# Formal Specification of an Erase Block Management Layer for Flash Memory [⋆]

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 work presents a formal specification and an implementation of an erase block management layer and a formal model of the flash driver interface. It is part of our effort to construct a verified file system for flash memory. The implementation supports wear-leveling, handling of bad blocks and asynchronous erasure of blocks. It uses additional data structures in RAM for efficiency and relies on a model of the flash driver, which is similar to the Memory Technology Device (MTD) layer of Linux. We specify the effects of unexpected power failure and subsequent recovery. All models are mechanized in the interactive theorem prover KIV.

**Keywords:** Flash File System, Specification, Refinement, Wear-Leveling, Power Failure, UBI, MTD, 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 [25] that got stuck in a reset cycle. This incident prompted a proposal to verify a file system for flash memory [18,12] as a small step towards Hoare's Grand Challenge [15]. In 2013, the Mars Rover Curiosity also had a bug in its file system implementation, that triggered an automatic switch to safe mode.

We are developing such a verified flash file system (FFS) as an implementation of the POSIX file system interface [29], using UBIFS [16]—a state-of-the-art FFS implemented in Linux—as a blueprint. 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.

File systems for flash memory differ from traditional ones because the hardware does not support overwriting data in-place (in contrast to magnetic disks).

[⋆] The final publication is available at Springer via `http://dx.doi.org/10.1007/978-3-319-03077-7_15`

The memory is physically partitioned into *blocks*, each consisting of an array of *pages* that can be empty or programmed with data. There are three 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. Typically, there is an additional constraint that pages in a block must be written in order [11,8]. 3) Erase a whole block, i.e., empty all of its pages. The erase 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 – between $10^4$ and $10^6$ for typical hardware. Such broken blocks are called *bad*.

To deal with these characteristics, data is always written to new locations (out-of-place updates); and erasing is performed asynchronously and in parallel to read/write access to the flash device. The software component responsible for this is called the *Erase Block Management* (EBM) layer. It maintains the information which blocks are currently available. The interface offered to clients mirrors the hardware operations, but it is based on *logical* block numbers instead of physical ones. The primary task of the EBM is therefore to maintain a mapping from logical to physical block numbers.

Several significant benefits follow from such a mapping. The EBM layer can *transparently* migrate a logical block to a different physical one. This enables *wear-leveling*, a method to distribute erase cycles evenly between physical blocks to prolong the hardware's lifetime. Furthermore, the client may reuse a logical block number after issuing an erase request, even before the corresponding physical erase has been performed.

This work presents the formal models of our project that are related to erase block management. As the bottom layer (Sec. 2) we specify a thin abstraction of the driver for flash memory that supports the operations read, write and erase. It is modeled after the *Memory Technology Device* (MTD) interface of Linux. We also define a *simple* EBM specification (Sec. 3) to capture the behavior visible to the upper layers. The main design goal is to abstract from implementation details as far as possible to facilitate the verification of clients wrt. the specification. Note that this abstract model only needs to consider logical blocks. Finally, we give an implementation (Sec. 4) that supports wear-leveling, handling of bad blocks and asynchronous erasure of blocks using additional data structures in RAM for efficiency. Its design is inspired by the state-of-the-art *Unsorted Block Image* layer (UBI) [22,14]. Our implementation also provides strong guarantees in the event of an unexpected power failure. However, the effects are subtle and visible to the client (and thus occur in the abstract EBM as well). For the EBM specification and the MTD layer, we also contribute a proof of certain invariants.

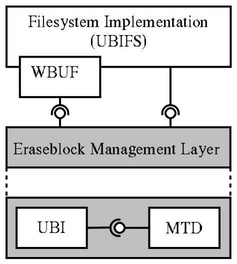
Filesystem Implementation (UBIFS)
WBUF
Eraseblock Management Layer
UBI
MTD
Fig. 1: Lower Layers

Figure 1 visualizes some of the layers of our FFS. The part shaded in grey is subject of this paper, namely the abstract EBM, the implementation UBI and driver abstraction MTD. The dashed lines indicate functional equivalence, or more formally, *refinement* relations. The erase block management is utilized by the file system either directly or through a write-back cache (the write buffer "WBUF"). The interface symbol $-\!\circledcirc\!-$ denotes dependencies between the components. The refinement is already proved, but a description is out of scope for this paper. A correctness proof of the FFS then only has to consider the abstract specification of an EBM's behavior, which is much more suitable for the verification of clients – especially wrt. the effects of unexpected power failure.

We have previously published models of the top-level POSIX specification [10], of the Virtual Filesystem Switch (VFS) [9] and of an abstract version of UBIFS [28]; [10] also presents a correctness proof of VFS. These models constitute the upper layers of the refinement stack that are not shown in Fig. 1.

We use KIV to mechanize our models as ASMs [1] based on structured algebraic specifications [26] with freely and non-freely generated algebraic datatypes. For proofs about programs we use the wp-calculus. All our models and proofs are available online [24].

## 2 Hardware Model (MTD)

This section defines our assumptions about the hardware, captured by the behavior of an abstract interface representing the driver.

Flash memory is organized as an array of *physical erase blocks* (PEBs):

$$\begin{aligned}&\textbf{state var } \mathit{pebs} : \mathit{Array}\langle \mathit{Peb}\rangle \qquad \text{where}\\ &\textbf{data } \mathit{Peb} = \mathtt{peb}(\mathtt{data} : \mathit{Array}_{\mathtt{PEB\_SIZE}}\langle \mathit{Byte}\rangle, \mathtt{fill} : \mathbb{N}, \mathtt{bad} : \mathbb{B})\end{aligned} \tag{1}$$

Each PEB stores a byte-array `data` of fixed length `PEB_SIZE` that is implicitly partitioned into pages of length `PAGE_SIZE`. A PEB stores a page-aligned counter `fill` that tracks the part of the block that contains programmed pages, i.e., only data above `fill` is known to be `EMPTY` and can be written to. Note that the fill counter cannot be accessed by software. It is an auxiliary state only used to enforce that pages are written sequentially and never overwritten. PEBs also carry a hardware-supported marker `bad` that is set by the EBM or the file system after access failures to prevent future usage of the block.

Figure 2 shows the specification of the operations on this layer. Value parameters are separated from reference parameters by semicolon. The state variable is passed implicitly. The `if`-test at the beginning of operations reflects the *precondition*. With the exception of `mtd_isbad`, each operation requires that the respective physical erase blocks is not marked as bad. Furthermore, all offsets must be in bounds; offsets must additionally be page-aligned for `mtd_write`. We tacitly omit an additional precondition $n < \#\mathit{pebs}$ for all operations.

The operation `mtd_write` models the fact that pages are written sequentially by a loop. The function $\mathtt{copy}(\mathit{src}, \mathit{off}_0, \mathit{dst}, \mathit{off}_1, n)$ returns the result of copying the value from index $\mathit{off}_0 + i$ in $\mathit{src}$ to index $\mathit{off}_1 + i$ in $\mathit{dst}$, for all $i$ with $0 \le i < n$. We also specify the possibility of *hardware failures*: either the body of the loop executes normally, or writing of the current page fails nondeterministically and

```
mtd_write($n, \mathit{off}, \mathit{len}, \mathit{buf}; \mathit{err}$)
 if  $\mathit{pebs}[n]$.fill $\leq \mathit{off} \wedge \mathit{off} + \mathit{len} \leq$ PEB_SIZE $\wedge \neg\, \mathit{pebs}[n]$.bad
    $\wedge$ page-aligned($\mathit{off}$) $\wedge$ page-aligned($\mathit{len}$) then
   $\mathit{err}$ := ESUCCESS, $m$ := 0
   while $\mathit{err}$ = ESUCCESS $\wedge\, m \neq \mathit{len}$ do
          { $\mathit{pebs}[n]$.data := copy($\mathit{buf}, m, \mathit{pebs}[n]$.data, $\mathit{off} + m$, PAGE_SIZE)
            $\mathit{pebs}[n]$.fill := $\mathit{off} + m$ + PAGE_SIZE
            $m$ := $m$ + PAGE_SIZE }
       or $\mathit{err}$ := EIO

mtd_read($n, \mathit{off}, \mathit{len}; \mathit{buf}, \mathit{err}$)
 if $\mathit{off} + \mathit{len} \leq$ PEB_SIZE $\wedge \neg\, \mathit{pebs}[n]$.bad then
   $\mathit{buf}$ := copy($\mathit{pebs}[n]$.data, $\mathit{off}, \mathit{buf}, 0, \mathit{len}$)

mtd_erase($n; \mathit{err}$)
 if $\neg\, \mathit{pebs}[n]$.bad then
   $\mathit{pebs}[n]$ := peb(EMPTY_PEB, 0, false)

mtd_isbad($n; \mathit{bad}$)
  $\mathit{bad}$ := $\mathit{pebs}[n]$.bad

mtd_markbad($n; \mathit{err}$)
  if $\neg\, \mathit{pebs}[n]$.bad then
    $\mathit{pebs}[n]$.bad := true
```

Fig. 2: MTD Operations

a corresponding error code EIO is returned. Similarly, all other operations may also fail nondeterministically. We omit the respective code in each operation for brevity.

This model makes the following assumptions about the hardware:

1. Page writes and block erasure can be viewed as atomic operations.
2. Success of an operation can be recognized, i.e., an error is not returned by mistake.
3. Conversely, hardware failure can also be detected reliably. In particular, reads that produce garbage can be recognized.
4. An unsuccessful page write/block erasure does not modify the state.
5. An unexpected power failure has no effect on the state of the flash device.

Assumption 4 is not realistic and we will relax it to a certain degree. For example, checksums can be used to recognize certain kinds of data corruption. However, on the level of MTD there is no possibility to express such application-specific concepts.

The model maintains the following invariant for all $\mathit{peb} = \mathit{pebs}[i]$ with $\neg \mathit{peb}.\mathtt{bad}$:

$$\textbf{invariant}\quad \mathtt{page\text{-}aligned}(\mathit{peb}.\mathtt{fill}) \wedge \mathit{peb}.\mathtt{fill} \leq \mathtt{PEB\_SIZE} \\ \wedge\ \forall n.\ \mathit{peb}.\mathtt{fill} \leq n < \mathtt{PEB\_SIZE} \rightarrow \mathit{peb}.\mathtt{data}[n] = \mathtt{EMPTY} \tag{2}$$

It specifies that the fill count is a multiple of PAGE_SIZE and that all bytes above (inclusive) are empty. The invariance of this trivially follows from the preconditions of the operations.

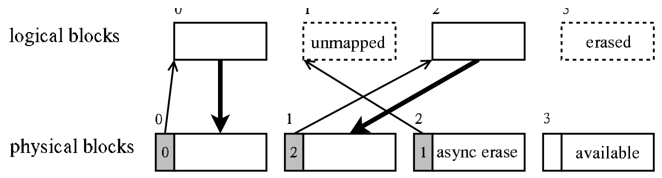

Fig. 3: Mapping of Logical Blocks to Physical Ones

## 3 Abstract EBM Layer Specification

The erase block management layer essentially provides the same functionality as the driver/MTD—namely read, write and erase—though it is based on *logical erase blocks* (LEBs). These are mapped on-demand to physical ones. This indirection enables a number of desirable features, namely asynchronous erase, hiding of bad blocks from the application, wear-leveling and trivial support for several *volumes* (i.e., partitions) on one device. However, the way this mapping is stored on flash leads to subtle differences between the behavior of the EBM and MTD in the presence of power failures. These effects can not be hidden completely by the implementation and are consequently present in the formal EBM specification as well. We therefore informally describe first how the implementation works, and then define an abstract EBM model.

Figure 3 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, block 0 is mapped to 0, whereas the data of logical block 2 is stored in physical block 1. This forward mapping is kept in RAM.

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 by reading the headers of each physical block, and it is lost during power-failure.

A logical block that has no associated physical one (such as the dashed blocks 1 and 3) is implicitly empty, i.e., it has previously been erased. As soon as a write to such a block occurs, a new physical block is allocated and the mapping is extended both in memory and on flash.

The mapping to a physical block is in general deallocated by requesting an *asynchronous* erase, also called *unmapping* the LEB. The logical block may be reused immediately after unmapping, however, the old physical block still contains the inverse mapping, as it is the case for LEB 1 in the example. When the system recovers from power failure in such a situation, the mapping for logical block 1 will *re-appear* with some old data. Since it would be rather difficult to prevent this effect without sacrificing the lazy allocation of physical erase blocks, the application/file system is expected to deal with it; or alternatively use a less efficient *synchronous* version of logical block erasure. Note that several PEBs with the same inverse mapping may exist simultaneously. These are distinguished by sequence numbers in PEB headers (see Sec. 4).

```
ebm_write($v, l, off, len, buf$)
 if  $avols[v][l]$.ismapped $\wedge$ $avols[v][l]$.fill $\leq off \wedge off + len <$ LEB_SIZE
    $\wedge$ page-aligned($off$) $\wedge$ page-aligned($len$) then
 choose $n$ with $n \leq len \wedge$ page-aligned($n$) in
  $avols[v][l]$.data := copy($buf, 0, avols[v][l]$.data$, off, n$)
  if $n \neq 0$ then $avols[v][l]$.fill := $off + n$
  if $n = len$ then $err$ := ESUCCESS else $err$ := EIO

ebm_read($v, l, off, len; buf$)
 if $off + len \leq$ LEB_SIZE then
  if $avols[v][l]$.ismapped then $buf$ := copy($avols[v][l]$.data$, off, buf, 0, len$)
  else                           $buf$ := fill-buffer($buf, len$, EMPTY)

ebm_erase($v, l$)
    { $avols[v][l]$ := erased,    $err$ := ESUCCESS }
 or { $avols[v][l]$ := unmapped, $err$ := EIO }

ebm_map($v, l$)
 if $\neg$ $avols[v][l]$.ismapped then
   $avols[v][l]$ := mapped(EMPTY_LEB, 0)

ebm_unmap($v, l$)
 $avols[v][l]$ := unmapped

ebm_create_volume($n; v$)
 choose $v_0$ with $\neg v_0 \in avols$ in $v$ := $v_0$
   $avols[v]$ := mkarray$\langle Leb \rangle(n)$
   forall $l < n$ do
     $avols[v][l]$ := erased
```

Fig. 4: EBM Operations

We will now formally specify the EBM layer in a way so that it only maintains logical blocks but encompasses the effect described above. The state of the model consists of a partial function *avols* mapping volume identifiers $\mathbb{V}$ to arrays of logical blocks: A mapped LEB stores an array `data` of bytes together with the counter `fill` similarly to MTD (1). However, a LEB has a smaller size than a PEB due to the inverse mapping stored at the beginning of each physical block by the implementation. Mapped blocks *leb* are recognized by the test *leb*.`ismapped`. Otherwise, a logical block has been erased asynchronously (`unmapped`) or synchronously (`erased`). Note that the EBM implementation handles bad blocks transparently, i.e., there is no need to model them in the abstract interface and state.

$$\begin{aligned} &\textbf{state var } avols : \mathbb{V} \nrightarrow Array\langle Leb\rangle \qquad \text{where} \\ &\textbf{data } Leb = \texttt{mapped}(\texttt{data} : Array_{\texttt{LEB\_SIZE}}\langle Byte\rangle, \texttt{fill} : \mathbb{N}) \\ &\qquad\qquad \mid \texttt{unmapped} \mid \texttt{erased} \end{aligned}$$

Figure 4 shows the operations on this layer. The preconditions—denoted by `if`-statements at the beginning of operations—are similar to the ones of MTD, namely the respective offsets must be in bounds and a multiple of `PAGE_SIZE`. Blocks are addressed by a volume identifier $v$ and the logical block number $l$. We

```
ebm_reset_recover(; $err$)
  choose $avols', err'$ with $(err' = \texttt{ESUCCESS} \rightarrow \texttt{inv}(avols') \wedge avols \subseteq avols')$
    $avols := avols'$
    $err \ \ := err'$
```

Fig. 5: Effect of a Power-failure on the state of the EBM

tacitly assume that $v$ denotes a valid volume $v \in avols$, and that $l < \#avols[v]$. Additionally, the operation `ebm_write` requires the block $l$ to be mapped.[1]

Writing to a block may fail nondeterministically. In contrast to Fig. 2 it is not realized by a loop but simply by writing a (non-strict) prefix of length $n$ of the actual data. The operation succeeds if the whole data is written. The field `fill` is updated only if $n \neq 0$.

A physical erase block for an LEB is allocated via the operation `ebm_map`. Operations `ebm_erase` and `ebm_unmap` request synchronous resp. asynchronous deallocation. Similar to our hardware model, nondeterministic failures may occur (partly omitted in Fig. 4), and we assume that failure as well as success can be detected reliably. In the case of such errors the state is not modified by any operation, with the exception of erase, which may set the respective logical erase block to `unmapped`. This means that erase may update the in-memory mapping although it failed to invalidate the remains of the inverse mapping stored on flash.

Unsurprisingly, an invariant `inv` analogous to formula (2) is maintained by all operations. We call a state of the EBM *consistent* if it satisfies this invariant.

Possible effects of a *power failure* and the subsequent recovery are specified by an extra operation `ebm_reset_recover` shown in Fig. 5. After a power failure, the EBM implementation reads the mapping stored in each physical erase block and tries to restore its state. This may fail due to read errors. For an unmapped logical erase block there may still be a physical erase block storing the inverse mapping, as for example PEB 2 in Fig. 3. Thus, the logical erase block 1 will be re-mapped with the contents found in PEB 2. In the model, this leads to a state $avols'$ that is “greater” than $avols$ before the crash, formally specified by the relation $\subseteq$, which holds iff

1. $avols$ and $avols'$ contain the same volume identifiers and corresponding volumes have the same size
2. if $avols[v][l] \neq \texttt{unmapped}$ then $avols'[v][l] = avols[v][l]$.

Thus, both states are identical with the exception of previously unmapped logical erase blocks, which may be arbitrary after a reset.

## 4 EBM Implementation

This section describes the implementation of the functionality of Sec. 3 on top of the MTD hardware model of Sec. 2. The implementation has several subcomponents as visualized by Fig. 6. Grey boxes denote functional components.

[1] The full model actually checks for this condition and maps the block on-demand. This is omitted for brevity here.

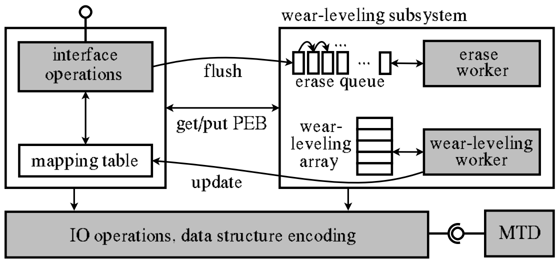

Fig. 6: Subsystems of the Implementation

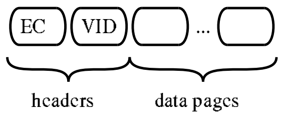

Fig. 7: Layout of a PEB

For example, the whole layer is represented by “interface operations” that provides the EBM interface to applications, as denoted by the knob at the top. It maintains the in-memory data structure that stores the forward mapping from logical to physical blocks, labeled “mapping table”.

Allocation “get” and asynchronous erase “put” of physical blocks are managed by the wear-leveling subsystem; it maintains the erase queue and some information about the state of physical blocks in the “wear-leveling array”. Asynchronous erasure and wear-leveling are background operations.

The I/O layer provides operations not only to read and write parts of the flash memory, but also to convert on-disk data structures such as block headers and the volume table to and from a byte-representation.

This section is structured as follows: First the data structures needed for an implementation of the interface operations are discussed. Afterwards, the asynchronous erasure and wear-leveling subsystem are discussed. Finally, we outline how the in-memory state is recovered from flash.

## 4.1 Data Structures & Interface Operations

The forward mapping *vols* (bold arrows in Fig. 3) is stored in RAM. It maps ($\nrightarrow$ indicates a finite map) each volume identifier $v \in vols$ to an array, which is indexed by logical block numbers. The value stored is either a physical block number if one has been allocated, or the constant `unmapped` otherwise.

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

Fig. 7 shows the layout of a PEB. The first two pages are used to store two headers. The remaining pages store application data. The first page contains an erase counter associated with the physical erase block (*erase counter*- or EC-header). The erase counter is used for wear-leveling.

The second page of allocated PEBs contains the inverse mapping (thin arrows in Fig. 3) as the *volume identifier header* (VID-header). It stores the corresponding volume identifier and logical block number. Sequence numbers `sqn` distinguish multiple PEBs with equal `vol`, `leb` pairs: During system startup/recovery,

```
write($v, l, off, len, buf$)
  if $vols[v][l] \neq$ unmapped then
    io_write_data($vols[v][l], off, len, buf$)

read($v, l, off, len; buf$)
  if $vols[v][l] =$ unmapped then
    $buf :=$ fill-buffer($buf, len$, EMPTY)
  else
    io_read_data($vols[v][l], off, len; buf$)

erase($v, l$)
  unmap($v, l$)
  wl_flush($v, l$)

map($v, l$)
  if $vols[v][l] =$ unmapped then
    wl_get_peb($; m$)
    io_write_vidhdr($m$, vidhdr($v, l, max\text{-}sqn, 0, 0$))
    $max\text{-}sqn := max\text{-}sqn + 1$
    if $err =$ ESUCCESS then $vols[v][l] := m$

unmap($v, l$)
  if $vols[v][l] \neq$ unmapped then
    $vols[v][l] :=$ unmapped
    wl_put_peb($v, l, vols[v][l]$)

create_volume($n; v$)
  choose $v_0$
  with $v_0 \notin vols \land v_0 \neq$ VTBL_VOLID in
    $v := v_0$
    $vols[v] :=$ mkarray$\langle PebRef \rangle(n)$
    forall $l < n$ do
      $vols[v][l] :=$ unmapped
    io_write_vtbl($vols$)
```

Fig. 8: Implementation of the Operations (slightly simplified)

the highest sequence number denotes the newest block for a given inverse mapping. An (optional) size and checksum of the contents of the block are used for atomic block-writes during wear-leveling. Two headers are necessary, because every PEB must store its erase counter, but only once a PEB is allocated an inverse mapping is required. Formally, the headers are defined as:

$$\textbf{data}\ \mathit{EcHeader} \ = \mathtt{echdr}(\mathtt{ec} : \mathbb{N})$$
$$\textbf{data}\ \mathit{VidHeader} = \mathtt{vidhdr}(\mathtt{vol} : \mathbb{V}, \mathtt{leb} : \mathbb{N}, \mathtt{sqn} : \mathbb{N}, \mathtt{size} : \mathbb{N}, \mathtt{checksum} : \mathbb{N})$$

We specify I/O operations (prefixed by `io_`) for reading and writing EC/VID-headers and data pages. Their purpose is twofold: On the one hand encoding from and to byte-representations is performed. On the other hand the operations do the necessary offset computations. For example `io_write_data`($n, off, len, buf$) simply calls `mtd_write`($n, 2 \cdot \mathtt{PAGE\_SIZE} + off, len, buf$). Furthermore, they add additional hardware failures on top of the hardware model of Sec. 2. Programming a VID-header for example may also fail by writing garbage, i.e., data that does not contain a valid VID-header, into the second page.

The main operations are shown in Fig. 8 in a slightly simplified version. In the actual implementation a hardware failure triggers several retries of an operation before giving up and returning an error.

Reading and writing of a logical block $(v, l)$ evaluates the mapping $vols[v][l]$ to obtain the physical block number and calls the respective I/O-operation. The operation `map` requests a new physical block $m$ from the wear-leveling subsystem by calling `wl_get_peb` and writes the VID-header using a new sequence number.

```
wl_put_peb($lebref, n$)
  $wla[n]$.status := erasing
  $eraseq$ := enqueue(eq-entry($n$, $lebref$), $eraseq$)

wl_get_peb(; $n$)
  let $ecs = \{wla[n].\mathtt{ec} \mid wla[n].\mathtt{status} = \mathtt{free} \land n < \#wla\}$ in
  if $ecs \neq \emptyset$ then
    choose $m$ with $wla[m].\mathtt{status} = \mathtt{free} \land \varphi(wla[m].\mathtt{ec}, ecs)$ in
      $n := m$
      $wla[n]$.status := used

atomic_change($v, l, m, len, buf, err$)
  $len$ := datasize($buf$)
  io_write_vidhdr($m$, vidhdr($v, l, max\text{-}sqn, len$, checksum($buf, len$)); $err$)
  $max\text{-}sqn := max\text{-}sqn + 1$
  if $err$ = ESUCCESS $\land\ len > 0$ then
    io_write_data($m$, 0, align$^\uparrow$($len$, PAGE_SIZE), $buf$)
```

Fig. 9: The Wear-Leveling Subsystem

If the write was successful, the mapping is updated. Conversely, unmap removes a logical block $(v, l)$ from the mapping and releases the corresponding physical block with wl_put_peb which puts the PEB into the erase queue. Similarly, erase first removes the in-memory mapping. Additionally, all PEBs that still store an inverse mapping for the LEB are erased synchronously via wl_flush.

A new volume is created by selecting an unused volume identifier, setting the state of each logical block to unmapped and writing the new volume table to flash. The volume table encodes a partial function from user-accessible, existing volumes to their size. Apart from user-accessible volumes, there are also hidden volumes. We currently only use the hidden volume VTBL_VOLID to store the volume table itself.

## 4.2 Asynchronous Erasure & Wear-Leveling

Whether a physical erase block is free, allocated, scheduled for erasure or is already unusable is stored alongside its erase counter in the wear-leveling array. It is used to find suitable free PEBs for the interface operations and appropriate free and used PEBs for wear-leveling.

**state var** $wla : Array\langle WlEntry \rangle$ where

**data** $WlEntry$ = wl-entry(ec : $\mathbb{N}$, status : $WlStatus$)

**data** $WlStatus$ = free | used | erasing | erroneous

Every free and used PEB has a valid EC-header and its erase counter stored on flash and in memory match. The page for the VID-header and the data pages of a free physical erase block are not yet programmed. Erroneous PEBs are already marked as bad on flash.

The PEBs scheduled for erasure are additionally kept in a queue. It is used to assign work to the background operation for asynchronous erasure. 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.

**state var** $\mathit{eraseq} : \mathit{Seq}\langle \mathit{EraseqEntry} \rangle$ where
**data** $\mathit{EraseqEntry} = \texttt{eq-entry}(\texttt{pnum} : \mathbb{N}, \texttt{lebref} : \mathit{LebRef})$
**data** $\mathit{LebRef} \ \ = \texttt{none} + \mathbb{V} \times \mathbb{N}$

Fig. 9 shows the implementation of allocation and deallocation of a physical erase block. Allocation choses a free PEB with certain restrictions $\varphi$ on its erase counter—e.g. medium wear among the free PEBs—and marks it as used. Deallocation of a PEB $n$ that was mapped at LEB *lebref* beforehand (or known to have an invalid VID-header if *lebref* is `none`) adds a corresponding entry to the erase queue.

The background operation for asynchronous erasure (not shown) dequeues an entry from the erase queue and then tries to erase the PEB synchronously by calling `mtd_erase` and to write a new EC-header with an increased erase counter multiple times. If this fails, the PEB is marked as bad via `mtd_markbad`. The operation `wl_flush` (not shown) iterates over the erase queue and similarly erases all PEBs that still belong to a specific LEB synchronously.

Wear-leveling is implemented as choosing a used and a free physical erase block of low resp. high wear. If the difference of the erase counters exceeds a certain threshold the VID-header and data region of the used PEB are read. The operation `atomic_change` as shown in Fig. 9 is the core of the wear-leveling algorithm. Conceptually, it must write a new inverse mapping for the logical erase block $(v, l)$ and the buffer's contents into the free physical erase block $m$. However, there are two problems that need to be addressed. First, programming *all* pages of the data region of the new PEB could preclude successive write operations from the client that were allowed on the previous PEB. Therefore, only the contents up to the last non-`EMPTY` byte are written, calculated as $\texttt{datasize}(\mathit{buf})$. From the MTD invariant (2) it follows that successive writes by a client remain allowed. Second, additional measures are needed to ensure correct recovery from an unexpected power-loss during wear-leveling. Fig. 10 shows the different intermediate states of the target physical erase block during wear-leveling. At the top the contents of a free PEB are shown. The bold arrows denote state transitions due to a call of an I/O operation. An unsuccessful write to the VID-header is easily detectable during recovery, either the VID-header is empty or contains garbage. After a successful write of the VID-header, the recovery would read the PEB and discover that it stores the newest inverse mapping for the logical erase block $(v, l)$. However, this is clearly wrong, since the actual data has not yet been copied to this PEB and successive read operations would just return bytes with the value `EMPTY`. Therefore, the data size of the contents of the original PEB is also stored in the VID-header. Rebuilding the mapping after a reset then only takes a PEB into consideration if the data size calculated over its data pages is

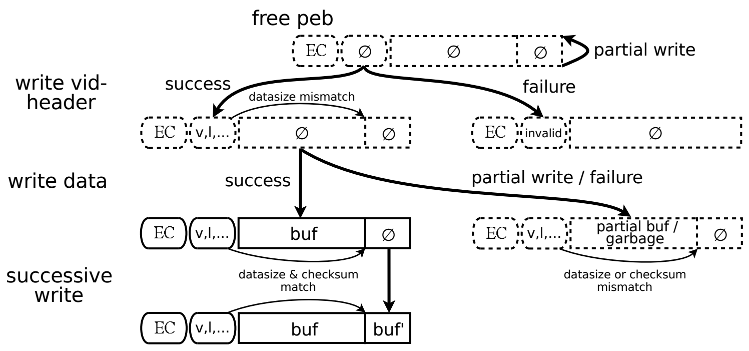

Fig. 10: States of the new PEB during and after `atomic_change`

at least as large as the value in the VID-header requires. This measure is also sufficient to detect a partial write of the data. We store a checksum in the VID-header and additionally allow failures during programming of the data pages that can be detected by either the data size or the checksum. If the copying was successful, the in-memory mapping is updated accordingly. Otherwise, the new physical erase block is scheduled for erasure and the old PEB is used.

Note that the checksum is only calculated up to the initial data size. Thus, a successive write to the LEB after wear-leveling maintains that the data size and checksum stored in the VID-header match the values calculated from the contents of the data region. In summary, these additional fields allow to distinguish *valid* (solid) from *invalid* (dotted) states of the target PEB.

The second problem is not specific to this implementation. Every model that either 1) updates the mapping before copying the actual data or 2) allows failures that write a valid mapping but invalid data simultaneously has to deal with this issue. In our model the inverse mapping must be updated first because it is stored in the second page and we enforce that pages are written sequentially.

If asynchronous erasure and wear-leveling are scheduled in between operations, do not fail and there are enough free PEBs to move to, the difference between erase counters of good PEBs is bounded by a constant. Thus, the device is worn out evenly.

With the operation `atomic_change` it is possible to implement an additional interface operation that atomically exchanges the contents of a logical erase block. On the abstract layer of Sec. 3 this is then specified as shown in Fig. 11. If the operation fails the LEB is unchanged. In contrast to `ebm_write`, `ebm_change` is more general and has a more favorable behavior wrt. failures. However, on the concrete layer this comes at the price of one additional erasure of a block. Thus, it is only desirable to use `ebm_change` if the additional guarantees are actually required. In UBIFS this functionality is for example used to write a new super block.

```
ebm_change($v, l, n, buf$)
  $avols[v][l] := $mapped(copy($buf$, 0, EMPTY_LEB, 0, $n$), $n$)
```

Fig. 11: Atomically Exchange the Contents of an LEB

```
recover(; $err$)
  let $recs = \emptyset$ in
    scan_all(; $recs, err$)
    if (VTBL_VOLID, VTBL_LNUM) $\in recs$ then let $vtbl$ in
      io_read_vtbl($recs$[VTBL_VOLID, VTBL_LNUM].pnum; $vtbl, err$)
      if $err =$ ESUCCESS then
        init_volume_sizes($vtbl$; )
        init_volume_mappings($recs$; )
```

Fig. 12: Rebuilding of the in-memory State from Flash

## 4.3 Power Failure & Recovery

The state of the EBM implementation is in RAM and only the MTD state is persistent. An unexpected power failure may invalidate the in-memory state, but is assumed to preserve everything stored on flash unaltered.

Fig. 12 shows how the in-memory state is rebuilt from the data structures stored on flash. We assume that after a power failure this operation is first executed, before any client can issue a call. First, all physical erase blocks are scanned (`scan_all`), i.e., it is checked whether a PEB is marked as bad and has valid EC- and VID-headers. The PEB's entry in the wear-leveling array, the erase queue and maximum of the sequence numbers are updated accordingly. Instead of updating the in-memory mapping *vols* directly an intermediate data structure

$$recs : \mathbb{V} \times \mathbb{N} \nrightarrow \mathit{RecoveryEntry} \qquad \text{where}$$
$$\mathit{RecoveryEntry} = \texttt{recovery-entry}(\texttt{pnum} : \mathbb{N}, \texttt{sqn} : \mathbb{N})$$

is introduced. In contrast to *vols*, the data structure contains *all* encountered combinations $(v, l)$ of volume identifiers and logical block numbers and the corresponding physical erase block. This includes hidden volumes and logical erase blocks beyond the—at this point unknown—size of the corresponding volume. The sequence number of the corresponding PEB is also cached. It is used to determine during the scanning which one of two PEBs belonging to the same LEB stores the most recent inverse mapping in case both are *valid*.

Afterwards, it is checked that a volume layout was found during scanning. Mounting fails if no layout is present. Otherwise, the volume table is read and for each non-hidden volume identifier a volume of the stored size initialized to `unmapped` is added to *vols* (`init_volume_sizes`). Finally, all mapping information from the intermediate data structure *recs* referring to an existing volume and within its bounds is transferred to *vols* (`init_volume_mappings`).

The recovery does not alter the MTD state. A power-loss during the operation therefore does not need any additional concepts.

It is crucial for the correctness of the recovery that the in-memory mapping corresponds to the *most recent* (inverse) mapping stored on-disk after each operation, among those PEBs that are *valid*.

To see that it is necessary to have the most recent mapping in RAM, assume the opposite: There are two PEBs $A$ and $B$ and both store a mapping for a LEB $(v, l)$. In memory $(v, l)$ is mapped to $A$, although $B$ has the higher sequence number. If the contents of both data regions are identical, assume that a write operation is requested by the client on LEB $(v, l)$ with non-empty data. Afterwards, $A$ and $B$'s contents definitely differ. In the event of a power failure, the subsequent recovery will restore a mapping from $(v, l)$ to $B$. Reading the mapped LEB $(v, l)$ before and after the power-loss will yield different results.

During wear-leveling there are intermediate states that do not yet have the correct data, but a newer version of the mapping—the dotted states in Fig. 10. Therefore, it is not sufficient to only consider the sequence number. The data size and checksum of the PEB also need to be taken into account, i.e., the mapped PEB must be valid.

## 5 Related Work

The models [3,2,4] in Z notation of an ONFI-compliant [11] device are conceptually below our model of a driver for flash memory. It would be possible to provide an implementation of our MTD model on top of their hardware model.

The block manager in the Alloy models [19,20] maps logical to physical pages and has a similar task as our EBM. However, storing and updating an on-disk mapping is not treated. Power failures are only considered during writing of a sequence of pages. Their specification of power failures and recovery is intertwined and uses auxiliary variables for the status of a pages. It is not immediately clear to us, how one would disentangle the specification in a real implementation.

Flash Translation Layers (FTLs) [5] and some FFSs [6,13] similarly store information about the state of a page or block in out-of-band (OOB) data, which allows programming of individual bits. This simplifies the recovery from power failures during wear-leveling, since it is possible to set a validity bit after copying the data. However, NOR flash devices do not have OOB data and some NAND devices use the whole area for error-correction codes [30]. Therefore, our EBM implementation is more generic. FTLs that support an operation similar to `unmap` (see "trim" command in Section 7.10 in [17], [21] clarifies the semantics) also have the problem that pages re-emerge after a power failure.

In the refinement-based approach [7] with Event-B, it is assumed that bookkeeping information is stored in every page, i.e., a page knows the version of the file it belongs to and the offset within the file. Updating the contents of one page is atomic. If two pages store the same inverse mapping after a power failure during wear-leveling, its contents are identical and chosing either suffices.

However, this approach uses more memory for the mapping and requires reading every page of the flash device during startup in order to rebuild the mapping.

None of the formal models [3,19,7] considers the limitation to sequential writes within an erase block, although non-sequential writes are often not supported by newer ONFI-compliant devices [8,11].

## 6 Conclusion

We have presented a formal specification of an erase block management layer and an implementation based on an ONFI-compliant hardware model. Performance aspects such as asynchronous erasure and quality aspects such as wear-leveling are hidden from clients of the abstract model. Only power failure is visible, but its abstract specification is much more tractable for the verification of clients. As a consequence we can focus on the log-structure, indexing and write buffering of a FFS in the future.

The refinement proof between the abstract EBM model and the implementation is already completed and establishes that the implementation's behavior is captured by the abstract EBM specification. We also show that the recovery works as specified if a power failure occurs in between or *during* operations using the temporal logic of KIV [27]. Due to space limitations, we could not provide a description of these proofs in this paper. Quite some time was spent on understanding which concepts are relevant and what assumptions regarding failures are necessary to ensure that power loss during operations is handled correctly.

We are currently working on an automatic translation from our models to Scala [23] code, allowing us to run and test our implementation on top of a Memory Technology Device in Linux.

Several aspects remain for future work. In the implementation of UBI wear-leveling and erasure are performed in a background thread and concurrent write operations are permitted. The implementation uses locks on a per-LEB level to ensure that the background operations do not interfere with the interface operations. We did not yet verify this kind of concurrency. There is also an unresolved issue with *unstable bits* [31], resulting from a power cut during an erase operation. They are not covered by our hardware model.

## References

1. E. Börger and R. F. Stärk. *Abstract State Machines — A Method for High-Level System Design and Analysis.* Springer, 2003.
2. A. Butterfield, L. Freitas, and J. Woodcock. Mechanising a formal model of flash memory. *Sci. Comput. Program.*, 74(4):219–237, February 2009.
3. A. Butterfield and J. Woodcock. Formalising Flash Memory: First Steps. *IEEE Int. Conf. on Engineering of Complex Computer Systems*, 0:251–260, 2007.
4. A. Butterfield and A. Cathin. Concurrent models of flash memory device behaviour. In *Formal Methods: Foundations and Applications*, volume 5902 of *Lecture Notes in Computer Science*, pages 70–83. Springer Berlin Heidelberg, 2009.

5. Tae-Sun Chung, Dong-Joo Park, Sangwon Park, Dong-Ho Lee, Sang-Won Lee, and Ha-Joo Song. A survey of flash translation layer. *J. Syst. Archit.*, 55(5-6):332–343, May 2009.
6. Intel Corp. Intel Flash File System Core Reference Guide, version 1. Technical report, Intel Corporation, 2004.
7. K. Damchoom and M. Butler. Applying Event and Machine Decomposition to a Flash-Based Filestore in Event-B. In *Formal Methods: Foundations and Applications*, pages 134–152. Springer, 2009.
8. Samsung Electronics. Page program addressing for MLC NAND application note, 2009. `http://www.samsung.com`.
9. 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)*, pages 33–45, 2012.
10. 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.
11. Intel Corporation et al. *Open NAND Flash Interface Specification*, June 2013. URL: `www.onfi.org`.
12. 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.
13. E. Gal and S. Toledo. Algorithms and Data Structures for flash memory. *ACM computing surveys*, pages 138–163, 2005.
14. T. Gleixner, F. Haverkamp, and A. Bityutskiy. UBI - Unsorted Block Images. `http://www.linux-mtd.infradead.org/doc/ubidesign/ubidesign.pdf`, 2006.
15. C.A.R. Hoare. The verifying compiler: A grand challenge for computing research. *Journal of the ACM*, 50(1):63–69, 2003.
16. A. Hunter. A brief introduction to the design of UBIFS. `http://www.linux-mtd.infradead.org/doc/ubifs_whitepaper.pdf`, 2008.
17. INCITS. ATA/ATAPI Command Set - 2 (ACS-2), Revision 2, August 3, 2009.
18. R. Joshi and G.J. Holzmann. A mini challenge: build a verifiable filesystem. *Formal Aspects of Computing*, 19(2), June 2007.
19. E. Kang and D. Jackson. Formal Modelling and Analysis of a Flash Filesystem in Alloy. In *Proc. of ABZ*, pages 294–308. Springer, 2008.
20. 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.
21. F. Knight. TRIM - DRAT/RZAT clarifications for ATA8-ACS2, Revision 2, February 23, 2010.
22. Memory Technology Device (MTD) and Unsorted Block Images (UBI) Subsystem of Linux. `http://www.linux-mtd.infradead.org/index.html`.
23. M. Odersky, L. Spoon, and B. Venners. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Incorporation, USA, 1st edition, 2008.
24. 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`.
25. G. Reeves and T. Neilson. The Mars Rover Spirit FLASH anomaly. In *Aerospace Conference*, pages 4186–4199. IEEE Computer Society, 2005.
26. W. Reif, G. Schellhorn, K. Stenzel, and M. Balser. Structured specifications and interactive proofs with KIV. In W. Bibel and P. Schmitt, editors, *Automated Deduction—A Basis for Applications*, volume II, pages 13–39. Kluwer, Dordrecht, 1998.

27. G. Schellhorn, B. Tofan, G. Ernst, and W. Reif. Interleaved programs and rely-guarantee reasoning with ITL. In *Proc. of the 18th Int. Symposium on Temporal Representation and Reasoning (TIME)*, IEEE Computer Society Press, pages 99–106, 2011.
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 Area. `http://www.linux-mtd.infradead.org/faq/ubi.html`.
31. UBIFS - Unstable Bits Issue. `http://www.linux-mtd.infradead.org/doc/ubifs.html`.