Modular Integration of Crashsafe Caching into a Verified Virtual File System Switch *

Stefan Bodenmüller, Gerhard Schellhorn, and Wolfgang Reif

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

Abstract. When developing file systems, caching is a common technique to achieve a performant implementation. Integrating write-back caches into a file system does not only affect functional correctness but also impacts crash safety properties of the file system. As parts of written data are only stored in volatile memory, special care has to be taken when integrating write-back caches to guarantee that a power cut during running operations 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.

Keywords: POSIX-compliant File Systems, Write-Back Caching, Crash-Safety, Refinement, Interactive Verification

1 Introduction

This paper addresses the modular specification of a caching mechanism to a virtual filesystem switch (VFS) and the formal verification of crash-safety.

The original VFS is the standard top-level layer of any file system adhering to the POSIX standard [15] for all file systems used by Linux. Standard file systems like ext2,3,4 or ReiserFS use it, as well as file systems specific for raw flash memory, such as JFFS, YAFFS, or UBIFS.

In our Flashix project, we have developed a POSIX-compliant, modular file system for flash memory, using UBIFS as a blueprint, that was verified to be functionally correct and crash-safe. This includes a verified implementation of `VFS` without caching described in [8, 9]. The implementation is one of ten components of the verified development, which altogether generates approximately

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

18k of C-Code that can be run in the Linux kernel or via the FUSE interface. Initially, the implementation was sequential, in recent work we have developed a concept for adding concurrency to components [16], which has led to a concurrent implementation of wear leveling and garbage collection (both necessary for Flash memory). Allowing concurrent calls for the top-level POSIX operations is work in progress.

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

Our implementation `VFS` does not use a cache so far. However, since writing data to a cache in RAM is about two orders of magnitude more efficient than writing data to flash memory, caching is essential for efficiency: updating a file (e.g. editing a file with a text editor) several times will write the last version only when using a cache instead of persisting each update. It also reduces the need to read data from flash memory significantly.

We have addressed integrating caches into a verified file system before. Write-through caches are simple as they just store a redundant part of the persistent data in RAM. On a crash, nothing is lost, and an invariant stating that cached data are always identical to a part of the persistent data will suffice for verification. In [13] we have looked at order-preserving write-back caches that are used near the hardware level to queue data before persisting them in larger chunks. We have shown that these can be integrated into the hierarchy of components still allowing modular verification of each component separately.

Caching in `VFS` is rather different, since it is not order-preserving, so for the top-level POSIX operations, a new weaker crash safety criterion compared to [13] is necessary. 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 cleared the cache 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 [10] to add `VFS` caches as a single new component. Functional correctness then just requires to verify the new component separately. Crash-Safety however, 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 has a specific interface to the individual file systems that exploits it. This interface is called `AFS` (abstract file system) in this paper.

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 (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. with persisting the least recently used page first. Within these limitations, however, our result enables to write 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 as used by the file system itself [6].

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` machines that are relevant for manipulating file content. Section 4 then shows the extension, that adds caching to `VFS`.

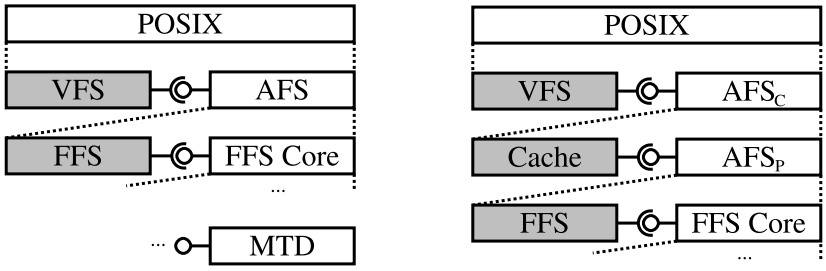
Section 5 defines the correctness criterion of write-prefix crash consistency and Section 6 gives some insight into its verification, that was done using our interactive verification system KIV [5]. We cannot fully go into the details of the proofs, which are very complex, the interested reader can find the full KIV proofs online [12]. Finally, Section 7 gives related work and concludes.

2 Formal Approach

The specification of the Flashix file system shown in Fig. 1a is organized into specifications of machines. An abstract state machine is an abstract data type, that consists of a state and some operations with inputs and outputs, that modify the state. Each operation is specified with a contract. Machines 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). The theory has been extended with proof obligations for crash safety, as detailed in [7].

To specify contracts uniformly, we prefer the style of abstract state machines (ASMs [2]) over using relational specifications as in Z [4]: 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 submachines (—©— in Fig. 1), which again are abstractly specified and then implemented as a separate component.

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. This 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. The `POSIX` interface is implemented by `VFS`, which uses a different data representation: Directory structure is now 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.



(a) Without caching layer. (b) With caching layer.

Fig. 1: Flashix refinement tower.

Files are now 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 ignore directory structure in the following: more information can be found in [8, 9]; the KIV specifications online [12] 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 [6, 14] to be crash-safe refinements according to the theory in [7, 16]. 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` ($\texttt{AFS}_\texttt{C}$ in the figure) that caches the data and calls operations of an identical copy of `AFS` (called $\texttt{AFS}_\texttt{P}$) 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` interface.

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

state $dirs : \mathsf{Ino} \nrightarrow \mathsf{Dir}$ $files : \mathsf{Ino} \nrightarrow \mathsf{File}$ where $\mathsf{Ino} \simeq \mathbb{N}$

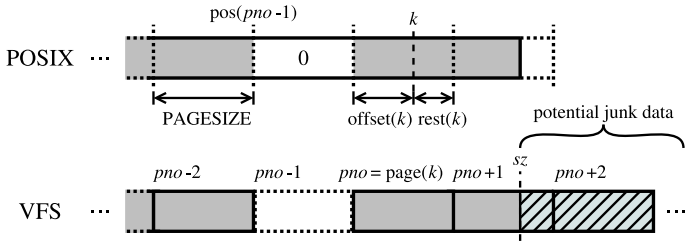


Fig. 2: Representation of file contents in `POSIX` and in `VFS`.

Since we are interested in adding write-back caches for file content, while directory structure only uses write-through caches, we just specify files

$$\textbf{data}\ \mathsf{File} = \mathtt{file}(\mathtt{meta} : \mathsf{Meta},\ \mathtt{size} : \mathbb{N},\ \mathtt{content} : \mathbb{N} \twoheadrightarrow \mathsf{Page})$$

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, metadata (access rights), and several pages. Each Page is an array of size `PAGESIZE`. Byte k of a file is accessed via $\mathtt{offset}(k)$ in $\mathtt{page}(k)$, 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 $\mathtt{pos}(pno) = pno * \mathtt{PAGESIZE}$. The pages are stored as a map, a missing page (e.g. page $pno - 1$ in the figure) indicates that the page contains zeros only. This sparse representation allows to create a file with a 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 $\mathtt{page}(sz)$ at the file size sz contains random junk data (hatched part of the page) above $\mathtt{offset}(sz)$ instead of just zeros. Extra (hatched) pages with a page number larger than $\mathtt{page}(sz)$ 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 as input and output (passed by reference). An inode has the form

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

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). `size` stores the file size for files, and the number of entries for a directory.

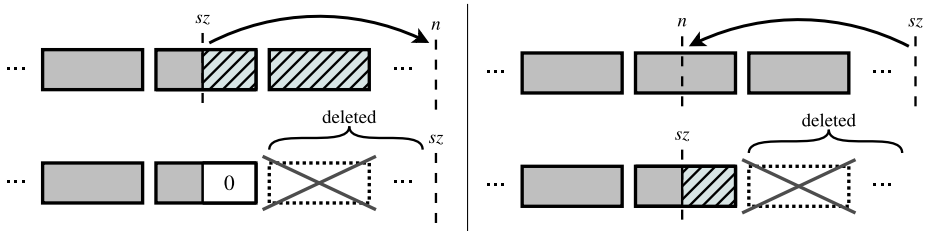
The relevant `AFS` operations for modifying file content are specified in Fig. 3. The operations use semicolons to separate input, in/out, and output parameters. We give a short description, which also gives some preconditions.

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

Fig. 3: File operations of AFS.

- **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 an inode *inode*, that points to a file. If the page does not exist, the buffer is set to all zeros (abbreviated as $\bot$), 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. 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 [13]) must empty this cache.
- **afs_truncate** is used to change the file size to n, checking that there are 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. 4. 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 filesize simply become junk, it does not need to



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

Fig. 4: Effects of a truncation to n on a file with size sz.

be modified. The implementation of the **afs_truncate** operation, therefore, avoids writing pages to persistent store whenever this is possible[1].
- **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.

All operations are allowed to non-deterministically (**or**) 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.

On the basis of the `AFS` operations, `VFS` implements two `POSIX` operations that modify file content, **vfs_truncate** and **vfs_write**. The first operation changes file size by just calling **afs_truncate**. Writing a buffer *buf* (an array of arbitrary size) of length n at position *pos* has the following steps:

- **afs_wbegin** is called first, to make sure that writing does not accidentally read junk data.
- Then the buffer is split at page boundaries, and individual pages are written by calling **afs_wpage**. Writing starts with the lowest page at `page`(pos) and proceeds upwards. If *pos* is not page-aligned, the first write requires to read the original page first by calling **afs_rpage** and to merge the original content below `offset`(pos) with the initial piece of *buf* of length `rest`(pos). Merging is necessary too for the last page when $pos + n$ is not aligned.
- Writing pages stops as soon as the first call to **afs_wpage** returns an error. If this is the case, the number n is decremented to the number of bytes actually written.
- Finally, if $pos + n$ is larger than the old file size, **afs_wsize** is called, to modify the file size, and **vfs_write** returns the number n of bytes written.

We will see in the next section that when adding caches it is crucially important 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. 4) is one of the main difficulties for adding caches correctly.

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

4 Integration of Caches into Flashix

Initially, Flashix was developed without having caches for high-level data structures in mind. 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. 5. This layer is implemented as a *Decorator* [10], i.e. it implements the same interface as the `FFS` and delegates calls from the `VFS` to the `FFS`. The `VFS` communicates with a Cache Controller which in turn communicates with the `FFS` and manages caches for inodes, pages, and an auxiliary cache for truncations.

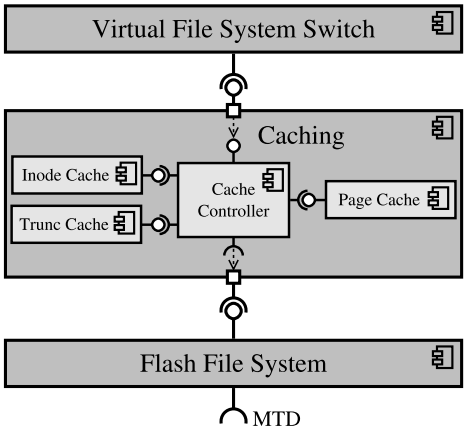


Fig. 5: Flashix component hierarchy.

The Inode- and Page-Cache components internally store maps from unique identifiers to the corresponding data structures. They all offer interfaces to the Cache Controller for adding resp. updating, reading, and deleting cache entries. The Cache Controller 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.

Similar to the Linux VFS, the caching layer includes further caches for data structures forming the basic structure of the file system. These caches however only operate in write-through mode to speed up read accesses. Otherwise, the integrity of the file system tree after a crash could be compromised since *structural operations* are usually highly dependent on one another and affect multiple data objects.

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. 6 lists the central operations of the `PCache` component using the state $pcache\colon \mathsf{Ino} \times \mathsf{Nat} \pfun \mathsf{Bool} \times \mathsf{Page}$. Analogously the component `ICache` is defined. It stores a mapping $icache\colon \mathsf{Ino} \pfun \mathsf{Bool} \times \mathsf{Inode}$ from inode numbers to entries

```
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. 7: `Cache` operations for reading and writing pages and updating file sizes.

containing the inode itself and a dirty flag.

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. 7 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. 7 on the left). But reading from flash yields the correct result only if there was no prior truncation that would have deleted the relevant page, i.e. an entry for this file exists in `TCache` and applying this truncation would delete the requested page (if $\texttt{min}(sz_T, sz_F) \leq \texttt{pos}(pno)$, i.e. *pno* is beyond the cached truncate size sz_T and 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.

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

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

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

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

Fig. 6: Core of the `PCache` component.

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. 8 on the left. 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 purpose, the two subcomponents provide dedicated truncation operations **pcache_truncate** resp. **tcache_update**. **tcache_update** aggregates multiple truncations by caching the minimal truncate size *n* for each file only. Additionally, the persisted size *sz* of

```
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);
}
```

```
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);
}

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. 8: File truncation (left) and synchronization (right) operations of `Cache`.

a file is stored in `TCache` to determine whether it is allowed to read a page from flash in **cache_rpage**. 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`.

The synchronization of files, i.e. transferring cached updates to the persistent storage, is also coordinated by `Cache`. Clients can use the `POSIX` *fsync* operation to trigger synchronization of a specific file. It is common practice that cached data is also synchronized concurrently, however, this is left for future work.

The implementation of *fsync* in `Cache` is shown in Fig. 8 on the right. 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 only in PCache or dirty inodes in ICache 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_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**). Finally, **cache_finode** updates the file size with **afs_wsize** if the cached size is greater than the persisted size sz_F.

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 $\mathtt{Cache}(\mathtt{AFS_P}) \sqsubseteq \mathtt{AFS_C}$ only. The proofs are done with a forward simulation $R \subseteq AS \times CS$ using commuting diagrams with states $AS \equiv \mathit{dirs}_{\mathtt{C}} \times \mathit{files}_{\mathtt{C}}$ of $\mathtt{AFS_C}$ and $CS \equiv \mathit{dirs}_{\mathtt{P}} \times \mathit{files}_{\mathtt{P}} \times \mathit{icache} \times \mathit{pcache} \times \mathit{tcache}$ of $\mathtt{Cache}(\mathtt{AFS_P})$.

$$R \quad \equiv \quad \mathit{dirs}_{\mathtt{C}} = \mathit{dirs}_{\mathtt{P}} \wedge \mathit{files}_{\mathtt{C}} = ((\mathit{files}_{\mathtt{P}} \downarrow \mathit{tcache}) \oplus \mathit{pcache}) \oplus \mathit{icache}$$

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 ($\_ \downarrow \mathit{tcache}$), overwriting all pages with their cached contents ($\_ \oplus \mathit{pcache}$), and updating the cached file sizes ($\_ \oplus \mathit{icache}$). As no structural operations are cached, $\mathit{dirs}_{\mathtt{C}}$ and $\mathit{dirs}_{\mathtt{P}}$ are identical.

While $\mathtt{AFS_C}$ functionally matches the original specification of AFS, it is easy to see that $\mathtt{AFS_C}$ differs quite heavily from $\mathtt{AFS_P}$ in terms of its crash behavior. A crash in $\mathtt{AFS_P}$, for example, has the effect of removing orphaned files [7], 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 $\mathtt{AFS_C}$ 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* $\lightning \subseteq S \times S$ 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 $\texttt{AFS}_\texttt{C}$ 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 $\texttt{AFS}_\texttt{C}$. If such an alternative run can always be found, crash-safety holds since all regular (non-crashing) runs of $\texttt{AFS}_\texttt{C}$ yield consistent states, and thus a crash results in a consistent state as well.

Definition 1 (Write-Prefix Crash Consistency). *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.

To prove that Flashix satisfies *WPCC* we need to show that for each possible occurrence of a crash in $\texttt{Cache}(\texttt{AFS}_\texttt{P})$ we can construct a matching alternative run for each file in $\texttt{AFS}_\texttt{C}$ and lift this to runs in $\texttt{VFS}(\texttt{AFS}_\texttt{C})$. 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 $\texttt{AFS}_\texttt{P}$ is unchanged since the last successful execution of **cache_fsync** for f. So we can choose a $\texttt{VFS}(\texttt{AFS}_\texttt{C})$ run in which all *writes* and *truncates* to f have failed and hence have not written or deleted any data. Constructing such a run is always possible as $\texttt{AFS}_\texttt{C}$ is *crash-neutral*, i.e. all operations of $\texttt{AFS}_\texttt{C}$ are specified to have a run that fails without any changes to the state (see Fig. 3 and [7]).

However, showing that *WPCC* holds for crashes during **cache_fsync** is hard. Initially, our goal was to prove this property locally on the level of $\texttt{AFS}_\texttt{C}$ resp. of Cache and $\texttt{AFS}_\texttt{P}$ only. For example, one approach was to construct matching prefix runs of $\texttt{AFS}_\texttt{C}$ by commuting and merging of operation calls. While we will not go into details of the many pitfalls we ran into, the main problem with these approaches was that the synchronization of aggregated *truncates*, as states resulting from prefix runs of **cache_fsync** could not be reconstructed by any combination of VFS prefixes from the corresponding $\texttt{AFS}_\texttt{C}$ run.

For example, given the sequence of three **afs_truncate** calls followed by an **afs_fsync** call as visualized in Fig. 9, starting with a synchronized file, i.e. the contents (and sizes) of the affected file are equal in $\texttt{AFS}_\texttt{C}$ and $\texttt{AFS}_\texttt{P}$. Considering

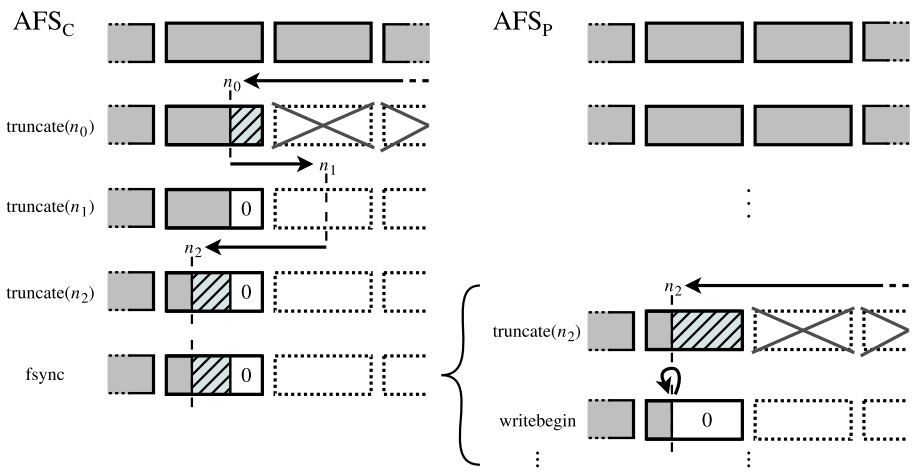


Fig. 9: Effect of a sequence of *truncate* operations and a following *fsync* on the states of one file in $\texttt{AFS}_\texttt{C}$ (left) and $\texttt{AFS}_\texttt{P}$ (right), including intermediate states of $\texttt{AFS}_\texttt{P}$ during *fsync*. The state of `Cache` is ommitted.

this run in $\texttt{AFS}_\texttt{C}$ on the left, the first truncation shrinks the file to a new size n_0 deleting all pages above $\texttt{page}(n_0)$. Since $\texttt{aligned}(n_0)$ is false, $\texttt{rest}(n_0)$ bytes of junk data remain in $\texttt{page}(n_0)$ 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 $\texttt{page}(n_0)$ is filled with zeros. When finally the third truncation shrinks the file again to n_2 with $n_2 < n_0$ but $\texttt{page}(n_0) = \texttt{page}(n_2)$, which yields a mixed page containing valid data, junk, and zeros.

These truncations do not have any effect on the persistent state of $\texttt{AFS}_\texttt{P}$ as `Cache` handles all requests. Conversely, a call to **afs_fsync** in $\texttt{AFS}_\texttt{C}$ leaves its state unchanged but its implementation `Cache` triggers a number of calls to $\texttt{AFS}_\texttt{P}$. 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 $\texttt{AFS}_\texttt{P}$ with the state after all truncations in $\texttt{AFS}_\texttt{C}$, one can see that the sizes and the valid part of the content match but there is some junk data left in $\texttt{AFS}_\texttt{C}$ that is not in $\texttt{AFS}_\texttt{P}$. In fact, if a crash occurs in a state after this **afs_wbegin** call and before the synchronization of $\texttt{page}(n_2)$ with **afs_wpage**, we cannot construct a `VFS` prefix run of $\texttt{AFS}_\texttt{C}$ that yields the state of $\texttt{AFS}_\texttt{P}$. Fortunately, the abstraction from $\texttt{VFS}(\texttt{AFS}_\texttt{C})$ to `POSIX` ignores bytes written beyond the file size anyway and the implementation $\texttt{Cache}(\texttt{AFS}_\texttt{P})$ may at most remove more junk data than $\texttt{AFS}_\texttt{C}$, so the implementation actually matches our crash-safety criterion under the `POSIX` abstraction as intended. But in order to prove this, we need to explicitly consider runs of $\texttt{AFS}_\texttt{C}$ in the context of `VFS`.

In the following section, we give a brief overview of the concrete proof strategy we pursued to construct such a *write-prefix* run.

6 Proving Crash-Safety

The main effort for proving that Flashix is actually *write-prefix crash consistent* was to show that a crash during **cache_fsync** actually has the effect of write-prefix runs of VFS. Given the implementation of **cache_fsync** and the fact that the operations of $\texttt{AFS}_\texttt{P}$ are atomic with respect to crashes, effectively two cases need to be considered, namely a crash occurs

1. between **afs_truncate** and **afs_wbegin** or
2. between persisting pages $k-1$ and k with **afs_wpage**.

Two additional cases are crashes before **afs_truncate** or after **afs_wsize**. These can be viewed as crashing before resp. after the complete **cache_fsync** operation since no persistent changes happen in these ranges. Note that we do not explicitly consider crashes immediately after **afs_wbegin** or before **afs_wsize** as separate cases, but instead we handle these as variants of case 2.

For case 1 finding a write-prefix run is quite obvious. As **cache_fsync** only executed a single persisting truncation to sz_T, only **vfs_truncate** calls to sz_T were successful in the alternative VFS run as well. For **vfs_truncate** calls to sizes n greater than sz_T the run is chosen in which **afs_truncate** fails, so we get a failing run of **vfs_truncate** too. For **vfs_write** calls the run is chosen in which the initial **afs_wbegin** fails which results in not calling any further $\texttt{AFS}_\texttt{C}$ operations (cf. Sec. 3).

Verifying case 2 requires more effort. As an example consider the crashed run shown in the upper half of Fig. 10 . We omit irrelevant arguments of operations and abbreviate **wbegin**, **wpage**, **wsize**, and **truncate** with **wb**, **w**, **ws**, and **t**, respectively. The run contains **vfs_truncate** and **vfs_write** calls, followed by an interrupted synchronization **fsync**↯. The former operations are performed in Cache only, so calls to $\texttt{AFS}_\texttt{P}$ are performed not until synchronization. **fsync** crashes after an ascending sequence $\mathbf{w}^*|_k$ of **wpage** operations, which contains only writes to pages $< k$. As for case 1, the write-prefix run we construct in the lower half of Fig. 10 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 operation runs are marked with $\_^{\texttt{ERR}}$), which results in a stutter step τ in Cache, i.e. no operation is executed in Cache.

The main aspect of *WPCC* is that alternative **vfs_write** runs write just as far as the interrupted **fsync** was able to persist pages. Hence, the alternative run successfully performs **wbegin** and a prefix of the original **wpage** sequence $\mathbf{w}^*$, namely the prefix of writes $\mathbf{w}^*|_k$ to pages $< k$. All other writes to pages $\geq k$ are again replaced by stutter steps τ in Cache. Depending on the range the original **vfs_write** has written to, the restricted sequence $\mathbf{w}^*|_k$ may be empty or the full sequence $\mathbf{w}^*$. However, the alternative run will not execute (stutter) for updates of the file size via **wsize**. With a complete system run constructed this way, a full **fsync** run has the same effect as the crashed execution if the original run (except for differences in junk data resulting from the problematic nature

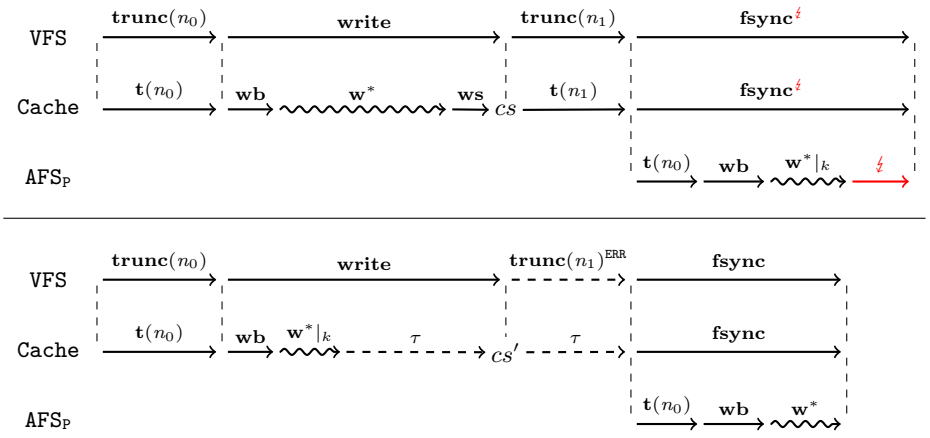


Fig. 10: Construction of a write-prefix run (lower half) matching a run with a crash in **cache_fsync** that occurs just before writing page k (upper half).

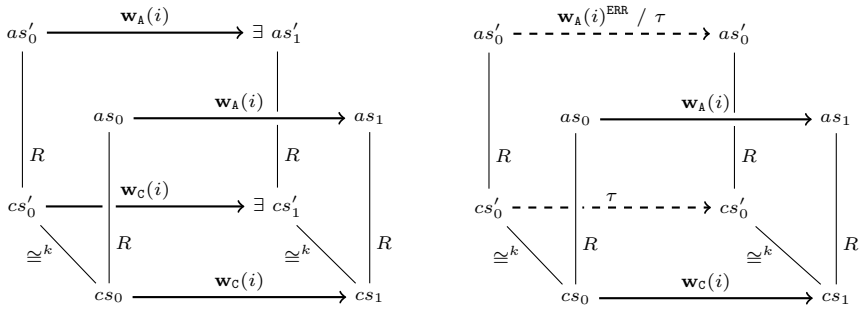
of synchronizing truncations discussed in Sec. 5), and thus the alternative run finishes in a synchronized state.

Proving that the runs constructed this way match the original crashed runs is done with a forward simulation $\cong^k \subseteq CS \times CS$ using commuting diagrams. Relation $\cong^k$ links all vertically aligned states in Fig. 10.

$$\begin{aligned} \cong^k \quad \equiv \quad & ((\mathit{files}_{\mathtt{P}} \downarrow \mathit{tcache}) \oplus \mathit{pcache}|_k).\mathtt{seq}(\mathit{ino}) \\ & = (((\mathit{files}_{\mathtt{P}}{}' \downarrow \mathit{tcache}') \oplus \mathit{pcache}') \oplus \mathit{icache}').\mathtt{seq}(\mathit{ino}) \end{aligned}$$

where $\mathit{pcache}|_k$ restricts pcache to entries for pages $i < k$ and $\mathit{files}.\mathtt{seq}(\mathtt{Ino})$ 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 the **wpage** calls the commuting diagrams as shown in Fig. 11 in the bottom plane are required. **wpage** operations of `AFS` and `Cache` are denoted $\mathbf{w}_{\mathtt{A}}$ and $\mathbf{w}_{\mathtt{C}}$, respectively. When writing a page $< k$, re-executing this operation maintains $\cong^k$ (Fig. 11a). In contrast, writing pages $\geq k$ maintains $\cong^k$ if the alternative run stutters (Fig. 11b). Since `VFS` is defined on $\mathtt{AFS}_{\mathtt{C}}$, these commuting properties must be lifted from `Cache` to $\mathtt{AFS}_{\mathtt{C}}$ in order to construct commuting diagrams for `VFS` runs. This is why the commuting diagrams are extended by R-corresponding $\mathtt{AFS}_{\mathtt{C}}$ runs, yielding the front and back sides of Fig. 11. So in addition we show that, given a run $as_0 \xrightarrow{\mathbf{w}_{\mathtt{A}}(i)} as_1$ as it is part of **vfs_write** or **vfs_truncate**, there is an R-corresponding run of `Cache`. Conversely, we have to show that the resulting alternative run of `Cache` can be lifted to an R-corresponding run of $\mathtt{AFS}_{\mathtt{C}}$ as well. Depending on the operation, up to two versions of this lifting are



(a) Successful $\mathtt{AFS_C}$ run for $i < k$. (b) Failing/stuttering $\mathtt{AFS_C}$ run for $i \geq k$.

Fig. 11: Commuting diagrams of a **wpage** run writing page i.

necessary if the run is stuttering: an $\mathtt{AFS_C}$ run that stutters and a failing run of the $\mathtt{AFS_C}$ operation. 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.

To construct a valid alternative `VFS` run, analogous commuting diagrams for **wbegin**, **wsize**, and **truncate** 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 base upon the step by step application of these commutative properties. Considering the final states of the runs shown in Fig. 10, *tcache*, $pcache|_k$, *tcache*$'$, *pcache*$'$, and *icache*$'$ do not contain any *dirty* data for *ino* and so applying them to $files_{\mathtt{P}}$ resp. $files_{\mathtt{P}}{}'$ does not have any effect. Consequently, in theses states $cs \cong^k cs'$ reduces to $files_{\mathtt{P}}.\mathtt{seq}(ino) = files_{\mathtt{P}}{}'.\mathtt{seq}(ino)$, which is exactly the property we wanted to achieve.

All in all, the verification of the crash-safety properties alone (not including earlier attempts) took about two months and comprises approx. 300 theorems. Most of the time was spent proving the commuting diagrams for $\cong^k$ on the level of `Cache`, since many different cases have to be considered. Lifting these to $\mathtt{AFS_C}$ could be done mainly by reusing the commuting diagrams for R together with some auxiliary lemmata over the `Cache` and `AFS` operations, which in turn enabled proving the commuting diagrams for `VFS` without major issues. For more details, the full proofs can be found online [12].

7 Related Work and Conclusion

In this paper we have 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.

For reasons of space we could not formally address how caching of `VFS` interacts with the order-preserving cache (called "write buffer" in [13]) as used by

lower levels of the implementation. However, informally the answer is as follows. $\texttt{AFS}_\texttt{P}$ operations are implemented atomically, i.e. a page is either persisted as a whole or not at all. This is necessary to imply linearizability of $\texttt{AFS}_\texttt{P}$ operations in a concurrent context. Removing the data of the write buffer on a crash has the effect of undoing some $\texttt{AFS}_\texttt{P}$ operations according to [7]. Therefore, discarding the write buffer has the same effect as crashing slightly earlier, and thus *WPCC* still holds.

We have also not discussed concurrent top-level calls of `POSIX` operations. We have augmented the specification to allow this, and are working on a verification using the approach given in [16], which has already been used to allow concurrent garbage collection. For the theory presented here to work, the implementation ensures that modifications to each file (write, truncate, fsync) are done sequentially only.

We have discussed lots of related work on verified file systems in general in earlier work [13, 16], here we discuss two related approaches, that generate running code and have addressed the correctness of write-back caching in file systems. These are BilbyFS [1] and DFSCQ [3]. 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 the functional correctness of this operation. However, the verification of crash-safety or caching on the level of `VFS` is not considered.

DFSCQ is a sequentially implemented file system not targeted to work specifically with flash memory. 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 $\texttt{AFS}_\texttt{P}$ operations are atomic (calls can never overlap) and the refinement proof of $\texttt{VFS} \sqsubseteq \texttt{POSIX}$ 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.

Our earlier crash-safety criterion for order-preserving caches can be viewed as the sequential case of buffered durable linearizability [11], which allows to undo a postfix of the history of invokes and responses for operations, thus allowing pending operations of the resulting prefix to have a new result. The criterion is also sufficient to specify the $\texttt{AFS}_\texttt{C}$ interface in a concurrent context (since the operations are linearizable). However, it is stronger than the criterion given here, as it does not allow to re-execute several sequentially executed operations (only one can be pending in the prefix).

Future work on the file system will be to add a background process that calls fsync to empty caches when no user operations are executed. 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.

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. Cogent: Verifying high-assurance file system implementations. In *Proc. of ASPLOS*, page 175–188. ACM, 2016.
2. E. Börger and R. F. Stärk. *Abstract State Machines — A Method for High-Level System Design and Analysis*. Springer, 2003.
3. H. Chen, T. Chajed, A. Konradi, S. Wang, A. İleriy, A. Chlipala, M. Kaashoek, and N. Zeldovich. Verifying a high-performance crash-safe file system using a tree specification. In *Proc. of the 26th Symposium on Operating Systems Principles (SOSP)*, pages 270–286, 2017.
4. J. Derrick and E. Boiten. *Refinement in Z and in Object-Z : Foundations and Advanced Applications*. FACIT. Springer, 2001. second, revised edition 2014.
5. G. Ernst, J. Pfähler, G. Schellhorn, D. Haneberg, and W. Reif. KIV — Overview and VerifyThis competition. *Software Tools for Technology Transfer (STTT)*, 17(6):677–694, 2015.
6. G. Ernst, J. Pfähler, G. Schellhorn, and W. Reif. Inside a Verified Flash File System: Transactions & Garbage Collection. In *Proc. of Verified Software: Theories, Tools, Experiments (VSTTE)*, volume 9593 of *LNCS*, pages 73–93. Springer, 2015.
7. G. Ernst, J. Pfähler, G. Schellhorn, and W. Reif. Modular, Crash-Safe Refinement for ASMs with Submachines. *Science of Computer Programming*, 131:3 – 21, 2016. Abstract State Machines, Alloy, B, TLA, VDM and Z (ABZ 2014).
8. G. Ernst, G. Schellhorn, D. Haneberg, J. Pfähler, and W. Reif. A Formal Model of a Virtual Filesystem Switch. In *Proc. of Software and Systems Modeling (SSV)*, EPTCS, pages 33–45, 2012.
9. G. Ernst, G. Schellhorn, D. Haneberg, J. Pfähler, and W. Reif. Verification of a Virtual Filesystem Switch. In *Proc. of Verified Software: Theories, Tools, Experiments (VSTTE)*, volume 8164 of *LNCS*, pages 242–261. Springer, 2013.
10. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley, 1995.
11. J. Izraelevitz, H. Mendes, and M. Scott. Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model. In *Distributed Computing*, volume 9888 of *LNCS*, pages 313–327. Springer, 2016.
12. KIV models and proofs for VFS Caching, 2020. URL: `https://kiv.isse.de/projects/VFSCaching.html`.
13. J. Pfähler, G. Ernst, S. Bodenmüller, G. Schellhorn, and W. Reif. Modular Verification of Order-Preserving Write-Back Caches. In *IFM: 13th International Conference, 2017, Proceedings*, volume 10510 of *LNCS*, pages 375–390. Springer, 2017.
14. J. Pfähler, G. Ernst, G. Schellhorn, D. Haneberg, and W. Reif. Formal specification of an Erase Block Management Layer for Flash Memory. In *Haifa Verification Conference (HVC)*, volume 8244 of *LNCS*, pages 214–229. Springer, 2013.
15. The Open Group Base Specifications Issue 7, IEEE Std 1003.1, 2018 Edition. The IEEE and The Open Group, 2017.
16. G. Schellhorn, S. Bodenmüller, J. Pfähler, and W. Reif. Adding Concurrency to a Sequential Refinement Tower. In *Rigorous State-Based Methods*, volume 12071 of *LNCS*, pages 6–23. Springer, 2020.