an overview. *Sādhanā 34*, 1 (Feb. 2009), 27–69.

[25] KLEIN, G., ANDRONICK, J., ELPHINSTONE, K., MURRAY, T., SEWELL, T., KOLANSKI, R., AND HEISER, G. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems 32*, 1 (Feb. 2014), 2:1–2:70.

[26] KLEIN, G., ELPHINSTONE, K., HEISER, G., ANDRONICK, J., ET AL. seL4: Formal verification of an OS kernel. In *ACM Symposium on Operating Systems Principles* (Big Sky, MT, US, Oct. 2009), pp. 207–220.

[27] LEROY, X. Formal certification of a compiler back-end, or: Programming a compiler with a proof assistant. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Charleston, SC, USA, 2006), pp. 42–54.

[28] LEROY, X. Formal verification of a realistic compiler. *Communications of the ACM 52*, 7 (2009), 107–115.

[29] LIEDTKE, J. Improving IPC by kernel design. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles* (Asheville, NC, US, Dec. 1993), pp. 175–188.

[30] Memory leak in Android Lollipop's rendering engine. https://code.google.com/p/android/issues/detail?id=79729#c177, Dec. 2014. Accessed March 2015.

[31] MARIC, O., AND SPRENGER, C. Verification of a transactional memory manager under hardware failures and restarts. In *Proceedings of the 19th International Symposium on Formal Methods (FM)* (May 2014), vol. 8442 of *Lecture Notes in Computer Science*, pp. 449–464.

[32] MATICHUK, D., MURRAY, T., ANDRONICK, J., JEFFERY, R., KLEIN, G., AND STAPLES, M. Empirical study towards a leading indicator for cost of formal software verification. In *International Conference on Software Engineering* (Firenze, Italy, Feb. 2015), p. 11.

[33] MORGAN, C., AND SUFRIN, B. Specification of the UNIX filing system. *IEEE Transactions on Software Engineering 10*, 2 (1984), 128–142.

[34] NIPKOW, T., PAULSON, L., AND WENZEL, M. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, vol. 2283 of *Lecture Notes in Computer Science*. 2002.

[35] POSIX filesystem test project. http://sourceforge.net/p/ntfs-3g/pjd-fstest/ci/master/tree/. Retrieved Jan., 2016.

[36] O'CONNOR, L., RIZKALLAH, C., CHEN, Z., AMANI, S., LIM, J., NAGASHIMA, Y., SEWELL, T., HIXON, A., KELLER, G., MURRAY, T., AND KLEIN, G. COGENT: Certified compilation for a functional systems language. *CoRR abs/1601.05520* (2016).

[37] Apple's SSL/TLS bug. https://www.imperialviolet.org/2014/02/22/applebug.html, Feb. 2014. Accessed March 2015.

[38] PALIX, N., THOMAS, G., SAHA, S., CALVÈS, C., LAWALL, J., AND MULLER, G. Faults in Linux: ten years later. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems* (Newport Beach, CA, US, 2011), pp. 305–318.

[39] REIF, W., SCHELLHORN, G., STENZEL, K., AND BALSER, M. Structured specifications and interactive proofs with KIV. In *Automated Deduction – A Basis for Applications*, vol. 9 of *Applied Logic Series*. 1998, pp. 13–39.

[40] REYNOLDS, J. C. Separation logic: A logic for mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science* (Copenhagen, Denmark, July 2002).

[41] RUBIO-GONZÁLEZ, C., AND LIBLIT, B. Defective error/pointer interactions in the Linux kernel. In *Proceedings of the 20th International Symposium on Software Testing and Analysis* (Toronto, ON, Canada, July 2011), pp. 111–121.

[42] SAHA, S., LAWALL, J., AND MULLER, G. An approach to improving the structure of error-handling code in the Linux kernel. In *Conference on Language, Compiler and Tool Support for Embedded Systems (LCTES)* (Chicago, IL, USA, Apr. 2011), pp. 41–50.

[43] SAHA, S., LOZI, J.-P., THOMAS, G., LAWALL, J. L., AND MULLER, G. Hector: Detecting resource-release omission faults in error-handling code for systems software. In *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on* (2013), IEEE, pp. 1–12.

[44] SCHELLHORN, G., ERNST, G., PFÄHLER, J., HANEBERG, D., AND REIF, W. Development of a verified Flash file system. In *Proceedings of the 4th International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z* (Toulouse, France, June 2014), vol. 8477 of *Lecture Notes in Computer Science*, pp. 9–24.

[45] SCHIERL, A., SCHELLHORN, G., HANEBERG, D., AND REIF, W. Abstract specification of the UBIFS file system for flash memory. In *Proceedings of the Second World Congress on Formal Methods (FM)* (Nov. 2009), vol. 5850 of *Lecture Notes in Computer Science*, pp. 190–206.

[46] SEWELL, T., MYREEN, M., AND KLEIN, G. Translation validation for a verified OS kernel. In *ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA, June 2013), pp. 471–481.

[47] SINGARAVELU, L., PU, C., HÄRTIG, H., AND HELMUTH, C. Reducing TCB complexity for security-sensitive applications: Three case studies. In *Proceedings of the 1st EuroSys Conference* (Leuven, BE, Apr. 2006), pp. 161–174.

[48] TSENG, H.-W., GRUPP, L., AND SWANSON, S. Understanding the impact of power loss on flash memory. In *Proceedings of the 48th Design Automation Conference* (2011), ACM, pp. 35–40.

[49] WADLER, P. Linear types can change the world! In *Programming Concepts and Methods* (1990).

[50] WINWOOD, S., KLEIN, G., SEWELL, T., ANDRONICK, J., COCK, D., AND NORRISH, M. Mind the gap: A verification framework for low-level C. In *International Conference on Theorem Proving in Higher Order Logics* (Munich, Germany, Aug. 2009), pp. 500–515.

[51] YANG, J., AND HAWBLITZEL, C. 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* (Toronto, Ont, CA, June 2010), pp. 99–110.

[52] YANG, J., SAR, C., AND ENGLER, D. EXPLODE: a lightweight, general system for finding serious storage system errors. In *USENIX Symposium on Operating Systems Design and Implementation* (Nov. 2006), pp. 131–146.