Verification of Crashsafe Caching in a Virtual File System Switch

STEFAN BODENMÜLLER, University of Augsburg, Germany
GERHARD SCHELLHORN, University of Augsburg, Germany
WOLFGANG REIF, University of Augsburg, Germany

When developing file systems, caching is a common technique to achieve a performant implementation. Integrating write-back caches is not primarily a problem for functional correctness, but is critical for proving crash safety. Since parts of written data are stored in volatile memory, special care has to be taken when integrating write-back caches to guarantee that a power cut during a running operation leads to a consistent state. This paper shows how non-order-preserving caches can be added to a virtual file system switch (VFS) and gives a novel crash-safety criterion matching the characteristics of such caches. Broken down to individual files, a power cut can be explained by constructing an alternative run, where all writes since the last synchronization of that file have written a prefix. VFS caches have been integrated modularly into Flashix, a verified file system for flash memory, and both functional correctness and crash-safety of this extension have been verified with the interactive theorem prover KIV.

CCS Concepts: • **Software and its engineering → Formal software verification**.

Additional Key Words and Phrases: POSIX-compliant File Systems, Write-Back Caching, Crash-Safety, Refinement, Interactive Verification

1 INTRODUCTION

In the Flashix project, we have developed a verified, realistic file system for flash memory that adheres to the POSIX standard [37].

Flash memory, in contrast to magnetic disks, does not have random access for writing. Instead, memory is partitioned into blocks consisting of pages, where pages can be written in ascending order only (no overwriting). Deleting old, obsolete data is possible only by erasing entire blocks, which is slow. Erasing also wears out blocks: they become unreadable after 10^4 to 10^6 erases, depending on the hardware. Therefore erasing has to be distributed evenly over all blocks: the use of an algorithm achieving *wear leveling* becomes necessary. The specific writing characteristics require a specialized file system for use in applications like the Mars rovers, which use *raw flash* for efficiency: problems with flash memory on these [38] were one of the motivations for NASA to propose a challenge to build a verifiable file system [26], which is a pilot project of the Grand Verification Challenge [23]. An alternative to a dedicated Flash file system used in today's SSDs is an intermediate file translation layer (FTL) that simulates an ordinary file system. This approach requires additional hard- and firmware and is somewhat less efficient.

Developing a correct file system implementation requires a strictly modular discipline that allows verifying individual components separately (the final source code amounts to 18k lines of C). However, there is an important

Authors' addresses: Stefan Bodenmüller, Institute for Software and Systems Engineering, University of Augsburg, Germany, stefan.bodenmueller@informatik.uni-augsburg.de; Gerhard Schellhorn, Institute for Software and Systems Engineering, University of Augsburg, Germany, schellhorn@informatik.uni-augsburg.de; Wolfgang Reif, Institute for Software and Systems Engineering, University of Augsburg, Germany, reif@informatik.uni-augsburg.de.

https://doi.org/10.1145/3523737

cross-cutting concern: power failures (*crashes*) can interrupt a system run anywhere in the middle of running operations, deleting the state still stored in RAM. A file system, therefore, not only has to be shown to be functionally correct but also to be *crash-safe*: a reboot must lead to a consistent state that contains "no surprises", e.g. no new files that were not created before the crash should appear in this state.

In the first phase of the project, we developed a sequential implementation consisting of a *refinement tower* containing 10 *components*. Each component consists of a specification and an implementation, where the implementation code calls operations from the specification of other (sub-)components. To be realistic, we have taken the concepts realized in UBIFS as a blueprint. UBIFS is the newest existing file system for flash memory integrated into the Linux kernel. Our sequential implementation already contains a modular implementation of all important concepts of UBIFS: a journal and an index are used for efficiency, algorithms for *garbage collection* and *wear leveling* are implemented. A write-buffer is used to queue data until a full page can be written to flash memory. Our top-level specification is an abstract POSIX specification defining directory trees and file content (we denote this specification with `POSIX`). An indirection is used to accommodate hard links to files. The refinement tower ends with a specification of MTD, which is the generic Linux interface for flash hardware.

We have implemented a code generator that generates Scala- or C-Code from all the component implementations. Combining them gives the full implementation, which can be used with the FUSE-library (both Scala and C) or in the Linux kernel (just C).

Formally, the specifications and implementations of a component are data types connected by data refinement, augmented with a concept that allows modular verification of conditions for crash-safety [13].

However, a purely sequential implementation of a file system is not efficient since the user has to wait for operations to finish. Waiting is necessary for three reasons: first, internal operations such as wear leveling and garbage collection, which are executed in between operations called by the user, cause waiting. Second, the top-level of POSIX operations can be used sequentially only, so one process has to wait for the other even when disjoint directories and files are addressed. Third, the user also has to wait for reads and writes to files to be performed on Flash memory, which are two orders of magnitude slower than writing to RAM.

Therefore the second phase of the project was concerned with making the file system efficient. To do this, we have developed a theory to "shift" parts of the refinement tower, allowing concurrency in specific components. This adds intermediate layers that have to be shown to be linearizable and deadlock-free. The previous data-refinements are preserved, so their proofs can be kept. The concept and its application to wear leveling are detailed in [40], and the application to the POSIX level and to garbage collection is presented in [3].

This paper addresses the third aspect: caching of written file content that is not already stored on Flash memory (*write-back caching*). In Linux, caching file content is done by the standard top-level layer implementing the POSIX standard: the virtual file system switch (VFS). This layer is common to all file systems used by Linux. Regular file systems like ext2,3,4 [32] or ReiserFS (Reiser4 in its latest version [39]) use it, as well as file systems specific for raw flash memory, such as JFFS [41], YAFFS [31], or UBIFS [24].

Apart from caching, the VFS is responsible for the generic aspects of file systems: mapping directory paths to individual nodes in the directory tree, checking access rights, and breaking up writing file data into updates for individual data pages[1]. VFS is specific to Linux, although Windows uses a similar concept called IFS.

Since VFS has to support many file systems and does write-back caching, its Linux implementation is enormously complicated. In the Flashix project, we have developed a much simpler, verified implementation of the VFS (we denote our implementation with `VFS`) described in [14, 15] that does not cache file content. We have, however, used the original interface of VFS to individual file systems, and have specified it abstractly as `AFS` (abstract file system), so it is also possible to use our implementation of `AFS` with the original VFS.

[1] different from the physical pages of flash memory

To address crash-safety, some approaches have explained the effect of a crash as an effect on the state that directly results in a consistent state. However, in a modular system, such an explanation in terms of state changes is impossible without using lots of auxiliary data in all components, since the states of high-level components (in POSIX: a directory tree and a sequence of bytes for every file) do not have a natural distinction between data structures that are already stored persistently on flash and volatile data in RAM.

Therefore, we think crash-safety in general is best expressed in an operational style. An alternative run to the crashed one is constructed that only partially executes the operations of the original one but also leads to the state after the crash. The alternative run leaves out some operations which have not taken effect (they are "retracted"). Others (even completed ones) may have taken partial effect only. This partial effect typically concerns write operations on files: they are "re-executed" to write fewer bytes. The alternative run trivially demonstrates that the crashed state is consistent (by functional correctness of the operations) and formalizes the "no surprises" idea.

The mentioned write buffer is an instance of an *order-preserving* cache that queues data. Our crash-safety criterion for such caches developed earlier in [35] guarantees that only the operations executed last are retracted, and at most one single operation is re-executed (details will be given in Section Section 7).

Caching in VFS is somewhat different since it is not order-preserving, so a new weaker crash safety criterion is necessary for the top-level POSIX operations compared to [35]. We define *write-prefix crash consistency*, which states that individual files still satisfy a prefix property: On a crash, all writes since the last *fsync* (that persisted all cached data of this individual file) are retracted. Instead, *all* of them have written a prefix of their data after recovery from the crash.

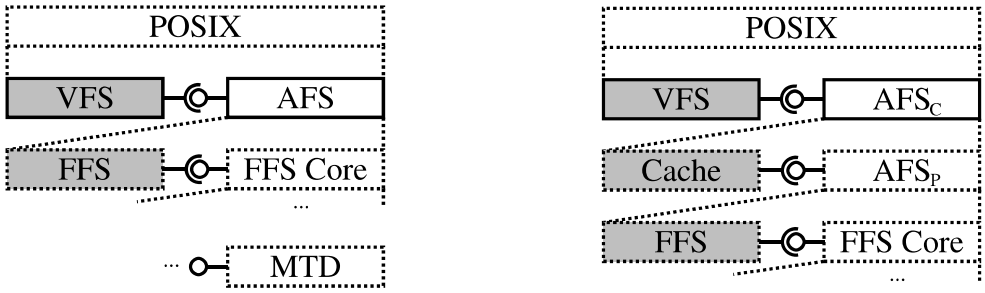
This paper also demonstrates that adding caching to VFS can be done without reimplementing VFS or breaking the implementation hierarchy represented as a formal refinement tower. In Software Engineering terms, we use the decorator pattern [19] to add VFS caches as a single new component. Functional correctness then requires verifying just the new component separately. However, crash-safety, which is the main topic of this paper, was quite hard to verify since VFS uses a data representation that is optimized for efficiency and uses the AFS interface to the individual file systems that exploit it.

Our result has two limitations. First, we assume that concurrent writes to a single file are prohibited. Without this restriction, very little can be said about the file content after a crash. Linux does not enforce this, but assumes that applications will use file locking (e.g. by using the *flock* operation) to ensure this. Second, we assume that emptying caches when executing the *fsync*-command is done with a specific strategy that empties caches bottom-up. This strategy is the default strategy implemented in VFS, but individual file systems can override this behavior, e.g. by persisting the least recently used page first. Within these limitations, however, our result enables the implementation of applications that use the file system in a crash-safe way: check-sums written before the actual data can be used to detect writes, that have not been persisted completely. Such a transaction concept would be similar to using group nodes for order-preserving caches used by the file system itself [12].

This paper is organized as follows. Section 2 gives background on the general concept of a refinement tower: components ("machines") specified as interfaces that are refined to implementations that call subcomponents, which are again specified and implemented the same way. Section 3 shows the data structures and operations of the VFS and AFS components that are relevant for manipulating file content. Section 4 then presents the addition of caching to VFS.

Section 5 defines the correctness criterion of write-prefix crash consistency and Section 6 gives some insight into its verification, which was done using our interactive verification system KIV [11]. We cannot fully go into the details of the proofs, which are very complex, the interested reader can find the complete KIV proofs online [28]. Sec. 7 discusses the compatibility of WPCC with other concepts used in Flashix that affect crash-safety: order-preserving caches and concurrency. Finally, Section 8 gives related work and Section 9 concludes.

This paper extends the conference publication [4] by providing more details on the VFS and caching operations (Sections 3 and 4). In particular, we now show the full implementation of the *write* operation of VFS (Sec. 3.2) and



(a) Without caching layer.

(b) With caching layer.

Fig. 1. Flashix refinement tower.

give a more in-depth description of the general approach for caching structural operations in a write-through manner (Sec. 4.1). A statistic on the efficiency gained by adding caches is added to Sec. 4. Our correctness criterion is now formalized in terms of histories (Sec. 5), and we give a more comprehensive description of the high-level proof strategy. Section 6 now provides information on how the formulas that prove commuting diagrams look and gives a more detailed explanation of how commuting diagrams on the granularity of AFS operations can be lifted to VFS operations. Section 7 on the integration with other caches and concurrency is new.

2 MODULAR DEVELOPMENT OF A VERIFIED FILE SYSTEM

The specification of the Flashix file system shown in Fig. 1a is organized into specifications of components. A component is an abstract data type, that consists of operations $\mathtt{Op}_i$ with inputs from In_i and outputs from Out_i, that modify a state in S.

Definition 1 (Abstract Data Type). *An abstract data type* $(S, \mathtt{Init}, (\mathtt{Op}_i)_{i\in I})$ *consists of a set* S *of states, a set* $\mathtt{Init} \subseteq S$ *of initial states, and a set of operations* $\mathtt{Op}_i \subseteq In_i \times S \times S \times Out_i$.

Each operation is specified with a contract. Components are used to either specify an interface abstractly (white boxes) or to describe an implementation (gray), from which code is generated. Both are connected by using the contract approach to data refinement (dotted lines in Fig. 1). We write $\mathtt{C} \preceq \mathtt{A}$ if component C is a refinement of component A. Note that the two components A and C must be *compatible* for a valid refinement $\mathtt{C} \preceq \mathtt{A}$, i.e. they implement the same interface (more precisely, they have the same index set of operations I). The refinement theory has been extended with proof obligations for crash safety, as detailed in [13].

To specify contracts uniformly, we prefer the operational style of abstract state machines (ASMs [5]) where the relation is computed from the semantics of operations over giving direct relational specifications as in Z [10]: we use a precondition together with an imperative program over algebraically specified data types that establishes the postcondition. The program is close to real code for implementations, but it may be as abstract as "**choose** *nextstate*, *output* **with** *postcondition*" in interfaces, using the **choose** construct of ASMs. Implementations may call operations of subcomponents (—©— in Fig. 1, we write M(A) to emphasize that A is a subcomponent of the component M), which again are abstractly specified and then implemented as a separate component. Using a program instead of a relation allows the definition of a fine-grained semantics [13] similar to control-state ASMs, where concurrent implementations result in interleaved executions, see [3]. The relation required for sequential runs considered here then abstracts runs to pairs of initial and final states.

Altogether we get the refinement tower shown in Fig. 1a. At the top-level is a specification of a POSIX-compliant interface to a file system. The component POSIX uses an algebraic tree to represent the directory structure and a

sequence of bytes (or words, the exact size is a parameter of the specification) to represent file content. POSIX is implemented by VFS using the specification component AFS as a subcomponent (i.e. VFS(AFS) ≤ POSIX), which uses a different data representation: Directory structure is represented by numbered nodes, which are linked by referencing these numbers. Refinement guarantees that the nodes always form a tree, resulting in a consistent file system.

Files are represented as a *header* and several *pages*, which are arrays of bytes of the same fixed size. Since file content is cached write-back by VFS while the directory structure only uses write-through caches, which are easy to verify, we will only briefly discuss the directory structure in the following: more information can be found in [14, 15]; the KIV specifications online [28] also have a full list.

VFS calls operations specific to each file system implementation via an interface we call AFS (abstract file system). Again this is specified abstractly, and the operations relevant for accessing file data will be defined in the next section.

Our implementation of AFS then is specific to flash memory (called FFS in the figure). Again it is implemented using subcomponents. Altogether we get a refinement tower with 11 layers. In earlier work, we have verified the various components [12, 36] to be crash-safe refinements according to the theory in [13, 40]. The bottom layer of this development is the MTD interface, that Linux uses to access raw flash memory.

To add caching in VFS, we extend the refinement tower as shown in Fig. 1b. Instead of implementing AFS directly with FFS, we use an intermediate implementation Cache of AFS (AFS_C in the figure) that caches the data and calls operations of an identical copy of AFS (called AFS_P) to persist cached data. Details on this implementation will be given in Section 4.

3 DATA REPRESENTATION IN VFS

The task of VFS is to implement POSIX operations like creating or deleting files and directories, or opening files and writing buffers to them by elementary operations on individual nodes, that represent a single directory or file. Each of these nodes is identified by a natural number $ino \in \mathsf{Ino}$, where $\mathsf{Ino} \simeq \mathbb{N}$. The operations on single nodes are implemented by each file system separately and we specify them via the AFS ("abstract file system") interface.

Our implementation can be used in two ways: One is with the FUSE library which offers a POSIX interface and uses our VFS (and our implementation of caching). The other is integrated into the Linux kernel, where we have to use the original VFS implementation with the AFS interface[2]. Therefore a precise specification of the AFS interface is given in the next subsection Section 3.1.

It should be noted that AFS specifies atomic operations (the concurrent version of VFS uses currently uses a lock around AFS operations), that are non-deterministic: they return error codes that signal low-level errors like out of memory or an unreadable page in a block which are not expressible in the abstract specification. The implementation of AFS (with or without cache) resolves this non-determinism towards returning errors only when this is stricly necessary. In contrast the VFS code is deterministic, and also resolves non-determinism present in the top-level POSIX specification (given in [14]).

There are two critical aspects in this section that are relevant when implementing a cache for file content. The first is, that file content is represented as a sequence of pages together with a size where pages containing zeros are ignored. As a result of power failures, pages above file size may exist and contain junk data (see Figure 2). The second is that operations of the interface are optimized to have as few modifications of pages as is possible. In particular, this will lead to an asymmetric implementation of the truncate operation, which changes file sizes shown in Figure 6.

[2]In fact, the interface of VFS to file systems is a strict superset of what we describe here. Since it is not well-document many experiments were needed to fix an exact specification.

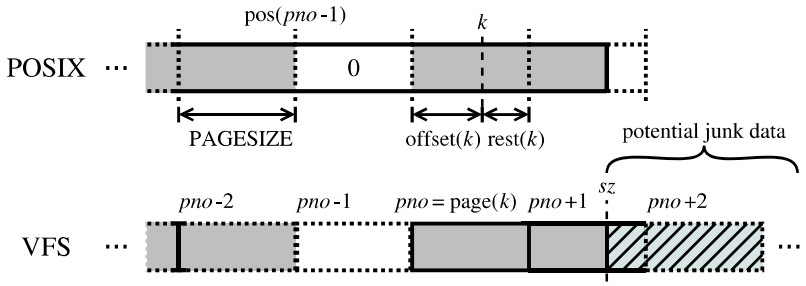


Fig. 2. Representation of file contents in POSIX and in VFS.

Both aspects will have a crucial impact on the cache implementation in Section 4 and the correctness proof in Section 5.

Section 3.2 will then give the implementation of writing a buffer to a file in VFS. This is sufficient to understand the implementation of caching, more details on other VFS operations was given in [14] and [15].

3.1 AFS: The interface of VFS to filesystems

The state of AFS is specified as abstractly as possible by two finite maps (Key $\rightarrow$ Value denotes a map from finitely many keys to values) with disjoint domains to store directories and files.

$$\textbf{state} \quad \mathit{dirs} : \mathsf{Ino} \rightarrow \mathsf{Dir} \quad \mathit{files} : \mathsf{Ino} \rightarrow \mathsf{File}$$

where $\mathsf{Ino} \simeq \mathbb{N}$ and

$$\textbf{data}\ \mathsf{Dir} = \mathtt{dir}(\mathtt{meta} : \mathsf{Meta},\ \mathtt{entries} : \mathsf{String} \rightarrow \mathsf{Ino})$$
$$\textbf{data}\ \mathsf{File} = \mathtt{file}(\mathtt{meta} : \mathsf{Meta},\ \mathtt{size} : \mathbb{N},\ \mathtt{content} : \mathbb{N} \rightarrow \mathsf{Page})$$

Both directories and files store some metadata `meta` that is mainly used to handle access rights. The entries of a directory are contained in the directory itself in the form of a mapping from entry names to inode numbers. These inode numbers identify another directory or file in the file system, so they also identify a Dir or File in *dirs* or *files* respectively.

Details on the representation of a file are shown in Fig. 2. The uniform representation as a sequence of bytes is broken up into file size and several pages. Each Page is an array of size PAGESIZE. Byte k of a file is accessed via $\mathtt{offset}(k)$ in $\mathsf{page}(k)$, which are the remainder and quotient when dividing k by PAGESIZE. We also use $\mathtt{rest}(k)$ to denote the length of the rest of the page above $\mathtt{offset}(k)$. We have $\mathtt{rest}(k) = \mathtt{PAGESIZE} - \mathtt{offset}(k)$, when the offset is non-zero. Otherwise $\mathtt{rest}(k) = 0$, k is (page-)*aligned*, and predicate $\mathtt{aligned}(k)$ is true. The start of page *pno* is at $\mathsf{pos}(\mathit{pno}) = \mathit{pno} * \mathtt{PAGESIZE}$. The pages are stored as a map, a missing page (e.g. page $\mathit{pno} - 1$ in the figure) indicates that the page contains zeros only. This sparse representation allows the creation of a file with large size without allocating all the pages immediately (which is important, e.g. for streaming data). Another important detail is that there may be irrelevant data beyond the file size. It is possible that the page $\mathsf{page}(\mathit{sz})$ at the file size sz contains random junk data (hatched part of the page) above $\mathtt{offset}(\mathit{sz})$ instead of just zeros. Extra (hatched) pages with a page number larger than $\mathsf{page}(\mathit{sz})$ are possible as well. Allowing such junk data is necessary for efficient recovery from a crash: writing data at the end of a file is always done by writing pages first, and finally incrementing the size. If a crash happens in between, then removing the extra data when rebooting would require to scan all files, which would be prohibitively expensive.

With this data representation AFS offers a number of operations that are called by VFS, using parameters of type Inode ("Index Node") and Dentry ("Directory Entry") as input and output (passed by reference). An inode represents an existing node of the file system tree and has the form

$$\textbf{data}\ \mathsf{Inode}\ =\ \mathtt{inode}(\mathtt{ino} : \mathsf{Ino},\ \mathtt{meta} : \mathsf{Meta},\ \mathtt{isdir} : \mathsf{Bool},\ \mathtt{nlink} : \mathbb{N},\ \mathtt{size} : \mathbb{N})$$

The boolean `isdir` distinguishes between directories and files, the `nlink`-field gives the number of hard links for a file (`nlink` = 1 for a directory). The `size`-field stores the file size for files and the number of entries for a directory.

Dentries form the edges of the file system tree by linking a parent directory to one of its children. Normal dentries store the `name` of the entry and the inode number `target` of the corresponding child. Negative dentries on the other hand are used to indicate that an entry with `name` does not exist in the directory.

$$\textbf{data}\ \mathsf{Dentry}\ =\ \mathtt{dentry}(\mathtt{name} : \mathsf{String},\ \mathtt{target} : \mathsf{Ino})\ |\ \mathtt{negdentry}(\mathtt{name} : \mathsf{String})$$

Directories are only affected by *structural operations*, i.e. operations that modify the tree structure of the file system, and hence *dirs* is only modified by those as well.

For example, the operation **afs_mkdir** shown in Fig. 3 is used to add a new directory node to the file system (all operations use semicolons to separate input, in/out, and output parameters). The operation creates this directory with the parent node *pinode* under the name *dent*.`name` and with metadata *md*. For this a fresh inode number *ino* is allocated (++) in *dirs*, a new empty directory (`dir`(*md*, ∅)) is stored under *ino*, and `entries` of the parent directory is updated by adding a link of *dent*.`name` to *ino*. Of course, it must be ensured that the operation does not accidentally overwrite an existing file or directory (if there is already an entry with *dent*.`name` in *dirs*[*pinode*.`ino`].`entries`), hence suitable preconditions are checked when the operation is called. These preconditions are established by the surrounding VFS operations (here **vfs_mkdir**). To ensure that existing inodes and dentries are still valid after the operation for the use in VFS, the operation also updates affected data structures at the end of the operation: the `target` of *dent* is set to the new *ino* and the size of the parent inode *pinode* is increased by one. In addition, the inode *cinode* of the created directory is also returned.

Fig. 3 also shows the approach for specifying errors in AFS. All AFS operations are allowed to non-deterministically (**or**) fail, i.e. return *err* = `true`. This allows the implementation to return errors, e.g. when there is not enough memory available, which can not be specified on this level of abstraction. The implementation will resolve the nondeterminism to success whenever possible.

Most VFS operations, in particular all structural operations, must traverse the directory tree using paths to the involved nodes before the respective AFS operation can be invoked. This is done by the operation **vfs_walk** listed in Fig. 4 on the left. Starting from an *ino*, it incrementally steps down one segment *path*.`head` of the path at a time by calling the AFS operation **afs_lookup** (Fig. 4 on the right). Before calling **afs_lookup**, **vfs_maylookup** verifies that the required permissions for reading the entries of *ino* are present. This includes to read the inode of *ino* from AFS in order to access its metadata. If the access is granted, **afs_lookup** checks whether an entry with the name of the requested dentry exists in the current directory *ino* and updates *dent* with the inode number of the entry *cino* if so. After successful traversal, the required inodes are loaded (for **afs_mkdir** the parent inode *pinode*), checks specific to the operation are performed, and the AFS operation is invoked.

```
afs_mkdir(md; pinode, cinode, dent; err) {
    choose ino with
        ino ≠ 0 ∧ ¬ ino ∈ dirs ∧ ¬ ino ∈ files
    in
        dirs := (dirs ++ ino)[ino, dir(md, ∅)];
        dirs[pinode.ino].entries[dent.name]
:= ino;
        dent := dentry(dent.name, ino);
        cinode := inode(ino, md, true, 1, 0);
        pinode.size := pinode.size + 1;
    err := false;
  or
    err := true;
}
```

Fig. 3. AFS operation for creating new directories.

```
vfs_walk(path; ino; err) {
  err := false;
  while path ≠ [] ∧ ¬ err do
    vfs_maylookup(ino; ; err);
    if ¬ err then
    let dent = negdentry(path.head)
in
      afs_lookup(ino; dent; err);
      if ¬ err then
        ino := dent.target;
        path := path.tail;
}
```

```
afs_lookup(ino; dent; err) {
    if dent.name ∈ dirs[ino].entries then
      let cino = dirs[ino].entries[dent.name]
in
        dent := dentry(dent.name, cino);
        err := false;
    else
      dent := negdentry(dent.name);
      err := true;
  or
    err := true;
}
```

Fig. 4. Operations for traversing the directory tree: walking along a path in VFS (left) and looking up a dentry in AFS (right).

The *content operations* of AFS, i.e. the operations for modifying file content, are specified in Fig. 5. Recall that unlike structural operations, they only access *files* and do not read or update *dirs*. Since we are interested in adding write-back caches for file content, we give a short description for each of these operations, which also includes some preconditions.

- **afs_rpage** reads the content of the page with number *pno* into a buffer *pbuf*: Page. The file is determined as the inode number of a file inode *inode*. If the page does not exist, the buffer is set to all zeros (abbreviated as $\perp$), and the *exists* flag is set to false. The flag is ignored by VFS but will be relevant for implementing a cache in the next section.
- **afs_wpage** writes the content of *pbuf* to the respective page *pno*. Note that the page is allowed to be beyond file size (which is not modified).
- The file size is changed with the operation **afs_wsize**. This operation does not check, whether there are junk pages above the new file size.
- **afs_fsync** synchronizes a file. If a crash happens directly after this operation, the file accessed by *inode* must retain its content. On this abstract level, the operation does nothing. Its implementation, which uses an order-preserving write-back cache (see [35]) must empty this cache.
- **afs_truncate** is used to change the file size to n, checking that there is no junk data that would end up being part of the file below the new file size. This operation first discards all pages above the minimum sz_T of n and the old sz: The expression *cont* upto sz_T keeps pages below sz_T only. For efficiency, the operation then distinguishes two cases, shown in Fig. 6. The first case a) is when the new size n is at least the old sz. In this case, the page page(sz) may contain junk data, which must be overwritten by zeros since this range becomes part of the file. Overwriting the part above $sz_T = sz$ with zeros is the result of the function call truncate($cont[pno], sz_T$). This call can be avoided, if the part is empty or if the old size was aligned. The second case b) is when the new file size is less than the old. In this case, the page above the new file size simply becomes junk, it does not need to be modified. The implementation of the **afs_truncate** operation, therefore, avoids writing pages to persistent store whenever this is possible[3].
- **afs_wbegin** is an optimized version of **afs_truncate** for the case $n = sz$. It is called at the start of writing content to a file in VFS. It makes sure that writing beyond the old file size will not accidentally create a page, which contains junk.

[3]Deleting a page does not write it, but adds a "page deleted" entry to the journal.

```
afs_rpage(inode, pno; pbuf, exists; err) {
    exists := pno ∈ files[inode.ino].content;
    if exists then
      pbuf := files[inode.ino].content[pno];
    else
      pbuf := ⊥;
    err := false;
  or
    err := true;
}

afs_wpage(inode, pno, pbuf; ; err) {
    files[inode.ino].content[pno] := pbuf;
    err := false;
  or
    err := true;
}

afs_wsize(inode, sz; ; err) {
    files[inode.ino].size := sz;
    err := false;
  or
    err := true;
}

afs_fsync(inode; ; err) {
    err := false;
  or
    err := true;
}
```

```
afs_wbegin(inode; ; err) {
    let sz = inode.size in
    let cont = files[inode.ino].content,
        pno = page(sz),
        aligned = aligned(sz) in
      if pno ∈ cont ∧ ¬ aligned then
        cont[pno] := truncate(cont[pno], sz);
      files[inode.ino].content := cont upto sz;
    err := false;
  or
    err := true;
}

afs_truncate(n; inode; err) {
    let sz = inode.size in
    let cont = files[inode.ino].content,
        sz_T = min(n, sz),
        pno = page(sz),
        aligned = aligned(sz) in
      if sz ≤ n ∧ pno ∈ cont ∧ ¬ aligned then
        cont[pno] := truncate(cont[pno], sz_T);
      files[inode.ino].content := cont upto sz_T;
      files[inode.ino].size := n;
      inode.size := n;
    err := false;
  or
    err := true;
}
```

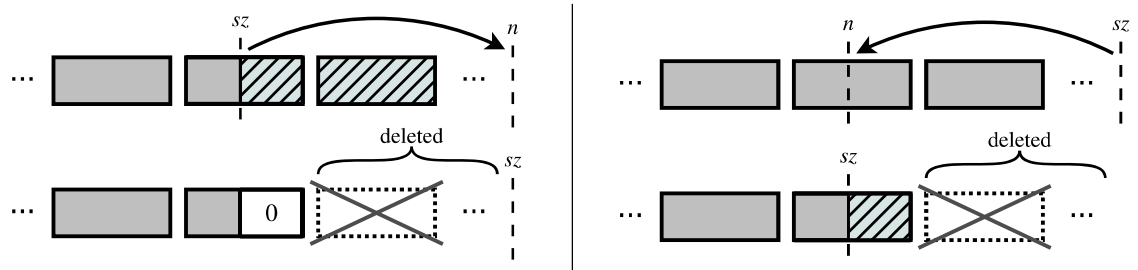
Fig. 5. AFS operations for accessing (reading and modifying) file contents.

3.2 The implementation of VFS

On the basis of these AFS operations, writing a buffer *buf* (an array) of length *n* to a file is implemented in VFS with the **vfs_write** operation shown in Fig. 7. This operation has the inode number *ino* of a file as input, and the seek position *pos*, where the buffer should be placed.

According to the POSIX specification, writing is allowed to write an initial segment of *buf*. Therefore the operation modifies *n* at the end to return the number of bytes that were actually written. The operation starts by calling **afs_iget**, which determines an *inode* and thereby the current size of the file. Then **afs_wbegin** is called to ensure that junk data above the file size is removed.

Writing data then is done by splitting up the range of the buffer (from *pos* to *end*) into pieces that align with page boundaries in **vfs_wloop**. The variable *written* keeps track of how many bytes have been written already.



(a) Growing truncation ($sz \leq n$). (b) Shrinking truncation ($n < sz$).

Fig. 6. Effects of a truncation to n on a file with size sz.

```
vfs_write(ino, pos, buf; n; err) {
  let inode in
    afs_iget(ino; inode; err);
    if ¬ err then
    let end = pos + n, written = 0 in
      afs_wbegin(inode; ; err);
      if ¬ err then
        vfs_wloop(inode, pos, end, buf; ;
                  written, err);
        if written ≠ 0 then
          err := false;
      if ¬ err then
        if inode.size < pos + written then
          afs_wsize(inode, pos + written; ;
err);
        if ¬ err then
          n := written;
        else let sz = inode.size in
          n := (pos ≤ sz ⊃ sz − pos ; 0);
}
```

```
  vfs_wloop(inode, start, end, buf; ; written, err) {
  err := false, written := 0;
  while ¬ err ∧ start + written ≠ end do
    let pos = start + written in
    let offset = offset(pos),
        pno = page(pos)
        rest = rest(pos) in
    let n = min(end − pos, rest) in
      vfs_wpage(inode, pno, buf,
                written, offset, n; ; err);
      if ¬ err then
        written := written + n;
}

vfs_wpage(inode, pno, buf, written, offset, n; ; err)
{
  err := false;
  let pbuf = ⊥, exists = false in
    if n < PAGESIZE then
      afs_rpage(inode, pno; pbuf, exists; err);
    if ¬ err then
      pbuf := copy(buf, written, pbuf, offset, n);
      afs_wpage(inode, pno, pbuf; ; err);
}
```

Fig. 7. Write operation of VFS (left) with auxiliary operations (right).

For each page-aligned piece of the buffer **vfs_wpage** is called. This still has the full buffer *buf* as input and writes n bytes starting at position *written*. The n bytes are placed into the page *pno*, starting at *offset*. The sum of n and *offset* always is less or equal to PAGESIZE. If it is less than page size (i.e. when either $offset \neq 0$ or $n \neq$ PAGESIZE), the old page must be loaded into *pbuf* by calling **afs_rpage** and the relevant part of *buf* must be copied into the page using the function copy. *offset* is set to offset(*pos*) for the smallest piece that is written in the loop and is set to zero for all the other pieces.

Since all calls of AFS operations can return an error and writing stops in this case, the number *written* of bytes actually written is returned from **vfs_wloop**. It is finally used in **vfs_write** to modify *n*. If the file size has been increased, which is the case when *start* + *written* is bigger than the old file size, **vfs_write** finally adjusts it by calling **afs_wsize**.

VFS also offers a **vfs_truncate** operation (not shown) for changing the size of a file. This operation takes a path as input and simply calls **afs_truncate** after walking to the corresponding file node. Writing and truncating are the central two operations that are affected when write-back caching is added (together with a non-trivial implementation of **afs_fsync**).

We will see in the next section that when adding caches it is crucially important for correctness that VFS implements writing by traversing the pages from low to high page numbers. We will also find that the data representation of VFS, where all calls are optimized for efficiency, which in particular results in an asymmetric **afs_truncate** (Fig. 6), is one of the main difficulties for adding caches correctly.

4 INTEGRATION OF CACHES INTO FLASHIX

Initially, Flashix was developed without caches for high-level data structures. To add such caches to Flashix we introduce a new layer between the Virtual File System Switch and the Flash File System, visualized in Fig. 8.

This layer is implemented as a *Decorator* [19], i.e. it implements the same interface namely AFS and delegates calls from the VFS to the FFS which also has AFS as its interface. The VFS communicates with a Cache Controller (Cache) which in turn communicates with the FFS and manages caches for inodes (ICache), dentries (DCache), pages (PCache), and an auxiliary cache for truncations (TCache).

The ICache, DCache, and PCache components internally store maps from unique identifiers to the corresponding data structures. They all offer interfaces to Cache for adding resp. updating, reading, and deleting cache entries. Cache is responsible for processing requests from the VFS by either delegating these requests to the FFS or fulfilling them with the help of the required caches. It also has to keep the caches consistent with data stored on flash, i.e. update cached data when changes to corresponding data on flash have been made.

4.1 Write-Through Caching of Structural Operations

Similar to the Linux VFS, information about the file system structure is cached in a write-through manner to speed up read accesses. This includes dentries in DCache as well as structural fields of inodes in ICache like `size` of directory inodes or `nlink`. By not caching changes to the file system structure we ensure the integrity of the file system tree after a crash since structural operations are usually highly dependent on one another and affect multiple data objects. For example the creation of a directory can only take place after the parent directory was created and linking of a file results in changes to the file itself as well as to the directory in which the link is created.

Nevertheless, adding such write-through caches has a noticeable impact on the performance of structural operations as well. Consider for example the typical sequence arising from the **posix_mkdir** operation in Fig. 9, which is representative for all structural operations. As explained in Sec. 3, the initial path traversal comprises multiple reads of inodes and dentries during **vfs_walk** (see Fig. 4). This is done by calling the AFS interface operations **afs_iget** and **afs_lookup** which typically results in slow reads from flash via the FFS when no cache is used. However, with the caching layer, these requests can potentially be handled by the ICache and the DCache, as seen in the example sequence. If a *cache miss* occurs, i.e. a requested data object is not cached, the object must still be read from flash, however, it is then added to the cache for future queries. Cache passes the actual update calls of VFS simply to FFS (**afs_mkdir** in the example) and hence the file system is still modified persistently. But the caching layer takes advantage of the fact that FFS can return all modified or created data structures during the operation (for **afs_mkdir** in Fig. 3 these are *pinode*, *cinode*, *dent*) and adds these data structures to the caches or updates them in the caches respectively (with the **icache_set** and **dcache_set** operations). This significantly reduces the number of read accesses to the flash storage for everyday workloads such as extracting an archive, which requires traversing the full path to the parent directory each time a node is created.

4.2 Write-Back Caching of Content Operations

Compared to structural operations, updates to file data can be considered mostly in isolation. This means that in particular reads and writes to different files do not interfere with each other. Therefore we allow write-back caching of POSIX operations that modify the content of a file, namely *write* and *truncate*. Hence, the Cache Controller does not forward page writes to the FFS and instead only stores the pages in the Page Cache. Updates to the size of a file are also performed in the Inode Cache only as garbage data could be exposed in the event of a power cut otherwise. To distinguish between up-to-date data and cached updates, entries of the Page Cache or the Inode Cache include an additional *dirty* flag. For the Page Cache, this results in a mapping from inode numbers and page numbers to entries consisting of a page-sized buffer and a boolean flag.

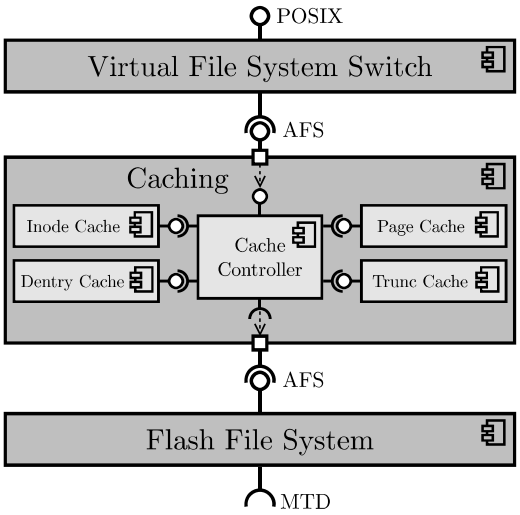


Fig. 8. Flashix component hierarchy.

Fig. 10 lists the central operations of the PCache component using the state *pcache*: Ino × Nat → Bool × Page. Analogously the components ICache and DCache are defined. ICache stores a mapping *icache*: Ino → Bool × Inode from inode numbers to entries containing the inode itself and a dirty flag. DCache uses a mapping *dcache*: Ino × String → Dentry from directory inode numbers and entry names to dentries (only real dentries are stored, no negdentrys). No dirty flag is necessary for the entries of *dcache* since they are always identical to their persisted counterparts. The auxiliary cache TCache records the minimal truncation size sz_T of a file since the last synchronization as well as its current persisted size sz_F using a mapping *tcache*: Ino → Nat × Nat.

Writing pages or file sizes results in putting the new data *dirty* in the particular caches. These operations of the controller component Cache are shown in Fig. 11 on the right. Reading pages on the other hand returns the page in question stored in PCache or, if it has not been cached yet, it tries to read it from flash (Fig. 11 on the left). But reading from flash yields the correct result only if there was no prior truncation that would have deleted the

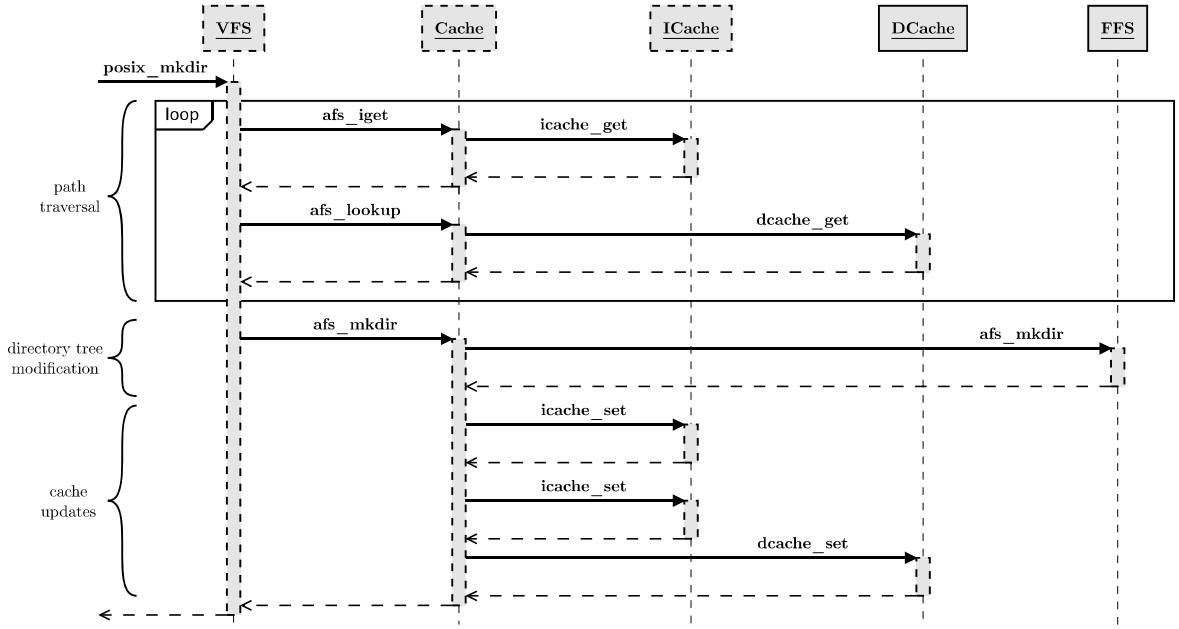


Fig. 9. Interaction of VFS, FFS, and the caching layer during a **posix_mkdir** operation.

```
state pcache: Ino × Nat → Bool × Page

pcache_set(ino, pno, pbuf, dirty) {
  let key = ino × pno in
    pcache[key] := dirty × pbuf;
}

pcache_delete(ino, pno) {
  let key = ino × pno in
    pcache := pcache −− key;
}
```

```
pcache_get(ino, pno; pbuf, dirty; hit) {
  let key = ino × pno in
    hit := key ∈ pcache;
    if hit then
      dirty := pcache[key].dirty;
      pbuf := pcache[key].page;
}

pcache_mark_clean(ino, pno) {
  let key = ino × pno in
    pcache[key].dirty := false;
}
```

Fig. 10. Core of the PCache component.

relevant page, i.e. an entry for this file exists in TCache and applying this truncation would delete the requested page (if *pno* is beyond the cached truncate size sz_T or the current persisted size of the file sz_F). If reading the page from flash is correct and the page actually stores any relevant data (*exists* is true), the resulting page is stored *clean* in PCache to handle repeated read requests.

For truncations of files, there are several steps Cache needs to perform. These steps are implemented with the operations **cache_truncate** and **cache_wbegin** as shown in Fig. 12. First, when an actual user truncation is executed, ICache needs to be updated by setting the size to the size the file is truncated to. Second, cached pages beyond *sz* resp. *n* have to be removed from PCache and the truncate sizes in TCache have to be updated. For this

```
cache_rpage(ino, pno; pbuf, exists; err) {
  let hit = false, dirty = false in
    pcache_get(ino, pno; pbuf, dirty; hit);
    if hit then
      exists := true, err := false;
    else let sz_T = 0, sz_F = 0 in
      tcache_get(ino; sz_T, sz_F; hit);
      if hit ∧ min(sz_T, sz_F) ≤ pos(pno) then
        pbuf := ⊥, exists := false, err := false;
      else
        afs_rpage(ino, pno; pbuf; exists, err);
        if ¬ err ∧ exists then
          dirty := false;
          pcache_set(ino, pno, pbuf, dirty);
}
```

```
cache_wpage(ino, pno, pbuf; ; err) {
  err := false;
  let dirty = true in
    pcache_set(ino, pno, pbuf, dirty);
}

cache_wsize(inode, sz; ; err) {
  err := false;
  let dirty = true in
    inode.size := sz;
    icache_set(inode, dirty);
}
```

Fig. 11. Cache operations for reading and writing pages and updating file sizes.

purpose, the two subcomponents provide dedicated truncation operations **pcache_truncate** and **tcache_update**, respectively. **tcache_update** aggregates multiple truncations by caching the minimal truncate size n for each file only. Additionally, the persisted size sz of a file is stored in TCache to determine whether it is allowed to read a page from flash in **cache_rpage**. **pcache_truncate** ensures that no truncated data is cached by removing all entries $ino \times pno$ from PCache where $sz_T \leq \mathsf{pos}(pno)$. Finally, if the truncate is growing, i.e. $sz \leq n$, the page at size sz may need to be filled with zeros. The auxiliary operation **cache_get_tpage** is used to determine if this page is existent. This is the case if the page is either cached in PCache or can be read from flash but would not have been truncated according to TCache. If necessary, the page is then filled with zeros beyond `offset`(sz) using the `truncate` function and the result is stored in PCache.

4.3 Synchronization of File Contents

The synchronization of files, i.e. transferring cached updates to the persistent storage, is coordinated by Cache too. Clients can use the POSIX *fsync* operation to trigger synchronization of a specific file.

The implementation of *fsync* in Cache is shown in Fig. 13. The general idea of this implementation is to first remove all pages from flash that would have been deleted by truncations on this file since the last synchronization and then mimic a VFS *write* that persists all *dirty* pages in PCache and updates the file size to the size stored in ICache if necessary.

The operation **cache_fbegin** is responsible for synchronizing truncations and prepares the subsequent writing of pages and updating the file size in **cache_fpages** resp. **cache_finode**. When using this synchronization strategy, it is sufficient to aggregate multiple truncations by truncating to the minimal size the file was truncated to, and only if this minimal truncation size is lower than the current file size on flash. As truncation is the only possibility to delete pages (except for deleting the file as a whole), this **afs_truncate** call deletes all obsolete pages. The following **afs_wbegin** call ensures that the whole file content beyond sz_T resp. sz_F is zeroed so that writing pages and increasing the file size on flash is possible safely. Since AFS enforces an initial **afs_wbegin** before writing pages or updating the file size and Cache is a refinement of AFS, it is guaranteed that there are dirty pages in PCache or dirty inodes in ICache only if there is an entry in TCache for the file that is being synchronized. Hence there is nothing to do if *hit* after **tcache_get** is false.

```
cache_truncate(n; inode; err) {
  let ino = inode.ino, sz = inode.size in
  let sz_T = min(n, sz) in
  let pno = page(sz_T), pbuf = ⊥,
      hit = false, dirty = true in
    cache_get_tpage(ino, pno; pbuf; hit,
err);
    if ¬ err then
      pcache_truncate(ino, sz_T);
      if hit ∧ sz ≤ n ∧ ¬ aligned(sz) then
        pbuf := truncate(pbuf, sz);
        pcache_set(ino, pno, pbuf, dirty);
      tcache_update(ino, n, sz);
      inode.size := n;
      icache_set(inode, dirty);
}
```

```
cache_wbegin(inode; ; err) {
  let ino = inode.ino, sz = inode.size in
  let pno = page(sz), pbuf = ⊥, hit = false in
    cache_get_tpage(ino, pno; pbuf; hit, err);
    if ¬ err then
      pcache_truncate(ino, sz);
      if hit ∧ ¬ aligned(sz) then
        pbuf := truncate(pbuf, sz);
        let dirty = true in
          pcache_set(ino, pno, pbuf, dirty);
      tcache_update(ino, sz, sz);
}

cache_get_tpage(ino, pno; pbuf; hit, err) {
  err := false;
  let dirty = false in
    pcache_get(ino, pno; pbuf, dirty; hit);
    if ¬ hit then let sz_T = 0, sz_F = 0 in
      tcache_get(ino; sz_T, sz_F; hit);
      if ¬ hit ∨ pos(pno) < min(sz_T, sz_F) then
        afs_rpage(ino, pno; pbuf; hit, err);
}
```

Fig. 12. Cache operations for truncations of file contents.

If there is dirty data to persist (*sync_data* was set to true), **cache_fpages** iterates over all possibly cached pages of the file and writes *dirty* pages with **afs_wpage**, marking them *clean* in PCache after writing them successfully. Similar to the implementation of **vfs_write** explained in Sec. 3, this iteration is executed bottom-up, starting at page 0 up to the maximal page cached in PCache (returned by **pcache_max_pageno**). After all pages have been synchronized successfully, **cache_finode** updates the file size with **afs_wsize** if the cached size is greater than the persisted size sz_F. Finally, the lower levels are synchronized as well by calling **afs_fsync** to ensure that all updates to the file have actually been written completely to flash as dictated by the POSIX standard.

4.4 Evaluation

To conclude this section, we want to discuss briefly the impact of the Cache integration on the performance of Flashix. We ran a couple of microbenchmarks with workloads that represent everyday usage of file systems, like extracting and creating archives. As shown in Fig. 14, adding the caches described in this section actually has the desired effects on such workloads. Extracting an archive to the file system, which results in the creation of many directories and files and yields many relatively small writes to these files, is over 50 times faster with our caches since most updates (more precisely all updates triggered by *content operations*) can be performed purely in RAM. Then the synchronization via *fsync* obviously takes more time, but the combined running time is significantly faster as repeated reads of directory and file nodes during path traversal can also be handled by the cache. This effect also shows when an archive is created from a directory structure: reading all nodes, dentries, and content pages necessary to create the archive from RAM is considerably faster as well (for the benchmark shown in Fig. 14, over 30 times if the cache is *hot*, i.e. all relevant data is present in the cache, and about 3 times if the cache is *cold* after a remount).

```
cache_fsync(inode; ; err) {
  let sz_F = 0, sync_data = false in
    cache_fbegin(inode; sz_F; sync_data, err);
    if ¬ err ∧ sync_data then
      cache_fpages(inode; ; err);
    if ¬ err ∧ sync_data then
      cache_finode(inode, sz_F; ; err);
    if ¬ err then
      afs_fsync(inode; ; err);
}

cache_fbegin(inode; sz_F; sync_data, err) {
  err := false;
  let hit = false, sz_T = 0 in
    tcache_get(inode.ino; sz_T, sz_F; hit);
    sync_data := hit;
    if sync_data then
      if sz_T < sz_F then
        afs_truncate(sz_T; inode; err);
        sz_F := sz_T;
      if ¬ err then
        afs_wbegin(inode; ; err);
      if ¬ err then
        tcache_delete(inode.ino);
}
```

```
cache_fpages(inode; ; err) {
  err := false;
  let ino = inode.ino, pno = 0, pno_max = 0 in
    pcache_max_pageno(ino; ; pno_max);
    while ¬ err ∧ pno ≤ pno_max do
      let pbuf = ⊥, hit = false, dirty = false in
        pcache_get(ino, pno; pbuf, dirty; hit);
        if hit ∧ dirty then
          afs_wpage(ino, pno, pbuf; ; err);
          if ¬ err then
            pcache_mark_clean(ino, pno);
      pno := pno + 1;
}

cache_finode(inode, sz_F; ; err) {
  if sz_F < inode.size then
    afs_wsize(inode, inode.size; ; err)
  else
    err := false
}
```

Fig. 13. File synchronization operations of Cache.

The results shown here are an excerpt of a more in-depth evaluation presented in [3]. There we also compared Flashix in its cached version with the state-of-the-art flash file system UBIFS [24]. This comparison showed that the integration of caches significantly closed the gap between Flashix and real-world file systems. However, Flashix is still up to a factor of 10 slower when considering raw in-memory runtimes only, i.e. when ignoring the delays that result from accessing the flash memory. A large part of this deficit comes from a non-optimal code generation which must bridge the gap between the simple value-semantics of our programs (inherited from predicate logic) to a pointer-semantics used by C (first experiments with ad-hoc optimizations support this assumption). Hence, we plan to improve our code generator in future work, especially with regard to avoiding unnecessary allocations and copies of data structures.

5 FUNCTIONAL CORRECTNESS AND CRASH-SAFETY CRITERION

Due to our modular approach, verifying the correctness of integrating caches into Flashix as shown in Fig. 1b requires to prove a single additional data refinement $\mathsf{Cache}(\mathsf{AFS_P}) \preceq \mathsf{AFS_C}$ only. The proofs are done with a forward simulation $R \subseteq AS \times CS$ using commuting diagrams with states $AS \equiv dirs_C \times files_C$ of $\mathsf{AFS_C}$ and $CS \equiv dirs_P \times files_P \times icache \times pcache \times tcache$ of $\mathsf{Cache}(\mathsf{AFS_P})$.

$$R \quad \equiv \quad dirs_C = dirs_P \wedge files_C = ((files_P \downarrow tcache) \oplus pcache) \oplus icache$$

[4] vim-7.4.tar: approx. 2570 elements, 40.9 MB

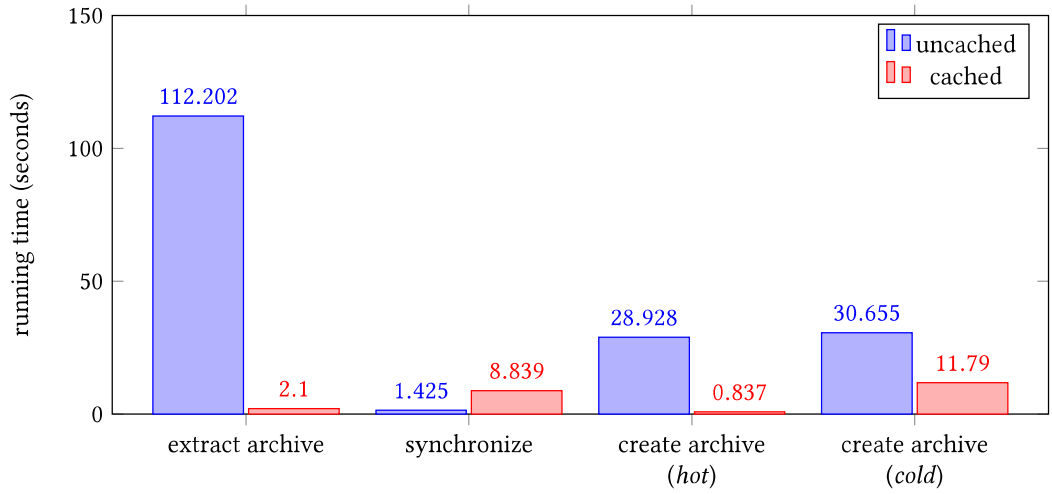


Fig. 14. Running time comparison between Flashix with and without the Cache component. As sample data an archive of the text editor Vim[4] was used.

Basically, R states that for each $(as, cs) \in R$ the cached AFS state as can be constructed from cs by applying all cached updates to the persistent AFS state, i.e. pruning all files at their cached truncate size ($\_ \downarrow tcache$), overwriting all pages with their cached contents ($\_ \oplus pcache$), and updating the cached file sizes ($\_ \oplus icache$). As no structural operations are cached, $dirs_C$ and $dirs_P$ are identical.

While AFS_C functionally matches the original specification of AFS, it is easy to see that AFS_C differs quite heavily from AFS_P in terms of its crash behavior. A crash in AFS_P, for example, has the effect of removing orphaned files [13], i.e. those files that are not accessible from the file system tree anymore but still opened in VFS for reading/writing at the event of the crash. However, if there are pending writes that have not been synchronized yet, a crash in AFS_C additionally may revert parts of these writes as all data only stored in the volatile state of Cache is lost.

Usually, we express the effect of a crash in *specification* components in terms of a state transition given by a *crash predicate* $\mathtt{Crash} \subseteq S \times S$ (see Sec. 7.1) and prove that the *implementations* of these components match their specification. But as soon as write-back caches - especially non-order-preserving ones - are integrated into a refinement hierarchy, it is typically not feasible to express the loss of cached data explicitly. This is the case for AFS_C and thus for POSIX, too. So instead of verifying crash-safety in a state-based manner, we want to explain the effects of a crash by constructing an alternative run where losing cached data does not have any effect on the state of AFS_C. If such an alternative run can always be found, crash-safety holds since all regular (non-crashing) runs of AFS_C yield consistent states, and thus a crash results in a consistent state as well.

In natural language, our new correctness criterion *Write-Prefix Crash Consistency* can be formulated as follows:

> *A file system is write-prefix crash consistent (WPCC) iff a crash keeps the directory tree intact and for each file f a crash has the effect of retracting all write and truncate operations to f since the last state it was synchronized and re-executing them, potentially resulting in writing prefixes of the original runs.*

This property results from the fact that files are synchronized individually by the *fsync* operation. Thus, all runs of operations that modify the content of a file, either cached or persistent, can be decoupled from runs of structural operations or operations accessing the content of other files.

We now formalize this criterion in terms of histories. Executions of POSIX-compliant file systems can be considered as *runs* $r = s_0 \xrightarrow{l_0} s_1 \xrightarrow{l_1} s_2 \xrightarrow{l_2} \dots \xrightarrow{l_{n-1}} s_n$ of a labeled transition system (LTS) or an IO-Automata [30] with POSIX states $s_0, s_1, s_2, \dots, s_n \in S$, labels $l_0, l_1, l_2, \dots, l_{n-1} \in \mathcal{L} \cup \{\tau\}$, and a state transition relation $\rightarrow \subseteq S \times \mathcal{L} \cup \{\tau\} \times S$. $\mathcal{L}$ consists of *invocation* events $inv_{tid}(\mathsf{op}, in)$ and *response* events $res_{tid}(\mathsf{op}, out)$ of POSIX operations op, and a *crash event* $\lightning$. An internal label τ is used for steps of the system that are not visible to the environment. The inputs and outputs of operations are recorded with *in* and *out*, respectively, and each operation event is marked with the thread identifier $tid \in Tid$ of the thread triggering it. A *history* $h = l_0 l_1 l_2 \dots l_{n-1}|_{\mathcal{L}}$ is a sequence of external labels $l_i \in \mathcal{L}$ that records the invocation and response events of operations executed in the system as well as system-wide crashes, we abbreviate $s \xrightarrow{h} s'$ for a run from s to s' producing the history h. For a given initial state $s_0 \in S$, a history h is *legal* iff there exists a state s with $s_0 \xrightarrow{h} s$. Crash events partition a history h into *eras* $h_0, h_1, \dots, h_n$ such that $h = h_0 \lightning_1 h_1 \lightning_2 \dots \lightning_n h_n$. Eras are *crash-free*, i.e. $\lightning \notin h_i$, and thus, $\lightning_n$ denotes the n-th crash of h. After each crash, a *recovery operation* Rec is run to re-establish a consistent state (on the level of POSIX, this is simply the mount operation). Hence, each era starts with an (I, R) pair of Rec. This operation does not interleave with any other operation (user calls to the file system are allowed only after the recovery has finished), and its correctness is proved separately, so we do not consider Rec explicitly in the following.

A history h is *sequential* iff it is crash-free, it starts with an invocation event, and every invocation event is immediately followed by its corresponding response event (except if it is the last event in h). A concurrent history h is *well-formed* iff $h|_{tid}$ is sequential for every thread *tid* where $h|_{tid}$ denotes the restriction of h to the events of thread *tid*.

Similar to (durable) linearizability [21, 25], we distinguish between *completed operations* and *pending operations*. A completed operation in a history h is any pair (I, R) of invocation $I = inv_{tid}(\mathsf{op}, in)$ and matching response $R = res_{tid}(\mathsf{op}, out)$. A pending operation is an invocation $I = inv_{tid}(\mathsf{op}, in)$ where I has no matching response in h. Pending operations I can be *completed* by appending a response R to h such that (I, R) is a completed operation in hR.

For now, we will consider only sequential runs of a single thread, i.e. we assume that there are no concurrent calls to the file system interface, and therefore omit the *tid* subscripts of the events. The compatibility of our criterion with concurrent POSIX calls is discussed in Sec. 7.2.

To define *Write-Prefix Crash Consistency*, we will now focus on content operations. Since content operations always affect only one file f, we restrict histories h to histories $h|_f$ that contain only the content operations to f. These include write, read, truncation, and synchronization operations so that $h|_f$ consists of matching pairs of the form

$$(inv(\texttt{write}, f, pos, buf, n), res(\texttt{write}, n', err))$$
$$(inv(\texttt{read}, f, pos, n), res(\texttt{read}, n', buf, err))$$
$$(inv(\texttt{truncate}, f, n), res(\texttt{truncate}, err))$$
$$(inv(\texttt{fsync}, f), res(\texttt{fsync}, err))$$

Each invocation is marked with the targeted file f and each response contains the error flag *err* recording whether the operation was successful, i.e. no error occurred and *err* = `false`, or failed, i.e. *err* = `true`. n' denotes the potentially altered value of the reference parameter n and hence records how many bytes were written or read, respectively.

Note that fixing the returned errors and the number of written/read bytes in the responses resolves all non-determinism (`POSIX` is non-deterministic just in succeeding or failing of operations and in writing or reading prefixes of the requested number of bytes, see [15])[5] and hence, the LTS of `POSIX` is *deterministic*. This property makes it possible to argue solely with histories about crashes since a history determines the complete run for a given initial state.

WPCC will use *write-prefix histories* to give an alternative explanation for crashed runs. A notion for *prefix operations* is needed to define such histories. For a completed content operation (I, R), a corresponding prefix operation is a tuple (I, R') where R' results from changing the outputs of R in the sense that

- operations that were originally successful, i.e. *err* = `false`, may fail in the prefix operation, i.e. have the output *err* = `true` (this includes that the remaining outputs may differ as well, e.g. the *buf* of a read response could still be filled with nothing but zeros), and
- write operations that have written m bytes originally, may write less bytes $m' \leq m$ in the *prefix write*.

Note that we do not define prefixes of read operation other than failed executions (that did not read anything at all). While it would be feasible to allow reads that have read some but not all bytes of the original operation, this is not necessary as reads do not affect the observable behavior of following operations and hence can simply considered as failed or omitted altogether when considering crash-safety.

Write-prefix histories should contain prefixes for non-synchronized operations only, for this a history h can be split in two histories $h_{\leq sync}$ and $h_{>sync}$ where $h = h_{\leq sync} h_{>sync}$, the non-synchronized part $h_{>sync}$ does not contain an event *res*(`fsync`, `false`), and the synchronized part $h_{\leq sync}$ either ends with an *res*(`fsync`, `false`) or is empty (if h does not contain any successful response of `fsync`). Thus, $(h|_f)_{\leq sync}$ contains all synchronized write, read, and truncate operations as well as all complete, successful fsync operations of f. Conversely, $(h|_f)_{>sync}$ contains all pending fsync operations of f and all write, read, and truncate operations of f that are not (fully) synchronized. Both histories, however, may contain failed fsync operations.

Definition 2 (Write-Prefix Crash Consistency). *A file system is write-prefix crash consistent (WPCC) iff for every legal history $h = h_0 \lightning_1 h_1 \lightning_2 \ldots \lightning_{n-1} h_{n-1} \lightning_n h_n$ with well-formed eras $h_0, h_1, \ldots, h_{n-1}, h_n$ there exists a legal write-prefix history $h' = h'_0 h'_1 \ldots h'_{n-1} h_n$ where for each h'_i with $0 \leq i < n$*

(1) $\lightning \notin h'_i$,
(2) *h'_i contains invocations and responses of the same operations in the same order as h_i with the exception that all pending operations of h_i are completed in h'_i,*
(3) *completed structural operations in h_i are identical in h'_i, and*
(4) *for each file f, $h'_i|_f = (h_i|_f)_{\leq sync}\ h''_i|_f$ where $h''_i|_f$ is derived from $(h_i|_f)_{>sync}$ by replacing write, read, and truncate operations with matching prefix operations, and by completing pending operations.*

The main point of this criterion is that runs containing crashes (represented by legal histories h) can be explained by alternative legal histories h' that do not contain crash transitions (point 1 of Def. 2). Points 2-4 of Def. 2 ensure that each h'_i is again well-formed and as close as possible to the original era h_i: the same operations must be executed in the same order and with the same inputs, structural operations must yield the same results (and hence, yield the same file system tree), only non-synchronized content operations may have slightly different results.

[5] The creation of a file also chooses fresh file identifier non-deterministically, however, these choices are unaffected by content operations and are thus negligible for our considerations.

Note that our definition completes all pending operations while general linearizability completes some and deletes the remaining pending operations. This simplification is possible since each POSIX operation has a completion that fails non-deterministically and does not change the state, so the completed operation is equivalent to not executing it.

The last era h_n of h is unchanged in the write-prefix history h'. Since the LTS is deterministic and h' must be legal, this ensures that h' actually gives an alternative explanation for states resulting from crashed runs. The era h_n must be legal for both the final states s and s' of the runs $s_0 \xrightarrow{h_0 \lightning_1 \ldots h_{n-1} \lightning_n} s$ and $s_0 \xrightarrow{h'_0 \ldots h'_{n-1}} s'$. As h_n is arbitrary, it could contain read operations to any information stored in the file system, such as reading the complete content of a file, reading metadata of a file or directory, or reading the entries of a directory. These operations must yield the same results in both runs (h would not be legal otherwise), and hence s and s' must be identical, i.e. must be the same abstract POSIX state (we will see in the following section that the internal representation of the file system may be slightly different).

6 PROVING CRASH-SAFETY

Given the WPCC criterion of Def. 2, this section will present how we have proved that Flashix is crash-safe, i.e. that the following theorem holds.

Theorem 1 (Flashix satisfies WPCC). *The Flashix file system with the* VFS *implementation given in Sec. 3, extended by the* Cache *component of Sec. 4, satisfies WPCC.*

The basic proof idea for Theorem 1 is to consider a history $h = h_0 \lightning h_1$ with just a single crash $\lightning$, construct a write-prefix era h'_0 satisfying Def. 2, and show that h'_0 is legal. We prove that runs of $h_0 \lightning$ and h'_0 yield identical POSIX states, which guarantees that 1) the write-prefix history $h' = h'_0 h_1$ is legal as well, and 2) legal write-prefix histories for histories with an arbitrary number of crashes can be constructed inductively.

Whereas WPCC could be formulated purely in terms of POSIX histories, we now have to consider implementation runs for proving, in particular the ones of $\mathsf{VFS}(\mathsf{Cache}(\mathsf{AFS_P}))$. In earlier work [13] we have already shown, that the lower levels of Flashix can be assumed to be atomic w.r.t. to crashes, i.e. a crash during an $\mathsf{AFS_P}$ operation has the same effect as a crash either directly before or directly after the operation. This still holds for $\mathsf{AFS_P}$ (the implementation by a transactional journal ensures that), hence crashes in $\mathsf{Cache}(\mathsf{AFS_P})$ have to be contemplated only between $\mathsf{AFS_P}$ operations. The proofs, therefore, revolve around constructing matching alternative runs in $\mathsf{AFS_C}$ for all possible crashes in $\mathsf{Cache}(\mathsf{AFS_P})$. In a next step, these runs have to be lifted to $\mathsf{VFS}(\mathsf{AFS_C})$ runs via the refinement $\mathsf{Cache}(\mathsf{AFS_P}) \preceq \mathsf{AFS_C}$, the refinement $\mathsf{VFS}(\mathsf{AFS_C}) \preceq \mathsf{POSIX}$ then ensures that they are also runs of POSIX.

As it turns out, for an arbitrary file f the only critical case is when a crash occurs during the execution of **cache_fsync** for this file. In all other cases, updates to the content of f have been stored in cache only, thus the persistent content of f in $\mathsf{AFS_P}$ is unchanged since the last successful execution of **cache_fsync** for f. So if the crash is outside of **cache_fsync**, we can choose a $\mathsf{VFS}(\mathsf{AFS_C})$ run in which all unsynchronized writes and truncates to f have failed and hence have not written or deleted any data. Constructing such runs is always possible as $\mathsf{AFS_C}$ is *crash-neutral*, i.e. all operations of $\mathsf{AFS_C}$ are specified to have a run that fails without any changes to the state (see Fig. 5 and [13]). In terms of histories, all operations in $(h_0|_f)_{>sync}$ are replaced by failed prefix operations.

However, showing that *WPCC* holds for crashes during **cache_fsync** is hard. The remainder of the section shows why caching of asynchronous truncations is one major difficulty when proving Theorem 1 (see Sec. 6.1), the approach for constructing write-prefix histories when crashes occur during synchronization (see Sec. 6.2), and how we formally proved in KIV that such write-prefix histories always yield a valid write-prefix run (see Sec. 6.3).

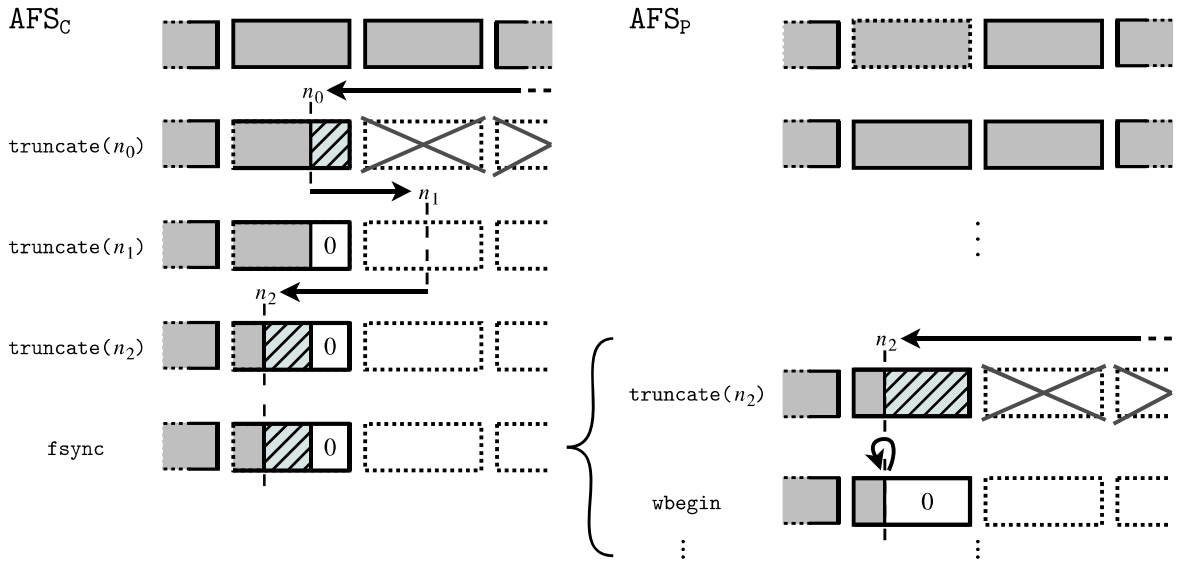


Fig. 15. Effect of a sequence of **afs_truncate** operations and a following **afs_fsync** on the states of one file in $\mathsf{AFS_C}$ (left) and $\mathsf{AFS_P}$ (right), including intermediate states of $\mathsf{AFS_P}$ during `fsync`. The state of `Cache` is omitted.

6.1 Truncations and Caching

Initially, our goal was to prove this property locally on the level of $\mathsf{AFS_C}$ resp. of `Cache` and $\mathsf{AFS_P}$ only. For example, one approach was to construct matching prefix runs of $\mathsf{AFS_C}$ by commuting and merging operation calls. While we will not go into the many pitfalls we ran into, the main problem with these approaches was the synchronization of aggregated truncates, as states resulting from an interrupted synchronization in `Cache` could not be reconstructed by any combination of `VFS` prefixes from the corresponding $\mathsf{AFS_C}$ run.

For example, given the sequence of three **afs_truncate** calls followed by an **afs_fsync** call as visualized in Fig. 15, starting with a synchronized file, i.e. the contents (and sizes) of the affected file are equal in $\mathsf{AFS_C}$ and $\mathsf{AFS_P}$. Considering this run in $\mathsf{AFS_C}$ on the left, the first truncation shrinks the file to a new size n_0 deleting all pages above $\mathsf{page}(n_0)$. Since $\mathtt{aligned}(n_0)$ is false, $\mathtt{rest}(n_0)$ bytes of junk data remain in $\mathsf{page}(n_0)$ for the moment. This junk data is removed not before the second truncation as it increases the file size then to n_1 and the remainder of $\mathsf{page}(n_0)$ is filled with zeros. Then finally, the third truncation shrinks the file again to n_2 with $n_2 < n_0$ but $\mathsf{page}(n_0) = \mathsf{page}(n_2)$, which yields a mixed page containing valid data, junk, and zeros.

These truncations do not have any effect on the persistent state of $\mathsf{AFS_P}$ as `Cache` handles all requests. Conversely, a call to **afs_fsync** in $\mathsf{AFS_C}$ leaves its state unchanged but its implementation `Cache` triggers a number of calls to $\mathsf{AFS_P}$. First, the file is truncated to n_2, the minimal truncation size since its last synchronized state. Second, junk data above n_2 is removed with **afs_wbegin** to prepare a potential synchronization of pages beyond n_2.

Comparing the state after **afs_wbegin** in $\mathsf{AFS_P}$ with the state after all truncations in $\mathsf{AFS_C}$, one can see that the sizes and the valid part of the content match but there is some junk data left in $\mathsf{AFS_C}$ that is not in $\mathsf{AFS_P}$. In fact, if a crash occurs in a state after this **afs_wbegin** call and before the synchronization of $\mathsf{page}(n_2)$ with **afs_wpage**, we cannot construct a `VFS` prefix run of $\mathsf{AFS_C}$ that yields exactly the state of $\mathsf{AFS_P}$. However, the abstraction from $\mathtt{VFS}(\mathsf{AFS_C})$ to `POSIX` ignores bytes written beyond the file size anyway and the implementation $\mathtt{Cache}(\mathsf{AFS_P})$ may at most remove more junk data than $\mathsf{AFS_C}$, so the implementation actually matches our crash-safety criterion

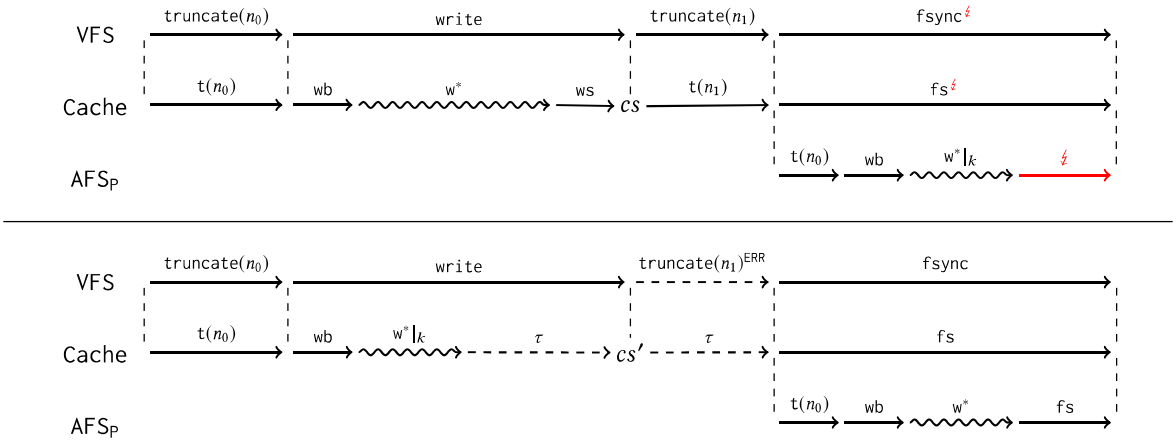


Fig. 16. Construction of a write-prefix run (lower half) matching a run with a crash ↯ in fsync that occurs just before writing page k (upper half).

under the POSIX abstraction as intended. But in order to prove this, we need to explicitly consider runs of AFS_C in the context of VFS.

6.2 Write-Prefix Histories for Crashed Synchronizations

To find a legal write-prefix history that satisfies Def. 2 for an arbitrary history that crashed during **cache_fsync**, one has to consider at which point the synchronization was interrupted. We will now omit arguments of operations and abbreviate the AFS/Cache operations **wbegin**, **wpage**, **wsize**, **truncate**, and **fsync** with wb, w, ws, fs, and t, respectively. Given the implementation of **cache_fsync** in Fig. 13, a typical run yields a AFS_P call sequence of the form t wb w^* ws fs. Because of the crash-atomicity of AFS_P, effectively two cases need to be addressed, namely a crash occurs

(1) between t and wb or
(2) between persisting pages $k-1$ and k with w.

Two additional cases are crashes before t or after ws (the fs call to AFS_P can be ignored for now, it only is relevant when low-level caches are used as discussed in Sec. 7.1). These can be viewed as crashing before resp. after the complete **cache_fsync** operation since no persistent changes happen in these ranges. We also do not explicitly consider crashes immediately after wb or before ws as separate cases, but instead we handle these as variants of case 2.

For case 1, finding a write-prefix history is quite obvious. As **cache_fsync** only executed a single persisting truncation to sz_T, only truncate operations to sz_T in $(h_0|_f)_{>sync}$ have been synchronized. Thus, only these operations remain unchanged in $(h_0'|_f)_{>sync}$, truncate operations to sizes n greater than sz_T are replaced by failed prefix operations. Similarly, all write operations are replaced by failed prefix operations as no pages have been persisted in **cache_fsync**.

Verifying case 2 requires more effort. As an example consider the crashed run shown in the upper half of Fig. 16. The run contains **vfs_truncate** and **vfs_write** calls, followed by an interrupted synchronization with **vfs_fsync** (denoted by the $\_^{↯}$ superscript in the figure). One can see that **vfs_truncate** triggers just a single t call to AFS_C resp. Cache, whereas **vfs_write** yields a call sequence of the form wb w^* ws (cf. Fig. 7). The **vfs_truncate** and **vfs_write** operations are performed in Cache only, so calls to AFS_P are performed not until synchronization.

The synchronization crashes after an ascending sequence $\mathtt{w}^*|_k$ of page writes, which contains only writes to pages $< k$.

A possible write-prefix run (and therefore a write-prefix history) is shon in the lower half of Fig. 16. As for case 1, the write-prefix run contains successful executions of **vfs_truncate** calls to the minimal truncate size. In the example, this is the size n_0, so the first truncation is performed as before. For the second truncation to n_1 on the other hand, we choose a failing run of **vfs_truncate** (failing operations are marked with $\_^{\mathsf{ERR}}$), which results in a stutter step τ in Cache, i.e. no operation is executed in Cache. So the first truncate operation is carried over unchanged to $(h_0'|_f)_{>sync}$ while the second truncate operation is replaced by a failed prefix operation.

The main aspect of *WPCC* is that non-synchronized **vfs_write** runs write just as far as the interrupted **vfs_fsync** was able to persist pages in the write-prefix run. Hence, the alternative **vfs_write** run successfully performs wb and a prefix of the original sequence $\mathtt{w}^*$, namely the prefix of writes $\mathtt{w}^*|_k$ to pages $< k$. All other writes to pages $\geq k$ are again replaced by stutter steps τ in Cache. When constructing the write-prefix history, a write operation with $I = inv(\mathtt{write}, f, pos, buf, n)$ and $R = res(\mathtt{write}, n', \mathtt{false})$ in $(h_0|_f)_{>sync}$ is replaced in $(h_0'|_f)_{>sync}$ by the prefix operation (I, R') where

$$R' \quad = \quad \begin{cases} R & \text{if} \quad pos + n < \mathtt{min}(\mathtt{pos}(k), sz) \\ res(\mathtt{write}, 0, \mathtt{true}) & \text{if} \quad \mathtt{min}(\mathtt{pos}(k), sz) \leq pos \\ res(\mathtt{write}, \mathtt{min}(\mathtt{pos}(k), sz) - pos, \mathtt{false}) & \text{otherwise} \end{cases}$$

Depending on the range the original **vfs_write** has written to, the restricted sequence $\mathtt{w}^*|_k$ may be empty or the full sequence $\mathtt{w}^*$. However, because the alternative run does not execute updates of the file size via ws, not all bytes written by $\mathtt{w}^*|_k$ become visible on the level of POSIX, but only bytes written below the persisted file size sz and below $\mathtt{pos}(k)$. Hence, the prefix operation (I, R') writes the same number of bytes as the original operation (first case), fails and writes no bytes at all (second case), or writes bytes up to $\mathtt{min}(\mathtt{pos}(k), sz)$ (third case).

With a complete write-prefix history constructed this way, a full, successful **vfs_fsync** run has the same effect as the crashed execution of the original run (except for differences in junk data resulting from the problematic nature of synchronizing truncations discussed in Sec. 6.1). Thus, the pending operation $I = inv(\mathtt{fsync}, f)$ can be completed with the response $R = res(\mathtt{fsync}, \mathtt{false})$.

6.3 From Write-Prefix Histories to Write-Prefix Runs

We now want to prove that write-prefix histories, as constructed in the previous section, are legal and that the corresponding write-prefix runs yield the same POSIX states as their original crashed runs. This is done with a forward simulation $\cong^k \subseteq CS \times CS$ over $\mathtt{Cache}(\mathtt{AFS}_\mathsf{P})$ states CS using commuting diagrams. The relation $\cong^k$ links all vertically aligned states in Fig. 16.

$$\cong^k \quad \equiv \quad ((files_\mathsf{P} \downarrow tcache) \oplus pcache|_k).\mathtt{seq}(ino) = (((files_\mathsf{P}' \downarrow tcache') \oplus pcache') \oplus icache').\mathtt{seq}(ino)$$

where $pcache|_k$ restricts $pcache$ to entries for pages $i < k$ and $files.\mathtt{seq}(ino)$ extracts the content of the file ino as a sequence of bytes up to the current size of ino in $files$. Intuitively, two Cache states cs and cs' are $cs \cong^k cs'$ if a synchronization interrupted at page k of cs yields the same content (up to the file size) as a complete synchronization of cs'. Note that $cs \cong^k cs'$ enforces implicitly that the file size of ino is identical in cs and cs' and hence the cached truncate sizes in $tcache$ and $tcache'$, as well as the cached size in $icache'$, must be equal.

For **wpage** calls the commuting diagrams as shown in Fig. 17 in the bottom plane are required. **wpage** operations of $\mathtt{AFS}_\mathsf{C}$ and $\mathtt{Cache}(\mathtt{AFS}_\mathsf{P})$ are denoted $\mathtt{w}_\mathsf{A}$ and $\mathtt{w}_\mathsf{C}$, respectively. When writing a page $< k$, re-executing this operation maintains $\cong^k$ (Fig. 17a). In contrast, writing pages $\geq k$ maintains $\cong^k$ if the alternative run stutters (Fig. 17b and Fig. 17c). Since VFS is defined on $\mathtt{AFS}_\mathsf{C}$, these commuting properties must be lifted from $\mathtt{Cache}(\mathtt{AFS}_\mathsf{P})$ to $\mathtt{AFS}_\mathsf{C}$ in order to construct commuting diagrams for $\mathtt{VFS}(\mathtt{AFS}_\mathsf{C})$ runs. This is why the commuting diagrams are extended by

R-corresponding $\mathsf{AFS_C}$ runs, yielding the front and back sides of Fig. 17. So in addition we show that, given a run $as_0 \xrightarrow{\mathtt{w_A}(i)} as_1$ as it is part of **vfs_write**, there is an R-corresponding run $cs_0 \xrightarrow{\mathtt{w_C}(i)} cs_1$ of $\mathsf{Cache}(\mathsf{AFS_P})$. Conversely, we have to show that the resulting alternative run of $\mathsf{Cache}(\mathsf{AFS_P})$ can be lifted to an R-corresponding run of $\mathsf{AFS_C}$ as well. Depending on the operation, up to two versions of this lifting are necessary if the run is stuttering: an $\mathsf{AFS_C}$ run that stutters (Fig. 17c) and a failing run of the $\mathsf{AFS_C}$ operation (Fig. 17b). For **wpage**, the former is used to skip writes of pages $> k$ while the latter is required to stop the loop of **vfs_write** when trying to write page k.

In KIV, these commuting diagrams have been proven using a sequent based weakest-precondition calculus. For example, for Fig. 17a we get the proof obligation

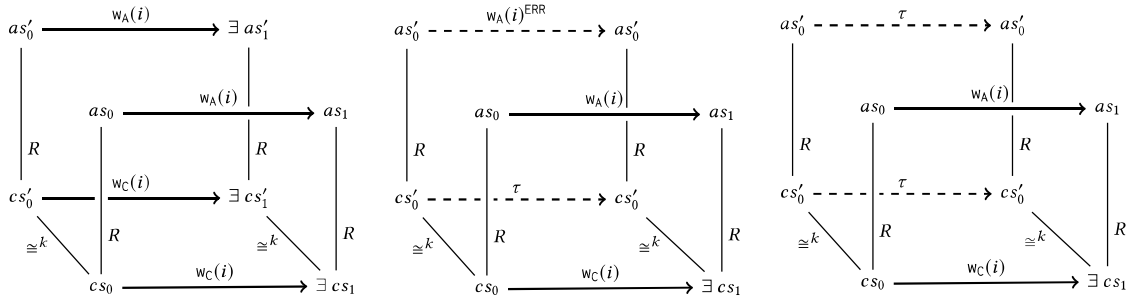
$$
\begin{aligned}
& \mathit{inv}_{\mathsf{A}}(as_0),\ \mathit{inv}_{\mathsf{C}}(cs_0),\ \mathit{inv}_{\mathsf{A}}(as_0'),\ \mathit{inv}_{\mathsf{C}}(cs_0'),\\
& R(as_0, cs_0),\ R(as_0', cs_0'),\ cs_0 \cong^k cs_0',\\
& \langle \textbf{afs\_wpage}(\mathit{inode}, \mathit{pno}, \mathit{pbuf}; as_0; \mathit{err})\rangle\ (\neg\ \mathit{err} \wedge as_1 = as_0),\\
& \mathit{pno} \leq k\\
& \vdash \exists\, cs_1, cs_1', as_1'.\\
& \qquad \langle \textbf{cache\_wpage}(\mathit{inode}, \mathit{pno}, \mathit{pbuf}; cs_0; \mathit{err})\rangle\ (\neg\ \mathit{err} \wedge cs_1 = cs_0)\\
& \quad \wedge \langle \textbf{cache\_wpage}(\mathit{inode}, \mathit{pno}, \mathit{pbuf}; cs_0'; \mathit{err})\rangle\ (\neg\ \mathit{err} \wedge cs_1' = cs_0')\\
& \quad \wedge \langle \textbf{afs\_wpage}(\mathit{inode}, \mathit{pno}, \mathit{pbuf}; as_0'; \mathit{err})\rangle\ (\neg\ \mathit{err} \wedge as_1' = as_0')\\
& \quad \wedge \mathit{inv}_{\mathsf{A}}(as_1) \wedge \mathit{inv}_{\mathsf{C}}(cs_1) \wedge \mathit{inv}_{\mathsf{A}}(as_1') \wedge \mathit{inv}_{\mathsf{C}}(cs_1')\\
& \quad \wedge R(as_1, cs_1) \wedge R(as_1', cs_1') \wedge cs_1 \cong^k cs_1'
\end{aligned}
$$

The formula $\langle\alpha\rangle\ \varphi$ is from Dynamic Logic [20] and expresses that the program α has a terminating run after which the formula φ holds. The weakest liberal precondition $\mathit{wlp}(\alpha, \varphi)$ is written as $[\alpha]\varphi$ in Dynamic Logic, which is equivalent to $\neg\ \langle\alpha\rangle\ \neg\ \varphi$. Note that while the declarations in our components (as shown in Sec. 3 and Sec. 4) do not have state variables as explicit parameters, our calculus requires that all non-local variables accessed in the body of a procedure are appropriate parameters. Hence, the state variables of a component (in the example, the vectors as_0, cs_0, as_0', or cs_0', respectively) are added automatically as reference parameters to the signature of its operations (as they can always read or update their component's state). Having the states as reference parameters also makes modifications to the states visible in postconditions of program formulas. For example, the equation $as_1 = as_0$ in the antecedent program formula fixes the $\mathsf{AFS_C}$ state reached by the **afs_wpage** run to as_1. This state can then be referenced in other formulas of the sequent, e.g. in the conjunct $R(as_1, cs_1)$ of the succedent formula.

In KIV, the primary technique for wp-calculus is symbolic execution, i.e. proofs execute the statements of a program sequentially, computing the strongest postcondition from the precondition in each step. [11] gives more details on KIV's calculus, proof automation support, and interaction possibilities using a graphical user interface.

To construct a valid alternative VFS run, analogous commuting diagrams for `wb`, `ws`, and `t` have been proven, not all commuting diagrams were necessary for each operation though. The proofs of commuting diagrams for **vfs_write** and **vfs_truncate** then are based upon the step by step application of these commutative properties.

Fig. 18 shows the construction of a **vfs_write** write-prefix run as an example. At the bottom there is the $\mathsf{Cache}(\mathsf{AFS_P})$ run cs_0 to cs_n resulting from the original **vfs_write** execution $\mathtt{write}_{\mathsf{VFS}}$. We proved for an arbitrary k (i.e. for every possible crash occurrence within a **cache_fsync** matching case 2 of Sec. 6.2) that there is a run $\mathtt{write}'_{\mathsf{VFS}}$ yielding $\mathsf{Cache}(\mathsf{AFS_P})$ states $cs_0'\ \ldots\ cs_{i+1}'$ which preserve $\cong^k$. Regardless of k, the initial `wb` call always needs to be executed successfully in $\mathtt{write}'_{\mathsf{VFS}}$ in order to get a valid run. So we use a commuting diagram analogous to Fig. 17a to get the states cs_1' and as_1'. Then we repeatedly append the commuting diagram for `w` of Fig. 17a for all writes to pages less than k. This yields the state cs_{i+1}' (and a R-corresponding as_{i+1}') for which



(a) Successful $\mathsf{AFS_C}$ run for $i < k$. (b) Failing $\mathsf{AFS_C}$ run for $i \geq k$. (c) Stuttering $\mathsf{AFS_C}$ run for $i \geq k$.

Fig. 17. Commuting diagrams of a w run writing page i.

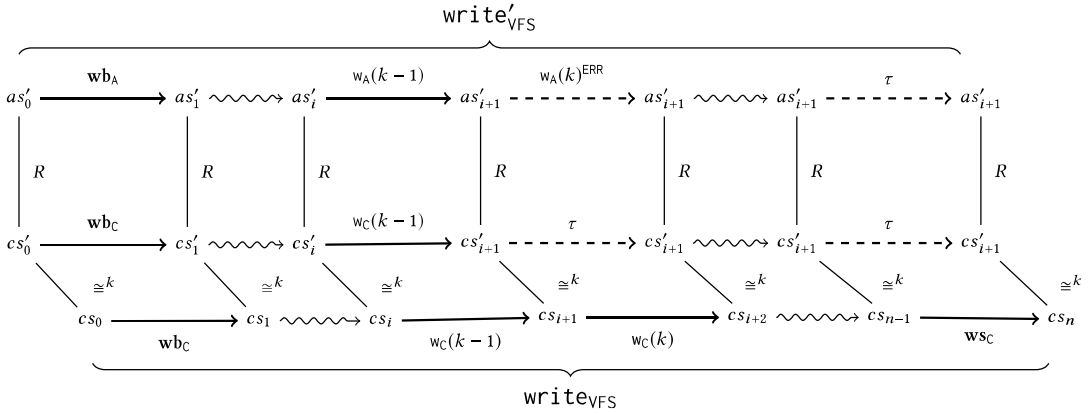


Fig. 18. Lifting commuting diagrams to VFS, exemplary for a **vfs_write** run (omitting the original $\mathsf{AFS_C}$ run for readability).

$cs_{i+1} \cong^k cs'_{i+1}$ obviously still holds, since so far the write-prefix run does not differ from the original one. To preserve $\cong^k$, neither a write to a page greater than or equal to k nor a size update may be performed in the write-prefix run, so $\mathtt{write}'_{\mathsf{VFS}}$ must stutter for further w and ws steps in $\mathtt{write}_{\mathsf{VFS}}$. To achieve a valid **vfs_write** run, the operation has to be aborted due to an error in w or ws resp. (see Fig. 7). Hence we append the commuting diagram of Fig. 17b to end the **vfs_wloop** and continue with repeating stuttering steps of the form of Fig. 17c until the state cs_{n-1} after **vfs_wloop** is reached. Finally, if the file size was increased in the original run, one last stuttering commuting diagram for **wsize** has to be applied. Depending on whether a page is written beyond the file size sz in the write-prefix run, this diagram is either of the form of Fig. 17c (if $\mathsf{pos}(k) \leq sz$ as shown in Fig. 18) or of Fig. 17b (if $sz < \mathsf{pos}(k)$). Note that state cs'_{i+1} of the write-prefix run after writing page $k-1$ remains unchanged as only stutter steps τ are taken and hence all cs_j with $i+1 \leq j \leq n$ satisfy $cs_j \cong^k cs'_{i+1}$.

For the construction of a write-prefix run for **vfs_truncate** only one modifying step needs to be considered, namely the call of t. As shown in Fig. 16 in the lower half, **vfs_truncate** calls may occur re-executed or failed in the write-prefix run, depending on whether they decrease the file size or they increase the file size. Hence, commuting diagrams analogous to Fig. 17a and Fig. 17b are necessary for $\mathtt{t_A}$ and $\mathtt{t_C}$. While re-executing only the minimal

truncation would be sufficient to get a complete write-prefix run, the forward simulation $\cong^k$ requires that the truncate sizes in *tcache* and *tcache'* are identical after each step. As a consequence, every update of the minimal truncate size in the original run (when $n < \mathtt{min}(sz_T, sz_F)$) must result in the same update in the write-prefix run. So for such **vfs_truncate** calls the successful run is chosen, for all others (where $n \geq \mathtt{min}(sz_T, sz_F)$) the failed run is chosen.

Considering the final states of the runs shown in Fig. 16, *tcache*, $pcache|_k$, *tcache'*, *pcache'*, and *icache'* do not contain any *dirty* data for *ino* and so applying them to $files_\mathsf{P}$ resp. $files_\mathsf{P}'$ does not have any effect. Consequently, in theses states $cs \cong^k cs'$ reduces to $files_\mathsf{P}.\mathtt{seq}(ino) = files_\mathsf{P}'.\mathtt{seq}(ino)$, which is exactly the property we wanted to achieve, namely identical POSIX states.

All in all, the verification of the crash-safety properties alone (not including earlier attempts) took about two months and comprises approx. 300 theorems. The proofs were significantly more complex and required about ten times more interactions than the proofs for functional correctness (only 79% of the proof steps could be automated compared to 98%). Most of the time was spent proving the commuting diagrams for $\cong^k$ on the level of $\mathtt{Cache}(\mathtt{AFS}_\mathsf{P})$ since many different cases have to be considered. Lifting these to $\mathtt{AFS}_\mathsf{C}$ could be done mainly by reusing the commuting diagrams for R together with some auxiliary lemmata over the $\mathtt{Cache}(\mathtt{AFS}_\mathsf{P})$ and AFS operations, which in turn enabled proving the commuting diagrams for $\mathtt{VFS}(\mathtt{AFS}_\mathsf{C})$ without major issues. For more details, the full proofs can be found online [28].

7 COMPATIBILITY WITH OTHER CACHES AND CONCURRENCY

This section discusses the integration of our crash-safety criterion with other concepts used in the refinement tower of Flashix. WPCC must be consistent with two crucial aspects: the *write buffer* and *concurrency*. The former is an instance of an order-preserving cache used on a low level of the implementation, the compatibility with WPCC is discussed in the first subsection. The latter is used in Flashix on lower levels as well as on top-level. On lower levels of the implementation, concurrency is used to perform *wear leveling* [36, 40] and *garbage collection* [3, 12] in separate threads so that ordinary operations (usually) do not have to wait for them. This is unproblematic, as both operations are invisible to higher levels of the refinement hierarchy. In refinement terminology, the algorithms of wear leveling and garbage collection are both parts of a concurrent implementation that linearizes to atomic operations of an abstract specification, where both algorithms linearize to the empty **skip**-operation. Concurrency on top-level, however, has effects on WPCC, which we discuss in the second subsection.

7.1 Order-Preserving Caches

Flash memory consists of blocks partitioned into pages (unrelated to the pages used in VFS). The pages of a block can only be written sequentially, and they can not be overwritten. Removing data is possible only by erasing an entire block.

Therefore, to avoid lots of half-filled pages, it is advisable to queue data written onto flash in a page-sized *write buffer*. Once the write-buffer becomes full, a filled page is written to flash memory. Like in the blueprint UBIFS, Flashix uses such a write buffer on a low level of the refinement hierarchy.

The write buffer is an instance of an order-preserving cache that queues data. It is generally possible to have several such caches on intermediate layers, and we have developed the crash-safety criterion of *Quasi-Sequential Crash Consistency (QSCC)* for such caches in earlier work [35].

The criterion requires two additional predicates for each specification: A predicate $\mathtt{Sync} \subseteq S$ which specifies synchronized states s, and a relation $\mathtt{Crash} \subseteq S \times S$ which specifies a residual effect on states. Additionally, a recovery operation Rec (one of the operations, for POSIX, this is just the mount operation) restores a consistent state after a crash.

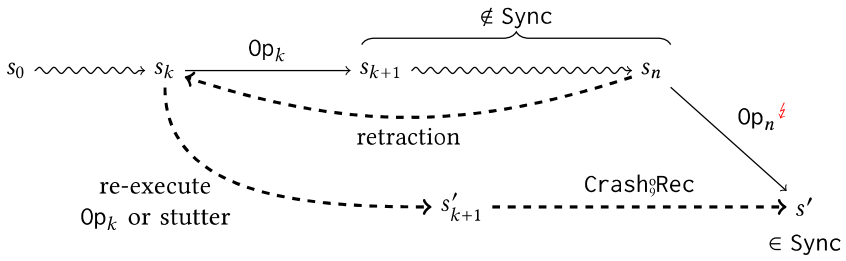


Fig. 19. Constructing a crash transition to s'.

Definition 3 (QSCC[35]). *An abstract data type* $(S, \mathtt{Init}, (\mathtt{Op}_i)_{i\in I}, \mathtt{Sync}, \mathtt{Crash})$ *is quasi sequential crash consistent (QSCC) iff for any sequence of executed operations, as shown in Fig. 19, a crash during the execution of* $\mathtt{Op}_n$ *will lead to a state* s' *that can be computed as follows:*

- *Some final operations* $\mathtt{Op}_k, \ldots, \mathtt{Op}_n$ *are retracted, where* $k \leq n$*. No operations are retracted that ended in synchronized states, i.e.* $s_{k+1}, \ldots, s_n \notin \mathtt{Sync}$
- $\mathtt{Op}_k$ *is executed with the original input in state* s_k*, resulting in a possibly different result state* s'_{k+1} *(and output) or* $\mathtt{Op}_k$ *is not executed at all (*$\mathtt{Op}_k$ *stutters), i.e.* $s'_{k+1} = s_k$*.*
- s' *is reached from* s'_{k+1} *by applying the crash relation and running the recovery operation:* $(s'_{k+1}, s') \in \mathtt{Crash}\fatsemi\mathtt{Rec}$ *must hold, where* $\fatsemi$ *composes the two relations. The final state* s' *must be synchronized to ensure that another crash cannot retract steps before the first one.*

The first two steps accommodate the loss of cached data in order-preserving caches. Intuitively, the modifications to the state done by operations $\mathtt{Op}_{k+1}, \ldots, \mathtt{Op}_n$ are still entirely in the cache, whereas the data of $\mathtt{Op}_k$ is partially in the cache. Re-execution of $\mathtt{Op}_k$ is relevant for POSIX writes, which may have written only an initial piece of their data.

Synchronized states are states with empty caches, so operations that lead to synchronized states will not be retracted. The criterion can be verified by additional proof obligations developed in [35] (in particular, executing $\mathtt{Crash}\fatsemi\mathtt{Rec}$ must preserve invariants specifying consistent states).

The first two points of QSCC correspond to buffered durable linearizability [25] when restricted to sequential runs: buffered durable linearizability allows the reduction of a (concurrent) history h consisting of invoke and response events to a prefix h' in which unfinished operations can be completed (linearized) anew. In our case, the history is sequential and consists of an alternating sequence of invoke and return events for the operations $\mathtt{Op}_1, \ldots, \mathtt{Op}_{k-1}$. There is then at most one pending operation $\mathtt{Op}_k$ that must be re-executed. Buffered durable linearizability requires that the final state reached with $h\lightning$ must then be a final state after executing the completion of h'. This would roughly correspond to the requirement that $s' = s'_{k+1}$ (the requirement is slightly weaker when operations are non-deterministic since it is not required that operations that have already been completed in h' have changed the state in the same way as in the original run).

For our context, this criterion is too strict, as crashes still have a (residual) observable effect that cannot be explained by just retracting and re-executing operations, as we will now explain. Therefore we allow the additional residual effect of Crash and Rec in the final step of the construction.

A residual effect is necessary for the POSIX specification already, since a crash closes all open files, which cannot be explained by retracting operations (retracting the open-operation would make all subsequent writes illegal. However, those which have already written their data permanently cannot be retracted). It could be

explained by adding closing operations for all open files, but this is not easier to specify than defining a `Crash` relation that deletes all open file handles.

Residual effects also are present on lower levels of the specification. This is due to *orphaned files*. Such orphans typically result from removing a file from the directory tree that is still open. The typical scenario is that of a running application that is replaced by a package update. The corresponding file will still be open for execution, but it has been replaced in the directory tree with a new version.

To ensure that orphaned files are deleted when the application is closed, the implementation of AFS_P has to remember which files are currently orphans. This set is first stored in RAM, and the effect of the `Crash` relation then is to reset this set to empty. The recovery operation then has to make sure that all current orphans are removed even without the current set. This is done by keeping information about orphans in the log. When the log is committed (written to flash memory), the current set of orphans is committed too. More details on these mechanisms are given in [12].

For compatibility with WPCC, it is relevant that AFS_P operations do not have different executions: re-executing an AFS_P operation will lead either to the same result or to an error without any state change, which is equivalent to not executing the operation at all. Therefore the effect of a crash is just to retract some of the final AFS_P operations. In the context of Def. 2, prefix-histories are not built based on the original eras h_i but on prefixes $h_i^{\sqsubseteq}$ of them. This means that for each file, some of its final AFS_P operations are retracted as well, and thus $h_i'|_f$ is also constructed based on a prefix $h_i^{\sqsubseteq}|_f$ of $h_i|_f$. For the crashed run of Fig. 16, the existence of the low-level cache will result in less persisting AFS_P operations being executed when a crash happens during **vfs_fsync**. However, this either is the same scenario as before (case 2 in Sec. 6.2 with $k' \leq k$ instead of k) or results in a simpler crash (case 1 in Sec. 6.2 or in a state before **vfs_fsync** started). Note that executing `fs` at the end of **cache_fsync** is crucial since it synchronizes the write buffer: As each successfull **vfs_fsync** operation yields a synchronized state $s \in \mathsf{Sync}$, a crash cannot retract AFS_P operations called before the response of **vfs_fsync**. Thus, $(h_i^{\sqsubseteq}|_f)_{\leq sync} = (h_i|_f)_{\leq sync}$ and $(h_i^{\sqsubseteq}|_f)_{>sync}$ simply contains shorter prefixes than $(h_i|_f)_{>sync}$, i.e. more failed operations and writes with fewer bytes written. This is entirely in line with Def. 2, so in summary, we have:

Theorem 2 (Compatibility of WPCC and QSCC). *WPCC is compatible with the use of order-preserving caches in lower levels of the implementation that satisfy QSCC.*

7.2 Top-Level Concurrency

Our top-level specification `POSIX` allows operations to be called concurrently. However, when the same file is written concurrently by several processes, only very weak guarantees can be given: since AFS_P operations are implemented atomically and preserve consistency, the resulting file system will still be consistent (even when crashes happen), but the resulting file content will be some random mix of old and new pages. Therefore our implementation ensures that `POSIX` writes to the same file are never done concurrently.

This can be specified at the level of AFS_C as the requirement that there must never be two concurrent, conflicting operations on the same file (one read and one write, or two writes). Formally, the concept of ownership defined in [40] is used but with files in place of blocks.

When allowing concurrent `POSIX` calls, runs of the file system yield well-formed concurrent histories. At first, these histories may contain arbitrary interleaved operations, particularly multiple content operations to the same file f. Such interleavings are problematic for WPCC because they would enable a **vfs_write** to "overtake" a **vfs_fsync** or vise versa (which could result in mixed file contents after a crash, even if it occurs outside of **vfs_fsync**). In order to prohibit such situations, the enhanced implementation uses a reader-writer lock per file set at the start of all content operations. As a consequence, all histories $h_i|_f$ are sequential, i.e. an interleaving of the form $inv_{tid}(\mathsf{op}, f, \ldots)$ $inv_{tid'}(\mathsf{op}', f, \ldots)$ $res_{tid}(\mathsf{op}, \ldots)$ $res_{tid'}(\mathsf{op}', \ldots)$ is not possible, and thus WPCC as defined in Def. 2 is still applicable.

However, ensuring exclusive access to files in VFS is not sufficient to avoid conflicts when accessing the cache. Currently, we use the simple solution of using a single mutex that ensures that calls to $\mathtt{AFS}_\mathtt{C}$ above the cache layer are sequentialized.

This has the disadvantage that all calls to **afs_wpage** in the **vfs_fsync** implementation can only run sequentially, which is still not optimally efficient. A more elaborate version, which will use a thread-safe cache and a background thread to flush the pages for a file, is concurrently work in progress. The improved implementation will ensure that $\mathtt{AFS}_\mathtt{P}$ operations are executed sequentially (there is not much efficiency to be gained by making their implementation concurrent as well), whereas Cache can be accessed concurrently.

8 RELATED WORK

File system correctness has been an active research topic for some time, starting with the earliest formal model of the POSIX standard written in Z by Morgan and Sufrin [33].

NASA's proposal to build a verifiable file system [26] has prompted a large body of related work, covering many aspects of file systems in general and also specific to flash memory. Mechanized models have been developed in [6, 8, 9, 16, 17, 22, 27].

There has also been work on specific issues, e.g. reading and writing files at byte-level has been addressed in [2, 27]. Model checking, which abstracts existing VFS code to enable proving specific properties, e.g. memory safety or the correct usage of locks [18, 34, 42], has also been used. A nice summary of this early work is [29].

While these approaches have made interesting contributions, they all considered specific aspects, specific layers of abstraction, or specific properties only. E.g. none of the approaches considers crash-safety or the separation of common functionality (VFS) and file system specific parts (AFS). By considering particular aspects only, none of them produced executable code.

Since then, apart from our approach, the only two other approaches that have generated running code we are aware of are BilbyFS [1] and DFSCQ [7].

BilbyFS is a file system for flash memory too. It implements caching mechanisms and gives a specification of the *sync* operation on the level of AFS, and proves its functional correctness. However, the verification of crash-safety or caching on the level of VFS we address in this paper is not considered.

Closest to our work is DFSQ, a sequentially implemented file system not targeted to work specifically with flash memory (and therefore much simpler than Flashix, the code size is ca. 4K). Similar to our approach, structural updates to the file system tree are persisted in order. DFSCQ also uses a page cache, however, it does not specify an order in which cached pages are written to persistent store. Therefore it is not provable that a crash leads to a POSIX-conforming alternate run. Instead, a weaker crash-safety criterion is satisfied, called *metadata-prefix* specification: it is proved that a consistent file system results from a crash, where some subset of the page writes has been executed.

In our context, the weaker criterion should be provable for any (functional correct) implementation of VFS caches, since we ensure that all $\mathtt{AFS}_\mathtt{P}$ operations are atomic (calls can never overlap) and the refinement proof of $\mathtt{VFS} \preceq \mathtt{POSIX}$ has lemmas for all AFS operations, that ensure that even these (and not just the VFS operations) preserve the abstraction relation to a consistent file system. Indeed, no stronger result is possible for the original VFS implementation in Linux, which offers many caching strategies, and does not even protect files from concurrent writes with mixed results.

For applications, when WPCC is not satisfied, the only strategy to achieve crash-safety is typically to write a new version of the whole file, call *fsync* on the new version, and atomically replace the old with the new version (using the *rename* operation). While this is efficient for small files, e.g. text files modified by an editor, it is inefficient for larger files.

When WPCC is satisfied, more efficient solutions become possible. When writes add data at the end of a file only, then a simple marker at the end of each write (and calling *fsync* to write the added data) is sufficient to ensure crash-safety. For general writes, a log-based strategy that adds modification entries with the modified data at the end of the file instead of a direct overwrite is sufficient. Both strategies would avoid copying entire files and would parallel the strategy of using a log and end-markers (for group nodes [12]) that is used in the implementation of the file system itself (below AFS) to ensure crash-atomicity of AFS operations.

9 CONCLUSION

So far, the Flashix project has been the largest project we have tackled to develop verified software. Rather than being a "mini challenge" (cf. the title of [26]), the project has been a hard challenge for developing techniques that allow verification of such a realistic system. Already making all the components of a modular, sequential implementation has been much harder than developing solutions for each relevant concept individually: all interfaces must seamlessly fit together and must consider all relevant concepts. For example, breaking up files into pages cannot be studied in a scenario where directory structure is ignored by using a flat set of files. Crash-safety, which has an enormous influence on the concepts used in any file system implementation, has been a cross-cutting concern from the beginning, and we have developed proof obligations for crash-safety together with the modularization concept [13].

In the second phase of the project, we have tackled the challenge of adding concurrency and caches. Any addition there affects several layers typically, and we have managed to "shift the refinement tower" by generating new proof obligations or new intermediate layers, avoiding the need to re-verify all components from scratch.

The concepts for adding concurrency have been detailed in other work [3, 40]. Adding caching to Flashix was an iterative process. Write-through caches have been used since the beginning as they are not problematic for crash-safety. We first developed a concept for order-preserving write-back caches in [35] that enables the usage of caches that queue data.

The page caches used in this paper have been more challenging to accommodate since standard criteria are not applicable, and the verification has to consider the top-level refinement $\mathsf{VFS}(\mathsf{AFS}_\mathsf{C}) \preceq \mathsf{POSIX}$ and the cache refinement $\mathsf{Cache}(\mathsf{AFS}_\mathsf{P}) \preceq \mathsf{AFS}_\mathsf{C}$ at the same time, which leads to rather complex proofs. We had initially hoped to confine the proof of crash-safety to the cache refinement, but confining the proof is impossible to reconcile with an efficient asymmetric implementation of the *truncate* operation, which implies that the AFS results that VFS sees are different when a cache is used compared to when no cache is used. That the states are different only modulo "junk data", which are irrelevant for POSIX, can only be verified by considering the POSIX to VFS refinement as well.

Nevertheless, this paper has shown how to integrate caching of file content as done by VFS into the modular development of a verified file system. We have defined the correctness criterion of *write-prefix crash consistency* for crash safety and have verified it with KIV, thus giving applications a formal criterion that can be used to verify that applications are crash-safe.

At its current stage, the Flashix file system now includes all necessary concepts for a realistic file system. There are still some inefficiencies due to using a code generator that must translate the simple value semantics of our programs (inherited from the semantics of predicate logic) to the pointer semantics of C that needs to allocate and copy data where necessary. The code generator can still be significantly optimized using data flow analysis to minimize copying.

For future work, there are also some technical improvements possible, like adding more fine-grained locking: Our VFS implementation currently locks the whole directory tree for traversal, a more fine-grained strategy we could adopt is the one analyzed for AtomFS [43].

Our current implementation empties a page cache that has grown too large by calling *fsync* in between operations called by the users. This blocks operations on the file until *fsync* has completed. Calling *fsync* could be moved to a background process, like it has been done for wear leveling and garbage collection. To imitate the behavior of Linux VFS, the crucial extension necessary there will be to allow *fsync* calls of this process to be interrupted when the user calls an operation on the file.

ACKNOWLEDGMENTS

We would like to thank the reviewers for their constructive feedback on preliminary versions of the paper. In particular this has now lead to a proper reformulation of WPCC in terms of histories. This work is supported by the Deutsche Forschungsgemeinschaft (DFG), "Verifikation von Flash-Dateisystemen" (grants RE828/13-1 and RE828/13-2).

REFERENCES

[1] S. Amani, A. Hixon, Z. Chen, C. Rizkallah, P. Chubb, L. O'Connor, J. Beeren, Y. Nagashima, J. Lim, T. Sewell, J. Tuong, G. Keller, T. Murray, G. Klein, and G. Heiser. 2016. Cogent: Verifying High-Assurance File System Implementations. In *Proc. of ASPLOS*. ACM, 175–188.

[2] K. Arkoudas, K. Zee, V. Kuncak, and M. C. Rinard. 2004. On Verifying a File System Implementation. In *ICFEM*. 373–390. https://doi.org/10.1007/978-3-540-30482-1_32

[3] S. Bodenmüller, G. Schellhorn, M. Bitterlich, and W. Reif. 2021. Flashix: Modular Verification of a Concurrent and Crash-Safe Flash File System. In *Festschrift dedicated to Egon Börger's 75th birthday (LNCS, Vol. 12750)*. Springer, 239 – 265.

[4] S. Bodenmüller, G. Schellhorn, and W. Reif. 2020. Modular Integration of Crashsafe Caching into a Verified Virtual File System Switch. In *Proc. of 16th International Conference on Integrated Formal Methods (IFM) (LNCS, Vol. 12546)*. Springer, 218 – 236.

[5] E. Börger and R. F. Stärk. 2003. *Abstract State Machines — A Method for High-Level System Design and Analysis*. Springer.

[6] A. Butterfield and J. Woodcock. 2007. Formalising Flash Memory: First Steps. In *Proc. of the 12th IEEE Int. Conf. on Engineering Complex Computer Systems (ICECCS)*. IEEE Comp. Soc., Washington DC, USA, 251–260. https://doi.org/10.1109/ICECCS.2007.23

[7] H. Chen, T. Chajed, A. Konradi, S. Wang, A. İleriy, A. Chlipala, M. Kaashoek, and N. Zeldovich. 2017. Verifying a High-Performance Crash-Safe File System Using a Tree Specification. In *Proc. of the 26th Symposium on Operating Systems Principles (SOSP)*. 270–286.

[8] K. Damchoom and M. Butler. 2009. Applying Event and Machine Decomposition to a Flash-Based Filestore in Event-B. In *Formal Methods: Foundations and Applications*, Marcel Vinícius Oliveira and Jim Woodcock (Eds.). Springer, Berlin, Heidelberg, 134–152. https://doi.org/10.1007/978-3-642-10452-7_10

[9] K. Damchoom, M. Butler, and J.-R. Abrial. 2008. Modelling and Proof of a Tree-Structured File System in Event-B and Rodin. In *Proc. of the 10th Int. Conf. on Formal Methods and Sw. Eng. (ICFEM)*. Springer LNCS 5256, 25–44. https://doi.org/10.1007/978-3-540-88194-0_5

[10] J. Derrick and E. Boiten. 2001. *Refinement in Z and in Object-Z : Foundations and Advanced Applications*. Springer. second, revised edition 2014.

[11] G. Ernst, J. Pfähler, G. Schellhorn, D. Haneberg, and W. Reif. 2015. KIV — Overview and VerifyThis competition. *Software Tools for Technology Transfer (STTT)* 17, 6 (2015), 677–694.

[12] G. Ernst, J. Pfähler, G. Schellhorn, and W. Reif. 2015. Inside a Verified Flash File System: Transactions & Garbage Collection. In *Proc. of Verified Software: Theories, Tools, Experiments (VSTTE) (LNCS, Vol. 9593)*. Springer, 73–93.

[13] G. Ernst, J. Pfähler, G. Schellhorn, and W. Reif. 2016. Modular, Crash-Safe Refinement for ASMs with Submachines. *Science of Computer Programming* 131 (2016), 3 – 21. Abstract State Machines, Alloy, B, TLA, VDM and Z (ABZ 2014).

[14] G. Ernst, G. Schellhorn, D. Haneberg, J. Pfähler, and W. Reif. 2012. A Formal Model of a Virtual Filesystem Switch. In *Proc. of Software and Systems Modeling (SSV) (EPTCS)*. 33–45.

[15] G. Ernst, G. Schellhorn, D. Haneberg, J. Pfähler, and W. Reif. 2013. Verification of a Virtual Filesystem Switch. In *Proc. of Verified Software: Theories, Tools, Experiments (VSTTE) (LNCS, Vol. 8164)*. Springer, 242–261.

[16] M.A. Ferreira, S.S. Silva, and J.N. Oliveira. 2008. Verifying Intel flash file system core specification. In *Modelling and Analysis in VDM: Proceedings of the Fourth VDM/Overture Workshop*. School of Computing Science, Newcastle University, 54–71. http://twiki.di.uminho.pt/twiki/pub/Research/VFS/WebHome/overture08sl.pdf Technical Report CS-TR-1099.

[17] L. Freitas, J. Woodcock, and Z. Fu. 2009. POSIX file store in Z/Eves: An experiment in the verified software repository. *Sci. of Comp. Programming* 74, 4 (2009), 238–257.

[18] A. Galloway, G. Lüttgen, J.T. Mühlberg, and R.I. Siminiceanu. 2009. Model-Checking the Linux Virtual File System. In *Proc. VMCAI'09*. Springer, 74–88. https://doi.org/10.1007/978-3-540-93900-9_10

[19] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. 1995. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley.

[20] D. Harel, J. Tiuryn, and D. Kozen. 2000. *Dynamic Logic*. MIT Press.

[21] M.P. Herlihy and J.M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12, 3 (1990), 463–492.
[22] W.H. Hesselink and M.I. Lali. 2012. Formalizing a hierarchical file system. *Form. Asp. Comput.* 24, 1 (Jan. 2012), 27–44. https://doi.org/10.1007/s00165-010-0171-2
[23] C.A.R. Hoare. 2003. The verifying compiler: A grand challenge for computing research. *J. ACM* 50, 1 (2003), 63–69.
[24] A. Hunter. 2008. A brief introduction to the design of UBIFS. (2008). http://www.linux-mtd.infradead.org/doc/ubifs_whitepaper.pdf
[25] J. Izraelevitz, H. Mendes, and M. Scott. 2016. Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model. In *Distributed Computing (LNCS, Vol. 9888)*. Springer, 313–327.
[26] R. Joshi and G.J. Holzmann. 2007. A mini challenge: build a verifiable filesystem. *Formal Aspects of Computing* 19, 2 (June 2007).
[27] E. Kang and D. Jackson. 2008. Formal Modelling and Analysis of a Flash Filesystem in Alloy. In *Proceedings of ABZ 2008*. Springer LNCS 5238, 294 – 308. https://doi.org/10.1007/978-3-540-87603-8_23
[28] KIV 2020. KIV models and proofs for VFS Caching. https://kiv.isse.de/projects/VFSCaching.html
[29] M. I. Lali. 2013. File system formalization: revisited. *International Journal of Advanced Computer Science* 3, 12 (2013), 602–606.
[30] N. Lynch and F. Vaandrager. 1995. Forward and Backward Simulations - Part I: Untimed Systems. *Information and Computation* 121, 2 (1995), 214–233.
[31] C. Manning. 2012. How Yaffs Works. (2012). https://yaffs.net/sites/yaffs.net/files/HowYaffsWorks.pdf
[32] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, and L. Vivier. 2007. The new ext 4 filesystem: current status and future plans. In *Proc. of the Linux Symposium*, Vol. 2.
[33] C. Morgan and B. Sufrin. 1987. Specification of the UNIX filing system. In *Specification case studies*. Prentice Hall Ltd., Hertfordshire, UK, 91–140.
[34] J.T. Mühlberg and G. Lüttgen. 2012. Verifying compiled file system code. *Formal Aspecpts of Computing* 24, 3 (2012), 375–391.
[35] J. Pfähler, G. Ernst, S. Bodenmüller, G. Schellhorn, and W. Reif. 2017. Modular Verification of Order-Preserving Write-Back Caches. In *IFM: 13th International Conference, 2017, Proceedings*. LNCS, Vol. 10510. Springer, 375–390.
[36] J. Pfähler, G. Ernst, G. Schellhorn, D. Haneberg, and W. Reif. 2013. Formal specification of an Erase Block Management Layer for Flash Memory. In *Haifa Verification Conference (HVC) (LNCS, Vol. 8244)*. Springer, 214–229.
[37] POSIX 2017. The Open Group Base Specifications Issue 7, IEEE Std 1003.1, 2018 Edition. The IEEE and The Open Group.
[38] G. Reeves and T. Neilson. 2005. The Mars Rover Spirit FLASH anomaly. In *Aerospace Conference*. IEEE Computer Society, 4186–4199.
[39] Reiser4 2011. Reiser4 file system for Linux OS. https://sourceforge.net/projects/reiser4 Accessed: 23.02.2022.
[40] G. Schellhorn, S. Bodenmüller, J. Pfähler, and W. Reif. 2020. Adding Concurrency to a Sequential Refinement Tower. In *Rigorous State-Based Methods (LNCS, Vol. 12071)*. Springer, 6–23.
[41] David Woodhouse. 2001. JFFS : The Journalling Flash File System. (2001). https://www.kernel.org/doc/mirror/ols2001/jffs2.pdf
[42] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. 2004. Using Model Checking to Find Serious File System Errors. In *Proc. of OSDI*. USENIX, 273–288.
[43] M. Zou, H. Ding, D. Du, M. Fu, R. Gu, and H. Chen. 2019. Using Concurrent Relational Logic with Helpers for Verifying the AtomFS File System. In *Proc. of SOSP (SOSP '19)*. ACM, 259–274.