Inside a Verified Flash File System: Transactions & Garbage Collection [⋆] [⋆⋆]

Gidon Ernst, Jörg Pfähler, Gerhard Schellhorn, and Wolfgang Reif

Institute for Software and Systems Engineering
University of Augsburg, Germany
{ernst,pfaehler,schellhorn,reif}@isse.de

Abstract. The work presented here addresses a long-standing conceptual gap in flash file system verification: We map an abstract graph-based representation down to the flat blocks of bytes of the storage medium. Specifically, we consider grouping of file system objects into atomic transactions together with layout, allocation and garbage collection of on-flash storage space. Two major concerns guide the design and verification: proper handling of errors and, more importantly, guaranteed recovery from unexpected power cuts. Finding *useful specifications* of intermediate interfaces to address these concerns realistically dominates the verification effort.

Keywords: Flash File Systems, Formal Verification, Specification, Transactions, Garbage Collection, Write Buffer, KIV

1 Introduction

NASA's proposal [19] to build a verified file system (FS) for flash memory has been received with a lot of interest and has prompted a great body of work. Many file system concepts have been modeled, formalized and verified by different researchers, with varying degrees of abstraction, such as a path-based interface in [17], and a graph-based view in [8]. Most of these approaches study some selected aspects in isolation only. The inner workings of realistic flash file systems have received relatively little formal treatment in comparison to high-level concepts.

As part of our ongoing effort [26] to construct a verified flash FS,[1] we bridge the remaining conceptual gap between a high-level structured representation of file system objects towards an encoding within the erase blocks and pages of flash hardware. We present the specifications and verified implementations of two intermediate file system layers: A transactional *journal* provides atomic writes of groups of file system objects alongside free-space management by garbage

[⋆] This work is part of the project "Verifikation von Flash-Dateisystemen" (RE828/13-1) sponsored by the Deutsche Forschungsgemeinschaft (DFG).

[⋆⋆] The final publication is available at Springer via http://dx.doi.org/10.1007/978-3-319-29613-5_5

[1] http://isse.de/flashix

collection of obsolete objects. A *persistence* layer provides the transition down to bytes, caching partial writes for efficiency. The two layers are fully integrated into the rest of our development by mechanized proofs, conducted in the interactive verification system KIV [10].

Besides functional correctness, it is of great interest that the file system can deal with *power cuts* anytime during the run of an operation. Whenever an operation is aborted in an intermediate state, a designated recovery procedure can reconstruct a state sufficiently similar to the pre- resp. post-state of the respective operation. The journal and the persistence layer work in close cooperation to provide strong guarantees in the presence of such power cuts and similarly for nondeterministic hardware errors, which have to be taken into account as well.

Alongside the presentation of the formal models we will demonstrate that artifacts tend to leak through abstractions and interfaces, disrupting "obvious" verification approaches, even when the implementation concerns are cleanly separated. We will show how we have addressed such difficulties in this specific case study, especially focusing on power cuts.

Section 2 provides an overview of our approach and of the core concepts of flash file systems. Section 3 explains the formal models that represent the boundaries and capture the requirements for this work. Sections 4 to 7 present the formal models and some verification artifacts with the details necessary to expose several intricate aspects. Sec. 8 discusses related work and Sec. 9 draws insights from the verification. In summary, the contribution of this paper consists of a significant step towards a realistic, fully verified file system for flash memory.

2 Background

This section gives an overview over the project, the basic idea behind modern flash file system implementations and how the various parts of the system play together in Sec. 2.1. The formalism that backs the verification of functional correctness and power cut safety is summarized in Sec. 2.2.

2.1 Project Overview & Flash File System Concepts

This work is part of an ongoing long-term project to construct a verified, POSIX-compliant [29] file system for flash memory, taking up NASA's proposal [19]. We take the existing UBIFS [18] as a design blueprint, which realizes state-of-the-art techniques to address the inherent access limitations of flash hardware. To tackle such a complex verification task we follow an incremental, correct-by-construction approach: a top-level specification of the textual POSIX standard is gradually refined towards an implementation.

The resulting layers are (partially) visualized in Fig. 1. These correspond to the various logical parts of the file system, and to different levels of abstraction. Technically, each box represents an Abstract State Machines (ASMs) [4], which are used to encode both specifications (white) and the implementations (gray) in an operational way. The interface symbol —◦— denotes that one component uses another. Correctness is established by a series of nested refinements, depicted by dotted lines.

In the first refinement step [12,13] a formal top-level POSIX specification is broken down into generic concepts (such as path lookup) realized by a Virtual Filesystem Switch (VFS) and flash-specific concepts realized by the flash file system core. An abstract specification of the behavior of the latter decouples the two.

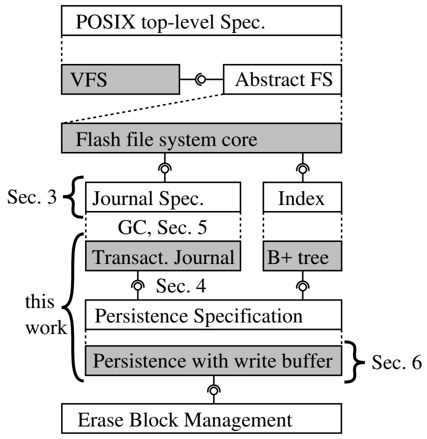


Fig. 1: System Structure

A POSIX-compliant file system can be thought of as a tree-like structure consisting of directories with files as leaves.[2] File and directory names are attached to the edges in the tree. As an example, Fig. 2 shows an excerpt of a typical file system hierarchy. Directories are visualized as grey circles, files as white ones. The root node at the top corresponds to the path `/`.

The VFS decomposes the tree structure shown in Fig. 2 into three types of file system objects: One for directories and files (storing metadata, such as size, access rights and timestamps), one for directory entries (carrying a name and a reference to the target object) and one for each segment of file data. In response to each top-level POSIX operation, the VFS instructs the file system core to create, modify, or delete a number of these objects. Creating a new file `/tmp/test.txt`, for example, yields three updates, corresponding to the part of Fig. 2 with a dotted contour: one for the new file, one for the new directory entry, and one to update some metadata of the parent directory `/tmp`.

Storing these file system objects has to take into account the restricted access characteristics of flash memory already at a very high level. Flash memory is structured into erase blocks, each consisting of a number of pages. Random reads are supported, but writes must be page aligned and sequential within a block. Overwriting is not supported and space can be reclaimed by erasing whole blocks only, which is slow and physically wears out the flash memory cells over time. These difficulties are addressed incrementally by our models.

The flash file system core in Fig. 1 tackles the problem that updates need to be written out-of-place to flash. For uniformity, file system objects (and their updates) are encoded within so-called *nodes*, which are ultimately written to flash memory through the *journal* layer. An *index* (implemented as a B^+-tree) tracks current versions of data by mapping *keys* to the respective addresses of the most recent node for a given object. The core relies on the index component to store this mapping in memory for efficient access and on flash (in an outdated version) to speed up startup time.

[2] With hard-links (which we support) this structure becomes an acyclic graph.

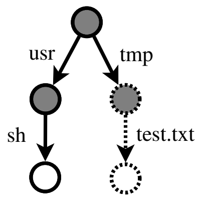


Fig. 2: File system tree, creation of `/tmp/test.txt`

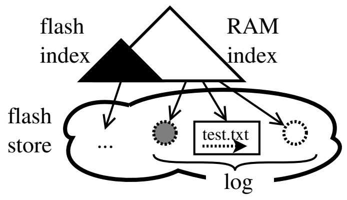


Fig. 3: Conceptual view of the index, flash store, and log; showing the update corr. to Fig. 2.

The integration of the index and the journal is shown in Fig. 3. At the bottom the flash memory is visualized as an unstructured storage, except that "recent" writes are recorded in a sequential log. At the top, the index is shown: a current version in main memory encompasses all modifications, but there is also an outdated version stored on flash. Informally, the log corresponds exactly to the difference between the two indices. At specific points, called *commits*, the current index is stored on flash and the log is emptied.

The flash file system core thus already introduces the concepts necessary to deal efficiently with out-of-place updates and recovery from power failures (by replaying the log starting from the flash index). However, the following aspects are delegated to lower layers: the core assumes that the journal 1) can write several nodes atomically, 2) can perform a commit atomically together with the index and 3) takes the block structure and sequential writes into account. Furthermore, garbage collection is just specified abstractly, because an implementation is meaningful only once blocks are considered.

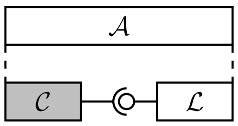
In this paper we show how this can be achieved in two steps: The transactional journal provides the atomicity of multiple updates based on blocks and includes garbage collection. The transition down to bytes is realized within the persistence layer, which in turn relies on the erase block management as a logical view of the flash hardware (which is similar to UBI [16], see [23]). It writes the nodes buffered and sequentially to flash. Additionally, atomicity of the commit and free space management is provided.

From the implementation ASMs (gray in Fig. 1) we generate executable Scala[3] code (for simulation and testing purposes) and C code, which is integrated into Linux via FUSE.[4]

2.2 Methodology

The formal foundations of our work are Abstract State Machines (ASMs) [4] and a corresponding refinement theory [3,25] with a recent extension [11] to support encapsulated submachines and the modular verification of power cuts.

[3] `http://scala-lang.org`
[4] `http://fuse.sourceforge.net`



Referring back to Fig. 1, the development follows a recurring pattern as shown on the right: an abstract model $\mathcal{A}$ is decomposed into an implementation part $\mathcal{C}$ which realizes a specific subtask, whereas some concepts remain abstract, encoded by a (local) subcomponent $\mathcal{L}$. Such a hierarchical construction of systems is modular in the sense that any correct implementation of $\mathcal{L}$ can be plugged in instead without compromising the proof that $\mathcal{C}$ adheres to $\mathcal{A}$. The critical aspect wrt. power cuts is how persistent data is modeled as a not-necessarily separable part of $\mathcal{L}$.

Technically, we encode components uniformly as Abstract State Machines $\mathcal{M} = ((\mathtt{OP}_i)_{i\in I}, \mathit{St}, \mathit{Init}, \mathit{Cr}, \mathtt{Rec})$. These expose some operations $\mathtt{OP}_i$ as external interface, which have input and output parameters and preconditions. Operations are defined by abstract imperative programs that compute on an internal state $s\colon \mathit{St}$, with the usual constructs such as assignments, conditionals, loops, recursion, nondeterministic choice, and calls to submachines. Power cuts are specified by a <u>cr</u>ash predicate $\mathit{Cr} \subseteq \mathit{St} \times \mathit{St}$, subsequent recovery is implemented by the designated recovery operation $\mathtt{Rec}$. A run of an ASM starts in an initial state $s_0\colon \mathit{St}$ with $\mathit{Init}(s_0)$ and repeatedly executes operations, that either terminate normally, or are interrupted in an intermediate state followed by a crash and execution of the recovery operation.

Correctness of a <u>c</u>oncrete ASM $\mathcal{C} = ((\mathtt{COP}_i)_{i\in I}, \mathit{CSt}, \mathit{CInit}, \mathit{CCr}, \mathtt{CRec})$ is defined not by giving a postconditions per operation, but instead in terms of another, more <u>a</u>bstract ASM $\mathcal{A} = ((\mathtt{AOP}_i)_{i\in I}, \mathit{ASt}, \mathit{AInit}, \mathit{ACr}, \mathtt{ARec})$ that encodes the specification and requirements. Intuitively, $\mathcal{C}$ refines $\mathcal{A}$, if for each run of $\mathcal{C}$ there is a matching run of $\mathcal{A}$ with the same inputs and outputs. Formally, we follow the contract approach to refinement [31]. We prove refinement by forward simulation with a coupling relation $R \subseteq \mathit{ASt} \times \mathit{CSt}$ and commuting diagrams (we omit the standard proof obligations for each pair $\mathtt{COP}_i$ and $\mathtt{AOP}_i$).

On a semantic level, it is easy to integrate correctness of power cuts into the refinement approach with a small-step semantics for operations: a crashed call to a concrete operation and its recovery must be matched by a crashed call of the corresponding abstract operation and recovery. This would lead, in principle to a temporal proof obligation of the form $\Box\, \mathit{crashsafe}$ that must hold during the run of any $\mathtt{COP}$. Such a property would lead to a huge number of verification conditions and cannot be expressed in weakest-precondition/Hoare calculus. One can, however, express the recovery condition *in between completed operations* as a variant of the standard forward simulation condition prefixed with a crash recovery:

$$\begin{aligned} & R(as, cs) \wedge \mathit{CCr}(cs, cs') \\ & \to \langle\!|\mathtt{CRec}(; cs')|\!\rangle\ (\ \exists as'.\ \mathit{ACr}(as, as') \wedge \langle \mathtt{ARec}(; as')\rangle\, R(as', cs')\ )\,, \end{aligned} \tag{1}$$

where $\langle\!|p|\!\rangle\, \varphi$ denotes the weakest precondition (total correctness) of the program p with respect to postcondition φ and $\langle p\rangle\, \varphi$ asserts the existence of some terminating execution of p satisfying φ in its final state.

Surprisingly, this property is sufficient, which can be derived on a purely semantic level given simple conditions about the concrete machine $\mathcal{C}$. This reduction exploits a close relationship between error handling and power cuts (see [21] for a similar idea): Intuitively, at the lowest level, each operation of the hardware has the possibility to fail without altering the flash memory. Conversely, all other state is in RAM and will be completely arbitrary after a crash (for a suitable definition of CCr). This means that each partial run has at least one completion that leaves the flash untouched, which implies that the crashed flash states are a subset of the final ones, reducing the verification burden to an entirely big-step setting (i.e., expressible with standard verification methodology).

This observation can be generalized to a state space of some intermediate machine that does not clearly separate flash and RAM data, which is important for flexibility in modeling. Formally, an operation $\mathtt{COP}_i$ of $\mathcal{C}$ is *crash-neutral*, if there is the possibility to postpone the effect of a crash to some final state of $\mathtt{COP}_i$, which leads to the additional proof obligation

crash-neutrality:

$$pre_{COP_i}(in, cs) \wedge CCr(cs, cs') \rightarrow \langle \mathtt{COP}_i(in; cs, out)\rangle\ CCr(cs, cs') \qquad (2)$$

For a state that is entirely in RAM, this condition is trivial, since CCr is not constrained then, i.e., admits arbitrary transitions. In practice this means that (2) must only be proved for abstract submachines $\mathcal{L}$ called by $\mathcal{C}$, which is typically easy. The formalization of this approach and the proofs are detailed in [11].

In the remainder of the paper, we use the following notational conventions: We write variables in *italic* and operations/functions/predicates in `typewriter` font. We frequently use partial functions/finite maps $f \colon A \nrightarrow B$. For key $a \in \mathtt{dom}(f)$ the value associated to it by f is written $f[a]$. Function override is denoted by $f[a \mapsto b]$. The assignment $f[a] \coloneqq b$ abbreviates $f \coloneqq f[a \mapsto b]$.

3 Formal Specification of the Journal and Index

This section presents the formal model of the journal, which defines the requirements for the work of this paper. It is based on our previous work [28]. The model reflects the limited access characteristics of flash memory. Operations presented here should be interpreted as atomic transitions, which captures the requirement of transactional behaviour to be implemented in Sec. 4. The model furthermore admits that the hardware may sporadically refuse to perform an operation. We present several invariants that can be expressed (and proved easily) at this level of abstraction and can be assumed later on for the verification of the refinement.

The abstract state is given by an unordered $\underline{f}$lash $\underline{s}$tore *fs* and a list *log* of addresses that have been written to since the last commit (c.f. Fig. 3)

$$\textbf{spec var} \quad fs \colon Address \nrightarrow Node, \qquad log \colon List\langle Address\rangle.$$

The journal has an operation to read a node from flash, and operations to store groups of n nodes,[5] extending the *log*. All operations may fail nondeterministi-

[5] A maximum group size of $n = 4$ nodes is sufficient for all operations of the FS core. Note that an entire write operation is already decomposed into fixed-size writes of individual segments by higher components.

cally without changing the state,[6] this can be observed with the returned error code err (recall that output parameters follow the semicolon). In case of success, the outputs $adr_{1\cdots n}$ contain the addresses of the new nodes on flash, which are later stored in the index.

```
jnl_spec_get($adr$; $nd, err$)
  { $nd := fs[adr], err :=$ ESUCCESS }  or  { $err :=$ EFAIL }

jnl_spec_append$_n$($nd_1, \dots, nd_n$;  $adr_1, \dots, adr_n$,  $err$)
      { choose $adr_{1\cdots n} \notin$ dom($fs$) distinct
        $fs \;:= fs[adr_1 \mapsto nd_1] \cdots [adr_n \mapsto nd_n]$          ($\star$)
        $log := log + adr_1 + \cdots + adr_n$
        $err :=$ ESUCCESS }
   or { $err :=$ EFAIL }
```

We also need a formal model of the index (its implementation is out of scope of this paper, though). The state of the corresponding ASM maintains two maps

$$\textbf{spec var} \quad ri,\; fi \colon Key \nrightarrow Address,$$

the RAM index ri and the flash index fi. All operations, except commit and recovery which are explained later on, access the RAM index only. There are ASM operations to lookup, store, and remove mappings that directly refer to their algebraic counterparts, e.g.,

```
idx_lookup($key$; $adr$) { $adr := ri[key]$ }
idx_store($key, adr$)  { $ri[key] := adr$ }
idx_remove($key$)      { $ri := ri - key$ }
```

The system maintains several invariants, for example that the RAM index does not contain unallocated addresses; and that all addresses in the log are valid.

$$\textbf{invariant} \quad \texttt{ran}(ri) \subseteq \texttt{dom}(fs) \quad \text{and} \quad \{adr \mid adr \in log\} \subseteq \texttt{dom}(fs)$$

Addresses $adr \notin \texttt{ran}(ri)$ are obsolete and can be cleaned up by garbage collection (see Sec. 5). However, the index is accessible (efficiently) only by keys. Each node nd stores its respective key, denoted by $nd.\texttt{key}$, and thus one can equivalently check $fs[adr].\texttt{key} \notin \texttt{dom}(ri)$. The induced invariant is

$$\textbf{invariant} \quad \forall\, key \in ri.\; fs[ri[key]].\texttt{key} = key \tag{3}$$

The RAM index determines exactly, which part of the flash memory constitutes the observable file system state. However, in the event of a power cut the RAM state is lost. We model this by setting ri to an arbitrary value, without changing fs. Formally, the effect of a crash is specified by

$$\begin{aligned} Cr_{\text{idx}}(ri, fi, ri', fi') \;&\leftrightarrow\; fi = fi' \\ Cr_{\text{jnl}}(fs, log, fs', log') \;&\leftrightarrow\; fs = fs' \wedge log = log' \end{aligned} \tag{4}$$

[6] The failure case also witnesses the crash-neutral run wrt. (4) as required by (2).

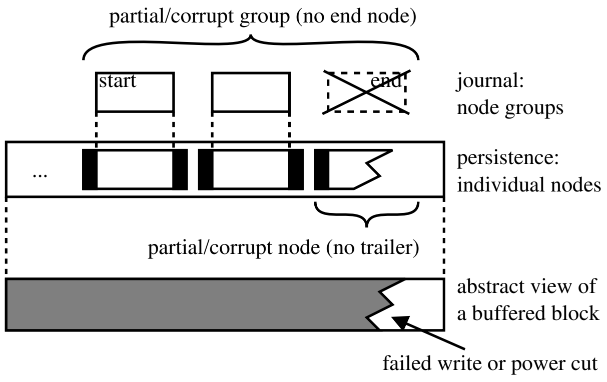

partial/corrupt group (no end node)
start
end
journal:
node groups
...
persistence:
individual nodes
partial/corrupt node (no trailer)
abstract view of
a buffered block
failed write or power cut

Fig. 4: Detecting partially written nodes and groups

(where the overall effect is the conjunction of the two predicates). That the RAM index is truly redundant and can be recovered to its previous state after a power cut is expressed by

$$\textbf{invariant} \quad ri = \texttt{replay}(log, fi, fs), \tag{5}$$

where `replay` is part of the recovery operation `Rec` of the FS core. It traverses the *log* from oldest to newest and (re-)applies all missing operations to the outdated *fi*. As a consequence, the *log* must be computable by the implementation (see Fig. 10), even though it is not part of the actual concrete state.

The size of the log determines how long it takes to mount the file system initially. In order to keep the log reasonably small, a periodic commit writes the current index to flash and empties the log. This has to happen atomically, otherwise power cuts in between can lead to inconsistent states. Note that a commit trivially establishes the recovery invariant. Commit is modeled as follows:

$$\texttt{spec\_commit}() \ \{ \ fi := ri, log := [\,] \ \}$$

4 Transactions in the Journal

The *transactional journal* layer introduces a structured view of the flash storage that takes the block structure of flash memory into account. It implements the specification given in Sec. 3 by mapping *fs* to an array of blocks, each of which contains a list of nodes. The *log* is represented implicitly within the blocks: since blocks already give a sequential ordering for the contained nodes, it is sufficient to maintain a list of those blocks which constitute the nodes referred to by the abstract *log*. The main difficulty is that the journal needs to implement transactions of multiple nodes *atomically* wrt. hardware errors and power cuts, based on a (abstract specification of the) *persistence* layer that caches writes until a page boundary is reached.

```
$\texttt{jnl\_append}_2(nd_1, nd_2; adr_1, adr_2, err)$
  let $size = \texttt{size}(nd_1) + \texttt{size}(nd_2)$
  $\texttt{jnl\_allocate}(size; loghead, err)$
  if $err = \texttt{ESUCCESS}$ then
    $\texttt{persistence\_add\_node}(loghead, \texttt{gnode}(nd_1, \texttt{true}, \texttt{false}); adr_1, err)$
  if $err = \texttt{ESUCCESS}$ then
    $\texttt{persistence\_add\_node}(loghead, \texttt{gnode}(nd_2, \texttt{false}, \texttt{true}); adr_2, err)$
  if $err = \texttt{ESUCCESS}$ then
    $\texttt{persistence\_flush}(loghead, err)$
  if $err \neq \texttt{ESUCCESS}$ then $validhead := \texttt{false}$
```

Fig. 5: Journal implementation to store two nodes on flash.

In order to guarantee this atomicity, the journal groups nodes per operation. The whole group must have been written successfully in order to make a valid contribution to the observable file system state. Atomicity at the level of individual nodes is required as well, but for the sake of modularization this concept is not addressed in the journal but in the persistence layer. This approach permits the journal to treat its underlying storage as a simple sequence of nodes in contrast to a more complicated view.

Figure 4 puts the two layers in relation. A single erase block is shown at the bottom, the grey area denotes the part that has already been written to (omitting its partitioning into pages). Within the block the persistence layer stores the sequence of nodes, each of which is marked by a header and a trailer. A node group has a start/end marker at the first/last node. The ragged delimitations at the right in Fig. 4 indicate a failed write or power cut, accordingly the last node lacks its trailer, hence it is invalid and so is the entire group.

Transactional Journal. Appending $n = 2$ nodes to the transactional journal is then implemented as shown in Fig. 5 (the cases $n = 1, 3, 4$ are similar). The algorithm first selects a block number *loghead* with sufficient remaining space to hold the new data.

$$\textbf{state var } loghead : \mathbb{N}, \quad validhead : \mathbb{B}$$

The current block can be reused if the last write did not fail, leaving partially written nodes at the end. So for example the erase block in Fig. 4 can not be reused.[7] We store in *validhead* whether the current block is still usable. Each node is then written individually wrapped in a group node

$$\textbf{data type } GroupNode = \texttt{gnode}(\texttt{nd}\colon Node, \texttt{start?}\colon \mathbb{B}, \texttt{end?}\colon \mathbb{B}) \ ,$$

[7] We need to be able to read all nodes from the erase block in order to perform garbage collection, but detecting partially written nodes reliably *in between* completely written ones is not possible.

with the additional start and end marker. The first flag indicates whether this node is the first one of a group, the second flag indicates whether it is the last one (c.f. Fig. 4). A singleton group has both flags set. Every call to the persistence layer can fail so the returned error code is checked after each step.

At the end of the operation, the corresponding block is flushed to ensure that all nodes have actually been written. The persistence layer has a write cache in order to improve efficiency—no guarantees are given about what has been written until a block is flushed. Note that the returned addresses adr_i are chosen by the persistence layer and are simply passed through.

Persistence Specification. The journal uses the persistence layer to write nodes. The specification of the persistence layer maintains the finite map *blocks* from block numbers to block content of type *GroupBlock*. Each block consists of a list of group nodes and additional data, that exposes some details of the persistence implementation in a controlled way in order to express preconditions and invariants precisely.

$$\begin{aligned} &\textbf{spec var } \mathit{blocks}\colon \mathbb{N} \nrightarrow \mathit{GroupBlock} \\ &\textbf{data type } \mathit{GroupBlock} = \texttt{gblock}(\texttt{nodes}\colon \mathit{List}\langle \mathit{GroupNode}\rangle, \\ &\qquad\qquad\qquad\qquad\qquad \texttt{addrs}\colon \mathit{List}\langle \mathit{Address}\rangle, \\ &\qquad\qquad\qquad\qquad\qquad \texttt{flushindex}\colon \mathbb{N}, \texttt{rsize}\colon \mathbb{N}) \end{aligned} \tag{6}$$

Field `addrs` gives for each node in `nodes` the address where it is stored; `rsize` stores the total size of all nodes in a block that are still referenced by the in-RAM index. It is used to determine blocks suitable for garbage collection as explained in Sec. 5. Finally, the `flushindex` exposes, which part of the list `nodes` has been persisted; nodes $\texttt{nodes}[i]$ at a position $i \geq \texttt{flushindex}$ are (conceptually) still cached in RAM and lost on a power cut. Flushing increases `flushindex` to the length of `nodes`.

The log itself is implicit in the final file system. It can be determined from the blocks that contain new nodes. For this purpose the persistence layer keeps their numbers in a list *logblocks*.

$$\textbf{spec var } \mathit{logblocks}\colon \mathit{List}\langle \mathbb{N}\rangle$$

Whenever the journal requests a fresh erase block to be used as part of the log, this block is recorded at the end of *logblocks*. Each such addition needs to be persisted to flash immediately in the implementation.

Verification. For the correspondence between *fs* and *log* on the one hand and *blocks* and *logblocks* on the other, unflushed nodes and partial groups need to be omitted. The abstraction considers *valid* nodes only, which are part of a proper group that has been flushed entirely. We write $\mathit{blocks}_\downarrow$ for the state *blocks* stripped of all invalid nodes (at the end of each erase block, c.f. Fig. 4) and corresponding addresses. The abstraction relation is formalized as

$$\textbf{coupling} \quad \mathit{fs} = \texttt{abs-fs}(\mathit{blocks}_\downarrow) \;\text{ and }\; \mathit{log} = \texttt{abs-log}(\mathit{logblocks}, \mathit{blocks}_\downarrow)$$

where

$$\begin{aligned}&\texttt{abs-fs}(\mathit{blocks})[\mathit{adr}] = \mathit{nd} \\ &\quad \text{iff}\quad \mathit{blocks}[n].\texttt{nodes}[i].\texttt{nd} = \mathit{nd} \qquad \text{and} \\ &\qquad\quad\ \mathit{blocks}[n].\texttt{addrs}[i] = \mathit{adr} \qquad \text{for some } n, i \text{ within bounds}\end{aligned} \tag{7}$$

and `abs-log` collects the addresses of the blocks in *logblocks* recursively

$$\begin{aligned}&\texttt{abs-log}([\,], \mathit{blocks}) = [\,] \\ &\texttt{abs-log}(n + \mathit{logblocks}, \mathit{blocks}) = \mathit{blocks}[n].\texttt{addrs} + \texttt{abs-log}(\mathit{logblocks}, \mathit{blocks})\end{aligned} \tag{8}$$

The difficulty during the verification of $\texttt{jnl\_append}_n$ is that assertions in intermediate states can not be expressed adequately in terms of $\texttt{abs-fs}(\mathit{blocks}_{\downarrow})$ and $\texttt{abs-log}(\mathit{logblocks}, \mathit{blocks}_{\downarrow})$. Both abstractions only reflect the changes *after* flushing the cache. Intermediate assertions therefore refer to $\mathit{blocks}_{\downarrow \mathit{loghead}}$, which removes all invalid nodes from all blocks except for the block *loghead*, where all the changes take place.

The other aspect crucial for the verification of $\texttt{jnl\_append}_n$ is that if the journal head is valid, it is the last block in the log and it ends on a complete, flushed group, i.e., if *validhead* is true then **invariant**

$$\begin{aligned}&\quad \mathit{loghead} \in \mathit{blocks} \wedge \mathit{logblocks} \neq [\,] \wedge \mathit{loghead} = \mathit{logblocks}.\texttt{last} \\ &\wedge (\mathit{blocks}[\mathit{loghead}].\texttt{nodes} \neq [\,] \rightarrow \mathit{blocks}[\mathit{loghead}].\texttt{nodes}.\texttt{last}.\texttt{end}?) \\ &\wedge \#\mathit{blocks}[\mathit{loghead}].\texttt{nodes} = \mathit{blocks}[\mathit{loghead}].\texttt{flushindex}\end{aligned}$$

also holds. Otherwise, it would be possible that a newly appended node completes a previously invalid node group.

5 Garbage Collection

The out-of-place updates of the transactional journal will necessarily accumulate a lot of obsolete data over time, i.e., data that is no longer referenced by the index. Garbage collection (GC) of the journal area remedies this problem by moving and compacting live data at the granularity of nodes. The GC procedure thus depends on and modifies the RAM index; furthermore, it is the only point where flash memory space is actually reclaimed.

The difficulties from a formal perspective are twofold: caching of writes of nodes is crucial for the effectiveness of garbage collection, but again considerably complicates the verification. Furthermore, choosing a block for garbage collection requires additional information and ties several layers closer together than already necessary, especially with respect to the recovery from power failures as explained in more detail in Sec. 7.

Specification. Again referring to the view in terms of *fs* and *ri* of the journal specification (Sec. 3), we can denote the central correctness property of the GC that no data is lost. Formally,

$$\mathit{fs} \circ \mathit{ri} = \mathit{fs}' \circ \mathit{ri}' \quad \text{and} \quad \texttt{dom}(\mathit{ri}) = \texttt{dom}(\mathit{ri}') \tag{9}$$

must hold for the primed state after the run of the GC, where $\_ \circ \_$ denotes function composition. The GC algorithm roughly corresponds to a number of transitions of the form

$$fs[adr'] := fs[ri[key]], \quad ri[key] := adr', \quad log := log + adr',$$

for some $key \in \mathtt{dom}(ri)$ and adr' fresh in fs. The first assignment moves live data to a different location, the second assignment updates the index, and the third records the operation in the log. Subsequently, some addresses $adrs \cap \mathtt{ran}(ri) = \emptyset$ can be deleted by

$$fs := fs \setminus adrs.$$

Implementation. In practice, a number of side conditions need to be satisfied, though. For example, a block that is part of the log cannot be collected until it is merged into the ordinary part of the journal during a commit, because it is needed for recovery (Sec. 7). Also, while the whole block is collected in one go, the corresponding index updates must be deferred: Due to caching, low-level write failures may not be detected immediately and only at the end (after flushing) it is clear whether the copying succeeded. The implementation of garbage collection is shown in Fig. 6. It first selects a block for garbage collection. Then the live nodes of the selected block are copied, which yields a list *keys* of affected keys and corresponding new addresses *dstadr* that are to be updated in the index. Finally, the now obsolete block is deallocated.

The heart of the garbage collection is the procedure `jnl_copy_block`. It reads all nodes *nds* and their addresses *srcadrs* from flash. In a loop, each node *nd* in *nds* is checked whether it is still in the index, i.e., if the key $nd.\mathtt{key}$ exists in the index and still maps to the node's address *srcadr*. Note that the index does not support queries by address, only by key, therefore each node has to store its own key, and by invariant (3) the keys match. If the node is not obsolete, we append a new copy to the journal and keep the index update $key \mapsto dstadr$ for later. At the end, we ensure that all nodes are persisted by flushing the block.

Verification. In the invariant (not shown) for the while loop in `jnl_copy_block`, it is necessary to state that *keys* and *dstadrs* collected so far correspond to the nodes that are still referenced by the index. Furthermore, not all of the nodes written are actually persisted immediately, so in the actual abstraction only a prefix of the written nodes appears in *fs* and *log*. It is therefore necessary to reason about the abstraction "after" a flush to the current journal head. Additionally, there may not always be a current journal head, leading to several distinct cases in the invariant.

Choosing Blocks for GC. From the perspective of functional correctness it is sufficient to choose any block of the journal that is outside the log,[8] but we certainly want to ensure that garbage collection picks a reasonable one. The

[8] Note that wear-leveling is performed by a lower layer and therefore is not limited by the choice of block of the garbage collection, i.e., blocks in the log can be moved by wear-leveling.

```
jnl_garbage_collection()
 persistence_get_gc_block(; $block$)
 jnl_copy_block($block$; $keys$, $dstadrs$, $err$)
 if $err$ = ESUCCESS then  idx_update_all($keys$, $dstadrs$; $err$)
 if $err$ = ESUCCESS then  persistence_deallocate($block$; $err$)

jnl_copy_block($block$; $keys$, $dstadrs$, $err$)
 let $srcadrs = [\,], nds = [\,]$
  persistence_read_block($block$; $srcadrs$, $nds$, $err$)
  while $srcadrs \neq [\,] \wedge err =$ ESUCCESS do {
   let $srcadr = srcadrs$.head, $nd = nds$.head.nd, $exists$, $idxadr$, $dstadr$ in
    idx_lookup($nd$.key; $exists$, $idxadr$, $err$)
    if $err =$ ESUCCESS $\wedge\ exists \wedge idxadr = srcadr$ then
     ...      // if necessary move the log head and flush the old block
     if $err$ = ESUCCESS then
      persistence_add_node($loghead$, gnode($nd$, $true$, $true$); $dstadr$, $err$)
      $keys := keys + key$, $dstadrs := dstadrs + dstadr$
      $srcadrs := srcadrs$.tail, $nds := nds$.tail
  }
  if $err$ = ESUCCESS then  persistence_flush($loghead$; $err$)
```

Fig. 6: Garbage collection procedures

information necessary for a good choice is for each block how many bytes still belong to live data, encoded in the `rsize` field (Sec. 4).

Here we have an example of coupling between components: although stored within the persistence layer, `rsize` is updated alongside index operations, which ultimately determines what data (addresses) are referenced. In order to make the index aware of the *size* of nodes without the need to access them directly, addresses carry the number of bytes the corresponding node occupies on flash. Addresses therefore are structured, they contain an erase block, a byte-offset in the block, and the size of the node stored:

$$\textbf{data type } \mathit{Address} = \texttt{@}(\texttt{block}\colon \mathbb{N}, \texttt{offset}\colon \mathbb{N}, \texttt{size}\colon \mathbb{N})$$

If an index update replaces address *adr* stored under *key* with new adr', the `rsize` field of the block *adr* belongs to is decreased by *adr*.`size`. Symmetrically, the field of adr''s block is increased. For the quality of the garbage collection this information should match the one we could obtain (inefficiently) from the index. Therefore, we prove the **invariant**

$$blocks[n].\texttt{rsize} = \sum \big\{ adr.\texttt{size} \mid adr \in \texttt{ran}(ri), adr.\texttt{block} = n \big\} \tag{10}$$

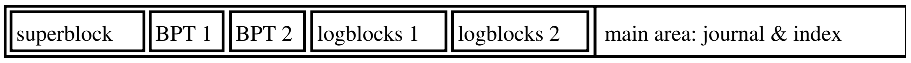
for all $n \in \texttt{dom}(blocks)$. The range operator yields the set of addresses in the index, we restrict to those addresses in block n and then sum up all their `size` fields. Note that invariant (10) is independent of the question of how many nodes are actually stored on flash in block n, i.e., it does not mention $blocks_{\downarrow}$ but only the size stored in addresses. This simplifies reasoning about changes to `rsize`.

6 Persistence: Atomic Commit & Write Buffering

The persistence layer encodes all data structures of the flash file system to bytes. It maintains the disk layout in order to decide which erase blocks are allocated for which purpose.

One challenge for the implementation and verification is again atomicity; this time in the form of the commit operation and writing of individual nodes. The second challenge is that free space management and the garbage collection (in the form of the `rsize` field) requires additional information stored per block. This information is kept in the Block Property Table (BPT) that is maintained in RAM and (in an outdated form) on flash, stored during the commit. As shown in Sec. 7, recovering the BPT after a power failure is quite delicate.

Disk Layout & Atomic Commit. The disk is partitioned into two parts:



The first part spans the superblock, a copy of the internal management data BPT from the last commit and the list of blocks allocated for the log (corresponding to *logblocks* in Sec. 4). For the BPT and log blocks, space for *two* versions is provisioned. The superblock references the "current" one to be read at startup time. The spare one is written during commit, a subsequent change of the superblock ensures atomic transition to the new state. Assuming the flash index is also written out-of-place by the index model, this already yields a correct implementation of the commit operation from Sec. 3.

The second part of the device consists of all erase blocks with group nodes, i.e., the blocks occupied by the journal, and erase blocks storing the on-flash index (not covered in this paper).

The Block Property Table (BPT) is an array with some data for each erase block of the main area:

$$\begin{aligned} &\textbf{state var } bpt \colon Array\langle BPTEntry\rangle \\ &\textbf{data type } BPTEntry = \texttt{bptentry}(\texttt{size}\colon \mathbb{N}, \texttt{rsize}\colon \mathbb{N}, \texttt{type}\colon BPTType) \\ &\textbf{data type } BPTType = \texttt{FREE} \mid \texttt{GROUP\_NODES} \mid \texttt{INDEX\_NODES} \end{aligned}$$

The BPT is consulted to find a `FREE` block when a fresh one is requested by the journal layer. The `rsize` field corresponds to the abstract counterpart shown in (6). The `size` field stores how many bytes have been written to the block.

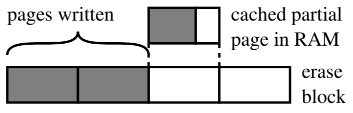


Fig. 7: Write buffer

Write Buffering. The main functionality required by the journal is appending a single (group-) node to a block. The implementation is shown in Fig. 8. It is based on a write buffer, which stores one flash page as a cache in RAM to aggregate non-aligned writes. Figure 7 visualizes how this cache is overlaid with the data on flash: the whole part marked in grey designates written bytes. Write buffers are allocated on demand and stored in the map *wbufs*.

$$\textbf{state var } \mathit{wbufs} : \mathbb{N} \nrightarrow \mathit{WBuf}$$
$$\textbf{data type } \mathit{WBuf} = \mathtt{wbuf}(\mathtt{off} : \mathbb{N}, \mathtt{buf} : \mathit{Array}\langle \mathit{Byte} \rangle, \mathtt{nbytes} : \mathbb{N})$$

In order to create a write buffer the offset where we want to start writing data must be known, which is readily available in the BPT as `size`. Procedure `persistence_add_node` then writes a header containing the length of the encoded node, the node itself and a trailer. The block number, offset and size are assembled into the returned address *adr*. A partially written node is detected by a missing trailer. Flushing of a block (`persistence_flush`) requires a write of a padding node that spans the space until the next page boundary.

Verification. The abstraction relation between the specification and implementation of the persistence layer basically states that

– the current version of *logblocks* is stored on flash,
– the BPT from the last commit is stored on flash,
– all group nodes are stored in the main area and
– the `flushindex` of each block corresponds to the exact number of nodes that have been written to flash and are no longer held in the write buffer.

Difficult in terms of verification and specification is that the encoding of all nodes in one block is not functional, since the abstraction needs to filter out padding nodes and partially written nodes at the end of each block.

7 Power Cuts & Recovery

In this section we describe how the various models interact in the event of a power cut. It is modeled as assigning arbitrary values to all in-RAM data structures. The persistent storage is left unchanged. In bottom-up fashion we give each model a chance to recover to a consistent and desirable state via the recovery operation. The machine of the journal layer, for example, starts with the recovered state of the persistence layer.

In general we show recovery property (1) for each refinement, i.e., the power cut and subsequent recovery between abstract and concrete model match, in the

```
persistence_add_node($block$, $gnode$; $adr$, $err$)
 if $\neg$ wbuf_is_buffered($block$) then  wbuf_create($block$, $bpt[block]$.size)
 let $buf$ = encode-group-node($gnode$)
  let $buf_0$ = encode-header(nodeheader($\#buf$, $false$))
   wbuf_write($block$, HEADER_SIZE, $buf_0$; $err$)
   if $err$ = ESUCCESS then  wbuf_write($block$, $\#buf$, $buf$; $err$)
   if $err$ = ESUCCESS then  wbuf_write($block$, HEADER_SIZE, trailer; $err$)
   if $err$ = ESUCCESS then
    let $size = 2 \cdot$ HEADER_SIZE $+ \#buf$
     $adr$ := @($block$, $bpt[block]$.size, $size$), $bpt[block]$.size += $size$
   else  $bpt[block]$.size = BLOCK_SIZE
```

Fig. 8: Writing a single Group Node

```
persist_recover(; $bpt$, $logblocks$)
  read_superblock(; $superblock$)
  read_log($superblock$; $logblocks$)
  read_bpt($superblock$; $bpt$)
  fix_bpt($logblocks$; $bpt$)
```

Fig. 9: Recovery of Persistence

```
jnl_recover($logblocks$; $log$)
 $log$ := [ ]
 while $logblocks \neq$ [ ] do
  persistence_read_blk($logblocks$.head; $adrs$, $nds$)
  remove_nonend_nodes(; $adrs$, $nds$)
  $log$ := $log$ + $adrs$, $logblocks$ := $logblocks$.tail
```

Fig. 10: Recovery of the Log

sense that invariants and abstraction relations hold afterwards. The difficulty from a specification and verification perspective is that different parts of the state behave differently: Some parts are restored to the state directly before the power cut while other parts are restored to the state of the last commit. Some parts need to be fixed and do not resemble any previous state. Furthermore, several aspects of recovery from power cuts leak through abstractions, making it an inherently collaborative effort of several models.

By definition (4) the journal and persistence implementation together restore to the same state and return the list of addresses of the log for its replay (see invariant (5)). The implementation of the recovery operations is shown in Fig. 9 and Fig. 10 (error handling omitted for brevity). After the persistence layer has read the necessary data from flash and fixed the outdated BPT (which is the hard part, as explained below), the journal takes over to scan the erase blocks that form the log, removing invalid group nodes at the end of each block and concatenating all the addresses. Not incidentally, this corresponds exactly to the abstraction `abs-log` in (8).

The BPT read from flash needs to be adapted in two ways: The blocks that have been allocated for group nodes since the last commit (subset of those in the log) need to be marked as allocated. The blocks constituting the log at the time of the power cut must be considered non-writable: it cannot be determined how far exactly they have been written. Therefore, we treat those blocks as full and set their `size` field in the BPT accordingly.

Abstractly, after the recovery by the persistence implementation the journal implementation sees the following changes to the group blocks *blocks*:

1. the nodes that were not yet persisted (i.e., are above `flushindex`) vanish,
2. the `rsize` field is reverted to the value from the last commit,
3. previously garbage collected and deallocated group blocks reappear.

The first point is no problem, because the abstraction (7) in terms of $blocks_{\downarrow}$ upwards to the flash store *fs* only considers the persisted nodes anyway.

The reverted `rsize` has the consequence that the invariant (10) is violated if one considers the in-RAM index before the power cut. However, after a power cut, we read the index from the last commit, too. And the index and the `rsize` fields from the last commit obviously satisfied invariant (10) at the point of commit. This establishes the invariant right after the recovery of the persistence layer and reading of the on-flash index. Replaying the index afterwards then also updates the `rsize` fields correctly.

The reappearing blocks are problematic, because they may contain garbage data (it is unknown whether they have been erased on flash or not) and reallocation is precluded until they are deallocated once again. We store the blocks that have been deallocated since the last commit abstractly, exclude their contents in the abstractions `abs-log` and `abs-fs` (and disallow reading and writing), propagating this constraint towards the upper layers. After the replay of the index by upper layers the RAM index will no longer reference them, since this was the reason for their deallocation, and we can now safely remove these blocks.

8 Related Work

NASA's proposal [19] has prompted a large body of related work, covering many aspects of file systems in general and also specific to flash memory.

High-level specifications include the early work of Morgan & Sufrin [22] and mechanized models and proofs [1,14,15,17]; a recent model of POSIX which is very complete and detailed is presented in [24]. These efforts are orthogonal to this paper, see [13] for a detailed comparison to our development. Formalizations of flash memory below the models presented here include [5].

Two developments actually connect a high-level view to the pages and blocks of flash hardware [20,8]. In both cases, only file content is mapped, written, and garbage collected at the granularity of flash pages, at the expense of extra state that is kept in memory. An encoding of the directory/file structure and any other auxiliary data structures (such as the BPT, log and on-flash index) down to flash and caching of writes are not considered. [20] deals only with crashes during a write operation and intertwines the recovery strategy with the implementation of the write operation. Some Flash Translation Layers (FTLs) and [20,8] have a page-based allocation scheme assuming additional, overwritable bits in each page that track the allocation status. These are not always present or might be used entirely for error-correction codes [30]. We have to recover newly allocated blocks and deallocate reappearing blocks after a power cut. Furthermore, the models do not consider the restriction to sequential writes within an erase block. [8] reads all pages during mounting/recovery in order to rebuild the index.

Chen et al. [6] discuss different formalisms to express crash and recovery on a high level, and settle for a pre/post verification in the Hoare-logic style, augmented with a crash specification and a designated recovery operation attached to individual operations. Very nice follow up work [7] introduces Crash Hoare Logic in more detail and presents the verification of a small but complete file system called FSCQ targeting conventional magnetic drives. In comparison, their approach requires one to reason about intermediate states using a special logic, whereas we are able to reduce the proof effort on a semantic level.

Sprenger et al. [21] consider a storage system with similar properties of that of a file system, but with a strong focus on redundancy.

9 Discussion & Conclusion

We have presented two central components for verified flash file systems, covering concepts not realistically addressed in previous work.

The work has been done in the context of the Flashix project and it is strongly connected to the design of the overall system. One observation is that it is non-trivial to find a good decomposition of the system. Since we have taken the existing implementation UBIFS as a blue-print, many concepts were already worked out properly, but isolating these from the verification point-of-view took quite some time—we estimate somewhat less than half a person-year in total for the models and proofs; the overall project effort is in the order of three to four person years. At least half can be attributed to errors and power cut safety. The specifications developed in this work specifically are in the order of 4k lines of ASM code and algebraic definitions in addition to around 800 theorems.

The large gap in representation of system state (abstract tree down to bytes) leads to a deeply nested hierarchy of layers, see the full version of Fig. 1 in [26]. It is beneficial to be able to pinpoint the individual concepts as abstract models (i.e., ASMs) in their own respect: one can verify invariants on the abstract level, executable specifications were also useful during testing and validation.

However, a deep hierarchy has the issue that models become semantically entangled, which breaks modularity in a way that is *hard* to resolve, as noted before in e.g. [2]. Resilience against hardware errors and abrupt power cuts aggravates the problem of finding suitable, sufficiently abstract specifications. It is likewise not obvious, to what extent such effects should be masked within the implementation of a specific component.

Specification entanglement manifests for example in the `flushindex` and `rsize` fields in Sec. 4 and the extra size field in addresses. Garbage collection has issues on its own: it should be pointed out that upper layers in the software stack must be able to deal with it, namely, the file system core should be agnostic to GC (which is established in terms of its specification). Another issue are the reappearing blocks in Sec. 7 caused by the fact that some internal data structures are stored only during a commit. We think that this emphasizes that specifying systems well is at least as hard as the verification itself.

It is doubtful whether it would pay off to further refactor the design, as we found that even small changes tend to affect large parts of the verification, mainly due to hardware failures and power cuts. Of course, improving tool support for such refactoring is one way to mitigate this problem.

With previous work [28,23,13] we have now completed the design of a fully functional flash file system. All models and proofs are available online at [9]. The verification is almost done (missing: parts of the B^+ trees) and we're generating preliminary C code, which is in the order of 10kLoC. An evaluation of the performance of the file system is currently under way.

Two important features that require further research are caching across POSIX operations and concurrency. Caching across operations is in principle supported by our implementation, but a suitable refinement theory still needs to be worked out. Internal concurrency for garbage collection and erasing of blocks reduces the latency of operations from the user's point-of-view. To support this eventually ($\diamond$), the semantics of our crash-refinement theory has been made compatible with the temporal logic RGITL [27] implemented by KIV.

Acknowledgement. We thank the anonymous reviewers for their detailed and helpful comments.

References

1. K. Arkoudas, K. Zee, V. Kuncak, and M.C. Rinard. On verifying a file system implementation. In *Proc. of ICFEM*, pages 373–390, 2004.
2. C. Baumann, B. Beckert, H. Blasum, and T. Bormer. Lessons Learned From Microkernel Verification – Specification is the New Bottleneck. In *SSV*, pages 18–32, 2012.
3. E. Börger. The ASM Refinement Method. *Formal Aspects of Computing*, 15(1–2):237–257, 2003.
4. E. Börger and R. F. Stärk. *Abstract State Machines — A Method for High-Level System Design and Analysis.* Springer, 2003.
5. A. Butterfield and J. Woodcock. Formalising Flash Memory: First Steps. *IEEE Int. Conf. on Engineering of Complex Computer Systems*, 0:251–260, 2007.
6. H. C., D. Ziegler, A. Chlipala, M. F. Kaashoek, E. Kohler, and N. Zeldovich. Specifying crash safety for storage systems. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*. USENIX Association, 2015.
7. H. C., D. Ziegler, A. Chlipala, N. Zeldovich, and M. F. Kaashoek. Using crash hoare logic for certifying the FSCQ file system. In *Proc. of SOSP*. ACM, 2015.
8. K. Damchoom. *An incremental refinement approach to a development of a flash-based file system in Event-B.* PhD thesis, University of Southampton, 2010.
9. G. Ernst, J. Pfähler, and G. Schellhorn. Web presentation of the Flash Filesystem. `https://swt.informatik.uni-augsburg.de/swt/projects/flash.html`, 2015.
10. G. Ernst, J. Pfähler, G. Schellhorn, D. Haneberg, and W. Reif. KIV - Overview and VerifyThis Competition. *Software Tools for Techn. Transfer*, pages 1–18, 2014.

11. G. Ernst, J. Pfähler, G. Schellhorn, and W. Reif. Modular, Crash-Safe Refinement for ASMs with Submachines. *Science of Computer Programming, ABZ 2014 special issue*, 2015 (submitted).
12. G. Ernst, G. Schellhorn, D. Haneberg, J. Pfähler, and W. Reif. A Formal Model of a Virtual Filesystem Switch. In *Proc. of Software and Systems Modeling (SSV)*, EPTCS, pages 33–45, 2012.
13. G. Ernst, G. Schellhorn, D. Haneberg, J. Pfähler, and W. Reif. Verification of a Virtual Filesystem Switch. In *Proc. of Verified Software: Theories, Tools, Experiments*, volume 8164 of *LNCS*, pages 242–261. Springer, 2014.
14. M.A. Ferreira, S.S. Silva, and J.N. Oliveira. Verifying Intel flash file system core specification. In *Modelling and Analysis in VDM: Proc. of the fourth VDM/Overture Workshop*, pages 54–71, 2008. Technical Report CS-TR-1099.
15. L. Freitas, J. Woodcock, and Z. Fu. Posix file store in Z/Eves: An experiment in the verified software repository. *Sci. of Comp. Programming*, 74(4):238–257, 2009.
16. T. Gleixner, F. Haverkamp, and A. Bityutskiy. UBI - Unsorted Block Images. `http://www.linux-mtd.infradead.org/doc/ubidesign/ubidesign.pdf`, 2006.
17. W.H. Hesselink and M.I. Lali. Formalizing a hierarchical file system. *Formal Aspects of Computing*, 24(1):27–44, 2012.
18. A. Hunter. A brief introduction to the design of UBIFS. `http://www.linux-mtd.infradead.org/doc/ubifs_whitepaper.pdf`, 2008.
19. R. Joshi and G.J. Holzmann. A mini challenge: build a verifiable filesystem. *Formal Aspects of Computing*, 19(2), June 2007.
20. E. Kang and D. Jackson. Formal Modelling and Analysis of a Flash Filesystem in Alloy. In *Proc. of ABZ*, pages 294–308. Springer, 2008.
21. O. Marić and C. Sprenger. Verification of a transactional memory manager under hardware failures and restarts. In *FM 2014: Formal Methods*, volume 8442 of *LNCS*, pages 449–464. Springer, 2014.
22. C. Morgan and B. Sufrin. Specification of the UNIX filing system. In *Specification case studies*, pages 91–140. Prentice Hall Ltd., Hertfordshire, UK, 1987.
23. J. Pfähler, G. Ernst, G. Schellhorn, D. Haneberg, and W. Reif. Formal Specification of an Erase Block Management Layer for Flash Memory. In *Hardware and Software: Verification and Testing*, volume 8244 of *LNCS*, pages 214–229. Springer, 2013.
24. T. Ridge, D. Sheets, T. Tuerk, A. Giugliano, A. Madhavapeddy, and P. Sewell. SibylFS: formal specification and oracle-based testing for POSIX and real-world file systems. In *Proc. of SOSP*. ACM, 2015.
25. G. Schellhorn. Completeness of Fair ASM Refinement. *Science of Computer Programming, Elsevier*, 76, issue 9:756 – 773, 2009.
26. G. Schellhorn, G. Ernst, J. Pfähler, D. Haneberg, and W. Reif. Development of a Verified Flash File System. In *Proc. of ABZ 2014*, volume 8477 of *LNCS*, pages 9–24. Springer, 2014.
27. G. Schellhorn, B. Tofan, G. Ernst, J. Pfähler, and W. Reif. RGITL: A temporal logic framework for compositional reasoning about interleaved programs. *Annals of Mathematics and Artificial Intelligence (AMAI)*, 71:1–44, 2014.
28. A. Schierl, G. Schellhorn, D. Haneberg, and W. Reif. Abstract Specification of the UBIFS File System for Flash Memory. In *Proceedings of FM 2009: Formal Methods*, pages 190–206. Springer LNCS 5850, 2009.
29. The Open Group. The Open Group Base Specifications Issue 7, IEEE Std 1003.1, 2008 Edition. `http://www.unix.org/version3/online.html` (login required).
30. UBI - Out-of-Band Data. `http://www.linux-mtd.infradead.org/faq/ubi.html`.
31. J. C. P. Woodcock and J. Davies. *Using Z: Specification, Proof and Refinement.* Prentice Hall International Series in Computer Science, 1996.