# A Methodology for Trustworthy File Systems

**Sidney Amani**

Submitted in fulfilment of the requirements for the degree of
Doctor of Philosophy


UNSW
THE UNIVERSITY OF NEW SOUTH WALES
SYDNEY • AUSTRALIA

School of Computer Science and Engineering

Faculty of Engineering

August 2016

## Originality Statement

'I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, or substantial proportions of material which have been accepted for the award of any other degree or diploma at UNSW or any other educational institution, except where due acknowledgement is made in the thesis. Any contribution made to the research by others, with whom I have worked at UNSW or elsewhere, is explicitly acknowledged in the thesis. I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project's design and conception or in style, presentation and linguistic expression is acknowledged.'

Signed .....

Date .....

## Copyright Statement



Signed .....

Date .....

## Authenticity Statement



Signed .....

Date .....

**Abstract**

The main contribution of this thesis is a methodology for designing, implementing and verifying realistic file systems with evidence of its effectiveness by application to a non-trivial flash file system. The goal of this research is to demonstrate that an implementation level machine-checked proof of correctness of a file system can be done at reasonable cost.

Our approach leverages COGENT, a purely functional, memory- and type-safe language we helped design, that bridges the gap between verifiable formal model and low-level code. COGENT takes a modular file system implementation as input and generates a C implementation and a formal proof that links it to its corresponding generated COGENT specification. COGENT specifications inherit the purely functional aspect of the input source code, and thus they proved much easier to reason about than the C code directly.

In order to prove the correctness of complex file system components at reasonable cost, we decompose the functionality into a set of components such that the correctness of each can be proven in isolation. The component proofs are mechanically composed into a theorem that holds on the C implementation by refinement.

To validate our approach, we designed and implemented BilbyFs, a modular flash file system. We formally specified BilbyFs' file system operations in Isabelle/HOL, and proved the functional correctness of two key operations: sync() and iget().

BilbyFs' design demonstrates the practicality of aggressive modular decomposition, and its COGENT implementation highlights the benefits and feasibility of using a linearly-typed language to implement a realistic file system. Our verification results show that we can exploit our modular design to reason about implementation components in isolation, and that overall our methodology drastically reduces the effort of verifying file system code.

## Acknowledgements

This thesis would not have been possible without the help of Toby Murray, my main supervisor during my PhD program, whose supervision style was great. Toby, thank you for your patience over the years and for cheering me up when the future did not look bright.

I am also indebted to Leonid Ryzhyk, who was my supervisor during the first year of my PhD. Leonid, you have been a model of diligence and integrity to me. I truly admire you as a colleague and, more importantly, as a friend.

This adventure would not have been possible without the support of Gernot Heiser, Gabi Keller and Gerwin Klein who were always willing to help and never lost their trust in me.

The rest of the *Trustworthy File Systems* team also played a significant role in this endeavour. It was a great pleasure to work with so many talented people. More specifically, I would like to thank Zilin Chen, Christine Rizkallah, Alex Hixon, Yutaka Nagashima, Liam O'Connor, Thomas Sewell, Joel Beeren, Japheth Lim and Keng Chai Ong.

I am grateful that I had the support of my family: my parents, Bernardine and N'Goran Amani; my siblings, Steve, Johan, Gisele and their own families. I cannot enumerate all the many friends I am fortunate to have, but I would like to give a special thank to the members of *Boyz In Da Hood*. Thank you all for being supportive during that journey.

Finally I would like to thank Lucy, whose love and support have been crucial. Lucy, without your support, I would certainly have given up. You went, with me, through the roller-coaster ride that is pursuing a PhD and I cannot thank you enough for this.

# Contents

# List of Figures

# List of Publications

This thesis is partly based on work in the following publications:

- Liam O'Connor, Zilin Chen, Christine Rizkallah, Sidney Amani, Japheth Lim, Toby Murray, Yutaka Nagashima, Thomas Sewell, and Gerwin Klein. Refinement through restraint: Bringing down the cost of verification. In *International Conference on Functional Programming*, Nara, Japan, September 2016

- 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, and Gernot Heiser. Cogent: Verifying high-assurance file system implementations. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 175–188, Atlanta, GA, USA, April 2016. doi:10.1145/2872362.2872404

- Sidney Amani and Toby Murray. Specifying a realistic file system. In *Workshop on Models for Formal Analysis of Real Systems*, pages 1–9, Suva, Fiji, November 2015

- Gabriele Keller, Toby Murray, Sidney Amani, Liam O'Connor-Davis, Zilin Chen, Leonid Ryzhyk, Gerwin Klein, and Gernot Heiser. File systems deserve verification too! In *Workshop on Programming Languages and Operating Systems (PLOS)*, pages 1–7, Farmington, Pennsylvania, USA, November 2013. doi:10.1145/2525528.2525530

# 1 Introduction

## 1.1 The problem

File systems must be trustworthy because they provide critical system functionality. Almost all user-space applications rely on the assumption that the underlying file system conforms to POSIX standard interfaces and is bug-free. Chen et al. [Chen et al., 2011] showed that file systems are a significant source of security vulnerabilities in the Linux kernel. Lu et al. [2014]'s recent study of file systems maintenance patches shows that we regularly find bugs even in mature file systems like ext3. More importantly, their study shows that most file systems defects are semantic bugs, i.e. they require an understanding of file system semantics to be detected and fixed.

Projects like seL4 [Klein et al., 2009] and CompCert [Leroy, 2009] have shown that interactive formal verification can be used to assert strong correctness guarantees on realistic system software. For instance, the seL4 project achieved a machine-checked proof of functional correctness for a general-purpose microkernel. This guarantees that every behaviour exhibited by the system has a corresponding behaviour captured by the system's specification. The seL4 microkernel is 8,700 source lines of C code. However, seL4 was a 12 person years effort, which is prohibitively expensive for many application domains. Verification will only become a practical solution to build trustworthy systems if it becomes more affordable. The research presented in this thesis is a step toward that goal, focussing specifically on file systems.

File systems present an interesting verification challenge because they provide a simple interface to user applications, yet their code base is large and convoluted with asynchronous interfaces, a lot of low-level manipulations of on-disk data structures and complicated code paths for error handling.

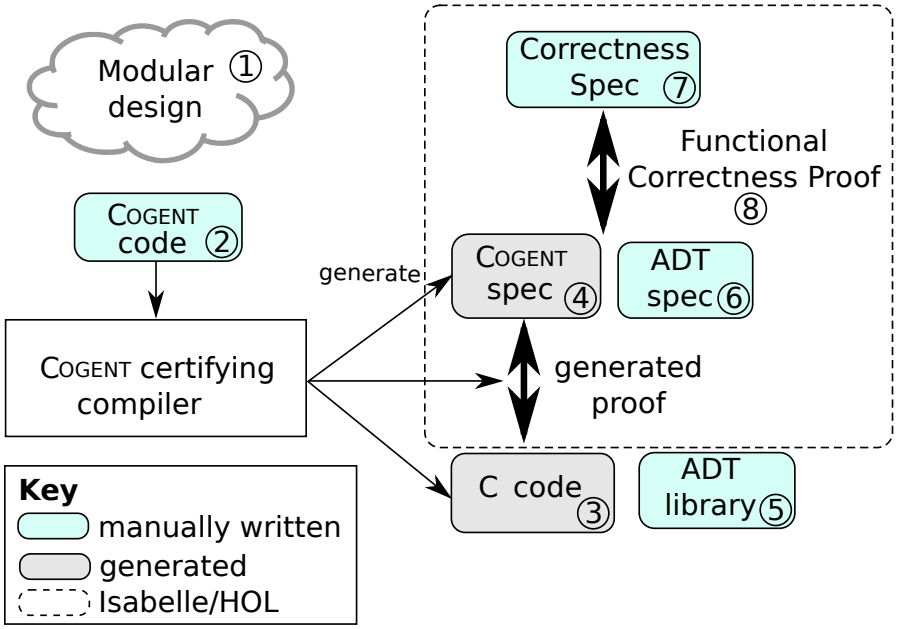


**Figure 1.1:** Methodology overview

## 1.2 Our approach

In this research we present our methodology to design, implement and verify realistic file systems. Our methodology to build trustworthy file systems relies on two key ideas.

First, we design and implement file systems modularly such that the implementation can be decomposed into a set of components that interact through well defined interfaces. The correctness of each component can be verified independently by relying solely on the interface specifications of the components it relies on. This modular decomposition enables incremental verification of file system functionality. By proving that each component adheres to its interface specification, the component proofs can be composed together mechanically to produce an overall proof of functional correctness for the entire file system implementation.

Second, we use a purely functional, linearly typed memory safe language, called COGENT, to co-generate firstly a modular design-level formal specification of the file system tailored to interactive verification, and secondly an efficient C implementation of the file system. COGENT' compiler also generates a proof that links the COGENT specification to the C implementation; however, that work was carried out by others and falls outside the scope of this thesis.

Figure 1.1 shows an overview of our methodology. To maximise verification pro-

ductivity, we design file systems following aggressive modular decomposition (1). This decomposition is recursive: each file system component may be further decomposed into a set of sub-components on which it relies, and each of whose correctness can be verified separately. The modular design is then implemented (2) in COGENT and uses the COGENT certifying compiler to generate the C implementation (3) and the design-level COGENT specifications of the file system implementation (4), and an automatic proof establishing that the former is a refinement of the latter.

COGENT was designed to allow verification engineers to reason about the code at the same level of abstraction as developers. For instance, C developers rarely think about pointer aliasing. Apart from a few corner cases, developers implicitly assume that a pointer is the unique reference to an object. Thus developers do not need to worry about the side-effects that memory aliasing can cause. COGENT increases the level of abstraction for reasoning about low-level code by using a linear type system that restricts pointer aliasing and facilitates local reasoning without resorting to cumbersome machinery like separation logic [Reynolds, 2002].

Code that requires pointer aliasing, such as the implementation of data structures like doubly linked-lists or red-black trees, cannot be implemented in COGENT and must instead be implemented directly in C (5). COGENT provides a foreign function interface allowing such data structures to be used from within COGENT code. One must manually specify (6) and verify their implementations in order to ensure soundness, using standard C code verification techniques [Leroy, 2009; Tuch et al., 2007; Greenaway et al., 2014; Leinenbach, 2008].

The foreign function interface is mainly used to access a library of manually verified abstract data structures (ADT) written in C, such as lists, arrays and trees. This library provides operations to search, alter, and iterate over these structures, because COGENT does not support arbitrary loops or recursion directly in the language.

To prove the absence of semantic bugs, we specify and verify the functional correctness of the file system. The specification (7) is a high-level formal description of the correct functionality of each file system operation, written manually, using the Isabelle/HOL proof assistant. We verify the functional correctness of file system operations (8) against their high-level functional correctness specification. We leverage the file system's rigorous modular decomposition to incrementally prove the correctness of file system operations. The reasoning is performed over the COGENT specifications, and applies to the file system's C code implementation by virtue of the compiler-generated proofs that link the two. All proofs are machine-checked with the Isabelle/HOL [Nipkow et al., 2002] proof assistant.

## 1.3 Research contributions

The main contribution of this thesis is a methodology for designing, implementing and verifying realistic file systems. We show that, using our implementation and verification framework, realistic file systems can be implemented and reasoned about modularly at reasonable cost. We use BilbyFs, a flash file system we designed, as a long running example to demonstrate that, when used carefully, design modularity does not significantly degrade systems performance. We present a set of design principles for building verifiable highly modular file systems (Chapter 3), with reference to BilbyFs. Since COGENT was in development while we were designing BilbyFs, we wrote a prototype implementation in C that we used to evaluate the performance of BilbyFs' initial modular design by comparing it to existing Linux flash file systems (Chapter 3).

We demonstrate that a linearly typed language can be made sufficiently expressive to implement efficient file systems. We present a set of design patterns for implementing a file system in a linearly typed language like COGENT in a way that enables modular reasoning about the implementation using a proof assistant. We evaluate and compare the performance of BilbyFs implemented in COGENT to its prototype implementation in C (Chapter 4).

We present a technique for compactly specifying the high-level functional correctness of file systems. We demonstrate this technique with reference to the formal correctness specification of BilbyFs' functionality. This specification is, to our knowledge, the first one to capture the semantics of asynchronous writes of a POSIX file system, a critical performance optimisation (Chapter 5).

We present a technique for modularly verifying file systems written in COGENT, by following their modular decomposition (Chapter 6). We exercise this technique by formally verifying that the COGENT implementation of the operations iget() and sync() of BilbyFs refine their high-level correctness specification. We demonstrate that purely functional COGENT specifications allow leveraging the implementation modularity to increase verification productivity and greatly simplify reasoning about the file system logic. Therefore, we show that our approach is practical. Finally, we use our current verification results to estimate the effort required to complete the full verification of BilbyFs.

All the work described in this thesis has been open sourced and is available as part of the COGENT project on github [1].

[1] https://github.com/NICTA/cogent

# 2 Related Work

In Chapter 1 we presented our overall methodology for designing, implementing and verifying file systems. Our methodology relies upon COGENT, a purely functional language that we co-designed. COGENT is linearly typed to assist the programmer with memory allocation and to enable the COGENT's certifying compiler to generate efficient C code with in-place updates from purely functional input code. At the core of our methodology, we design the file system modularly such that the COGENT implementation can be decomposed into a set of components that can each be reasoned about in isolation in the COGENT-generated specifications.

Our work thus lies at the intersection of file system verification, language-based verification and modular verification. We address these areas separately in the next sections.

## 2.1 File system verification

File system verification has been a challenge of interest for the formal methods community for more than a decade [Joshi and Holzmann, 2007]. The first attempt to formalise a file system interface was published in 1987 by Morgan and Sufrin [1987]. They presented an abstract specification of a UNIX file system. The specification describes file system operations at the level of system calls. At this time it was common for file systems to be responsible for keeping track of user-level state such as open files, seek-pointers, etc. Contemporary operating systems support multiple file systems running simultaneously and abstract away this bookkeeping burden within the Virtual File system Switch layer (VFS). Bevier et al. [1995] built upon Morgan and Suffrin's model to specify the Synergy file system, a custom file system that supports access control on files. BilbyFs' correctness specification shares similarities with Morgan and Suffrin's specification, but we specify

the functionality of file system operations as expected by the VFS layer.

Freitas et al. [2007] formalised the correctness of the IBM CICS [IBM] file control interface using the Z/Eves [Saaltink, 1997] prover. The CICS file control interface is a non-POSIX interface with no directory operations and where files are manipulated at record (i.e. small portion of a file) granularity. The specification presented in Freitas et al. supports the "syncpoint" operation that ensures that the result of previous updates on a file are made persistent. Freitas et al. proved that the specification is free of undefined expressions and they specified the pre-conditions under which the semantics of each operation is well-defined. By contrast, we prove the functional correctness — a much stronger property — of two operations of a POSIX compliant file system, including the operation `sync()` that makes in-memory pending updates persistent. As opposed to the one of CICS, BilbyFs' specification captures asynchonous meta-data updates, such as file and directory creation.

More recently, Freitas et al. [2009] refined the specification of Morgan and Sufrin [1987] to a file system implementation based on the Java HashMap object. They used the Z/Eves theorem prover combined with Java Modelling Language (JML) annotations to prove that the HashMap implementation is a refinement of the abstract specification. Z/Eves is a theorem prover supporting the Z notation [Spivey, 1992]: a specification language based on set theory, first-order logic and lambda calculus. JML is a tool-set that gives extra formalisation support for specifying pre/post conditions and loop invariants for Java code. The specification represents a file as an object containing attributes (e.g size) and a sequence of bytes. The main contribution of their work is a proof of forward simulation showing that the HashMap implementation refines the abstract file store, but ignores several aspects of a realistic file system implementation that we aim to cover, such as directories, error handling and block representation of files.

In the same line of research Damchoom et al. [2008]; Damchoom and Butler [2009]; Damchoom [2010] used the specification by Hughes [1989] to formalise a hierarchical file system in Event-B. Event-B is an evolution of B with a simpler notation and a tool chain more suitable for low-level specifications, that can be related to an actual implementation. Their most abstract specification supports the operations create, delete, move and copy to manipulate files and directories. Each of these operations is shown to preserve the properties of the file system tree, such as every node in the tree has a parent. The specification is refined multiple times to gradually introduce the notions of file/directory, file content, permissions and file attributes such as, creation and modification times. However, many implementation details remain unspecified such as: error handling, block allocation, concrete storage medium layout description, etc. Although ultimately the

Event-B model is meant to be used for automatically generating a Java implementation of a flash file system, in their research the Event-B model was used as a guide to write a Java implementation manually. By contrast, our methodology removes humans intervention from the code that runs and the model that is verified. Specifically, the COGENT certifying compiler automatically generates a low-level C implementation that can run in the Linux kernel, and the proofs on BilbyFs' COGENT-generated specifications hold on the C code by refinement.

Hesselink and Lali [2009] formalised a hierarchical file system using the PVS theorem prover [Owre et al., 1992]. They proposed a refinement proof from an abstract model of the VFS to an inode-based file system specification. The VFS model is a partial mapping from *path* to *data*, where a path is a sequence of names and data is an option type storing the file content or nothing when the path refers to a directory. The VFS model comes with with an invariant asserting that all the prefixes of the list of names forming a path in the mapping must also be part of the mapping. The inode-based specification is another partial mapping from *inode number* to *inode*. An inode number uniquely identifies an inode, which is represented as a record with a field *content* to store file contents and a field *dir* for directory entries. Directory entries are modeled as partial mapping from name to inode number. The main contribution of Hesselink et Lali's work is the proof that the inode-based specification is a refinement of their VFS model. In order to prove the refinement Hesselink et Lali introduced an intermediate specification between the two that has a representation of files and directory closer to the VFS model while still being inode-based. However, the VFS model is too abstract to capture the functionality of a VFS layer for POSIX file systems, because it lacks the notion of user-level state such as open files, seek-pointers, etc. Similarly, the inode-based specification is still far from an actual file system implementation. In particular, it omits errors handling and concrete representation of data blocks; all of which we aim to verify with our functional correctness proofs.

Arkoudas et al. [2004] published the first data refinement proof of a file manipulated as a sequences of bytes and stored into a set of fixed size blocks. They used Athena, an interactive theorem prover developed by the authors. Their model describes the implementation of read and write operations more precisely than the research above, but the overall file system remains a simplified one that supports a single directory and assumes infinite storage capacity.

The most realistic flash file system verification work to date is Flashix [Schierl et al., 2009; Ernst et al., 2013; Schellhorn et al., 2014]. Flashix is a userspace flash file system based on UBIFS [Hunter, 2008] and for which the verification is on-going. Flashix is

implemented in the KIV [Reif et al., 1998] theorem prover, which supports a specification language based on Abstract State Machines (ASM) [Börger and Stärk, 2012]. The Flashix verification is organised as a total of 8 refinement layers going from the top-level POSIX file system interface to the flash storage device interface, each layer introducing more implementation details. Several layers of their refinement stack have already been formally verified, including the VFS layer that goes from the top-level POSIX interface to the core file system logic interface [Ernst et al., 2013], the flash memory abstraction that handles eraseblock management [Pfähler et al., 2013] and the on-flash garbage collector [Ernst et al., 2015]. Their earlier work [Schierl et al., 2009] proved the functional correctness of several file system operations of a model of UBIFS, using a different verification methodology closer to Hoare-logic style proofs with pre and post-conditions rather than refinement. Their latest work focuses on crash-safety of the eraseblock management system [Pfähler et al., 2014]. While their ultimate goal is to generate low-level C code, the current code generation from the ASM specifications targets Scala running on the Java Virtual Machine in userspace with FUSE [Szeredi et al., 2010]. Hence, their work requires trusting the code-generation phase and the large language run-time Flashix depends on, which also implies performance overhead. Another current limitation of their work is the lack of support for asynchronous writes.

Chen et al. [2015b] proved the functional correctness of a complete crash-safe UNIX file system implementation called FSCQ. They present a framework for specifying and verifying correctness properties, even in the presence of crashes. Using standard Hoare-logic reasoning and separation logic techniques, augmented for reasoning about crash-safety, their framework also allows them to reason about file system components in isolation. They managed to prove a theorem saying that under any sequence of crashes followed by reboots, FSCQ will recover the file system correctly without losing data. They implement the file system in the Coq proof assistant [Barras et al., 1997], which they use to generate a Haskell implementation that runs as a userspace file system with FUSE [Szeredi et al., 2010]. While this work greatly advances the state-of-the-art in file system verification, FSCQ suffers from a few important limitations that might prevent it from being used on real systems. First, FSCQ does not support the `sync()` operation, hence the file system has to wait for all disk writes to finish before each system call can return to the user. As a result, FSCQ does not support asynchronous writes, a feature imperative for performance[1]. The lack of support for `sync()` means that FSCQ is more than 20 times slower than ext4 with asynchronous writes enabled when running user applications that

[1]Note that FSCQ does supports asynchronous disk operations within a system call, but a file system operation must wait for all writes to commit to disk before returning to the user.

write to the disk. Second, FSCQ's reliance the Haskell's run-time might prevent its use in high-assurance systems on resource-constrained embedded devices, because it decreases the predictability of the system. Finally, the code generation from Coq to Haskell, the Haskell run-time and the FUSE server are all part of the trusted code base, and a single defect in any of these may invalidate the assumptions of their proof.

Our work differ from both Flashix and FSCQ because we present a methodology for verifying low-level file system implementations that do not require a language run-time, and thus can run in the kernel, as a Linux module for instance. We also aim to verify performant file system implementations that support asynchronous writes.

Provably correct crash recovery of file system has drawn a lot of attention over the last two years [Maric and Sprenger, 2014; Pfähler et al., 2014; Ernst et al., 2014; Gardner et al., 2014; Chen et al., 2015b; Koskinen and Yang, 2016; Bornholt et al., 2016]. Much of this research are complementary to ours, for instance it would be interesting to investigate how to integrate Chen et al. [2015a]'s crash-Hoare-logic framework to our methodology and take the verification a step further to include a crash-resilience proof for the file system similar to theirs.

Another stream of work in the literature focuses on more automatic techniques, such as model checking, static analysis and testing [Yang et al., 2006; Gunawi et al., 2008; Rubio-González and Liblit, 2011; Ridge et al., 2015; Li and Wang, 2016]. While in theory these techniques could be used to provide similar guarantees as interactive software verification, 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, rather than ruling them out altogether as we aim to.

## 2.2 Language-based verification

Using language-based techniques to increase the productivity of producing robust low-level system implementations has been an active research area for several years. Muller et al. [2000] proposed using domain-specific languages to implement robust operating system components while fostering code re-use and extensibility.

The verified compiler CompCert [Leroy, 2009], Verisoft's C0 [Leinenbach, 2008], C-to-Isabelle parser [Tuch et al., 2007] and AutoCorres [Greenaway et al., 2014] gives us a formal language semantics to reason about C code using a proof assistant. All of these provide only weak type system guarantees and force the verification engineer to reason about low-level C intricacies when proving the functional correctness of the code.

Cyclone [Jim et al., 2002] and CCured [Necula et al., 2005] are safe dialects of C that ensure the absence of several common file system bugs like buffer overflows, null pointer dereferences, and many other undefined behaviours of C. These languages, however, do not provide a formal semantics to prove higher-level properties about the code such as functional correctness.

Programming languages like CakeML [Kumar et al., 2014] and Dafny [Leino, 2010] come with verified compilation, strong typing and a convenient theorem proving interface, but they rely on a language run-time and a garbage collector whereas COGENT was specifically designed to avoid those.

The Rust [Rust] language comes with high performance, strong type system, and without a language run-time if used in the right subset. But Rust lacks verified compilation, and a formal semantics, so it cannot be used as a foundation for reasoning and formal verification.

ATS [Danish and Xi, 2014] is a ML-like functional programming language which compiles into C. ATS has a linear type system that supports dependent types but does not have a formal semantics, nor verified compilation.

Ivory [Pike et al., 2014] is a domain-specific language embedded in Haskell for implementing programs that interact intimately with the hardware. Ivory compiles to safe C code and is Turing-complete, but it targets the domain of programs that do not require heap allocation of memory, and the code generation to C is unverified.

Habit [HASP project, 2010] is functional language for general system programming, with a verified garbage collector. Although the authors aim to achieve a complete formal description of the language, so that it can be used as a foundation for reasoning and formal verification, in its current state, Habit lacks a formal semantics and verified compilation.

Several data description languages provide operations for converting data between a high-level data format and a bit-level data representation. For example, Idris [Brady, 2013] is a dependent typed domain-specific language directly embedded in Haskell. Idris compiles to C, and is implemented on top of the Ivor [Brady, 2006] theorem proving library, providing an interface for proving functional correctness. However, Idris lacks verified compilation and is only a data description language, thus is not suitable for implementing file system control code. Other data description languages like PADS [Fisher et al., 2006] and PacketTypes [McCann and Chandra, 2000] have no verified compilation and no formal semantics that can be used to reason about functional correctness of programs interfacing with them.

The Fiat library [Delaware et al., 2015] for the Coq proof assistant supports generating efficient, correct-by-construction abstract data types implementations in OCaml. Fiat

targets the code synthesis of simple programming tasks found in mainstream web applications interacting with persistent data stores. By contrast, COGENT is geared towards producing provably correct low-level file system code without a language run-time.

Linear types have sometimes proved too restrictive for general-purpose system programming [Kobayashi, 1999; Smith et al., 2000; Walker and Morrisett, 2001; Aspinall and Hofmann, 2002]. In COGENT, however, we only use linear types to keep track of memory and error handling, and we use COGENT's foreign function interface to implement loops iterators and data structures that require escaping the restrictions of the type system. In addition to strong typing preventing many common file system implementation errors, COGENT's linear type system provides support for automatically verifying the compilation to C.

## 2.3 Modular verification

Feng et al. [2006] present a Hoare-style framework for modular verification of assembly code which aims to be as general as possible. Their framework supports control operations such as function calls/return, long jumps, stack unwinding, coroutine, context switch, etc. The framework abstracts the control operations just mentioned in a way that allows linking together components and proofs developed with different instantiations of these control operations.

Vaynberg and Shao [2012] present a framework that extends from Feng et al. [2006] to verify the functional correctness of low-level code modularly. Vaynberg and Shao applied their framework to verify a small virtual memory manager (VMM) of about 100 lines of C and for which the functionality is decomposed into seven components organised in five layers of abstraction. Their framework allows specifying layers of abstraction via ASM and custom operational semantics enabling them to change the memory model from one abstraction to another — a feature required to verify a VMM for instance. Vaynberg and Shao's provides a meta-theory calculus to compose theorems developed on different levels of abstraction, however, they report that such endeavour requires lengthy relations and proofs[Vaynberg and Shao, 2012].

Gu et al. [2015] present a framework to reason modularly about low-level system code implemented in potentially different languages. The framework relies on each component functionality being described by a *deep specification*. A deep specification is a formal description of the functionality of a component that captures everything we want to know about its implementation, such that if another implementation of the component that satisfies the deep specification is constructed, we may substitute it for the original

component without changing the overall behaviour of the system. Similarly to Vaynberg and Shao, Gu et al.'s deep specification framework can compose theorems developed on different levels of abstractions and for different languages. For example, they instantiate their framework with CLightX, a subset of the C language, and LAsm, a 32-bit x86 assembly subset similar to CompCert's x86 machine model. Gu et al. show the practicality of their framework by verifying three variants of the CertiKOS kernel, which is around 3,000 lines of code (C and assembly).

Much this research focus on composing proofs developed in multiple languages which involves proving relations between different level of abstractions described by the operational semantics of each language. Since all our level of abstractions deal with the same language and memory model, the framework we present in Chapter 6 is much simpler than these. In addition, the methodology presented in this thesis provides a strong theorem proving interface to reason about modularly designed file system code. Formal reasoning about COGENT-generated specifications is done over a purely functional shallow embedding model of the code. Such specification is much easier to read and reason about using a proof assistant than a low-level model of the C code with a mutable memory heap (e.g. CLightX). With purely functional specifications, producing a functional correctness proof is greatly simplified, because we can leverage the prover's built-in automation (e.g. Isabelle's rewriting engine: the simplifier) by exploiting equational reasoning.

## 2.4 Summary

A major part of research in file system verification relies on a language run-time and running the file system in userspace, which implies a performance overhead, and undermines predictability of the system, thus is not good enough for high-assurance systems on potentially resource-constrained embedded devices. In addition, none of the research we are aware of verified properties on a POSIX file system that supports asynchronous writes (i.e. `sync()`). File systems typically buffer updates in memory, and updates are propagated to the storage medium periodically, or explicitly via the file system `sync()` operation, meaning that they occur asynchronously. Asynchronous writes are crucial for performance, but they complicate the file system specification and implementation.

Chen et al. [2015a] deal with an asynchronous disk interface where several disk writes can be queued, but every file system operation has to wait for all writes to be flushed out to disk before returning to the user.

Previous languages for writing verified code either require a garbage-collector or a run-time, hence are not suitable for our purpose. Others do not have a formal semantics geared

to proving higher-level properties such as functional correctness of the code implemented in the language.

We address this by providing a language-based verification framework that does not require a language run-time. We generated C code from COGENT as well as a COGENT specification designed to facilitate straightforward reasoning about COGENT code, which is an accurate model of the C code.

Previous work in modular verification of system software does not integrate easily with existing frameworks. So we designed a verification framework to connect to COGENT-generated specification to a high-level correctness specification. Many of these are complementary to ours, they address the problem of linking components implemented in different languages with potentially different memory models. Our framework is tailored for reasoning about highly modular file system control code with simple data sharing in cross component interfaces. Our approach is geared around equational reasoning over purely functional shallow-embedding so that we can make the most of the prover's built-in automation in order to increase verification productivity as much as possible.

# 3 Modular Design

---

This chapter draws on work presented in the following papers:

- Gabriele Keller, Toby Murray, Sidney Amani, Liam O'Connor-Davis, Zilin Chen, Leonid Ryzhyk, Gerwin Klein, and Gernot Heiser. File systems deserve verification too! In *Workshop on Programming Languages and Operating Systems (PLOS)*, pages 1–7, Farmington, Pennsylvania, USA, November 2013. doi:10.1145/2525528.2525530;

- 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, and Gernot Heiser. Cogent: Verifying high-assurance file system implementations. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 175–188, Atlanta, GA, USA, April 2016. doi:10.1145/2872362.2872404.

---

In Chapter 1 we presented the key ideas of our methodology to design, implement and verify trustworthy file systems. In this chapter, we focus on the design phase of the methodology and provide the key design ingredients required to make a file system verifiable. We illustrate these ideas on BilbyFs, our flash file system that we use as a case study throughout this thesis.

The complexity of modern operating systems' storage stack makes it challenging to specify and design a verifiable file system for several reasons:

1. File systems typically run in the context of a multiprocessor system which can invoke operations concurrently. From a verification perspective, the complexity

introduced by such concurrent calls is considerable. File system concurrency is hard to specify, and even harder to verify.

2. For efficiency, file system implementations and caching mechanisms are tightly coupled with the memory management subsystem and the Virtual File system Switch (VFS) layer. This is not specific to one type of operating system, Windows systems have the "File Buffering" and "File caching" layers [MSD-IO-CONCEPTS], most unix-like systems have a buffer cache and an inode cache of some sort. Tsai et al. [2015] note that FreeBSD, OS X, and Solaris, "all have a directory cache that is structurally similar to Linux's".

   The interfaces through which storage stack components communicate are loosely defined and not structured in a way that facilitates formal reasoning, as noted by Alagappan et al. [2015]. Storage API documentation is often insufficient to formally specify the semantics of operations, thus such specifications have to be manually inferred from the source code with support from the developers.

3. Fully verified software is hard to produce and often imposes practicality tradeoffs to its users. The difficulty of producing large machine-checked correctness proofs of systems is mostly determined by the size and complexity of the implementation of the system. In particular, it is desirable to be able to reason modularly about the implementation by proving the correctness of each code component in isolation.

4. For file systems, we need to be very careful about not introducing performance overhead by oversimplifying the design. The complexity of file systems is the consequence of performance optimisations such as asynchronous writes. When an application updates the file system, the changes are buffered in memory and synchronised to disk later on. Synchronisation happens either when a periodic timer is triggered or if an application directly calls `sync()`. Implementing asynchronous writes is a significant source of complexity because it increases dramatically the number of possible states that need to be considered when proving the correctness of the code.

We make several design decisions to make file system verification tractable.

1. As explained in Section 3.1, we prevent multiple file system operations from running concurrently by acquiring a global lock every time we invoke an entry point of the file system.

2. Section 3.2 shows that we can decouple the file system implementation from the caching mechanisms so that we can focus on verifying the core file system functionality and leave the correctness of caching layers for future work. The resulting file system interface is cleaner, more modular and portable. It can be easily specified using an abstract logical representation of the file system as we demonstrate in Chapter 5.

3. We rely on strong code modularity to manage the complexity of the implementation by decomposing the file system functionality into a set of components each of which can be reasoned about in isolation. We keep the verification tractable by chosing a simple log structured design for BilbyFs where the index is only kept in memory. Although BilbyFs' design is simple, it supports asynchronous writes which is crucial for the performance of the file system. We describe BilbyFs' modular design and its limitations in Section 3.3.

4. All our design decisions were made carefully in order to avoid introducing unacceptable performance overhead. We implemented a C prototype of BilbyFs modular design. We aimed to keep the performance overhead lower than 20% compared to other flash file systems (apart from benchmarks where fined grained concurrency matters). We show that we achieved our objective in Section 3.4

## 3.1 Concurrency

Proofs about concurrent programs are much harder than proofs about sequential ones. While there exists promising approaches to construct verifiable systems on multiprocessors, at the moment none of them would scale to the size of a file system implementation, and they are outside the scope of this thesis as we leave concurrency for future work.

In order to ensure sequential execution of file system code without enforcing sequential execution of the entire operating system kernel, we write a small wrapper around each file system operation that acquires a global lock before invoking the operation.

We put a semaphore in the file system state and invoke file system operations `fsop()` as follows:

```
1  down(&bi->wd.lock);
2  fsop(bi);
3  up(&bi->wd.lock);
```

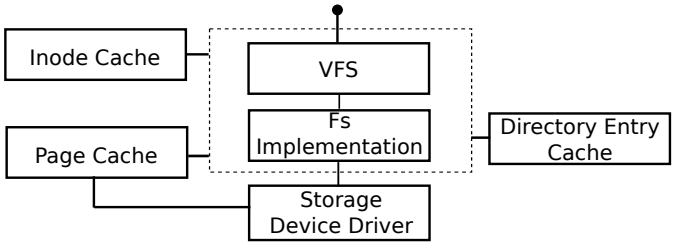


**Figure 3.1:** Overview of file system interactions with caches in the Linux kernel

This way, the invocation of file system operations (e.g. `fsop()`) is guaranteed to be serialised. In this example, `bi` is BilbyFs' state containing the semaphore used to implement our concurrency policy.

However, some data structures like Virtual File system Switch inodes (VFS inodes; described in Section 3.2) may be updated by other parts of the kernel (that are unaware of the `bi` semaphore), concurrently to file system operations. Certain updates to these structures must be performed with kernel preemption disabled to ensure maintenance of VFS invariants. A good example is modifications to the `i_size` field, which must be updated with kernel preemption disabled, so that it always remains consistent with another field used internally by VFS, namely `i_size_seqcount`.

To avoid reasoning about synchronisation primitives in file system code, we implement small write accessors to update inode fields that require locking or disabling kernel preemption. We axiomatize the behaviour of these write accessors when we verify the code.

## 3.2 Isolating caching mechanisms

The file systems interface is tightly coupled with the memory management subsystem and the *virtual file system switch* (VFS), which is Linux's layer that combines the functionality shared by all the file systems available in the kernel. For instance in Linux, file systems must implement an interface cluttered with interactions with the inode cache, the directory entry cache and the page cache.

Figure 3.1 gives an overview of the current architecture of part of the Linux storage stack. The diagram is merely an attempt to define some logical boundaries between different layers of the stack and associate a name to them. In practice, all these components are strongly interconnected (denoted by the dotted box). For example, directory entries located in the *Directory Entry Cache* contain C pointers to the inodes in the *Inode cache*, which in turn reference memory in the *Page cache* that holds memory pages of data

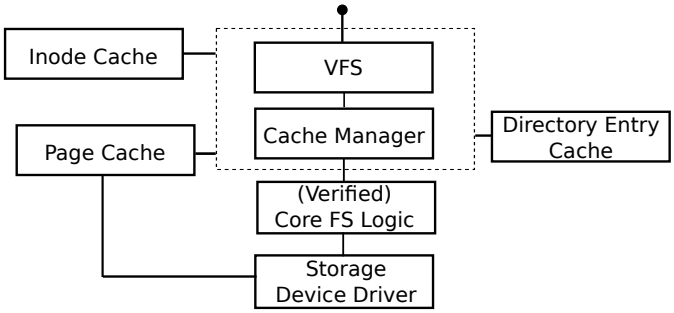


**Figure 3.2:** Reorganisation of caching layers for verifiability

blocks that belong to files and directories. We describe the functionality of each of these caches in the rest of this section.

The VFS invokes the *Fs Implementation* (file system implementation) component to perform operations such as reading and writing files and directories. The file system implementation must handle all the surrounding caches directly; there is no separation between the core file system logic and cache interactions.

In this section, we describe how we decouple file system core logic from the interactions with Linux cache mechanisms. Figure 3.2 shows how we isolate cache interactions in a well confined component called *Cache Manager*. The *Core File system logic* is only invoked when the cache does not satisfy the requests made by the VFS component. Beside handling cache interactions, the Cache Manager also implements the global locking policy described in Section 3.1. By contrast to Figure 3.1, in Figure 3.2 the core file system logic is now completely oblivious to caching mechanisms; it has no direct interaction with any of the caching components. The separation achieves three objectives:

Firstly, the core file system logic that reads and writes on-disk data structures is now separate from cache interactions. Thus we can now reason about the former while assuming the correctness of the latter. In this thesis, we focus on proving that the file system core functionality is correctly implemented, leaving the verification of cache interactions for future work.

Secondly, by decoupling the file system logic from the operating system storage stack, it is easier then to plug our file system into another operating system with a different storage stack, possibly based on seL4 [Klein et al., 2014], a micro-kernel with a mathematical machine-checked proof of functional correctness. This would remove the need to trust the Linux kernel and provide a pathway to truly dependable storage stack. Our design produces a file system interface independent from the operating system, thus not tightly connected to the specifics of cache mechanisms exposed by the Linux kernel.

Finally, our restructuring of file system interfaces avoids the introduction of high performance overhead: it does not reduce the cache hit rates, which would otherwise unnecessarily cause more disk operations.

**The Virtual File system Switch (VFS)** In this paragraph, we give a quick overview of the VFS functionality and discuss the challenges of isolating cache mechanisms from the core file system logic.

Since the VFS interacts with a wide range of file systems, each of which has its own custom inode format, the VFS provides a common inode abstraction that we call a *VFS inode*. Once the file system core logic reads an inode from disk, it converts it to a VFS inode which is kept in memory in the *inode cache*. The purpose of the inode cache is to speed up further accesses to inodes.

As VFS inodes are shared by all file systems in Linux, it is hard to isolate Linux specific inode manipulations from the file system logic. However, as we show in the rest of this section, it is possible to provide an abstraction for VFS inodes much simpler than the one directly exposed by the Linux kernel.

VFS inodes are much more complex than their corresponding on-disk data structure; they have their own slab allocator for efficient memory allocation; they have five different locks to prevent concurrent access to some fields and they have a reference counter to ensure the inode is not destroyed by one thread while another thread holds a reference for it. VFS inodes also contain several function pointers to be filled with file system callbacks so that the VFS code can deal with VFS inodes from several different file systems in a uniform way. Hence, when we allocate a new VFS inode, we need to make sure that its callbacks are consistently initialised with its type. For instance, the VFS expects directory inodes to have the function pointer `iterate()` initialised with the file system `readdir()` operation, whereas for a file inode, the VFS' expectations differ.

For the most part, the file system core logic (see Figure 3.2) treats VFS inodes as simple data structures that match their counter-part stored on-disk. It only needs to manage reference counters and locking primitives in a few places, typically not far from the entry points of file system operations. This last point suggests that isolating inode caching from the core file system logic is feasible.

**Inode cache** Every time an inode is read off disk, it is converted to a VFS inode and cached in memory; until it gets evicted from the cache. Inodes can only be evicted from the cache when the inode reference counter is zero, i.e. all user applications have closed the file. However, it is up to the cache policy to decide when to evict the inode from

```
1  static int bilbyfs_create(struct inode *dir, struct dentry *dentry,
2                            umode_t mode, bool excl)
3  {
4          struct bilbyfs_info *bi = dir->i_sb->s_fs_info;
5          struct inode *inode;
6          int err;
7
8          inode = bilbyfs_new_inode(dir->i_sb, dir, S_IFREG | mode);
9          if (IS_ERR(inode))
10                 return PTR_ERR(inode);
11         down(&bi->wd.lock);
12         err = fsop_create(bi, dir, dentry->d_name.name, mode, excl, inode);
13         up(&bi->wd.lock);
14         if (!err) {
15                 insert_inode_hash(inode);
16                 d_instantiate(dentry, inode);
17                 return 0;
18         }
19         make_bad_inode(inode);
20         iput(inode);
21         return err;
22 }
```

**Figure 3.3:** BilbyFs Cache Manager for the *create()* operation

the cache and recycle the memory it occupies. When the file system no longer needs a VFS inode, it cannot simply free it, instead the inode needs to be marked as "bad" which schedules it for recycling. The Linux kernel function *iput()* is used to decrement the reference counter of a VFS inode and potentially free it.

The inode cache interface is very specific to Linux, so our intent is to isolate handling of VFS inodes outside the core file system operations so that our implementation can be verified independently of Linux. To this end, we pre-allocate VFS inodes in the Cache Manager when they might be needed by the core file system logic, and we handle the reference counting in our Cache Manager (see Figure 3.2).

Figure 3.3 shows the Cache Manager implementation for the *create()* operation. When the VFS invokes the Cache Manager operation to create a file, it calls the function *bilbyfs_create()*. *bilbyfs_create()* takes four arguments: the directory inode *dir*, a directory entry *dentry* holding the name of the file to create, *mode* specifying a set of flags for the file to create and a boolean value *excl* that can be safely ignored. The *create()* operation has to allocate an inode for the new file it creates, thus our Cache Manager pre-allocates an inode by calling *bilbyfs_new_inode()*. *bilbyfs_new_inode()* allocates

an inode from the slab allocator and initialises it with the right set of callbacks for the inode according to its type. On lines 9-10, we check whether a memory allocation error occurred, if so we propagate the error code. Lines 11 and 13 implement the global locking policy described in Section 3.1. On line 12, we invoke the core file system logic operation `create()`, that is oblivious to caching, with a reference to the newly allocated VFS inode variable `inode`. `fsop_create()` updates `inode` with the values that match the on-disk inode created by the operation.

If `fsop_create()` succeeds, we call `insert_inode_hash()` to add `inode` to the inode cache. On line 16, `d_instantiate()` links our new inode to `dentry`, the cached directory entry for the file we just created. This link is useful to tell the directory entry cache that the directory entry is connected to the inode. We discuss the directory entry cache in the next paragraph. The next line returns zero to indicate success. Note that because the directory entry is now linked to `inode`, we need to keep the reference number incremented so that the inode does not get evicted from the cache. When `fsop_create()` returns an error, we mark the inode as "bad" on line 19, and we decrement the reference counter on the next line, before propagating the error code.

This example shows how we isolate inode cache interactions from the core file system logic for `create()`. The same principle can be applied to all operations that allocate inodes except for `lookup()`, that is: `link()`, `mkdir()`, `symlink()`. We address `lookup()` shortly.

However, by pushing inode allocation into an external component, we modified the behaviour of the file system slightly. Normally, when a file system implementation detects an error early (e.g. invalid name for the file), it does not allocate the inode at all. By contrast, we allocate before any check can occur. If the core file system operation returns an error, the inode is not needed. There is little semantic difference between the two implementations but, by allocating inodes unconditionally, our design causes a small performance degradation. However, we only impose extra CPU cycles when an error occurs, in which case performance rarely matters. One notable exception is `lookup()` because checking whether a file exists is a very frequent operation. We address `lookup()` next.

**Directory cache and `lookup()`** The `lookup()` operation is used internally by the VFS to implement the `open()` system call, which provides a way to open files and directories to user applications. The `lookup()` operation reads the contents of a directory to find the *inode number* matching a given name and then reads the corresponding inode data structure from disk. An inode number uniquely identifies an inode, and is used by file

```
1  static struct dentry *bilbyfs_lookup(struct inode *dir,
2                                       struct dentry *dentry,
3                                       unsigned int flags)
4  {
5          struct bilbyfs_info *bi = dir->i_sb->s_fs_info;
6          struct inode *inode;
7          ino_t inum;
8          int err;
9
10         down(&bi->wd.lock);
11         err = fsop_lookup_ino(bi, dir, dentry->d_name.name, &inum);
12         up(&bi->wd.lock);
13         if (err == -ENOENT) {
14                 return NULL;
15         }
16         if (!err) {
17                 inode = bilbyfs_iget(bi, inum);
18                 if (!IS_ERR(inode)) {
19                         d_add(dentry, inode);
20                         return NULL;
21                 }
22                 err = PTR_ERR(inode);
23         }
24         return ERR_PTR(err);
25 }
```

**Figure 3.4:** BilbyFs Cache Manager for the `lookup()` operation

systems to logically link file system objects together on the medium[1] (e.g. link a directory entry to an inode).

In most file systems the `lookup()` operation is very costly, because it requires at least one disk read operation to locate the name in the directory and another to read the inode itself. Thus, it is essential to cache directory entries and inodes to speed up name lookups. The directory entry cache is largely handled by the VFS and the file system implementation is mostly oblivious to it. For example, when a directory entry exists in cache for a file, the VFS does not call the file system operation `lookup()` at all. The VFS caches the result of `lookup()` operations in the directory entry cache even if they return an error indicating the file does not exist. Such entries, called *negative directory entries*, record the fact that a file was not found by `lookup()` in order to speed up upcoming identical requests [Tsai et al., 2015].

[1]Throughout this thesis, we use the term medium to refer to the storage medium that could be a disk or a flash device.

As explained earlier, the first step of a *lookup()* request is to find if there exists a directory entry that matches the name passed as argument and extract the inode number it points to. Once the inode number is known, file systems look up the inode cache to check whether there is already a copy of the inode in memory. If not, an inode is allocated in memory and initialised with the values stored in the inode read from disk. The fact that *lookup()* has to probe the inode cache after reading from the disk and possibly allocate a new VFS inode, makes it difficult to isolate interactions with the inode cache.

Our solution to completely isolate inode cache mechanisms is to break up the Linux *lookup()* operation into two sub-operations: *iget()* and *lookup_ino()*. Both operations provide core file system functionality. *lookup_ino()* finds the matching directory entry on-disk, and returns the inode number associated with the entry or an error if it does not exist. *iget()* merely reads an inode from disk given an inode number as argument.

Figure 3.4 shows the Cache Manager code for *lookup()*. *bilbyfs_lookup()* is invoked when an application attempts to open a file or a directory. The operation takes three arguments: *dir* the directory inode; *dentry* a directory entry holding the name of the file to look-up and *flags* that can be safely ignored here[2]. *dentry* is "negative", meaning it is not linked to an inode. For historical reasons, file systems can allocate their own directory entry and discard the one passed as argument. If a file system decides to do so, it simply returns the newly allocated directory entry. In our case, unless an error occurs, *lookup()* always returns *NULL* to indicate that the file system did not allocate its own directory entry. VFS detects whether the file was found or, not by checking whether the directory entry is still negative i.e. is not linked to a VFS inode. On line 11, we call the core file system logic function *fsop_lookup_ino()*, which takes four arguments: *bi* the file system state; *dir* the directory inode; the name of the file to lookup; and *&inum* a reference to a local variable to be filled with the inode number, if the operation is successful. If *fsop_lookup_ino()* returns an error code, we use *ENOENT* (no entry found) to indicate that the name was not found in the directory. On lines 13–14, we return *NULL* instead of *ENOENT* because, as mentioned earlier, we want the VFS to cache the fact that no file with such name exists. The condition on line 16 is true if *lookup_ino()* was successful. In this case we call a helper function *bilbyfs_iget()* that will return the inode with *inum* as inode number, either by fetching it from the inode cache or by reading it from the disk. If the *iget()* operation is successful, we link the directory entry to the returned inode using *d_add* and we return *NULL*. On line 18, we use Linux's macro *IS_ERR* to detect whether the

[2]The flags indicate what part of the Linux VFS is calling lookup allowing *lookup()* to behave differently when it is called from the *open()*, *create()* or *rename()* system calls. The only file system that uses these flags in the Linux kernel is NFS.

```
1  static struct inode *bilbyfs_iget(struct bilbyfs_info *bi,
2                                    unsigned long inum)
3  {
4          struct inode *inode;
5          int err = 0;
6
7          inode = iget_locked(bi->vfs_sb, inum);
8          if (!inode)
9                  return ERR_PTR(-ENOMEM);
10         if (!(inode->i_state & I_NEW))
11                 return inode;
12         down(&bi->wd.lock);
13         err = fsop_iget(bi, inum, inode);
14         up(&bi->wd.lock);
15         if (!err) {
16                 init_inode_by_type(bi->vfs_sb->s_bdi, inode);
17                 unlock_new_inode(inode);
18                 return inode;
19         }
20         iget_failed(inode);
21         return ERR_PTR(err);
22 }
```

**Figure 3.5:** BilbyFs helper function Cache Manager *lookup()*

*inode* pointer is used to encode an error. If it is we convert the error code to an integer on line 22 and convert it back to a pointer of different type on line 24.

Figure 3.5 presents *bilbyfs_iget()*. Its purpose is to return a VFS inode given the inode number *inum* passed as argument. If the inode is in the inode cache, we fetch it from the cache, otherwise we read from disk. The VFS provides *iget_locked()* to retrieve an inode from the inode cache. When the inode is missing in the cache *iget_locked()* allocates a new inode and returns it. Thus, the only reason *iget_locked()* might return an error, i.e. *NULL*, is when a memory allocation error occurred. We handle this case on lines 8–9. To distinguish between a cache miss and a cache hit, we need to check the *I_NEW* flag in the *i_state* field which is only set for newly allocated inode. If we hit the inode cache, we can return the inode directly from the cache (lines 10–11). Otherwise, we read the inode from disk by calling the file system core logic function *fsop_iget()* on line 13. When successful, *fsop_iget()* returns zero. Before returning the inode to the VFS, we need to initialise the set of callbacks according to the type of the inode. We do so on line 16 by calling *init_inode_by_type*. On the next line, the VFS function *unlock_new_inode()* clears the *I_NEW* flag while following the appropriate locking rules.

The fully initialised inode is returned on line 18. When `fsop_iget()` returns an error, we simply invoke VFS' `iget_failed()` function to mark the inode as bad and we release it so that it gets recycled by the cache.

**Page cache** The last obstacle to completely isolate cache mechanisms from the file system logic is the page cache. The page cache speeds up accesses to disk blocks that have been read or written to in the past. It works by mapping blocks to memory pages and bookkeeping these in memory so that they can be accessed quickly given an index in a file.

Similarly to the directory entry cache, the VFS checks whether a block is available in the page cache before invoking a file system operation to read it. Hence, file system implementations do not need to fetch a page from the cache in order to satisfy a request. However, they need to populate the page cache.

Due to the proximity of the page cache with the virtual memory manager, handling page cache interactions in file system code is a very hazardous task. The file system must make several memory manager API calls in order to put a page in the page cache. The semantics of many of these API functions are undocumented. As reported by Gorman [2007]: "Virtual Memory is fully understood only by a small number of core developers".

Figure 3.6 shows how the Cache Manager deals with the page cache for the `readpage()` operation. When an application reads a file, it passes a buffer that must be filled with data read from disk. The VFS handles the `read()` system call by calling `readpage()` as many time as needed to read the file in page-size chunks.

The `readpage()` function is implemented by `bilbyfs_readpage()` which takes two arguments: `filep`, a pointer to a VFS structure representing files, and `page`, a pointer to a memory page. The function is expected to return zero when successful and an error code otherwise. In order to access the data referenced by `page`, we need to map the page in the current address space. We do so by calling the function `kmap()`, which returns the address at which the page was mapped. On line 10, we call the core file system logic function `fsop_readpage()` to read the data blocks of `inode` at index `page->index`. `page->index` is the index of the page if we were to divide the data stored in the file in chunks of the size of a page. `fsop_readpage()` stores the blocks read in the memory page referenced by `page`. Our core file system logic interface is designed such that when `fsop_readpage()` does not find any data block at the index passed as argument, it returns `ENOENT`. However, the VFS expects `readpage()` to return zero even when no data block was found, so we convert the `ENOENT` error to the value zero. This choice is purposeful to allow `fsop_readpage()` to better support verification, by more precisely constraining its behaviour for each error

```
1  static int bilbyfs_readpage(struct file *filep, struct page *page)
2  {
3          struct inode *inode = page->mapping->host;
4          struct bilbyfs_info *bi = inode->i_sb->s_fs_info;
5          void *addr;
6          int err;
7
8          addr = kmap(page);
9          down(&bi->wd.lock);
10         err = fsop_readpage(bi, inode, page->index, addr);
11         up(&bi->wd.lock);
12         if (err == -ENOENT)
13                 err = 0;
14         if (err) {
15                 ClearPageUptodate(page);
16                 SetPageError(page);
17                 flush_dcache_page(page);
18                 kunmap(page);
19                 unlock_page(page);
20                 return err;
21         }
22         SetPageUptodate(page);
23         ClearPageError(page);
24         flush_dcache_page(page);
25         kunmap(page);
26         unlock_page(page);
27         return 0;
28 }
```

**Figure 3.6:** BilbyFs Cache Manager *readpage()* operation that interacts with the page-cache

code. By returning a specific error code, we can easily differentiate between an improper request for reading a non-existing block and a successful one.

When *fsop_readpage()* returns an error, we need to carefully update flags to indicate to the page cache what to do with the page. On line 15, we indicate that the page is not up-to-date, meaning that the page cache still needs our support to retrieve its contents from disk. On the next line we set the self-explanatory error flag to the page. *flush_dcache_page()* flushes the CPU data cache line for *page*. We unmap *page* on line 18, and we unlock the page so that it can be used elsewhere and finally we propagate the error code. The success case is very similar except that instead of clearing the up-to-date flag, we set it, and we do the opposite for the page error flag.

By doing all the page cache handling in *bilbyfs_readpage()* and only passing a pointer

to the actual memory page to be filled in to `fsop_readpage()`, we removed all interactions with the page cache from the core file system logic.

The same idea is applied to isolate page cache mechanisms from the VFS operations `write_begin()`, `write_end()` and `write_page()`. The rest of file system operations do not interact with any VFS caches, so their Cache Manager wrapper only implements the concurrency policy described in Section 3.1.

**Portability** An earlier version of BilbyFs was successfully ported to a proof-of-concept VFS implementation running on *CAmkES* [Kuz et al., 2007], the component platform for micro-kernel based operating systems. Of course, we had to use compatible data structure interfaces for both platforms in order to compile with no source code change. In particular, VFS inodes of both platforms must have the fields accessed by the core file system logic. The VFS implementation on CAmkES is significantly simpler than Linux's. There is no directory entry cache and the inode cache is greatly simplified. Yet we were able to run the file system with very little modification. In fact the only modifications we had to make were to fix bugs uncovered by the difference between Linux and CAmkES environments. For instance, BilbyFs was not initialising all the fields of a VFS inode and it turned out not to cause any problem on Linux since its VFS implementation was initialising these fields by default. The fix consisted of fixing BilbyFs to initialise the field to the right value.

Although our focus in this thesis is on BilbyFs and the storage stack for raw flash file systems, all the ideas presented so far can be applied to the storage stack for block file systems too.

## 3.3 Design Modularity

We might reasonably expect that the benefits of software engineering principles like separation of concerns [Dijkstra, 1982] and information hiding [Parnas, 1972] to also apply to verification. If a developer can decompose file system functionality into components that can be specified and implemented in isolation, proving the correctness of each component can in principle also be done in isolation.

An examination of Linux file systems suggests that their C code is naturally amenable to modular decomposition. File systems typically manipulate a large state which is passed around as a pointer to a C structure with several fields. Functions that only operate on a few fields of the state, often take the whole state as pointer. By contrast, a carefully

designed file system could be decomposed modularly, such that each component of the decomposition only takes the part of the state it needs to provide its functionality.

Design modularity is a key aspect of our approach. As we demonstrate in Chapter 6, it allows us to reason about components of the file system in isolation which is highly desirable for managing the proof effort and increasing verification productivity. We decompose the implementation into a set of (logical) components that communicate through well defined interfaces specified by simple abstractions. When designing verifiable components we often have to manage a trade-off between simplicity and efficiency of the implementation. By structuring the system this way, we aim to obtain a design that enforces strong separation of concerns between components and facilitates both formal and informal reasoning, without sacrificing performance. In the rest of this section we describe the modular decomposition of BilbyFs and we discuss design decisions we made to facilitate modular reasoning about the file system functionality. In Section 3.4 we evaluate the performance of our modular design by showing that BilbyFs performance is within 20% when compared to the most similar existing Linux flash file systems.

### BilbyFs introduction

BilbyFs targets small safety critical embedded systems that require strong reliability guarantees on the storage. BilbyFs is a highly modular flash file system that combines the design simplicity of *JFFS2* [Woodhouse, 2003] with the runtime performance of *UBIFS* [Hunter, 2008]. Many of the design decisions we made when designing BilbyFs were informed by a careful examination of the strengths and weaknesses of these two file systems. BilbyFs is built on top of the flash abstraction layer *UBI* [Gleixner et al., 2006] and is designed to live underneath the virtual file system switch (VFS) layer described in Section 3.2. When BilbyFs is plugged into the Linux kernel, user applications can use it via the standard POSIX file system interface. BilbyFs supports all operations specified by the POSIX standard, including symbolic links and hard links. The current implementation of BilbyFs does not support extended attributes, but this decision was made in order to reduce implementation effort. There is no fundamental limitation preventing us from implementing such a feature.

To facilitate compositional verification, we make sure that components communicate through well-defined interfaces. Whenever it can be done without unacceptably sacrificing performance, we trade design flexibility for verification simplicity, by avoiding complex component interactions involving callbacks.

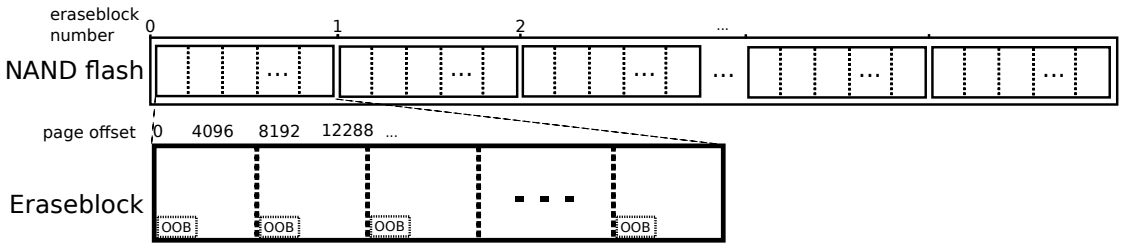


**Figure 3.7:** Internal structure of a NAND flash device, pages in eraseblocks must be written sequentially

### Flash file systems

Raw flash devices, which are mostly found in embedded devices, provide a different interface to file systems than the more commonly used rotational disks or solid state drives (SSDs). Rotational disks and SSDs are block devices that provide an interface exposing an array of blocks; typically of size 4 K each. Blocks are the minimum data unit that a disk file system can read or write. Each block can be randomly accessed via its block number.

By contrast, raw flash devices provide a more restrictive interface. There are two types of such flash devices: NAND and NOR. Since NAND flash devices are a lot more common than NOR ones, we focus on NAND flash. Nevertheless, BilbyFs should transparently support NOR devices thanks to the UBI [Gleixner et al., 2006] flash abstraction.

The internal structure of NAND Flash devices is more complicated than disks. They are made of an array of *eraseblocks*; typically of size 512 K each, i.e. much larger than disk blocks. An eraseblock is itself an array of *pages* that must be written in sequential order. The minimum unit for reading or writing data to the flash is a page, typically of size 4 K, i.e. similar to a disk block. Some NAND flash devices have a small spare area in every page, called the *out of band* area. The out of band area is typically 1/32 of the page-size and it can sometimes be used by file systems [STLinux].

Figure 3.7 shows the internal structure of a NAND flash with the separation between eraseblocks and flash pages. The key difference between disks and flash devices is that the latter's interface to write to the device is more restrictive. Pages in eraseblocks can be read randomly but, as mentioned earlier, they have to be written in sequential order. Another limitation of the NAND flash interface, is that it forbids the file system to overwrite a page until the entire eraseblock has been erased.

Erasing damages an eraseblock permanently and the operation is two orders of magnitude slower than writing a page [Hunter, 2008]. An eraseblock can typically be

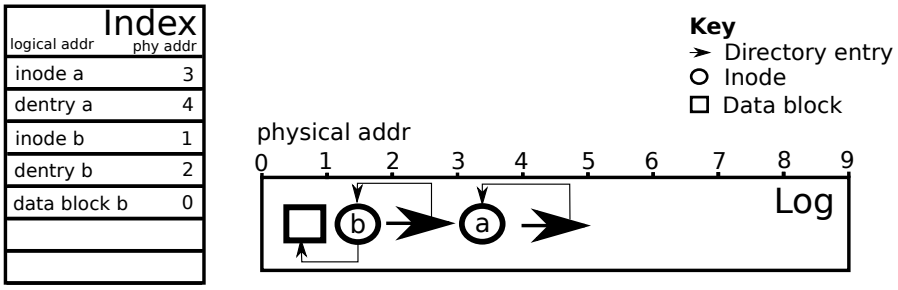


**Figure 3.8:** Log representation with file system objects and the index

erased about 100,000 times, before the wear begins to deteriorate its integrity.

To circumvent these restrictions flash file systems write data out-of-place and spread writes over the entire flash device as much as possible, in order to level the wear of the flash across all eraseblocks and minimise the number of erasures. This concept, named *wear levelling*, is transparently implemented by UBI, which we describe later in this section.

### Log-structured flash file systems

BilbyFs is a log-structured [Rosenblum and Ousterhout, 1992] flash file system. The log-structured file system design is a natural fit for raw flash devices, because it naturally accommodates out-of-place updates imposed by the flash interface.

The basic idea of log-structured file systems is simple: the entire medium is treated as a sequential circular log. Every update of the file system appends a record to the log. A record can consist of file system objects such as data blocks, inodes and directories, or any meta data managed by the file system. In order to be able to retrieve information from the log, the file system maintains indexing information in a structure called the *index*. When stored on medium, file systems objects link to each other using logical addresses. The index maps the logical address of each object to its on-medium physical address.

Figure 3.8 shows the log with file system objects and the logical links between them. For now, we can assume that the log is not circular, later in this section we explain how flash file systems implement a circular log. Every time an object is appended to the log, its address is recorded in the index. Links between objects in the log go via the index. For instance, the inode labelled "a" at physical address 3 is referenced by the directory entry located at physical address 4. When stored on medium, the directory entry and inode "a" are linked with logical addresses. The file system needs to look up the index in order to find that the physical address of inode "a" is 3.

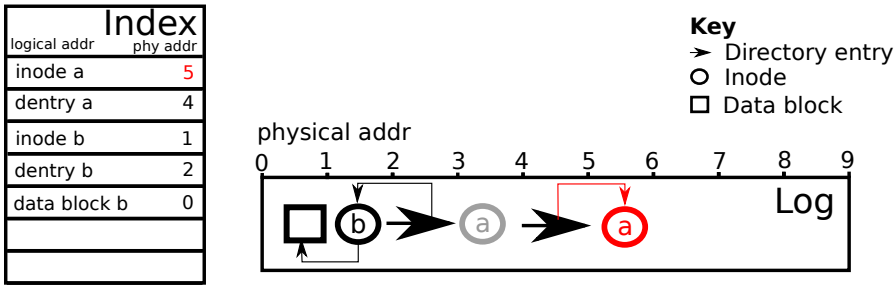


**Figure 3.9:** Update file system object in the log and the index

When a file system object (e.g. an inode) is updated, a new instance of the object that supersedes the previous one is appended to the log. As a result, the index is updated with the new physical address of the object. Figure 3.9 shows the modifications incurred by updating the inode "a" in the log, with updated parts of the diagram colored red. The address of inode "a" stored in the index is updated to 5.

Notice that the old version of the object remains in the log even though it is not referenced by any file system object anymore. We call such objects *obsolete*, and we color them in grey in our diagrams. Since the file system keeps adding objects to the log, eventually the log will be full, and a large part of it will be occupied by obsolete objects. We call the space wasted by obsolete objects *dirty space*. Log-structured file systems need to recycle dirty space by implementing garbage collection. In order to garbage collect efficiently, the file system needs to keep track of how much free space is available on medium, and how much dirty space is occupied by obsolete objects.

Before we explain garbage collection, we describe the log more precisely and we explain how it is made circular. The log is made of segments, and a segment is typically an eraseblock for a flash file system. The file system keeps track of a list of free segments and when the log is full, it allocates the next segment to use for the log from this list. A segment is also the unit of garbage collection. When a segment contains a lot of dirty space, it is recycled by appending the few live objects it contains to the end of the log, so that they supersede their previous version. This process creates a segment full of dirty space (i.e. all its objects are obsolete), which can safely be removed from the log, erased and added to the list of free segments.

Figure 3.10 shows the log with one segment containing a lot of dirty space (i.e. grey objects). When the garbage collector decides to clean this segment, it copies its live objects at the end of the log (segment 2), it updates the index so that all the logical links remain valid and do not reference segment 0 anymore. Finally, the garbage collector removes the segment from the log.

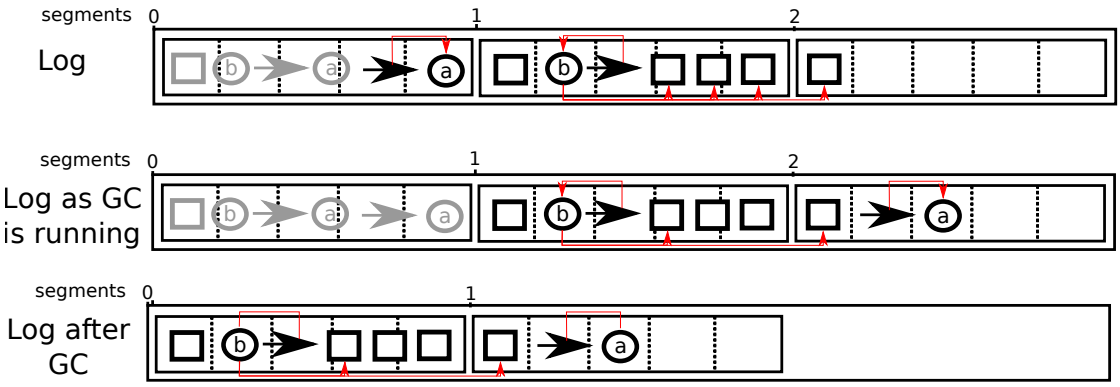


**Figure 3.10:** Garbage collection workflow

Note that the segments are not necessarily consecutive eraseblocks on flash. Since garbage collection can shuffle segments around by removing dirty segments and allocating empty ones, the file system needs to store some metadata in each segment to indicate its position in the log. A common technique used in flash file systems, and taken on by BilbyFs, is to store an increasing sequence number (64 bit machine word) in each log entry. Then, by reading the first log entry of each segment we can reconstruct the order in which they were allocated.

### Memory vs on-flash index

One of the main sources of complexity in UBIFS is to keep the index synchronized in memory and on flash [Hunter, 2008]. Storing the index on flash is hard, because just like any other metadata, it must be updated out-of-place. We examined the source code of UBIFS and found that a large fraction of its code (UBIFS full source code is 22 K source lines of C code) is devoted to updating the index on flash. By contrast, in BilbyFs we opted for simplicity: we only keep the index in memory, which allows us to keep of BilbyFs under 4 K source lines of C code. One obvious disadvantage to this is that the index has to be reconstructed at mount time by scanning the flash. We describe how we speed up BilbyFs' mount time with *summaries* later in this section, and we evaluate mount time performance in Section 3.4.

### BilbyFs Modular decomposition

Figure 3.11 shows the design of BilbyFs. The FsOperations component implements the interface expected by the Cache Manager wrapper code, as described earlier in Section 3.2. The FsOperations component is a client of the ObjectStore component. ObjectStore

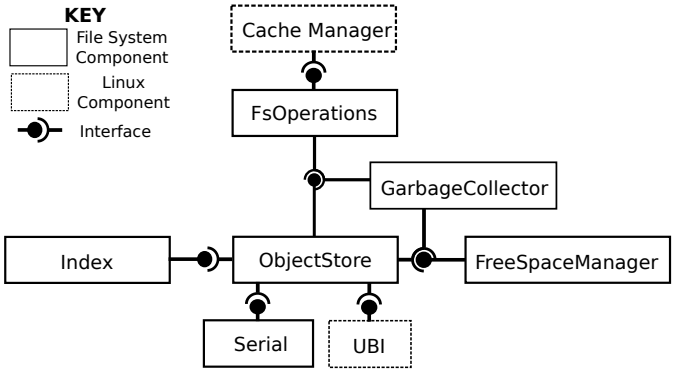


**Figure 3.11:** BilbyFs modular decomposition

provides a uniform API for storing generic objects on flash, which FsOperations uses to implement file system objects such as inodes, directory entries and data blocks. This decomposition confines the key file system logic to FsOperations, and makes that component independent of the on-medium representation of file system objects.

ObjectStore is used to store file system objects where each object has a unique identifier that can be used to retrieve, modify or delete the object. ObjectStore enables grouping several object updates in a transaction to ensure atomicity of file system updates when a crash happens. In order to increase throughput and meta-data packing, ObjectStore provides an asynchronous interface to write objects, i.e. updates are buffered in memory and synchronised to flash later on, either when a periodic timer is triggered, or when the client calls the `sync()` operation to flush the buffer to flash. ObjectStore relies on two components: the Index, which stores the on-medium address of each object and FreeSpaceManager (free space manager). FreeSpaceManager keeps track of how much dirty space is present in each eraseblock. As explained earlier in this section, the dirty space is the amount of obsolete data resulting from overwriting or deleting objects out-of-place. FreeSpaceManager also serves as a bookkeeper for unused eraseblocks and deleted objects in the store.

ObjectStore's implementation itself relies on the Serial component to serialise and de-serialise file system object to and from a buffer, and the UBI layer to read, write and erase flash blocks. Finally, the garbage collector which in turn depends on FreeSpaceManager, maximises the space available for the ObjectStore component by recycling eraseblocks with a lot of dirty space. Ideally, the garbage collector would run whenever the system is idle and stop when file system operations are invoked concurrently. But since our verification technology does not support reasoning about concurrency, the current design calls the garbage collector when the file system runs out of free eraseblocks. This design is

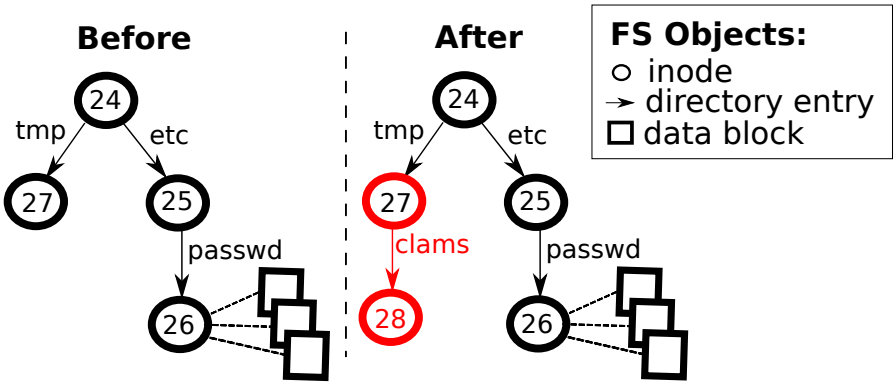


**Figure 3.12:** Create operation on-top of ObjectStore

similar to UBIFS, which also starts garbage collecting when it runs out of free eraseblocks.

### FsOperations: File System Operations

ObjectStore provides a convenient abstraction to implement file system operations that manipulate files and directories. The logic of these operations can be described concisely by a succession of updates on inodes, directory entries and data blocks. For the file system to remain consistent when a crash occurs, these updates are grouped into a transaction and the implementation of ObjectStore must ensure transactional semantics when writing to the flash. By transactional we mean two properties of database transactions: atomicity, that is no transaction is partially applied; and durability, meaning that once committed, a transaction is guaranteed to persist.

**FsOperations's create implementation** We present the create operation as a worked example of the use of the ObjectStore abstraction to implement file system operations. Figure 3.12 shows a diagram depicting the modification applied to the ObjectStore to create a file. In this example, the file system contains two directories, `/tmp` and `/etc`. `/etc/passwd` is a file composed of three data blocks of file contents. Red colored objects are those that are added or modified by the create operation. Circles, arrows and squares respectively represent inode, directory entries and data blocks. In Figure 3.12, we create a new file `clams` in `/tmp`. The operation involves adding a new directory entry `clams` pointing from the parent inode (inode number 27) to our new inode (inode number 28). The parent inode is red, because when creating a file, the time-stamp and the size of the parent directory inode gets updated. Thus the create operation implemented on top of the ObjectStore simply writes a transaction touching three file system objects. The

transaction combines three modifications: it updates the parent inode, it adds a new inode and a new directory entry.

The code of the create operation from our C implementation of BilbyFs is shown in Figure 3.13. Our object store abstraction greatly facilitates the implementation of file system operations. As a result, the source code for the core logic of the create operation `create()` is only 50 source lines of C code.

Recall from Section 3.2 that file system operations, like `create()`, are invoked by our Cache Manager, which is itself called by the VFS. VFS inodes have the C type `struct inode`. Top-level file system operations can update VFS inodes by dereferencing the pointers passed as arguments. They also return a negative error code to indicate that an error occurred. Much of the complexity of `create()` comes from error handling and manipulating VFS inodes. The rest is implementing the logic of file creation.

The operation takes five arguments: the file system state (`bi`), the directory VFS inode (`dir`), the name of the file to create (`name`), the mode indicating permissions and other flags for the file (`mode`) and a newly allocated VFS inode to be used for the file to be created (`vnode`). The file system state is of type `struct bilbyfs_info` which combines the state of BilbyFs top-level components depicted in Figure 3.11: FsOperations, ObjectStore and GC. In turn ObjectStore's state combines the states of the sub-components it relies upon. The VFS expects that the value of all parameters except for `vnode` and `bi` remain unchanged, when the operation returns an error. For instance, it is incorrect for `create()` to modify a field of the directory inode `dir`, if the operation returns an error. By contrast, `vnode`'s value is irrelevant as the VFS discards it, when an error occurs. We present a formal specification of such behaviours in Chapter 5.

`create()` returns the appropriate error code if the file system is in read-only mode on line 12-13. On line 15, `inode_init()` initialises `vnode` passed as argument with a newly allocated inode number and other inode parameters, some of which are copied from `dir`. The binary "or" with `S_IFREG` ensures that the type of `vnode` is a regular file. `inode_init()` returns an error code when the file system fails to allocate a new inode number for the file, which is propagated on lines 16-17.

On lines 18-21, we store the new directory entry that links the parent inode to the file inode in the ObjectStore. For reasons we explain in the next sub-section, we do not directly store directory entries in the ObjectStore, but instead we store *dentarrs* (directory entry arrays). Conceptually, BilbyFs directories comprise a collection of dentarrs, where a dentarr is analogous to a directory entry in traditional file systems. In order to provide quick lookup of an entry in a directory, each BilbyFs directory entry is indexed by the hash of the name stored in entries. A dentarr can store multiple directory entries that happen to

```
1  int fsop_create(struct bilbyfs_info *bi, struct inode *dir,
2                  const char *name, umode_t mode,
3                  struct inode *vnode)
4  {
5          struct timespec mtime, ctime;
6          struct obj_dentarr *dentarr;
7          obj_id id;
8          int sz_change;
9          int err;
10         void *objarr[3];
11
12         if (bi->is_ro)
13                 return -EROFS;
14
15         err = init_inode(bi, dir, vnode, S_IFREG | mode);
16         if (err)
17                 return err;
18         id = dentarr_id_init(dir->i_ino, name);
19         dentarr = dentarr_read_or_create(bi, id);
20         if (IS_ERR(dentarr))
21                 return PTR_ERR(dentarr);
22         sz_change = dentarr_add_dentry(bi, dentarr, vnode, name);
23         if (sz_change < 0) {
24                 err = sz_change;
25         } else if (dir->i_size > BILBYFS_MAX_FILESIZE - sz_change) {
26                 err = -EOVERFLOW;
27         } else {
28                 objarr[0] = dentarr;
29
30                 mtime = dir->i_mtime;
31                 ctime = dir->i_ctime;
32                 dir->i_size += sz_change;
33                 dir->i_mtime = vnode->i_ctime;
34                 dir->i_ctime = vnode->i_ctime;
35
36                 pack_obj_inode(bi->inode_obj, vnode);
37                 objarr[1] = bi->inode_obj;
38
39                 pack_obj_inode(bi->dir_obj, dir);
40                 objarr[2] = bi->dir_obj;
41
42                 err = ostore_write(bi, objarr, 3, OSW_NONE);
43                 if (err) {
44                         dir->i_size -= sz_change;
45                         dir->i_mtime = mtime;
46                         dir->i_ctime = ctime;
47                 }
48         }
49         kfree(dentarr);
50         return err;
51 }
```

**Figure 3.13:** BilbyFs implementation of create

give the same hash for different names, i.e. they handle hash collisions. `dentarr_id_init()` makes an object identifier (`id`) to access the dentarr that will contain our new directory entry establishing a link from the parent inode to the new one. The directory entry carries the name of the newly added file. If there exists a dentarr with the same hashed name already, `dentarr_read_or_create()` reads it from the object store. Otherwise the function allocates a new dentarr with this identifier. `dentarr_read_or_create()` returns a pointer to the dentarr object when the function is successful. Otherwise, it returns an error code when a memory allocation error occurs. The C macro `IS_ERR` allows detecting whether the value returned by the function is a valid pointer or an error code.

On line 22, `dentarr_add_dentry()` checks whether the name is valid (e.g. not too long). If it is, we append a new entry and we return the number of bytes occupied by the new entry in the directory. Otherwise, an appropriate error code is returned. On lines 25–26, we check that the new size of the directory does not exceed the maximum size of file in BilbyFs. From lines 27 to 42 we create an array of file system objects `objarr` that will form a *transaction*, i.e. either all objects will be updated or none despite possible failures, such as a crash or a memory allocation error. The first object of `objarr` is the updated dentarr that we initialise on line 28.

On lines 30–34, we update the size and timestamps of the directory VFS inode and keep a copy of the previous value. That copy will be used to undo the in-memory changes, if an error occurs when updating the object store (lines 43–47). On lines 36–40, we convert the directory and file VFS inodes to file system objects that can be passed to the object store.

Finally on line 42, we invoke the operation `ostore_write()` to update the object store. The function takes the file system state `bi`, the array of objects forming the transactional update, its length, and a flag that can be used to specify options (e.g. force a synchronous update). `OSW_NONE` indicates that no option is enabled. Finally, we free `dentarr`, the only allocated object on line 49.

This completes the implementation of `create()`. We apply the same ideas to implement all file systems operations on top of the ObjectStore, namely: unlink, mkdir, rmdir, symlink, readdir, link, readlink, read, write. BilbyFs symbolic links are merely a restricted kind of file: they can only have a single data block attached to them and the `I_LNK` must be set in their mode attribute to indicate that they are symbolic links. The operation `sync()` is trivially implemented by calling `ostore_sync()` presented in the next section.

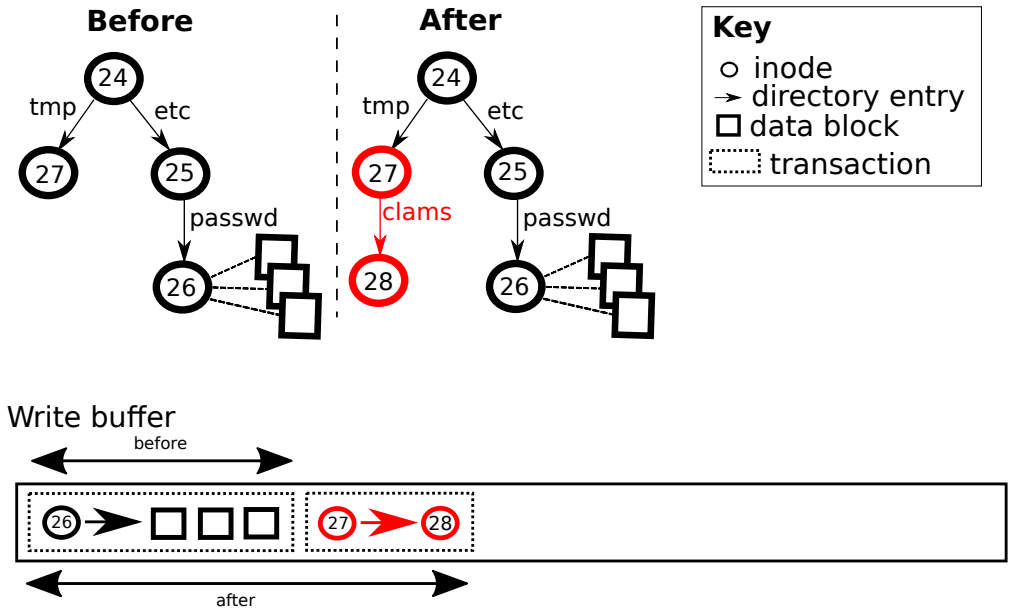


**Figure 3.14:** High-level description of transaction atomicity.

### ObjectStore: Object store

The goal of the ObjectStore is to provide an asynchronous, transactional interface to store file system objects to the flash. The interface allows retrieving, updating and deleting objects from the store. File system objects are uniquely identified with an object identifier that we use to access the store. When the client updates the object store, the changes are written to a buffer in memory called the *write buffer*. The changes stored in the write buffer get applied to the flash when the client of the interface calls `ostore_sync()`. The write buffer is a log-structured segment, i.e. it is formatted as a sequence of transactions, where a transaction is an array of objects.

Figure 3.14 depicts the mapping between the abstract representation of the file system and the write buffer containing the updated objects packed in a transaction. We reuse the example presented earlier in this section where the file "/tmp/clams" is created by updating the parent inode (inode 27), and by adding a directory entry and a new inode (inode 28). All three objects, coloured red, are combined into a transaction serialised into the write buffer. The write buffer is flushed to flash using the UBI operation to write a buffer into an eraseblock. Although such writes are not atomic, the atomicity of transactions is guaranteed by the BilbyFs crash recovery process.

When a crash occurs, BilbyFs checks the integrity of all the transactions in the log

segment that was used to store the write-buffer contents before the crash. We discard all the transactions after the first transaction that fails the integrity check, by only keeping the valid part of the log segment. The segment is cleared of the invalid transactions using the UBI operation that replaces the contents of an eraseblock atomically. This also ensures that the file system is able to recover from crashes that occur during the crash recovery process.

**Object identifiers and Dentarrs** Object identifiers uniquely identify file system objects to be accessed in the store. We use the ObjectStore to access three types of objects: inode, data blocks and dentarrs (directory entry arrays). We can easily build a small unique identifier (e.g. 64 bit integer) for both inodes and data blocks, by encoding a combination of block offset and inode number. But directory entries are harder to uniquely identify, because their uniqueness depends on the name they carry, and the name does not fit in a small fixed size identifier. One way to circumvent this issue is to use a hash of the name as identifier, however, this requires us to handle hash collisions. In BilbyFs we handle such hash collisions by storing an array of directory entry (dentarr) instead of a single directory entry. The array stores all directory entries that belong to the same directory whose hashed names collide.

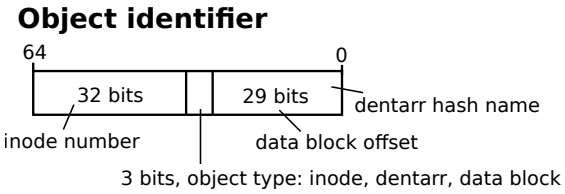


**Figure 3.15:** File system objects identifier encoding

Figure 3.15 depicts the bit format for file system object identifiers, which is similar to that of UBIFS node identifiers [Hunter, 2008]. The first 32 bits (the most significant bits) are used to store the inode number. Depending on next the three bits that determine the type of the object, the last 29 bits are used to store either the hash of the name if the type is dentarr, or a data block offset if the type is data block. For dentarr identifiers, the inode number in the first 32 bits corresponds to the inode number of the directory the entries belong to. When the object type is an inode, the last 29 bits are zeroed. The values for the inode, data and dentarr types are respectively: 0, 1, 2. Note that with three bits, we can encode eight different types of objects, which is more than what we need for BilbyFs. The extra bit enables us to add other types of objects such as extended

attributes, while preserving backward compatibility.

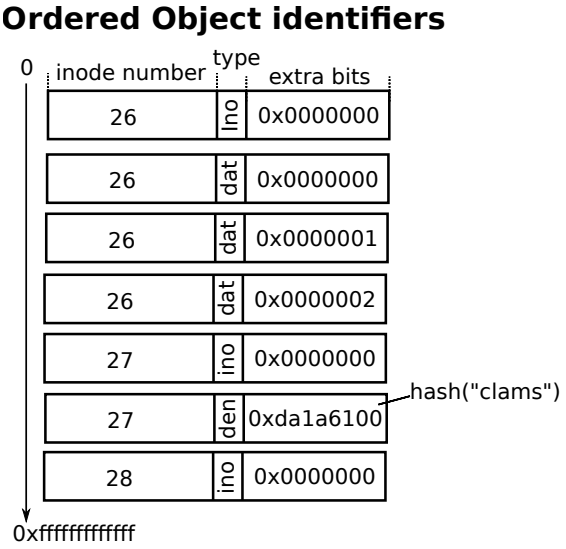


**Figure 3.16:** Ordered object identifiers matching a subset of Figure 3.12 objects

One of the key properties of the identifier format is that an identifier for an inode always has a lower value than the identifiers for objects that belong to it. For instance, for a file inode, all the identifiers of data blocks belonging to that file will be between the inode identifier and the next inode identifier. The same holds for directory inodes as shown in Figure 3.16. This ordering is important for enabling efficient object deletion. BilbyFs also relies on this ordering to list directories: we begin with the directory inode identifier and we iterate through the dentarr identifiers until we have exhausted all the dentarrs in the directory. We now describe the interface exposed by the ObjectStore to access file system objects, using their identifiers.

**ObjectStore's interface** ObjectStore provides the following interface:

- ostore_read: takes an object identifier as argument and returns a pointer to the requested object. If the object does not exist, `ostore_read()` returns an error code.

- ostore_write: takes an array of objects (a transaction) as argument and for each object it adds it (if it does not already exist) or overwrites the existing one with the same identifier. `ostore_write()` also supports writing *deletion objects* that can be used to delete other objects. Although all the changes are made visible to a client

of the interface as soon as *ostore_write()* returns, they are only made persistent once *ostore_sync()* is invoked, or when the buffer is full and needs to be emptied.

- ostore_sync: writes the write-buffer, i.e. the in-memory buffer storing pending transactions, to the flash. This operation waits for the changes to be committed to flash before returning.

- ostore_next_oid: takes an object identifier as argument and returns the next identifier in the index. If the argument is the last identifier in the index, *nilObjId* (-1) is returned. This operation is used to iterate over directory entries when listing a directory.

- ostore_erase: erase the eraseblock given the eraseblock number passed as argument. This operation is only used by the garbage collector when recycling obsolete data from the log. The garbage collector must ensure that there are no live objects in the eraseblock passed as argument before calling this function.

- ostore_scan_objs: takes an eraseblock number and reads all the objects stored in the eraseblock. This function is used internally by ObjectStore when recovering from a crash and by the garbage collector.

**Deletion objects** One of the main challenges in designing the ObjectStore abstraction was to provide an efficient way to delete objects. A naive design would be to provide an operation that deletes an object given an object identifier. This design works, but would lead to poor performance in practice, because some operations need to delete several file system objects at once. This means that they would have to iterate on objects using *ostore_next_oid*, hence repeatedly searching the index at each call. For instance, when a file gets truncated at offset 0, all the data blocks that belong to the file inode should be deleted from the ObjectStore. Asking the client of the ObjectStore interface to repeatedly delete file system objects is highly inefficient, since each inter-component call will unnecessarily have to repeat a lot of the processing that could be batched together if the appropriate interface was provided.

To solve this problem we add deletion objects — an idea analogous to UBIFS's truncation nodes [Hunter, 2008]. When written to the ObjectStore via *ostore_write*, deletion objects cause all objects with identifiers in a specific range to be deleted. As explained earlier in this section, BilbyFs selects object identifiers in a way that objects that must be deleted together have adjacent identifiers.

The range is specified by pattern-matching on the object identifier of the deletion object itself. To describe the pattern matching mechanism, we need to look at the different cases where the file system needs to delete objects from the store. There are three cases: (1) a file gets deleted, so the inode and all the data blocks must be deleted; (2) a directory entry must be removed, so the dentarr must be deleted if it is empty; (3) a file gets truncated, so all data blocks beyond the truncation offset must be deleted. Deletion objects carry a pattern to specify the range, which is stored in the deletion object's identifier. Note that deletion objects are not referenced by any file system object on flash, thus they do not need to be in the index, nor do they need a conventional object id.

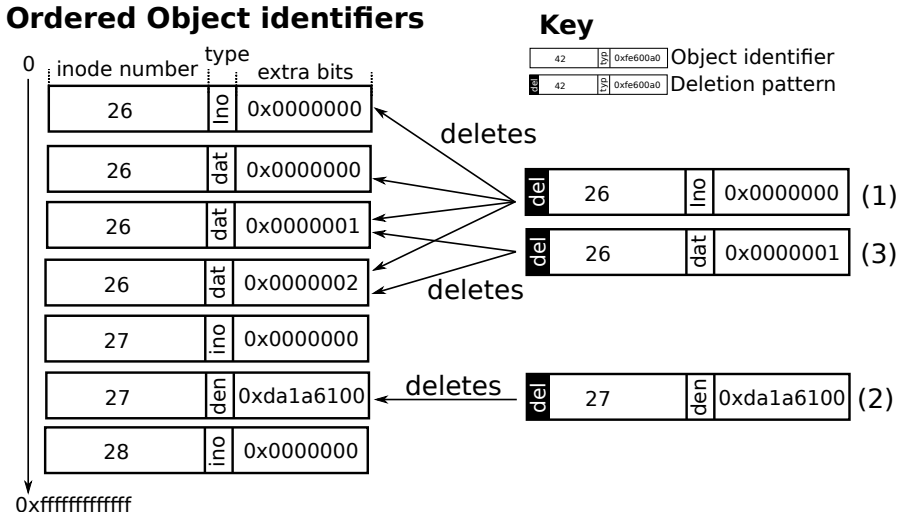


**Figure 3.17:** Specifying ranges of object identifiers with deletion patterns

To interpret the range from a deletion object's identifier, we pattern match on the type field (see Figure 3.17). If the type is inode (1), all the object identifiers that have the same inode number get deleted. For dentarr patterns (2), only the dentarr object with the exact identifier is deleted. When the pattern is of type data block (3), we use the identifier to delete all data blocks with identifiers with an offset greater or equal to the block offset stored in the pattern and with the same inode number.

The deletion mechanism iterates through the index (a red-black tree) and removes the entries that match the pattern. One slight complication is that FreeSpaceManager needs to keep track of the number of objects that become obsolete because of each deletion object. The reason we need to keep track of this number is to know when it is safe for the garbage collector to remove a deletion object from the log. It is only safe to do so, when all the objects that match the pattern have been completely erased from the log.

Otherwise, objects that have been deleted might reappear the next time the file system is mounted. This problem seems to be fundamental to deleting objects in a log-structured file system, Rumble et al. [2014] had to deal with similar issues when handling their *tombstone* objects. Tombstones are analogous to BilbyFs' deletion objects, and Rumble's solution to safely remove tombstones from the log is to store the address of the object being deleted inside the tombstone. When removing a tombstone, they need to check whether the object still exists in order to prevent its resurrection.

It seems unlikely that such a design would work better for us, since it would convert the bookkeeping memory overhead of our technique into a more complex on-medium representation of deletion objects. And garbage collection would have to read deletion objects off the flash to check whether they are obsolete or not.

**ObjectStore's implementation** When reading an object from the ObjectStore, we first inspect the index to find the address of the object in the log. If the object was recently written to the write buffer, we simply read it from the buffer, i.e. the write buffer behaves as a cache. Otherwise, the object is read from the flash by invoking a UBI operation to read flash pages.

As mentioned previously, the write buffer is structured as a log of transactions. When `ostore_write` is invoked, we first check whether the transaction passed as argument fits in the write buffer. If it does not, we synchronise the buffer using `ostore_sync` and we allocate a new eraseblock, i.e. the next segment of the log. Otherwise, the transaction is added to the write buffer.

We assign a unique number to each transaction (we increment a 64-bit integer), which serves two purposes. First, it is used to reorder transactions at mount time. Transactions numbers are always increasing within a segment, hence by comparing the first transaction numbers between segments, BilbyFs reconstructs the order in which segments were appended to the log. Second, transaction numbers are crucial for the deletion mechanism: a deletion object can only delete objects with a lower transaction number.

`ostore_write` serialises the transaction to the write buffer and updates the index with the address of new or modified objects. `ostore_write` also accounts for the additional dirty space by updating the FreeSpaceManager.

The index is implemented using a red-black tree that maps object identifiers to object addresses, where an address is a structure holding the eraseblock number, the offset and the length of the object. The FreeSpaceManager keeps a table where each entry records the amount of dirty space the corresponding eraseblock has, and a bitmap indicating

which eraseblocks are in used. Unused eraseblocks are in the erased state on flash, and in that sense they are "empty" and free to use.

The ObjectStore maintains the write buffer, which is of the size of an eraseblock, in memory. The write buffer is synchronised to flash as soon as a user application invokes the file system operation `sync` or `fsync`[3]. In BilbyFs, `sync` and `fsync` perform the same operation, they both flush to flash the contents of the write buffer. When `ostore_sync` is called and the write buffer is full, it is synchronised to flash and a new eraseblock is allocated to form the next segment. The number of the eraseblock used to store the contents of the write buffer on flash is recorded in the ObjectStore's state, we call it the *current eraseblock number*. When the write buffer is synchronised, it is a perfect image of the on-flash contents of the current eraseblock[4]. When it is not yet synchronised, the write buffer is ahead of the current eraseblock on flash.

The ObjectStore needs to allocate a new eraseblock when the write buffer is full and completely synchronised to flash. The ObjectStore does so by searching the FreeSpaceManager's bitmap for unused eraseblocks and updating the ObjectStore's state to record the new current eraseblock number.

Some extra care is taken when updating in-memory data structures to ensure that no error happens part way through the update that may otherwise leave the in-memory state inconsistent and prevent further use of the file system. In order to avoid memory allocation errors, BilbyFs ensures it pre-allocates all memory required when updating the Index and the FreeSpaceManager. But some errors are unavoidable, for example: UBI operations to write and erase the flash device can return an error and BilbyFs can not prevent this from occurring. Thus, BilbyFs orders updates in a way that these operations are either invoked when the in-memory changes can be undone, or are benign for the correct behaviour of further file system operations. In Chapter 6, we prove that the operations `sync()` and `iget()` maintain the file system invariant, regardless of whether an error occurred or not. This guarantees that the ObjectStore state remains consistent when any of these two operations is invoked.

### Serialisation and de-serialisation

BilbyFs manipulates on-medium data by reading flash blocks into a buffer in memory, de-serialising a file system object from the buffer, modifying it, and serialising the object

[3] It is fairly easy to add a periodic timer to synchronise BilbyFs' write buffer, ensuring that transactions in the buffer are automatically made persistent after a long enough period of system inactivity.

[4] Assuming no silent data corruption.

again. This is done by the Serial component (see Figure 3.11), which was designed to be verified in isolation. Although BilbyFs only updates data out-of-place, for the sake of simplicity the serialisation functions do not take advantage of this pattern.

The role of serialisation functions is to pack an in-memory data structure into a sequence of bytes (i.e. a buffer). De-serialisation functions do the opposite: they interpret a portion of a buffer and create an data structure that the file system can easily manipulate. The interface of this component satisfies a contract that describes how a serialisation function modifies a buffer, and how a de-serialisation function constructs an object from a buffer. The formal description of the behaviour and the correctness proof of these functions is presented in Appendix A.1.

On-medium data manipulation is a common source of file system implementation errors [Bangert and Zeldovich, 2014]. A method commonly used to avoid these errors consist of writing a formal description of the data format in a data description language [Bangert and Zeldovich, 2014; Fisher et al., 2006] to generate code automatically for serialising and de-serialising objects. In principle, this approach can provide strong guarantees that the serialisation and de-serialisation functions are consistent with each other. We chose not to use a data description language and to manually write and prove correct BilbyFs' serialisation and de-serialisation functions, leaving investigation of an appropriate data description language for future work. As explained in Chapter 6, using a data description language could also help improving the productivity verifying of file system code.

### UBI abstraction

Unsorted Block Images (UBI) is a volume management system for raw flash memories that supports NAND and NOR flash. UBI maps logical eraseblocks to physical ones, and transparently implements a wear levelling algorithm to maximise the lifetime of the flash by spreading out eraseblock erasure on the entire device. A logical eraseblock automatically gets mapped when the file system writes data to it, and it gets unmapped when it is erased. This means that the file system can continuously write and erase the same logical eraseblock of a UBI volume. UBI will map a different eraseblock every time it remaps the eraseblock until all the physical eraseblocks have been erased at least once. UBI provides an interface to read and write flash pages, and to erase eraseblocks. UBI's interface is as restrictive as the NAND flash one: the file system has to write pages in an eraseblock in sequential order, and it cannot overwrite a page until the whole logical erase block is erased. In other words, UBI only deals with wear levelling. The file system running on top still has to perform out-of-place updates and implement garbage collection.

UBI detects and handles bad eraseblocks (a common defect in NAND flash) by checksumming UBI's metadata. When a bad eraseblock is detected, UBI allocates another one from its pool of reserved physical eraseblock, and it automatically copies the valid data to the new eraseblock. As a result, flash file systems implemented on top of a UBI volume do not notice I/O errors that are automatically handled by UBI. However, UBI does not protect against silent data corruptions such as bit flips that might occur when storing data in eraseblocks. A file system implemented on top of UBI has to check the integrity of the data it stores on flash manually. BilbyFs handles this by doing a cyclic redundancy check (CRC) on each object it stores and reads on flash.

Since UBI provides an operation to check whether a logical eraseblock is mapped or not, BilbyFs can build a bitmap for free eraseblocks, that it uses to keep track of the list of free log segments.

### BilbyFs fault assumptions

As indicated earlier in this section, the UBI abstraction does not detect potential silent data corruptions affecting the data stored in eraseblocks. Hence, since such errors are common on flash media [Tseng et al., 2011], file systems implemented on top of UBI must manually ensure data integrity.

BilbyFs was designed to be robust against a realistic set of failures, including fail stop failures and crash corruptions. In particular, BilbyFs is able to recover from a crash, so long as the failure does not cause retroactive data corruption. When a crash occurs and interrupts a write operation, chances are the data written is corrupted. It has been observed [Tseng et al., 2011] that power failures can also cause the corruption of previously written adjacent flash pages. This is what we call retroactive data corruption and BilbyFs design does not handle such failures.

However, BilbyFs is designed to cope with (non-retroactive) crash failures even if such failures happens during the recovery process. BilbyFs' design for crash recovery is similar to UBIFS' [Hunter, 2008], and its implementation is greatly facilitated by the use of the UBI operation that replaces the contents of an eraseblock atomically. BilbyFs detects a crash by merely checking for a summary in each eraseblock. If an eraseblock contains data but no summary, the file system has not been unmounted cleanly and the recovery process must start.

BilbyFs detects data corruption by storing a cyclic redundancy check (CRC) for every object written to flash. CRCs ensure that each object in a transaction is intact, i.e. the checksum stored is the same as the one calculated on the data of the object. When

BilbyFs finds an invalid CRC, it signifies we encountered a corrupted object. When such an error occurs, the file system switches to read-only mode and must be remounted once the flash has been checked via an interactive off-line file system checker [McKusick et al., 1986]. File system checkers attempts to fix a corruption, either by looking for redundant copy of an object in the log or by asking the user for input via its interactive interface. Although it is certainly possible to do, we have not yet implemented such a tool for BilbyFs.

Not all the failure assumptions described in this section apply to our verification work, which relies on stronger assumptions of the UBI interface that do not specify potential data corruption and would require further work to be made fully realistic. Hence, we defer the formal treatment of the key concepts of file systems to Chapter 5 and Chapter 6.

### The garbage collector

The garbage collector's purpose is to recycle obsolete data in the log caused by out-of-place updates that are characteristic to log-structured file systems. When an operation overwrites an object, the ObjectStore merely writes the new object to the log, and the new object's higher transaction number ensures that it supersedes the old one. The same principle applies to deletion objects: they can only delete objects with a lower transaction number.

The garbage collector has to make sure that live objects do not get removed from the log. Live objects are non-obsolete ones, i.e. objects that have not been superseded or deleted by a deletion object.

The main complexity of the garbage collector comes from deletion objects. When the garbage collector removes a deletion object from the log, it needs to ensure that no obsolete objects covered by the deletion object will resurrect. The FreeSpaceManager stores enough information to know the number of objects in the log covered by each deletion object. The garbage collector makes use of this information to only remove deletion objects that no longer cover objects in the log.

The garbage collector relies on the ObjectStore's interface to read and write objects, and erase flash blocks. The garbage collector reads information from the FreeSpaceManager to find the log segment that has the most dirty space. It obtains a list of transactions using `ostore_scan_objs()`, and then writes a single transaction containing all the (live) objects that could not be recycled during the garbage collection run. This design makes our garbage collection algorithm relatively easy to implement as all the functions to manipulate objects are exported by the ObjectStore's interface. In particular, we reuse

`ostore_scan_objs()` which is used by the ObjectStore at mount time to parse transactions from an eraseblock when recovering from a crash. Our prototype C implementation of the garbage collector is about 70 source lines of C code.

### FreeSpaceManager: Free Space Manager

The FreeSpaceManager's functionality is straightforward. Its purpose is to keep track the following information: the amount of dirty space in each segment; which eraseblocks are in use; and the number of deleted objects still in the log, per deletion object.

To store the amount of dirty space in each eraseblock, the FreeSpaceManager keep an array of 16-bit unsigned integers in memory where each entry records the number of bytes occupied by obsolete object in the eraseblock. The array is indexed by eraseblock numbers, and is updated every time a deletion object is written to the ObjectStore.

We use an in-memory bitmap to keep track of whether an eraseblock is in use. Similarly to the dirty space, each bit corresponds to an eraseblock.

In order to keep track of the number of deleted objects remaining in the log for each deletion object, we use a red-black tree. The tree uses deletion patterns for keys, and values are stored in a structure with two fields: a 64-bit transaction number; and a 16-bit unsigned integer counter storing the number of objects that must be erased from the log before it is safe to erase the deletion object. Every time the garbage collector deletes an object from the log, it informs the FreeSpaceManager of the object identifier, and the FreeSpaceManager decrements the count of the deletion objects that match the object identifier. When the count is zero, the deletion object is obsolete and it is safe to remove the deletion object from the log.

### Eraseblock summaries

BilbyFs' in-memory index has the advantage of simplicity. For comparison storing the index out-of-place on flash is the main source of complexity in UBIFS. One obvious disadvantage is that the index has to be reconstructed at mount time by scanning the contents of the flash, and scanning each eraseblock is costly. To minimize the cost, BilbyFs borrows a technique from JFFS2 called *eraseblock summaries* [JFFS2-EBS; Ferenc].

In this technique, the end of each eraseblock stores a small amount of data that describes how to update the index and the FreeSpaceManager. Summaries contain enough information such that rebuilding the index can be done solely by reading summaries. More specifically, a summary is an array of *summary entries*, where each entry stores the following: an object identifier; an offset indicating where the object is located in the

segment; the length of the object; and bit indicating whether it is a deletion object. When the entry refers to a deletion object, the offset is used to store the number of objects it deletes, which are still present in the log. As explained earlier in this section, the garbage collector needs that information.

Reading a summary is much faster than reading the entire contents of an eraseblock, speeding up the mount operation of BilbyFs. This change makes BilbyFs suitable for flash devices of at least 1 GB.

### The ObjectStore abstraction

In this sub-section, we discuss the benefits of our ObjectStore abstraction. The ObjectStore provides a simple interface to read and write objects identified by object identifiers as input. The interface can easily be formally specified using a partial map from object identifier to object. This highlights one of the key modular design improvements in BilbyFs compared to UBIFS. Although UBIFS is much more complicated than BilbyFs because it stores its index on flash, our analysis suggests that its structural organisation could be made more modular without sacrificing performance. For instance, UBIFS' index interface is significantly more complicated than BilbyFs' because it allows multiple file system objects to share the same object identifier.

As seen in the previous section, directory entry identifiers cannot be unique due to the limited amount of information we can store in a fixed size identifier. As a result, UBIFS' index interface does not assume uniqueness of identifiers, which leads to a convoluted interface. The function to retrieve elements from the index is no longer uniform for any object in the index. For instance, a directory entry lookup in the index requires calling a specific function for directory entries that takes the name of the entry as argument to resolve potential hash collisions.

BilbyFs has a more modular design that does not allow multiple objects to share the same identifier. Instead the client has to resolve collisions on top of the abstraction, and BilbyFs' dentarr objects serve this purpose.

In BilbyFs the key file system logic is confined to the components running on top of the ObjectStore abstraction, which allows us to reason about it without having to worry about how objects are stored on flash. Similarly, the ObjectStore can be verified without having to worry about file system semantics.

## 3.4 Modular Design Evaluation

In this section we evaluate the performance of BilbyFs against UBIFS and JFFS2 in order to show that BilbyFs' modular design does not introduce unacceptable performance overhead. UBIFS and JFFS2 are the two most popular flash file systems available in the mainline Linux kernel tree. JFFS2 is older than UBIFS, and was designed for low capacity flash devices of type NAND and NOR. The main difference between JFFS2 and UBIFS is that the former keeps the index in memory, whereas UBIFS stores it on medium. As a result, UBIFS scales to larger flash devices and has a much faster mount time because, as opposed to JFFS2, it does not need to scan the medium to rebuild the index.

JFFS2's mount time was recently improved with eraseblock summaries [JFFS2-EBS]. As explained in the previous section, this feature consists of storing some data at the end of each eraseblock that summarises the index information of the log. Another limitation of JFFS2 is that it does not support write-back, i.e. writes are not buffered, they are issued synchronously to the flash device. An application using JFFS2 must wait for the flash device to acknowledge write commands every time it invokes a system call.

BilbyFs has the simplicity of JFFS2's design, with runtime performance on par with UBIFS and a mount time in between the two. Similarly to JFFS2, BilbyFs keeps the index in memory and it supports eraseblock summaries. However, as opposed to JFFS2, BilbyFs supports write-back. As a result it shows much better performance than JFFS2 on any workload that writes to the flash.

Our evaluation aims to demonstrate that BilbyFs' verifiable modular design does not inhibit performance when compared to mainstream flash file systems. BilbyFs code base is much simpler than UBIFS', for example we do not need to update the index on flash. This is an obvious limitation that needs to be addressed in future work for BilbyFs to be used comfortably with flash devices of several GB in size.

We ran our benchmarks on the MiraBox [Global-Scale-Technology] development board, which features a Marvell Armada 370 single-core 1.2 GHz ARMv7 processor and 1 GB of DDR3 memory. The NAND flash we used to run the benchmark has a capacity of 1 GB with 512 KB eraseblocks and 4 KB flash pages. We use Debian 6a distribution running Linux kernel 3.17 which is installed on a separate SD-card; the NAND flash is solely used for benchmarking the flash file systems. All benchmarks are run with the default size for the page cache which is maximum 20% of total memory. The Linux kernel starts flushing pages to the flash when the memory consumed by the page cache reaches 10% of the total memory. In order to achieve reproducible results and compare fairly between all file systems, each experiment starts from a freshly formatted medium and with cold caches.

To this end, we ran the command "echo 3 > /proc/sys/vm/drop_caches". This means that the benchmarks do not stimulate the file systems' garbage collector. File systems performance becomes less predictable when they start running out of space, especially because garbage collection can be delayed due to external events. Thus, in this evaluation we focus on workloads that do not trigger garbage collection. We also make sure that file system compression is disabled. Results show the average of a minimum three samples of data, and error bars show the standard deviation of the measurements.

## Throughput

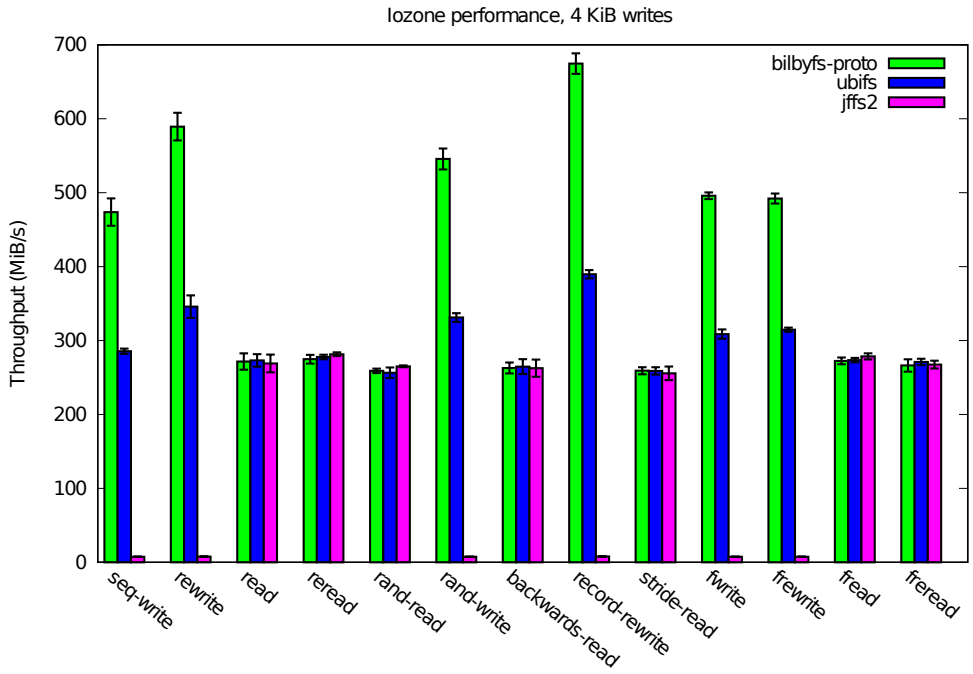


**Figure 3.18:** IOZone throughput benchmark, 4 KiB writes, 64 MB file

We use IOZone [Norcott and Capps] to measure data access throughput. We configured IOZone to work on a single file and with a record size of 4KiB.

To get an overview of the throughput comparison between Linux flash file systems, Figure 3.18 shows the results when running IOZone with maximum file size of 64MiB and the automatic configuration option '-A' which measures throughput in the following scenarios: sequential write, rewrite, read, reread, random read and write, backward read, record rewrite (rewriting a particular offset within a file), strided read (reading

sequentially non adjacent blocks), as well as access through the standard library functions fwrite and fread. In this benchmark, for each test IOZone measures the throughput in KiB per second by counting the time lapse between the first system call and the last, without calling `sync()`. As expected, we can see that JFFS2 is much slower on all write intensive benchmarks because it lacks write-back support. For read intensive tests, most file systems exhibit the same performance as the tests are I/O bound.

BilbyFs is faster or equivalent to UBIFS on all write tests. The throughput advantage of BilbyFs is due to two reasons. First, BilbyFs in-memory index reduces the number of metadata updates that need to be written to the flash device. UBIFS on-flash index must be updated out-of-place which incurs some extra flash operations when compared to BilbyFs. Second BilbyFs has a larger internal buffer where writes get batched in, whereas UBIFS uses the maximum unit of write supported by the flash device as size for its internal buffer. BilbyFs uses a larger buffer of the size of an eraseblock, thus achieving better metadata packing and more updates get buffered in memory. Because measurements do not include `sync()`, the last few writes of the benchmark are likely not flushed out to flash. This is true for UBIFS and more so for BilbyFs because its buffer is larger. We confirm this explanation with the next two diagrams which compare measurements that do not include the `sync()` call with the ones that do.

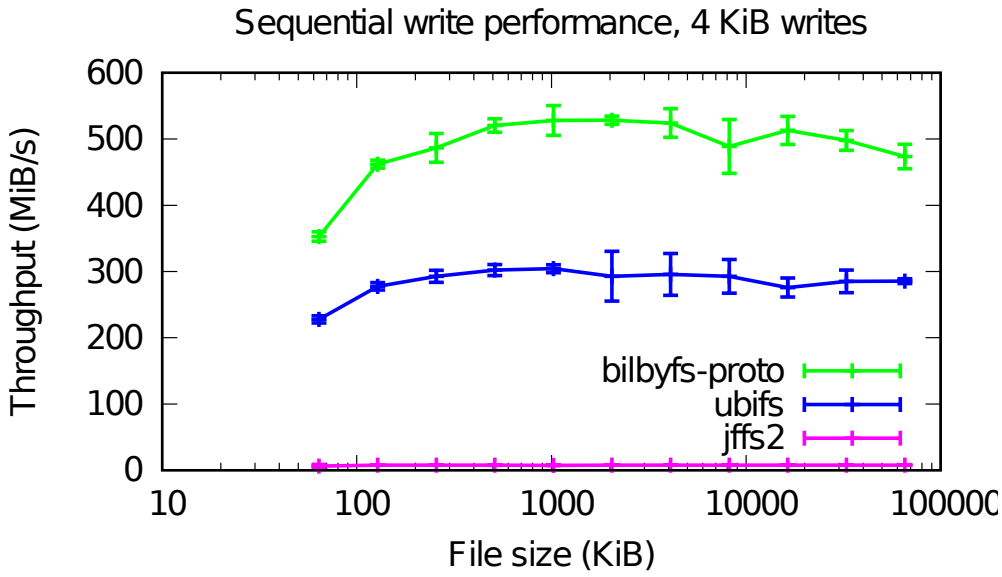


**Figure 3.19:** Sequential write performance, 4KiB writes

Figure 3.19 shows the throughput for sequential writes as we progressively increase

the size of the file to write from 64 KiB to 64 MiB. Again here IOZone measures the throughput by timing the `write()` system call, and when write back caching is enabled this does not necessarily mean that data was written to medium.

We observe that BilbyFs is faster than both UBIFS and JFFS2 all along the line. The difference of throughput is the lowest for 64 KiB files with BilbyFs showing a 55% increase in throughput. The highest throughput difference is shown for files of 2 MiB with 123% throughput increase. This difference is explained by the same reasons presented when describing Figure 3.18. When kernel profiling [Levon] the CPU load we find $25\% \pm 5$ pp. (percentage point) of CPU usage for both BilbyFs and UBIFS, whereas JFFS2, which exhibits lower throughput also has a lower CPU usage.

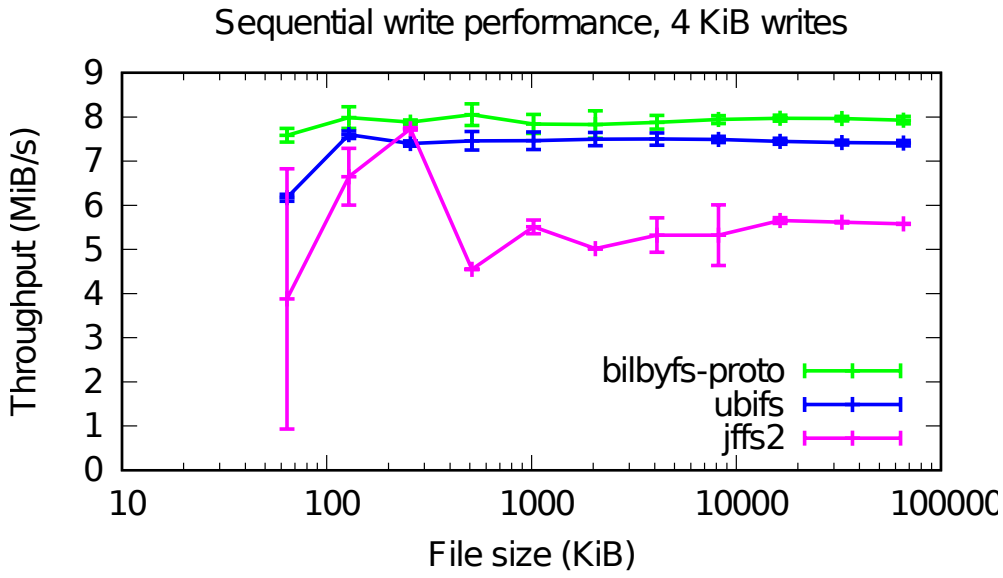


**Figure 3.20:** Sequential write performance, 4KiB writes with sync

To confirm that BilbyFs performance is comparable to other flash file systems when writing the flash, we run the same IOZone benchmark but with the extra option '-e' which forces `sync()` to be included in the throughput measurements. Figure 3.20 shows the results. The overhead of UBIFS index metadata updates is still visible, but BilbyFs only exhibits slightly more than 5% throughput increase compared to UBIFS on average. JFFS2 results fluctuate a lot more than the other two file systems and we have not been able to precisely identify the cause of this variation. When `sync()` is called, JFFS2 interacts with the page cache to flush the page mapping associated to the inode of the file `sync()` was called on. This process seems to sporadically trigger writes to the medium.

### Realistic workloads

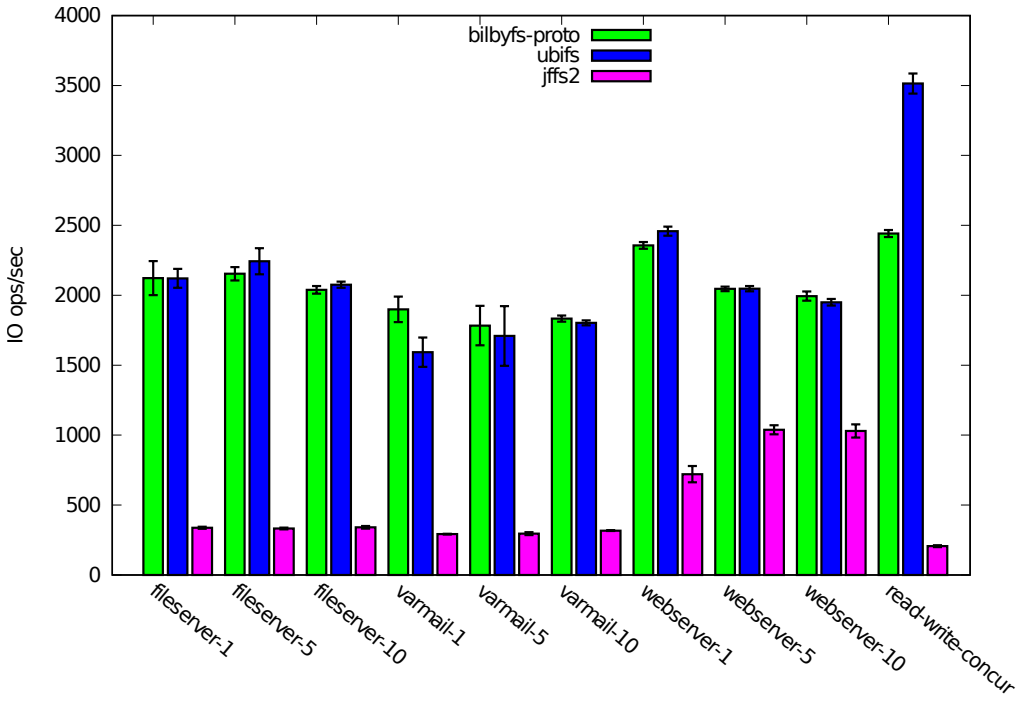


**Figure 3.21:** Filebench workloads

We use Filebench [Filebench] to evaluate BilbyFs against some of the pre-defined workload personalities that come with Filebench 1.4.9.1. Each experiment runs for two minutes and executes as many file system operations as it can during that time.

We first run Filebench with the file server workload (*fileserver*) where a (configurable) number of threads create files of 128 KiB. Each thread of the workload reads and writes data sequentially, and requests file's metadata using the `stat()` system call, before deleting the created files. The second workload is an email server (*varmail*) that has threads creating files and writing 16 KiB chunks of data, and flush the writes to medium by calling `sync()`. This workload stimulates the `sync()` code when dealing with small files. The third benchmark emulates a web server (*webserver*) and is a more read-intensive workload. The web server workload makes each thread read 10 files of 128 KiB, followed by appending 16 KiB of data to one file. The file append emulates the web page access logging mechanism as implemented by most web servers. Finally, the forth workload is *read-write-concur*, that we constructed in order to exhibit the limitations of our global

lock.

In order to better understand the limitations of BilbyFs' global lock we design a Filebench test read-write-concur that benefits from fine-grained concurrency in the file system. For instance, when the file system is blocked by an operation that reads from the flash while another thread attempts to perform several write operations, chances are the writes will be buffered in memory and will not require waiting for the flash device. In such scenario, the global lock will force the writer thread to wait for the currently running read operation to terminate, before starting to process the in-memory write operation.

We constructed such scenario using Filebench where a process creates 2 threads, one that consecutively reads several files, while the other simultaneously appends small chunks of data to a large file.

The results of our experiments are shown in Figure 3.21. The Y-axis is the number of file system operations executed per second (higher is better). We ran the experiment with 1, 5 and 10 threads running concurrently. The name of the workload is hyphened with number of threads each workload was configured with.

The results of BilbyFs and UBIFS are within 5% of each other for all workloads except for varmail-1 and read-write-concur. We notice that the lack of fine grained concurrency in BilbyFs does not make a significant difference in these settings: a single core machine and workloads that do not intensively read and write concurrently.

For the varmail-1 workload, BilbyFs performs approximately 200 operations per second more than UBIFS. This is explained by the same reasons we described previously.

The results of the read-write-concur workload on Figure 3.21 shows that indeed, in such setting BilbyFs suffers from its coarse grained locking implementation. BilbyFs performs 1,000 operations per second less than UBIFS, which accounts for around 30% less than UBIFS. However, as shown by JFFS2 performance with all write intensive benchmarks, the lack of asynchronous writes is a much more important performance bottleneck.

We analyse the CPU usage of the file systems while running Filebench with webserver workload with 1 thread using the OProfile [Levon] kernel profiler version 0.9.6. Figure 3.22 shows the CPU usage for all file systems. The diagram shows that UBIFS and BilbyFs spend about 30% of CPU time in memcpy and 13% in the standard library (libc). 11% and 4% of the samples collected for BilbyFs and UBIFS respectively were in the idling function `arch_cpu_idle`. By contrast, JFFS2 shows 65% of idle samples and a lot less operations per seconds as shown by the diagram on the right. The rest of the CPU is spent on other parts of the Linux kernel each of which accounts for less than 3% of CPU.

Finally, we run postmark [Katcher, 1997], a benchmark that emulates a busy mail

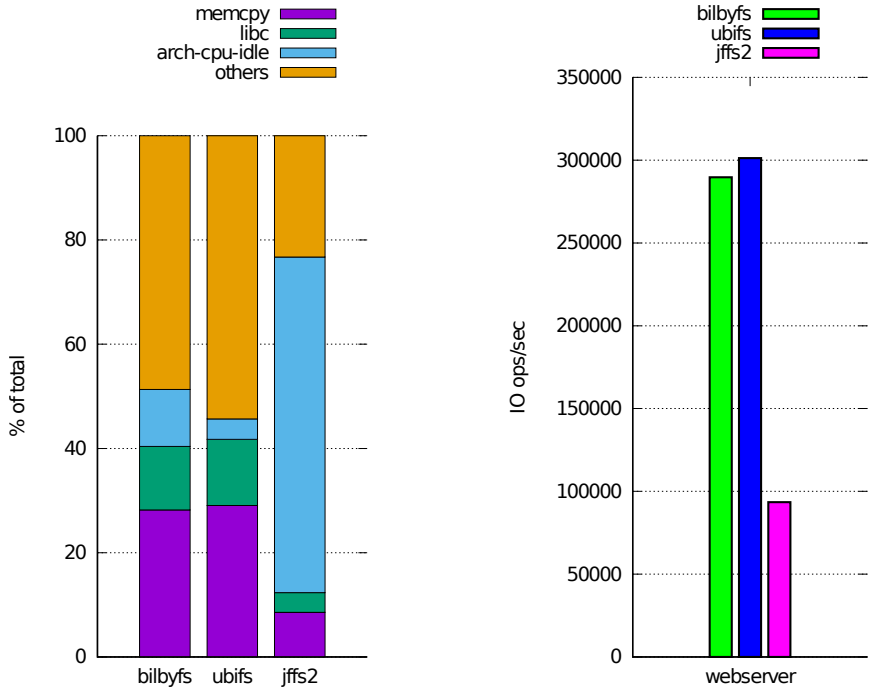


**Figure 3.22:** CPU usage for webserver workload, 1 thread

server by creating and deleting many small files. We configure the benchmark for 50,000 sets of size 10,000. Figure 3.23 shows that UBIFS outperforms BilbyFs by about 70 create operations per seconds, that is around 20% more. By contrast, the number of read operations per seconds for both UBIFS and BilbyFs is the same. However, the left hand side diagram shows that BilbyFs' function `sum_obj()` is a bottleneck. Almost 8% of the CPU is spent in this function. `sum_obj()` is the function that collects the indexing information to add to the summary. BilbyFs' summary, which is an unordered array of indexing entries, is kept in-memory, and is committed to flash when the write buffer is full. The present implementation of `sum_obj()` is quite naive, because every time a file system object is written to the ObjectStore, BilbyFs iterates through the array of summary entries to find and replace an existing entry. If no entry is found, `sum_ojb()` appends an entry at the end. The lookup algorithm is linear time in the size of the array. We could improve the performance of BilbyFs by choosing a data structure that provides faster lookups (e.g. a red-black tree) for the summary in memory.

Overall, this evaluation shows that BilbyFs performance is on par with UBIFS and, unsurprisingly, BilbyFs is much faster than JFFS2 on workloads that are write intensive.

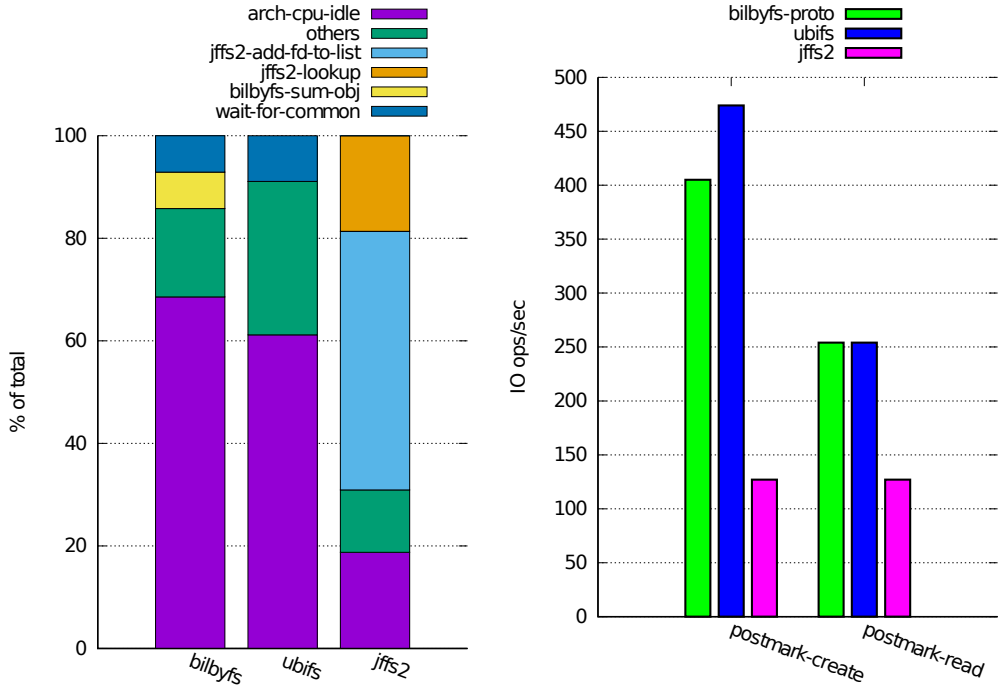


**Figure 3.23:** Performance and CPU usage for postmark benchmark

## Mount time

| FS image | Files | Directories | Size |
|---|---|---|---|
| archlinux | 24470 | 1857 | 486MB |
| raspbmc | 27734 | 4522 | 786MB |
| bodhi | 28059 | 3620 | 852MB |

**Figure 3.24:** Mount time measurement results

BilbyFs main weakness is its mount time. We pay for the simplicity of our design with an increase in mount time compared to UBIFS. Since the index is not stored on flash, it needs to be rebuilt at mount time. As explained earlier, BilbyFs borrows JFFS2's eraseblock summaries idea to reduce the mount time and scale to larger devices.

In order to evaluate BilbyFs' mount time, we selected a few pre-packaged file system images for the Raspberry Pi [Upton et al.]. We use them to populate the flash device under test. The list of packages and the number of files and directories they contain is presented in Figure 3.24.

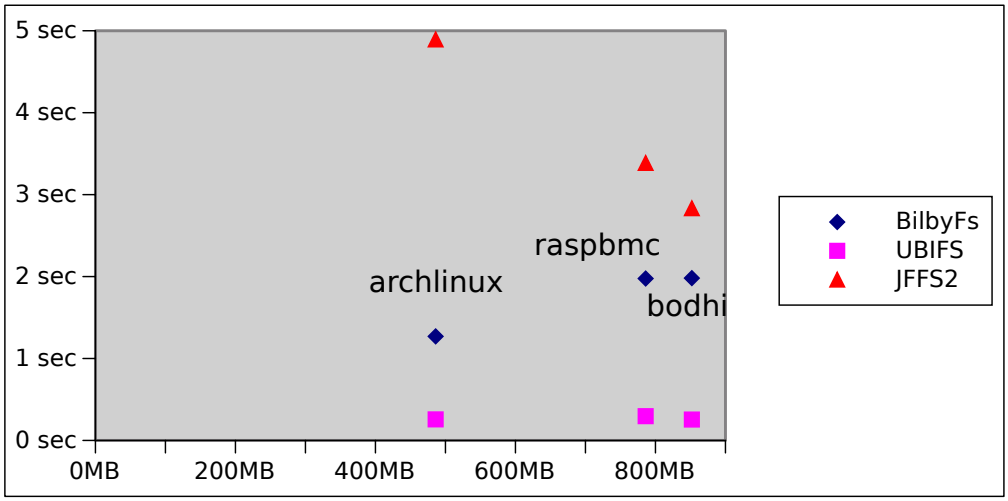


**Figure 3.25:** Mount time measurement results

Our experiment consists of copying the contents of each image to the flash, unmounting the file system, flushing all caches and timing the mount operation. The results are shown in Figure 3.25, the X-axis is the size in MB of the file system image being mounted. The Y-axis is the time (real time) it took to mount the file system in seconds (smaller is better). As expected UBIFS is much faster to mount than both BilbyFs and JFFS2.

Surprisingly, as the size of the file system image increases, JFFS2 mount time decreases. Our analysis revealed that JFFS2 has to read the out-of-band area of the first page of every eraseblock regardless of whether the eraseblock is empty or not. The out-of-band area is located at the beginning of each page of the flash and can be used to store file system metadata. This area is ignored by UBI since some NAND flash devices do not support it. When out-of-band area is available and summaries are enabled, JFFS2 mount implementation reads the out-of-band area of empty eraseblocks twice. Once to check whether the eraseblock is empty, this time JFFS2 reads the smallest I/O unit possible. If the eraseblock is empty, JFFS2 needs to read it again, but this time in its entirety to ensure that the area is completely clean. If it is not clean, JFFS2 schedules the eraseblock for erasure. As a consequence, JFFS2, which was designed for small flash devices, is faster to mount when the file system image produces fewer empty eraseblocks, and is slower the emptier the flash is.

## 3.5 Summary

Designing a verifiable file system demands several pragmatic trade-offs. In this chapter, we presented the key ingredients to make a file system verifiable, and illustrated them on BilbyFs design. To manage the complexity of verifying a file system, we restrict file system concurrency at the level of the VFS interface, and we decouple the core file system logic from Linux caching mechanisms. We designed BilbyFs modularly such that each component can be specified and reasoned about in isolation; we demonstrate this in Chapter 6. We evaluated the performance of BilbyFs' modular design, and we compared it against existing Linux flash file systems. Our evaluation shows that BilbyFs' modular design does not significantly degrade systems performance. Even though BilbyFs' design is quite simple, its C implementation is about 3,400 lines of code. Previous work showed that verifying such a code base is a labour-intensive task. For example, the seL4 project [Klein et al., 2014] verified 8,700 lines of C code with 12 person years of effort. In the next chapter, we present our approach to implementing the BilbyFs design in a way that makes verification more affordable.

# 4 Implementation

This chapter draws on work presented in the following papers:

- 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, and Gernot Heiser. Cogent: Verifying high-assurance file system implementations. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 175–188, Atlanta, GA, USA, April 2016. doi:10.1145/2872362.2872404;
- Liam O'Connor, Zilin Chen, Christine Rizkallah, Sidney Amani, Japheth Lim, Toby Murray, Yutaka Nagashima, Thomas Sewell, and Gerwin Klein. Refinement through restraint: Bringing down the cost of verification. In *International Conference on Functional Programming*, Nara, Japan, September 2016.

In Chapter 3, we explained how to design a file system modularly to facilitate reasoning about the implementation while achieving performance on par with existing Linux flash file systems. In this chapter, we present the domain specific language at the core of our methodology: COGENT. We implement file systems in COGENT, and COGENT's certifying compiler generates an efficient C implementation as well as a COGENT specification. The COGENT specification is a purely functional formal model of the source code in Isabelle/HOL [Nipkow et al., 2002]. The compiler also generates a mathematical machine-checked proof that the generated C code formally refines [de Roever and Engelhardt, 1998] the generated COGENT specification. This proof ensures memory and type-safety of the C code, which guarantees the absence of all memory bugs and which also rules out many error-handling bugs (e.g. memory leaks, use-after-free etc.). The study by Lu et al.

[2014] of 1800 file system bugs revealed that memory and error code defects account for about 20% of file system bugs. Many of these bugs are provably eliminated by COGENT's certifying compiler.

The syntax of the COGENT compiler's input program is very close to the generated COGENT specification in Isabelle/HOL. COGENT imposes a set of restrictions on programmers to ensure that it can generate efficient C code without the need for a language run-time and a garbage collector. Properties proved on COGENT specifications hold on the generated C code by the compiler-produced refinement theorem. Because COGENT is purely functional, reasoning about COGENT specifications in Isabelle/HOL is relatively straightforward, especially compared to reasoning about imperative code which actually requires cumbersome machinery like separation logic or the use of memory heaps [Reynolds, 2002; Tuch et al., 2007; Greenaway et al., 2012, 2014]. By implementing modular file systems in a purely functional language under these restrictions, we make them far easier to verify, increasing their verification productivity. The contribution of this chapter is to present a set of worked design patterns for implementing file systems in a linearly typed language such as COGENT and to show how we use these design patterns to implement BilbyFs.

COGENT avoids a garbage collector, unlike most existing high-level languages such as Java, Haskell and OCaml, by implementing a *linear type system* [Wadler, 1990]. Essentially, variables that are *linearly typed* must be used exactly once, and all non-linearly-typed values must be read-only. This ensures that aliases can exist only on read-only variables, which ensures that each linearly typed object is used exactly once and allows for destructive updates in the implementation. COGENT is domain specific in the sense that it is restricted to the domain of programs that do not require a lot of sharing (i.e. pointer aliasing). COGENT is not Turing-complete; it does not support loops, instead loops are implemented directly in C via the FFI as iterators over ADTs. COGENT relies on an external library of *abstract data types* (ADTs) implemented in C and accessed via its *foreign function interface* (FFI) to implement file system code that requires sharing.

I helped to design COGENT in collaboration with Liam O'Connor, Gabriele Keller, Zilin Chen, Alex Hixon and Toby Murray [O'Connor et al., 2016]. The original idea of using linear types to allow reasoning over purely functional specifications was mine, and was ultimately brought to fruition by Liam O'Connor who designed COGENT's type system and Zilin Chen who implemented its compiler. The automatic generation of theorems that shows that COGENT specifications are an accurate model of the the generated C code is the work of Christine Rizkallah, Japheth Lim, Yutaka Nagashima, Thomas Sewell, Zilin Chen, Liam O'Connor, Toby Murray, Gabriele Keller and Gerwin

Klein [Rizkallah et al., 2016].

We provided the file system expertise to the team by specifying the requirements for the language to enable implementing verifiable yet efficient file system code. We designed COGENT to leverage our observation that file system code does not require a lot of sharing aside from reusable ADTs such as red-black trees, linked lists, etc. By implementing BilbyFs in COGENT we confirmed that observation: sharing was only required in a few places that could easily be isolated and implemented using COGENT's FFI.

Linear type systems have sometimes proved too restrictive for general systems programming in the past [Smith et al., 2000; Walker and Morrisett, 2001; Aspinall and Hofmann, 2002]. In COGENT, however, the trade-off is different. We keep the linear types where they are most useful: for tracking memory allocation, error-handling, and higher-level control code. ADTs provide a safety-hatch for escaping the confines of the linear type system when necessary. In this chapter, we show that we can implement the core logic of a carefully designed file system with a linearly typed language, that the language enables design modularity, and that it greatly simplifies verification (as shown in Chapter 6).

In Section 4.1, we introduce COGENT, followed by an explanation of its FFI in Section 4.2. In Section 4.3, we present a few ADTs and their interfaces, and we distill a set of design pattern for designing ADTs in a linearly typed language. In Section 4.4, we present a function of BilbyFs to show how COGENT allows us to implement recurrent code idioms in file systems designed modularly. Finally, in Section 4.5 we evaluate the performance overhead of implementing BilbyFs in COGENT by comparing our C prototype implementation of BilbyFs to its COGENT counterpart. While the current COGENT compiler is a prototype, this evaluation sheds the light on practical utility of implementing file systems in purely functional language like COGENT. We also analyse the bugs we found when testing COGENT code, most of which can be provably eliminated with a functional correctness proof over the compiler-generated COGENT specifications, as presented in Chapter 5 and Chapter 6.

## 4.1 Cogent: language introduction

Our objective with COGENT was to design a language tailored for modular verification of file system code. We observed that file system code is composed of a lot of low-level data structure manipulations to parse and update data on storage. It is common for file systems to rely on a library of reusable data structures to implement file system

functionality [Lu et al., 2014], e.g. Linux's file systems share implementation for red-black trees and doubly linked lists.

COGENT allows direct manipulation of primitive types such as U8 (8 bit machine words), U16, U32, U64, Bool and Unit (denoted `()`, analogous to the C type `void`) as well as composite types like records, tuples and tagged unions. Tuples and records provide similar functionality; tuples are merely syntactic sugar for records with anonymous fields. Due to the restrictions of COGENT's linear type system, ADTs like arrays, trees and lists cannot be implemented directly in COGENT. As explained in Section 4.2, they have to be implemented in C via COGENT's FFI.

COGENT's main strength rests upon its linear type system which restricts pointer aliasing and provides memory and type safety. The basic idea behind linear types is not complicated: any object of linear type has to be used exactly once. This rule is statically checked at compile time, and programs that do not comply with the rule are rejected by the compiler. As a consequence, any linear value passed as argument to a function written in COGENT must be returned, unless the function internally frees it. We show an example demonstrating this in the next sub-section. In addition, a COGENT function that allocates a new value of linear type and returns it, will have an extra linear return type showing in its type signature. This way we guarantee the absence of memory leaks and use-after-free errors at compile time.

### Cogent example

Figure 4.1 shows a fragment of COGENT code that we use to familiarise the reader with the language and its linear type system. The first line declares the type `Obj`, a record with two fields: `id` of type unsigned 64-bit word (`U64`) and `data` of type `WordArray U8`, a type from our ADT library. By default COGENT records are *boxed*, meaning they are allocated on the heap with an explicit call to the memory allocator and are passed by reference (i.e. via a pointer). Hence, they must eventually be freed, and the linear type system ensures that this happens; code which may result in a memory leak triggers a type error. COGENT also supports *unboxed* records (passed by value) that by default are not linear, which means they may be accessed freely without the restrictions imposed on linear variables. Unboxed records are linear when one of their fields contains a linear value. Unboxed record declarations are prefixed by the symbol #.

Functions that are declared but not defined in COGENT are automatically assumed to be foreign functions, thus must be implemented in C. On line 2, we prototype a foreign function `newObj` that takes an `ExState` as argument, which is a reference to an

```
1  type Obj = {id:U64, data:WordArray U8}
2  newObj: ExState -> (ExState, <Success Obj| Error ()>)
3  freeObj: (ExState, Obj) -> ExState
4
5  bar: Obj -> (Obj, <Success () | Error ()>)
6
7  foo: (ExState, Obj) -> (ExState, <Success Obj | Error ()>)
8  foo (ex0, obj0) =
9    let (obj1,res) = bar obj0
10   in res
11   | Error () ->
12     let ex1 = freeObj (ex0, obj1)
13     in (ex1, Error ())
14   | Success () ->
15     (ex0, Success obj1)
```

**Figure 4.1:** COGENT's Linear type system by example.

ADT embodying the external environment of the file system, and returns a tuple of a new *ExState* and a tagged union with two possibilities, either Success with a variable of type Obj or Error. From the type signature, we can deduce that *newObj* allocates an object of type *Obj*. Since *Obj* is boxed and must be allocated on the heap, for *newObj* to be implementable in C, the type must allow the function to return an error when the memory allocation fails. On line 3, we prototype another foreign function *freeObj* which takes two arguments: an *ExState*; and an *Obj*, a reference to a COGENT record. *freeObj* only returns an *ExState*, which means the operation must free the object to satisfy the linearity requirements of the type-system.

*ExState* is necessary to enable us to model events external to the file system when we verify COGENT code. For instance, we use *ExState* to model memory allocation or the system's current time. Purely functional languages such as COGENT allows no side-effect, hence the external environment of the file system has to be passed explicitly. Without *ExState*, the Isabelle model of a function that allocates an object like *newObj* would either always return an error, or never do so. By contrast, with *ExState*, depending on the state of the external environment, a function like *newObj* will either return an error or an allocated object and a new external environment. By updating the external environment every time we call a function that depends on it, we allow the next call to return a different result.

Back to our code fragment in Figure 4.1, line 5 declares a function prototype *bar*

defined elsewhere that takes a variable of type `Obj` as parameter. The function returns a tuple of two values: `Obj` and a tagged union with two possible tags `Success` or `Error`. In either case, only the value `()` is returned.

Note that COGENT's structural typing allows us to interchangeably refer to a type via its name or its structure. This means that the COGENT compiler identifies the type Obj and {id:U64, data:WordArray U8} as being equal because they have identical structure[1].

On line 8, we define the function `foo`. From the definition of the type `Obj` on line 1, we can infer that the argument `obj0` is boxed, and thus linear. `foo` returns a tagged union with two possibilities, either `Success` with a variable of type `Obj` or `Error`. From the type signature of `foo`, we deduce that the function frees `obj0` when it returns an error, since the `Error` return value does not include an `Obj`.

On line 9, we use `let` to bind the tuple returned by the `bar` function to two variables: `obj1` and `res`. While `bar` is not implemented here, suppose it updates its argument (`obj0`) in-place, so that the linear variable of type `Obj` that it returns always refers to the same memory as the argument `Obj` it was passed. Thus `obj1` and `obj0` refer to the same object in memory. While we used different names for the two variables in this example to aid explanation, as a convention in future we will reuse the same name when referring to linear values that denote the same in-memory object. The linear type system ensures that at any given point in time at most one name can be used to refer to a particular object in memory, i.e. it ensures that aliases cannot exist to linear objects.

On line 9, the function `bar` returns a new reference `obj1` to the object and a tagged union `res` indicating whether it succeeded. On line 10, we use pattern matching to make a case distinction on `res`, to determine whether it was `Success` or `Error`. In order to ensure that all COGENT programs are total, all COGENT pattern matching must be exhaustive.

On line 11 and 14 we pattern match on the tagged union `res`: if `bar` returned an error, we have to free the object before constructing `foo`'s return value on line 13. In case of success, `foo` returns `Success` with reference to the object.

### Controlled sharing: let!

Linear types are effective at restricting memory aliasing, however, they can also be overly restrictive in certain circumstances. Consider a function `sizeofObj`, which returns some information about a linear object. It would have to return a new reference to the argument object it was passed and be of type: sizeofObj: Obj −> (U32, Obj) in order to allow

[1]Structural typing simplifies some aspects of the language but also makes generating COGENT specifications in Isabelle/HOL more involved, since Isabelle/HOL's type system is not structural. We explain how we bridge this difference in Section 6.2.

referring to the object again once *sizeofObj* has returned. To eliminate this overhead COGENT has a second binding form borrowed from Wadler [1990].

```
let w = expr1 !v in expr2
```

This binding is called let! due to the presence of the "!" preceding *v*. Here, the value of *expr1* is bound to *w* in *expr2*, as in a regular let-expression, except that inside *expr1* we may mention *v* multiple times as if it were non-linear provided that *v* is not modified (i.e. is only ever read inside *expr1*). We say that, inside *expr1*, *v* is *shareable* and read-only. We denote shareable types by using the *!* type modifier; therefore, we can write a function that takes a shareable *Obj* as argument like this:

```
sizeofObj :: Obj! -> U32
```

and call it as follows:

```
baz: Obj -> ...
...
  let size = sizeofObj obj !obj
  in baz obj
```

*sizeofObj* expects its argument to be shareable and read-only, i.e. a value obtained using *let!*. In this example, *obj* remains in scope after the *let!* expression because *sizeofObj* only reads from *obj*. *baz*, however, takes a linear *Obj* as indicated by its function signature fragment.

The expressiveness of COGENT's type system allows specifying that a function does not modify a linear variable and, as we will show in Section 4.5, the COGENT framework leverages the type information to prove that such variables are only read from. This feature is particularly useful for code that interacts with the storage layer, for instance the *ubi_leb_read* operation that reads part of a flash eraseblock in a buffer has the following type:

```
ubi_leb_read : #{ex:ExState, ubi_vol:UbiVol!,
                 ebnum:EbNum, buf: Buffer,
                 buf_offs:BufOffs, nb_bytes:U32} ->
        ((ExState, Buffer), <Success (), Error ErrCode>)
```

The function takes six arguments packed in an unboxed record: *ex* the external environment of the file system; *ubi_vol* is a reference to the UBI volume the file system is reads from and writes data to; *ebnum* is the eraseblock number indicating which eraseblock to read; *buf* is the buffer we want to read into; *buf_offs* is a byte offset in the buffer; and

*nb_bytes* is the number of bytes to be read. The key idea is that we can use the type system to show that *ubi_leb_read* does not modify the UBI volume *ubi_vol*. *ubi_vol* is passed as a shareable argument, thus it does not need to be returned and the function only needs to return the linear arguments, namely: *ex* and *buf*. When reasoning about COGENT code in Isabelle, it follows trivially that the UBI volume that *ubi_vol* refers to is unmodified; Isabelle infers this from the type of the COGENT-generated specification.

### The Take and Put operators

File system code typically maintains a large state, but each component of a modular implementation only affects a small portion of that state, namely the local state of the component. To access record fields in COGENT code, we first need to logically remove them in order to avoid introducing multiple references to the same variable, and then later, the fields are logically replaced in the record. This is easily modelled using linear types: the required parts of the data structure may be taken in isolation, altered, and then re-inserted.

Consider the type *Obj* declared above; we may take a field out of a record as follows:

```
let obj1 {data = obj1_data} = obj0 in ...
```

*obj0* is the initial object with all its fields. We take the field *data* which introduces two variables in scope: *obj1*, which is *obj0* with the field *data* taken; and *obj1_data*, which is the field *data* of obj0. The type system records which fields of a record have been taken, to ensure the absence of aliases. Otherwise one could break linearity by taking a field more than once and introduce multiple linear variables that refer to the same record field.

We can put *obj1_data* back into the data field of the *obj1* record as follows:

```
let obj2 = obj1 {data = obj1_data}
in ...
```

*obj2* now refers to the record with its *data* field replaced. The syntactic sugar {data} may be used in take and put expressions as shorthand for {data=data} when the record field and the variable have the same name.

By providing methods to decompose the state at arbitrary granularity, COGENT enables verification engineers to benefit from the programmer's state decomposition when reasoning formally about file system functionality. This is a key aspect of our approach, and is required to fully benefit from the design modularity of the file system. We further discuss the decomposition of file system state in Section 4.4 as it is an important factor

in the increase in verification productivity afforded by our approach, which we report on in Chapter 6.

## 4.2 Cogent Foreign function interface

COGENT is intentionally restricted to simplify formal reasoning while ensuring type and memory safety in the absence of garbage collection. However, there are legitimate reasons to temporarily bypass COGENT's restrictions, including implementing data structures that require memory aliasing to be efficient or loops. To lift COGENT's restrictions, we invoke C code from COGENT via its *foreign function interface* (FFI). In order to conserve the strong correctness guarantees provided by COGENT, foreign functions have to be proved correct manually. These proofs involve showing that the C code does not break COGENT's type system assumptions about memory and type safety. For instance, foreign functions must not introduce memory aliases that are accessible from COGENT code. In the rest of this section, we present an example of a loop iterator declared and implemented via COGENT's FFI.

### Declaring a simple loop iterator

All COGENT loop iterators are declared using its FFI. COGENT functions that are declared but do not have a body are assumed to be foreign functions. For instance, the following code fragment declares the `seq32` iterator:

```
seq32: all (acc,obsv,brk). Seq32P acc obsv brk -> LoopResult acc brk
```

The types of the parameters for `seq32` are defined in Figure 4.2. seq32 is a polymorphic function that allows iterating over a range of values and storing the result of the loop in an accumulator. Since COGENT is a purely functional language, functions cannot have side effects, thus seq32 loops must return the effect of the loop in an accumulator. The `all` keyword indicates that the function takes polymorphic type parameters. The polymorphic types `acc, obsv, brk` are parametrised for each loop. `seq32`'s argument is of type `Seq32P` which is an unboxed record that defines the range, `frm` to `to`; the increment `step`; and a function pointer, `f`, to the body of the loop (see Figure 4.2). Other fields include the initial value for the accumulator, and the observable state which corresponds to the read-only part of the state accessible to the loop body.

The accumulator type `acc` is used to pass the part of the state that can be modified by the loop. The observable part of the state, which is shareable so that the type system

```
1  -- Type of seq32 parameters
2  type Seq32P acc obsv brk = #{
3    frm: U32,
4    to: U32,
5    step: U32,
6    f: Seq32F acc obsv brk, -- loop body
7    acc: acc,   -- accumulator
8    obsv: obsv! -- observables
9  }
10
11 type LoopResult acc brk = <Iterate acc | Break brk>
12
13 -- Type of seq32 loop body
14 type Seq32F acc obsv brk = Seq32FP acc obsv ->
15                             LoopResult acc brk
16
17 -- Type of seq32 loop body parameters
18 type Seq32FP acc obsv = #{
19   acc: acc,
20   obsv: obsv!,
21   idx: U32
22 }
```

**Figure 4.2:** seq32 types

ensures it cannot be modified, is passed via the type `obsv`. Finally, the type `brk` is used to carry the value returned when breaking out of the loop.

The body of the loop is defined by the type `Seq32F`, which is the type of a COGENT function that takes as input an unboxed record `seq32FP` declared on lines 18–22 of Figure 4.2. `seq32FP` is composed of three fields: the accumulator `acc`, the observable part of the state `obsv`, and the loop counter `idx`. The loop body function returns a tagged union `LoopResult` defined as follow:

```
type LoopResult acc brk = <Iterate acc | Break brk>
```

`LoopResult` is a parametric polymorphic type that carries the result of a loop iteration. The two constructors of this tagged union type allow the loop body to indicate that an iteration should either continue with accumulator `a` (Iterate a) or that the loop should terminate early with result `b` (Break b), where `a` and `b` are of the type parameters `acc` and `brk` respectively.

```
1  check_flash_is_empty: (ExState, MountState!, UbiVol!) ->
2            (ExState, <Success Bool|Error ErrCode>)
3  check_flash_is_empty(ex, mount_st, ubi_vol) =
4    let (ex, r) = seq32[ExState, UbiVol, ErrCode]
5                    #{frm=0,
6                      to=mount_st.super.nb_eb,
7                      step=1,
8                      f=check_flash_empty_loop,
9                      acc=ex, obsv=ubi_vol}
10   in r
11   | Break err ->
12     if err == eNotEmpty then
13       (ex, Success False)
14     else
15       (ex, Error err)
16   | Iterate () ->
17     (ex, Success True)
18
19 check_flash_empty_loop: #{acc:(ExState), obsv:(UbiVol!), idx:U32} ->
20           (ExState, LoopResult () ErrCode)
21 check_flash_empty_loop #{acc=ex, obsv=ubi_vol, idx=i} =
22   let (ex, r) = ubi_is_mapped #{ex, ubi_vol, ebnum=i}
23   in r
24   | Error err ->
25     (ex, Break err)
26   | Success is_mapped ->
27     if is_mapped then
28       (ex, Break eNotEmpty)
29     else
30       (ex, Iterate ())
```

**Figure 4.3:** seq32 example

### Loop example in Cogent

In the previous sub-section we have seen how to declare an loop iterator in COGENT. Here, we show how to use such loop iterators in file system code, before explaining how implement them in C using COGENT's FFI in the next sub-section.

Figure 4.3 shows BilbyFs' function for checking whether the flash device is empty, which makes used of the *seq32* iterator. As described in Section 3.3, UBI is a volume management system for flash memories that provides a mapping from logical eraseblock

addresses to physical ones to transparently implement a wear-levelling algorithm. UBI automatically maps an eraseblock when the file system writes to the UBI volume. When no logical eraseblock address is mapped to a physical address, it means that the UBI volume has never been written to and is empty.

We check for emptiness by scanning all logical eraseblock addresses, from 0 to `mount_st.super.nb_eb` which corresponds to the number of logical eraseblocks present on the UBI volume. `check_flash_empty` takes three arguments: `ex` is external environment of the file system; `mount_st` is the read-only part of the state initialised at mount time; `ubi_vol` is a reference to the UBI volume. On lines 4–9, we invoke the loop iterator `seq32` incrementing by one (i.e. step=1) from 0 to `nb_eb`. We pass the loop body `check_flash_empty_loop` as a callback which gets called for every iteration of the loop. `ex` may be modified by the loop body, so it is passed as the accumulator (i.e. `acc`) which is also returned by the loop body. The UBI volume reference `ubi_vol` is passed as a shareable variable (i.e. read-only) in the observable parameters. We make the most of COGENT type system to ensure that the loop body does not modify the UBI volume. As explained in Section 4.1, by passing `ubi_vol` as a shareable argument, we do not need to return it. The loop can either terminate with `Iterate ()`, meaning the last iteration of the loop returned `Iterate ()`, or it can terminate with `Break err`, meaning that one of the iterations interrupted the loop and returned `Break err`.

The loop body definition on lines 21–30 simply calls `ubi_is_mapped`; if it returns an error (line 24) we break out of the loop by constructing and returning a tuple made of `ex` and a tagged union `Break` with the error code. When `ubi_is_mapped` succeeds (line 26), we only continue iterating if the logical eraseblock being tested is not mapped. In this case, we return a tuple including the accumulator and the tagged union `Iterate ()`. Otherwise, if the `ubi_is_mapped` indicates that the logical eraseblock is mapped, we break out of the loop with the special error code `eNotEmpty`. Note that in `check_flash_is_empty` on line 12, we check if value of the error code is `eNotEmpty` to decide whether the loop succeeded or not.

### Antiquoted-C implementation of seq32

Recall that we implement ADTs, allocation functions and loop iterators directly in C. To facilitate the interactions between COGENT and C, the COGENT compiler supports *antiquoted-C*, a small template language designed to access COGENT types and call COGENT functions from C. The COGENT compiler generates monomorphised C from the polymorphic antiquoted-C templates.

```
1  static inline $ty:(acc, <Iterate ()|Break rbrk>)
2    $id:seq32($ty:(Seq32P acc obsv rbrk) args)
3  {
4      u32 i;
5      $ty:(acc, <Iterate ()|Break rbrk>) ret;
6      $ty:(Seq32FP acc obsv rbrk) fargs;
7
8      ret.p2.tag = TAG_ENUM_Iterate;
9      ret.p1 = args.acc;
10
11     if (!args.step)
12         return ret;
13     fargs.acc = args.acc;
14     fargs.obsv = args.obsv;
15     for (i = args.frm; i < args.to; i += args.step) {
16         fargs.idx = i;
17         ret = (($spec:(Seq32F acc obsv rbrk))args.f)(fargs);
18         if (ret.p2.tag == TAG_ENUM_Break)
19             return ret;
20         else
21             fargs.acc = ret.p1;
22     }
23     return ret;
24 }
```

**Figure 4.4:** Antiquoted-C seq32 implementation

Figure 4.4 shows the antiquoted-C implementation of the seq32 iterator. We refer to the C type that implements the COGENT type *TYPE* with *$ty:(TYPE)* and we refer to the C identifier for a COGENT function *FUNC* with *$id:FUNC*. *($spec:(FTYPE) f)(FARGS)* denotes calling a COGENT function *f* whose type is *FTYPE* passing arguments *FARGS*. COGENT n-tuples are implemented as a C data structure with *n* fields named p1 to p*n*. Records are implemented as C data structures with field names matching the record fields. All COGENT functions take one argument, but the argument may be of tuple type. The COGENT type system assumes that the execution of *seq32* is well defined and terminates whenever its arguments are well typed under COGENT's type system. On lines 11–12 we perform a sanity check on the *step* parameter of the loop. On lines 13–14, we initialise the loop-body arguments. On line 15 we initiate the for-loop that iterates from *args.frm* to *args.to* by adding *args.step* to the loop counter at each iteration. On lines 16–17, we initialise the last loop body argument *args.idx* with the current loop counter *i* and

we call the loop body passed in `args.f` using the `$spec` antiquoted-C directive. On the next line, we check whether the loop body broke out of the loop by returning `Break`. The COGENT compiler implements tagged unions by storing the tagged union's tag in the field `tag`. Each tag `t` is identified in C via the value `TAG_ENUM_t`, thus `TAG_ENUM_Break` identifies the COGENT tag `Break` of the `LoopResult` tagged union (see Figure 4.4).

## 4.3 Abstract data types interfaces

This chapter demonstrates the feasibility of implementing a the core logic of a file system in a linearly typed language. The main source of difficulty presents itself in the design of ADT interfaces that integrate with and respect the restrictions of COGENT's linear type system. In particular, foreign functions must be semantically well defined for all type correct COGENT arguments. They must also avoid creating COGENT-visible aliases. This way, any correctly typed program is guaranteed to follow COGENT's well defined semantics.

Our ADT library implements two kind of polymorphic arrays. The `Array` type is used to store boxed objects, such as linear values. COGENT also supports a more specialised type of arrays, called `WordArray`, that is used to more conveniently store non-linear values such as machine words[2]. We discuss the difference between `Array`s and `WordArray`s later in this section.

Boxed objects need to be allocated on the heap and are of linear type so their use has to comply with the linear type system. Hence, we have to proceed with care when designing an interface for manipulating arrays in COGENT, imagine we declare the following operation that takes an array and an index of type U32, and returns the element of the array at this index:

```
array_get:all(a). (Array a, U32) -> a
```

This operation cannot be implemented in C such that it always respects the linear type system while having well defined C semantics. What should the C code do if the element requested does not exist in the array? We could allocate an object and return it, but allocating memory in the kernel is never guaranteed to succeed. Just as important, however, consider what happens to the array once the element is returned? The returned element must be logically taken from the array (akin to COGENT's take operation on records) to prevent further invocations from introducing multiple references to the same object, which would otherwise break the linear type system. Logically removing the

[2] `WordArray` is a misnomer since it can be used to store more than merely words. `WordArray`s can be parameterised with any non-linear type.

object from the array also means that a program that only wants to read the object will have to logically put it back in afterwards. Such an interface is clearly cumbersome both for expressing file system logic but also for verifying it: a program that only needs to temporarily read an element from an array will have to prove that the array modifications leave the array visibly unchanged.

Fortunately, we can leverage COGENT's type system to design ADT interfaces that avoid these problems. The idea is to design a library of functions to manipulate ADTs using combinator-like operations such that part of the data structure can be logically taken in isolation, inspected or modified by the combinator, and then logically re-inserted. We use the `Array` ADT to show an example of such interface in the rest of this section.

Our `Array` ADT library exports operations to read, remove and replace boxed objects for an array. Instances of `Array a` are implemented in C as an array of pointer to COGENT objects of type `a`. Each slot of the array either contains a pointer to a valid linear object in memory or `NULL` to denote an empty slot which arises when removing an element from the array for example. The operation to read an element of an `Array` is called `array_use_value`.

```
1  type ArrayUseValueF a acc obsv = #{
2    elem:a!,
3    acc:acc,
4    obsv:obsv!
5  } -> acc
6  type ArrayUseValueP a acc obsv = #{
7    arr: (Array a)!,
8    idx: U32,
9    f: ArrayUseValueF a acc obsv,
10   acc: acc,
11   obsv: obsv!
12 }
13 array_use_value: all(a, acc, obsv).
14   ArrayUseValueP a acc obsv -> acc
```

Conceptually, `array_use_value` allows temporarily taking in isolation an element of the array as shareable (i.e. read-only). `array_use_value` takes five parameters packed in an unboxed record (denoted by the # on line 6): the array, the index of the element to read from, a combinator callback function that will operate on the element if it exists, an accumulator that can be used to store the result of the callback, and the state observable

```
1  static inline $ty:(acc)
2    $id:array_use_value($ty:(ArrayUseValueP a acc obsv) args)
3  {
4      if (args.idx < args.arr->len) {
5          $ty:(#{elem:a!, acc:acc, obsv:obsv!}) fargs;
6
7          fargs.elem = args.arr->values[args.idx];
8          if (!fargs.elem)
9              return args.acc;
10         fargs.acc = args.acc;
11         fargs.obsv = args.obsv;
12         return
13         (($spec:(ArrayUseValueF a acc obsv)) args.f)(fargs);
14     }
15     return args.acc;
16 }
```

**Figure 4.5:** Antiquoted-C implementation of `array_use_value`.

from the callback. The callback is of type `ArrayUseValueF`, the parameters include the shareable element of the array.

We note that `array_use_value` does nothing if the index passed as argument is out-of-bounds, or if the index refers to an empty slot. The operation is easy to implement such that any correctly typed input values leads to well behaved C code.

The antiquoted C implementation of `array_use_value` is in Figure 4.5. The function checks whether the index passed as argument is in bound on line 4, and if there exists an element at this index on line 8. If so, we pack the arguments of the callback on lines 7–11 and calls it on lines 12–13. If the element referenced by the index does not exist, `array_use_value` merely returns the accumulator unchanged on lines 8–9. The function assumes that the arguments are well-typed, for instance the COGENT type checker guarantees that `args.arr` is a valid pointer to a COGENT `Array a`.

### Cogent array types

The interface for `Array` has to be designed in a way that it never introduces two references to the same linear object. On the other hand, for `WordArray`s (which, recall, hold unboxed non-linear values) there is no such restriction, because elements are non-linear, so they may be freely duplicated. Such freedom makes the interface for `WordArray`s more convenient to use than `Array`s. Moreover, elements of a `WordArray` do not need to be freed individually

since they are unboxed. By contrast, elements of an `Array` are boxed and must be freed one at a time. We illustrate the simplicity of the `WordArray` interface with the following example:

wordarray_get: all(a :< DSE). ((WordArray a)!, U32) -> a

It reads an element of a `Wordarray` at a given index. Before we dive into a description of `wordarray_get`, we briefly describe the type kinding rules supported by COGENT's type system. A kind is a type specifier that can put restrictions on a the type parameter of a polymorphic type. Polymorphic types can be tagged by three different kinds: $D$ for *discardable*, $S$ for *shareable* and $E$ for *escapable*. Discardable means that the type of the variable does not force the variable to be used (as opposed to linear variables that have to be used exactly once). Shareable type means that the variable can appear multiple times without breaking linearity (such as let! bindings). Finally, escapable means that when the variable is bound by a let! statement, it can escape the scope of the let! statement (linear variables cannot).

In the type signature of `wordarray_get`, the kinding constraint `a :< DSE` ensures that the type parameter `a` (the array element type) is not linear. COGENT's compiler will thus reject programs that attempt to instantiate a `WordArray` with a linear type. `wordarray_get` takes a `WordArray` and an index of type U32 as argument. It is implemented as follows:

```
1 $ty:a $id:wordarray_get ($ty:(((WordArray a)!, U32)) args)
2 {
3         if (args.p2 >= args.p1->len)
4                 return args.p1->values[0];
5         return args.p1->values[args.p2];
6 }
```

Again, here we cannot assume anything about arguments of the function beyond type correctness. A well typed COGENT program can pass an index that is out of bounds. This should not cause an invalid memory access, hence we check whether the index argument, namely args.p2, is in-bounds, and we return the first element of the WordArray if it is not. Note that `wordarray_create`, the operation that allocates a WordArray, returns an error when the requested length is zero. Hence, all `WordArray` ADTs always have at least one element.

| Array | |
|---|---|
| Function name | Description |
| array_create | create an array of specific size |
| array_free | free an array |
| array_use_value | temporary access an element read-only |
| array_remove | remove an element from the array |
| array_modify | modify an element of the array |
| array_length | returns the length of the array |
| array_map | iterate over all elements of the array with read/write access |
| array_map_ex | same as array_map but only iterate through existing elements |
| array_map_no_break | map all elements, the loop body cannot break out of the loop |
| array_filter | remove all elements that match the filtering callback |
| array_fold | loop over elements of the array with read access only |
| array_fold_no_break | same as above but cannot break out of the loop |
| WordArray | |
| Function name | Description |
| wordarray_create | create an array of machine words |
| wordarray_free | free an array of words |
| wordarray_get | reads the data at a given index |
| wordarray_modify | modify the data a given index |
| wordarray_length | returns the length of the word array |
| wordarray_map | iterate over range of the word array, each iteration can update the element |
| wordarray_map_no_break | same as wordarray_map but cannot break out of the loop |
| wordarray_fold | iterate over a range of the word array, each iteration has read-only access to the element |
| wordarray_fold_no_break | same as wordarray_fold but cannot break out of the loop |

**Figure 4.6:** Arrays vs WordArray interfaces

**Comparison of array interfaces** Figure 4.6 shows the list of `Array` and `WordArray` ADT operations [3]. The most striking difference between the two interfaces is how elements are read from the array. Arrays have two operations to access an element at a specific index: `array_use_value` and `array_remove`, whereas WordArrays have only `wordarray_get`. We described `array_use_value` at length earlier in this section, however we did not discuss `array_remove`. `array_remove` updates the slot of the element to NULL in the array and returns the element. If the index is not in-bounds or refers to an empty slot, `array_remove` does nothing. Since its slot in the array is set to NULL, the element can only be returned once, satisfying the linearity requirements of the type system.

Both interfaces provide iterators to traverses elements of the array. The map variants (whose names are of the form `*_map*`) allow iterating through array elements while potentially modifying them, whereas fold iterators only provide read-only access to the elements being iterated over. All iterators provide a no-break variant (e.g. `array_map_no_break`) for loops that cannot be interrupted.

The Array interface also has an operation `array_filter` that allows removing several elements at once from an array. `array_filter` takes a callback function using the same mechanism we use for the loop body of the `seq32` iterator. The callback may remove an element by freeing it or leave it in place by simply returning it. No such operation is needed for `WordArray`s since all elements are allocated in batch at creation, thus only the `WordArray` itself must to be freed, not every single elements of it.

### Red-black tree interface

A red-black tree [Cormen et al., 2009] is a self-balancing in-memory tree data structure commonly used in systems programming. For instance, red-black trees are used in several parts of the Linux kernel including: in the anticipatory, deadline, and CFQ I/O schedulers to track requests; in the ext3 file system to track directory entries; in the memory management sub-system to track virtual memory areas; and many other places [Corbet, 2006]. The functionality of a red-black tree is easily modelled using a key-value partial map with a well defined ordering on keys. This provides a convenient abstraction to reason about code that manipulates the tree in COGENT-generated specifications.

File system code may have critical I/O paths where failures such as memory allocation errors are not allowed, or they would otherwise leave the in-memory data structures of the file system in inconsistent state. To avoid allocation failures in critical paths, file

[3] The complete interface, including the types, is available in the COGENT repository at https://github.com/NICTA/cogent/tree/master/impl/libgum/array.cogent and https://github.com/NICTA/cogent/tree/master/impl/libgum/wordarray.cogent

| Red-black tree | |
|---|---|
| Function name | Description |
| rbtnode_create | allocate a node for the tree |
| rbtnode_free | free a node of the tree |
| rbtnode_get_key | return key of a node |
| rbtnode_put_key | set new key for a node |
| rbtnode_get_val | return value stored in a node |
| rbtnode_put_val | set new value for a node |
| rbt_create | allocate a red-black tree |
| rbt_free | free a red-black tree |
| rbt_modify | update node for a given key (existing or not) |
| rbt_get_value | return value for a given key |
| rbt_iterate | iterate over a range of keys |
| rbt_iterate_no_break | same as rbt_iterate but cannot break out of loop |
| rbt_filter | iterate over range of tree node and conditionally remove nodes that satisfy a condition |
| rbt_cond_erase | same as rbt_filter but for a single key/node |
| rbt_next | find the next node in the tree given a key |

**Figure 4.7:** Red-black tree interface

systems may pre-allocate data structures. Thus a red-black tree interface for in-kernel file systems must provide means to pre-allocate nodes such that the insertion of a node can be guaranteed regardless of the state of the system's memory when the insertion is undertaken. Hence, both in C and in COGENT, the interface of red-black trees node allocation and insertion are decoupled.

Another particularity of kernel code that makes use of red-black trees is that for performance reasons, searching the tree is often done in a ad-hoc manner. This is especially true in the Linux kernel, where each client of the red-black tree library implements its own iterator on the tree[4], enabling clients to perform modifications to the tree while minimising the number time the tree is traversed, for instance.

Designing a red-black tree ADT interface for COGENT, which allows us to write COGENT code that provides the same functionality and that is asymptotically equivalent to the equivalent code in C, attests of the expressiveness of COGENT's FFI. Similarly to the `Array` interface, red-black tree operations to access the tree cannot merely return a node as linear variable because the type-system would not be able to guarantee linearity. Hence, we design an efficient combinator-like interface to access our red-black tree.

In Figure 4.7, all operations prefixed with `rbtnode_` create, free and manipulate

[4] https://www.kernel.org/doc/Documentation/rbtree.txt

red-black tree nodes [5]. Operations prefixed with `rbt_` manipulate the tree directly. By decoupling node allocation from tree manipulation, we can guarantee that the operation `rbt_modify` will always succeed, for instance. This happens to be critical in BilbyFs, when the ObjectStore has successfully written to the flash and must update several nodes of the red-black tree used for the in-memory index, a failure would lead to an inconsistency between on-flash data and in-memory index, and would not be recoverable without remounting the file system.

Apart from `rbt_next`, other operations are analogous to the Array ADT interface. `rbt_next` takes a tree and a key as argument, and will return the key of next adjacent node in the tree structure. If no such node exists, the operation returns a constant key value known to be invalid.

The interface choice for the `rbt_modify` function was crucial. It is common for file systems to search for a node in the tree and insert a new one if the node could not be found. Looking up a node in a red-black tree is an expensive operation because it involves traversing the tree to find the node in question. Inserting a node is even more costly, since in addition to finding the node's location, we also need to insert it. Performing these two operations one after another is highly inefficient since we are traversing the tree twice.

We solved this problem by providing the `rbt_modify` function that looks for a node and invokes the callback passed as argument regardless of whether the element was found in the tree. `rbt_modify` takes a pre-allocated node as argument that is only used when the element requested is not found in the tree. To save traversing the tree again if the callback decides to add a node, `rbt_modify` keeps a pointer to the location in the tree of the node as the callback is invoked. The callback modifies the element if it exists, or returns the node to be inserted otherwise. To perform the insertion, we use the pointer to the location we kept in memory, avoiding traversing the tree again. When the element is found in the tree, the pre-allocated node is returned and must be freed by the COGENT code.

This design allows searching for an element and inserting one if the element was not found without traversing the tree a second time. This happened to be crucial for the performance of any operation that writes to the ObjectStore, because it involves updating the Index which is implemented using a red-black tree.

[5] The complete interface is available at https://github.com/NICTA/cogent/tree/master/impl/libgum/rbt.cogent

### Design pattern for ADT interfaces

Having introduced the interfaces of several ADTs, we distill the general principle we follow when designing ADT interfaces for a linearly typed language. ADTs interfaces must provide operations to temporarily focus a part of the data structure and manipulate it in COGENT code. As we have seen for the `Array` and red-black tree interface, we would like to be able to take in isolation a part of the data structure, read or modify it, and then logically re-insert it.

The design principle we follow consists of providing operations that invoke a combinator callback that works on the part of the data structure the client of the interface wants to focus on. Since the rest of the data structure is inaccessible from the combinator, there is no need to logically remove part from the data structure. This way COGENT code can focus on a part of the data structure (e.g. element of an `Array`, or a node of a red-black tree) and manipulate that part of the data structure while respecting the constraints imposed by the linear type-system.

In order to facilitate verification of COGENT code, we encode as much information in the COGENT types as possible. Thus, ADT operations only take the part of the state that they access, and we separate explicitly the part they read from and the part they modify. We leverage COGENT's parametric polymorphism to achieve this separation. Concrete types are specified to instantiate the type parameters of an ADT operation when it gets invoked. The read-only part of the state is passed as a shareable type so that the COGENT's type system guarantees that the combinator callback only reads from it. By contrast, the part of the state that can be modified is passed as a linear accumulator that must be returned by the combinator.

## 4.4 Cogent as a system implementation language

In this section we explain how we use COGENT to implement efficient file system code. We present another COGENT function from BilbyFs' implementation in order to highlight how the language deals with code idioms that appear in modular file system code such as: error-handling, state decomposition, memory management and low-level data structures manipulation. We conclude this section by explaining how to design top-level file system interfaces that do not expose potential memory aliasing, in order to be implementable in COGENT without breaking the linear type-system.

```
ostore_read: (ExState, MountState!, OstoreState, ObjId)
 -> ((ExState, OstoreState), <Success Obj| Error ErrCode>)
ostore_read(ex, mount_st, ostore_st, oid) =
  index_get_addr (ostore_st.index_st, oid) !ostore_st
  | Error err ->
    ((ex, ostore_st), Error err)
  | Success addr ->
    let wbuf_eb = ostore_st.wbuf_eb !ostore_st
    and ((ex, ostore_st), r) =
      (if addr.ebnum == wbuf_eb then
        let ostore_st {wbuf} = ostore_st
        and wbuf = wbuf {bound=ostore_st.used} !ostore_st
        in deserialise_Obj (ex, wbuf, addr.offs) !wbuf
        | Error (e, ex) ->
          let wbuf = wbuf {bound=mount_st.super.eb_size}
          in ((ex, ostore_st {wbuf}), Error e)
        | Success (ex, obj, sz) ->
          let wbuf = wbuf {bound=mount_st.super.eb_size}
          in ((ex, ostore_st {wbuf}), Success obj)
       else
          let ostore_st {rbuf} = ostore_st
          and ((ex, rbuf), r) = read_obj_pages_in_buf (ex, mount_st,
                                ostore_st.ubi_vol, rbuf,
                                addr) !ostore_st
          in r
          | Error e -> ((ex, ostore_st {rbuf}), Error e)
          | Success () ->
            deserialise_Obj (ex, rbuf, addr.offs) !rbuf
            | Error (e, ex) -> ((ex, ostore_st {rbuf}), Error e)
            | Success (ex, obj, sz) ->
               ((ex, ostore_st {rbuf}), Success obj)
      )
    in r
    | Success obj ->
      let oid' = get_obj_oid obj !obj
      in if oid == oid' then
        ((ex, ostore_st), Success obj)
      else
        ((deep_freeObj(ex, obj), ostore_st), Error eRoFs)
    | Error e -> ((ex, ostore_st), Error e)
```

**Figure 4.8:** Example of COGENT source code

### Cogent file system code

Figure 4.8 gives a flavour of what file system code in COGENT looks like. As described in Section 3.3, `ostore_read` reads a file system object from the ObjectStore. We present the correctness proof of `ostore_read` in Chapter 6.

**ostore_read Functionality** `ostore_read` searches the index to retrieve the physical address of the object to read and checks whether the object is located in the current eraseblock. Recall that BilbyFs maintains a buffer in memory called the write-buffer that is a synchronised to the current eraseblock on flash when `ostore_sync` is called. Thus, if the object requested is located in the current eraseblock, we directly de-serialise it from the write-buffer, without requiring any read operation on the flash storage. Otherwise, we need to read the portion of the eraseblock containing the object from the flash into a buffer and de-serialise the object from that buffer. Note that the buffer used to read objects which we call the *read-buffer* is different from the write-buffer.

`ostore_read` takes four arguments: `ex` of type `ExState` is used to model the external environment of the file system in the proofs; the read-only part of the state `mount_st`; the ObjectStore's state `ostore_st` of type `OstoreState`; and the identifier of the object to read `oid`. `ostore_read` returns a tuple containing the reference to the external environment of the file system and the ObjectStore's state, as well as a tagged union indicating whether the function succeeded or returned an error. The `Success` case also references to the object that was read, whereas the `Error` case returns an error code.

On line 4, `ostore_read` retrieves the physical address of the object from the index using `index_get_addr`. `index_get_addr` works only on the index's part of the ObjectStore's state, i.e. `ostore_st.index_st`. Since `index_get_addr` only reads from `index_st`, we pass a read-only version of the state, denoted by `!ostore_st` at the end of the line. If `oid` references no object in the store, `index_get_addr` returns `eNoEnt`, and so does `ostore_read`.

On lines 9–10, we bind the result of a if expression to a tuple:
`((ex, ostore_st), r)` where `ex` and `ostore_st` are the value described above updated and `r` is a tagged union capturing the result of the expression. The `if` expression checks whether the object is stored in the current eraseblock, and if so we de-serialise the object from the write-buffer by calling `deserialise_Obj` on line 13. Prior to calling `deserialise_Obj`, we set the bound of the buffer to `ostore_st.used`, indicating that the buffer contains valid data (only) up to that offset. This bound prevents `deserialise_Obj` from accessing data beyond that offset and potentially observing uninitialised data[6].

[6]Note that from a proof point of view, this check is not needed. As we will see in Chapter 6,

Regardless of `deserialise_Obj`'s result, we reset `wbuf.bound` to the size of an eraseblock (`mount_st.super.eb_size`).

If the object needs to be read from an eraseblock the object is not in the current eraseblock, so needs to be read from flash, we call `read_obj_pages_in_buf` on line 22 to read all the flash pages that contain a part of the object into the read-buffer (`ostore_st.rbuf`). Once we successfully read the flash, we attempt to de-serialise the object from the read-buffer. In the error case, we simply recompose `ostore_st` and propagate the error code.

Finally, on line 33, we pattern match on `r`, the result of the if expression, and in the success case we return a reference to the read object after performing a sanity check to ensure that the object we read has the correct identifier[7].

**Error handling** `ostore_read` contains a relatively large amount of error-handling code. This is no surprise, as reported by Saha et al. [2011], file systems have among the highest density of error-handling code in Linux. A lot of work report on the number of bugs found in error handling which highlights the importance of verifying error-handling code [Gunawi et al., 2008; Rubio-González and Liblit, 2011; Saha et al., 2013; Lu et al., 2014]. The current version of COGENT requires all error handling logic to be expressed explicitly.

As the `ostore_read` example shows, most of the error handling is straightforward, but verbose, and more often than not merely consists of recomposing the state and propagating error codes. On the one hand it may be frustrating to write so much error handling, on the other hand error handling is part of the file system logic, and is unavoidable as the file system cannot prevent all kinds of errors from occurring. Hence, correct handling of errors must be verified in order to achieve trustworthy file systems.

For the moment, we force the developer to explicitly consider all these branches during the development, which may increase the incentive to keep the control flow of function as simple as possible. We believe that introducing a mechanism that merely compresses the amount of error handling by introducing language constructs to propagate error codes and provide no extra verification support would be of little value. Verifying functions that have a lot of error-handling code often involves proving that for every error case, the error-handling code allows an invariant to be preserved. Removing the need for such proof obligations solely by adding features to the language seems rather difficult. By

the invariant of the file system guarantees that every object referenced by the index has a valid corresponding object stored on flash or in the write-buffer. However, such run-time check guarantees that even if the storage device assumptions get violated e.g. the flash exhibits a bit-flip error, we will not read uninitialised data from the buffer.

[7] Again, the verification proves this check unnecessary assuming that BilbyFs' invariant holds.

contrast, investigating how these proof obligations can be automatically discharged on the verification side by a combination of language and verification techniques might be a more fruitful alternative. Since COGENT specifications are purely functional, they already greatly facilitate proving that file system state preserves an invariant in the presence of errors. We discuss error-handling proofs further in Section 6.6.

**Memory management** COGENT provides control over which variables are located on the stack and which variables are allocated on the heap. Unboxed variables (i.e. tuples, unboxed records, primitive types and tagged unions) are all by default located on the stack and managed automatically by the COGENT compiler, whereas boxed records need to be allocated on the heap via a FFI call and must obey the linearity requirements of COGENT's type system. This gives developers control over memory while enabling the use of a simple purely functional shallow embedding representation to reason about the file system code. When verifying COGENT code, the difference between variables located on the stack or on the heap is essentially irrelevant. We present the memory model we use to reason about COGENT code in Chapter 6.

**State decomposition and let!** File systems often maintain a large state, but each individual function of the implementation tends to work only on a small part of that state. Exploiting such state decomposition is important to be able to reason about file system code modularly. As part of the modular design phase presented in Section 3.3, we decompose the file system state into sub-states, one for each component.

In order to understand the benefit of a modular state decomposition for verification productivity, we briefly explain the problem with verifying non-modular file system code with traditional software verification techniques. The state maintained by file systems is a large C structure with several fields. Often, by convention the whole state is passed via a pointer to most functions, which only access the subset of fields they need to provide their functionality. When using such code idioms, verifying functional correctness is extremely tedious because for every function that only accesses a small subset of the fields of the state, we have to prove that all fields that are not in that subset are irrelevant and that they are not modified by the function. Although these proofs are usually easy, they unnecessarily increase the verification effort.

In COGENT, the `take` and `put` language constructs allow decomposing and recomposing the state of COGENT programs easily. However, for the underlying C code to be efficient, the developer needs to be careful about the data structure layout of the file system state.

A naive layout for the file system state would be to use a large record with a field for each sub-state as an unboxed record, as follows:

```
type FsState = { comp1_st:#{...}, comp2_st:#{...}, ...}
```

Here, `FsState` is an boxed record, and each field corresponds to a component's sub-state is an unboxed record. `FsState` emulates the layout of traditional file systems state written in C. This way, when the file system is initialised, the code merely allocates the large `FsState` data structure on the heap, with a single call to the memory allocator and keeps the same state as long as the file system runs. The problem with such layout is that in COGENT, when we take a sub-state out of `FsState` and pass it to a component's function that only operates on that sub-state, because the sub-state is unboxed, it is passed by copy. Hence, we would pass a rather large sub-state by copy on the stack to all the operations of that component which leads to inefficient C code.

The solution for avoiding passing large sub-states on the stack is to make sub-state boxed records and allocate them on the heap when the file system is initialised. This way, when a function only operates on a sub-state, we use the `take` language constructs extract the state and pass that sub-state as a pointer, which is much cheaper than copying an unboxed record.

Combined with `let!`, `take` and `put` enable Isabelle/HOL to leverage the type information of the COGENT-generated specifications. More precisely, since COGENT specifications are purely functional and the state is explicitly passed around by value, by only passing a small part of the state to a function, we avoid having to prove that the parts untouched by the code are indeed unmodified by the function. This is one the key improvements of reasoning about COGENT as opposed to a language where memory must be modelled via a mutable heap like C or Java. With such models one must reason about how an object is updated in the heap, but also specify how the rest of the heap is unmodified. Our simpler memory model, where a state is passed explicitly by value, contributes to the increase in verification productivity when we prove the functional correctness of the COGENT code. We discuss this further in Chapter 6.

### Low-level data structure manipulations

A significant fraction of file system code is dedicated to parsing and updating data while respecting the on-medium data layout of the file system. Manipulating low-level data structures in memory is a hazardous task [Bangert and Zeldovich, 2014] and file systems often fail to do so correctly [Lu et al., 2014].

Typically, a file system reads data from storage into a buffer and manipulates data structures in that buffer. Such data manipulation makes heavy use of pointer aliasing. For instance, when file systems implement directories, part of the directory is loaded in memory and the file system parses the buffer by casting a C pointer that points to an offset within the buffer into a directory entry structure within the buffer. The file system jumps from one directory entry to another by using C pointer arithmetic.

COGENT restricts the amount of data sharing which makes parsing and updating the on-medium data format quite different from traditional file systems. In COGENT the programmer writes serialisation and de-serialisation functions to parse and update file system objects encoded in a buffer. A buffer is implemented as a `WordArray` and serialisation functions simply write to the buffer by using the `WordArray` ADT interface presented in the previous section. By implementing serialisation functions in COGENT we avoid all the memory bugs, which account for around 10% of all bugs in Linux file systems [Lu et al., 2014].

Serialisation and de-serialisation code can be automatically generated [Bangert and Zeldovich, 2014; Fisher et al., 2006]; Kanav [2014]'s recent work also automates the functional correctness proof of a subset (essentially excluding loops) of the generated code. However, for BilbyFs all the serialisation and de-serialisation code was written manually directly in COGENT. As we show at the end of this chapter, serialisation and de-serialisation functions were the main source of bugs we found when testing COGENT code.

A clear disadvantage of our approach to manipulate file system objects stored on medium is that every time we de-serialise an object from a buffer, we need to copy data from the buffer and to create a COGENT object. These objects are often too large to live on the stack, so they must be allocated on the heap with an explicit call the memory allocator. Both the copy and the memory allocation are sources of performance overhead in COGENT file systems.

### File system interface in a linear type system

COGENT's linear type system required careful design of the top-level file system interface, which is called by the VFS, to reduce the friction with the parts of the VFS interface that expect memory aliasing. Specifically, the *rename* operation expects the possibility of aliasing in its top-level entry point called from the VFS.

Rename can move a file or a directory from one directory to another or within the same directory. When a user renames a file within the same directory, the source and

```
ostore_read: (ExState, MountState!, OstoreState, ObjId)
 -> ((ExState, OstoreState), <Success Obj| Error ErrCode>)
```

**Listing 4.1:** Cogent prototype

```
struct ostore_read_args {
    ExState* p1;
    struct MountState* p2;
    struct OstoreState* p3;
    ObjId p4;
};

struct ostore_read_ret {
    struct {
        ExState* p1;
        struct OstoreState* p2;
    } p1;
    struct {
        tag_t tag;
        u32 Error;
        struct Obj* Success;
    } p2;
};
struct ostore_read_ret ostore_read(struct ostore_read_args a1);
```

**Listing 4.2:** Generated C code

target directory inode pointers passed to rename refer to the same inode structure in memory and thus alias each other. Since COGENT's linear type system does not permit aliasing, our solution was to split the rename operation into two. We write a simple C wrapper around rename that compares the addresses stored in the pointers, and invokes a new top-level operation called *move* when the pointers reference different inode structures in memory and invokes rename when they are pointing to the same one. Our COGENT rename operation only takes one directory inode, eliminating the memory alias. Spliting the operation in two led to about 150 lines of duplicated code out of a total of 3,500 lines of COGENT for BilbyFs.

### Code generation

We hinted at the COGENT to C compilation in Section 4.1. Here we elaborate on this and illustrate the systematic translation of COGENT structures to C. Most COGENT language constructs can be translated to C in a straightforward manner. More specifically, record definitions, tuples, if then else, let! and record updates have obvious equivalent

constructs in C. The most challenging aspect of code generation is dealing with tagged union types and case distincions with COGENT pattern matching. However, as we show next, once we use the appropriate C structure for tagged unions, pattern matching is straightforward to implement.

To describe the systematic translation of COGENT code to C, we present the generated data structures and prototype for the `ostore_read` function discussed earlier. Listing 4.1 shows the COGENT prototype of `ostore_read`, while Listing 4.2 shows its generated C prototype and the accompanying data structure definitions. The generated code was slightly polished for presentation. In particular, we replaced the C structs with generated names by inline structs in order to increase readability.

As described earlier, `ostore_read` takes a tuple of four elements as argument. COGENT's code generator creates a C data structure `ostore_read_args` with four fields, one for each element of the tuple. Tuples are systematically converted into a C structure with fields named $p_n$, where $n$ is the index of the field in the tuple. `ExState, OstoreState` and `MountState` are all boxed record types, thus they are linear and passed by pointer references in the C code. ObjId, for which the definition is not shown, is a 64-bit machine word, so it is passed by value.

To facilitate systematic translation of COGENT code to C, every function takes a single argument. Hence, when COGENT code calls `ostore_read`, it must pack the arguments into the `ostore_read_args` structure that represents a tuple and then call `ostore_read` by passing it to the function. Argument structures are passed by copy because our verification framework for C code does not support passing local (i.e. stack-allocated) variables by reference. Despite the fact that such C code is unconventional, our experiments showed that in most cases, an optimising compiler like gcc [Stallman, 2001] is able to detect such patterns and, when it is safe to do so, pass the data structure by reference instead in the compiled object code.

`ostore_read_ret` is the return type of the `ostore_read` function. The tagged union is described by the inline struct definition on lines 13–17. The first field, `tag`, is of type `tag_t` which is an enum indicating which constructor is in use, i.e. `Error` or `Success`. The `tag` field is used in pattern matching, to compare against the values of the enum and match the case matching the expression. The two next fields, `Error` and `Success` are C structures storing the value held by the constructor matching their respective field names. One might wonder why we do not merely pack the tagged union fields in a C union. Again, here, our experiments with gcc showed that when we use C unions, the compiler refrains from performing stack optimisations, producing a slower binary executable.

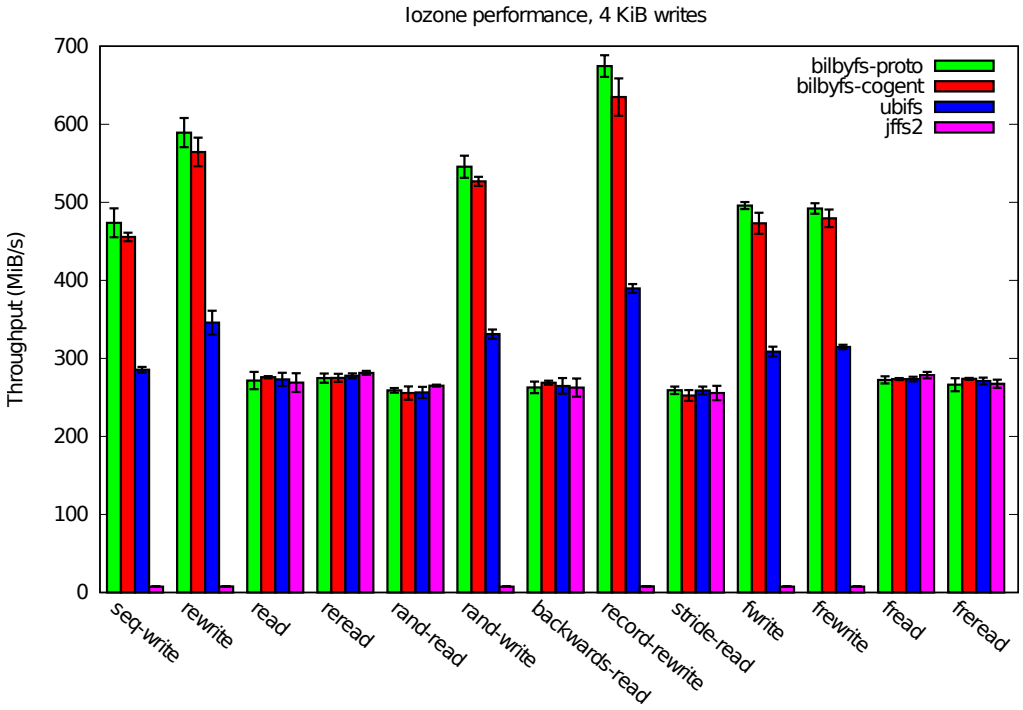


**Figure 4.9:** COGENT BilbyFs under all IOZone scenarios

## 4.5 Evaluation of BilbyFs implementation in Cogent

In this section we evaluate the performance of BilbyFs' COGENT implementation. More specifically, we are interested in making sure that the design patterns we presented in Section 4.3 do not incur unacceptable performance overhead. Since the current COGENT compiler is only a prototype, this evaluation does not aim to show that a COGENT file system achieves the same performance as the handwritten C implementation. But instead, we focus on identifying the source of the overhead and we show that despite the COGENT compiler not being optimised, BilbyFs COGENT implementation exhibits reasonably low overhead.

To evaluate the BilbyFs in COGENT, we run the same benchmarks presented in Section 3.4, on the same hardware, in order to compare the COGENT implementation of BilbyFs to its native C implementation, presented in Section 3.4. As before, UBIFS and JFFS2 serve as useful baselines for comparison.

Figure 4.9 shows that the throughput overhead of the COGENT version of BilbyFs is less than 10% on all IOZone tests. When profiling the CPU load we find $29\% \pm 2$ pp.

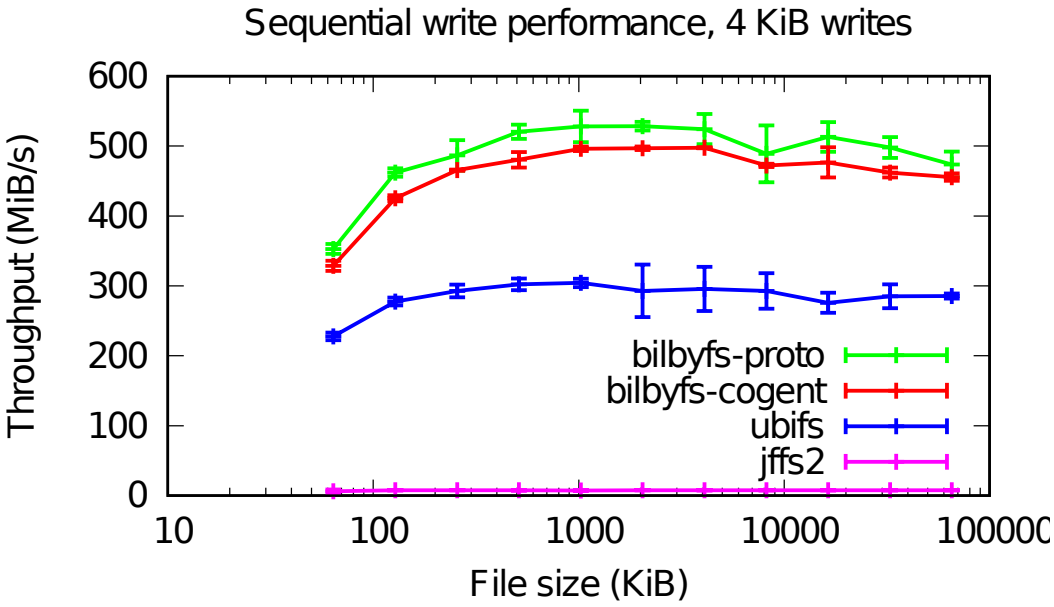


**Figure 4.10:** COGENT BilbyFs sequential writes 4K with file size increasing

(percentage point) of CPU usage for BilbyFs-COGENT compared to $26\% \pm 2$ pp. for BilbyFs-proto. As the diagrams show, despite the slow down, BilbyFs-COGENT remains faster than UBIFS on most tests.

To further analyse the overhead of BilbyFs-COGENT, we run the sequential write test of IOZone while progressively increasing the size of the file to write. Figure 4.10 plots the throughput between the first and the last write() without calling the sync() operation. Thus this test measures the run-time performance of the file system excluding the 'flush' operation. Note that the X-axis, which represents the file size, is on a log scale.

BilbyFs-COGENT is consistently slower than BilbyFs-proto with less than 10% overhead on average with a CPU usage of $23\% \pm 2$ pp. compared to the $18\% \pm 2$ pp. of the prototype version. However, despite the slow down, BilbyFs-COGENT achieves a throughput increase of around 25% when compared to UBIFS. Kernel profiling of the file system revealed that the CPU overhead is mostly the effect of redundant memory copy operations in the generated C code when passing structures on the stack. This causes to the modest slow down observed on the benchmarks. but this result shows that the read() and write() I/O path of BilbyFs can be implemented in COGENT with reasonably low performance overhead.

Figure 4.11 shows the same benchmarks but IOZone is configured with the -e option

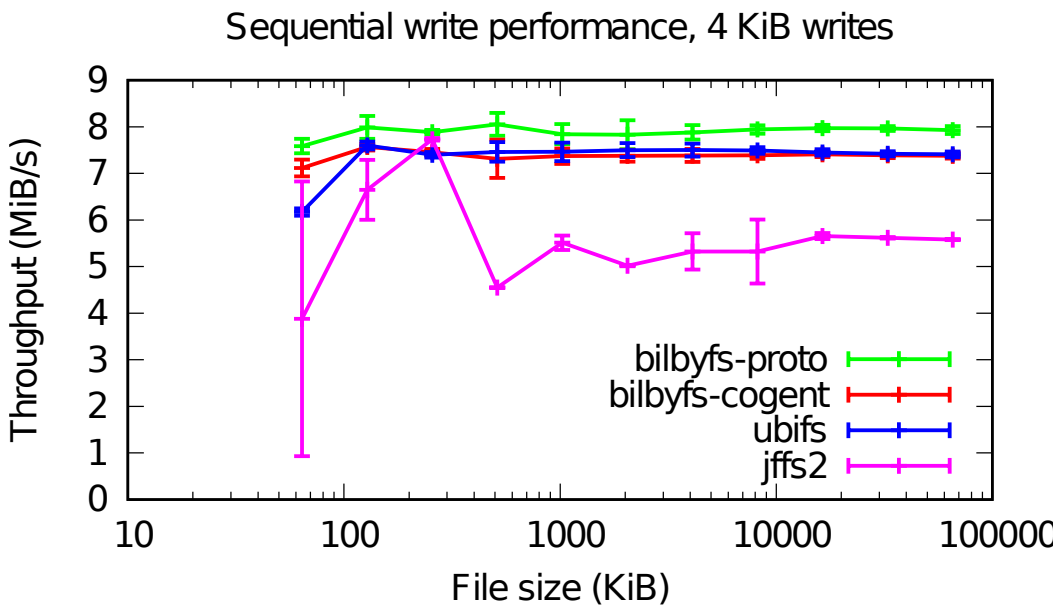


**Figure 4.11:** COGENT BilbyFs sequential writes 4K with file size increasing including sync()

to include the cost of performing a *sync()* after the last write. We observe that as the size increases, BilbyFs-COGENT and UBIFS data points get closer and are almost indistinguishable from 2 MiB onwards. BilbyFs-COGENT throughput is about 5% lower than its C counterpart. BilbyFs-COGENT uses $23\% \pm 1$ pp. of CPU compared to the $18\% \pm 1$ pp. of the C prototype. The reason for this overhead is the same as for Figure 4.10.

We evaluate BilbyFs-COGENT against some pre-defined workload personalities of Filebench [Filebench] in Figure 4.12. The detailed description of the workload is provided in Section 3.4. The overhead of COGENT on BilbyFs is less than 10% on all the Filebench tests, which is consistent with the degradation we see on the IOZone results. We now further analyse this overhead by running the postmark benchmark and profiling the file systems.

Figure 4.13 shows postmark [Katcher, 1997] results while profiling the CPU usage of each file system with OProfile [Levon]. BilbyFs-COGENT uses 51% of the CPU whereas BilbyFs-proto only uses 31%. In both implementations, the bottleneck is in the function *bilbyfs-sum-obj*, which collects the indexing information and adds it to the summary. However, the COGENT version of the function takes three times as long because the C compiler fails to optimize the COGENT loop on the WordArray ADT, to pass the element

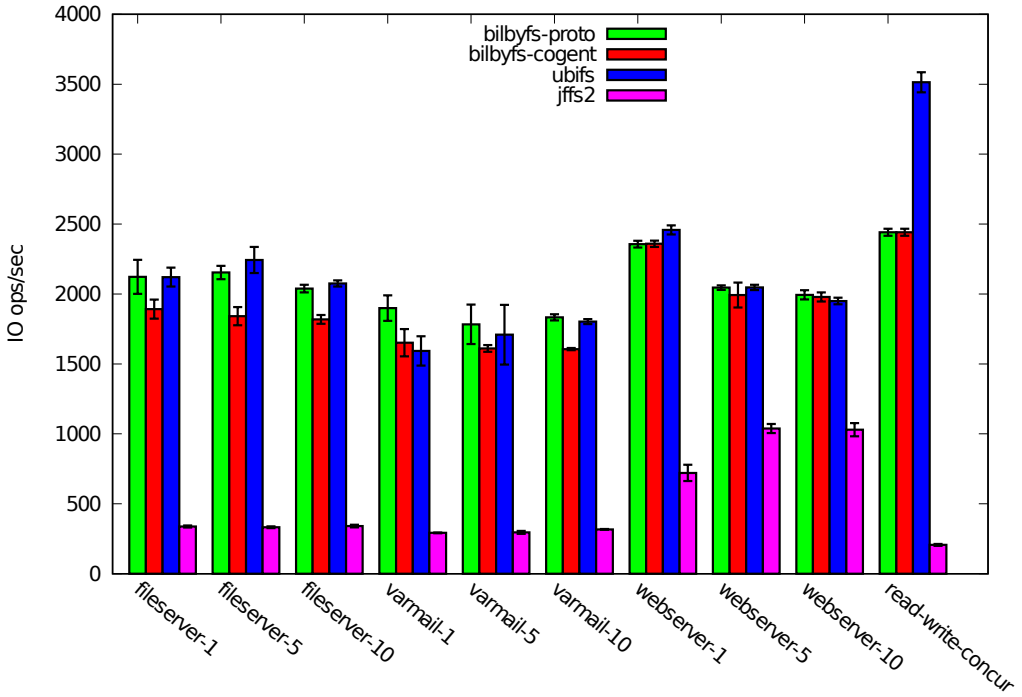


**Figure 4.12:** COGENT BilbyFs under Filebench workloads

of the array by reference for each iteration of the loop instead of copying. Since a summary is a large WordArray of summary entry of size 18 bytes each, the loop copies that many bytes in memory for each iteration every time we an object is written to the ObjectStore. A summary can easily have hundreds entries, since the number of entries depends on the number of unique objects (i.e. uniquely identified by their object identifier) serialised in the write-buffer.

The underlying reason for these results is the COGENT compiler is at present overly reliant on the C compiler's optimiser. In particular, it passes many structures on the stack, which result in much extra copying that 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. In addition, as mentioned earlier in this chapter, low-level manipulation of data structures stored on flash is not as efficient in COGENT as it is in C, because we have to invoke the memory allocator to create an object, before de-serialising which involves copying data from the buffer to the object. However, since in most cases we de-serialise objects when an operation reads from the flash, in our benchmark the overhead

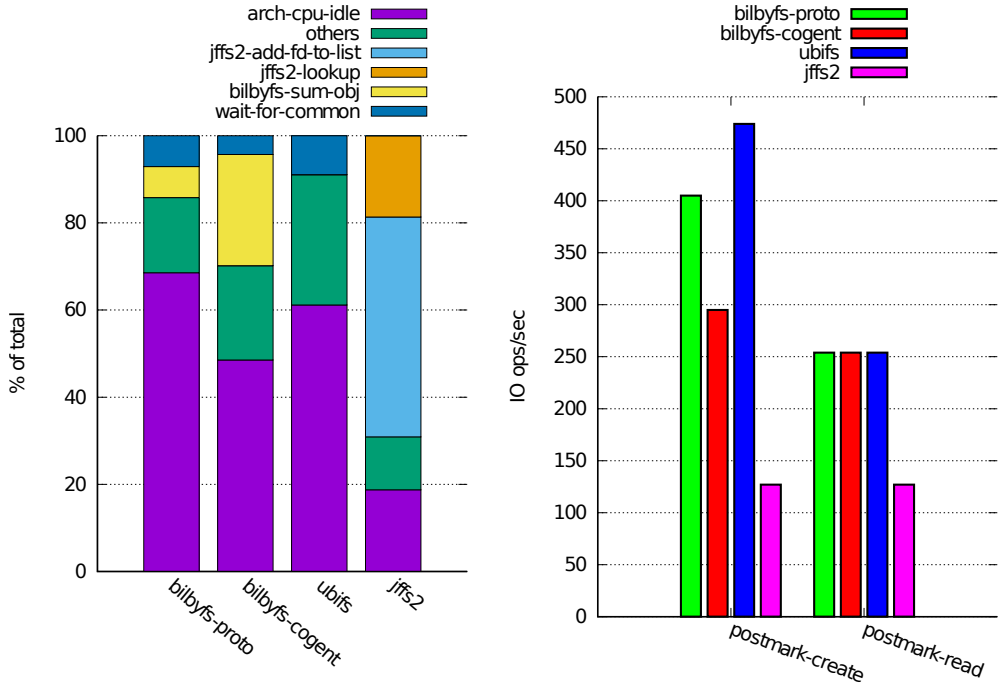


**Figure 4.13:** Performance and CPU usage for postmark benchmark for BilbyFs-COGENT

of de-serialisation is mostly hidden by the cost of waiting for the flash device.

## Bugs in Cogent implementation of BilbyFs

Although our COGENT compiler provably prevents type and memory-safety bugs, it is far from being enough to claim that a file system implemented in COGENT is bug-free. In fact we found several bugs when testing the COGENT version of BilbyFs.

We present an overview of the bugs found in order to explain the kinds of bugs that still arise in type-correct COGENT code that can be found without the use of further verification. Chapter 6 shows how to verify the absence of these kinds of bugs via a proof of functional correctness.

Most of the bugs presented here were found when running BilbyFs-COGENT for the first time. None of these defects were caught by COGENT's type system and we found them solely by testing the file system before we began the verification of the code. Some bugs were found when running the Fstest POSIX test suite [Dawidek]. This suite includes 1830 tests for operations such as: chmod, chown, link, mkdir, open, read, write, rename, rmdir, symlink, truncate and unlink.

| Bugs | Count |
|---|---|
| Logic | 11 |
| Wrapper | 7 |
| Concurrency | 1 |
| Total | 19 |

**Figure 4.14:** Bugs found when running bilby-COGENT

Figure 4.14 shows a summary of the bugs we encountered when running BilbyFs-COGENT. We found 19 bugs in total. Four bugs were typographical errors where the wrong value was assigned to a variable or where the field of a data structure was not initialised. We count such bugs in the logic category which accounts for 11 of the 19 bugs found in total. The remainder of the logic bugs were missing checks, such as not checking the maximum length of the file name in lookup(), or bugs occurring when manipulating objects stored on flash. For instance, we found two bugs caused by de-serialising an on-flash data structure from the wrong offset.

All the logic bugs would have been uncovered with a functional correctness proof on the COGENT specification of the code akin to the ones presented in Chapter 6. These two de-serialisation logic bugs could also have been avoided by using a data description language to generate the serialisation code for on-medium data structure from a declarative specification of the on-medium encoding of file system objects, as mentioned in Section 4.4.

Seven of the bugs discovered when testing BilbyFs-COGENT were found in the manually written C wrapper code that invokes COGENT entry points from the VFS (i.e. the Cache Manager, as explained in Chapter 3). Most of these bugs were very shallow, and simple testing was sufficient to find all of them because the logic of the wrapper code is intentionally very simple.

Finally, the one concurrency bug we found was due to a missing lock acquisition in the `sync()` wrapper code of the Cache Manager, meaning that the operation could run concurrently with other the file system operations. Although our verification approach presented in the next chapter does not support reasoning about concurrent code, this could easily have been found using a static analysis tool such as SDV [Ball et al., 2006] by writing usage rules for BilbyFs' API specifying that the lock must be acquired before invoking an entry point for instance.

Overall this short survey teaches us that the Cache Manager wrapper code is a non negligible source of bugs (8 out of 19) in COGENT file systems and our methodology does not prevent against these. Logic bugs, however, can be provably eliminated by proving

the functional correctness of COGENT code as explain in the rest of this thesis.

## 4.6 Summary

We presented the use of COGENT as a provably safe language designed to write and verify file system code. We described the key design decisions that made COGENT successful. COGENT is a purely functional linearly typed language. It is purposefully restricted to enable automatic correspondence proof between a formal model of the code in Isabelle/HOL and the low-level C implementation of the file system. COGENT code relies on a library of abstract data structures and loops are implemented as iterators over these data structures.

Linear type systems present several challenges that have sometimes precluded their use for general systems programing in the past. But COGENT keeps the linear types where they are most useful: for tracking memory allocation, error-handling, and higher-level control code. In addition, ADTs provide a safety-hatch for escaping the confines of the linear type system when necessary. We demonstrated that COGENT can be used to implement a carefully designed file system, and presented the key design patterns we developed when implementing BilbyFs in COGENT. These design patterns include ADT interfaces that take combinator-like callbacks as argument to perform an operation on a small part of an ADT, decomposing the file system state in a way that leads to efficient C code, and manipulating low-level data structures via serialisation/de-serialisation functions. We evaluated the COGENT implementation of BilbyFs and showed that COGENT's overhead is lower than 10% on most benchmarks and 25% in the worst case for the postmark benchmark. Finally, we analysed the bugs we found when testing COGENT code, most of which can be eliminated with a functional correctness proof on the generated-COGENT specifications as presented in the rest of this thesis.

In the following chapter, we present the high-level specification of BilbyFs that we use to prove the functional correctness of key top-level file system operations in Chapter 6.

# 5 Specification

This chapter draws on work presented in the following paper:

- ◇ Sidney Amani and Toby Murray. Specifying a realistic file system. In *Workshop on Models for Formal Analysis of Real Systems*, pages 1–9, Suva, Fiji, November 2015.

In the previous chapter, we described COGENT, a language tailor made to implement verifiable file systems. In this chapter, we focus on the first step of the verification phase of our methodology: *specifying* the correctness of the file system. BilbyFs' *correctness specification* is a formal description of the intended functionality of each file system operation. We present the most interesting elements of the correctness specification of BilbyFs. The specification supports asynchronous writes, a feature that has been overlooked by several file system verification projects [Ernst et al., 2013; Hesselink and Lali, 2009; Chen et al., 2015a]. We write the correctness specification as a functional program in Isabelle's higher-order logic and we used it in Chapter 6 to prove that BilbyFs' implementation of the `iget()` and `sync()` operations satisfy the specifications presented in this chapter.

In Section 5.1 we present the detail of the formalisation we chose to specify the correctness of BilbyFs. To support fully asynchronous writes, BilbyFs' specification explicitly separates the in-memory and on-medium file system state. We give an overview of our file system model in Section 5.2 and explain this separation further in Section 5.3. In Section 5.4 we show how to use this model to specify the correctness of file system operations including: `create()`, `iget()` and `sync()`. Finally, in Section 5.5 we discuss the limitations of our correctness specification and we conclude in Section 5.6.

## 5.1 Formalisation

Our objective is to prove the *functional correctness* of BilbyFs, specifically that every behaviour exhibited by the implementation is captured by its specification (i.e. formal refinement de Roever and Engelhardt [1998]). As explained in Section 3.1, BilbyFs implementation uses a global lock to avoid all concurrency, meaning that its specification can be entirely sequential. This specification should be concise enough to be easily audited to ensure it captures the intended behaviour of BilbyFs.

We chose to shallowly-embed the correctness specification of BilbyFs in Isabelle/HOL [Nipkow et al., 2002]. Our specification is formalised in the *nondeterminism monad*, inspired by the nondeterministic state monad of Cock et al. [2008], where computations are modelled as sets of values, one for each possible execution and associated return-value of the computation. A computation that returns results of type $\sigma$ is modelled as a value of type $\sigma$ `set`: singleton sets model deterministic computations; the empty set models computations whose semantics are not well-defined.

We define `return x` $\equiv$ `{x}`, so the primitive computation `return x` simply yields the result `x`. The "bind" operation, written `>>=`, sequences two computations together. `f >>= g` is the specification that for each result `r` of the computation `f`, executes `g` passing `r` as `g`'s argument: `f >>= g` $\equiv \bigcup_{r \in f}$ `g r`. The nondeterministic choice `x` $\sqcap$ `y` between two computations `x` and `y` is defined trivially: `x` $\sqcap$ `y` $\equiv$ `x` $\cup$ `y`. We write `do x` $\leftarrow$ `f; g x od` as sugar for `f >>= g`, in the style of Haskell.

The do-notation makes the specification readable by ordinary programmers. Being shallowly-embedded in Isabelle/HOL, the specification is more straightforward to reason about than if it were deeply-embedded. Finally, following Cock et al. [2008], this formalism supports a scalable, compositional framework for proving refinement.

Unlike Cock et al., we eschew a state monad for our correctness specification. Our specifications are simple enough that passing the file system state explicitly adds little overhead. Further, we found that a state monad makes our specifications harder to read while imposing extra mental overhead due to having to unpack and repack the state when calling into sub-modules that only touch a small part of it.

## 5.2 File System Abstraction

Having described the formalism of our specification, we now describe more precisely the level of abstraction at which we specify file system behaviours and we present our abstract model of the file system state used to specify asynchronous writes.

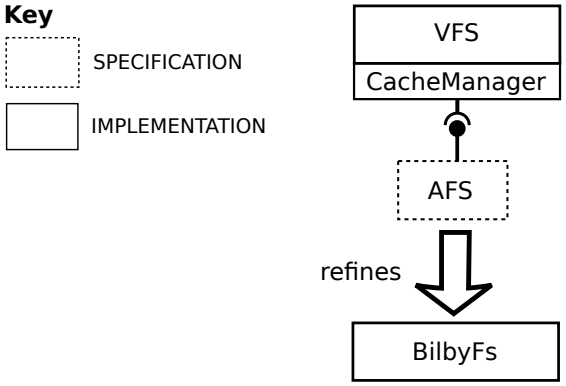


**Figure 5.1:** Correctness specification overview

Modern operating systems include a variety of file systems. Each in-kernel file system does not directly provide functionality to the user, but instead interacts via the kernel's VFS providing a common interface to all file systems. We introduced the cache manager layer that handles Linux's kernel file system caching mechanisms (see Section 3.2) to obtain a cleaner interface that we can more easily specify. Thus the top-level operations provided by BilbyFs, and described by its correctness specification, are those expected by the cache manager. These include a total of 19 file system operations:

- *`create()`*, *`unlink()`*, *`mkdir()`* and *`rmdir()`* for respectively creating and removing both files and directories.
- *`readpage()`*, *`write_begin()`*, *`write_end()`*, *`readdir()`* for reading and writing files and listing directories.
- *`lookup()`* finds an inode number by name and *`iget()`* reads an inode from disk.
- *`rename()`* renames and *`move()`* moves a file or directory.
- *`symlink()`* creates a symbolic link and *`follow_link()`* reads the path stored in a symbolic link.
- *`link()`* creates a hardlink to a file.
- *`setattr()`* and *`getattr()`* for accessing and altering attributes of file and directories.
- *`evict_inode()`* is called when an file is evicted from the cache.
- *`sync()`* synchronises in-memory updates to storage.

Figure 5.1 shows how the correctness specification, which we call the *Abstract File system Specification* (AFS), relates to the Linux VFS and to the BilbyFs implementation. BilbyFs operations are invoked by the VFS through the cache manager introduced in Section 3.2. The AFS describes concisely how each file system operation should behave. The complete correctness specification of BilbyFs for these 19 operations is about 850 lines of Isabelle/HOL.

The BilbyFs functional correctness proof, completed for the operations `sync()` and `iget()`, establishes that the BilbyFs implementation correctly implements the AFS, by proving that the former formally *refines* [de Roever and Engelhardt, 1998] the latter. Although most of the AFS is not specific to BilbyFs and could be reused to formally verify another file system, a refinement proof is greatly facilitated when the operation's specification is structurally similar to the implementation it describes. Hence, in that sense the AFS is specific to BilbyFs. However, since reasoning about our correctness specification is much easier than directly on the implementation, it would be little work to link our specification to a higher-level of abstraction that is common to all the feature-equivalent file systems.

### Modelling the File System State

Internally, a file system implements its hierarchical structure on disk in linked objects called *inodes*, which are file system specific. Similarly to previously-mentioned work [Ernst et al., 2013; Hesselink and Lali, 2009], the AFS models the file system state as the type `afs_map`, which is a synonym for a (partial) mapping from *inode numbers* (32-bit integers that uniquely identify inodes) to `afs_inode` objects. `afs_inode`s abstractly models the on-medium representation of BilbyFs inodes.

```
type_synonym afs_map = "ino ⇀ afs_inode"

datatype afs_inode_type =
  IDir "filename ⇀ ino"
| IFile "vfspage list"

record afs_inode =
  i_type :: "afs_inode_type"
  i_ino :: "ino"
  i_nlink :: "32 word"
  i_size :: "64 word"
    ...
```

An `afs_inode` represents either a directory or a file, and the `i_type` field stores both what kind of object it represents as well as the content it holds in each case. Each directory inode holds a mapping from filenames (byte strings) to associated inode numbers. Each file inode links to the data pages that hold its file's contents.

This abstraction allows the AFS to be as simple as possible while comprehensively capturing the behaviour of the various file system operations. In practice, inode implementations are a source of complexity and defects [Lu et al., 2014] for file systems. For instance the inode contents described above, which the AFS models as stored directly in the `i_type` field of the `afs_inode` object, will in practice be stored in separate data structures on the medium to which the inode links, often via multiple levels of indirection.

The AFS represents a file's contents as a list of VFS pages (each of which is a list of bytes). An alternative would have been to use just a single flat list of bytes. However, we chose the former because the VFS layer interacts closely with the Linux page cache (i.e. memory management subsystem) which is directly reflected in the VFS API that BilbyFs implements.

Directory inodes are usually implemented as a collection of directory entries stored in a file. The AFS instead models directory contents as a mapping from filenames to inode number: the inode number for a particular filename identifies the inode that implements the file or directory with that name. In file systems like BilbyFs that support *hardlinks*, the same file inode may be linked to by multiple filenames (in possibly different directories). Like most file systems, BilbyFs restricts hardlinks to point only to file inodes, to ensure that the file system's logical structure is acyclic.

### AFS Invariant

The aforementioned requirement on hardlinks is encoded as part of the global invariant of the AFS, which we describe only briefly because (like the AFS' state itself) it is very similar to that of Ernst et al. [2012]. The invariant assertion includes the following requirements of the `afs_map` structure:

- A root inode exists
- for each directory inode, there is only one directory with an entry pointing to that inode — i.e. no hardlinks on directories
- the `i_ino` field of each inode matches the `afs_map` mapping
- the `i_nlink` field of each inode correctly tracks the number of links to that inode

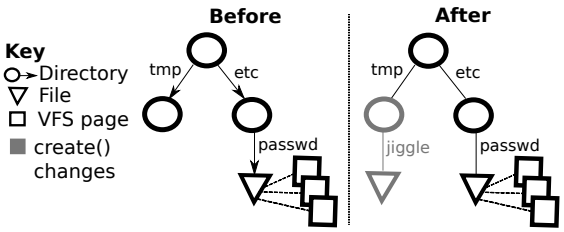


**Figure 5.2:** The effect of the *create()* operation

- the *i_size* field of each file inode is consistent with the number of VFS pages attached to the inode.

## Modelling File System Updates

The AFS' abstract representation of the file system state makes describing the effects of the file system operations easy, and in turn facilitates auditing of the specification.

Figure 5.2 depicts the changes to the *afs_map* structure when the *create()* operation is called to create a new file `jiggle` in the `/tmp` directory. Grey nodes are those altered or added by the operation. Creating a file involves adding a file inode and a link to it in the parent directory. Directories are pictured as circles with arrows denoting each entry in the directory. Small triangles denote files and are linked to VFS pages shown as tiny squares. The newly created file contains no data so no VFS page is attached to its inode.

The effect on the *afs_map* structure *m* of creating a new file "jiggle" in a directory (whose inode is) *itmp*, by installing the new file inode *ijiggle*, is specified by the following single line of Isabelle/HOL[1].

$$m\langle i\_ino\ itmp \mapsto i\_dir\_upd\ (\lambda dir.\ dir\langle ''jiggle'' \mapsto i\_ino\ ijiggle\rangle)\ itmp,\ i\_ino\ ijiggle \mapsto ijiggle\rangle$$

We write $m\langle a \mapsto b\rangle$ to denote updating the (partial) map *m* such that *a* maps to *b*.

Each transformation to the file system state may be captured by a function of type $afs\_map \Rightarrow afs\_map$. We call such functions *file system updates*. We exploit this idea in the following section, where we describe the model of asynchronous writes used in the AFS.

[1]Note that the specification for the *create()* operation (see Figure 5.3) is a lot more complex than this single line, because it needs to also incorporate error checking and handling, and interactions with the Linux VFS.

## 5.3 Specifying Asynchronous Writes

The *afs_map* type models the state of the file system as stored on the physical storage medium, which in the case of BilbyFs is the raw flash storage device, and updates to the storage medium are simply transformations: *afs_map* ⇒ *afs_map*. Like many other realistic file systems, although the effects of file system operations become visible as soon as those invocations return, the actual storage medium update that they implement may not be applied until some point in the future, for instance when *sync()* is next invoked. Thus writes to the storage medium are performed *asynchronously*, an essential feature of file systems since the original UNIX [Thompson, 1978]. Storage medium updates are therefore buffered in-memory, allowing operations such as *create()* and *unlink()* to return straightaway, without incurring the cost of writing to the storage medium. For the file system correctness specification, this implies that the in-memory file system state and the state of the physical storage medium (*afs_map* in the AFS) need to be distinguished, especially if the semantics of operations like *sync()* are to be precisely specified.

As mentioned earlier, several file system models [Ernst et al., 2013; Hesselink and Lali, 2009; Chen et al., 2015a] overlook this requirement. In the AFS for BilbyFs, the pending writes buffered in-memory are modelled as a sequence of file system transformations, each of type *afs_map* ⇒ *afs_map*. The global state of the specification is captured by the type *afs_state*, for which the record definition follows.

```
record afs_state =
  a_is_readonly :: "bool"
  a_current_time :: "time"
  a_medium_afs :: "afs_map"
  a_medium_updates :: "(afs_map ⇒ afs_map) list"
```

Besides the state of the physical medium (of type *afs_map*) and in-memory pending updates (of type *(afs_map ⇒ afs_map) list*), *afs_state* also includes a boolean flag that tracks whether the file system has been placed into *read-only mode*, which can occur for instance if a disk failure arises; as well as a record of the current time, which is used for instance when updating the timestamps on an inode that track e.g. the last time it was modified.

To model the idea that file system modifications become visible straightaway, even when they have not yet been applied to the physical storage medium, the AFS needs a way to calculate the (hypothetical) file system state that includes both the physical medium and the pending in-memory updates, i.e. the state that *would* arise if all of those updates were applied to the medium. It is this hypothetical state that must be

considered, for instance, when the *unlink()* operation is invoked to remove an inode that was previously *create()*ed but has not yet been *sync()*ed to disk. It is calculated by the function *updated_afs*, that makes use of the standard *fold* function to apply the in-memory updates to the medium state:

**Definition 1.** *updated_afs afs_state* ≡
```
fold (λx. x) (a_medium_updates afs_state) (a_medium_afs afs_state)
```

An operation like *create()* that updates the file system state may buffer the updates it performs in memory, or (if the in-memory buffer is full) it may cause preceding updates to be applied to the storage medium. Given that the size of the in-memory buffer is below the level of abstraction of the AFS, the precise number of updates that may be propagated to the storage medium could vary upwards from zero. The AFS captures this effect via the following helper function, which nondeterministically splits the list of updates into two parts: *to_apply*, the updates to be applied to the medium; and *rem*, the remainder. It then applies the updates in *to_apply* and updates the in-memory list of pending updates to *rem*.

**Definition 2.** *afs_apply_updates_nondet afs* ≡
```
do (to_apply, rem) ← {(t, r) | t @ r = a_medium_updates afs};
   return
    (afs(|a_medium_afs := fold (λx. x) to_apply (a_medium_afs afs),
           a_medium_updates := rem|))
od
```

The following helper function *afs_update* then generically specifies the process for updating the file system state, and is used in the specifications of the various file system operations (see e.g. *create()* in Figure 5.3). It takes an update function *upd* of type *afs_map* ⇒ *afs_map*. It adds it to the back of the list of in-memory updates and then calls *afs_apply_updates_nondet*. If after *afs_apply_updates_nondet* returns, the list of in-memory updates is empty, then *afs_apply_updates_nondet* caused all in-memory updates (including *upd*) to be propagated to the storage medium, in which case the modification must report *Success*. Otherwise, it might succeed (if, for instance, the new update is simply buffered in-memory without touching the storage medium), or report an appropriate error (because a write to the medium failed, or a memory allocation error occurred etc.). If an error is reported, the new update *upd* is forgotten, ensuring operations that report an error do not modify the (combined in-memory and on-medium) file system state.

**Definition 3.** *afs_update afs upd* ≡

```
do afs ←
   afs_apply_updates_nondet
    (afs(|a_medium_updates := a_medium_updates afs @ [upd]|));
   if a_medium_updates afs = [] then return (afs, Success ())
   else return (afs, Success ()) ⊓
       nondet_error {eIO, eNoSpc, eNoMem}
        (λe. (afs(|a_medium_updates := butlast (a_medium_updates afs)|),
              Error e))
od
```

Importantly, this specification requires that no updates get lost when an error occurs: each is either applied (in order), or is still in the list of pending updates (in order). It also requires the BilbyFs implementation not to report an error if it succeeds in propagating all updates to disk. Thus the implementation cannot attempt to allocate memory, for instance, after successfully writing to disk. In practice, file system implementations structure each operation such that all resource allocation (and other actions that could potentially fail) occur early, so no potentially-failing operation needs to be performed after successfully updating the storage medium.

The `afs_update` definition is the heart of how the AFS specifies asynchronous file system operations while keeping the AFS concise and readable. In the following section, we present specifications of the most interesting top-level file system operations from the AFS.

## 5.4 Specifying File System Operations

### Specifying `create()`

The specification for the `create()` operation is shown in Figure 5.3. Recall that BilbyFs' top-level operations, like `create()`, are those expected by the cache manager presented in Section 3.2. The cache manager is invoked by Linux's VFS. Since the VFS interacts with a range of different file systems, each of which may have its own custom inode format, the VFS provides a common inode abstraction, called a *vnode*. Top-level file system operations invoked by the VFS often take vnodes as their arguments and return updated vnodes in their results. The `afs_inode` structure mentioned in Section 5.2 is very similar to the generic vnode structure of the VFS.

Much of the complexity of Figure 5.3 comes from error checking and handling, as well as specifying the correct interaction with the VFS (e.g. conversion from vnodes to

```
afs_create afs vdir name mode vnode ≡
1  if a_is_readonly afs then return ((afs, vdir, vnode), Error eRoFs)
2  else do r ← afs_init_inode afs vdir vnode (mode || s_IFREG);
3          case r of Error (afs, vnode) ⇒ return ((afs, vdir, vnode), Error eNFile)
4          | Success (afs, vnode) ⇒
5            do r ← read_afs_inode afs (v_ino vdir);
6               case r of Error e ⇒ return ((afs, vdir, vnode), Error e)
7               | Success dir ⇒
8                 do r ← return (Success (i_dir_update (λd. d⟨αwa name ↦ v_ino vnode⟩) dir))
9                       ⊓ return (Error eNameTooLong);
10                   case r of Error e ⇒ return ((afs, vdir, vnode), Error e)
11                   | Success dir ⇒
12                     do r ← Success ‘ {sz | v_size vdir < sz} ⊓ return (Error eOverflow);
13                        case r of Error e ⇒ return ((afs, vdir, vnode), Error e)
14                        | Success newsz ⇒
15                          do time ← return (v_ctime vnode);
16                             dir ← return (dir(|i_ctime := time, i_mtime := time|));
17                             inode ← return (afs_inode_from_vnode vnode);
18                             (afs, r) ← afs_update afs (λf. f⟨i_ino inode ↦ inode,
19                                                              i_ino dir ↦ dir⟩);
20                             case r of Error e ⇒ return ((afs, vdir, vnode), Error e)
21                             | Success () ⇒
22                                 return ((afs, vdir(|v_ctime := time, v_mtime := time,
23                                                     v_size := newsz|), vnode),
24                                         Success ())
25                          od
26                   od
27               od
28         od
29     od
```

**Figure 5.3:** Functional specification of the *create* operation.

`afs_inode`s).

The state of the file system, of type `afs_state`, is passed as the argument `afs`. `create()` also takes the parent directory vnode `vdir`, the name `name` of the file to create, a mode attribute `mode`, and a vnode `vnode` to fill with the information of the newly created file. It returns three values: the updated file system state, the updated parent directory vnode, and the updated vnode.

The specification precisely describes the file system behaviour expected by the VFS, including possible failure modes. For instance, the implementation needs to return an error if the file system is in read-only mode (line 1). On line 2, it allocates a new inode number and initialises the vnode fields by calling the function `afs_init_inode` (not shown). If `afs_init_inode` returns an `Error` (line 3), `create()` returns `afs`, `vdir` and `vnode` unchanged, as well as the error `Error eNFile` indicating the file system ran out of inode numbers. This pattern is repeated on lines 6, 10, 13 and 20, which each check for errors in preceding operations and specify that these errors must be propagated to `create()`'s caller, leaving the file system state unchanged.

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

**Figure 5.4:** Functional specification of the *iget* operation.

On line 5, the specification reads the parent directory given by the `vnode` argument, and computes what the new directory should look like (with the file being created added to it). Line 12 implies that the size of the directory must increase. The core of `create()` is specified on lines 18–19 where the file system state is updated with the updated directory and file inode.

### Specifying `iget()`

Figure 5.4 shows the correctness specification of the `iget()` operation, which looks-up inodes on the physical medium. It takes a inode number `inum` and a vnode structure `vnode` as argument. It first (line 1) 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 (line 6) an appropriate error code is returned. Otherwise, the `iget()` specification reads the inode with that number from the medium (line 2) and then returns appropriately based on whether the inode read succeeded (line 4) or produced an error (line 3). In the case of `Success`, the inode must be converted to a vnode structure for inter-operating 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.

### Specifying `sync()`

Figure 5.5 shows the specification for `sync()`, the file system operation that propagates all in-memory updates to the disk. `sync()` returns an appropriate error when the file system is in read-only mode. Otherwise, it propagates the in-memory updates to the medium using the `afs_apply_updates_nondet` function of Section 5.3. Recall that this function

```
afs_sync afs ≡
1   if a_is_readonly afs then return (afs, Error eRoFs)
2   else do afs ← afs_apply_updates_nondet afs;
3           if a_medium_updates afs = [] then return (afs, Success ())
4           else do e ← select {eIO, eNoMem, eNoSpc, eOverflow};
5                   return (afs(|a_is_readonly := e = eIO|), Error e)
6               od
7       od
```

**Figure 5.5:** Functional specification of the *sync* operation.

applies the first $n$ in-memory updates, with $n$ chosen nondeterministically to model the effect of e.g. a disk failure happening part-way through. `sync()` returns successfully when all updates are applied; otherwise it returns an appropriate error code. When an I/O error occurs (`eIO`), indicating a storage medium failure, the file system is put into read-only mode to prevent any further updates to the medium (whose state may now be inconsistent).

The economy of the `sync()` specification shows the advantage we obtain by carefully choosing an appropriate representation for the in-memory updates, separate from the on disk state of the file system.

## 5.5 Limitations

We discuss the BilbyFs AFS in order to tease out its limitations and avenues for future improvement. An obvious limitation of the AFS is that it supports no form of concurrency, and so implicitly specifies that top-level operations cannot run concurrently to one another. This limitation arises primarily because the verification methodology (based the COGENT framework) we used to prove the functional correctness of file system operations does not support reasoning about concurrent programs.

Another limitation of the AFS is that it imposes a strict ordering on all updates to be applied to the storage medium. In practice, many other file systems impose weaker ordering constraints, especially file systems that are highly concurrent or those built on top of low-level block interfaces that are asynchronous. Some file systems can reorder asynchronous writes of data but not those for meta-data.

The AFS, besides excluding concurrency, also does not specify the interaction between BilbyFs and the Linux kernel's inode, directory entry and page caches. These interactions are isolated and implemented in the cache manager component pictured in Figure 5.1 that itself must be verified in isolation.

Another limitation of the AFS arises because function arguments are passed by value (rather than e.g. as pointers to variables in a mutable heap). This prevents the AFS from specifying VFS operations that take multiple pointer arguments that point to the same variable (i.e. are aliases for each other). As explained in Chapter 4, we eliminate data sharing from the interface in order to be able to implement all file system operations in COGENT.

A final limitation of the AFS presented here is that, unlike e.g. the recent work of Chen et al. [2015a], it does not specify the correct behaviour of the *mount()* operation, which is called at boot time, nor does it specify the file system state following a crash, for instance to require that the file system is crash-tolerant.

## 5.6 Summary

In this chapter, we presented the most interesting elements of BilbyFs' correctness specification, a formal description of the intended behaviour of each file system operation. We write the specification as functional program shallowly embedded in higher-order logic and we keep it concise by using nondeterminism. As a result, the specification is easy to reason about using the Isabelle/HOL proof assistant, and is easy to audit to ensures it captures the intention of the system's designers and users. BilbyFs' correctness specification is, to our knowledge, the first to model full asynchronous writes for a POSIX file system, by separating the in-memory file system state and the on-medium one. This separation enables us to precisely specify the behaviour of the *sync()* operation. In the next chapter, we use our correctness specification to prove the functional correctness of two operations of BilbyFs, namely: *iget()* and *sync()*.

# 6 Verification

This chapter draws on work presented in the following paper:

- ◇ 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, and Gernot Heiser. Cogent: Verifying high-assurance file system implementations. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 175–188, Atlanta, GA, USA, April 2016. doi:10.1145/2872362.2872404.

In Chapter 5 we presented BilbyFs' correctness specification called the *Abstract File system Specification* (AFS). We explained how it concisely captures the expected functionality of the file system. The AFS separates the in-memory and on-medium file system state which enables us to accurately specify the `sync()` operation. In this chapter, we present our verification methodology to *modularly* verify the *functional correctness* of COGENT code. By functional correctness we mean writing a mathematical machine-checked proof that guarantees that every behaviour exhibited by the implementation is captured by its specification, i.e. formal refinement as presented in de Roever and Engelhardt [1998]. And by modular verification we mean that the structure of the proof follows the modular design of the file system and reasons about each component of the implementation in isolation. We show evidence of the effectiveness of our modular verification methodology by proving the functional correctness of two key operations of BilbyFs: `iget()` and `sync()`. We also analyse the effort required for the aforementioned proofs, in order to determine whether the overall methodology, for designing and verifying file systems, leads to a reduction in verification effort compared to traditional systems

software verification. We use the seL4 [Klein et al., 2014] verification as a point of comparison, and find that we achieve a reduction in effort by about 50%.

Existing modular verification techniques [Vaynberg and Shao, 2012; Feng et al., 2006; Ernst et al., 2014; Gu et al., 2015; Chen et al., 2015b] are not well suited for our purposes for the following reasons. First, almost none of the frameworks we are aware of work for a shallow embedding like our COGENT specifications; most of them require a meta theory to compose proofs about deeply embedded programs [Vaynberg and Shao, 2012; Gu et al., 2015]. Second, some of these meta theories target system software with components implemented in multiple languages (e.g. C, assembly, etc.), and they require writing relations linking the semantics of one language to another. Hence, these frameworks tend to be more complicated than necessary for our purposes. The goal of the COGENT certifying compiler is to simplify reasoning about file system code. To this end, we want COGENT-generated specifications to be a purely functional shallow embedding that matches the input source code as closely as possible to ease verification, so that the programmer's intent is exactly what the verification engineer reasons over, not some convoluted formal encoding of it. COGENT-generated specifications do not require a memory model with mutable heaps, and thus we do not need to resort to cumbersome machinery like separation logic [Reynolds, 2002]. In COGENT specifications, $x$ and $y$ being different objects follows trivially from $x$ and $y$ being separate variables. This way, COGENT specifications enable the verification engineer to make heavy use of equational reasoning and Isabelle's powerful rewriting engine; we return to these claims in Section 6.2 and Section 6.4. Finally, we also value the readibility of high-level specifications written as a functional program so that both the system's users and designers can check it correctly captures their intentions and understanding of the system. Some of the modular verification frameworks mentioned earlier do not support functional specifications [Feng et al., 2006; Chen et al., 2015b].

To bridge the gap between a high-level functional specification and COGENT specifications, we present a modular and lightweight verification framework that allows us to fully exploit the design modularity of COGENT code, and we relate our experience using it to prove the correctness of the `iget()` and `sync()` operations of BilbyFs.

## 6.1 Verification strategy and chapter overview

Figure 6.1 shows BilbyFs' proof strategy that we use to explain the general verification methodology of our approach. Our objective is to prove modularly the *functional correctness* of the COGENT file system implementation.

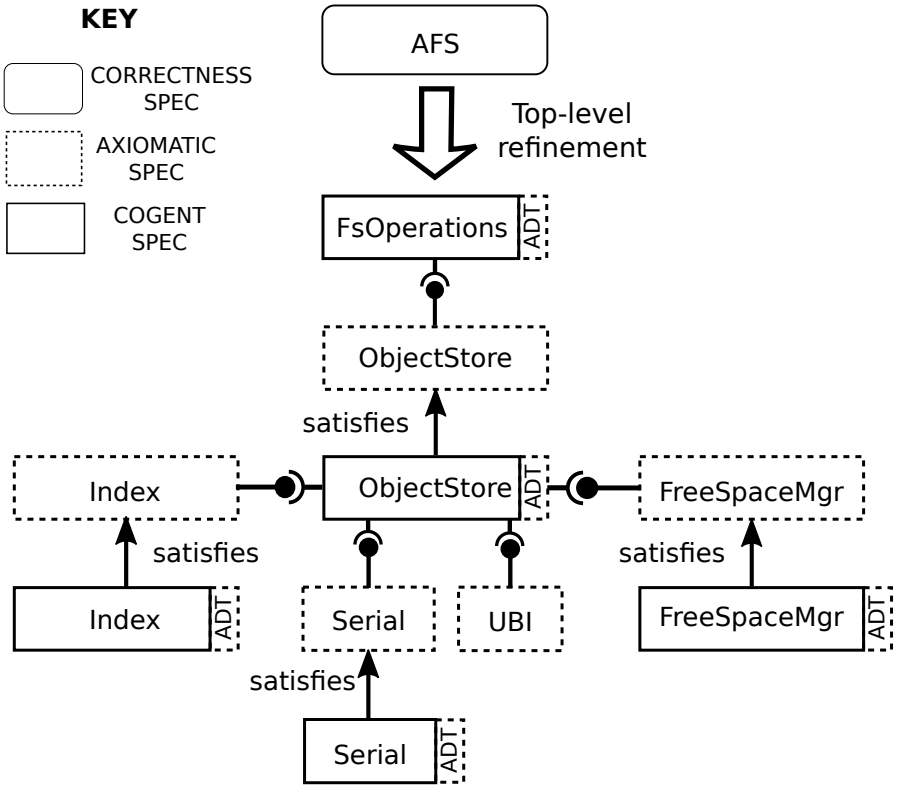


**Figure 6.1:** BilbyFs' verification strategy

From the COGENT specification of a file system operation, we want to prove that it *refines* the functionality described by its AFS. To this end, we developed a *refinement framework* for COGENT specifications in the spirit of Cock et al. [2008]. We return to this framework in Section 6.3.

The refinement proof operates over the top-level component of BilbyFs, namely the FsOperations component, and proves that each of its behaviours is present in the AFS. This proof exploits the modular design of BilbyFs by assuming that all components used by FsOperations are themselves correct; later steps of the proof formally prove these components are indeed correct. The only component used by FsOperations is the ObjectStore.

Figure 6.1 depicts the structure of the full refinement proof. It begins by assuming the correctness of the ObjectStore component, whose correct functionality is formally described by an *axiomatic specification*. An axiomatic specification is a list of formal assumptions (i.e. axioms) that abstractly describe the interface provided by a component and allow client components to be verified without having to reason about the internal

implementation details of the components on which they rely. Axiomatic specifications must be validated later on by proving that the component's implementation satisfies every axiom of the specification.

We follow the same principle to prove the correctness of the complete chain of components; each is in turn shown to satisfy its axiomatic specification by relying on its sub-components' axioms. For instance, we show that the ObjectStore correctly implements its axiomatic specification by relying on the axioms of the four components: Index, Serial, UBI and FreeSpaceMgr. The proof bottoms out at the ADTs (abstract data types) and the UBI components, which we leave for future work.

In the rest of this chapter we detail our verification methodology, and describe how we used it to prove the full component chain up to the UBI component and ADTs, for the functional correctness of two file system operations: `iget()` and `sync()`. `iget()` reads an inode from the flash device, whereas `sync()` writes the pending in-memory updates to the flash. Both are crucial file system operations that form key building blocks for proving the correctness of other file system operations or for a crash resilience proof. The `iget()` proof involves proving the ObjectStore operation that reads generic file system objects from the flash, which is used by all the operations that read from the medium. `sync()`, on the other hand, as we have seen in Chapter 5, may nondeterministically be called by any operation that writes to the medium. To our knowledge, ours is the first proof of functional correctness for `sync()` of any file system.

We begin by presenting an overview of COGENT specifications in Isabelle higher-order logic (HOL), where we highlight their similarities with the input COGENT source code and report on the challenges of generating a convenient Isabelle/HOL shallow embedding representation of COGENT code in Section 6.2. We present the refinement framework for COGENT specifications in Section 6.3 and we explain how we specify components interface using axiomatic specifications in Section 6.4. Then we use BilbyFs as a case study for our modular verification methodology. In Section 6.5, we describe the refinement relation of BilbyFs and explain how to encode file system invariants in it. In Section 6.6, we outline the modular proof of BilbyFs' operations `iget()` and `sync()`. Finally, in Section 6.7 we report on our experience using the methodology presented in this chapter and evaluate the proof modularity and the productivity of our approach, before summarising this chapter in Section 6.8.

## 6.2 Cogent specifications

COGENT allows verification engineers to reason about file system code at roughly the same level of abstraction as programmers. Hence, from COGENT code we generate COGENT specifications that are syntactically very close to the input source code. These specifications are shallowly embedded in Isabelle/HOL, a language of total functions with a type system reminiscent of ordinary functional programming languages (without support for sophisticated typing features like linear types for instance). Thus the generated COGENT specifications in Isabelle/HOL preserve all of the type information that is useful to prove correctness over those specifications, while omitting details like linearity.

For instance, all record types, tuples and tagged-union types are preserved in COGENT specifications because they are of substantial help to understand the specification. By contrast, the difference in semantics between linear objects and non linear ones, let and let!, or unboxed and boxed records is irrelevant when reasoning over pure functions — these are implementation details only — thus we eliminate them in COGENT specifications.

As explained in the Chapter 3 and Chapter 4, a key requirement of our methodology is to produce performant file systems. COGENT's linear type system allows us to generate efficient C code with in-place updates of variables in the heap, from a purely functional input source code. If we were to implement a file system directly in Isabelle/HOL, without language support such as the linear type system, it would be much harder to generate efficient C code without implementing garbage collection, for instance. Hence, we opted for designing our own language and generating Isabelle/HOL COGENT specifications instead. Nevertheless, in order to facilitate the verification of COGENT programs, it is important for the COGENT-generated specifications to be as intelligible as the handwritten COGENT code.

In the rest of this section, we compare COGENT's input source code and the matching COGENT-generated specifications with an example. Figure 6.2 shows the top-level function of the `sync()` operation of BilbyFs both in COGENT source and in COGENT specification. Many of the language constructs such as let, if, tuples, cases are syntactically very similar and semantically equivalent. COGENT-generated specifications have extra variables, named $ds_n$, where $_n$ is a unique number generated by the COGENT compiler. These variables are introduced by the de-sugaring phase of the compiler, which converts syntactic sugar into core language constructs. But in general the code and its specification show the same structure.

COGENT provides syntactic sugar for the 'take' language construct, which takes a field out of a record, e.g. the arguments of `fsop_sync` use the expression $\#\{ex, fs\_st\}$,

```
fsop_sync #{ex, fs_st} =
  let is_ro = fs_st.fsop_st.is_ro !fs_st
  in if is_ro then
    (#{ex, fs_st}, Error eRoFs)
  else
    let fs_st {ostore_st,fsop_st} = fs_st
    and ((ex, ostore_st), r) = ostore_sync(ex,
          fs_st.mount_st, ostore_st,
          ostoreWriteNone) !fs_st
    in r
    | Error err ->
      (#{ex, fs_st = fs_st {ostore_st,
                fsop_st = fsop_st {is_ro= err == eIO }}},
       Error err)
    | Success () ->
      (#{ex, fs_st = fs_st {ostore_st,fsop_st} },
       Success ())
```

```
fsop_sync ds₀ ≡
let (ex, ds₂) = take_CG ds₀ ex_f; (fs_st, ds₁) = take_CG ds₂ fs_st_f;
   is_ro = is_ro_f (fsop_st_f fs_st)
in if is_ro then (make ex fs_st, Error eRoFs)
  else let (ostore_st, ds₃) = take_CG fs_st ostore_st_f;
         (fsop_st, fs_st) = take_CG ds₃ fsop_st_f;
         (ds₅, r) =
           ostore_sync (ex, mount_st_f fs_st, ostore_st, ostoreWriteNone);
         (ex, ostore_st) = ds₅
      in case r of
        Error err ⇒
          (make ex
            (fs_st
             (|ostore_st_f := ostore_st,
                fsop_st_f := fsop_st(|is_ro_f := err = eIO|)|)),
           Error err)
        | Success v_G0 ⇒
            (make ex
              (fs_st(|ostore_st_f := ostore_st, fsop_st_f := fsop_st|)),
             Success ())
```

**Figure 6.2:** COGENT code (top) and COGENT specification in Isabelle/HOL (bottom)

which is syntactic sugar to take the fields `ex` and `fs_st` out of the unboxed record passed as argument of the function. To emulate the semantics of 'take' expressions, COGENT specifications use the Isabelle definition $take_{CG}$. Recall from Chapter 4 that in order to accommodate the linear type system, COGENT 'take' expressions return two values: the record with the field marked as taken, and the field's value. Our $take_{CG}$ definition does just that, it takes a record and a projection extracting a field to read from a record as argument, and it returns the record and the value read.

COGENT 'put' expressions are replaced by the Isabelle record update syntax, where $r(\!|a_f := v|\!)$ updates the field `a` of record `r` with the value `v`. To construct a record, we use the `make` function that takes the value of each field as argument and returns the newly constructed record[1].

Like the type linearity information, `let!` disappears in the COGENT specifications. In COGENT code, `let!` makes the linear type system more convenient to use by removing the need to return a linearly typed object when we only read from it. Type linearity and `let!` are only useful to provide hints, leveraged by the COGENT certifying compiler, to generate efficient C code. This type information is indispensable to implement in-place updates of linear objects in the C source code. However, it is no longer necessary in Isabelle COGENT specifications because they are purely functional and variables are passed by value which greatly simplifies high-level reasoning in a proof assistant.

The rest of the code and the specification match very closely, giving the verification engineer a convenient formal model of the COGENT code to reason about, with a proof that the model is an accurate representation of the low-level C implementation. In the next sub-section, we discuss the main ideas that underpin the generation of COGENT specifications while converting COGENT structural typing to an equivalent representation in Isabelle/HOL.

### Generating Cogent specifications

The main ideas underpinning the generation of COGENT specifications in Isabelle/HOL are all geared towards ensuring that the COGENT compiler can produce an automatic proof of correspondence between the generated specifications and the (de-sugared) COGENT code. Proof-producing generation of shallow-embeddings is well covered terrain [Myreen, 2012; Greenaway et al., 2012] so here we focus on just those elements that are new. Specifically, representing structural types in Isabelle/HOL and case expressions. Structural typing

[1]Each record type has a `make` constructor. To avoid type ambiguity COGENT specifications qualify `make` with the name of the record type. However, for readability we do not show qualified names in our examples.

means a type may be referred to by its name or its structure interchangeably. This greatly simplifies some aspects of a language but also substantially complicates generating specifications in a Isabelle/HOL, since its type-system is not structural.

**Polymorphism and structural typing** Consider the following COGENT types:

```
type R1 = {f1 : T1, f2 : T2}
type P a = {f1 : T1, f2 : a}
type R2 = P T2
```

Because COGENT types are structural, `R1` and `R2` are the same, and they may be used interchangeably. However, a naive translation of these types into Isabelle/HOL records produces irreconcilable record types.

```
record R1 =
  f1_f :: "T1"
  f2_f :: "T2"

record 'a P =
  f1_f :: "T1"
  f2_f :: "'a"
```

Specifically, there is no easy way to convince Isabelle that instantiating P with T2 is the same as R1, i.e. in Isabelle two types with the same structure but with have different names are different. The fragment of Isabelle/HOL above declares two records `R1`, and `P` that takes a type parameter `'a` for its field $f2_f$.

Our approach to emulate structural typing and to propagate COGENT polymorphism to type definitions in Isabelle is to encode type *structures* as Isabelle/HOL records, rather than specific types. To do so we make every field of every record (or tagged-union) polymorphic, and to generate only one record for all the records with the same structure. For instance, the COGENT code above leads to the following Isabelle definitions:

```
record ('a, 'b) P =
  f1_f :: "'a"
  f2_f :: "'b"

type_synonym  R1_T = "( T1,  T2) P"
type_synonym 'a P_T = "( T1, 'a) P"
type_synonym  R2_T = " R1_T"
```

Where P is a record with two type parameters, one for each of its fields. Note that the record is named `P` and not `R1` merely because `P` is a shorter name; i.e. we choose the

```
type V = <A U32 | B U16 | C U8>

foo: V! -> U64
foo v =
  v
  | A x -> u32_to_u64 x
  | B x -> u16_to_u64 x
  | C x -> u8_to_u64 x
```

```
foo ds_0 ≡
let v = ds_0
in case v of A x ⇒ u32_to_u64 x
  | B v_G1 ⇒
      case B v_G1 of B x ⇒ u16_to_u64 x
      | C v_G0 ⇒ Let (case C v_G0 of C x ⇒ x) u8_to_u64
  | C v_G1 ⇒
      case C v_G1 of B x ⇒ u16_to_u64 x
      | C v_G0 ⇒ Let (case C v_G0 of C x ⇒ x) u8_to_u64
```

**Figure 6.3:** COGENT code (top) and COGENT specification in Isabelle/HOL (bottom)

shortest name among all the types that have the same structure.

Along with the record definition, we generate a type synonym with the suffix $_T$ for each COGENT type synonym. We apply the same trick to translate COGENT tagged-unions into Isabelle data types, but tagged-unions lead to further complications as we explain in the next paragraph.

**Cogent case expressions and tagged-union** Figure 6.3 shows a small fragment of COGENT code with a case expression that does a case distinction on a tagged-union type with three constructors A, B and C, each holding a machine word of different size.

The COGENT specification is more verbose than the input COGENT code. This happens because the COGENT compiler internally de-sugars case expressions into a cascade of case statements, each matching a single constructor at a time, as shown in Figure 6.4. The universal pattern `_` matches any constructor. Since the COGENT pattern maching expressions consumes the constructor matched, the variable that the follows the universal pattern has a smaller tagged-union type where the constructor that was matched on is removed. In the de-sugared version of `foo`, `v'` is of type `<B U16 | C U8>` and `v''` is of type `<C U8>`, i.e. a tagged-union with a single constructor `C`.

To emulate this behaviour in Isabelle, we generate the following data type definitions that respectively match the types of `v''`, `v'` and `v`:

```
type V = <A U32 | B U16 | C U8>

foo: V! -> U64
foo v =
  v
  | A x -> u32_to_u64 x
  | _ v' ->
    v'
    | B x' -> u16_to_u64 x'
    | _ v'' ->
      v''
      | C x'' -> u8_to_u64 x''
```

**Figure 6.4:** De-sugared case expression in COGENT

```
datatype 'a V001 =
  C "'a"

datatype ('a, 'b) V011 =
  B "'a"|
  C "'b"

datatype ('a, 'b, 'c) V111 =
  A "'a"|
  B "'b"|
  C "'c"

type_synonym  VT = "(32 word, 16 word, 8 word) V111"
```

Note that all data types take a type parameter for each of their constructors for the reason explained in the previous named paragraph. Each data type name has a subscript bit-field to indicate which of the constructors are enabled/disabled. Since $V_{001}$ only has the constructor `C` enabled, we use the bit-field $_{001}$ to indicate that the first and second constructors are disabled. From the COGENT source code, the COGENT compiler infers all the different combinations of constructors disabled actually used in the code, and it only generates data types for these. For the example in Figure 6.3 we need the three data types declared above.

While in Figure 6.3, the Isabelle specification seems closer to the de-sugared version in Figure 6.4 than the original source code, using the rewriting rules automatically generated for every Isabelle/HOL data type, the COGENT specification of `foo` is trivially simplified

by Isabelle's rewriting engine "simp", as follows:

```
foo ds₀ ≡
case ds₀ of A x ⇒ u32_to_u64 x | B x ⇒ u16_to_u64 x | C x ⇒ u8_to_u64 x
```

Conveniently, this form is syntactically closer to the input COGENT code shown on Figure 6.3. This highlights the benefits of having specifications directly in shallowly embedded in Isabelle/HOL, as it allows us to use Isabelle's powerful rewriting engine to simplify the terms. All we had to do is call the simplifier with the rules `V₀₀₁.simps` and `V₀₁₁.simps` which are automatically generated by Isabelle when a data type is declared. These simplification rules eliminate the case expressions for which the matching constructor can be determined statically.

Our COGENT specifications provide a convenient formal model of the underlying C code, with a machine-checked proof that guarantees that it is an accurate abstraction of the implementation. They provide the same level of abstraction as COGENT code, while discarding the linear type information, which, as mentioned, is unnecessary for reasoning on top of the specifications.

In the next section we present the framework we built to prove refinement between the generated COGENT specifications we just introduced and abstract specifications in the style described in Chapter 5.

## 6.3 Refinement framework for Cogent specifications

Our objective is to prove refinement between an abstract program acting as a specification and a concrete one corresponding to the implementation of the system. Formal refinement [de Roever and Engelhardt, 1998] consists of showing that every behaviour exhibited by the implementation is also present in its specification. One way to show refinement is to prove *forward simulation*, whereby each execution step of concrete program is shown to correspond to some matching step of the abstract one. Correspondence is defined by constructing a state relation `R` that relates abstract and concrete program states. When `R` relates states that use different data structures, as it is the case here, the proof technique is called *data refinement*.

In our context, the initial state of the state machine is the file system's state after the `mount()` operation is called and the steps are a sequence of file system operations. Our verification framework allows us to show that a step on the concrete state machine corresponds to a step on the abstract one, via the state relation `R`.

Refinement may be viewed as a proof technique between two possibly nondeterministic programs [Cock et al., 2008; de Roever and Engelhardt, 1998], where we quantify over all the elements of the set of results returned by the implementation, and show that there exists a computation produced by the specification that is related via `R`. However, since our COGENT specifications are deterministic purely functional higher-order logic (HOL) programs, our framework only allows proving refinement between an nondeterministic specification and a deterministic implementation. This greatly simplifies the correspondence statement which formally states our notion of refinement. Correspondence is defined as follows:

**Definition 4.**

```
cogent_corres R absspec cogentspec ≡ ∃ra∈absspec. R ra cogentspec
```

It says that the COGENT specification `cogentspec` *refines* the abstract specification `absspec`, if and only if, the non-deterministic `absspec` can produce a computation related, by the relation `R`, to the computation of the `cogentspec` program, which is a shallowly embedded program in HOL. `R` relates the return values of the abstract specification and the COGENT-generated specification. This correspondence statement will be proved for each pair of corresponding functions between the abstract specification and concrete implementation.

Typically the arguments to those functions must also be related via a proof assumption, so that it is possible to prove the relation of the return values. The following example of theorem shows how we relate arguments and return values when specifying a correspondence statement:

**Theorem 1** (Example of correspondence statement relation)**.**

```
rel abs_st cogent_st ⟹
cogent_corres
 (λ(abs_st', ra) (cogent_st', rc). rel abs_st' cogent_st' ∧ ra = rc)
 (afs_op abs_st) (fsop cogent_st)
```

The result of abstract and concrete programs are usually a pair of values containing the updated state (`abs_st'` and `cogent_st'`) and a tagged-union indicating whether the operation succeeded or returned an error (`ra` and `rc`). The lambda expression extracts the pair of values and specifies the relation between them. Thus, this example of correspondence statement states that assuming that the arguments `afs_st` and `fsop_st`

$$\frac{a \qquad \neg\, a \qquad a' = a}{\texttt{cogent\_corres R (if a then b else c) (if a' then b' else c')}}\,\text{COGENT\_IF}$$

$$\frac{\bigwedge \texttt{ra rc.}\ \dfrac{\texttt{r = Error ra} \qquad \texttt{r' = Error rc}}{\texttt{cogent\_corres R (c ra) (c' rc)}} \qquad \bigwedge \texttt{ra rc.}\ \dfrac{\texttt{r = Success ra} \qquad \texttt{r' = Success rc}}{\texttt{cogent\_corres R (d ra) (d' rc)}} \qquad (\exists \texttt{v. r = Error v}) = (\exists \texttt{v. r' = Error v})}{\begin{array}{c}\texttt{cogent\_corres R (case r of Error rc} \Rightarrow \texttt{c rc | Success rd} \Rightarrow \texttt{d rd)} \\ \texttt{(case r' of Error rc} \Rightarrow \texttt{c' rc | Success rd} \Rightarrow \texttt{d' rd)}\end{array}}\,\text{CG\_RES}$$

$$\frac{\texttt{R x y}}{\texttt{cogent\_corres R (return x) y}}\,\text{COGENT\_RETURN}$$

$$\frac{\texttt{v} \in \texttt{S} \qquad \texttt{cogent\_corres R (a v) c}}{\texttt{cogent\_corres R (select S >>= a) c}}\,\text{COGENT\_SELECT}$$

$$\frac{\texttt{cogent\_corres R a (c v)}}{\texttt{cogent\_corres R a (Let v c)}}\,\text{COGENT\_LET\_EXEC}$$

**Figure 6.5:** Correspondence rules of our refinement framework

are related by the relation *rel*, the COGENT file system operation *fsop* corresponds to the non-deterministic specification *afs_op* such that the state *cogent_st'* returned by *fsop* and the abstract state *afs_st'* returned by *afs_op* are related by the relation *rel* too, and the tagged-union results *rc* and *ra* are equal.

Our correspondence statement is much simpler than the one of Cock et al. [2008] for two reasons. First, as explained in Section 5.1, our AFS avoids using a state in the non-deterministic monad because our specifications are simple enough that passing the state as function argument manually adds little overhead. Second, since our concrete COGENT specifications are deterministic, they are shallowly embedded in HOL, and as opposed to Cock et al. [2008], they do not require a nondeterminism nor mutable state.

As alluded to in Chapter 5, our refinement framework exploits the structural similarity between the abstract specifications and the COGENT ones. Figure 6.5 defines the set of syntax-directed rules we use to prove the top-level refinement step of our verification strategy described earlier. The rules are meant to be used in backward reasoning, i.e. they are applied when the conclusion of the rule matches the proof goal. All of them are straightforward, the key idea is that the conclusion of each rule syntactically matches on

both the abstract specification and the COGENT one.

For instance, the rule `cogent_if` matches a `if _ then _ else` statement on both specifications. Once applied, the `cogent_if` rule creates sub-goals to show that both abstract and concrete programs correspond when the condition is true, when it is false and that the conditions on both sides are equivalent. The rule `cg_Res` follows the same principle for the case construct of Isabelle, but in addition to splitting the goal, it extracts the value held in each case of the `Result` data type (i.e. the cases `Error` and `Success`).

On the other hand, `cogent_select` simply match syntactically on the abstract specification level because its purpose is to instantiate a nondeterministic choice specified at the abstract level with a value matching the one picked by the concrete level, where the choice does not exist. Furthermore, `cogent_select` forces us to pick a value in the set of choices imposed by the abstract specification.

All these rules are lemmas that we proved, i.e. they are not axioms but are *sound* proof rules. However, we omit the detail of our proofs because they follow from the definition of correspondence and are straightforward.

## 6.4 Axiomatic specifications

Having introduced the framework we use to prove the top-level refinement proof, we now present our methodology for describing the interface of sub-components of the modular decomposition of the file system. These components include the ObjectStore, as well as all those below it in Figure 6.1.

An *axiomatic specification* is a list of formal assumptions or axioms describing the interface of a component using abstractions that expose just enough information to capture the properties of the interface without detailing the underlying implementation. The correctness proof of each implementation component relies on the axiomatised behaviour of its sub-components. This way, each component is shown to satisfy its specification in isolation.

These axiomatic specifications are used to establish the top-level refinement or to prove another axiomatic specification. Hence, as opposed to the top-level correctness specification presented in Chapter 5 which favours readability, axiomatic specifications are carefully crafted to facilitate their usage in proofs. Thus, whenever possible, we phrase axiomatic specifications as equalities, fostering equational reasoning using Isabelle's rewriting engine, the *simplifier*. Alternatively, we write axioms as HOL proof rules.

To illustrate our purpose, we show a fragment of the axiomatic specification for the COGENT WordArray ADT (see Section 4.3).

```
axiomatization
  α :: "'a WordArray ⇒ 'a list"
and
  make :: "'a list ⇒ 'a WordArray"
and
  inv_wa :: "'a WordArray ⇒ bool"
where
  wordarray_make:
  "⋀xs. α (make xs) = xs"
and
  wordarray_length_ret:
  "inv_wa arr ⟹ unat (wordarray_length arr) = length (α arr)"
and
  wordarray_get_ret:
  "inv_wa arr ⟹ unat i < length (α arr) ⟹
     wordarray_get (arr, i) = α arr ! unat i"
and
  wordarray_modify_ret:
  "⋀P. ⟦ inv_wa arr ; unat i < length (α arr);
      ⋀r arr'. ⟦ r = modifier (((α arr)!unat i), acc, obs) ;
           arr' = make (α arr[unat i:=fst r]);
           inv_wa arr' ⟧ ⟹
       P (arr', snd r)
   ⟧ ⟹
     P (wordarray_modify (arr, i, modifier, acc, obs))"
and
  wordarray_create_ret:
   "⋀P.⟦
      ⋀ex'. (ex', None) = malloc ex ⟹ P (Error ex');
      ⋀ex' v. ⟦ sz > 0 ;
                  (ex', Some v) = malloc ex;
                  length (α v) = unat sz;
                  inv_wa v
                  ⟧ ⟹
                 P (Success (ex', v))
        ⟧ ⟹
      P (wordarray_create (ex, sz))"
```

To better describe the experience of the verification engineer when using axiomatic specifications, we show the proof rules directly in the format understood by the Isabelle proof assistant. In Isabelle/HOL, a proof rule written as ⟦A; B⟧ ⟹ C is equivalent to

$\frac{A \qquad B}{C}$; and $\bigwedge$ is a meta universal quantifier, so $\bigwedge x.\ P\ x$ is is equivalent to $\forall x.\ P\ x$.

The `axiomatization` command allows positing the existence of a set of constants with associated properties (axioms). In our WordArray specification example, we fix three constants, $\alpha$, `make` and `inv_wa`; all three are used to state the assumptions of the axiomatic specification. The WordArray interface abstracts over the array element type `'a`. $\alpha$ is an *abstraction* that provides an abstract view of WordArrays and allows specifying its interface without revealing its implementation details. Thus $\alpha$ function turns a concrete `'a WordArray` into a `'a list`, i.e. Isabelle's type for lists. `make` goes the other way: it converts a list back into a WordArray. We borrowed this idea from the Isabelle Collection Framework [Lammich and Lochbihler, 2010] which has a similar function for describing the array interface of the library. We only use `make` to specify WordArray axioms. `inv_wa` is the invariant required and maintained by every axiom of the WordArray specification. We have an invariant because in practice any implementation may have an internal invariant that is required to hold for its API functions to work correctly. Invariants ensure that the ADT specifications are implementable. For instance, `inv_wa` guarantees that the size of the array is always greater than zero. Note that `inv_wa` is established by `wordarray_create` when we create a WordArray and the rest of the interface guarantees that the invariant is maintained. Since ADTs are meant to be verified separately, we leave the invariant underspecified such that each specific ADT implementation can define its own internal invariant.

In general, axiomatising the behaviour of a component requires capturing the effects of its functions with respect to its internal state. But this internal state cannot be precisely described in the interface specification, as this would break the design principle of information hiding [Parnas, 1972]. Abstraction functions are a known technique for specifying interfaces of abstract data types [Lammich and Lochbihler, 2010]. They expose just enough information to capture properties of the interface without detailing the actual implementation.

The axiom `wordarray_make` establishes that $\alpha$ is the inverse of `make`, for any list `xs`. `wordarray_length_ret` specifies that the results of `wordarray_length`, which returns a 32-bit word, converted to natural number must be equal to the length of the list.

`wordarray_get_ret` describes the axiom to read an element of the array. The axiom requires two premises: the invariant `inv_wa arr` and that the index to access the array is within the bound of the array[2]. `wordarray_get_ret` is written to enable equational

[2] Although the semantics of out-of-bound accesses is undefined in the WordArray axiomatic specification, in COGENT code out-of-bound accesses have well-defined semantics. Since BilbyFs

reasoning. Specifically, its conclusion is phrased as an equality, which makes it easy to use with Isabelle's automatic rewriting engine, and therefore is well suited to increase proof automation.

The `wordarray_modify_ret` axiom describes the function that modifies an element of the array. When writing an element, we also need to know that the index is within the bound of the array as specified by the assumption of the axiom. `wordarray_modify` takes a `modifier` callback that is invoked to modify the element at index `i`. The callback takes three arguments: the element to modify, an accumulator `acc` to pass the mutable part of the state, and `obs` the observable part of the state. The modifier returns the element modified and the accumulator which are both used to form the return value of the `wordarray_modify` function. This axiom uses the free higher-order predicate `P` to be able to describe the changes of the array through the abstract functions $\alpha$ and `make`, while still being able to assert that the invariant holds on the updated array. In the `wordarray_modify_ret` rule, the predicate `P` is used to match the surrounding context (the program text surrounding the call to `wordarray_modify`. When the rule is applied, the proof goal has the function call substituted with the result of the function call as modeled by the axiomatic specification. Because we can only describe the update to the array via these abstract predicates, this rule's conclusion cannot be written as an equality unlike `wordarray_get_ret`.

Ideally, every axiom would be phrased as an equality in order to foster equational reasoning, but as we have seen with `wordarray_modify_ret`, sometimes it is not a possible, e.g. while maintaining abstraction. This is also the case when the second part of the equality is so complicated that breaking up the problem into multiple smaller ones might help. The `wordarray_create_ret` axiom demonstrates this. When a COGENT program calls the function `wordarray_create`, there are two possible outcomes: the WordArray is successfully created, or an error occurs. Even if we could phrase the axiom as an equality with a disjunction on the right-hand-side, this would force us to split the goal manually once the rule has been applied. We found that a better solution is to phrase the axiom as a introduction rule, that splits the current goal into multiple ones, one for each possible outcome in which the axiomatised term is substituted with a term representing the outcome. The first outcome of `wordarray_create` is when `malloc`[3] returns an error and the second one is when `malloc` is successful. In the latter case we also know that the size requested with the second argument of `wordarray_create` is greater than zero and

always uses in-bound indices (so long as our proof assumptions hold), out-of-bound accesses need not be defined.

[3] `malloc` is itself axiomatised as a function returning either an object or an error.

that the size of the allocated array is consistent with the requested size.

Axiomatic specifications of BilbyFs' components tend to be more complicated than the WordArray's one, though not fundamentally different. In particular, the ObjectStore has an interface that requires differentiating between two types of invariants: *internal* and *external* ones. The external invariant asserts consistency on the $\alpha$ abstraction of the component functionality, whereas the internal one ensures the consistency on the internal fields of the concrete state of the component, so that the implementation complies with its abstract specification. We describe BilbyFs' ObjectStore axiomatic specification further when we relate our experience applying our verification methodology in Section 6.6 and Section 6.7. But before that, we describe BilbyFs' refinement relation and its invariants in the next sections.

## 6.5 BilbyFs' Refinement relation and invariants

Having introduced the framework we use to prove the top-level refinement step and the axiomatic specifications for the rest of our proof strategy, we now present BilbyFs' refinement relation that we instantiate the framework with, i.e. the `R` parameter of the correspondence statement introduced in Definition 4. In our framework, the `R` relation specifies how the AFS state relates to the concrete COGENT file system state, and we also use it to specify that the invariant must be preserved by the refinement step. Presenting its complete definition is not the purpose of this section, instead we focus on the interesting parts of the relation that would be transferable to proofs of other file systems.

BilbyFs' refinement relation, called `afs_fsop_rel`, is composed of six predicates that either relate various parts of the abstract and concrete states, or assert invariants on them:

- `afs_fsop_match_step_updates`, which establishes the connection between the AFS logical representation of a file system and BilbyFs' ObjectStore state for all the prefixes of the list of pending updates.
- Read-only flags of the AFS' and BilbyFs' state, which must be equal.
- The AFS' invariant, presented in Section 5.2, holds on abstract logical representation of the file system, with any prefix of the list of pending updates applied to the AFS state.
- `inv_`$\alpha$`_ostore`, the ObjectStore external invariant, which asserts consistency on the abstraction of the component functionality. This invariant is necessary to prove the

| Logical representation of | file system | pending updates | read-only flag |
|---|---|---|---|
| Abstract specification | `a_medium_afs` | `a_medium_updates` | `a_is_readonly` |
| Related via | `afs_fsop_match` | | = |
| Concrete implementation | $\alpha$`_ostore_medium` | $\alpha$`_updates` | $is\_ro_f$ |

**Figure 6.6:** Refinement relation overview

correctness of the file system operations implemented on top of the ObjectStore (i.e. FsOperations component proofs).

- `inv_ostore`, the ObjectStore internal invariant that restricts the set of values the ObjectStore's state can have and ensures we can prove that the ObjecStore's implementation satisfies its axiomatic specification.

- `inv_superblock`, the superblock invariant. The superblock stores information about the flash device such as the number of eraseblocks, the size of a flash page, etc. The superblock invariant ensures that these values are consistent with each other.

The relation `afs_fsop_rel` is defined as a conjunction of all these properties. The rest of this section gives an overview of the refinement relation, and BilbyFs' invariants mentioned above. In order to describe the ObjectStore's internal invariant, we explain how we formalise BilbyFs on-flash log format, and to describe the ObjectStore's external invariant, we show the abstractions functions used to specify the interface presented by the ObjectStore. Later in Section 6.6 of this chapter, we use `afs_fsop_rel` to formulate the top-level correspondence statement for the operations `iget()` and `sync()`.

### BilbyFs' refinement relation

Recall from Section 5.2 that the AFS abstraction is composed of `a_medium_afs` and `a_medium_updates`. `a_medium_afs` is an abstract logical view of the file system of type `afs_map`, and `a_medium_updates` is a list of pending file system updates of type `(afs_map ⇒ afs_map) list`. To prove data refinement [de Roever and Engelhardt, 1998], these parts of the AFS must be formally connected to the concrete state of BilbyFs. We leverage the simplicity of BilbyFs' ObjectStore abstraction to specify the relation between the abstract and the concrete state of the refinement. Figure 6.6 summarises the mapping between the AFS state and the implementation one. The AFS logical representation of the on-medium files and directories has a direct mapping via `afs_fsop_match` to the

ObjectStore abstraction $\alpha$*_ostore_medium*. $\alpha$*_ostore_medium* returns the abstract view of the ObjectStore, but only considering the data stored on flash. To construct this abstract view, $\alpha$*_ostore_medium* logically mimics behaviour of the *mount()* operation which requires parsing the contents of the flash. $\alpha$*_updates* returns the list of in-memory pending updates of the ObjectStore. The *afs_fsop_match* relation has the type *afs_map* $\Rightarrow$ *ostore_map* $\Rightarrow$ *bool*. We use the relation to match the AFS' state to the ObjectStore's state with and without the pending updates applied.

The *afs_fsop_match* relation requires that there is a file system object in the AFS *iff* (i.e. if and only if) there is a matching one in the ObjectStore. More precisely, the relation is valid when all the following conditions are satisfied:

- There is an inode in the *afs_map* iff there exists a matching one in the ObjectStore.
- For every file inode in the AFS, VFS pages (i.e. data blocks) are attached to the inode iff the matching data block objects are in the ObjectStore.
- For every directory inode in the AFS, there is a file name to inode number mapping in the directory iff there exists a dentarr (i.e. directory entry array — see Section 3.3) object in the ObjectStore with the same name and inode number combination.

We use *afs_fsop_match* in *afs_fsop_match_steps*, the relation asserted by *afs_fsop_rel*, to specify that all the prefixes of the list of pending updates in the abstract and concrete BilbyFs states match.

The abstraction provided by the ObjectStore, implemented via the $\alpha$*_ostore_medium* and $\alpha$*_updates* functions of its axiomatic specification, makes this refinement relation straightforward to specify. This highlights the benefits of designing a file system *modularly*, with verification in mind.

In the rest of this section, we describe the specification of a valid BilbyFs log segment and the abstract representation of the ObjectStore in more detail. Both are necessary to describe the *external* and *internal* invariants of the ObjectStore. The external invariant asserts the well-formedness of the ObjectStore directly on its abstract representation, whereas the internal invariant ensures the consistency of the fields of the ObjectStore implementation. We also describe the invariant on the super-block at the end of this section. All these invariants are asserted in the refinement relation of our framework.

### BilbyFs on-flash physical representation

In order to describe the refinement relation between the AFS and BilbyFs' COGENT state, we first need to explain how the file system meta-data written as raw bytes on the physical

storage is abstracted into a high-level representation of file system objects. We use such abstractions abundantly in our proofs; they are necessary to talk about transactions in BilbyFs' write-buffer as well as to define $\alpha$`_ostore_medium` which involves parsing objects from the log.

As explained in Chapter 4, the COGENT implementation of BilbyFs uses serialisation and de-serialisation functions to store and parse file system objects in a buffer. To verify BilbyFs, we had to specify and verify the behaviour of these functions. In this sub-section we describe the HOL definitions that specify abstractly the effect of de-serialising a file system object from a buffer and serialising an object into a buffer. In Section 6.6, we prove that when successful, the serialisation and de-serialisation functions have the effect described by these abstractions.

**On-flash encoding of file system objects** The main difficulty of specifying BilbyFs' on-flash format is that some objects have a variable length. The ability to store variable length objects is one of the fundamental tasks performed by realistic file systems. For instance, each entry in a directory records a name that is chosen by user applications of the file system, hence does not have a constant length. The only way to store directory entries efficiently, that is without always allocating the maximum length allowed for a valid name, is to have directory entries of variable length.


**Figure 6.7:** BilbyFs log format: objects encode their own length in their header

BilbyFs' on-flash format of log segments is almost identical to UBIFS' [Hunter, 2008] journal on-flash format, and both of them use variable length directory entries. In BilbyFs, the length of an object is encoded in its header, hence we must first parse the header and then parse the rest of the object according to its length. This complicates the format specification because functions to parse file system objects are only well defined[4] when the

[4]HOL allows for `undefined` statements, that is statements that lead to unspecified results. `undefined` statements often lead to dead ends in Isabelle/HOL proofs, thus we always try to guard against them by writing predicates that guarantee every result we have to explore in the proof is specified.

buffer contains a valid object header. Figure 6.7 shows a log segment, which is a sequence of log transactions, each of which comprises a sequence of objects. The validity of a log segment depends on the length of the first objects in the log, which indicates where to parse the next object, and so on. When objects encode their own length, specifying the validity of a log segment must be done inductively. We return to these issues later in this sub-section.

The definition we use to *parse* the header of file system objects follows:

**Definition 5.**

```
pObjHeader data offs ≡
make (ple32 data offs) (ple32 data (offs + 4)) (ple64 data (offs + 8)) offs
 (ple32 data (offs + 0x10)) data[unat (offs + 0x16)] data[unat (offs + 0x17)]
 undefined
```

`pObjHeader buf offs` parses an object header from the buffer `buf` of type `U8 list` (where U8 means 8-bit word) at the byte offset `offs` of type `U32` and returns a record of type $Obj_T$ with the fields initialised with the value parsed from the buffer. `ple32 buf offs` parses a litte-endian 32-bit machine word from the buffer `buf` at the byte offset `offs`, whereas `ple64` does the same for 64-bit words. `pObjHeader` parses the object field per field, in the same order as the record definition of $Obj_T$ which follows[5]:

**Definition 6.**

```
record Obj_T =
magic_f :: "U32"  (* Magic number *)
crc_f :: "U32"    (* Checksum value *)
sqnum_f :: "U64"  (* Transaction number *)
offs_f :: "U32"   (* Offset of object in segment *)
len_f :: "U32"    (* Length of object in segment *)
trans_f :: "U8"   (* Transaction tag *)
otype_f :: "U8"   (* Type of object *)
ounion_f :: "ObjUnion_T" (* Tagged-union object value *)
```

`pObjHeader` parses every field except for $offs_f$ which is only only kept in memory since it is easy to recover at run-time, and $ounion_f$ which corresponds to the payload of

[5]Note that the record definition is polished for presentation, otherwise, as explained in Section 6.2, all fields would be type parameters and $Obj_T$ would be a type synonym.

the object and is not part of its header. *pObjHeader* also skips two bytes at offsets 0x14 and 0x15 which are filled with padding. The payload in $ounion_f$ will be filled in later on, with a value of type $ObjUnion_T$, that is consistent with the field $otype_f$ representing the type of the object, and $len_f$ its length. $ObjUnion_T$ is a data type with a constructor for each object type, including inode, data blocks, directory entry arrays and padding objects which we discuss later in this section. *pObjHeader* is only well defined if the buffer is large enough to contain an object header at the position of the offset passed as argument.

Having seen how we abstract *parsing* (i.e. deserialisation), we now briefly explain the opposite function to *serialise* an object header as list of bytes:

**Definition 7.**

```
sObjHeader obj ≡
sle32 (magic_f obj) @
sle32 (crc_f obj) @
sle64 (sqnum_f obj) @
sle32 (len_f obj) @
[bilbyFsPadByte] @ [bilbyFsPadByte] @ [trans_f obj] @ [otype_f obj]
```

*sObjHeader* takes an object as argument and converts it to *U8 list* by following the $Obj_T$ record definition and serialising field by field the object and appending the lists of bytes using Isabelle's list append operator *@*. *bilbyFsPadByte* is the value we used as padding in BilbyFs, so we set the bytes 0x14 and 0x15 to *bilbyFsPadByte*. *sle32* and *sle64* encodes 32-bit and 64-bit machine words into a list bytes, providing the opposite functionality of their parsing counter-parts *ple32* and *ple64*. We use lemmas like the one that follows to prove the connection between serialising and parsing functions:

```
pObjHeader (sObjHeader obj @ xs) 0 = obj(|ounion_f := undefined, offs_f := 0|)
```

This lemma says that serialising an object *obj* and parsing the buffer at offset 0 returns that same object, modulo the fields $ounion_f$ and $offs_f$ which have a constant value.

We use the same idiom to specify parsing and serialising abstract functions for each file system object. Figure 6.8 shows the call graph of parsing functions. *pObj* parses file system objects and is defined using *pObjHeader*. Depending on the type of the object parsed in the header, *pObjUnion* calls the parsing function for a specific file system object type. As the call graph shows, all the objects are made of primitive types encoded in little-endian via the functions *ple16*, *ple32* and *ple64*.

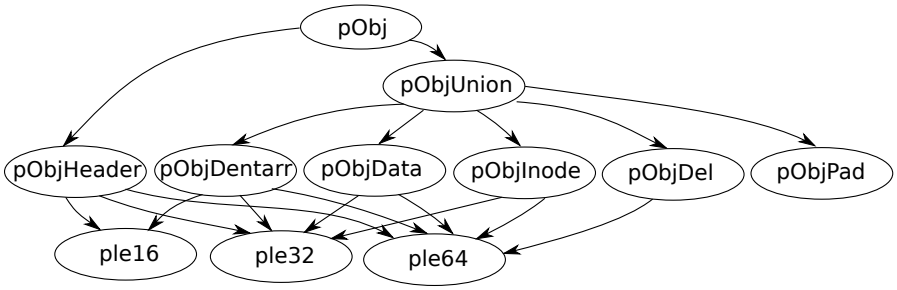


**Figure 6.8:** Object parsing call graph

*pObj* is unspecified when the buffer does not contain a valid object header. Hence, we define a predicate *is_valid_ObjHeader* to check whether an object header is valid. *is_valid_ObjHeader* checks include making sure the object's length is within the bound of the buffer and the type of the object is known. In the next paragraph, we use *is_valid_ObjHeader* to define a function that recursively parses objects in a sequence to form a transaction.

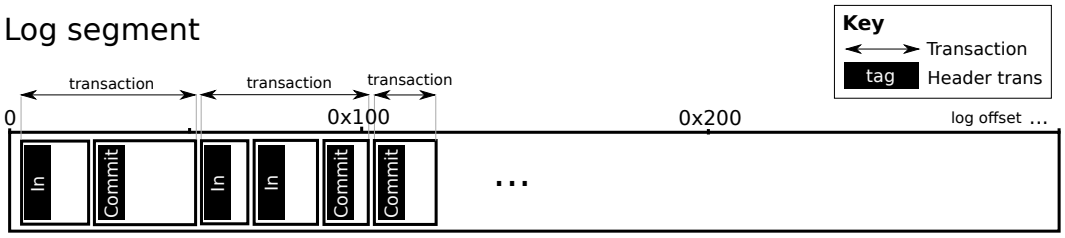


**Figure 6.9:** Log segments transactions made of objects with incomplete and commit tags

**Parsing transactions** Figure 6.9 shows an overview of BilbyFs' transactions encoding, where transactions are made of a sequence of objects and the last object header marks the end of the transaction with a commit tag. The transaction tag is stored in the $trans_f$ field of object headers. The $trans_f$ field of a valid object header may be of one of two values: *bilbyFsTransIn* to indicate that the transaction is incomplete, or *bilbyFsTransCommit* to mark the end of a transaction. In order to convert an array of bytes into a transaction of objects, we define *pTrans* as follows:

**Definition 8.**

**fun**

```
pTrans :: "U8 list ⇒ (U8 list × Trans)"
```

**where**

```
"pTrans data =
   (let obj = pObj data 0
    in if ¬is_valid_ObjHeader obj data then
      (drop (max (unat bilbyFsObjHeaderSize) (unat (Obj.len_f obj))) data, [])
    else if Obj.trans_f obj = bilbyFsTransIn then
      (λ(d,os). (d, (obj#os))) (pTrans (drop (unat (Obj.len_f obj)) data))
    else
      (drop (unat (Obj.len_f obj)) data, [obj]))"
```

*pTrans* takes a buffer as argument and returns a pair with the remaining part of the buffer and the transaction parsed (i.e. a list of objects). If the object parsed at offset zero of the buffer has an invalid header, we return an empty transaction. The use of *drop*, which removes at least the length of an object header from the beginning of the buffer, ensures that the length of the buffer is always decreasing, which in turns allows us to call *pTrans* recursively to parse all the transactions in a buffer without worrying about termination (see Definition 9 below). Otherwise, when the object header is valid, we check whether the object is the transaction is completed by comparing the field $trans_f$ to *bilbyFsTransIn*. When the transaction is incomplete, we call *pTrans* recursively on the rest of the buffer and we add the object to the list of objects forming a the transaction (*Trans* is a type synonym for $Obj_T$ *list*). When the transaction is complete, we terminate the recursion by returning the the rest of the buffer and a transaction with a single object.

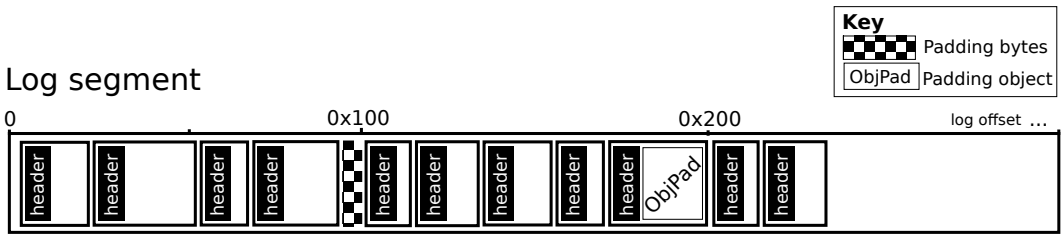


**Figure 6.10:** Example of log segment with padding bytes and padding objects

We use *pTrans* to define the function *list_trans* that recursively parses all the transactions in a buffer and returns them packed in a list. In addition, we use *list_trans* to ignore the padding between transactions. Inspired by the UBIFS on-flash format, BilbyFs supports two types of padding: *padding objects* and *padding bytes*. Figure 6.10 illustrates a log segment with the two types of padding. In this example the page-size is 256 bytes (i.e. 0x100 bytes) and both kinds of padding fill the segment up to the next page-size offset in the segment.

*Padding objects* are just another type of file system object that are used to add padding in the log in order to be able to write complete pages to the flash. When the amount of padding to fill a flash page with is too small to fit an object header, we use *padding bytes* to pad individual bytes by setting them to the value `bilbyFsPadByte`.

The definition of the Isabelle function `list_trans` follows:

**Definition 9.**

```
function
  list_trans :: "U8 list ⇒ (U8 list × Trans list)"
where
"list_trans data =
    (case pTrans data of
      (_, []) ⇒ (data, [])
    | ([],objs) ⇒ ([], [objs])
    | (newdata, objs) ⇒
       let (d,txs) = list_trans (nopad newdata)
       in (d,objs#txs))"
```

`list_trans` parses a transaction using `pTrans` and aborts when the transaction parsed is empty. When the first element of the pair returned by `pTrans` is empty, we have reached the end of the buffer. Otherwise, we call `list_trans` recursively with the buffer stripped of its leading padding bytes thanks to the `nopad` function which is defined as follows:

```
nopad xs ≡ dropWhile (op = bilbyFsPadByte) xs
```

`dropWhile` removes elements from the beginning of a list as long as they satisfy the condition taken as first argument. `bilbyFsPadByte` is the value used for padding bytes, so `nopad` makes sure that a list is not prefixed by padding bytes. Note that BilbyFs on-flash format guarantees that object headers' first byte value (defined by the constant value `bilbyFsMagic`) differs from `bilbyFsPadByte`. We also define `list_trans_no_pad`, a wrapper around `list_trans` that filters out all transactions composed of a single padding object.

When BilbyFs functions correctly and no system crash occurs, all log segments on flash as well as the write-buffer should be free of invalid transactions. So we must specify what a valid log segment is. We do so by writing two functions: `trans_len` to calculate the size of a transaction; and `valid_trans` to check the validity of a transaction. `valid_trans` checks the validity of each object header in the transaction, including that the value of the object's $\texttt{trans}_f$ fields is either `bilbyFsTransIn` or `bilbyFsTransCommit`. Then in order to ensure the validity of a complete log segment, we define a predicate `valid_log_segment`

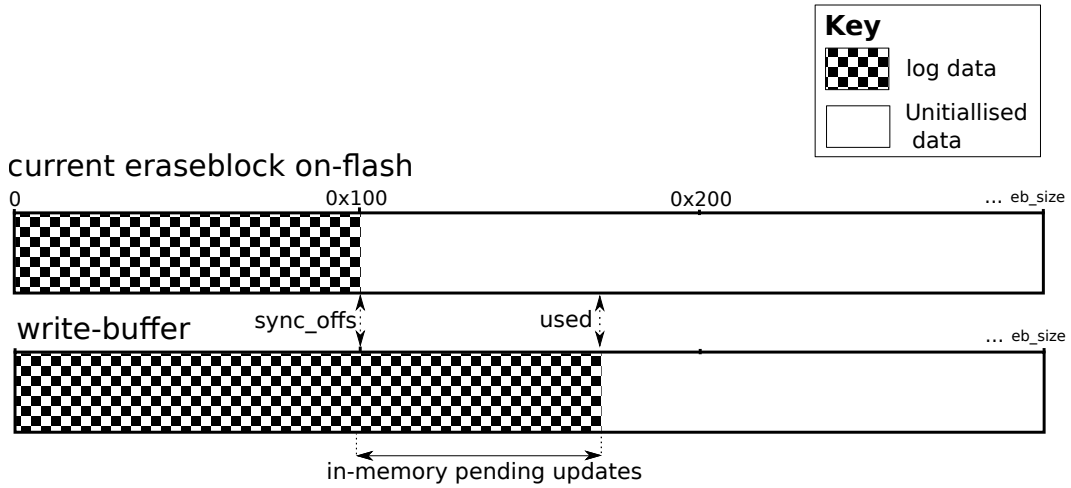


**Figure 6.11:** BilbyFs' write-buffer description

which ignores padding between transactions and ensures that all the transactions are valid by calling `valid_trans` recursively while truncating the buffer to length `trans_len`. We use `valid_log_segment` in the refinement relation to specify the ObjectStore's internal invariant which we describe in the next sub-section.

### Internal invariant of the ObjectStore

We focus on the ObjectStore invariant because it is the most interesting part of BilbyFs' invariant. As opposed to the *external* one which asserts properties on the ObjectStore abstraction exposed to the ObjectStore's clients, the *internal invariant* talks about the COGENT state (i.e. implementation) of the ObjectStore. It restricts the set of values the internal ObjecStore's state can have and ensures that the fields have consistent values with each other; without it, we cannot prove that the ObjecStore's implementation satisfies its axiomatic specification. The internal invariant is need to prove that the ObjectStore's implementation satisfies its axioms.

Figure 6.11 illustrates the write-buffer and describes the fields of the ObjectStore's state. At the top we show the contents of the current eraseblock on flash, i.e. the eraseblock to which the write-buffer is synchronised (e.g. when `sync()` is called). At the bottom, we show the write-buffer which caches in memory the contents of the current eraseblock plus some extra log data which encodes transactions describing each in-memory pending update. The $\texttt{sync\_offs}_f$ field of the ObjectStore's state indicates up to where the contents of the current eraseblock and the write-buffer are synchronised. The $\texttt{used}_f$

field, on the other hand, indicates up to where the write-buffer contains log data. Data in the write-buffer beyond *sync_offs*$_f$ (as depicted in Figure 6.11) has not yet been synchronised to flash when *sync()* is called, the write buffer and the current eraseblock are synchronised by overwriting the latter with the contents of the former. Since flash memories only allow one to write data at page-size granularity, *sync_offs*$_f$ must be aligned to the size of a flash page, whereas *used*$_f$ may be an arbitrary value but must be greater than or equal to *sync_offs*$_f$. Both fields must not exceed the field *eb_size*$_f$ of the mount state, which records the length of an eraseblock.

To specify the well-formedness of the log data stored in the write-buffer, we define a HOL function *buf_slice* that extracts a range of bytes from a buffer. Then we use *buf_slice* to specify that part of the buffer synchronised is a valid portion of log:
*valid_log_segment (buf_slice (wbuf*$_f$ *ostore_st) 0 (sync_offs*$_f$ *ostore_st))*
where *wbuf*$_f$ *ostore_st* is the write-buffer. Similarly, the internal invariant of the ObjectStore also contains:
*valid_log_segment (buf_slice (wbuf*$_f$ *ostore_st) (sync_offs*$_f$ *ostore_st) (used*$_f$ *ostore_st))*
From these two facts we are able to deduce that the entire log data in the write-buffer is a valid log using the rule *valid_log_segment_append*:

$$\frac{\textit{valid\_log\_segment xs} \qquad \textit{valid\_log\_segment ys}}{\textit{valid\_log\_segment (xs @ ys)}}$$

This rule is proved from the definition of *valid_log_segment*, rather than assumed as an axiom. The ObjectStore internal invariant also asserts *valid_log_segment* on all the non-empty eraseblocks of the UBI volume. When we prove that the ObjectStore's of *ostore_sync* operation correctly satisfies its axiomatic specification, we need to show that updating the contents of the flash makes the ObjectStore abstraction that only considers the contents of the flash equal to the in-memory ObjectStore abstraction with all the pending updates applied. Since the ObjectStore abstract is constructed by parsing the contents of the flash and the write-buffer, we need to know that they all constitute valid log segments.

This completes the description of the most interesting elements of the internal invariant of the ObjectStore. In the next sub-section, we describe the high-level abstract representation of the ObjectStore, used for the external invariant.

### Abstract representation of the ObjectStore

The interface presented by the ObjectStore (as described in its axiomatic specification, which we discuss later in Section 6.6) is characterised by three abstraction func-

tions: $\alpha$*_ostore_medium*, which presents an abstract view of the ObjectStore, by only considering the contents on flash; $\alpha$*_updates*, which presents an abstract view for the in-memory updates of the ObjectStore that have not yet been synchronised to flash; and $\alpha$*_ostore_uptodate*, which combines the two preceding abstractions to present a unified view of the ObjectStore state as it would be, were the in-memory updates applied to the flash. It is over the abstraction implemented by these three functions that the ObjectStore's external invariant is defined. We provide the complete definition of $\alpha$*_ostore_medium* (Definition 11) and other abstract functions in Section 6.6, when we discuss the *ostore_sync* axiomatic proof. Here, we only give enough details to describe the external invariant of the ObjectStore.

We hinted at the abstract representation of the ObjectStore in Section 3.3. The ObjectStore was designed such that its functionality is easily captured with a partial mapping from object identifier to object; hence we define the type *ostore_map* as:

**type_synonym** *ostore_map = "ObjId* $\rightharpoonup$ *Obj*$_T$*"*

In order to construct this *ostore_map* abstraction from the BilbyFs' state in COGENT, we define $\alpha$*_ostore_medium*, a function that logically mimics BilbyFs' *mount()* operation to parse the entire log on flash, and returns the abstract view of the ObjectStore only considering the contents permanently stored on flash.

Since the ObjectStore provides an asynchronous interface to write file system objects, its abstraction must capture this behaviour and separate the in-memory and on-flash state of the ObjectStore. Hence, analogously to the list of file system updates of the AFS, the ObjectStore abstraction exposes a list of *ObjectStore updates* representing the updates in the in-memory write buffer that have not been written to flash yet.

$\alpha$*_ostore_uptodate* takes the *ostore_map* abstraction returned by $\alpha$*_ostore_medium* and apply all the updates in $\alpha$*_updates* to it, by folding over the list of updates and applying them sequentially, similarly to the AFS presented in Section 5.3 of Chapter 5. As a result, $\alpha$*_ostore_uptodate* returns the *ostore_map* as visible by a client of the ObjectStore interface.

### External invariant of the ObjectStore

The external invariant of the ObjectStore *inv_*$\alpha$*_ostore* is used to prove the correctness of file system operations implemented on top of the ObjectStore. It is simpler than the internal one because the internal invariant talks about implementation details whereas the external one talks about the ObjectStore abstraction presented in the previous sub-section.

The external invariant *inv_α_ostore* asserts facts about the objects stored in the ObjectStore using the *α_ostore_uptodate* such as:

- Objects in the object store must be of one of the following types: inode, data block or dentarr.
- Objects identifiers of the *ostore_map* must be consistent with the object identifier stored in the object.
- If an object is of type dentarr, the following must hold:
  - the number of entries must be greater than zero but must not exceed the maximum number of entries allowed in a dentarr.
  - No two entries in the dentarr have the same name.
  - None of the names stored in the entries exceed the maximum length for a name in BilbyFs.

The external invariant captures all the information about the ObjectStore abstraction needed when proving file system operations on top of the ObjectStore (i.e. the FsOperations component of the modular decomposition in Figure 6.1). Next we explain the superblock invariant, before outlining BilbyFs modular proofs in Section 6.6.

### Superblock invariant

The superblock is a file system data structure read at mount time which records information about the flash device such as the size of an eraseblock, the size of a flash page, etc. The superblock does not change at run-time, hence we decomposed the state in such a way that isolating the superblock is easy so that the superblock can have its own separate invariant. This way, the ObjectStore operations take the superblock as read-only, meaning that we do not need to prove that the superblock invariant is maintained by the ObjectStore,

At mount time, BilbyFs reads the superblock and ensures that each of its fields is consistent with the values that can be retrieved from the UBI interface. Recall from Section 3.3, that UBI is a flash memory abstraction BilbyFs is implemented on top of.

The superblock is a record with a field $io\_size_f$ recording the size of a flash page and a field $eb\_size_f$ recording the size of an eraseblock in bytes.

The superblock invariant *inv_superblock*, ensures the following:

- The flash page size is a power of two.

- The flash page size is greater than 512.
- The eraseblock size is divisible by a flash page size.
- The eraseblock size is lower than `bilbyFsMaxEbSize` which is 16MB.

These conditions are needed to specify the constraints on UBI operations and to specify properties on the write-buffer. For instance, since the ObjectStore's write-buffer is written to flash up to the value of the field $sync\_offs_f$, the internal invariant asserts that $sync\_offs_f$ is divisible by the flash page size.

All these conditions were extracted from the sanity checks in UBI's source code, except for the maximum size of an eraseblock, for which we chose a value we deemed reasonable, because the specification by the Open Nand-Flash Interface (ONFi) consortium [ONFi] does not specify a maximum number of pages per eraseblock.

This completes the overview of the refinement relation `afs_fsop_rel` that we use to instantiate our correspondence statement with. In the next section we outline the modular proof of `iget()` and `sync()`.

## 6.6 BilbyFs' modular proof

At the beginning of this chapter, we presented our methodology for verifying BilbyFs modularly. In this section we relate our experience applying this methodology to prove the functional correctness of `iget()` and `sync()`. We follow the modular decomposition shown earlier in Figure 6.1, to reason about each component of the implementation in isolation. We prove the correctness of the file system operations by relying on the axiomatic specification of the ObjectStore. Then we prove that the ObjectStore satisfies its axiomatic specification by relying on the specification of the Index, the FreeSpaceMngr, and the Serial components, each of which in turn prove to satisfy their axiomatic specification.

### FsOperations: file system operations component

`iget()` **proof** To prove the functional correctness of `iget()`, we need to prove the following correspondence statement:

**Theorem 2** (corres_iget)**.**

`afs_fsop_rel afs_st fs_st` $\Longrightarrow$

```
cogent_corres
 (λ(avnode, ra) (fsr, rc).
     ra = rc ∧
     afs_fsop_rel afs_st (fs_st_f fsr) ∧ avnode = αinode (vnode_f fsr))
 (afs_iget afs_st inum (αinode vnode))
 (fsop_iget (make ex fs_st inum vnode))
```

It says that assuming `afs_st` is related to `fs_st` via `afs_fsop_rel`, which is the refinement relation described in the previous section, running `fsop_iget` and `afs_iget` with an inode related via the $\alpha$`_inode` abstraction, leads to results related via the lambda expression which we explain below. $\alpha$`_inode` converts the `VfsInode` type taken as argument by `fsop_iget` into a `vnode` type: its abstraction at the AFS level.

The lambda expression instantiates the relation `R` of the correspondence statement `cogent_corres` (see Definition 4 of Section 6.3), it is a function that takes two pairs of arguments. The first pair carries the values returned by `afs_iget`: the updated `vnode` `avnode`; and the value `ra` indicating success or failure of the operation. The second pair is the corresponding values at the implementation level. The first element of the pair is `fsr`, a record containing the updated external state `ex`, the updated `fs_st` and the updated `vnode`. And `rc`, the second part of the pair, is the value indicating the result of the operation. The lambda expression asserts that `afs_iget` and `fsop_iget` must return the same result for `ra` and `rc`, that the unchanged `afs_st` still relates to the updated `fs_st` state, and that the `vnode` returned by `afs_iget` is the same as abstracting the updated `vnode` contained in `fsr`. Note that, as explained in the previous section, `afs_fsop_rel` includes all the invariants, hence proving this correspondence statement also asserts that the invariants are preserved by `iget()`.

We presented the AFS of `iget()` in Section 5.4; the specification essentially says that if the inode number exists in the AFS logical representation of the file system, `afs_iget` must initialise the `vnode` passed as argument with the values of the inode present in the logical representation of the file system.

The `corres_iget` proof is straightforward and follows the structure of the implementation. We apply the proof rules presented in Section 6.3 and rely on the ObjectStore's axiomatic specification for `ostore_read` to progressively prove the correspondence between each statement in the specification and its implementation.

**`ostore_read`'s axiomatic specification** We describe `ostore_read`'s axiomatic specification. `ostore_read` reads the object matching the object identifier passed as argument from the ObjectStore, and returns `eNoEnt` when the object is not found or some other

error code when an unexpected error occurs. To better understand the specification, recall *ostore_read*'s COGENT prototype:

```
ostore_read: (ExState, MountState!, OstoreState, ObjId) ->
  ((ExState, OstoreState), <Success Obj | Error ErrCode>)
```

*ostore_read*'s first argument is the external state which (as explained in Section 4.1) helps us to model the behaviour of functions outside the semantics of COGENT like memory allocation. The state of the file system initialised at mount time and unmodified at run-time is packed into the *MountState* record and is passed as a shareable argument. ObjectStore's state is passed as the third argument and, like the *ExState*, is returned, meaning it may be updated by the function. And finally, the last argument is the object identifier, selecting which object to read from the ObjectStore.

The axiomatic specification for *ostore_read* follows:

**Lemma 1** (ostore_read_ret)**.**

```
⟦inv_ostore mount_st ostore_st; inv_superblock mount_st;
 inv_α_ostore (α_ostore_uptodate ostore_st);
 ⋀ex' ostore_st' obj.
    ⟦inv_ostore mount_st ostore_st';
     α_ostore_uptodate ostore_st' oid = Some obj;
     inv_α_ostore (α_ostore_uptodate ostore_st');
     α_ostore_medium ostore_st' = α_ostore_medium ostore_st;
     α_updates ostore_st' = α_updates ostore_st⟧
    ⟹ P ((ex', ostore_st'), Success obj);
 ⋀ex' ostore_st' e.
    ⟦inv_ostore mount_st ostore_st';
     e ∈ {eIO, eNoMem, eInval, eBadF, eNoEnt};
     (e = eNoEnt) = (α_ostore_uptodate ostore_st' oid = None);
     inv_α_ostore (α_ostore_uptodate ostore_st');
     α_ostore_medium ostore_st' = α_ostore_medium ostore_st;
     α_updates ostore_st' = α_updates ostore_st⟧
    ⟹ P ((ex', ostore_st'), Error e)⟧
⟹ P (ostore_read (ex, mount_st, ostore_st, oid))
```

The rule describes the effect of *ostore_read* through the ObjectStore's abstraction. The abstraction is obtained by the projection *α_ostore_medium*, that returns the abstract representation of the ObjectStore, and *α_updates* that returns the abstract representation of ObjectStore's updates; both were introduced in Section 6.5. The *α_ostore_medium*

projection takes the ObjectStore's state *ostore_st* and constructs an abstract mapping of type *ostore_map* (*ObjId* $\rightharpoonup$ $Obj_T$). By contrast, the other projection returns a list of *ostore_map* transformers (of type *(ostore_map* $\Rightarrow$ *ostore_map) list*), analogous to the list of pending updates at the AFS level described in Section 5.2. Additionally, $\alpha$*_ostore_uptodate* takes the *ostore_map* abstraction returned by $\alpha$*_ostore_medium* and applies all the pending updates returned by $\alpha$*_updates*.

The file system state is decomposed in multiple parts, and each part has its own invariant specifying the consistency requirements on its fields. The ObjectStore's internal invariant is captured by the first assumption of the *ostore_read* axiomatic specification: *inv_ostore*. Since the ObjectStore state is also decomposed into multiple parts including the Index, the FreeSpaceMngr and UBI, each part has its own invariant embedded within *inv_ostore*. The superblock invariant is captured by the next assumption: *inv_superblock*. The external invariant, which asserts the consistency requirements on the *ostore_map* abstraction of the ObjectStore, is captured by third assumption: *inv_*$\alpha$*_ostore*. Both the internal and external invariants are preserved by *ostore_read* regardless of the result of the operation.

*ostore_read* may modify the ObjectStore's state[6], however, its axiomatic specification ensures that any modification performed on the state must keep it consistent by preserving the invariants, and that modifications keep the abstract representation of the ObjectStore unchanged.

*ostore_read*'s axiomatic specification is structured as a rule that transforms the goal into two sub goals: one for when *ostore_read* succeeds, and another when *ostore_read* returns an error. The higher-order predicate *P* is used to match the surrounding context (i.e. the program text surrounding the call to *ostore_read*. In the success case, we obtain a new ObjectStore state *ostore_st'* and an object *obj*. *obj* is the object obtained by looking up the up-to-date abstract representation of the ObjectStore (i.e. with all the pending updates applied). Both the internal invariant and the external one hold on *ostore_st'*, and the abstract ObjectStore representation, including pending updates, must be the same as the one of *ostore_st*.

Similarly, when *ostore_read* returns an error, we obtain a new ObjectStore state *ostore_st'* with the same properties described for the success case except for one point: the object identifier passed as argument does not exist in the up-to-date *ostore_map*

[6] Since *ostore_read* merely reads an object from the ObjectStore, it is legitimate to wonder why would the function need to update the ObjectStore's state? As it turns out, *ostore_read* modifies the state when it reads from the storage device, because it temporarily stores the data read in a buffer that is part of the ObjectStore's state.

abstraction, iff the error code returned by the function is `eNoEnt`. Additionally, the axiom asserts that the error code returned must be one of the following values:

- `eIO`: input/output error when accessing the storage device.
- `eNoMem`: memory allocation error.
- `eInval`: invalid argument.
- `eBadF`: invalid data read from storage device.
- `eNoEnt`: described above.

In summary, the `ostore_read` axiom specifies that the ObjectStore's state may be modified during a `ostore_read` call but clients of the interface must not see the difference. This is enough to prove the correctness of any operation that reads a file system object from the ObjectStore. The specification describes `ostore_read` functionality directly on the abstract representation of the ObjectStore, regardless of the details of the underlying implementation.

The only proof obligations of `iget()`'s functional correctness proof that required manual intervention to help the proof assistant were discharged by deriving facts from the ObjectStore's external invariant and AFS' invariant (both are part of the `afs_fsop_rel` assumption). The ObjectStore's external invariant asserts that an object identifier of type inode maps to an inode object, so we wrote a lemma to extract this fact from the invariant. Similarly, we proved another lemma to infer from the `afs_fsop_rel` assumption that there is an inode matching the `inum` argument in the AFS state if and only if there is one in the ObjectStore. These two lemmas and a mindful use of Isabelle/HOL's rewriting engine were enough to complete the `corres_iget` proof.

**`sync()` proof** The correspondence statement for `sync()` shows a lot of similarity with `iget()`'s one:

**Theorem 3** (corres_sync)**.**

```
afs_fsop_rel afs_st fs_st ⟹
cogent_corres
 (λ(afs, ra) (fsr, rc). ra = rc ∧ afs_fsop_rel afs (fs_st_f fsr))
```

```
(afs_sync afs_st) (fsop_sync (make ex fs_st))
```

We make the same assumption: *afs_st* and *fs_st* must be formally related via the *afs_fsop_rel* relation. The lambda expression relating the results of both *afs_sync* and *fsop_sync* is simpler; it asserts that the return values must always match and that the updated versions of *afs_st* and *fs_st* must still be related with *afs_fsop_rel*.

We only summarise *fsop_sync*'s functionality briefly as we presented *afs_sync* specification in Figure 5.5 of Section 5.4, and showed its implementation at the beginning of this chapter in Figure 6.2 of Section 6.2. *fsop_sync* checks whether the read-only flag is set, if so we return the *eRoFs* error code. Otherwise, we unpack the file system state *fs_st* and call *ostore_sync* to apply all the ObjectStore's pending updates. When *ostore_sync* returns an error, we repack the file system state, propagate the error code, and set the read-only flag if a critical I/O error happened (i.e. *err* equals *eIO*). When *ostore_sync* succeeds, we repack the state and return *Success*.

The proof follows the implementation's structure, applying the proof rules presented in Section 6.3 to match implementation statements to specification ones. The most interesting part of the proof is the application of the *ostore_sync* axiom. *ostore_sync* synchronises the in-memory state of the ObjectStore with the medium. More specifically, it flushes out to flash the in-memory write-buffer that contains the pending updates of the ObjectStore. *ostore_sync*'s COGENT prototype follows:

```
ostore_sync:(ExState, MountState!, OstoreState, U32) ->
  ((ExState, OstoreState), <Success () | Error ErrCode>)
```

The prototype is the same as *ostore_read*'s one except for the last argument of the function, which is a set of flags encoded in a 32-bit machine word value. One of flags is used for instance to force *ostore_sync* to fill up the rest of the write-buffer with padding entirely before synchronising and allocating a new eraseblock for the next segment of the log.

*ostore_sync*'s axiomatic specification is similar to *ostore_read*'s one in several aspects, hence we focus on the differences between the two specifications.

**Lemma 2** (ostore_sync_ret)**.**

```
⟦inv_ostore mount_st ostore_st; inv_superblock mount_st;
 inv_α_ostore (α_ostore_uptodate ostore_st);
 ⋀ostore_st' ex'.
```

```
    ⟦inv_ostore mount_st ostore_st';
     inv_α_ostore (α_ostore_uptodate ostore_st');
     α_ostore_medium ostore_st' = α_ostore_uptodate ostore_st;
     α_updates ostore_st' = []⟧
    ⟹ P ((ex', ostore_st'), Success ());
 ⋀e ostore_st' ex' n.
    ⟦inv_ostore mount_st ostore_st';
     inv_α_ostore (α_ostore_uptodate ostore_st');
     e ∈ {eIO, eNoMem, eNoSpc, eOverflow}; n < |α_updates ostore_st|;
     α_ostore_medium ostore_st' =
     apply_n_updates n (α_ostore_medium ostore_st) (α_updates ostore_st);
     α_updates ostore_st' = (drop n $ α_updates ostore_st)⟧
    ⟹ P ((ex', ostore_st'), Error e)⟧
⟹ P (ostore_sync (ex, mount_st, ostore_st, ostoreWriteNone))
```

Like *ostore_read*'s axiomatic specification, *ostore_sync*'s is structured as a rule that transforms the goal into two sub goals: one for when *ostore_sync* succeeds, and another for when *ostore_sync* returns an error. The higher-order predicate *P* is used to match the surrounding context (i.e. the program text surrounding the call to *ostore_sync*. In the success case, the specification gets a new ObjectStore state *ostore_st'* with all the updates applied such that the abstract representation of the new *ostore_map* only considering the on-flash contents (*α_ostore_medium ostore_st'*) is the up-to-date *ostore_map* of the old ObjectStore state (*α_ostore_uptodate ostore_st*). The new list of pending updates projected from *ostore_st'* is empty, since all the updates must have been applied.

In the more interesting error case, the axiomatic specification allows an arbitrary number of pending updates to have been applied (i.e. any prefix of the list). *n* specifies the number pending updates applied, *n < |α_updates ostore_st|* ensures that it is less than the total number of pending updates. The new abstract representation of the ObjectStore *α_ostore_medium ostore_st'* is the same as applying the first *n* pending updates to the old ObjectStore state. The list of pending updates is the same as the old one but with the first *n* pending updates removed, as specified by *α_updates ostore_st' = drop n (α_updates ostore_st)*.

The semantics of *ostore_sync* are aligned with the ones allowed by the AFS of *sync()*, assuming we can relate the AFS to the ObjectStore when they have the same number of pending updates applied consecutively. In *afs_sync*, the number of updates applied in the error case is non-deterministically selected using the choice function *select* of the non-deterministic monad presented in Section 5.1.

The *corres_sync* proof has to show that all execution paths of the implementation have a corresponding one in the specification. The fragment of the proof that deals with the non-determinism proceeds as follows. First we apply the *ostore_sync* axiom, which fixes the number of updates applied in the error case, then we use the *COGENT_select* rule presented in Section 6.3 to pick a concrete value within the set of non-deterministic choice allowed by the specification. We have to show that the value we selected is indeed allowed by the specification, before we proceed to finish the correspondence proof with the selected value.

For the *corres_sync* proof, showing that the selected list of pending updates applied is a non-deterministic choice allowed by the specification simply consists of showing that taking the *n* first elements of a list returns a prefix of the list, and dropping *|α_updates ostore_st| - n* elements returns a suffix of the list. Such proofs are discharged automatically using Isabelle/HOL's lemmas library for lists.

**Discussion** Both correspondence statements for *iget()* and *sync()* were straightforward to prove using our refinement framework. This is not surprising because we benefit significantly from dealing only with the high-level abstraction of the ObjectStore to read and write file system objects. *iget()* proof is merely ≈120 lines of Isabelle/HOL proof-script and ≈100 for *sync()*'s. Although, the high-level logic of the operation *iget()* is simple and the complexity of proving the *sync()* operation is delegated to the *ostore_sync* proof, the economy shown by the size of these proofs still highlights the benefit of designing a file system modularly with interfaces that follow the design principles of information hiding [Parnas, 1972] and separation of concerns [Dijkstra, 1982]. Proving the correctness of the file system logic on-top of the ObjectStore abstraction is much easier than if we were to reason directly about its implementation. At this level, details about how file system objects are stored on flash are completely abstracted away and the logic of operations that manipulate these objects is confined into the *FsOperation* component. In the next subsection we address the ObjectStore axiomatic specification proof.

### ObjectStore: object store component

We saw that *iget()* relied on the *ostore_read* axiom and *sync()* on *ostore_sync*'s one to complete the top-level proof of correctness. This section explains how we proved that both these functions satisfy their axiomatic specifications.

**`ostore_read` axiomatic proof** We only briefly summarise the control flow of the `ostore_read` operation in this section, because its implementation was presented in Figure 4.8 of Section 4.4. `ostore_read` uses the object identifier passed as argument to look-up the index which stores the addresses of objects on flash. Addresses of objects are kept in the index as a record that includes eraseblock number, offset within the eraseblock and the length of the object. If the object identifier is not found in the index, `ostore_read` returns the error code `eNoEnt`. Otherwise, we check whether the object happens to be located in the current eraseblock, i.e. the eraseblock where the in-memory write-buffer of the ObjectStore is being synchronised to. If so, we de-serialise the object from the write-buffer. Otherwise, we read all the flash pages overlapping the object's data on flash into the read-buffer of the ObjectStore. At this point, the read-buffer is partly initialised, since only the page-size chunks of the eraseblock with data belonging to the object are filled with data read from the flash. Then, we de-serialise the object from the read-buffer, specifying the offset position indicating where to de-serialise from. We also specify a bound on the buffer to ensure we do not access uninitialised parts of the buffer. Finally, as a sanity check, we compare the requested object identifier to the value stored in the object we read, before returning the object.

The axiomatic specifications of the Index and the FreeSpaceManager components are similar to the WordArray axiomatic specification presented in Section 6.4. Both their functionality are captured by abstraction functions. The Index abstraction is a partial mapping from object identifier to an address on flash (i.e. an eraseblock number, an offset in the eraseblock and the length of the object). Since we implemented the Index using a red-black tree (and our axiomatic specification for red-black trees already describes the effects of the tree operations on a partial mapping), the proof that the Index correctly implements its axiomatic specification was trivial. Similarly, the FreeSpaceManager is implemented using the COGENT ADTs WordArray and red-black tree, and its axiomatic specification makes use of these ADTs abstractions to describe the component's functionality. As the proof that both these components satisfy their axiomatic specifications was straightforward, we focus on the more challenging part of `ostore_read`'s proof in the rest of this section.

In order to prove that objects are read correctly by the file system (and *ostore_read* in particular), we need the right specification for object de-serialisation. The specification for the *deserialise_Obj* function called by *ostore_read* is:

**Lemma 3** (deserialise_Obj_ret)**.**

```
⟦wellformed_buf buf; |α $ data_f buf| ≤ unat bilbyFsMaxEbSize;
 offs + bilbyFsObjHeaderSize ≤ bound_f buf;
 offs < offs + bilbyFsObjHeaderSize;
 ⋀ex e. e ∈ {eInval, eNoMem} ⟹ P (Error (e, ex));
 ⋀ex obj offs'.
    ⟦is_valid_ObjHeader (pObjHeader (α (data_f buf)) offs) (α (data_f buf));
     obj = pObj (α (data_f buf)) offs; offs' ≤ offs + len_f obj;
     let end_offs = offs + len_f obj;
        data = take (unat end_offs) (α (data_f buf))
     in dentarr_entries_len_le_end_offs (otype_f obj) data offs end_offs⟧
    ⟹ P (Success (ex, obj, offs'))⟧
⟹ P (deserialise_Obj (ex, buf, offs))
```

The *deserialise_Obj* axiom is part of the Serial axiomatic specification. The details of the Serial component axiomatic proof are presented in Appendix A.1. The specification uses the HOL parsing function *pObj* described in Section 6.5 to specify the decoding of a file system object from a buffer. *deserialise_Obj* COGENT prototype follows:

```
deserialise_Obj: (ExState, Buffer!, U32) ->
   <Success (ExState, Obj, U32)| Error (ErrCode, ExState)>
```

The only unfamiliar type is *Buffer*, it is defined as a record made of a *WordArray U8* where the actual data is stored and a *U32* specifying a *bound* on the buffer. The bound field, ensures that if a buffer is only initialised up to a specific offset, i.e. the bound, deserialisation operations do not read beyond that bound.

*deserialise_Obj* takes a shareable read-only buffer and an offset as argument. When successful, it returns the external state, the object de-serialised and an offset (namely *offs'*) specifying the byte position in the buffer right after the object de-serialised. *offs'* must be lesser than or equal to the initial offset plus the length of the object de-serialised, i.e. *end_offs*. When unsuccessful, *deserialise_Obj* simply returns an error code and the external state. *deserialise_Obj* takes the external state as argument because it allocates objects.

*deserialise_Obj* relies on two assumptions on the input buffer. First, the buffer must be well-formed, and its bound must not exceed the length of its data array. Second, the maximum size of the data array must not exceed *bilbyFsMaxEbSize*, the maximum eraseblock size allowed by the UBI flash memory abstraction on top of which BilbyFs is implemented. We also need to know there is enough room in the buffer to de-serialise at least an object header, and that the offset does not overflow when we add the object header size to it.

Given all these assumptions, the axiomatic specification says that calling *deserialise_Obj* may result in one of two possible outcomes. If an error occurs, then all we know is that the error code returned is either *eInval* when the function detected an invalid object header, or *eNoMem* when a memory allocation failed. When *deserialise_Obj* succeeds, the axiom asserts that the object header at the position *offs* in the buffer is valid, that the object returned by the function is equivalent to the result of the HOL abstraction *pObj*[7] at this offset, and that the offset returned by *deserialise_Obj* is positioned after the object just de-serialised in the buffer. Finally, the assumption *dentarr_entries_len_le_end_offs* which we describe shortly.

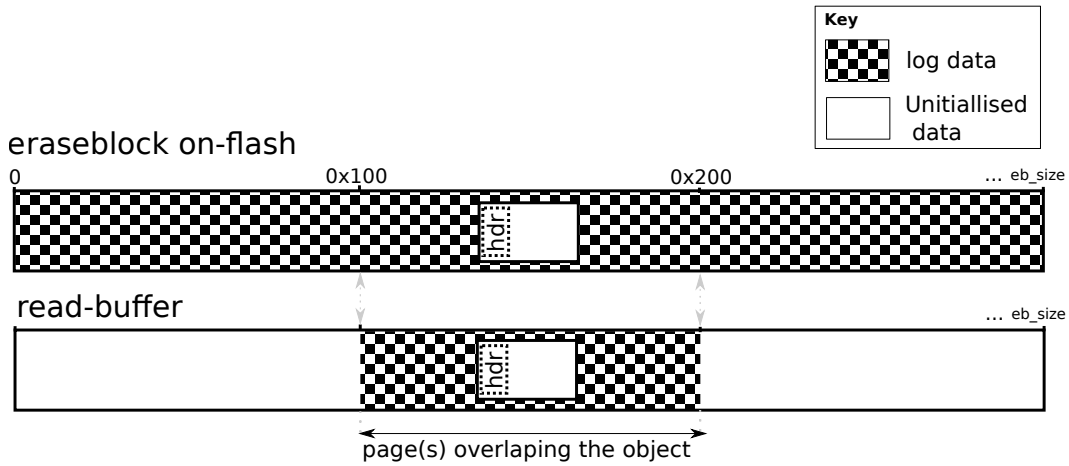


**Figure 6.12:** Reading an object from an eraseblock on flash

The major proof obligation for the *ostore_read* axiomatic proof requires showing that parsing an object from a buffer where only the relevant part of the eraseblock is loaded in memory is equivalent to parsing the object on the entire eraseblock. This proof obligation

[7] *pObj* was introduced in Section 6.5, and makes use of *pObjHeader* and *pObjUnion* to parse an object header and its payload. For completeness, we present outline proof of *deserialise_ObjHeader*'s axiomatic specification in Appendix A.1.

arises because our ObjectStore internal invariant talks about entire eraseblocks. The internal invariant says that for every address kept in the index, there is a valid object header on flash (or in the write-buffer) and that the object returned by the ObjectStore corresponds to calling `pObj` on the (entire) eraseblock at the offset recorded in the index. Yet, as illustrated by Figure 6.12, for efficiency reasons `ostore_read` only reads into the read-buffer the flash pages that contain parts of the object it needs to de-serialise.

This means that at some point in the `ostore_read` proof we have to prove a lemma like the following:

```
slice (unat frm) (unat to) entire_eb = slice (unat frm) (unat to) read_buffer
⟹ pObj entire_eb frm = pObj read_buffer frm
```

Where slice is a function that returns the sub-list from offset `frm` to `to` and `unat` converts 32-bit word to natural number. We need to show that if the relevant part of the entire eraseblock (`slice (unat frm) (unat to) entire_eb`) is equal to the part read from the flash (`slice (unat frm) (unat to) read_buffer`), then the `pObj` function return the same result. We prove this by showing that `pObj` only accesses elements of the input list that are between `frm` and `to`.

Specifying the extent of the buffer read by `pObj` is complicated by dynamically sized file system objects, particularly directory entry arrays (dentarrs) because each directory entry encodes its own length and the length of the previous entry is used to jump to the next entry when `pObjDentarr` parses them (called by `pObj`; see Figure 6.8). This is the reason for the `dentarr_entries_len_le_end_offs` predicate we alluded to earlier.

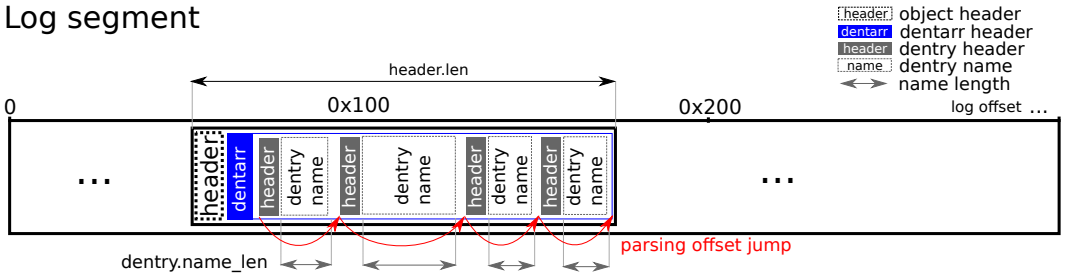


**Figure 6.13:** Parsing object dentarr where each directory entry stores its own length.

Figure 6.13 illustrates the recursive parsing of directory entries. A directory entry array is encoded after the object header, and starts with a dentarr header which stores the number of directory entries in the array. Then each directory entry has a small directory entry header colored in grey, which stores the length of the name following the directory entry header. So parsing the array involves reading a directory entry header and using

the length of the name to jump to the next directory entry in the array.

The *dentarr_entries_len_le_end_offs* predicate tells us that when we successfully de-serialise an object from a buffer, if that object is of type dentarr, recursively parsing each directory entry in the array never leads to accessing an area of the buffer beyond the length of the object stored in its header. *dentarr_entries_len_le_end_offs* is defined recursively, and it asserts that for each directory entry the length of the current entry added to the length of all the previous directory entries does not go beyond the length of the buffer. Note that this is not equivalent to asserting that the sum of all the directory entry lengths plus the current buffer position is less than its length, because it guarantees the absence of overflow when the additions are performed on 32-bit words.

This gives us enough information to prove that parsing the object on the partially initialised read-buffer is equivalent to parsing the object on the entire erase-block, allowing us to complete the proof of the *ostore_read* axiom.

*ostore_sync* **implementation** *ostore_sync* writes all the pending updates on flash, this involves adding padding to the write-buffer such that it contains enough data to write complete flash pages and handling potential errors.

Before we outline the proof of the *ostore_sync* axiom we show the COGENT specification of *ostore_sync*:

**Definition 10** (ostore_sync_def)**.**

```
ostore_sync ds_0 ≡
let (ex, mount_st, ostore_st, osw_flags) = ds_0;
   sync_offs = sync_offs_f ostore_st; used = used_f ostore_st
in if sync_offs = used ∧ ¬ is_set (osw_flags, ostoreWriteNewEb)
  then ((ex, ostore_st), Success ())
  else let pad_to = padding_to (mount_st, ostore_st, osw_flags);
         (ds_12, r) = prepare_wbuf (ex, mount_st, ostore_st, pad_to);
         (ex, ostore_st) = ds_12
       in case r of Error e ⇒ ((ex, ostore_st), Error e)
          | Success v_G2 ⇒
              let (ds_21, r) =
                    ostore_sync_summary_if_eb_new
                     (ex, mount_st, ostore_st, osw_flags);
                  (ex, ostore_st) = ds_21
              in case r of Error e ⇒ ((ex, ostore_st), Error e)
                 | Success v_G1 ⇒
```

```
let nb_bytes = used_f ostore_st - sync_offs;
    (ds_33, r) =
      ostore_write_buf
       (ex, mount_st, ostore_st, sync_offs, nb_bytes,
        osw_flags);
    (ex, ostore_st) = ds_33
in case r of Error e ⇒ ((ex, ostore_st), Error e)
   | Success v_G0 ⇒
       let used = used_f ostore_st;
          ostore_st = ostore_st(|sync_offs_f := used|)
       in ((ex, ostore_st), Success ())
```

*ostore_sync* takes four arguments: the external state, the mount state, the ObjectStore state and a set of flags *osw_flags*. Recall from Section 6.5 that the write-buffer is kept consistent with several fields of the ObjectStore state in accordance with the internal invariant *inv_ostore*. As shown by Figure 6.11, the write-buffer which is of the size of an eraseblock, the $sync\_offs_f$ field indicates the offset up to where the write-buffer is synchronised with the on-flash content of the eraseblock. The $used_f$ field is the position in the buffer up to where the write-buffer has been written to in memory. Therefore, the range of the buffer between the offsets $sync\_offs_f$ and $used_f$ contains the pending updates of the ObjectStore.

The implementation of *ostore_sync* checks whether there are no pending updates and the flag *ostoreWriteNewEb* is not set, and returns *Success* if so. When $sync\_offs_f$ equals $used_f$, it means that no data has been written to the write-buffer since it was last synchronised to flash. The *ostoreWriteNewEb* flag is used by *ostore_write* to force *ostore_sync* to allocate a new eraseblock. The code path when *ostoreWriteNewEb* is set true has not been verified, since when *ostore_sync* is called by *fsop_sync* this flag is not set. Otherwise, when $sync\_offs_f$ and $used_f$ differ, the invariant guarantees that $sync\_offs_f$ is less than $used_f$. We call *padding_to* to calculate the offset up to which we need to add padding to the write-buffer. Adding padding may be necessary to avoid writing partially filled pages of data to the flash device. We pass the result of *padding_to* to *prepare_wbuf*, which modifies the write-buffer to add padding to it. *prepare_wbuf* may return an error when it fails to find a unique transaction number for the padding transaction. When it succeeds, it updates the write-buffer and the $used_f$ field in the ObjectStore's state. The next step is to update the write-buffer by serialising the eraseblock summary at the end of the buffer, when it is getting full. This is done by *ostore_sync_summary_if_eb_new* which can return an error or succeed. Then, in the success case, we call *ostore_write_buf* to

write the buffer to the eraseblock used to store the write-buffer on-flash. `ostore_write_buf` too may return an error and when it does, the error code is propagated. Otherwise, we set the $sync\_offs_f$ field to $used_f$'s value to indicate that the buffer is synchronised with its on-flash eraseblock. Note that we re-read the $used_f$ field from the ObjectStore's state, because `prepare_wbuf` may have modified it.

**`ostore_sync` axiomatic proof** The non trivial parts of the `ostore_sync` proof were to specify what a valid log segment is, showing that writing the write-buffer to flash is equivalent to applying all the pending updates on the abstract representation of the ObjectStore. We explained how to specify a valid log segment in Section 6.5 with the predicate `valid_log_segment`.

Showing that the write-buffer remains valid when we add padding to it is relatively easy once we have such specification. However, as mentioned earlier in Section 6.5, BilbyFs supports two types of padding: padding objects and padding bytes, the latter of which is used when the area to fill is smaller than the size of an object header.

Padding objects are merely another type of object, and form a normal transaction with a single object in the log that are just ignored by `mount()`, so they do not require any special handling. We serialise them just like any file system object and we need to prove that they produce a valid transaction in the buffer. This is guaranteed by the axiomatic specification of the serialisation function `serialise_Obj`, which says that when we call `serialise_Obj (buf, offs, obj)` to serialise an `obj` in `buf` at offset `offs`, the slice of the buffer between `offs` and $offs + len_f\ obj$ is replaced with the value returned by `sObj obj`. We introduced `sObj` in Section 6.5, and it is a HOL function that returns an object encoded as a list of bytes. Padding objects also need to be ignored by specifications that talk about transactions in a log segment. `list_trans`, the HOL function that recursively parses transactions in the log and returns all the transactions found, including those made of a single padding object. Any specification that requires ignoring padding objects uses the wrapper `list_trans_no_pad` that post-processes the list of transactions returned by `list_trans`, to filters out transactions composed of a single padding object.

Padding bytes, on the other hand, are directly ignored by `valid_log_segment` and `list_trans`. As seen in Section 6.5, both `valid_log_segment` and `list_trans` use the `nopad` HOL function to eliminate padding bytes from the beginning of a list of bytes.

The function `prepare_wbuf` invoked by `ostore_sync` adds a padding object or padding bytes to the write-buffer depending on the size of the area to pad. To deal with the padding in `ostore_sync`'s axiom proof, we had to prove a lemma akin to the following:

```
snd (list_trans_no_pad buf_with_padding) = snd (list_trans_no_pad buf)
```

This says that calling `list_trans_no_pad` on the write-buffer with and without padding returns to the same list of transactions, where the `buf_with_padding` is the write-buffer returned by `prepare_wbuf` thus it potentially includes padding bytes or a padding object. From such lemma, we conclude that `prepare_wbuf` does not alter the list of non-padding transactions in the write-buffer, and that the padding added does not affect the ObjectStore's abstract representation.

The heart of proving correct synchronisation (i.e. of the `ostore_sync` proof) requires showing that updating the ObjectStore's state with the write-buffer completely written to flash and the $\texttt{sync\_offs}_f$ field updated to $\texttt{used}_f$ is equivalent to applying all the pending updates on the abstract representation of the ObjectStore. This corresponds to the assertion $\alpha$`_ostore_medium ostore_st' =` $\alpha$`_ostore_uptodate ostore_st` of the `ostore_sync` axiom when the function succeeds. Recall that `ostore_st'` is the new ObjectStore state and `ostore_st` the one `ostore_sync` was invoked with. $\alpha$`_ostore_medium`'s definition follows:

**Definition 11** ($\alpha$_ostore_medium)**.**

```
α_ostore_medium ostore_st ≡
let alltrans = concat (list_eb_log (αwubi (ubi_vol_f ostore_st)));
   alltrans' = sort_key trans_order alltrans
in fold id (map ostore_update alltrans') empty
```

The $\alpha$`wubi` function projects an abstract representation of the UBI volume, returning a list of eraseblocks, where an eraseblock is a list of bytes. `list_eb_log` takes the abstract UBI volume as argument and returns all the segments (i.e. eraseblocks) that belong to the log and applies `list_trans_no_pad` on them to obtain a list of lists of transactions, which we flatten into a list of transactions. Then we sort all transactions by transaction number using `sort_key trans_order`, and convert the transactions into a list of map transformers using `ostore_update`. `ostore_update` takes a transaction of objects and returns a transformation of type `ostore_map` $\Rightarrow$ `ostore_map`, like the one introduced in Section 5.3 for the AFS. `ostore_update` transformations are functions that take an `ostore_map` and update it by iterating over each object of the transaction. For each object, the transformation adds a mapping from the object identifier to the object, or if the object is a deletion object, it removes the range of objects specified by the object's deletion pattern, as explained in Section 3.3. Finally, since the list of map transformer is sorted by transaction number, $\alpha$`_ostore_medium` can sequentially apply

each map transformer by calling `fold id` on the list and an empty map, to obtain the abstract view of the ObjectStore only considering the contents on flash. $\alpha$`_ostore_medium` projects a `ostore_map` from the ObjectStore's state by logically mimicking the `mount()` operation[8].

The overall idea of the proof is that the `ostore_map` obtained by writing the extra part of the write-buffer containing the pending updates and then parsing the entire log is the same as parsing the entire log and applying the pending updates while still volatile in memory. This is true because all the transactions in the pending updates have a higher transaction number than any transaction currently in the log, and because conceptually `mount()` (and indirectly $\alpha$`_ostore_medium`) sorts the list of transactions by transaction number before applying them. In addition, we also need to know that there is a one-to-one mapping between transactions and transaction numbers, i.e. the function that extracts a transaction number from a transaction is injective. Otherwise, two transactions having the same number lead to ambiguity as to which one should be applied first. These properties are guaranteed by the ObjectStore's internal invariant.

**Discussion** Using our modular proof methodology, we were able to reason about the ObjectStore's correctness in isolation: the components on which it relies, namely Serial, UBI, Index and FreeSpaceMngr functionality are solely described by their axiomatic specifications. The ability to abstract the on-flash format and reason abstractly about binary data stored on flash was key to verifying the ObjectStore component. Verifying error handling requires proving that the invariants still hold, and we found that often it seems we are proving the same property multiple times but in fact the property is slightly different. For instance, in the `ostore_sync` proof, when `prepare_wbuf` succeeds, we have to prove that most of the `inv_ostore` invariant is preserved since the ObjectStore state is slightly different from before the call, and as we follow the control flow, each function call updates the state slightly, requiring another invariant proof. None of these proofs are particularly difficult, but are time consuming, and often require writing a lemma that

[8] The actual `mount()` implementation is optimised to read eraseblocks summaries instead of the entire log, and it only falls back to parsing the contents of a segment when a summary is missing, which could be caused by the system crashing for instance. Despite `mount()` implementation being much more complicated than the abstraction we just described, its functionality is captured by it. In addition, with a similar framework for reasoning about system crashes as Chen et al. [2015a], the current $\alpha$`_ostore_medium` definition could be used for specifying and verifying crash-tolerance properties of BilbyFs. In this thesis we mainly focus on the functional correctness of run-time operations, we leave the `mount()` refinement proof and any crash-tolerance related properties for future work.

is proved by unfolding the relevant definitions and leveraging Isabelle/HOL automated tactic to discharge the parts of the invariant that make use of $\forall$ or $\exists$ quantifiers.

### UBI axiomatic specification

In Section 3.3, we introduced UBI, the flash memory abstraction on top of which BilbyFs is implemented. In this sub-section, we present the high-level abstraction we use to describe UBI's axiomatic specification. As explained in the rest of this section, some of the formal assumptions we make about UBI are not fully realistic, for instance we assume that when we write a buffer to an eraseblock, either the entire buffer is written or none, without considering potential bit-flip errors that flash memories are subject to [Tseng et al., 2011]. Additional work can make these assumption fully realistic, but we leave this for future work. Our specification of UBI is straightforward; and shares some similarities with the specifications the eraseblock manager of Pfähler et al. [2013] and the NAND flash specification of Butterfield et al. [2009]. For completeness we detail our axiomatic specification for UBI in Appendix A.2.

## 6.7 Evaluation

In this section, we give an overview of the bugs we found during the proofs of `iget()` and `sync()`, and analyse the effort required to determine whether the overall methodology, for designing, implementing and verifying file systems is practical. In particular, we want to know whether it leads to a reduction in verification effort compared to traditional systems software verification. To this end, we use the seL4 verification as a point of comparison, and find that we achieve a reduction in effort of at least a third compared to traditional software verification in C.

### Bugs

The verification found six defects in the already tested and benchmarked BilbyFs COGENT implementation. Three of these occurred in serialisation functions. The first bug was a missing check to ensure that the transaction flag of an object is one of two values: `bilbyFsTransIn` for an incomplete transaction and `bilbyFsTransCommit` for committing a transaction. The second one was the omission of an overflow check for a value read from medium. The last serialisation bug is more interesting, so we devote the next few paragraphs to it.

When we deserialise a dentarr object, each entry in the array encodes the length of the name it stores. So after decoding the entry we had the following check:

offs $<$ offs + bilbyFsObjDentryHdrSize + name_len

Where *name_len* is the length of the name stored in the directory entry, and *bilbyFsObjDentryHdrSize* is the size of the fixed-length header for each entry. The intent of the check was to ensure that no overflow is possible when we add *name_len* to *offs*. When we perform the check above, we know for fact that offs $<$ offs + bilbyFsObjDentryHdrSize. Hence, we mistakenly assumed we could derive from these two facts the following:

offs + bilbyFsObjDentryHdrSize $<$ offs + bilbyFsObjDentryHdrSize + name_len

However, during the proof, we realised we cannot because the check only prevents *name_len* to overflow when *bilbyFsObjDentryHdrSize* is added to it. But if, for instance, *name_len* equals the maximum value that fits in a 32-bit word, since *bilbyFsObjDentryHdrSize* is greater than one, the check is true despite the right-hand-side of the inequality overflowing. The fix consisted of using the following check instead:

offs + bilbyFsObjDentryHdrSize $<$ offs + bilbyFsObjDentryHdrSize + name_len.

Serialisation and de-serialisation are mechanical and tedious to write, which makes them prime candidates for further language and proof generation support.

The three other bugs were found in the *sync()* implementation. Two of them were caused by not restoring the ObjectStore's state properly when an error occurs, and the bugs were found because we could not prove the internal invariant of the ObjectStore was preserved. The last one was found when verifying the top-level correspondence refinement step of the *sync()* operation. We forgot to set the file system in readonly-mode when a critical I/O error happens, therefore the execution path where the error occurred in the implementation was not allowed by the specification.

## Proof modularity

As explained in the previous section, BilbyFs' functional correctness proofs were done by following the modular decomposition of the implementation. We found that we could divide the effort and work on separate components in isolation. For instance, the Serial component's proof was done in part by Christine Rizkallah and Yutaka Nagashima who had no familiarity with BilbyFs' internals. All of us were able to prove functionalities of BilbyFs that rely on one another without interfering with each other. This was greatly facilitated by our design and verification methodology presented in this thesis, where each implementation components communicate through well defined interfaces that are described by abstractions, exposing just enough information to capture the functionality

of the component, without detailing the underlying implementation. Our specification of these interfaces has been surprisingly stable, none of the axiomatic specifications experienced major changes. We did find a few typos, for instance our `ostore_read` axiom was referencing the `ostore_st` instead of `ostore_st'` in one occurrence. The axiom that required the most significant change was the `deserialise_Obj` one where, we did not anticipate that we needed to specify the extent of the buffer read by `pObj` to finish the `ostore_read` proof.

### Proof productivity

The effort for verifying the complete chain from the FsOperations component to the UBI axiomatic specification 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/deserialisation 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 [Klein et al., 2014], 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. Although this calculation indicates a significant gain in productivity when reasoning about a purely functional COGENT-generated specifications, we should keep in mind a few caveats when considering these numbers.

First, COGENT restrictions on data sharing makes it unsuitable for implementing a high-performance micro-kernel where several optimisations require memory aliasing. Second, micro-kernels like seL4 are tightly coupled systems where performance is so critical that they cannot be designed modularly. By contrast, in this thesis we have shown file systems can be designed modularly, without incurring unacceptable performance overhead. Finally, the seL4 functional correctness proof is complete for the entire micro-kernel, whereas BilbyFs's proof establishes the functional correctness of operations `iget()` and `sync()` only. This means that BilbyFs's invariant might be incomplete and might need modifications to include missing information required to prove the functional correctness of other file system operations. Klein et al. [2014] reported experiencing multiple omissions

of this kind for seL4.

By considering the effort per line of the 1,350 lines of verified COGENT code and the remaining 2,150 lines unverified, we estimate completing the proof of BilbyFs would require an effort of ≈15 person months. However, we believe that taking into account the effort already spent on the specification of BilbyFs' abstractions and on reusable libraries for reasoning about COGENT code the effort would be closer to one person year.

## 6.8 Summary

In this chapter we presented our approach to prove modularly the functional correctness of a file system. We explained how the COGENT specifications generated by COGENT's certifying compiler are purely functional and allow reasoning about the file system logic using equational reasoning. We presented our methodology to decompose a functional correctness proof by following the modular decomposition of the implementation, and our refinement framework to prove that a high-level functional specification is refined by a COGENT-generated specification with a level of abstraction higher than traditional verification frameworks for C. The methodology is organised in two phases. First, we use our refinement framework to prove the top-level refinement step, showing that the high-level correctness specification of a file system operation is refined by the COGENT implementation. We strongly rely on the axiomatic specification of the ObjectStore, which is specified through a simple partial map abstraction. Second, we prove that the implementation of each component (including the ObjectStore), correctly satisfies its axiomatic specification. An axiomatic proof subsequently relies on the axiomatic specification of the components relied on by the implementation. We provided evidence of the practicality of our methodology by proving two operations of BilbyFs: `iget()` and `sync()`. Finally, we evaluated our approach to analyse its strengths and weaknesses, and to determine whether it increases verification productivity when compared to traditional C software verification techniques and found that it produces an increase of up to 50% compared to the seL4 verification.

# 7 Conclusion

In this thesis we have presented a methodology for designing, implementing and verifying realistic file systems.

In Chapter 3, we provided the key design ingredients required to make a file system verifiable. We presented a file system design that keeps the verification tractable by forcing sequential execution of file system operations, separating the virtual file system switch (VFS) caches interactions from the core file system logic, and pursuing aggressive modular decomposition of the file system functionality. With BilbyFs, we have shown that when designed carefully, a file system can not only be performant, but also verifiable. In order to keep the verification effort manageable, we chose a simple log structured design for BilbyFs, where the indexing information of the log is kept only in memory and must be reconstructed at mount time. BilbyFs supports asynchronous writes, a crucial performance optimisation that has been overlooked by several file system verification projects in the past. We have shown that BilbyFs' modular design does not incur unacceptable performance overhead: its runtime performance is on par with existing Linux flash file systems except for carefully designed benchmarks where concurrency is a bottleneck.

In Chapter 4, we presented COGENT — a linearly typed language we co-designed — to implement file systems that are much easier to reason about using a proof assistant than reasoning directly in C. We have described the main features that made the COGENT language sufficiently expressive to implement efficient file systems. In particular, the language needs to provide a way to bypass type restrictions when necessary, e.g. for accessing data structures that require memory aliasing (i.e. ADTs). We presented a set of design patterns for implementing a file system in a linearly typed language like COGENT in a way that enables decomposing the implementation into a set of components such that each of them can be reasoned about in isolation. We implemented BilbyFs in

COGENT to demonstrate the practicality of the language and to compare the performance of the COGENT implementation to the handwritten C prototype version introduced in Chapter 3. We showed that this implementation approach does not impose unacceptable performance overhead: COGENT's overhead is lower than 10% on most benchmarks and 25% in the worst case for benchmarks with intensive meta-data updates like postmark.

In Chapter 5, we showed how to specify the functional correctness of file systems in higher-order logic, including the correctness of asynchronous writes — a feature omitted from prior work on verified file systems. Specifically, we presented the formal correctness specification of BilbyFs called the AFS. The AFS uses non-determinism for conciseness, and is shallowly embedded in higher-order logic to facilitate reasoning about it using the Isabelle/HOL proof assistant. The AFS is a description of the correct behaviour of each file system operation as expected by the VFS. The AFS captures the semantics of asynchronous writes by separating the in-memory and on-medium file system state. This allows us to precisely specify the `sync()` operation.

In Chapter 6 we presented a lightweight modular verification technique for proving the functional correctness of file system operations while exploiting the file system modular decomposition. We demonstrated the practicality of this technique, and evaluated the utility of the overall methodology presented in this thesis for producing verified file systems at reasonable cost, by verifying the functional correctness of two file system operations of BilbyFs. Specifically, we used the AFS to prove the functional correctness of `iget()` and `sync()`. We demonstrated that purely functional COGENT specifications allow leveraging the implementation modularity to increase verification productivity and greatly simplify reasoning about the file system logic. Therefore, we showed that our approach for designing, implementing and verifying file systems is practical, and increases the productivity of building trustworthy file systems. A comparison with the seL4 verification showed an increase in verification productivity up to 50%.

## 7.1 Future work

We conclude this thesis by discussing three research challenges not directly addressed by our methodology that we deem interesting directions for future research.

First, our approach leaves out the verification of the VFS layer, the cache manager (the wrapper between the VFS and the COGENT file system implementation — see Chapter 3) and ADTs. Since these components are implemented in C, verifying them requires supporting cross-language verification between COGENT and C. Second, our methodology currently does not consider correctness guarantees related to crash-safety.

Finally, supporting concurrency and other more complicated file system designs. We discuss each of these research directions in a separate sub-section next.

### VFS, cache manager and ADTs verification

Our methodology left out the verification of the parts of the file system that cannot be implemented in COGENT and must be implemented in C. The VFS layer, the cache manager and ADTs, all extensively use data sharing to be implemented efficiently. The VFS layer and the cache manager would be hard to implement in COGENT because they are tightly coupled and interact with several parts of the Linux kernel like the virtual memory manager. For example, directory entries located in the directory entry cache contain C pointers to the inodes in the inode cache, which in turn reference memory in the page cache that holds memory pages of data blocks that belong to files and directories. The restrictions on memory aliasing in COGENT, makes the language unsuitable for implementing components with highly inter-dependent data structures like the VFS and the cache manager.

Currently, our verification rests on the validity of our ADT specifications and the assumption that they are correctly implemented. We also assume that the VFS layer and our cache manager are bugfree (we need to trust the rest of the Linux kernel too). To discharge these assumptions, we would need to prove that the C stubs are correct and have a formal link between parts of the system implemented in C and COGENT.

Previous research such as Feng et al. [2006]; Vaynberg and Shao [2012]; Gu et al. [2015] focus on designing verification frameworks that support cross-languages verification. However, since the COGENT certifying compiler generates C code, and all the properties we prove on COGENT hold on the C code by refinement, proving C stubs can be done directly using the C semantics used by the COGENT compiler. The COGENT certifying compiler works on top of AutoCorres [Greenaway et al., 2014] a tool that automatically abstracts low-level C semantics into higher-level representations and provides a Isabelle/HOL machine-checked proof that guarantees that the higher-level representation is an accurate model of the low-level C code. Hence, when reasoning about the implementation of C stubs, we can still benefit from a shallow embedding in Isabelle/HOL, but unlike COGENT specifications, we must deal with many of C's low-level details, such as memory aliasing, undefined behaviours, etc.

### Crash resilience proof

Although this thesis did not consider file system consistency when a crash occurs, we can reason about crash resilience on top of COGENT-generated specifications. Such reasoning would probably be easier with a crash-Hoare logic, akin to to the one presented in Chen et al. [2015a], to specify and prove properties about the file system when a crash occurs. Chen et al. [2015a]'s crash Hoare-logic is an augmented version of Hoare logic, where in addition to defining pre and post-conditions for a program statement, the logic allows one to specify a crash-condition encompassing the correct behaviour of the file system when a crash occurs. Using such a framework it is possible to prove that under any sequence of crashes followed by reboots, the file system will be able to recover without losing data.

Currently, our methodology does not use a Hoare logic, instead preferring more lightweight equational reasoning over the purely functional COGENT specifications. In this style of equational reasoning, preconditions are represented as formal assumptions about function arguments, and post conditions are simply formal assertions about the return values of functions. Designing a suitable analogue of this kind of reasoning for talking about program state and behaviour when a crash occurs is certainly feasible, but it remains to see whether the same compactness of expression achieved by Chen et al. [2015a] can be replicated in the absence of Hoare logic.

### High-performance journaling file systems

The methodology presented in this thesis was only applied to BilbyFs. However, an ext2 implementation was also written in COGENT by Alex Hixon and Peter Chubb. This shows that COGENT is expressive enough to implement other types of file systems than simple log structure flash file systems like BilbyFs. However, both BilbyFs and ext2 remains relatively simple file system implementations when compared to a full-blown journaling file system like ext3, ext4 or XFS or JFS. Unfortunately, these high-performance file systems have huge code bases with at least 30 K lines of C code each. The size of these file system implementations was driven by many factors including performance optimisations like extents, copy-on-write, concurrency, etc.; support for larger files as storage capacity increased; user requested features like extended attribute, quotas, snapshots, etc.; and backward compatibility.

Many of these features do not present fundamental challenges to our methodology, and we can reasonably expect that users of a fully verified file system would be willing to make practicability trade-offs, and hence not all these features would be absolutely necessary in

all domains. However, we can conceivably assume that adding concurrency will cause fundamental changes to the way we specify and reason about file system correctness.

At the moment we cannot implement such concurrent file systems with COGENT because it is a sequential language. Although it allows implementing file systems with fully asynchronous writes, it does not support full concurrency (i.e. where multiple file system operations are invoked concurrently). While a transition to concurrent language semantics would be non-trivial, we believe the linear type system of COGENT will help: because COGENT programs pass the file system's state explicitly, it lets the compiler keep track of memory locations and side effects. Hence, we could allow the COGENT functions that access different parts of the state to run concurrently, and provide additional synchronisation primitives that make more fine-grained concurrency available for functions that are likely bottleneck to the concurrent execution of file system operations. One problem here is the external state, which one may want to decompose to allow different threads to access different bits of Linux kernel concurrently.

## 7.2 Concluding remarks

The goal of this thesis was to demonstrate that an implementation-level proof of correctness of a file system can be done at a reasonable cost. As we have argued, our methodology for building trustworthy file systems accomplishes that goal. We hope that the work presented in this thesis will be used in the future to build fully verified file systems incorporated into mainstream operating systems.

# A Modular proof details

## A.1 Serial: serialisation/de-serialisation proofs

We serialise file system objects to a buffer using the function *serialise_Obj*. and we deserialise them using *deserialise_Obj*. As explained in Section 6.5, *serialise_Obj* calls *serialise_ObjHeader* to serialise the header of the object and *serialise_ObjUnion* to serialise its payload. All serialisation and de-serialisation routines follow the same idiom. Since *serialise_Obj* is trivially implemented by calling *serialise_ObjHeader* and *serialise_ObjUnion*, here we focus on the more interesting proofs of *serialise_ObjHeader* and *deserialise_ObjHeader*.

### Serialisation

The following function serialises an object header in a buffer.

**Definition 12** (Serialising an object header)**.**

```
serialise_ObjHeader ds_0 ≡
let (buf, offs, o_r) = ds_0; buf = serialise_le32 (buf, offs, magic_f o_r);
   buf = serialise_le32 (buf, offs + 4, crc_f o_r);
   buf = serialise_le64 (buf, offs + 8, sqnum_f o_r);
   buf = serialise_le32 (buf, offs + 0x10, len_f o_r);
   buf = serialise_u8 (buf, offs + 0x14, bilbyFsPadByte);
   buf = serialise_u8 (buf, offs + 0x15, bilbyFsPadByte);
   buf = serialise_u8 (buf, offs + 0x16, trans_f o_r);
   buf = serialise_u8 (buf, offs + 0x17, otype_f o_r)
in (buf, offs + bilbyFsObjHeaderSize)
```

The function serialises each field of the object header, starting at the position of the offset *offs* passed as argument. The functions *serialise_le16*, *serialise_le32*, *serialise_le64* serialises machine words in little endian as explained in Section 6.5. *serialise_ObjHeader* and its HOL abstraction *sObjHeader* (see Definition 7) follow the record definition of BilbyFs' object header introduced in Definition 6,

The axiomatic specification of *serialise_ObjHeader* follows:

**Lemma 4** (serialise_ObjHeader_ret)**.**

```
⟦offs ≤ offs + bilbyFsObjHeaderSize;
 is_valid_ObjHeader obj (drop (unat offs) (αwa (data_f buf)));
 ⋀buf' offs'.
    ⟦offs' = offs + bilbyFsObjHeaderSize;
     buf' = buf
     (|data_f :=
        make
         (buf_sub_slice buf offs (offs + bilbyFsObjHeaderSize)
           (sObjHeader obj))|)⟧
    ⟹ P (buf', offs')⟧
⟹ P (serialise_ObjHeader (buf, offs, obj))
```

The specification assumes that the offset does not overflow when added the object header size *bilbyFsObjHeaderSize*, and that *obj* has a valid object header, with the *is_valid_ObjHeader* assumption. Lemma 4 specifies that *serialise_ObjHeader* returns a pair of buffer and offset, where the buffer is updated with the object header is serialised, as encoded by *sObjHeader*, at offset *offs*, and the offset returned is the initial offset plus the object header size.

The proof that *serialise_ObjHeader* satisfies Lemma 4 was straightforward. We use lemmas to convert *serialise_ple16*, *serialise_ple32* and *serialise_ple64* to their simpler HOL abstraction. To show that none of the offset calculations used for serialisating fields of the object header can lead to a word overflow, we use Lemma 4's first assumption and invoke Isabelle's tactic to solve word arithmetic proof goals.

## De-serialisation

To outline the de-serialisation proofs, we present the definition of *deserialise_ObjHeader*.

**Definition 13** (De-serialising an object header)**.**

```
deserialise_ObjHeader ds₀ ≡
let (buf, offs, obj) = ds₀; magic = deserialise_le32 (buf, offs);
   crc = deserialise_le32 (buf, offs + 4);
   sqnum = deserialise_le64 (buf, offs + 8);
   len = deserialise_le32 (buf, offs + 0x10);
   trans = deserialise_u8 (buf, offs + 0x16);
   otype = deserialise_u8 (buf, offs + 0x17);
   obj = obj
     (|magic_f := magic, crc_f := crc, sqnum_f := sqnum, offs_f := offs,
        len_f := len, trans_f := trans, otype_f := otype|);
   end_offs = len + offs
in if end_offs < offs ∨
     bound_f buf < end_offs ∨
     magic ≠ bilbyFsMagic ∨
     trans ≠ bilbyFsTransIn ∧ trans ≠ bilbyFsTransCommit ∨
     ¬ is_len_and_type_ok (otype, len)
   then (obj, Error 0x16) else (obj, Success (offs + bilbyFsObjHeaderSize))
```

The function de-serialises each field of the object header, initialises the object `obj` passed as argument, and performs several sanity checks to ensure that the header is valid (i.e. satisfy the checks performed by Definition 8). When it passes the sanity checks, the de-serialised object header is returned together with an offset to the position in the buffer right after the header. Otherwise, the object and an error code is returned.

To prove the `deserialise_Obj_ret` lemma presented in the previous section, we proved the following lemma:

**Lemma 5** (deserialise_ObjHeader_ret)**.**

```
⟦wellformed_buf buf; offs + bilbyFsObjHeaderSize ≤ bound_f buf;
 offs < offs + bilbyFsObjHeaderSize; ⋀obj. P (obj, Error eInval);
 ⋀obj offs'.
    ⟦is_valid_ObjHeader (pObjHeader (αwa (data_f buf)) offs) (bounded buf);
     ∃v. obj(|ounion_f := v|) = pObjHeader (αwa (data_f buf)) offs;
     offs' = offs + bilbyFsObjHeaderSize; offs + len_f obj ≤ bound_f buf;
     offs < offs + len_f obj⟧
    ⟹ P (obj, Success offs')⟧
⟹ P (deserialise_ObjHeader (buf, offs, obj))
```

Which says that when `deserialise_ObjHeader` succeeds, the buffer contains a valid object header at the offset we de-serialised the object. Proving the lemma involves using

lemmas to convert de-serialise functions to their HOL parsing abstraction, e.g. replace `deserialise_le32` with `ple32`. The most interesting part of the proof was to find a way to automatically show that none the arithmetic operations on the offset overflow. The definition of `deserialise_ObjHeader` shows that we add a fixed constant value to the offset passed as argument every time we de-serialise a field. This trivial arithmetic operation complicates the proof because we have to show that none of them leads to an unsigned overflow[1]. We know that none of these operations overflow because the offset passed as argument must not be greater than the maximum length of an eraseblock, which is much smaller than the maximum value of a 32-bit word; and since any of the constant values plus the maximum length of an eraseblock does not overflow, any of the constant values plus the offset does not overflow either. We automated the proofs by writing an Isabelle tactic that carefully applies a bunch of rewriting rules to unfold the necessary information about the offset bound and then uses the Isabelle's tactic to solve word arithmetic proofs.

## A.2 UBI axiomatic specification

UBI is a volume management system, that allows partitioning a flash device into UBI volumes. UBI also provides an interface to write and erase to the flash without having to worry about wear-levelling. In COGENT, a UBI volume is accessed via an ADT called `UbiVol`. When reasoning about the generated COGENT specifications, we use `UbiVol` to project a simple representation of the state of the UBI volume as visible through the UBI interface. To this end, we axiomatize the following function:

**axiomatization**
$\alpha$`wubi :: "UbiVol ⇒ ubi_leb list"`

$\alpha$`wubi` projects a list of `ubi_leb` which is a type synonym for `U8 list`, i.e. a list of bytes which stores the concatenated contents of the flash pages written in each eraseblock. We chose a list of bytes instead of a list of flash pages, because the UBI interface already deals with buffers which are a list of bytes. Hence, using a list of pages would complicate the specification unnecessarily, given that we are able to specify all the page-size related restrictions on a buffer. The abstract representation of the UBI volume comes with an invariant ensuring that no flash page is partially written, no eraseblock is larger than the size of an eraseblock, etc.

[1] Unsigned overflow are allowed in COGENT, just like in C, however, in this context an overflow must not occur for the function to behave correctly.

Recall from Section 3.3 that eraseblock flash pages have to be written in sequential order and that the UBI interface maintains that restriction. UBI's behaviour is undefined if a file system does not write flash pages sequentially. Thus when we write to an eraseblock, we update our abstract representation merely by concatenating the buffer the `ubi_leb` representing the eraseblock.

Although the abstraction we just presented is simple, UBI's axiomatic specification is quite verbose due to dealing with records packing imposed by COGENT's foreign function interface (FFI) and due to the several pre-conditions required for the semantics of UBI operations to be defined.

A fragment of the UBI axiomatic specification follows:

```
axiomatization
where
 wubi_leb_read_ret:
 "⋀P. ⟦ buf_length rbuf = eb_size_f (super_f mount_st);
        unat buf_offs + unat nb_bytes  ≤ unat (eb_size_f (super_f mount_st));
        inv_ubi_vol mount_st ubi_vol;
        ⋀ex rbuf' . ∃v. rbuf'(|data_f:=v|) = rbuf ⟹ buf_length rbuf' = buf_length
rbuf ⟹ P ((ex,rbuf'), Error eBadF);
        ⋀ex rbuf'.  ⟦
  rbuf' = rbuf(|data_f:= WordArrayT.make (buf_take rbuf buf_offs @
            slice (unat buf_offs) (unat buf_offs + unat nb_bytes) (αwubi ubi_vol
!(unat ebnum) @ replicate (unat nb_bytes) 0xff) @
            buf_drop rbuf (buf_offs+ nb_bytes)) |)
        ⟧ ⟹ P ((ex,rbuf'), Success ())
  ⟧ ⟹ P (wubi_leb_read (WubiLebReadP.make ex ubi_vol ebnum rbuf buf_offs nb_bytes))"
and  wubi_leb_write_ret:
 "⋀P. ⟦ length ( (αwubi ubi_vol) ! (unat ebnum)) = unat frm;
        buf_length wbuf = eb_size_f (super_f mount_st);
        unat frm + unat nb_bytes  ≤ unat (eb_size_f (super_f mount_st));
        io_size_f (super_f mount_st) udvd nb_bytes;
        inv_ubi_vol mount_st ubi_vol;
        ⋀ex . P ((ex,ubi_vol), Error eIO);
        ⋀ex ubi_vol'.  ⟦ inv_ubi_vol mount_st ubi_vol';
        (αwubi ubi_vol') = αwubi ubi_vol[(unat ebnum):=(αwubi ubi_vol!(unat ebnum)@buf_slice
wbuf frm (frm + nb_bytes))]
        ⟧ ⟹ P ((ex, ubi_vol'), Success ())
  ⟧ ⟹ P (wubi_leb_write (WubiLebWriteP.make ex ubi_vol ebnum wbuf frm nb_bytes))"
```

The `wubi_leb_read_ret` axiom describes the behaviour of `wubi_leb_read`, the operation

to read part of an eraseblock into a buffer.

```
type WubiLebReadP = #{
  ex:ExState,
  ubi_vol:UbiVol!,
  ebnum:EbNum,
  buf: Buffer,
  buf_offs:BufOffs,
  nb_bytes:U32
}
wubi_leb_read: WubiLebReadP -> ((ExState, Buffer),
                                <Success ()|Error ErrCode>)
```

The function takes six arguments packed into an unboxed record of type *WubiLebReadP*: *ex*, the external environment of the file system; *ubi_vol*, the UBI volume; *ebnum*, the eraseblock number we want to read from; *buf* a buffer to read data into; *buf_offs*, the offset indicating where to start reading the eraseblock from; *nb_bytes*, indicating the number of bytes to read. The function returns a tuple including: the external environment of the file system and the buffer containing the data read; and a tagged-union recording whether the function succeeded or returned an error code.

Several conditions must be satisfied before BilbyFs can read an eraseblock, hence we specify them as assumptions of the *wubi_leb_read_ret* axiom. Some of these conditions are more restrictive than necessary, but since we do not prove that UBI axioms are satisfied by their implementation, we decided to take a conservative approach when specifying them to make sure that UBI axioms are implementable. The first two assumptions of the *wubi_leb_read_ret* axiom ensure that the operation does not write beyond the length of the buffer. $eb\_size_f$ *(super*$_f$ *mount_st)* corresponds to the size of an eraseblock, so the first assumption which ensures that the length of the buffer is also the length of the eraseblock.

Since the buffer only needs to be large enough to contain the data read, this condition is more restrictive than it needs to be, but this is enough prove the correctness of BilbyFs' operations that read from the flash. *inv_ubi_vol* is the invariant ensuring the abstract representation of the UBI volume is consistent with the state of the file system initialised at mount time. For instance, *inv_ubi_vol* asserts that the length of the list of eraseblocks in the abstract UBI representation is equal to the number of eraseblocks reported by the flash device. Similarly to specifications seen in Section 6.4, *wubi_leb_read_ret* forces us to consider success and error cases. When *wubi_leb_read* returns an error, the function

returns the error code `eBadF` and the contents of the buffer is unknown. This indicated by the existentially quantified variable `v` which replaces the contents of the buffer. In case of success, `wubi_leb_read` updates the contents of the buffer, replacing the part of the buffer requested with a slice of the eraseblock taken from the $\alpha$`wubi` abstraction. `buf_take` takes a buffer and a 32-bit word `n` indicating the number of bytes, and must return the list with the `n` first bytes from the buffer. `buf_drop` takes the same parameters, but returns the buffer's list of bytes with the `n` first bytes removed. Both `buf_take` and `buf_drop` are helper functions that first convert COGENT types to their high-level abstraction and return a prefix or a suffix of the list of the length passed as argument. Note that when we extract the slice of the eraseblock requested, we make sure to pad the eraseblock with `0xff` bytes because our UBI abstract representation of eraseblocks represents partially written eraseblocks as the list of bytes containing only the data that has been written. A UBI client is allowed to read flash pages that have never been written to, and when this occurs, UBI writes `0xff` bytes to the buffer.

`wubi_leb_write_ret` shares similarities with `wubi_leb_read_ret`. The arguments are the same, but `wubi_leb_write` does not update the buffer passed as argument, instead `ubi_vol` of type `UbiVol` is updated because the operation writes the flash device.

The first assumption, ensures that we always write sequentially to eraseblocks, by asserting that the length of the eraseblock identified by `ebnum` must be equal to `frm`, the offset we start writing from. Since $\alpha$`wubi` only contains the data that has been written to flash, the length of the `ubi_leb` tells us the next offset we are allowed to write from. The second and third assumptions are the same as for `wubi_leb_read`. The forth one ensures that we only write complete flash pages of data, as required by UBI. $\texttt{io\_size}_f$ `(super`$_f$ `mount_st)` corresponds to a flash page-size, and `udvd` is the `divides` operator for machine words. Hence the assumption ensures that page-size divides `nb_bytes`, guaranteeing that we write complete flash pages.

The specification forces us to consider two possibilities when we invoke `wubi_leb_write`, either the function returns an error or succeeds. When the function returns an error, the error code `eIO` is returned and the `ubi_vol` is unchanged; we return to this shortly. Otherwise, when the function succeeds, we obtain `ubi_vol'` a new `UbiVol` with an abstract representation capturing the changes of the write operation. The axiomatic specification uses the list update syntax, where `xs[a := b]` returns the list `xs` with the element at index `a` replaced with `b`. In addition, `xs!i` is syntactic sugar for accessing the `i`th element of the list `xs`. Thus, `wubi_leb_write` updates the eraseblock `ebnum` by appending the data stored in the buffer between the offsets `frm` and `frm + nb_bytes`, as specified with the `buf_slice` call.

# Bibliography

Ramnatthan Alagappan, Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Aws Albarghouthi, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Beyond storage APIs: Provable semantics for storage stacks. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*. USENIX Association, 2015.

Sidney Amani and Toby Murray. Specifying a realistic file system. In *Workshop on Models for Formal Analysis of Real Systems*, pages 1–9, Suva, Fiji, November 2015.

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, and Gernot Heiser. Cogent: Verifying high-assurance file system implementations. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 175–188, Atlanta, GA, USA, April 2016. doi:10.1145/2872362.2872404.

Konstantine Arkoudas, Karen Zee, Viktor Kuncak, and Martin C. Rinard. Verifying a file system implementation. In Jim Davies, Wolfram Schulte, and Michael Barnett, editors, *Proceedings of the 6th International Conference on Formal Engineering Methods*, volume 3308 of *Lecture Notes in Computer Science*, pages 373–390, Seattle, WA, USA, November 2004. Springer. doi:10.1007/978-3-540-30482-1_32.

David Aspinall and Martin Hofmann. Another type system for in-place update. In Daniel Le Metayer, editor, *Programming Languages and Systems*, volume 2305 of *Lecture Notes in Computer Science*, pages 36–52. Springer Berlin Heidelberg, 2002. ISBN 978-3-540-43363-7. doi:10.1007/3-540-45927-8_4. URL http://dx.doi.org/10.1007/3-540-45927-8_4.

Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. Thorough static analysis of device drivers. In *Proceedings of the 1st EuroSys Conference*, pages 73–85, Leuven, BE, April 2006.

Julian Bangert and Nickolai Zeldovich. Nail: A practical tool for parsing and generating data formats. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*, pages 615–628, 2014.

Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al. The Coq proof assistant reference manual: Version 6.1. 1997.

William Bevier, Richard Cohen, and Jeff Turner. A specification for the Synergy file system. Technical Report Technical Report 120, Computational Logic Inc., Austin, Texas, USA, September 1995. URL http://computationallogic.com/reports/files/120.pdf.

Egon Börger and Robert Stärk. *Abstract state machines: a method for high-level system design and analysis*. Springer Science & Business Media, 2012.

James Bornholt, Antoine Kaufmann, Jialin Li, Arvind Krishnamurthy, Emina Torlak, and Xi Wang. Specifying and checking file system crash-consistency models. pages 83–98, 2016.

Edwin Brady. Ivor, a proof engine. In *Implementation and Application of Functional Languages*, pages 145–162. Springer, 2006.

Edwin Brady. Programming and reasoning with algebraic effects and dependent types. In *Proceedings of the 18th International Conference on Functional Programming*, pages 133–144, Boston, MA, USA, 2013. ACM. doi:10.1145/2500365.2500581.

Andrew Butterfield, Leo Freitas, and Jim Woodcock. Mechanising a formal model of flash memory. *Science of Computer Programming*, 74(4):219–237, 2009.

Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M Frans Kaashoek. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, page 5. ACM, 2011.

Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Using Crash Hoare logic for certifying the FSCQ file system. In *ACM Symposium on Operating Systems Principles*, pages 18–37, Monterey, CA, October 2015a.

Haogang Chen, Daniel Ziegler, Adam Chlipala, M Frans Kaashoek, Eddie Kohler, and Nickolai Zeldovich. Specifying crash safety for storage systems. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, 2015b.

David Cock, Gerwin Klein, and Thomas Sewell. Secure microkernels, state monads and scalable refinement. In Otmane Ait Mohamed, Cesar Munoz, Solene Tahar, editor, *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics*, pages 167–182, Montreal, Canada, August 2008. Springer. doi:10.1007/978-3-540-71067-7_16.

Jonathan Corbet. Trees II: red-black trees. https://lwn.net/Articles/184495/, 2006. Accessed: 2016-03-09.

Thomas H Cormen, Charles E Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.

Kriangsak Damchoom. *An Incremental Refinement Approach to a Development of a Flash-Based File System in Event-B*. PhD thesis, University of Southampton, October 2010. URL http://eprints.ecs.soton.ac.uk/21655/.

Kriangsak Damchoom and Michael Butler. Applying event and machine decomposition to a flash-based filestore in Event-B. In *SBMF 2009*, volume 5902, pages 134–152, 2009. URL http://eprints.ecs.soton.ac.uk/18301/. Springer LNCS 5902.

Kriangsak Damchoom, Michael Butler, and Jean-Raymond Abrial. Modelling and proof of a tree-structured file system in Event-B and Rodin. In Shaoying Liu, Tom Maibaum, and Keijiro Araki, editors, *Proceedings of the 10th International Conference on Formal Engineering Methods*, volume 5256 of *Lecture Notes in Computer Science*, pages 25–44, Kitakyushu-City, Japan, October 2008. Springer. doi:10.1007/978-3-540-88194-0_5.

Matthew Danish and Hongwei Xi. Using lightweight theorem proving in an asynchronous systems context. In Julia M. Badger and Kristin Yvonne Rozier, editors, *Proceedings of the 6th NASA Formal Methods Symposium*, volume 8430 of *Lecture Notes in Computer Science*, pages 158–172, Houston, TX, USA, April 2014. Springer. doi:10.1007/978-3-319-06200-6_12.

Pawel Jakub Dawidek. POSIX file system test suite. https://github.com/zfsonlinux/fstest.

Willem-Paul de Roever and Kai Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Number 47. United Kingdom, 1998.

Benjamin Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. Fiat: Deductive synthesis of abstract data types in a proof assistant. In *Proceedings of the 42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Mumbai, India, January 2015.

Edsger W Dijkstra. On the role of scientific thought. In *Selected Writings on Computing: A Personal Perspective*, pages 60–66. Springer, 1982.

Gidon Ernst, Gerhard Schellhorn, Dominik Haneberg, Jörg Pfähler, and Wolfgang Reif. A formal model of a virtual filesystem switch. *arXiv preprint arXiv:1211.6187*, 2012. doi:10.4204/EPTCS.102.5.

Gidon Ernst, Gerhard Schellhorn, Dominik Haneberg, Jörg Pfähler, and Wolfgang Reif. Verification of a virtual filesystem switch. In Ernie Cohen and Andrey Rybalchenko, editors, *Proceedings of Verified Software: Theories, Tools and Experiments 2013*, volume 8164 of *Lecture Notes in Computer Science*, pages 242–261, Menlo Park, CA, USA, May 2013. Springer. doi:10.1007/978-3-642-54108-7_13.

Gidon Ernst, Jörg Pfähler, Gerhard Schellhorn, and Wolfgang Reif. Modular refinement for submachines of asms. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pages 188–203. Springer, 2014.

Gidon Ernst, Jörg Pfähler, Gerhard Schellhorn, and Wolfgang Reif. Inside a verified flash file system: Transactions & garbage collection. In *Proceedings of the 7th Working Conference on Verified Software: Theories, Tools and Experiments, San Francisco, CA*, 2015.

Xinyu Feng, Zhong Shao, Alexander Vaynberg, Sen Xiang, and Zhaozhong Ni. Modular verification of assembly code with stack-based control abstractions. In *PLDI*. ACM, 2006.

Havasi Ferenc. JFFS2 erase block summary. http://www.inf.u-szeged.hu/projectdirs/jffs2/jffs2-anal/node21.html. Accessed: 2015-09-17.

Filebench. Filebench filesystem benchmark. http://filebench.sourceforge.net/.

Kathleen Fisher, Yitzhak Mandelbaum, and David Walker. The next 700 data description languages. In *ACM Sigplan Notices*, volume 41, pages 2–15. ACM, 2006.

Leo Freitas, Konstantinos Mokos, and Jim Woodcock. Verifying the CICS file control API with Z/Eves: An experiment in the verified software repository. In *Engineering Complex Computer Systems, 2007. 12th IEEE International Conference on*, pages 290–298. IEEE, 2007.

Leo Freitas, Jim Woodcock, and Zheng Fu. POSIX file store in Z/Eves: An experiment in the verified software repository. *Sci. Comput. Program.*, 74:238–257, February 2009. doi:10.1016/j.scico.2008.08.001.

Philippa Gardner, Gian Ntzik, and Adam Wright. Local reasoning for the POSIX file system. In *Programming Languages and Systems*, pages 169–188. Springer, 2014.

Thomas Gleixner, Frank Haverkamp, and Artem Bityutskiy. UBI - unsorted block images. http://linux-mtd.infradead.org/doc/ubidesign/ubidesign.pdf, 2006.

Global-Scale-Technology. Mirabox: The power to adapt. URL: https://www.globalscaletechnologies.com.

Mel Gorman. *Understanding the Linux virtual memory manager*. Prentice Hall Upper Saddle River, 2007.

David Greenaway, June Andronick, and Gerwin Klein. Bridging the gap: Automatic verified abstraction of C. In Lennart Beringer and Amy Felty, editor, *International Conference on Interactive Theorem Proving*, pages 99–115, Princeton, New Jersey, USA, August 2012. Springer Berlin / Heidelberg. doi:10.1007/978-3-642-32347-8_8.

David Greenaway, Japheth Lim, June Andronick, and Gerwin Klein. Don't sweat the small stuff: Formal verification of C code without the pain. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 429–439, Edinburgh, UK, June 2014. ACM. doi:10.1145/2594291.2594296.

Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. Deep specifications and certified abstraction layers. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 595–608. ACM, 2015. doi:10.1145/2676726.2676975.

Haryadi S. Gunawi, Cindy Rubio-González, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dussea, and Ben Liblit. EIO: error handling is occasionally correct. In Mary Baker and Erik Riedel, editors, *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, pages 207–222, February 2008.

HASP project. The Habit programming language: The revised preliminary report. Technical Report http://hasp.cs.pdx.edu/habit-report-Nov2010.pdf, Department of Computer Science, Portland State University, Portland, OR, USA, November 2010.

Wim H. Hesselink and Muhammad Ikram Lali. Formalizing a hierarchical file system. In *Proceedings of the 14th BCS-FACS Refinement Workshop*, volume 259 of *Electronic Notes in Theoretical Computer Science*, pages 67–85, Eindhoven, The Netherlands, November 2009. doi:10.1016/j.entcs.2009.12.018.

J. Hughes. Specifying a visual file system in Z. In *Formal Methods in HCI: III, IEE Colloquium on*, pages 3/1 – 3/3, 1989.

Adrian Hunter. A brief introduction to the design of UBIFS. http://linux-mtd.infradead.org/doc/ubifs_whitepaper.pdf, 2008.

IBM. IBM Official Website. https://www.ibm.com/us-en/.

JFFS2-EBS. JFFS2 erase-block summeries. http://www.linux-mtd.infradead.org/doc/jffs2.html.

Trevor Jim, J. Gregory Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In Carla Schlatter Ellis, editor, *Proceedings of the 2002 USENIX Annual Technical Conference*, pages 275–288, Monterey, CA, USA, June 2002. USENIX.

Rajeev Joshi and Gerard J Holzmann. A mini challenge: build a verifiable filesystem. *Formal Aspects of Computing*, 19(2):269–272, 2007. doi:10.1007/s00165-006-0022-3.

Sudeep Kanav. Compiler verification for a data format DSL. Master's thesis, 2014.

Jeffrey Katcher. PostMark: A new file system benchmark. Technical Report TR-3022, NetApp, October 1997.

Gabriele Keller, Toby Murray, Sidney Amani, Liam O'Connor-Davis, Zilin Chen, Leonid Ryzhyk, Gerwin Klein, and Gernot Heiser. File systems deserve verification too! In *Workshop on Programming Languages and Operating Systems (PLOS)*, pages 1–7, Farmington, Pennsylvania, USA, November 2013. doi:10.1145/2525528.2525530.

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *ACM Symposium on Operating Systems Principles*, pages 207–220, Big Sky, MT, USA, October 2009. ACM.

Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems*, 32(1):2:1–2:70, February 2014. doi:10.1145/2560537.

Naoki Kobayashi. Quasi-linear types. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 29–42. ACM, 1999.

Eric Koskinen and Junfeng Yang. Reducing crash recoverability to reachability. *ACM SIGPLAN Notices*, 51(1):97–108, 2016.

Ramana Kumar, Magnus Myreen, Michael Norrish, and Scott Owens. CakeML: A verified implementation of ML. In Peter Sewell, editor, *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 179–191, San Diego, January 2014. ACM Press. doi:10.1145/2535838.2535841.

Ihor Kuz, Yan Liu, Ian Gorton, and Gernot Heiser. CAmkES: A component model for secure microkernel-based embedded systems. *Journal of Systems and Software Special Edition on Component-Based Software Engineering of Trustworthy Embedded Systems*, 80(5):687–699, May 2007. doi:10.1016/j.jss.2006.08.039.

Peter Lammich and Andreas Lochbihler. The Isabelle collections framework. In Matt Kaufmann and Lawrence C. Paulson, editors, *Proceedings of the 1st International Conference on Interactive Theorem Proving*, volume 6172 of *Lecture Notes in Computer Science*, pages 339–354. Springer, 2010.

Dirk Leinenbach. *Compiler Verification in the Context of Pervasive System Verification*. PhD thesis, Saarland University, SaarbrÃijcken, 2008. URL http://www-wjp.cs.uni-saarland.de/publikationen/Lei08.pdf.

K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Proceedings of the 17th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370, Yogyakarta, Indonesia, October 2010. Springer. doi:10.1007/978-3-642-17511-4_20.

Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009. doi:10.1145/1538788.1538814.

John Levon. Oprofile a system profile for linux. URL: http://oprofile.sourceforge.net.

James Bornholt Antoine Kaufmann Jialin Li and Arvind Krishnamurthy Emina Torlak Xi Wang. Specifying and checking file system crash-consistency models. 2016.

Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. A study of Linux file system evolution. *ACM Transactions on Storage*, 10(1):3:1–3:32, January 2014. doi:10.1145/2560012.

Ognjen Maric and Christoph Sprenger. Verification of a transactional memory manager under hardware failures and restarts. In Cliff B. Jones, Pekka Pihlajasaari, and Jun Sun, editors, *Proceedings of the 19th International Symposium on Formal Methods (FM)*, volume 8442 of *Lecture Notes in Computer Science*, pages 449–464. Springer, May 2014. doi:10.1007/978-3-319-06410-9_31.

Peter J. McCann and Satish Chandra. PacketTypes: abstract specification of network protocol messages. In *Proceedings of the ACM Conference on Communications*, pages 321–333, Stockholm, Sweden, 2000. ACM.

Marshall Kirk McKusick, Willian N Joy, Samuel J Leffler, and Robert S Fabry. Fsck - the unixâĂă file system check program. *Unix System ManagerâĂŹs Manual-4.3 BSD Virtual VAX-11 Version*, 1986.

Carroll Morgan and Bernard Sufrin. *Specification of the UNIX filing system*, pages 91–140. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1987. ISBN 0-13-826579-8. URL http://dl.acm.org/citation.cfm?id=102598.102602.

MSD-IO-CONCEPTS. Microsoft Developer I/O Concepts. https://msdn.microsoft.com/en-us/library/windows/desktop/aa365199%28v=vs.85%29.aspx.

Gilles Muller, Charles Consel, Renaud Marlet, Luciano Porto Barreto, Fabrice Merillon, and Laurent Reveillere. Towards robust OSes for appliances: a new approach based on domain-specific languages. In *Proceedings of the 9th workshop on ACM SIGOPS European workshop: beyond the PC: new challenges for the operating system*, pages 19–24. ACM, 2000.

Magnus O Myreen. Functional programs: conversions between deep and shallow embeddings. In *Interactive Theorem Proving*, pages 412–417. Springer, 2012.

George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. CCured: type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems*, 27(3):477–526, 2005.

Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002. ISBN 978-3-540-43376-7. doi:10.1007/3-540-45949-9.

W. Norcott and D. Capps. IOzone file system benchmark. URL: www.iozone.org.

Liam O'Connor, Zilin Chen, Christine Rizkallah, Sidney Amani, Japheth Lim, Toby Murray, Yutaka Nagashima, Thomas Sewell, and Gerwin Klein. Refinement through restraint: Bringing down the cost of verification. In *International Conference on Functional Programming*, Nara, Japan, September 2016.

ONFi. Open Nand-Flash Interface. http://www.onfi.org/specifications/.

Sam Owre, John M Rushby, and Natarajan Shankar. PVS: A prototype verification system. In *Automated Deduction at CADE-11*, pages 748–752. Springer, 1992.

D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

Jörg Pfähler, Gidon Ernst, Gerhard Schellhorn, Dominik Haneberg, and Wolfgang Reif. Formal specification of an erase block management layer for flash memory. In *Hardware and software: verification and testing*, pages 214–229. Springer, 2013.

Jörg Pfähler, Gidon Ernst, Gerhard Schellhorn, Dominik Haneberg, and Wolfgang Reif. Crash-safe refinement for a verified flash file system. Technical report, Technical Report 2014-02, University of Augsburg, 2014.

Lee Pike, Patrick Hickey, James Bielman, Trevor Elliott, Thomas DuBuisson, and John Launchbury. Programming languages for high-assurance autonomous vehicles: Extended abstract. In *Proceedings of the 2014ACM SIGPLAN Workshop on Programming Languages Meets Program Verification*, pages 1–2, San Diego, California, USA, 2014. ACM. doi:10.1145/2541568.2541570.

Wolfgang Reif, Gerhard Schellhorn, Kurt Stenzel, and Michael Balser. Structured specifications and interactive proofs with KIV. In Wolfgang Bibel and Peter H. Schmitt, editors, *Automated Deduction – A Basis for Applications*, volume 9 of *Applied Logic Series*, pages 13–39. Springer, 1998. ISBN 978-90-481-5051-9. doi:10.1007/978-94-017-0435-9_1.

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

Tom Ridge, David Sheets, Thomas Tuerk, Andrea Giugliano, Anil Madhavapeddy, and Peter Sewell. SibylFS: formal specification and oracle-based testing for POSIX and real-world file systems. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 38–53. ACM, 2015.

Christine Rizkallah, Japheth Lim, Yutaka Nagashima, Thomas Sewell, Zilin Chen, Liam O'Connor, Toby Murray, Gabriele Keller, and Gerwin Klein. A framework for the automatic formal verification of refinement from cogent to c. In *International Conference on Interactive Theorem Proving*, Nancy, France, August 2016.

Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, February 1992. doi:10.1145/146941.146943.

Cindy Rubio-González and Ben Liblit. Defective error/pointer interactions in the Linux kernel. In Matthew B. Dwyer and Frank Tip, editors, *Proceedings of the 20th International Symposium on Software Testing and Analysis*, pages 111–121, Toronto, ON, Canada, July 2011. ACM. doi:10.1145/2001420.2001434.

Stephen M Rumble, Ankita Kejriwal, and John K Ousterhout. Log-structured memory for dram-based storage. In *FAST*, volume 1, page 16, 2014.

Rust. The Rust programming language. http://rustlang.org, 2014. Accessed March 2015.

Mark Saaltink. The Z/Eves system. In *ZUM'97: The Z Formal Specification Notation*, pages 72–85. Springer, 1997.

Suman Saha, Julia Lawall, and Gilles Muller. 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)*, pages 41–50, Chicago, IL, USA, April 2011.

Suman Saha, Jean-Pierre Lozi, Gaël Thomas, Julia L Lawall, and Gunter Muller. 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*, pages 1–12. IEEE, 2013.

Gerhard Schellhorn, Gidon Ernst, Jörg Pfähler, Dominik Haneberg, and Wolfgang Reif. Development of a verified Flash file system. In Yamine Aït Ameur and Klaus-Dieter

Schewe, editors, *Proceedings of the 4th International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z*, volume 8477 of *Lecture Notes in Computer Science*, pages 9–24, Toulouse, France, June 2014. Springer. doi:10.1007/978-3-662-43652-3_2.

Andreas Schierl, Gerhard Schellhorn, Dominik Haneberg, and Wolfgang Reif. Abstract specification of the UBIFS file system for flash memory. In Ana Cavalcanti and Dennis R. Dams, editors, *Proceedings of the Second World Congress on Formal Methods (FM)*, volume 5850 of *Lecture Notes in Computer Science*, pages 190–206. Springer, November 2009. doi:10.1007/978-3-642-05089-3_13.

Frederick Smith, David Walker, and Greg Morrisett. Alias types. In Gert Smolka, editor, *Programming Languages and Systems*, volume 1782 of *Lecture Notes in Computer Science*, pages 366–381. Springer Berlin Heidelberg, 2000. ISBN 978-3-540-67262-3. doi:10.1007/3-540-46425-5_24. URL http://dx.doi.org/10.1007/3-540-46425-5_24.

J Michael Spivey. *The Z notation*. Prentice Hall Hemel Hempstead, 1992.

Richard Stallman. Using and porting the gnu compiler collection. In *MIT Artificial Intelligence Laboratory*. Citeseer, 2001.

STLinux. Data storage on NAND Flash. http://www.stlinux.com/howto/NAND/data.

Miklos Szeredi et al. FUSE: Filesystem in userspace. https://sourceforge.net/projects/fuse, 2010.

Ken Thompson. Unix time-sharing system: Unix implementation. *Bell System Technical Journal*, 57(6):1931–1946, 1978. doi:10.1002/j.1538-7305.1978.tb02137.x.

Chia-Che Tsai, Yang Zhan, Jayashree Reddy, Yizheng Jiao, Tao Zhang, and Donald E Porter. How to get more value from your file system directory cache. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 441–456. ACM, 2015.

Hung-Wei Tseng, Laura Grupp, and Steven Swanson. Understanding the impact of power loss on flash memory. In *Proceedings of the 48th Design Automation Conference*, pages 35–40. ACM, 2011.

Harvey Tuch, Gerwin Klein, and Michael Norrish. Types, bytes, and separation logic. In Martin Hofmann and Matthias Felleisen, editor, *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 97–108, Nice, France, January 2007. ACM.

Eben Upton, Rob Mullins, Jack Lang, and Alan Mycroft. The making of pi. https://www.raspberrypi.org/about.

Alexander Vaynberg and Zhong Shao. Compositional verification of a baby virtual memory manager. In *Certified Programs and Proofs*, pages 143–159. Springer, 2012.

Philip Wadler. Linear types can change the world! In *Programming Concepts and Methods*, 1990.

David Walker and Greg Morrisett. Alias types for recursive data structures. In Robert Harper, editor, *Types in Compilation*, volume 2071 of *Lecture Notes in Computer Science*, pages 177–206. Springer Berlin Heidelberg, 2001. ISBN 978-3-540-42196-2. doi:10.1007/3-540-45332-6_7. URL http://dx.doi.org/10.1007/3-540-45332-6_7.

David Woodhouse. JFFS2: The journalling flash file system, version 2. In *Proceedings of Ottawa Linux Symposium*, 2003.

Junfeng Yang, Can Sar, and Dawson Engler. EXPLODE: a lightweight, general system for finding serious storage system errors. In Brian N. Bershad and Jeffrey C. Mogul, editors, *USENIX Symposium on Operating Systems Design and Implementation*, pages 131–146, November 2006.