Universität Augsburg

Crash-Safe Refinement for a Verified Flash File System

J. Pfähler, G. Ernst, G. Schellhorn, D. Haneberg, W. Reif

Report 2014-02 April 2014

Institut für Informatik
D-86135 Augsburg

Crash-Safe Refinement for a Verified Flash File System

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

Institute for Software & Systems Engineering
University of Augsburg, Germany
{joerg.pfaehler,ernst,schellhorn,haneberg,reif}
@informatik.uni-augsburg.de

Abstract. This paper presents formal proof obligations for data refinement in the presence of unexpected crashes, notably due to a power failure. The work is part of our effort to construct a verified file system for flash memory. We apply the theory to one of the components in the flash file system, namely the erase block management layer. We show its functional correctness with respect to a high-level specification. We prove that the system can always recover from power loss to a desired state. We observe two simplifications that greatly reduce the proof effort for crashes in practice. Proofs are mechanized in the theorem prover KIV.

Keywords: Flash File System, Specification, Refinement, Wear-Leveling, Power Failure, KIV

1 Introduction

Flaws in the design and implementation of file systems already lead to serious problems in mission-critical systems. A prominent example is the Mars Exploration Rover Spirit [21] that got stuck in a reset cycle. In 2013, the Mars Rover Curiosity also had a bug in its file system implementation, that triggered an automatic switch to safe mode. The first incident prompted a proposal to verify a file system for flash memory [17,10] as a small step towards Hoare's Grand Challenge [14]. We are developing such a verified flash file system (FFS) as an implementation of the POSIX file system interface [27,8].

Flash file systems differ from traditional ones as the hardware doesn't support overwriting data in-place (in contrast to magnetic disks). In our approach, we follow the design of UBIFS [16]. It is part of the Linux kernel and implements state-of-the-art strategies to deal with the characteristics of flash memory.

In order to tackle the complexity of the verification of an entire file system implementation, we refine a top-level abstract POSIX specification in several steps down to an implementation. Figure 1 shows the high-level structure of the project. There are four conceptual layers. Each consists of one or more subcomponents, which are modeled with different degrees of abstraction. At the top-level is a specification of the functional correctness requirements. At the bottom is a driver interface model that encodes our assumptions about the hardware.

The file system implementation in between relies on a separate layer, the *Erase Block Management*, to provide advanced features on top of the hardware interface.

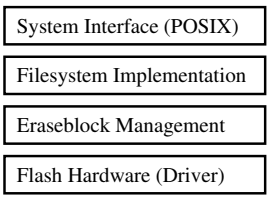

Fig. 1: FFS Layers

Besides functional correctness, it is of great interest that the file system can deal with unexpected power-failures anytime during the run of an operation. Concretely, "crash-safety" means that whenever an operation is aborted in an intermediate state, a special *recovery* operation can reconstruct a state sufficiently similar to the pre- resp. post-state of the operation. In particular, consistency of on-flash data structures must be preserved, and no previously written data may be lost. The contributions of this paper are:

- We develop an extension of *data refinement* [13] that supports the analysis of power-failures. Briefly, possible effects of power-loss and subsequent reconstruction of the state are defined by an *abstract* crash specification and recovery operation, which allows us for example to specify to which extent crashes must be handled transparently by the implementation. In addition to the normal simulation proof obligations the theory requires one to show similar commutations for each *intermediate* state.
- We apply the theory to the Erase Block Management layer presented in our previous work [20]. For this particular application we observe that the intermediate states of the flash memory are actually a subset of the final states of normal runs, due to the way nondeterministic hardware errors are handled. Therefore, for the verification a standard big-step approach is sufficient.
- We formalize the conditions for this reduction and we expect that the same reduction can be applied uniformly to the analysis of crash-safety of all refinements in our project.
- The verification can be modularized further, resulting in a single additional proof obligation for the recovery operation. This is a considerable simplification over the general case.

The theory furthermore applies to a larger class of problems where programs are aborted in response to *external* events, for example signals sent by the UNIX `kill` command, Java's (unsafe) `Thread.stop` method and unchecked exceptions.

The paper has three main parts, describing the application domain in Sec. 2, the crash-safe data refinement theory in Sec. 3, and invariants and the refinement proofs in Sec. 4. We compare to related work and draw conclusions in Sec. 5.

2 Background

This section explains the purpose of *Erase Block Management* (EBM) and briefly shows the formal models that are related to this layer. Namely, we describe the implementation in Sections 2.1 to 2.3 and the specification in Sec. 2.4. These correspond to the "concrete" resp. "abstract" data types CDT and ADT in terms of data refinement. The models are inspired by the Unsorted Block Image (UBI) [12] layer of UBIFS and the Memory Technology Device (MTD) layer of

Linux. An in-depth description of the models can be found in [20]. All models and proofs are browsable online [19].

Our specification language is based on *Abstract State Machines* [3] (ASMs). We use algebraic specifications to axiomatize data types, and the weakest-precondition calculus implemented in KIV [22] to verify properties. ASMs maintain a state as a vector of logical variables that store algebraically defined data structures. Operations are defined by abstract programs ("rules" in ASM terminology), featuring parallel (function) assignments, conditionals, loops, recursive procedures, and also nondeterministic choice.

To study crashes, we distinguish between state held in volatile memory (denoted by **RAM state** subsequently) and state stored on the flash device (**flash state**). Conceptually, all information in RAM is lost during a crash. We assume that the flash state is unaltered by crashes.

2.1 Flash Hardware

Flash memory is physically partitioned into *blocks*, each consisting of *pages* that can be empty or programmed with data. There are three main operations:
1. Read a consecutive part of a block, possibly across page boundaries. Empty pages yield default values, typically bytes `0xFF`.
2. Write/Program data to a whole page that was previously empty. There may be an additional constraint that pages in a block must be written in order [9,7].
3. Erase a whole block, i.e., empty all of its pages. This operation enables reuse of memory, though it comes at considerable costs: Erasing is slow and physically degrades the memory. The number of erase cycles until a block breaks down is thus limited—typically between 10^4 and 10^6. Such broken blocks are called *bad*.

In practice, flash memory exhibits *nondeterministic errors*. A failed access does not mean that a block is ultimately unusable. The file system may retry an operation several times or backup the data to a different block after a failed write, since a block may work perfectly again after one erase cycle. Once the file system determines a block as broken, it can set a hardware-supported bad-block marker to prevent further use of the block.

We formalize flash memory in our hardware model as an array of *physical erase blocks* (PEBs) stored in the state variable *pebs*. It is the only persistent variable in our models on this level of abstraction.

flash state $pebs : Array\langle Peb\rangle$ where
data $Peb = \texttt{peb}(\texttt{data} : Array_{\texttt{PEB\_SIZE}}\langle Byte\rangle, \texttt{fill} : \mathbb{N}, \texttt{bad} : \mathbb{B})$

Peb is an algebraic data type with one constructor, `peb`. It contains the block's data as an array `data` of bytes of uniform length `PEB_SIZE`. The page-aligned counter `fill` denotes how many bytes have been written to `data`, subsequent bytes are empty. Subdivision into pages is represented implicitly.

The operations that model the interface to the hardware are defined by ASM rules (abstract programs) that modify the flash state *pebs*. Nondeterministic errors are incorporated into each operation: Either the modification of flash

```
mtd_erase(n; err)
   { pebs[n] := peb(EMPTY_PEB, 0, false),
     err := ESUCCESS }
or { err := EIO }

mtd_markbad(n; err)
   { pebs[n].bad := true,
     err := ESUCCESS }
or { err := EIO }
```

Fig. 2: Hardware operations (MTD)

```
erase(l, v)
 for e ∈ eraseq do
  if e.lebref = (v, l) then
   eraseq -= e
   let ec = // get erase counter
   in mtd_erase(e.pnum; err)
      mtd_write_ec(e.pnum, ec + 1; err)
      if err ≠ ESUCCESS
      then mtd_markbad(e.pnum; err)
```

Fig. 3: Synchronous erase (EBM)

memory is successful, or there is an `EIO` error and the flash state is *unmodified*. In the full model, this assumption is relaxed by permitting certain kinds of data corruptions as long as they can be recognized, e.g., by checksums.

As an example, Fig. 2 shows the ASM rule to erase physical block number n, and the rule to mark a block as bad. We assume that flash (MTD) operations can be viewed as atomic state transitions, consequently, the model updates $pebs[n]$ by a single assignment in each operation.

2.2 Logical and Physical Blocks

The Erase Block Management layer provides an abstraction of flash memory based on *logical* erase blocks (LEBs) instead of physical ones. The primary task of the EBM is therefore to maintain a mapping from logical to physical blocks.

Several advanced features can be implemented with such a mapping. For example, bad blocks can be hidden from upper layers. More importantly, the EBM layer can *transparently* migrate a logical block to a different physical location. This enables *wear-leveling*, a method to distribute erase cycles evenly between physical blocks to prolong the hardware's lifetime. The basis for wear-leveling is an *atomic write* of a whole logical block, which breaks down to multiple physical writes internally. This operation is useful for applications, too.

Another benefit is that erasing of blocks (which is slow) can be performed asynchronously in the background (called *unmapping*). An application may thus reuse a logical block directly after an erase request, even before the corresponding physical erase has been performed. Finally, several *volumes* (i.e., partitions) on one device can trivially be supported. In summary, the EBM operations are `read`, `write`, `map` (allocate), `unmap` (asynchronous), `erase` (synchronous, see below).

The *forward* mapping from logical blocks to physical ones is kept in RAM, given for each volume $v : \mathbb{V}$ as an array indexed by logical block numbers:

RAM state $vols : \mathbb{V} \nrightarrow Array\langle PebRef \rangle$ where
type $PebRef = \mathbb{N} + \texttt{unmapped}$

Figure 4 shows the logical view of the device at the top with consecutive blocks numbered $0, 1, \ldots$, and the physical device at the bottom. Bold arrows denote which physical block is allocated for a logical one. For example, logical

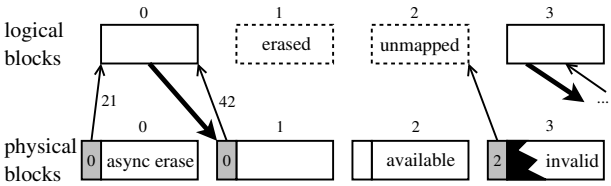

Fig. 4: Mapping of logical blocks to physical ones

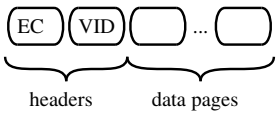

Fig. 5: Layout of a PEB

block 0 is mapped to 1, and logical blocks 1, 2 are unallocated, and we would have $vols[v] = [1, \texttt{unmapped}, \texttt{unmapped}, \ldots]$ for some volume v.

An *inverse* mapping (displayed by thin arrows) is stored on flash in the grey headers of physical blocks. The in-memory representation of the forward mapping is initially built during system startup (resp. recovery).

A logical block that has no associated physical one (such as the dashed blocks 1 and 2) is implicitly empty, i.e., it has previously been erased. As soon as a write to such a block occurs, a new PEB is allocated and the mapping is extended in both directions.

Note that the stored inverse mapping need not to be injective (as suggested by PEB 0 and 1 in Fig. 4). The recovery algorithm scans the headers of all physical blocks and resolves such ambiguities by *sequence numbers*. For example, the inverse mapping from PEB 0 (to LEB 0) has sequence number 21, which is lower than the sequence number 42 of PEB 1.

In summary, physical blocks can have the following states. En-passant we introduce several critical *invariants* of the erase block management layer.

- *Mapped* PEBs, such as block 1, have a valid header with an inverse mapping. Their respective sequence number is the highest among all PEBs with the same inverse mapping. These blocks are the only ones referred to by the forward mapping *vols* in RAM.
- *Obsolete* blocks, such as block 0, have a valid header, however, there is another PEB with a higher sequence number. Such blocks are not mapped and are scheduled for erase.
- *Pending obsolete* blocks have a valid header, however, in contrast to truly *obsolete* blocks they have the highest sequence number but are not referred to by *vols*. These blocks were recently asynchronously erased but the corresponding physical erase has not occurred yet. *Pending obsolete* blocks are *remapped* by the recovery operation, as described in Sec. 2.3.
- *Available* blocks, such as block 2, do not store an inverse mapping and can be allocated.
- *Invalid* blocks, such as block 3, contain partially written data of a failed atomic update. In order to recognize partial writes, a checksum (of the relevant prefix of the block) is stored as well. These blocks are never mapped and must be scheduled for erase, too.
- *Bad blocks* are not allocated and their content is ignored.

Pending obsolete blocks are an artefact that could be prevented with difficulty and additional flash state only. Our EBM implementation refrains from the extra

```
recover(pebs; vols, eraseq)
 sqn : 𝕍 × ℕ ↛ ℕ
 for i = 1...pebs.length do
  let peb = pebs[i] in
   if      peb.bad          then ...
   else if peb.isavailable then ...
   else let v = peb.vol, l = peb.leb in
    if      peb.isinvalid    then eraseq += eq-entry(i, none)
    else if peb.sqn < sqn(v, l) then eraseq += eq-entry(i, (v, l)) // obsolete
    else  // peb ∈ {mapped, pending, obsolete}, previous PEB obsolete
     { eraseq += eq-entry(vols[v][l], (v, l)); vols[v][l] := i; sqn(v, l) := peb.sqn }
```

Fig. 6: Recovery from Crashes (conceptually)

effort. Instead, the problem is pushed into the application (which is not a problem in practice). Alternatively, a less efficient *synchronous* erase can be used, which disposes of *all pending* blocks for a given mapping.

Fig. 5 shows the layout of a PEB. The first two pages store two headers. The remaining pages store application data. The first page contains an *erase counter* (EC) associated with the physical block. The second page of allocated PEBs contains the inverse mapping (thin arrows in Fig. 4) in the *volume identifier header* (VID). Two headers are necessary, because every PEB stores an erase counter, but only once a PEB is allocated an inverse mapping is required.

Both kinds of obsolete blocks are kept in the *erase queue*. It is used to assign work to the corresponding background operation. For synchronous erasure of *one* LEB $(v, l) \in \mathbb{V} \times \mathbb{N}$ it is necessary to locate *all* PEBs that belonged to (v, l). To easily locate them without reading from flash, each entry of the queue caches the inverse mapping stored in the corresponding PEB.

RAM state $eraseq : Seq\langle EraseqEntry\rangle$ where

data $EraseqEntry = \texttt{eq-entry}(\texttt{pnum} : \mathbb{N}, \texttt{lebref} : LebRef)$

type $LebRef = \texttt{none} + \mathbb{V} \times \mathbb{N}$

The ASM rule for synchronous erasure is shown in Fig. 3. All relevant entries in the erase queue are erased and a new EC header is written. If an error occurs, the block is marked as bad. The actual model performs several retries, which is omitted here. Note that this operation inherits from the failure cases of the MTD operations that there is a run that doesn't modify the flash state. We will need such runs in Sec. 3.4.

2.3 Recovery

Recovery rebuilds the RAM state from flash after a crash. Fig. 6 shows conceptually how the recovery operation works. The actual implementation calls flash operations instead of directly using selectors on PEBs. It scans the headers of every physical block and incrementally rebuilds the mapping table *vols*. The sequence number of the most recent mapping encountered during the scan is cached. For each physical block it is determined in which state according to the

classification of Sec. 2.2 it is, taking into consideration the mappings (and their sequence numbers) that were already seen. *Obsolete* blocks may be mapped temporarily until the block with the higher sequence number is encountered. *Pending obsolete* blocks can not be distinguished from *mapped* blocks and are therefore also mapped.

2.4 The Specification

The specification of the Erase Block Management layer is designed to be as simple as possible. Its state collapses the in-RAM mapping and the contents of a LEB into one data structure *avols*.

state $avols : \mathbb{V} \nrightarrow Array\langle Leb \rangle$ where

data $Leb = \texttt{mapped}(\texttt{data}: Array_{\texttt{LEB\_SIZE}}\langle Byte \rangle, \texttt{fill} : \mathbb{N}) \mid \texttt{unmapped} \mid \texttt{erased}$

The operations are specified as an atomic modification of the state. Asynchronous and synchronous erasure for example just set the given LEB to `unmapped` resp. `erased`. The limitation to sequential writes is achieved by the precondition that the offset of every write must be above `fill`.

The distinction between `unmapped` and `erased` is introduced to specify the effects of a crash and subsequent recovery (which is complex in the implementation) as simple as possible:

```
abstract_recover(avols)
  choose avols' with avols ⊆ avols' in
    avols := avols'
```

Informally, the relation $\subseteq$ says that the state afterwards has "more information" than before. More formally, the same volumes are present in both states and have the same size. The only difference between the states is that LEBs that were `unmapped` in *avols*, may be arbitrary in $avols'$. This captures the effect that *pending obsolete* blocks may reappear during the recovery.

3 Crash-Safe Refinement

This section reviews standard data refinement and introduces the notation for program semantics. In Sec. 3.3 we present the theoretical contribution: an extension of data refinement for the analysis of crash-safety. The immediate proof obligations for (forward) simulation of crashing operations consider all intermediate states of both the abstract and concrete run. The proof effort can be reduced in practice, and we present an approach to do so in Sec. 3.4.

3.1 Data Refinement

Data refinement [13,6] is a formal theory that permits substitution of an *abstract* data type ADT (the specification) by a *concrete* (refined) version CDT (the implementation). CDT "refines" ADT, written $CDT \sqsubseteq ADT$, if the two cannot be distinguished just by invoking sequences of operations.

Formally, a data type $DT = (S, Init, Fin, (Op_i)_{i \in I})$ defines the space S of its values, initialization $Init \subseteq G \times S$, finalization $Fin \subseteq S \times G$ (for some global state G), and a family of operations $Op_i \subseteq S \times S$ indexed by a set of names I.

A concrete data type $CDT = (CS, CInit, CFin, (COp_i)_{i \in I})$ refines an abstract data type $ADT = (AS, AInit, AFin, (AOp_i)_{i \in I})$ iff for all sequences of indices $is = i_1, \ldots, i_n \in I^*$:

$$CInit \fatsemi COp_{is} \fatsemi CFin \subseteq AInit \fatsemi AOp_{is} \fatsemi AFin \tag{1}$$

where $\fatsemi$ denotes relational composition and $Op_{i_1,\ldots,i_n} := Op_{i_1} \fatsemi \ldots \fatsemi Op_{i_n}$.

Refinement can be proved per operation by forward simulation with a simulation relation $R \subseteq AS \times CS$ that has the following properties:

$$\begin{array}{lll} \text{initialization} & CInit \subseteq AInit \fatsemi R & \\ \text{correctness} & R \fatsemi COp_i \subseteq AOp_i \fatsemi R & \text{for all } i \in I \\ \text{finalization} & R \fatsemi CFin \subseteq AFin & \end{array} \tag{2}$$

3.2 Semantics

A *data type specification* $(S, Init, Fin, (\mathtt{P}_i)_{i \in I})$ is similar to a data type, but the operations are given as programs (corresponding to ASM rules in this work).

Such a specification induces a data type $DT = (S, Init, Fin, (Op_i)_{i \in I})$ where operation Op_i is defined in terms of the semantics of program $\mathtt{P}_i$. To reason about intermediate steps later on in Sec. 3.3, we base this work on a small step semantics with atomic steps $\langle p, \sigma \rangle \to \langle p', \sigma' \rangle$ of program p from state σ to σ' with remaining program p'. Atomic steps are for example assignments and evaluation of conditionals.

Op_i is simply the pairs of states that are produced by terminating runs[1] of $\mathtt{P}_i$, i.e, any number of steps of $\mathtt{P}_i$ terminating with empty remaining program ϵ:

$$Op_i := \{(\sigma, \sigma') \mid \langle \mathtt{P}_i, \sigma \rangle \to^* \langle \epsilon, \sigma' \rangle\}$$

A standard definition of $\to$ can be found in [1]. In KIV, we actually use the more elaborate trace-based semantics of RGITL [24], which also supports concurrent reasoning. The semantics of control-state ASMs [3] is adequate, too. Note that for simplicity we assume local variables, input/output, and call stacks to be part of S. For information on the use of ASMs for refinement see [2,23].

As a special case, we consider all *flash operations* as atomic steps, since their effect occurs atomically. For example, the trace of $\mathtt{P} \equiv \mathtt{erase}$ shown in Fig. 3 interleaves 1) steps that assign to RAM variables only and 2) MTD operations.

3.3 Extension to Crashes

A data type specification with crash behavior is a tuple

$$(S, Init, Fin, (\mathtt{P}_i)_{i \in I}, Crash, \mathtt{Recover})$$

with state space S, initialization $Init$ and finalization Fin, where programs $\mathtt{P}_i$ define the operations, similar to an ordinary data type specification in Sec. 3.1.

[1] This relational view is sufficient here and in the rest of the paper, since we prove that all programs terminate, i.e., $\mathbf{wp}(p, \mathtt{true})$, liftings with $\bot$ are thus unnecessary.

The relation $Crash \subseteq S \times S$ specifies the effect of a crash. Intuitively, its purpose is to remove from the current state all information stored in volatile memory (RAM). In particular, we assume that the output of a crashed operation is lost.

Recovery is implemented by the program `Recover`, which must be called after a crash (and only then) and before any subsequent operation on the data type. The behavior observed by a program using the data type is therefore that either a call to an operation is executed normally (and may or may not terminate) or an operation is interrupted in an intermediate state immediately followed by the effects of a crash and subsequent recovery.

The data type $DT_\lightning$ specified by $(S, Init, Fin, (\mathtt{P}_i)_{i\in I}, Crash, \mathtt{Recover})$ is defined in terms of two semantics for its operations. Normal runs of an operation Op_i are defined as usual

$$\begin{aligned} Op_i &:= \{(\sigma,\sigma') \mid \langle \mathtt{P}_i, \sigma\rangle \to^* \langle \epsilon, \sigma'\rangle\} \qquad \text{and} \\ Recover &:= \{(\sigma,\sigma') \mid \langle \mathtt{Recover}, \sigma\rangle \to^* \langle \epsilon, \sigma'\rangle\} \end{aligned} \tag{3}$$

Crashed runs $Op_{i\lightning} \subseteq S \times S$ execute program $\mathtt{P}_i$ only partially, followed by the crash and a complete[2] execution of the recovery operation:

$$Op_{i\lightning} := \{(\sigma,\sigma') \mid \exists\, p.\ \langle \mathtt{P}_i, \sigma\rangle \to^* \langle p, \sigma'\rangle\} \fatsemi Crash \fatsemi Recover \tag{4}$$

Observable indices of the data type $DT_\lightning$ are therefore $I_\lightning := I \cup \{i_\lightning \mid i \in I\}$ and the runs produced by a program using the data type can then be characterized by $Init \fatsemi Op_{is\lightning} \fatsemi Fin$ with $is_\lightning \in I_\lightning^*$.

Definition 1 (Data types with crash behavior). *A data type specification $(S, Init, Fin, (\mathtt{P}_i)_{i\in I}, Crash, \mathtt{Recover})$ with crash behavior induces a standard data type $DT_\lightning$ as follows*

$$DT_\lightning := (S, Init, Fin, (Op_i)_{i\in I\lightning})$$

From this embedding all properties of standard data refinement hold. Specifically, it is possible to prove a refinement $CDT_\lightning \sqsubseteq ADT_\lightning$ by a forward simulation $R \subseteq AS \times CS$ satisfying (2) and the additional condition

$$\text{correctness wrt. crashes} \qquad R \fatsemi COp_{i\lightning} \subseteq AOp_{i\lightning} \fatsemi R \quad \text{for all } i \in I \tag{5}$$

However, this property is difficult to establish, because it is necessary to reason about every intermediate step of the execution. Therefore, theorems reducing the proof effort in specific settings are desirable.

3.4 Reduction to Completed Operations

Purpose of this section is to get rid of the crash-semantics $Op_{i\lightning}$ in practice and thus to eliminate the need for small-step reasoning. Furthermore, crash-safety proofs can be modularized so that it is sufficient to consider the refinement of crash+recovery in isolation: $R \fatsemi CCrash \fatsemi CRecover \subseteq ACrash \fatsemi ARecover \fatsemi R$.

The first simplification is not tied to a particular application: The abstract run witnessing proof obligation (5) can always be chosen to be a complete run.

[2] Crashes during recovery can be neglected if $Crash \fatsemi Recover$ does not modify persistent state. Of course, a suitable proof obligation could be defined.

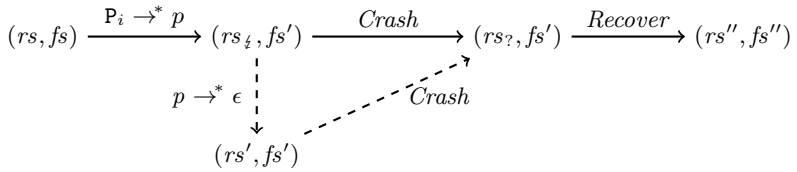

Fig. 7: Completion of a trace with unaltered flash state.

Lemma 1 (complete abstract runs). *The following proof obligation for crashed runs is* sufficient *and can be employed instead of* (5)*:*

$$R \fatsemi COp_{i\lightning} \subseteq (AOp_i \cup Id) \fatsemi ACrash \fatsemi ARecover \fatsemi R$$

where Id is the identity relation (witnessing concrete runs that crash without any flash modifications). Furthermore, the condition is trivially complete *if the program defining AOp_i has one step only.*

Proof. From (3) and (4) for $p := \epsilon$ we have $AOp_i \fatsemi ACrash \fatsemi ARecover \subseteq AOp_{i\lightning}$. Similarly, taking the whole program for p in (4) justifies the inclusion of Id. Lemma 1 then holds by transitivity of $\subseteq$ and monotonicity of $\_ \fatsemi R$ wrt. $\subseteq$. □

The hardware model in Sec. 2.1 has the property that there is always a run with a hardware error, leaving the flash state unaltered. Consequently, the same holds for all sequences of flash operations and thus all erase block management operations. This makes the set of intermediate flash states a subset of the final ones and one has to consider only these for the verification.

This second simplification is specific to the separation of $S = RS \times FS$ into volatile RAM state RS and persistent Flash state FS with $\sigma = (rs, fs)$. Crashes change the RAM state arbitrarily but leave the flash state intact:

$$Crash := \{(rs, fs, rs', fs') \mid fs = fs'\} \tag{6}$$

Figure 7 illustrates the basic idea. Assume a partial run of program $\mathtt{P}_i$ resulting in state $(rs_\lightning, fs')$, crashing to $(rs_?, fs')$ with an arbitrary RAM state $rs_?$. The final state after recovery is (rs'', fs''). We can construct the alternative dashed run by executing the remaining program p without modifying the flash state: for all subsequent flash operations we pick the error cases (c.f. Fig. 2).

Definition 2 (unaltered state assumption). *A program $\mathtt{P}_i$ satisfies the* unaltered state assumption*, if for every partial run $\langle \mathtt{P}_i, rs, fs\rangle \to^* \langle p, rs_\lightning, fs'\rangle$ a completion $\langle p, rs_\lightning, fs'\rangle \to^* \langle \epsilon, rs', fs'\rangle$ with the same final flash state fs' exists.*

Lemma 2 (completion of crashed runs). *If $\mathtt{P}_i$ satisfies assumption 2, then*

$$Op_{i\lightning} \subseteq Op_i \fatsemi Crash \fatsemi Recover \quad \text{for all } i \in I$$

Proof. Let $(rs, fs, rs'', fs'') \in Op_{i\lightning}$ be arbitrary. Then by definition (4) there is a partial run $\langle \mathtt{P}_i, rs, fs\rangle \to^* \langle p', rs_\lightning, fs'_1\rangle$ with $(rs_\lightning, fs'_1, rs_?, fs'_2) \in Crash$ and $(rs_?, fs'_2, rs'', fs'') \in Recover$. By assumption, there is also a final state (rs', fs'_1) of the complete run of $\mathtt{P}_i$. By (6) $fs'_1 = fs'_2$, and—since $Crash$ admits arbitrary RAM transitions—we furthermore have $(rs', fs'_1, rs_?, fs'_2) \in Crash$. It follows that $(rs, fs, rs'', fs'') \in Op_i \fatsemi Crash \fatsemi Recover$. □

Corollary 3 (complete concrete runs) *Given that CDT satisfies assumption 2, the following proof obligation for crashed runs is* sufficient *and can be employed instead of* (5)*:*

$$R \fatsemi COp_i \fatsemi CCrash \fatsemi CRecover \subseteq AOp_{i\lightning} \fatsemi R$$

Proof. By Lemma 2, transitivity of $\subseteq$ and monotonicity of $R \fatsemi \_$ wrt. $\subseteq$. □

The unaltered state assumption 2 can be broken down to a property about single steps as follows:

Lemma 4 (trace existence) *If* $\mathtt{P}_i$ *always terminates and all call to flash operations* $\mathtt{mtd\_}op(\ldots)$ *can leave the flash unmodified, then* $\mathtt{P}_i$ *satisfies definition 2.*

Proof. Let $\langle \mathtt{P}_i, rs, fs\rangle \to^* \langle p, rs_\lightning, fs'\rangle$ be a partial run. We show that for every n there is a remaining program $p'' \neq \epsilon$ and RAM state rs'' with

$$\text{a)}\ \ \langle p, rs_\lightning, fs'\rangle \to^{\leq n} \langle \epsilon, rs'', fs'\rangle \quad \text{or} \quad \text{b)}\ \ \langle p, rs_\lightning, fs'\rangle \to^{n} \langle p'', rs'', fs'\rangle \tag{7}$$

This means that it is possible to construct a run that leaves the flash state unmodified, and a) terminates in at most n steps or b) has length n. The proof is by induction over n. In the inductive case $n \mapsto n+1$:

Given case a) of the induction hypothesis, (7) trivially holds for $n+1$, too. Otherwise, b) yields $\langle p, rs_\lightning, fs'\rangle \to^n \langle p'', rs'', fs'\rangle$ with $p'' \neq \epsilon$.

If $p'' = \mathtt{mtd\_}op(\ldots); p'''$ for some flash operation $\mathtt{mtd\_}op$, then $\langle p'', rs'', fs'\rangle \to \langle p''', rs''', fs'\rangle$ holds for some rs''' by assumption. Otherwise, no potential first step of p'' modifies the flash state and at least one such step exists, since all runs terminate (and therefore do not get stuck). In both cases the trace is extended by one step and either a) or b) for $n+1$ follows.

Since all runs of $\mathtt{P}_i$ terminate by assumption, (7) implies that a completion without further flash modifications exists as required by definition 2. □

Theorem 5 (reduction) *If* $CDT_\lightning$ *satisfies the unaltered state assumption, and all operations of* $CDT_\lightning$ *and* $ADT_\lightning$ *terminate, a refinement* $CDT_\lightning \sqsubseteq ADT_\lightning$ *holds if in addition to the ordinary proof obligations given in Sec. 3.1, just one additional property for crash and subsequent recovery can be shown:*

$$\text{recovery} \qquad R \fatsemi CCrash \fatsemi CRecover \subseteq ACrash \fatsemi ARecover \fatsemi R \tag{8}$$

Proof. By combining the above commutation with correctness (2) of each operation $i \in I$, Lemma 1, and Corollary 3. □

4 Correctness of the EBM Implementation

This section outlines the invariants and proofs for crash-safe refinement of the specification (Sec. 2.4) to the EBM implementation (Sections 2.1 to 2.3). The operations are $I = \{\mathtt{read}, \mathtt{write}, \mathtt{map}, \mathtt{unmap}, \mathtt{erase}, \mathtt{wearlevel}, \mathtt{backgrounderase}\}$ as outlined in Sec. 2.2, where the last two are background operations invisible on the abstract layer.

The abstract data type ADT_ℓ has only one state variable, *avols*, that is considered to be persistent, and thus $ACrash = Id$, and recovery is specified by the program `abstract_recover` shown in Sec. 2.4.

The concrete data type CDT_ℓ has persistent state *pebs*. The RAM state includes the block mapping *vols*, the erase queue and other data structures that we have omitted from this paper. *CCrash* only guarantees that *pebs* is unmodified, all other state becomes arbitrary. We have shown the recovery operation `recover` in Fig. 6.

Every operation of our hardware model (Sec. 2.1) has the possibility to fail nondeterministically without effect. In particular, it is easy to check for all MTD operations the syntactic shape `atomic modification or fail`. According to Lemma 4 and Theorem 5 the standard data refinement proof obligations and separately (8) for crash followed by the recovery remain to be shown. Of course, we prove termination and that all system invariants stated here (and others) are preserved by all operations.

4.1 Refinement of Normal Operations

Abstract and concrete states are related as follows. A logical block l in volume v that is mapped to a physical block n is also mapped in the abstract states *avols* with the same (application) data. Unmapped LEBs are erased or unmapped abstractly. Formally:

$$avols[v][l] = \texttt{mapped}(data, fill) \qquad \text{when} \qquad vols[v][l] = n$$
$$avols[v][l] \in \{\texttt{erased}, \texttt{unmapped}\} \qquad \text{when} \qquad vols[v][l] = \texttt{unmapped}$$

where *data* is the sub-array of $pebs[n]$ starting at the third page, and the abstract counter $fill \geq pebs[n].\texttt{fill}$ is at least as high as the concrete counter.[3] These two conditions are sufficient for correctness (2) of all seven operations in I. □

4.2 Refinement of Recovery

The proofs for the correctness of recovery (8) are much more involved and require additional knowledge about the state. A major difficulty is to come up with a loop invariant for `recover`. The intermediate value of *vols* is not necessarily related to the mapping before the crash, since *vols* may refer to *obsolete* PEBs temporarily. The following *maximality criterion* establishes the connection:

Definition 3 (maximal mappings). *A* candidate *for the mapping of a logical block (v, l) is a physical block that stores (v, l) as inverse mapping in its header and is* mapped, pending obsolete, or truly obsolete. *The* best candidate *has the highest sequence number among these.*

A mapping table vols is maximal, *if $vols[v][l]$ is the number of the best candidate for all (v, l), if it exists, and* `unmapped` *otherwise.*

A mapping vols is partially maximal, *if $vols[v][l] \neq$* `unmapped` *implies that $vols[v][l]$ is the best candidate.*

[3] During wear-leveling the fill count may decrease as part of the migration of data. Allowing a higher fill count in the specification hides the effect of wear-leveling.

A partial maximal mapping *vols* is not necessarily maximal, since some logical blocks may be unmapped, namely those blocks that are *pending obsolete*.

Proposition 6 *The recovery loop of concrete operation* `recover` *has the invariant that vols is maximal with respect to the subrange* $\{1, \ldots, i\}$*, where* i *is the loop counter, and thus the result is a maximal mapping for pebs.*

Theorem 7 (recovery) `recover` *refines* `abstract_recover`*, i.e.,* (8) *holds.*

Proof. The high-level argument can be summarized as follows:
1. Establish the system invariant that *vols* is partially maximal
2. The mapping $vols'$ produced by `recover` is maximal by Proposition 6
3. Uniqueness of best candidates implies $vols \subseteq vols'$, defined by

   $$\forall v, l.\ vols[v][l] \neq \texttt{unmapped} \rightarrow vols'[v][l] = vols[v][l]$$

   (There is an additional system invariant that sequence numbers are unique.)
4. Show that $\subseteq$ propagates through the simulation relation R:
   $avols \subseteq avols'$ holds, as required by the abstract recovery in Sec. 2.4. □

Step 4. needs another property that must be included in R, namely that *pending obsolete* blocks are not `erased` in the abstract layer, because only `unmapped` blocks may reappear. This can be formalized by referring to the erase queue:

$$avols[v][l] = \texttt{erased} \quad \text{implies} \quad \texttt{eq-entry}(n, (v, l)) \notin eraseq \quad \text{for all } n$$

5 Related Work & Conclusion

In this paper, we present the mechanized verification of an erase block management layer for flash memory, dealing with nondeterministic hardware errors, including data corruption. We demonstrate safety of the layer against unexpected power loss. We develop a novel extension to data refinement for this purpose, which considers crashes as an additional and orthogonal concept. We provide reduction criterions to greatly reduce the proof effort in practice, based on a correspondence between crashes and nondeterministic hardware failures.

Recovery from failure has been studied quite a bit in the context of transactions. However, we are not aware of an approach that supports a uniform way to specify unexpected crashes or that integrates into refinement. The main difference is that we have to consider crashes at each program location, whereas rollback of transactions is triggered explicitly, see for example Hoare's work on compensable transactions [15]. Freytag et. al. [11] use a specialized predicate transformer semantics to study crashes, but their approach is very specific to databases and too restricted for our purpose.

The main challenges of the verification are that mappings can reappear after a power loss as a consequence of asynchronous erase and that wear-leveling can produce *invalid* blocks.

Related work on verification of flash memory access includes [18] (using Alloy), [26] (Promela/Spin), and [5] (Event-B). We think that the limitations of

real hardware are not addressed adequately. In particular, all three formalizations specify mappings at the granularity of pages, which is in principle less space-efficient and leads to longer mount times. More importantly, the existence of additional bits per page is assumed to (in-)validate mappings. This simplifies atomic migration of blocks greatly, but modern hardware tends not to support such bits [28]. The same criticism applies to the assumption made by all three models that pages in a block can be written in any order.

Models [18,26] are validated by bounded model-checking and they are tuned accordingly to prevent state space explosion. In [18] the effect of power-loss is intertwined with the write operation, and it is therefore hard to judge the adequacy of the model.

Except for the low-level flash model [4], existing work neglects data corruptions as far as we know. We are not aware of the verification of an application layer (i.e., the EBM) that has to deal with such corruptions.

The size of the formal models is around 2300 lines, half of which is ASM code, the rest can be attributed to algebraic specifications (i.e., data types, functions and predicates including invariants, not including the KIV libraries). We estimate that the net-effort for specification and verification was seven person-months.

Ongoing work integrates the models with the actual Linux MTD interface so that we can run our EBM on real hardware. In the future, we will apply the crash-safe refinement theory to other layers in our flash file system, e.g. [25,8].

Acknowledgement. We thank Timo Hochberger for digging through the UBI documentation and source code and creating the initial models.

References

1. K.R. Apt, F. S. de Boer, and E.-R. Olderog. *Verification of Sequential and Concurrent Programs*. Springer-Verlag, 3rd edition, 2009.
2. E. Börger. The ASM Refinement Method. *Formal Aspects of Computing*, 15(1–2):237–257, 2003.
3. E. Börger and R. F. Stärk. *Abstract State Machines — A Method for High-Level System Design and Analysis*. Springer, 2003.
4. A. Butterfield and J. Woodcock. Formalising Flash Memory: First Steps. *IEEE Int. Conf. on Engineering of Complex Computer Systems*, 0:251–260, 2007.
5. K. Damchoom and M. Butler. Applying Event and Machine Decomposition to a Flash-Based Filestore in Event-B. In M.V. Oliveira and J. Woodcock, editors, *Formal Methods: Foundations and Applications*, pages 134–152. Springer, 2009.
6. W. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*, volume 47 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1998.
7. Samsung Electronics. Page program addressing for MLC NAND application note, 2009. `http://www.samsung.com`.
8. 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, pages 242–261. Springer, 2014.

9. Intel Corporation et al. *Open NAND Flash Interface Specification*, June 2013. URL: `www.onfi.org`.
10. L. Freitas, J. Woodcock, and A. Butterfield. POSIX and the Verification Grand Challenge: A Roadmap. In *ICECCS '08: Proc. of the 13th IEEE Int. Conf. on Engineering of Complex Computer Systems*, 2008.
11. J.C. Freytag, F. Cristian, and B. Kähler. Masking System Crashes in Database Application Programs. In *Proc. of the 13th Int. Conf. on Very Large Data Bases*, VLDB '87, pages 407–416. Morgan Kaufmann Publishers Inc., 1987.
12. T. Gleixner, F. Haverkamp, and A. Bityutskiy. UBI - Unsorted Block Images. `http://www.linux-mtd.infradead.org/doc/ubidesign/ubidesign.pdf`, 2006.
13. C.A.R. Hoare. Proof of Correctness of Data Representation. *Acta Informatica*, 1:271–281, 1972.
14. C.A.R. Hoare. The verifying compiler: A grand challenge for computing research. *Journal of the ACM*, 50(1):63–69, 2003.
15. C.A.R. Hoare. Compensable Transactions. volume 9 of *NATO Security through Science Series - D: Information and Communication Security*, pages 116–134. IOS Press, 2007.
16. A. Hunter. A brief introduction to the design of UBIFS. `http://www.linux-mtd.infradead.org/doc/ubifs_whitepaper.pdf`, 2008.
17. R. Joshi and G.J. Holzmann. A mini challenge: build a verifiable filesystem. *Formal Aspects of Computing*, 19(2), June 2007.
18. E. Kang and D. Jackson. Designing and Analyzing a Flash File System with Alloy. *Int. J. Software and Informatics*, 3(2-3):129–148, 2009.
19. J. Pfähler, G. Ernst, D. Haneberg, G. Schellhorn, and W. Reif. KIV Models and Proofs of the Erase Block Management Layer, 2013. `http://www.informatik.uni-augsburg.de/swt/projects/flash.html`.
20. J. Pfähler, G. Ernst, G. Schellhorn, D. Haneberg, and W. Reif. Formal Specification of an Erase Block Management Layer for Flash Memory. In Valeria Bertacco and Axel Legay, editors, *Hardware and Software: Verification and Testing*, volume 8244 of *LNCS*, pages 214–229. Springer International Publishing, 2013.
21. G. Reeves and T. Neilson. The Mars Rover Spirit FLASH anomaly. In *Aerospace Conference*, pages 4186–4199. IEEE Computer Society, 2005.
22. W. Reif, G. Schellhorn, K. Stenzel, and M. Balser. Structured Specifications and Interactive Proofs with KIV. In *Automated Deduction—A Basis for Applications*, volume II, pages 13–39. Kluwer, Dordrecht, 1998.
23. G. Schellhorn. Completeness of Fair ASM Refinement. *Science of Computer Programming, Elsevier*, 76, issue 9, 2009.
24. G. Schellhorn, B. Tofan, G. Ernst, J. Pfähler, and W. Reif. RGITL: A Temporal Logic Framework for Compositional Reasoning about Interleaved Programs. *AMAI*, 2014. (appeared online first, draft available at `https://swt.informatik.uni-augsburg.de/swt/projects/RGITL.html`).
25. 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.
26. P. Taverne and C. Pronk. RAFFS: Model Checking a Robust Abstract Flash File Store. In *Proc. of the 11th Int. Conf. on Formal Engineering Methods: Formal Methods and Software Engineering*, ICFEM '09, pages 226–245. Springer-Verlag.
27. 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).
28. UBI - Out-of-Band Data Area. `http://www.linux-mtd.infradead.org/faq/ubi.html`.