Adding Concurrency to a Sequential Refinement Tower

Gerhard Schellhorn(✉), Stefan Bodenmüller, Jörg Pfähler, and Wolfgang Reif

Institute for Software and Systems Engineering,
University of Augsburg, Augsburg, Germany
{schellhorn,stefan.bodenmueller,reif}@informatik.uni-augsburg.de,
joerg.pfaehler@gmx.de

Abstract. This paper defines a concept and a verification methodology for adding concurrency to a sequential refinement tower of abstract state machines, that is based on data refinement and a component structure. We have developed such a refinement tower for the Flashix file system earlier, from which we generate executable (C and Scala) Code.

The question we answer in this paper, is how to add concurrency based on locks to such a refinement tower, without breaking the initial modular structure. We achieve this by just enhancing the relevant components, and adding intermediate atomicity refinements that complement the data refinements that are already there. We also give a verification methodology for such atomicity refinements.

1 Introduction

Development of formally proved software systems using incremental refinement has been successfully used in many case studies. Often the system developed is a sequential system, e.g. a compiler. The standard technique used then is data refinement [8,9,14] or closely related definitions [2].

Our group has developed a verified file system for flash memory [12,13,22,26] using a strategy based on data types specified as abstract state machines (ASMs, [4]), data refinement, and subcomponents. The resulting refinement tower is shown in Fig. 1. It starts with an abstract state machine that specifies the POSIX file system operations. This interface is then refined to an implementation VFS (denoted by VFS $\sqsubseteq$ POSIX), which calls operations of a submachine AFS. This machine acts as an abstract interface to the next implementation. This continues until the MTD layer is reached, which is the generic interface for flash hardware used in Linux.

Scala code for simulations as well as C code integrated into the Linux kernel has been generated from the implementations (shown in grey). The file system so far is strictly sequential, i.e., all operations are called in sequential order. Adding

Supported by the Deutsche Forschungsgemeinschaft (DFG), "Verifikation von Flash-Dateisystemen" (grants RE828/13-1 and RE828/13-2).

concurrency is however relevant for practical usability and efficiency on at least three levels: top-level operations, garbage collection and wear leveling.

Since existing refinement strategies are typically designed to start with an atomic specification that is refined to a concurrent system, this raises the question how to add concurrency a posteriori to intermediate levels of such a refinement tower without losing modularity and without having to start verification from scratch. This paper gives a positive answer to the question, by "shifting" parts of the refinement towers, i.e., by modifying individual specifications and implementations, to make them concurrent.

We will use erase block management (the EBM interface) and the concurrent implementation of wear leveling (WL) based on the interface Blocks as an example to demonstrate how concurrency is added. A specification of the sequential specifications and refinements involved has already been published in [23].

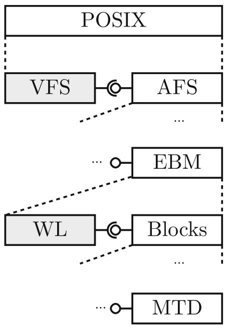

Fig. 1. Flashix refinement tower

The next section will give a simplified version of the relevant sequential specifications and implementation, to demonstrate in Sect. 3 how concurrency using locks is added and how restrictions are encoded as *ownership* constraints. Section 4 informally introduces the well-known concept of *linearizability* as the relevant concept to verify correctness of concurrent implementations, and shows how the proof of linearizability can be split into one of data refinement (that reuses the original proof) and one of *atomicity refinement*. Section 5 will give a proof strategy based on *rely-guarantee* proofs and *reduction*. Both have been implemented in our KIV [11] theorem prover. The specifications and proofs for the case study are available online [18]. Section 6 gives related work, and Sect. 7 concludes.

2 The Refinement for Wear Leveling

Flash hardware is partitioned into erase *blocks*. Blocks can be written sequentially, and erased as a whole. Erasing wears out the block until it becomes unusable. Therefore, for efficient usage of a flash device, blocks must be worn out evenly. In particular if a device is filled to a large part with static data, the blocks with these data must sometimes be swapped with other (currently empty) blocks, that have often been modified and erased. This is called *wear leveling*. Wear leveling is hidden from the more abstract levels of the file system by the erase block manager (EBM) interface. The interface offers access to *logical blocks*. The task of the implementation (WL) is to map them to the *physical blocks* offered by the hardware, and to change the mapping when this is advisable, using an internal operation for wear leveling that has no effect (implements skip) for the interface EBM.

An abstract specification of the erase block manager is given with the ASM `EBM`. The state consists of a function that maps logical block numbers to actual content and a set of currently used ("mapped") block numbers.

$$\textbf{state} \quad \mathit{Contents} : \mathit{nat} \to \mathit{content} \qquad \mathit{Mapped} : \mathit{set}\langle \mathit{nat}\rangle$$
$$\textbf{initial state} \quad \mathit{Contents} = \lambda\, n.\, \texttt{empty} \wedge \mathit{Mapped} = \emptyset$$

For simplicity, we do not specify *content*, except for a default value `empty`. The interface of `EBM` shown in Fig. 2 allows to read and to write the content of logical blocks. The operations use a semicolon to separate input and output parameters.

```
ebm_write($lnum$, $c$)
  $Contents(lnum) := c$
  $Mapped := Mapped \cup \{lnum\}$

ebm_read($lnum$; $c$)
  if $\neg\ lnum \in Mapped$ then
    $c :=$ empty
  else
    $c := Contents(pnum)$
```

Fig. 2. Sequential specification of the erase block manager (`EBM`)

The implementation of `EBM` is given by the ASM `WL` together with a specification `Blocks` as a submachine. This refinement introduces the distinction between logical and physical blocks. `Blocks` allows reading and writing of physical blocks while `WL` is responsible for the mapping of logical to physical blocks. Furthermore, the wear leveling algorithm is implemented in `WL`.

To enable wear leveling each physical block in `Blocks` contains a header. This header stores which logical block is mapped to the physical block or if the block is currently unmapped ($\perp$).

$$\textbf{data}\ \mathit{header} = \texttt{mapped}(\texttt{blockno} : \mathit{nat}) \mid \perp$$
$$\textbf{data}\ \mathit{block} = \texttt{mkb}(\texttt{header} : \mathit{header}, \texttt{content} : \mathit{content})$$

The state of `Blocks` is a function that maps physical block numbers to blocks. Initially all blocks are unmapped and empty.

$$\textbf{state}\ \mathit{Blocks} : \mathit{nat} \to \mathit{block} \qquad \textbf{initial state}\ \mathit{Blocks} = \lambda\, m.\, \texttt{mkb}(\perp, \texttt{empty})$$

The interface of `Blocks` as shown in Fig. 3 provides additional functionality to write and read the header of a physical block. Accessing the content of a block requires it to be mapped, i.e., the header of the block must not be $\perp$. For wear leveling the interface also offers an interface operation **blocks_get_wl** that returns two physical blocks *from* and *to*, that are suitable for wear leveling. The actual decision is based on erase counts (also stored in block headers), but we leave the concrete implementation open here. To signal that wear leveling is currently unnecessary, the operation returns a block *from* with an unmapped header.

The operations of `WL` are depicted in Fig. 4. To avoid scanning the headers of all blocks, the state of `WL` maintains an in-memory mapping from logical block numbers to headers, which contain the corresponding physical block numbers if the logical block is mapped.

$$\textbf{state}\ \mathit{LMap} : \mathit{nat} \to \mathit{header} \qquad \textbf{initial state}\ \mathit{LMap} = \lambda\, n.\, \perp$$

```
blocks_write($pnum$, $c$)
pre $Blocks(pnum)$.header $\neq \bot$
  $Blocks(pnum)$.content $:= c$

blocks_read($pnum$; $c$)
pre $Blocks(pnum)$.header $\neq \bot$
  $c := Blocks(pnum)$.content

blocks_write_h($pnum$, $h$)
  $h := Blocks(pnum)$.header

blocks_read_h($pnum$; $h$)
  $Blocks(pnum)$.header $:= h$

blocks_map(; $pnum$)
  choose $m$ with
    $Blocks(m)$.header $= \bot$
  in
    $pnum := m$

blocks_get_wl(; $from$, $to$)
  choose $m_1$, $m_2$ with
    $Blocks(m_2)$.header $= \bot$
    /* $\wedge$ $m_1$, $m_2$ are suitable
          for wear leveling */
  in
    $from := m_1$, $to := m_2$
```

Fig. 3. Sequential specification of the physical block layer (Blocks)

Reading and writing of content delegates to the corresponding operations of Blocks by following *LMap*. If a logical block is unmapped, the write operation first maps this block to an unused physical block by writing a header and updating *LMap*. Therefore Blocks provides an operation **blocks_map** that returns a fresh block that can be mapped.

The wear leveling operation **wl_wear_leveling**, that is not visible to the clients, first requests a pair of blocks to be wear leveled by calling **blocks_get_wl**. If the *from* Block is mapped, its header and content are copied to the *to* Block and *LMap* is updated. We leave away many details here, that ensure, that crashing in the middle of wear leveling will result in a consistent state, see [23].

To prove the refinement WL $\sqsubseteq$ EBM three invariants are established in WL.

$$\begin{aligned}
&\mathit{injective}(\mathit{lmap}) \leftrightarrow\\
&\quad \forall\, n_1, n_2.\ \mathit{lmap}(n_1) \neq \bot \wedge \mathit{lmap}(n_2) \neq \bot \rightarrow \mathit{lmap}(n_1) \neq \mathit{lmap}(n_2)\\
&\mathit{lmapblocks}(\mathit{lmap}, \mathit{blocks}) \leftrightarrow\\
&\quad \forall\, n.\ \mathit{lmap}(n) \neq \bot \rightarrow \mathit{blocks}(\mathit{lmap}(n).\mathtt{blockno}).\mathtt{header} = \mathtt{mapped}(n)\\
&\mathit{blockslmap}(\mathit{blocks}, \mathit{lmap}) \leftrightarrow\\
&\quad \forall\, m.\ \mathit{blocks}(m).\mathtt{header} \neq \bot \rightarrow \mathit{lmap}(\mathit{blocks}(m).\mathtt{header}.\mathtt{blockno}) = \mathtt{mapped}(m)
\end{aligned}$$

The three predicates guarantee a valid mapping between logical and physical blocks. *injective* prohibits that two logical blocks are mapped to the same physical block, *lmapblocks* ensures that each mapped physical block in *lmap* points to the correct logical block, and *blockslmap* ensures that each mapped physical block also has a matching entry in *lmap*.

The abstraction relation between states of the specification and states of the implementation ensures that mapped blocks in *Mapped* conform with mapped logical blocks in *LMap* and that contents of *Contents* conform to the contents of the mapped physical blocks in *Blocks*.

$$\begin{aligned}
&(\forall\, n.\ n \in \mathit{Mapped} \leftrightarrow \mathit{LMap}(n) \neq \bot)\\
\wedge\ &(\forall\, n.\ n \in \mathit{Mapped} \rightarrow \mathit{Contents}(n) = \mathit{Blocks}(\mathit{LMap}(n).\mathtt{blockno}).\mathtt{content})
\end{aligned}$$

```
wl_write($lnum$, $c$)
  let $pnum = 0$ in
    if $LMap(lnum) = \bot$ then
      blocks_map(; $pnum$);
      blocks_write_h($pnum$, mapped($lnum$));
      blocks_write($pnum$, empty);
      $LMap(lnum)$ := mapped($pnum$);
    else
      $pnum$ := $LMap(lnum)$.blockno;
    blocks_write($pnum$, $c$);

wl_read($lnum$; $c$)
  if $LMap(lnum) = \bot$ then
    $c$ := empty;
  else
    let $pnum = LMap(lnum)$.blockno in
      blocks_read($pnum$; $c$);

wl_wear_leveling()
internal
  let $h = \bot$, $c$ = empty,
      $from = 0$, $to = 0$
  in
    blocks_get_wl(; $from$, $to$);
    blocks_read_h($from$; $h$);
    if $h \neq \bot$ then
      let $lnum = h$ .blockno in
        blocks_read($from$; $c$);
        blocks_write_h($to$, $h$);
        blocks_write($to$, $c$);
        $LMap(lnum)$ := mapped($to$);
        blocks_write_h($from$; $\bot$) ;
```

Fig. 4. Sequential implementation of the wear leveling layer (WL)

Together with the invariants this is sufficient to prove a data refinement using forward simulation.

3 Adding Concurrency and Ownership

The sequential code calls the wear leveling operation at the end of every other operation. This causes small pauses in between operations. A better solution is to call wear leveling in a separate thread concurrently. This exploits that even the MTD hardware interface is capable of reading and writing different blocks concurrently. This is not possible for individual blocks, since these do not provide random access, but can be written sequentially only.

Adding concurrency implies that interface operations are now called concurrently by several threads, and it is natural to assume that they now have an *atomic* semantics (which is the natural semantics of ASMs, but was not required in a sequential context). We emphasize this, by writing $\texttt{EBM}_{\text{At}}$ and $\texttt{Blocks}_{\text{At}}$ for EBM and Blocks with atomic semantics, although the machines are the same. Assuming an atomic semantics for the implementation is however unrealistic.

A simple solution that enforces an atomic semantics for an implementation is to use a single global mutex, that is set before each operation and released afterwards. Doing so for the operations of WL would however prevent wear leveling from running concurrent.

An implementation of Blocks that uses such a simple locking strategy would be correct to enforce atomicity, but too restrictive as it would prevent concurrent access to different blocks. It would also not be sufficient for the correctness of WL. To understand this, consider the implementation of **wl_write** in Fig. 4 and a potential interleaving of two concurrent executions of this operation as depicted in Fig. 5. Here two threads tid_1 and tid_2 write two contents to different logical blocks $lnum_1$ resp. $lnum_2$. Both logical blocks are unmapped so by calling **blocks_map** unmapped physical blocks are chosen to be mapped. Although the operation is atomic it is possible that for *tid₂* the same physical block *pnum* is returned as for *tid₁* since *tid₁* has not written the new header yet. Both threads would then write to the same physical block, first different headers that point to *lnum₁* resp. *lnum₂*, then different contents c_2 resp. c_1. After both writes finish an inconsistent state is reached to the effect that the written data of *tid₂* is lost and the injectivity of the block mapping is violated.

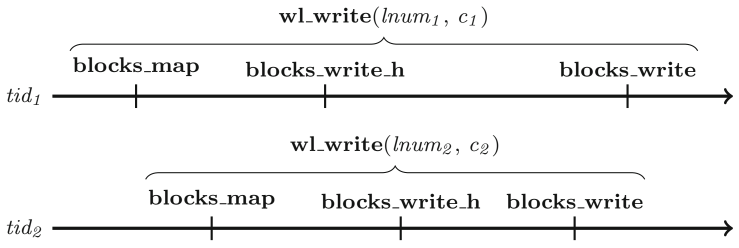

Fig. 5. Critical interleaving of two **wl_write** executions

A concept is needed that enforces on the level of *Blocks* that its implementation can assume that only one thread is writing each block at one time, and that headers are written by a single thread only.

The concept we use is that of threads *owning* data structures.

$$\textbf{data}\ \mathit{owner} = \texttt{readers}(\texttt{tids} : \mathit{set}\langle \mathit{threadid}\rangle)\ \mid\ \texttt{writer}(\texttt{tid} : \mathit{threadid})$$
$$\textbf{ghoststate}\ \mathit{OBlocks} : \mathit{nat} \rightarrow \mathit{owner} \qquad \mathit{OHeaders} : \mathit{owner}$$

An owner can either own a data structure non-exclusively (typically for reading) or exclusively for writing. That a thread owns all headers or some block for reading or writing is specified as two ghost variables *OHeaders* and *OBlocks*. To ensure, that clients of the extended interface $\texttt{Blocks}_{\text{Owns}}$ shown in Fig. 6 respect the ownership, we add preconditions to the operations, that request read-ownership for reading and write-ownership for writing blocks and headers. A thread that wants to call an operation of $\texttt{Blocks}_{\text{Owns}}$ must now *acquire* ownership before it and can *release* ownership afterwards. For this purpose the interface is extended with two auxiliary acquire and release operations. These acquire and release full ownership, which is sufficient for the concurrent implementation of wear leveling given below. It is possible to add operations that acquire and release read-ownership too. Acquiring full ownership has the precondition that there is no current owner. If two threads now try to write the same block, one of them will violate the precondition of acquire (if it tries to acquire) or it will violate the precondition of writing (if it does not). But this is impossible, since submachine calls in implementations are checked to satisfy their preconditions.

data *mutex* = `free` `locked`(`tid` : *threadid*)

```
blocks_acquire(pnum)
pre OBlocks(pnum) = readers(∅)
atomic ghost
  OBlocks(pnum) := writer(tid)

blocks_acquire_h()
pre OHeaders = readers(∅)
atomic ghost
  OHeaders := writer(tid)

blocks_write(pnum, c)
pre    Blocks(pnum).header ≠ ⊥
     ∧ tid ∈ OBlocks(pnum).writers
atomic
  Blocks(pnum).content := c

blocks_write_h(pnum, h)
pre    tid ∈ OHeaders.writers
     ∧ tid ∈ OBlocks(pnum).writers
atomic
  h := Blocks(pnum).header

blocks_get_wl(; from, to)
pre tid ∈ OHeaders.readers
atomic
  choose m1, m2 with
    Blocks(m2).header = ⊥
    /* ∧ m1 is good for WL */
  in
    from := m1, to := m2
```

```
blocks_release(pnum)
pre tid ∈ OBlocks(pnum).readers
atomic ghost
  OBlocks(pnum) := release(tid, OBlocks(pnum))

blocks_release_h()
pre tid ∈ OHeaders.readers
atomic ghost
  OHeaders := release(tid, OHeaders)

blocks_read(pnum; c)
pre    Blocks(pnum).header ≠ ⊥
     ∧ tid ∈ OBlocks(pnum).readers
atomic
  c := Blocks(pnum).content

blocks_read_h(pnum; h)
pre tid ∈ OHeaders.readers
atomic
  Blocks(pnum).header := h

blocks_map(; pnum)
pre tid ∈ OHeaders.readers
atomic
  choose m with
    Blocks(m).header = ⊥
  in
    pnum := m
```

Fig. 6. Atomic specification of the physical block layer with ownership ($\texttt{Blocks}_{\mathrm{Owns}}$)

Calls to acquire and release in the augmented code of wear leveling will now ensure, that ownership is properly acquired. They are used for verification, but are "ghost code" that is eliminated when generating executable code.

To make sure, that calls to acquire never violate their precondition, we have to use locks in the extended implementation of `WL` given in Fig. 8. The simple implementation we give here just uses mutexes.

data *mutex* = `free` | `locked`(`tid` : *threadid*)

The locking and unlocking operations **mutex_lock** and **mutex_unlock** are specified as the atomic program statements given in Fig. 7. The definition of **mutex_lock** uses the program construct **atomic** φ { α }. The **atomic** construct blocks the current thread until its guard φ is satisfied. Immediately afterwards, the program α is executed in a single, indivisible step.

Figure 8 shows the result of applying sufficient locking and ownership acquisition to `WL`. Additionally, each atomic step gets an individual label (`W1`–`W18`, `R1`–`R8`, and `WL1`–`WL21`) to give assertions for this program point when reasoning about atomicity (see Sect. 5). We refer to this concurrent implementation as $\mathtt{WL}_{\mathrm{Conc}}$. The state of $\mathtt{WL}_{\mathrm{Conc}}$ is enhanced by a lock that protects the headers of all blocks, and locks for each logical block that protects its contents.

```
mutex_lock(mutex)
  atomic (mutex = free) {
    mutex := locked(tid)
  }

mutex_unlock(mutex)
pre mutex = locked(tid)
  mutex := free
```

Fig. 7. Mutex locking operations

$$\textbf{state} \quad \ldots \quad Lock : mutex \qquad Locks : nat \to mutex$$

We use mutexes for all locks, since they match our simplification of acquiring write-ownership only. The actual Erase-Block-Manager in Flashix employs reader-writer locks whenever parallel reading is unproblematic. The general locking concept of $\mathtt{WL}_{\mathrm{Conc}}$ is to acquire *Lock* only if the mapping from logical to physical blocks needs to be updated. This is the case when writing to an unmapped block or when wear leveling is active. Otherwise, locking only one individual *Locks*(*lnum*) of a specific logical block *lnum* is sufficient. This lock protects the corresponding entry *LMap*(*lnum*) of the block mapping as well as the content of the physical block *LMap*(*lnum*).`blockno`. With this strategy multiple reads and writes to different, mapped logical blocks are possible, even in parallel to wear leveling.

```
      wl_write(lnum, c)
W1      let pnum = 0 in
W2        mutex_lock(Lock);
W3        mutex_lock(Locks(lnum));
W4        if LMap(lnum) = ⊥ then
W5          blocks_acquire_h();
W6          blocks_map(; pnum);
W7          blocks_acquire(pnum);
W8          blocks_write_h(pnum, mapped(lnum));
W9          blocks_write(pnum, empty);
W10         blocks_release(pnum);
W11         LMap(lnum) := mapped(pnum);
W12         blocks_release_h();
          else
W13         pnum := LMap(lnum).blockno;
W14       mutex_unlock(Lock);
W15       blocks_acquire(pnum);
W16       blocks_write(pnum, c);
W17       blocks_release(pnum);
W18       mutex_unlock(Locks(lnum));

      wl_read(lnum; c)
R1      mutex_lock(Locks(lnum));
R2      if LMap(lnum) = ⊥ then
R3        c := empty
        else
R4        let pnum = LMap(lnum).blockno in
R5          blocks_acquire(pnum);
R6          blocks_read(pnum; c);
R7          blocks_release(pnum);
R8      mutex_unlock(Locks(lnum));
```

```
      wl_wear_leveling()
      internal
WL1     let h = ⊥, c = empty,
            from = 0, to = 0
        in
WL2       mutex_lock(Lock);
WL3       blocks_acquire_h();
WL4       blocks_get_wl(; from, to);
WL5       blocks_read_h(from; h);
WL6       if h ≠ ⊥ then
WL7         let lnum = h .blockno in
WL8           mutex_lock(Locks(lnum));
WL9           blocks_acquire(from);
WL10          blocks_read(from; c);
WL11          blocks_acquire(to);
WL12          blocks_write_h(to, h);
WL13          blocks_write(to, c);
WL14          LMap(lnum) := mapped(to);
WL15          blocks_write_h(from; ⊥);
WL16          blocks_release(to);
WL17          blocks_release(from);
WL18          mutex_unlock(Locks(lnum));
WL19      blocks_release_h();
WL20      mutex_unlock(Lock);
```

Fig. 8. Concurrent implementation of the wear leveling layer ($\mathtt{WL}_{\mathrm{Conc}}$)

One exception is that the *Lock* has to be acquired in every **wl_write** execution (`W2`–`W14` in Fig. 8), at least for a short amount of time. This is due to the locking hierarchy that is employed to avoid deadlocks. When running in parallel, it is possible that a **wl_write** and **wl_wear_leveling** may both need to acquire *Lock* and the same *Locks*(*lnum*), so it must be ensured that those operations request the locks in the same order. Because **wl_wear_leveling** needs to be owner of *OHeaders* to get suitable physical blocks at `WL4` before a logical block can be locked, **wl_write** must request *Lock* (`W2`) ahead of requesting *Locks*(*lnum*) (`W3`).

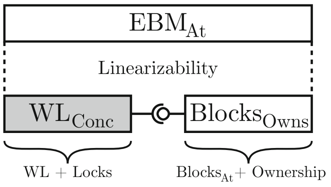

Fig. 9. Concurrency refinement of the erase-block-manager

Figure 9 shows the resulting refinement of $\mathtt{EBM}_{\mathrm{At}}$. Proving $\mathtt{WL}_{\mathrm{Conc}} \sqsubseteq \mathtt{EBM}_{\mathrm{At}}$ using linearizability is discussed in detail in the next sections. It remains to integrate the new "shifted" refinement into the refinement tower. The layers above $\mathtt{EBM}_{\mathrm{At}}$ can remain untouched since $\mathtt{EBM}_{\mathrm{At}}$ is identical to `EBM`, and sequential use of $\mathtt{EBM}_{\mathrm{At}}$ is not problematic. Below $\mathtt{Blocks}_{\mathrm{Owns}}$ an adjustment is necessary: a simple one is to use a global lock around the operations of its implementation. Since the level is already close to the MTD hardware interface, the real solution propagates ownership down to ownerships at the hardware level (where blocks store a sequence of bytes instead of a header and content).

4 Linearizabilty and Atomicity Refinement

The standard correctness criterion we use to prove correctness of the refinement of $\mathtt{EBM}_{\mathrm{At}}$ to $\mathtt{WL}_{\mathrm{Conc}}$ from Fig. 9 is *linearizability.* A formal definition can be found in [15], we only give an informal description here.

A concurrent implementation `CASM` with nonatomic programs COP_i is linearizable to an atomic specification `AASM` with atomic operations AOP_i, if the input/output behaviors of each concurrent run can be explained by mapping them to the sequential input/output behavior of some sequential run of `AASM`.

The mapping between a concurrent and a sequential run is as follows: for each concurrent call of an operation COP_i that is started at time t_i and returns at time t'_i find some point in time l_i with $t_i \leq l_i \leq t'_i$, such that all l_i are different. The point is called the *linearization point* of the operation call. Then construct some sequential run of AASM that executes each corresponding abstract operation AOP_i atomically at time l_i. Note that even for fixed linearization points this may give several sequential runs if the abstract operations are nondeterministic.

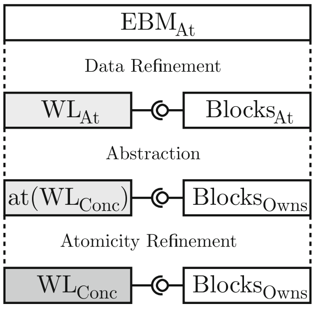

Fig. 10. Splitting the refinement

A refinement from AASM to CASM then is *linearizable*, if for every concurrent run linearization points and an abstract sequential run can be found, such that all operation calls *have the same inputs and outputs*.

The clients of the interface then cannot distinguish the concurrent run from one, where each operation call is delayed until time l_i, executes AOP_i atomically and then is delayed again until time t'_i.

Our proof technique will use an intermediate machine $\texttt{at}(\texttt{WL}_{\text{Conc}})$ that is the same as $\texttt{WL}_{\text{Conc}}$, but executes the code of each operation as one atomic step. This splits the refinement problem into three parts as shown in Fig. 10. The data refinement $\texttt{WL}_{\text{At}} \sqsubseteq \texttt{EBM}_{\text{At}}$, that we have already proved (since the ASMs are the same as WL and EBM). Second, a trivial refinement $\texttt{at}(\texttt{WL}_{\text{Conc}}) \sqsubseteq \texttt{WL}_{\text{At}}$ that *abstracts* from the locking/unlocking (and acquire/release) instructions in $\texttt{at}(\texttt{WL}_{\text{Conc}})$, since the overall effect of locking/unlocking in one atomic step is empty. Finally, the *atomicity refinement* $\texttt{WL}_{\text{Conc}} \sqsubseteq \texttt{at}(\texttt{WL}_{\text{Conc}})$, where both machines have the same data and operations, but different atomicity. Splitting the refinement from an atomic AASM to a concurrent CASM by using an intermediate at(CASM), which executes the operations of CASM atomically, has the advantage that data refinement is completely decoupled from atomicity refinement.

The next section will describe a proof strategy for proving the atomicity refinement between $\texttt{at}(\texttt{WL}_{\text{Conc}})$ and $\texttt{WL}_{\text{Conc}}$, which is the new problem we get from adding concurrency to the refinement tower.

5 Proof Strategy for Atomicity Refinement

The proof strategy we use to prove atomicity refinement consists of two steps. First we prove that the concurrent runs of $\texttt{WL}_{\text{Conc}}$ satisfy some assertions at all program points. These proofs use thread-local reasoning with the rely-guarantee calculus. They additionally ensure termination and deadlock-freedom, which are not implied by linearizability alone. Second we prove that based on the assertions, atomic program steps can be *reduced* to larger and larger atomic steps, until we arrive at $\texttt{at}(\texttt{WL}_{\text{Conc}})$. We sketch the basic strategy in the first subsection, and give results for the case study in Sect. 5.2.

5.1 Rely-Guarantee Proofs and Reduction

The variant of the rely-guarantee calculus used here is similar to the one given in [30], Section 5. The basic correctness statement[1] is of the form

$$pre \wedge I \rightarrow \langle R, G, I, run, \alpha \rangle \; post$$

[1] The notation in [30] is: $\alpha \; sat \; (pre, R \wedge (I \rightarrow I(\underline{x'})), run, G \wedge (I \rightarrow I(\underline{x'})), post)$.

where program α is assumed to be the sequential program of some thread, that executes atomic steps. These alternate with environment steps, where one environment step is an arbitrary sequence of steps of other threads.

The program is assumed to use the state variables $\underline{x}$. Precondition *pre*, postcondition *post*, predicate *run*, and global invariant I are predicates over this state. The rely R and the guarantee G restrict environment and program steps. They are predicates over $\underline{x}$ and $\underline{x'}$ We write arguments in predicates if they differ from the standard ones only.

The formula asserts, that program α, when started in a state that satisfies precondition *pre* and global invariant I, will execute steps that satisfy G and preserve the invariant I, as long as all previous environment steps satisfy R and preserve I too. No program step will block, when at that time *run* holds. In addition, when all environment steps satisfy R and preserve I, then the program will either terminate and the final state will satisfy *post*, or it will stop in a blocked state where *run* is false.

The calculus to prove such formulas in KIV is based on symbolic execution. The basic rule to execute one atomic step at label L, that is annotated with an assertion φ_L is

$$\frac{\begin{array}{l} \mathit{pre} \wedge I \rightarrow \varphi_L \wedge (\mathit{run} \rightarrow \varphi) \\ \mathit{pre} \wedge I \wedge \langle\alpha\rangle\ \underline{x} = \underline{x}' \rightarrow G(\underline{x}, \underline{x}') \wedge I(\underline{x}') \\ \mathit{pre}(\underline{x}_0) \wedge \langle\alpha(\underline{x}_0)\rangle\ \underline{x}_0 = \underline{x}_1 \wedge R(\underline{x}_1, \underline{x}) \wedge I(\underline{x}) \rightarrow \langle R, G, I, \mathit{run}, \beta\rangle\ \mathit{post} \end{array}}{\mathit{pre} \wedge I \rightarrow \langle R, G, I, \mathit{run}, L : /*\, \varphi_L *\!/\ \mathbf{atomic}\ \varphi\ \{\alpha\}; \beta\rangle\ \mathit{post}}$$

The rule reduces the conclusion at the bottom to premises. The first premise states that before executing α the assertion at the initial label holds, and that the first step does not block (φ holds) whenever the *run* predicate is true.

The second premise uses the Dynamic Logic formula $\langle\alpha\rangle\ \underline{x} = \underline{x}'$ which asserts that the sequential program α has a terminating run that yields a state $\underline{x'}$. The premise ensures that the first atomic step of the program, which executes α is a step that satisfies G and preserves the invariant I.

The third premise continues symbolic execution with the rest of the program. Its precondition uses two sets $\underline{x}_0$ and $\underline{x}_1$ of fresh variables, to represent the two old states before and after the first atomic program step. The subsequent environment step from $\underline{x}_1$ to the current state $\underline{x}$ is assumed to satisfy R. Since rely steps preserve the invariant, it can be assumed for the current state again.

One common instance of the rule is a parallel assignment $\underline{y} := \underline{t}$, which can be viewed as an abbreviation for **atomic true** $\{\underline{y} := \underline{t}\}$. In this case the formula $\langle\alpha\rangle\ \underline{x} = \underline{x}'$ reduces to $\underline{y'} = \underline{t} \wedge \underline{z'} = \underline{z}$, where $\underline{z}$ are the remaining variables from $\underline{x}$ that are not assigned.

The rules for other constructs like conditionals resemble the usual rules for symbolic execution of programs, except that similar to the rule above they have rely steps in between program steps and side conditions for assertions and guarantee. For loops, a loop invariant (that holds at the start of each iteration) and a variant, that decreases with a wellfounded order are needed. Proofs for recursive routines need wellfounded induction.

Individual rely-guarantee proofs for single threads can be combined to a rely-guarantee property of a concurrent system. The crucial property that needs to hold for this to work, is that the relies and guarantees must be *compatible*: the guarantee of each thread G_{tid} must imply the relies $R_{tid'}$ of other threads $tid' \neq tid$. For our state machines where all threads are known to execute the same operations, the guarantee can be chosen to be $G_{tid} := \bigwedge_{tid' \neq tid} R_{tid'}$, the weakest guarantee possible that is trivially compatible. The system is deadlock-free, if the disjunction of all $\bigvee_{tid} run_{tid}$ holds. When a mutex is used, run_{tid} is chosen to be $lock = locked(tid) \vee lock = Free$ which implies this condition. This easily generalizes to the hierarchy of locks used in the case study.

In summary, to verify assertions for a specification of a concurrent state machine with operations OP_i, the user has to provide an invariant I, a rely R_{tid} and a predicate $idle_{tid}$. The latter describes states, where a thread is not currently executing an operation. From these predicate logic proof obligations (e.g. the R must be reflexive, initial states satisfy the invariant etc.) are generated, together with the following rely guarantee proof obligation for each operation.

$$tid \neq tid', I, idle_{tid}, pre_{tid} \vdash \langle R_{tid}, R_{tid'}, I, run_{tid}, OP_i \rangle \; idle_{tid}$$

Successful verification guarantees that each of the assertions φ_L holds every time a thread reaches label L, that the operations terminate and that the implementation is deadlock-free.

The verified assertions are then used to combine atomic statements to larger ones following Lipton's [19] strategy of reduction. The idea is that a thread executing two atomic steps At_{L1} and At_{L2} (at labels L1 and L2) with an environment step in between is often equivalent to first executing the environment step, then At_{L1} and At_{L2} with no intermediate environment step. In this case the two steps can be merged together to form one atomic step.

$$\begin{tikzcd}
 & s'_1 \arrow[r, "At_N"] & s'_2 \arrow[dr, "At_{L1}"] \arrow[drr, bend left, "\textbf{atomic}\ \{At_{L1}; At_{L2}\}"] & & \\
s_0 \arrow[ur, bend left, "At_M"] \arrow[r, "At_{L1}"] & s_1 \arrow[r, "At_M"] & s_2 \arrow[r, "At_N"] & s_3 \arrow[r, "At_{L2}"] & s_4
\end{tikzcd}$$

Fig. 11. At_{L1} commutes to the right of environment step $At_M; At_N$

Reverting the order of first executing At_{L1} and then an environment step is possible, if all steps of other threads, that could be a part of the environment step, commute to the right with At_{L1}, in the sense that executing them in both orders gives the same final state. In this case At_{L1} is called a *right mover*. Analogous to this, a step that commutes to left with all steps is called a *left mover*. Figure 11 shows an example, where the environment step consists of two steps At_M and At_N of other threads. The original run is shown at the bottom, the alternative run which allows executing At_{L1} and At_{L2} as one atomic step at the top. The intermediate states of the runs are different, but they reach the same final state.

The atomic steps of the programs can all be written in the form

$$At_L \equiv L : / * \varphi_L * / \ \textbf{atomic}\ \varepsilon_L\ \{\alpha_L\}$$

where L is the label, and φ_L the assertion established. The guard ε_L is **true** for all statements, except locking instructions, cf. Figure 7. Program α_L is either an assignment, or the call of a submachine operation. For a conditional or a while loop with test δ, α_L is defined to be $b := \delta$ using a fresh variable b, while binding a local variable **let** $y = t$ **in**... gives $\alpha_L \equiv \{y := t\}$. The formal condition for At_{L1} to commute to the right with At_{L2} executed by another thread is

$$\varphi_L \wedge \varphi'_M \wedge \varepsilon_L \wedge \varepsilon'_M \wedge tid \neq tid' \wedge \langle \alpha_L; \alpha'_M \rangle\ \underline{x} = \underline{x}_0 \rightarrow \langle \alpha'_M; \alpha_L \rangle\ \underline{x} = \underline{x}_0 \qquad (1)$$

In the formula, $\varphi'_M, \varepsilon'_M, \alpha'_M$ are variants, that rename thread local variables used in At_M to new, primed variables disjoint from the shared state and the local variables of At_L. The criterion critically uses the assertions at both labels, since they often show that the preconditions of the implication contradict each other, trivializing the proof. If, for example the two steps are both in a region where a common lock is needed, they commute trivially: φ_L implies $lock = locked(tid)$, while φ'_M implies $lock = locked(tid')$, so the proof obligation trivially holds. A general result is that locking is always a right mover, while unlocking is always a left mover.

Combining steps to larger steps can be translated into rules for making statements like sequential composition, conditionals and loops atomic, when their parts are atomic already. We use rules similar to the reduction rules given in [10]. Iterated application gives larger and larger atomic blocks. Ideally, the final result is that the whole concurrent program of one operation has been combined into a single atomic step. If this is possible, then a linearizability proof becomes trivial, as the linearizability point then simply *is* the single atomic step.

5.2 Proving the Case Study

The main task for proving the atomicity refinement of the case study is to find assertions, rely conditions and a global invariant that are strong enough to allow atomicity refinement.

The rely conditions are derived from the crucial ideas what data structures are protected from being changed, when thread *tid* has a certain lock or ownership. This results in the following clauses.

$$tid \in OHeaders.\texttt{readers} \rightarrow \forall\ m.\ Blocks(m).\texttt{header} = Blocks'(m).\texttt{header}$$

$$tid \in OBlocks(m).\texttt{readers} \rightarrow Blocks(m) = Blocks'(m)$$

$$Lock = \texttt{locked}(tid) \rightarrow LMap' = LMap$$

$$Locks(n) = \texttt{locked}(tid) \rightarrow LMap'(n) = LMap(n)$$

$$\begin{aligned}&\quad Locks(n) = \texttt{locked}(tid)\\ &\rightarrow \forall\ m.\ Blocks(m).\texttt{header} = \texttt{mapped}(n) \leftrightarrow Blocks'(m).\texttt{header} = \texttt{mapped}(n)\end{aligned}$$

The only rely that is somewhat difficult to find is the last one: if a thread locks logical block n, then other threads are not allowed to change the block header to point to or to point away from n.

The global invariant and the assertions are derived from several sources. First, ownership as used in the interface $\texttt{Blocks}_{\text{Owns}}$ has to be compatible with the use of locks.

$$\mathit{OHeaders} \subseteq \mathit{Lock}.\texttt{owner} \tag{2}$$

$$\begin{aligned}\forall\ m.\quad & \mathit{Blocks}(m).\texttt{header} \neq \bot \\ & \rightarrow \mathit{OBlocks}(m) \subseteq \mathit{Locks}(\mathit{Blocks}(m).\texttt{header}.\texttt{blockno}).\texttt{owner}\end{aligned} \tag{3}$$

$$\forall\ m.\ \mathit{Blocks}(m).\texttt{header} = \bot \rightarrow \mathit{OBlocks}(m) \subseteq \mathit{Lock}.\texttt{owner} \tag{4}$$

The invariant (2) states that headers are owned only if the lock has been taken. Invariant (3) states that a mapped physical block m can be owned (and therefore changed) only if the corresponding logical block that is stored in its header is locked. For unmapped blocks property (4) states that they can be owned only if $\texttt{WL}_{\text{Conc}}$ has taken the header lock.

Second, the three global invariants of the sequential code are relevant. Dropping them completely would result in illegal states where e.g. the block mapping is no longer injective. However, the invariants of the sequential verification are only guaranteed to hold in idle states, where no thread is running. So it is necessary to give weaker assertions for intermediate states, that are still sufficient to avoid illegal ones.

For the given case study, it turns out that *lmapblocks* and *injective* are preserved by all steps, but that *blockslmap* does not hold while the headers are locked. As a result the global invariant can include $\mathit{blockslmap}(\mathit{Blocks}, \mathit{LMap})$ only when the headers are currently not owned ($\mathit{Oheaders} = \mathit{readers}(\emptyset)$). To establish this assertion, after a step that releases *OHeaders*, assertions have to be given for all labels, where *OHeaders* is taken. For writing the predicate is violated between line `W9` after the header of block *pnum* has been set to *lnum* and line `W11`, where $\mathit{LMap}(\mathit{lnum})$ is set to *pnum*. For all lines in this range $\mathit{blockslmap}(\mathit{Blocks}, \mathit{LMap}(\mathit{lnum}; \mathit{pnum}))$ holds: if *LMap* were already updated, then *blockslmap* would hold. The wear leveling algorithm gives similar assertions for the range `WL13`–`WL15`.

Finally, assertions are sometimes necessary for the code after a test or after assignments to a variable. In a purely sequential setting, the test for $\mathit{LMap}(\mathit{lnum}) \neq \bot$ at `R2` ensures that this formula holds, until the subsequent **let** binding $\mathit{pnum} = \mathit{LMap}(n).\texttt{blockno}$ at line `R4`, which will ensure $\mathit{pnum} = \mathit{LMap}(\mathit{lnum}).\texttt{blockno}$ when the variable *pnum* is used later on. However, in the concurrent setting *LMap* may be assigned by other threads, destroying each of these properties. In the given case, the rely conditions are strong enough to propagate the formulas, so we assert that at line `R4` the first formula holds, while for lines `R5`–`R7` the second holds. A number of similar assertions are needed for other local variables.

Proving the rely-guarantee proof obligations for the individual programs requires the main effort in proving the concurrent setting correct. This is in line with case studies we have done for lock-free algorithms [25,27–29], where proving rely-guarantee assertions caused the main effort too.

After establishing assertions for all program points, the program can then be reduced, combining atomic steps to larger ones. This requires to find out, which steps are left or right movers (or both). The current strategy implemented in KIV does simple syntactic checks to check whether the resulting commutativity requirement (1) is trivial: either the accessed variables are disjoint, or the preconditions of the proof obligation trivially reduce to false. Otherwise it is possible to generate proof obligations, by manually asserting that certain steps (identified by their label) are left or right movers (or both).

For the case study, manual specifications of mover types are currently necessary for the atomic calls **blocks_acquire** (right mover) and **blocks_release** (left mover) of $\texttt{Blocks}_{\mathrm{At}}$. The reader may check, that this trivially implies that the other operations of $\texttt{Blocks}_{\mathrm{At}}$ are left and right movers. After the mover types have been determined, the reduction rules are then applied automatically, to form maximally large atomic blocks.

This immediately results in a single atomic block for **wl_write** and **wl_read**. Reducing **wl_wear_leveling** creates three atomic blocks. The first ends at the conditional at line `WL6` and is a right mover. The second is for the **let**-block `WL7`–`WL19`. The third is for the last two lines `WL20`–`WL21`, and is a left mover. The conditional cannot be reduced, since its then-branch requires the lock for block *lnum* to be free, while the empty else-branch does not have this guard. With the atomic blocks now being much larger than before, it becomes possible to prove much stronger invariants that just hold in between blocks, but did not hold for the original programs. In particular, since all locking and unlocking of blocks is now within atomic regions, the simple invariant that all $Locks(lnum)$ are always free can be established using another simple rely-guarantee proof. With the new invariant established, another reduction step finds, that the conditional at line `WL6` can now be reduced to an atomic block. Together with the initial and the final block being right resp. left movers already, the wear leveling code is combined by another reduction step into a single step. This implies that the concurrent implementation of wear leveling is indeed linearizable and a correct refinement.

6 Related Work

Related work on wear leveling and the flash file system we have developed has already been given in [23], where the full version of the sequential wear leveling algorithm has been specified.

This paper is based on the PhD of Jörg Pfähler [21], where concurrency was added to the full wear leveling algorithm. The full version needs to add ownership annotations and locks to several refinements. This version is now used in our actual flash file system implementation. The PhD also contains extensions that allow verifying crash-safety, which we could not address in this paper.

The flash file system by Damchoom et al. [7] has concurrent wear leveling. The synchronization between threads is implicitly performed by the semantics of Event-B models, i.e., an event in an Event-B model is always executed atomically, and not explicitly via locks or other synchronization primitives. This makes the step to actual running code more difficult and less straightforward. The full erase block management used in our flash file system is also more general, because it does not use additional bits of out-of-band data of an erase block.

Verification of concurrent, lock-based systems is of course a very broad topic with lots of important contributions, and the proof techniques we use are from this field. We are not aware of other formal methods that specifically address the question of this paper: how to add concurrency a posteriori to an existing modular, sequential system, without having to prove the system from scratch. Adding concurrency to components of an existing software system to increase efficiency is however a recurring software engineering task that should be supported by formal methods.

Refinement and abstraction of atomicity is quite common for concurrent systems, and many refinement definitions for concurrent systems like [1] or [20] address refinements of atomicity. The refinement calculus of Back [3] uses the opposite direction. It starts out with an atomic program and splits it into smaller actions in refinement steps.

The calculus of atomic actions due to Elmas et al. [10] is an extension of Lipton's [19] original approach for highly concurrent, linearizable programs. It provides a more incremental verification methodology than the calculus given here for highly concurrent systems and its implementation is better automated. The assertions and invariants are incrementally validated in [10], whereas here a rely/guarantee proof is used to validate them before applying any reductions. The rules of the calculus in [10] address partial correctness, so termination would have to be proven differently. Nevertheless, many of the reduction rules given in this paper are directly used in our approach too.

Ownership annotations are used in the C verifier VCC [6] and Spec# [16] in order to ensure data-race freedom of the code. They are typically coupled to objects of the programming language, while we decouple the use of ownership from objects. Fractional permissions [5] in concurrent versions of separation logics [24] serve a similar purpose as ownership. These are for example supported by the C code verifier VeriFast [17].

7 Conclusion

We have presented an approach for adding concurrency to an existing refinement tower. The given approach allows to add concurrency by enhancing some of the components of the refinement tower. Abstract interfaces are extended with acquire and release operations, that specify allowed concurrency. In our case study concurrent writes on different blocks are possible, while concurrent writes on the same block are disallowed. Concurrent code using these interfaces is then possible, that enhances the existing sequential code with suitable locking strategies. We have evaluated this strategy of "shifting parts of the refinement" tower by making wear-leveling concurrent in the Flashix file system. Specifications using the same concept have been defined for concurrent garbage collection, with executable code already running. Verification is work in progress. We also work on a allowing concurrent calls for POSIX file system operations.

References

1. Abadi, M., Lamport, L.: The existence of refinement mappings. Theoret. Comput. Sci. **2**, 253–284 (1991). Also appeared as SRC Research Report 29
2. Abrial, J.-R.: Modeling in Event-B - System and Software Engineering. Cambridge University Press, Cambridge (2010)
3. Back, R.J.R.: A method for refining atomicity in parallel algorithms. In: Odijk, E., Rem, M., Syre, J.-C. (eds.) PARLE 1989. LNCS, vol. 366, pp. 199–216. Springer, Heidelberg (1989). https://doi.org/10.1007/3-540-51285-3_42
4. Börger, E., Stärk, R.F.: Abstract State Machines—A Method for High-Level System Design and Analysis. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-642-18216-7
5. Boyland, J.: Checking interference with fractional permissions. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 55–72. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-44898-5_4
6. Cohen, E., et al.: VCC: a practical system for verifying concurrent C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 23–42. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03359-9_2
7. Damchoom, K., Butler, M.: Applying event and machine decomposition to a flash-based filestore in Event-B. In: Oliveira, M.V.M., Woodcock, J. (eds.) SBMF 2009. LNCS, vol. 5902, pp. 134–152. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-10452-7_10
8. de Roever, W., Engelhardt, K.: Data Refinement: Model-Oriented Proof Methods and their Comparison. Cambridge Tracts in Theoretical Computer Science, vol. 47. Cambridge University Press, Cambridge (1998)
9. Derrick, J., Boiten, E.: Refinement in Z and in Object-Z: Foundations and Advanced Applications. FACIT. Springer, Heidelberg (2001). https://doi.org/10.1007/978-1-4471-5355-9. Second, revised edition 2014
10. Elmas, T., Qadeer, S., Tasiran, S.: A calculus of atomic actions. In: Proceeding POPL 2009, pp. 2–15. ACM (2009)
11. Ernst, G., Pfähler, J., Schellhorn, G., Haneberg, D., Reif, W.: KIV - overview and verifythis competition. Softw. Tools Techn. Transf. **17**(6), 677–694 (2015)
12. Ernst, G., Pfähler, J., Schellhorn, G., Reif, W.: Inside a verified flash file system: transactions & garbage collection. In: Gurfinkel, A., Seshia, S.A. (eds.) VSTTE 2015. LNCS, vol. 9593, pp. 73–93. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-319-29613-5_5
13. Ernst, G., Pfähler, J., Schellhorn, G., Reif, W.: Modular. Crash-Safe Refinement for ASMs with Submachines. Science of Computer Programming (SCP) (2016)
14. He, J., Hoare, C.A.R., Sanders, J.W.: Data refinement refined resume. In: Robinet, B., Wilhelm, R. (eds.) ESOP 1986. LNCS, vol. 213, pp. 187–196. Springer, Heidelberg (1986). https://doi.org/10.1007/3-540-16442-1_14
15. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. (TOPLAS) **12**(3), 463–492 (1990)

16. Jacobs, B., Leino, K.R.M., Piessens, F., Schulte, W.: Safe concurrency for aggregate objects with invariants. In: Software Engineering and Formal Methods (SEFM) 2005, pp. 137–146. IEEE (2005)
17. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: a powerful, sound, predictable, fast verifier for C and Java. NASA Formal Methods **6617**, 41–55 (2011)
18. KIV proofs for wear leveling (2020). https://kiv.isse.de/projects/WearLeveling.html
19. Lipton, R.J.: Reduction: a method of proving properties of parallel programs. Commun. ACM **18**(12), 717–721 (1975)
20. Lynch, N., Vaandrager, F.: Forward and backward simulations - part i: untimed systems. Inf. Comput. **121**(2), 214–233 (1995). Also: Technical Memo MIT/LCS/TM-486.b, Laboratory for Computer Science, MIT
21. Pfähler, J.: A modular verification methodology for caching and lock-based concurrency in file systems. Ph.D. thesis, Universität Augsburg, Fakultät für Informatik (2018). https://opus.bibliothek.uni-augsburg.de/opus4/frontdoor/index/index/docId/41890
22. Pfähler, J., Ernst, G., Bodenmüller, S., Schellhorn, G., Reif, W.: Modular verification of order-preserving write-back caches. In: Polikarpova, N., Schneider, S. (eds.) IFM 2017. LNCS, vol. 10510, pp. 375–390. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66845-1_25
23. Pfähler, J., Ernst, G., Schellhorn, G., Haneberg, D., Reif, W.: Formal specification of an erase block management layer for flash memory. In: Bertacco, V., Legay, A. (eds.) HVC 2013. LNCS, vol. 8244, pp. 214–229. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-03077-7_15
24. Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. In: Proceedings of 17th Annual IEEE Symposium on Logic in Computer Science, pp. 55–74. IEEE (2002)
25. Schellhorn, G., Derrick, J., Wehrheim, H.: A sound and complete proof technique for linearizability of concurrent data structures. ACM Trans. Comput. Logic **15**(4), 31:1–31:37 (2014)
26. Schellhorn, G., Ernst, G., Pfähler, J., Haneberg, D., Reif, W.: Development of a verified flash file system. In: Ait Ameur, Y., Schewe, K.D. (eds.) ABZ 2014, vol. 8477. LNCS, pp. 9–24. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-43652-3_2
27. Schellhorn, G., Travkin, O., Wehrheim, H.: Towards a thread-local proof technique for starvation freedom. In: Ábrahám, E., Huisman, M. (eds.) IFM 2016. LNCS, vol. 9681, pp. 193–209. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33693-0_13
28. Tofan, B., Schellhorn, G., Reif, W.: Formal verification of a lock-free stack with hazard pointers. In: Cerone, A., Pihlajasaari, P. (eds.) ICTAC 2011. LNCS, vol. 6916, pp. 239–255. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23283-1_16
29. Tofan, B., Travkin, O., Schellhorn, G., Wehrheim, H.: Two approaches for proving linearizability of multiset. Sci. Comput. Program. **96**(P3), 297–314 (2014)
30. Xu, Q., de Roever, W.-P., He, J.: The rely-guarantee method for verifying shared variable concurrent programs. Formal Aspects Comput. **9**(2), 149–174 (1997)