Flashix: modular verification of a concurrent and crash-safe flash file system

Stefan Bodenmüller, Gerhard Schellhorn, Martin Bitterlich, Wolfgang Reif

Angaben zur Veröffentlichung / Publication details:

Bodenmüller, Stefan, Gerhard Schellhorn, Martin Bitterlich, and Wolfgang Reif. 2021. "Flashix: modular verification of a concurrent and crash-safe flash file system." *Lecture Notes in Computer Science* 12750: 239–65. https://doi.org/10.1007/978-3-030-76020-5_14.

Flashix: Modular Verification of a Concurrent and Crash-Safe Flash File System*

Stefan Bodenmüller, Gerhard Schellhorn, Martin Bitterlich, and Wolfgang Reif

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

Abstract. The Flashix project has developed the first realistic verified file system for Flash memory. This paper gives an overview over the project and the theory used. Specification is based on modular components and subcomponents, which may have concurrent implementations connected via refinement. Functional correctness and crash-safety of each component is verified separately. We highlight some components that were recently added to improve efficiency, such as file caches and concurrent garbage collection. The project generates 18K of C code that runs under Linux. We evaluate how efficiency has improved and compare to UBIFS, the most recent flash file system implementation available for the Linux kernel.

1 Introduction

Modular software development based on refinement has always been a core concern of our Formal Methods group.

One of the constant positive and inspiring influences on our work has always been Prof. Börger's research on the formalism of Abstract State Machines (ASMs) [7].

The earliest starting point of this has been the Prolog Compiler specified in [6] that describes compilation to the Warren Abstract Machine as stepwise ASM refinement. Mechanized verification of this refinement tower was posed by him as a challenge in a DFG priority program. Our solution to this case study led to a formalization of the ASM refinement theory [5] proposed there [33] and later on to a completeness proof [34]. Using this theory we managed to do a mechanized verification of the compiler [35] using our theorem prover KIV [15]. The work led to the PhD of one of the authors of this paper [32], with Prof. Börger being one of the reviewers.

It was also Prof. Börger who pointed us to the Mondex challenge [10, 41], which consists of mechanizing the proofs of a security protocol for Mondex electronic purses. Among other groups [25] we verified the original protocol [21],

* Partly supported by the Deutsche Forschungsgemeinschaft (DFG), “Verifikation von Flash-Dateisystemen” (grants RE828/13-1 and RE828/13-2).

but also proposed an improvement that would avoid a weakness. We extended the case study to the development of a suitable cryptographic protocol and to the verification of a Java implementation [19]. The Java calculus [40] we used in KIV was influenced by the semantics of Prof. Börger's book [39]. The work also influenced our work on the development of a systematic development of security protocols using UML specifications [20].

Since then, we have tackled our most ambitious case study: Development of a fully verified, realistic file system for flash memory, called Flashix.

In a first phase of the project we had to develop the necessary theory that allowed to manage the complexity of such an undertaking. A concept of components and subcomponents was developed that are connected by refinement. This allowed modular software development as well as modular verification of proof obligations for each component. Together the proofs of this refinement tower guarantee functional correctness as well as crash-safety of the resulting implementation. An overview was given in [17]. The generated code from this first phase is a sequential implementation that can be run in Linux.

This paper gives an overview of the second phase of the project, where we tackled aspects crucial for efficiency: we enhanced the theory with a concept that allows to add concurrency incrementally to a refinement tower. We now also allow caches for files, which lead to a new crash-safety criterion called *write-prefix crash consistency*. We summarize the concepts and the theory in Section 2 and give an overview over the structure of Flashix in Section 3.

We then highlight two of the new features of Flashix. File content is now cached as described in Section 4, and write-prefix crash consistency has been proved [4]. Like wear leveling (described in [36]) garbage collection is now concurrent (Section 5).

The specifications of the Flashix file system implementation uses a language of abstract programs similar to the rules of Turbo-ASMs [7], although the concept for concurrency is based on interleaving [37]. We generate Scala- as well as C-Code from such abstract programs. Currently the generated C-Code has 18k loc, which can be used in Linux either with the Fuse library [42] or integrated in the kernel.

Section 6 evaluates the performance of our implementation. Since the concepts of UBIFS (the newest implementation of a file system in the Linux kernel) served as a blue-print for the concepts we used and verified in Flashix, we also compare efficiency to UBIFS.

Finally Section 7 gives related work, and Section 8 concludes the paper.

2 Methodology

This section gives an informal summary of the methodology. It consists of three core concepts that together establish that the top-level specification of POSIX is correctly implemented having crash-safe and linearizable behavior. The three concepts detailed in the following subsections are

– State-based components with specifications of operations. These are refined to implementations which are components, too. The implementations may call operations of other *subcomponent* specifications.
– Refinement from specifications comes in two flavors: (a form of) *data refinement* that allows to exchange abstract data structures (e.g. a set) with more efficient, concrete ones (e.g. a red-black tree) and *atomicity refinement*, which replaces an atomic operation with the non-atomic steps of a program, which allows concurrent execution.
– A concept for verifying crash-safety.

2.1 Components

A component is similar to an ASM. We distinguish between specification and implementation components, although they are specified using the same syntax.

A component specifies a number of state variables that store values of data types like numbers, lists, sets, arrays, maps, or tuples. Data types themselves are axiomatized using simply-typed lambda calculus. Most axioms use many-sorted first-order logic only, but there are exceptions like infinite sequences which use function variables (which represent dynamic functions as used in ASMs).

The operations of a component are given by a precondition, inputs and outputs, together with an imperative program that modifies the state. Programs contain assignments (function updates are possible), conditionals, loops and recursion. Using non-deterministic **choose** is allowed in specifications. Thereby an arbitrary postcondition can be established, simply by choosing a state that satisfies the predicate. Implementations allow specific versions only for which executable code can be generated. Two common cases are the allocation of a new reference for a heap-based data structure and choosing an element from a set.

There are two distinguished operations: Initialization, which computes initial states, and recovery, which is called after a crash when remounting to re-initialize the state.

Specification components are used to describe parts of the file system in an abstract and simple way, mainly by specifying functionality algebraically. Implementation components, on the other hand, implement functionality programmatically using low-level data structures.

For example, in Flashix we use a specification component to access a set of ordered elements. The component provides interface operations to add or delete an element. Another operation returns the greatest element below some given threshold. The precondition of this operation requires the set to be non-empty. The programs for these operations typically consist of a single assignment as the functionality is axiomatized over algebraic sets.

The corresponding implementation component gives an efficient realization of the interface using a red-black tree defined as a heap-based pointer structure. The separation into specification and implementation components allows to generate high-performance code from implementations while client components do not have to deal with their complexity but can rely on their abstract specification instead.

The semantics of a specification component is always that of a data type as in data refinement: it is a set of traces (also called histories). A trace is a sequence of matching pairs $[inv(in_1, op_1),\ ret(op_1, out_1),\ \ldots,\ inv(in_n, op_n),\ ret(op_n, out_n)]$ of *invoke* events $inv(in_i, op_i)$ and *return* events $ret(op_i, out_i)$. The first corresponds to invoking the operation op_i with input in_i, the second to the call returning with output out_i. Such a trace corresponds to a client sequentially calling operations $op_1, \ldots, op_n$ that execute atomically. Note that we immediately use a pair instead of a single call event to simplify the description of concurrency and crash-safety. The trace is observable (i.e. an element of the semantics) if there is a suitable sequence of states $[s_0, \ldots, s_n]$ (the run of the ASM) which is hidden from the client. State s_0 must be initial, and if calling operation op_i in state s_{i-1} with input in_i has a valid precondition, then it must have an execution that terminates and leads to state s_i with output out_i. Since the client is responsible for calling an operation only, when its precondition is satisfied, observations after calling an operation with violated precondition are arbitrary: the called operation should behave chaotically, yielding an arbitrary (even an illegal) successor state or none at all by non-termination.

Implementation components differ from specification components in two aspects. First, they may call operations from one or more subcomponents. Second, their semantics can be either *atomic* or *interleaved*. In the first case, its semantics is the same as the one of a specification component. In the latter case the semantics of the implementation are interleaved runs of the programs. The semantics then is similar to a control-state ASM, where the control-state is encoded implicitly in the program structure. To accommodate the fact, that we can have an arbitrary number of threads (or agents in ASMs), the events in traces are now generalized (as in [22]) to consist of matching pairs of $inv_t(in_i, op_i)$ and $ret_t(op_i, out_i)$ events that are indexed with a thread t. Matching pairs of different threads may now overlap in a history, corresponding to interleaved runs of the programs. The atomic steps of executing an operation are now: first, the invoking step (where the invocation event is observed), then the execution of individual instructions of the program like assignments, conditionals, and subcomponent calls, and finally a returning step (if the program terminates). The execution of operations of several threads are now interleaved. Formally, a concurrent history is legal if the projection to events of each thread t gives a sequence of matching pairs, possibly ending with an invoke event of a still running (pending) operation.

2.2 Refinement

Refinement of a specification component to an implementation component with atomic semantics is done using the contract approach to data refinement (see [13] for proof obligations in Z and the history of the approach) adapted to our operational specification of contracts, with proof obligations in wp-calculus, as detailed in [17].

All our refinements in the first phase of the Flashix project were such sequential refinements. Adding concurrency by replacing an atomic implementation with a thread-safe implementation with interleaved runs typically requires to add locks and ownership concepts, together with information which lock protects which data structure. Details of this process of 'shifting the refinement tower' are described in [36].

In the second phase of the Flashix project we have now added concurrency in all three places, where this is useful: The implementation of the top-level POSIX specification is now thread-safe, as mentioned in Section 3. Wear leveling and garbage collection (see Section 5) are now executed concurrent to regular POSIX operations called by the user of the file system.

Proving an implementation with interleaved semantics to be correct can be done in one step, proving linearizability [22] (progress aspects such as termination and deadlock-freedom must additionally be proved).

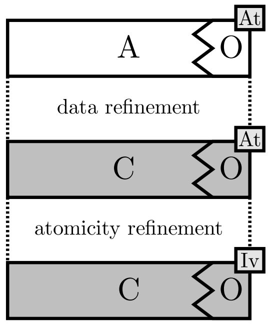


Fig. 1: Refinement to a concurrent implementation.

Our approach uses two steps, using the implementation with atomic semantics (denoted by $\boxed{\text{At}}$) as an intermediate level, see Figure 1. This allows to have an upper refinement that (ideally) is the same as before and can be reused. In practice it is often necessary to add auxiliary operations that acquire/release ownership to specifications, as indicated by the **O** in the figure. These additional operations do not generate code, but they ensure that the client of the specification (the machine that uses this machine as a submachine) will not use arbitrary concurrent calls, but only ones that adhere to a certain discipline, as detailed in [36]. This leads to additional proof obligations for the refinement as well as for the client, which must call acquire/release to signal adherence to the discipline.

The lower *atomicity refinement* shows that the interleaved implementation (denoted by $\boxed{\text{Iv}}$) behaves exactly as if the whole code of the implementation would be executed atomically at some point in between the invoking step and the returning step. This point is usually called the linearization point. Correct atomicity refinement (and linearizability in general) can be expressed as reordering the events of the concurrent history H to a sequential history S (i.e. a sequence of matching pairs that do not interleave) that is correct in terms of the atomic semantics. The (total) order of matching pairs in the sequential history is determined from the order of linearization points. It preserves the partial order (called the *real-time order*) of operations in the concurrent history. If an operation is pending in the concurrent history, the corresponding sequential history may be *completed* in two ways: either the invoke event can be dropped, when the operation has not linearized, or a matching return can be added, when it has.

The proof method for proving atomicity refinement uses Lipton's theory of commuting operations [26] and borrows ideas from [14].

The proof has two phases, which may be alternated. In the first phase, we verify that specific assertions hold at all program points using a rely-guarantee approach. Thos proof also guarantees termination and deadlock freedom with a calculus similar to the one in [43]. Essentially, the steps must satisfy two conditions: First, assertions before and after a step must satisfy the usual conditions of Hoare's calculus for total correctness. Second, all assertions must be stable with respect to a rely condition that is proven to abstract the steps of other threads.

The second phase is to iteratively show that two sequential steps of one thread can be combined into a single atomic step. This is done by showing that the first step commutes (leaves the same final state, if it is a returning step it must also produce the same output) with every step of another thread to the right (the step is a *right mover*), or dually that the second step is a *left mover.*

We found that this proof technique is suitable for locking-based algorithms, where locking/unlocking instructions are simple cases of right/left movers. If a data structure is written in a section of the code, where the thread holds a suitable lock, then the operation is both a left as well as a right mover (a *both mover*).

Note that the assertions proven for program points play an essential role in proving such commutativity properties, since they are often incompatible, resulting in trivial commutativity. Usually writing a data structure does not commute with writing it in another thread. But if it is asserted that the updating thread holds a lock that protects it, then they trivially commute, since the two assertions that the lock is held by both threads contradict each other.

Combining steps into larger steps can be iterated. It typically leads to the innermost locking range to be contracted to one atomic instruction. Repeating the first phase, we can now prove that the lock is free at all times, which again allows new instructions to become left or right movers in phase two. Alternating phases ends, when all instructions of the program have been combined into a single, atomic step.

The approach so far guarantees *functional correctness.* For our instance this says that our concurrent implementation of the POSIX standard (which combines the code from all implementation components) has the same behavior (in the sense of linearizability [22]) as atomically executing POSIX operations. In particular, all operations terminate and there are no deadlocks.

To prove that this is the case, we have to show that refinement is compatible with the use of subcomponents: If C refines A, then implementation M that calls operations of A, is refined by machine M, calling their implementation from C, as shown in Figure 2.

This is a folklore theorem ("substitutivity") that should hold for all meaningful definitions of refinement. For data refinement we are aware of the formal proof in [12], for linearizability it is informally stated in [22]. We have not given a detailed proof yet, the sequential case (including crash-safety, see below) is

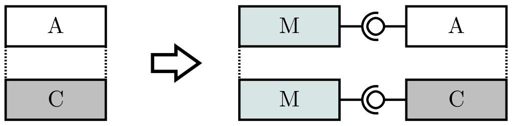


Fig. 2: Substitutivity of Refinement.

proven in [17]. For the concurrent case we recently found an elegant proof in terms of combining IO-Automata [3], though this proof does not yet take into account non-terminating behavior of operations. Like for other refinement definitions (see e.g. [13] for data refinement or [34] for ASM refinement) the proof would have to be lifted to a scenario where states include a bottom element that represents non-termination.

2.3 Crash Safety

In addition to functional correctness, crash-safety is the second important aspect for a file system to work correctly. Informally it guarantees that when a crash happens (typically a power failure), the file system can be rebooted to a state that is consistent and does not contain any surprises: files and directories that have not been modified before the crash still should keep their content. Files where a write operation was running should not have modified content outside the range that was overwritten, and data within the range should be either old or new data, but nothing else.

A first observation relevant for crash-safety is that the only persistent state of the file system that is left unchanged by a crash is flash memory, which is the state of the lowest-level MTD interface. All other state variables are state, stored in RAM, that is deleted by a crash. Meaningful values for these states are constructed by running the recovery operations of all implementation components bottom-up.

A second, crucial observation is that if a running operation (on any level of the hierarchy) is aborted in the middle due to a crash, the resulting state can also be reached by crashing in a state after the operation has completed. The reason is that the flash hardware can always (nondeterministically) refuse any writes due to being worn out. Therefore, the alternative run that leads to the same state as the one with the crash is the one where all flash modifications fail after the point of the crash in the original one and the crash happens at the end. Proving this can be reduced to the simple proof obligation (expressible in wp-calculus) that all implementations of operations have a run such that running crash and recovery in the initial and final state yields the same state.

As a consequence, the question whether crashes at any time are guaranteed to lead to a consistent state can be reduced to the question whether crashes in between operations lead to a consistent state. Again, the latter gives a simple proof obligation in wp-calculus for the recovery program.

However, this does not specify how the final state looks in comparison to the final state of the original run, so we still might see an unexpected result.

A simple idea to specify the effect of a crash would be to specify a (total) *Crash* relation between states before and after the crash. However, this becomes intractable for higher levels of the refinement tower, due to the use of write-back caches. Such caches are used in all file system implementations for efficiency, since RAM access is significantly faster than writing or reading from persistent memory. In our flash file system such a cache, called the *write buffer*, is even necessary, since a page can only be written as a whole, and can not be overwritten. The write buffer therefore collects data until full page size is reached before writing the page to flash memory. The write buffer follows a queue discipline, it persists data first that was received first. We call such a buffer *order-preserving* and allow arbitrary use of such order-preserving caches on any level of an implementation.

The use of a cache makes it difficult to just specify a crash relation, since on higher level specifications, the information which part of the data is still in cache is no longer present. After all, the top-level POSIX specification specifies a directory tree and file contents with no information which parts are still cached. In principle, such information can be added as auxiliary data (used for verification only, deleted in the running code), but we found such an encoding to become intractable.

Instead we specify the effect of crashes mainly in an *operational* style, where the effect of a crash is to construct an alternative run that explains the resulting state after the crash. This alternative run mainly retracts a final piece of the original run with the intuition that the results of the retracted operations are still in the cache. We found that this is compatible with order-preserving caches, where losing the content of a queue corresponds to losing the effects of some of the final operations.

In addition to undoing part of the run it is however necessary that operations (one in a sequential setting, several in concurrent implementations) that are running at the time of the crash may be executed with a different result: when writing some bytes to a file crashes, an initial piece of the data may have persisted while the remaining bytes have not. Constructing an alternative run that writes fewer bytes is consistent with POSIX: A top-level write is allowed to return having only written a prefix of the data to the file (the number of written bytes is returned). The alternative run will therefore have a different completion of the write operation with fewer bytes written.

Therefore, in addition to undoing a final part of the run, we allow all running operations (which have a pending invoke in the shortened history) to be completed differently in the replacement run.

Two more considerations are necessary to ensure that crashes do not give surprising results. First, POSIX offers a *sync* operation that empties all cached data. This leads to synchronized states, which we specify on all levels of the refinement hierarchy. Operations that lead to a synchronized state are then forbidden to be retracted in the alternative run.

Finally, we still need a crash relation on all levels to specify an additional residual effect of the crash on RAM state. For the top-level of POSIX this is obvious, since even when no operation is running, a crash will at least close all open files and remove the resulting orphaned files (files that were removed in the directory tree, but are still open).

In summary, the effect of a crash after a run that went through states $(s_1, \ldots, s_n)$ will be a (consistent) final state s' of an alternative run, which executes an initial piece of the original run, say up to s_i, then completes operations that are running at this point to reach a state s''. Finally, the crash relation is applied and the recovery program is executed to reach s'.

The proof obligations resulting from this concept were formally verified to imply this crash-safety criterion for a sequential setting in [17]. However, it is applicable without changes in a concurrent setting, too. Note that it is again crucial that all operations that run at the point where a crash happens have an alternative run without any more changes to the persistent flash memory. Thus, when proving linearizability by reordering steps according to Lipton's theory, we can already consider a run with completed operations to show that an equivalent sequential execution exists where all programs execute atomically.

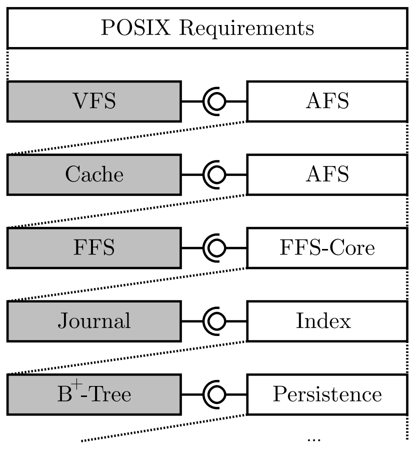
The theory given here must be extended when caching the data of individual files is considered as retracting a part of the run is no longer sufficient. We consider an appropriate extension in in Section 4.

3 The Flashix File System – Overview

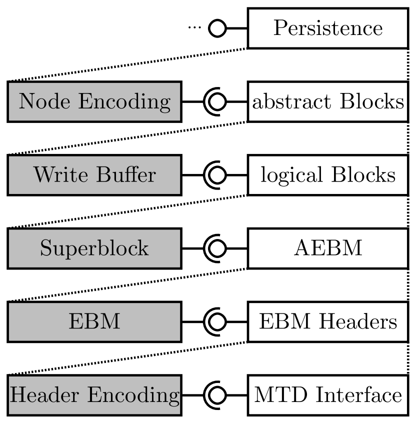
The Flashix file system is structured into a deep hierarchy of incremental refinements as shown in Fig. 3. Boxes represent formal models that can be connected via refinements (dashed lines) and can call operations of their subcomponents through a well-defined interface (—⊙—). We distinguish between specification components in white and implementation components in gray. Combining all implementation components then results in the final implementation of the file system.

The top layer of Fig. 3a is a formal specification of the POSIX standard [31]. It defines the interface and the functional correctness requirements of the file system. Here, the state of the file system is given by a directory tree where leaves store file identifiers, and a mapping of file identifiers to the corresponding file contents, represented by a sequence of bytes. An indirection between file identifiers and file content is necessary to allow hard links, where the same file is present in several directories. *Structural operations*, i.e. operations that modify the directory tree like creating/deleting directories or (un)linking files, are defined on paths. *Content operations*, such as reading or writing parts of the content of a file, work directly on file identifiers.

The bottom layer of the hierarchy in Fig. 3b is a formal specification of the Linux MTD Interface (Memory Technology Devices). It acts as a lower boundary of the file system and provides low-level operations to erase flash blocks and to read and write single pages within flash blocks. Preconditions ensure that calls



(a) Upper layers of Flashix.



(b) Lower layers of Flashix.

Fig. 3: Component hierarchy of the Flashix file system.

to these operations comply with the characteristics of flash memory, i.e. that pages are only written as a whole and that pages are only written sequentially within a block. Additionally, it formalizes assumptions about hardware failures or the behavior of the flash device in the event of a crash.

In a first refinement step, the POSIX model is refined by a Virtual Filesystem Switch (VFS) that uses an abstract specification of the core file system (AFS). Similar to the Linux Virtual Filesystem, the VFS component implements the resolution of paths to individual file system objects, permission checks, and the management of open files. Basically, the AFS provides an interface analogous to the POSIX interface but on the level of file system objects instead of paths. This specification abstracts completely from any flash-specific concepts and thus the VFS is not limited to be used exclusively with flash file systems. Details of the POSIX specification as well as the sequential refinement to VFS and AFS can be found in [18].

Recently, we worked on a locking concept for the VFS that allows concurrent calls to the file system interface. The approach taken focuses on enabling parallel access to file contents, in particular we want to allow arbitrary concurrent reads as well as concurrent writes to different files. Therefore, we chose a fine-grained locking strategy for files, whereas we applied a coarse-grained strategy for the directory tree. This means that each file is protected by an individual reader-writer lock while a single reader-writer lock is used for the entire directory tree. It should be noted, that parallel traversal of the directory tree is still possible as long as no *structural operation* is performed. Thus, we think this is a good trade-off between development or verification effort and performance gain. We augmented the existing sequential versions of VFS and AFS with locks and ownerships respectively and proved that the interleaved implementation of VFS is linearizable and deadlock-free using atomicity refinement as explained in Sec. 2. The verification showed that a strict order for acquiring and releasing locks is beneficial for our approach.

AFS is refined by the actual Flash File System (FFS). Additionally, AFS is refined by a Cache component that caches data structures used at the interface of the core file system. The Cache is integrated as a decorator, i.e. it wraps around the AFS in the sense that it uses AFS as a subcomponent and also implements the interface of AFS. This allows the file system to be used both with and without Cache. The main goal of this integration was to allow *write-back* caching of content operations. However, write-back caching can have significant effects on the crash behavior of a system. In [4] we presented a novel correctness criterion for this sort of file system caches and proved that Flashix complies with it. We sketch the most important concepts of this addition and the proof idea in Sec. 4.

The FFS was the layer at which we started the development of the Flashix file system in [38]. It introduces concepts specific to flash memory and to log-structured file systems. Updates to file system objects must be performed out-of-place and atomically. For this purpose, the FFS is built upon an efficient Index, implemented by a wandering B^+-Tree, and a transactional Journal. Both are specified abstractly in the component FFS-Core. New versions of file system objects are encapsulated in nodes and grouped into transactions that are then written to a log. To keep track of the latest versions of objects, the locations of them on the flash memory are stored in the Index. The Index exists in two versions, one persisted on flash and one in RAM. Updates on the Index are initially performed only in RAM in order to improve performance as these update are quite costly to perform on flash. Only during *commits* that are executed regularly, the latest version of the Index is written to flash. The transactional Journal ensures that, in the event of a crash, the latest version of the RAM Index can be reconstructed. This can be done by replaying the uncommitted entries in the log starting from the persisted Index on flash. In doing so, incomplete transactions are discarded to comply to the atomicity properties expected by the VFS.

Another crucial mechanism implemented in this layer is *garbage collection*. Due to their out-of-place nature, updates to the file system leave garbage data behind. This data must first be deleted before the storage space it occupies can be used again. But since flash blocks can only be erased as a whole, garbage collection chooses suitable blocks for deletion (preferably blocks with a high percentage of garbage), transfers remaining valid data of that block to another one, and finally triggers the erasure of the block. This mechanism is not triggered explicitly by calls to the file system, instead it must be performed periodically to ensure that the file system does not run out of space. Hence we extracted garbage collection into a separate thread, we give more details on this concurrency extension in Sec. 5.

Both the transactional Journal and the B^+-Tree write nodes on the flash device. The Node Encoding component is responsible for serializing these nodes to bytes before they can be written to flash. It also keeps track of the allocation of erase blocks and, for each block, the number of bytes still referenced by live data, i.e. by nodes of the index or nodes that store current versions of file system objects. This information is used to determine suitable blocks for garbage collection. Besides that, the layer ensures that writing of nodes appears to be atomic to the Journal and Index. It detects partially written nodes that may occur through crashes or hardware failures and takes care of them. A more in-depth view on these components and the garbage collection is given in [16].

All serialized nodes pass a Write Buffer. This buffer cache tackles the restriction that flash pages can only be written sequentially and as a whole. It caches all incoming writes and only issues a page write once a page-aligned write is possible, i.e. the write requests have reached the size of one flash page in total. Otherwise, padding nodes would have to be used in order to write partially filled pages, which both would increase the absolute number of writes to flash and the amount of wasted space on the flash device. Introducing such an *order-preserving* write-back cache (written data leaves the cache in the same order as it entered it) also affects the crash behavior of the file system. In [29] we give a suitable crash-safety criterion as well as a modular verification methodology for proving that systems satisfy this criterion.

The Superblock component is responsible for storing and accessing the internal data structures of the file system. A specific part of the flash device is reserved for this data. They are written during a *commit* only, since persisting each update would have a significant negative impact on the performance of the file system. A critical task of this layer is to ensure that commits are performed atomically using a data structure called *superblock*.

Finally, the Erase Block Manager (EBM) provides an interface similar to the one of MTD (read, write, erase). However, the EBM introduces an indirection of the *physical blocks* of the flash device to *logical blocks* and all of its interface operations address logical blocks only. These logical blocks are allocated on-demand and mapped to physical blocks. The indirection is used to move logical blocks transparently from one physical block to another one which is necessary to implement *wear leveling*. Wear leveling ensures that within some bounds all blocks are erased the same number of times. This is necessary to maximize the life time of the flash memory, as erasing a flash block repeatedly wears it out, making it unusable. To ensure a bound, the number of performed erases is stored in an *erase counter*. Wear leveling finds a logical block that is mapped to a physical block with low erase count and re-maps it to a block with high erase count. Since a logical block with low erase-count typically contains a lot of *stale data* that has not been changed for some time and therefore is not likely to change soon, the number of erases is kept at the same level and the lifetime of the flash device increases.

The EBM uses the Header Encoding component for the serialization and deserialization of administrative data, most important an inverse mapping stored in the physical blocks containing the numbers of the logical blocks they are mapped to.

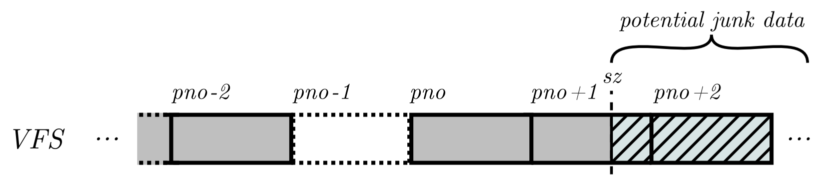


Fig. 4: Representation of file content in VFS.

A sequential version of the Erase Block Manager is explained in detail in [30]. But similar to garbage collection, wear leveling has to be performed regularly without being triggered by the user and so we adjusted the EBM to run wear leveling in a separate thread as well. Another thread is used to perform the erasure of blocks asynchronously, too. We illustrate this extension and the verification methodology for introducing concurrency to a refinement hierarchy on a simplified version of the EBM in [36].

4 Crash-Safe VFS Caching

A common technique to get a highly efficient file system implementation is the use of caching. Flashix features several caches in multiple layers: the B$^+$-Tree contains a *write-through* cache for the directory structure, the Write Buffer uses an *order-preserving* write-back cache for flash pages, and lately we added a *non-order-preserving* write-back cache for file contents to the VFS layer.

Since all in-memory data is lost in the event of a crash, crash-safety is a critical aspect when integrating caches. For write-through caches this is unproblematic as cached data is only a copy of persisted data on flash. In [29] we presented a crash-safety criterion for order-preserving caches: basically a crash has the effect of retracting a few of the last executed operations. But this criterion is too strong for non-order-preserving caches and so in [4] we proposed a more relaxed criterion and proved that our VFS caches comply with it. We will now give an overview over the crucial aspects of this latest extension.

While file contents are represented as a finite sequences of bytes in POSIX, VFS breaks this abstract representation down to a map of fixed-size byte arrays (*pages*) and an explicit file size as shown in Fig. 4. Each box depicts a page and is identified by a page number *pno*. The map offers the advantage of a sparse representation with the convention that missing pages are filled with zeros only. This is indicated by a white dashed box (*pno-1* in the figure). A important detail is the possibility of random data (hatched) beyond the file size *sz*, resulting from prior crashes or failed operation executions. This is especially relevant when the file size is not *page-aligned* and the page of *sz* (*pno+1* in the figure) contains actual garbage data (non-zero bytes) beyond *sz*.

When extending a file, such garbage data must not become visible as this would not match the POSIX requirements. There are two ways to change the file size: explicitly with a *truncation* or by writing content beyond the current

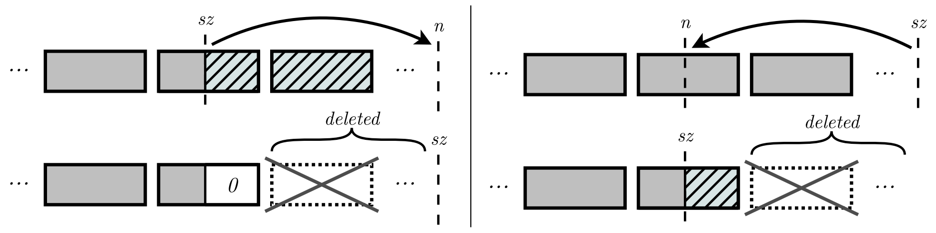


Fig. 5: Truncation to a larger size $sz \leq n$ (left) and to a smaller size $n < sz$ (right).

file size. A truncation crops the content of a file to a new size n and ensures that all data within the file is valid. Hence, in addition to updating the size, the actual content may also need to be updated. To increase the file size, possible junk data in the page of sz needs to be cleared with zeros and pages beyond the old file size are deleted (Fig. 5 on the left). On the other hand, junk data can remain in the page of the new size when shrinking the file (Fig. 5 on the right).

The VFS breaks down a write into three steps. First, possible junk data beyond the file size is removed. This is done by a truncation to the current file size ($n = sz$ in Fig. 5). Then the respective pages are written bottom up individually. If writing a page causes an error, writing stops and hence only a prefix of the requested bytes are written. Finally, the file size is adjusted to the position of the last written byte if data was written beyond the old file size.

By using the decorator pattern, VFS caching could be integrated into our refinement hierarchy without changing existing components by adding a single new component. The component Cache refines and uses AFS at the same time as shown by Fig. 3a. To cache content operations in a write-back manner, writes and truncations are aggregated in local page and size caches. Additionally, write-through caches for header nodes (containing meta data of files and directories) and directory entries are implemented. Page writes are stored in the page cache and truncations or size updates lead to updates in the respective size caches. We only cache the most current file size while multiple truncations are aggregated by caching the minimal truncation size since the last synchronization. Reading a page tries to find a corresponding cache entry. If it does not exist in the cache, the page is read from the Flash File System (FFS) and (if it exists) stored in the cache.

Updating the persisted content happens only if it is triggered by a call of the **fsync** operation for the respective file. A call to **fsync** starts the synchronization with a single truncation of the persisted content to the minimal truncation size. Then, similar to a VFS write, all dirty pages of the file in the cache are written to the FFS bottom up and finally the file size is adjusted if necessary.

Showing functional correctness of this addition can be done by a single data refinement and will not be considered further here. However, proving crash-safety is quite difficult. If a crash occurs, all data in the volatile state of the Cache component is lost such that unsynchronized writes and truncations have not taken place. To ensure crash-safety, it must be shown that each crash results

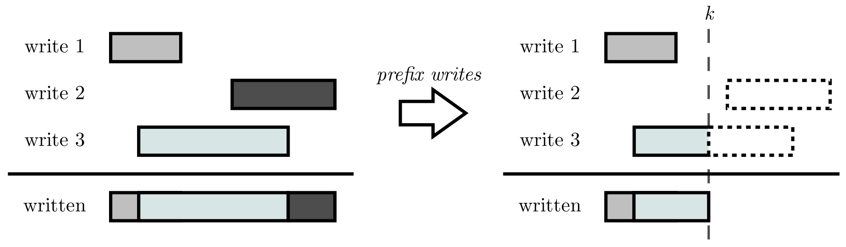


Fig. 6: Write-Prefix Crash Consistency: prefix writes for a crash during the synchronization of a file just before syncing byte k.

in a consistent state. Normally, the effect of a crash would be described by an explicit change of state. But this is usually not practicable for write-back caches and the crash-safety criterion introduced in [29] is not suitable for non-order-preserving caches, too. So we use a new criterion called *Write-Prefix Crash Consistency (WPCC)*. It states that for any file a crash has the effect of retracting all write and truncate operations since the last synchronization of that file and re-executing them, potentially resulting in writing prefixes of the original runs [4].

This follows from the effects of a crash during persisting a cached file as shown in Fig. 6 for the POSIX data representation. On the left, there is shown how multiple overlapping writes combine to a sequence of written bytes. Since cached pages are written bottom up during synchronization, a crash in the middle of **fsync** results in a prefix of these bytes being persisted (on the bottom right in Fig. 6). If the crash occurs just before persisting the byte at position k, the resulting state can be explained by writing prefixes of the original instructions (namely those prefixes that have written exactly the bytes beyond k). This can result in complete writes (write 1 in Fig. 6), partial writes (write 3), or writes that are completely lost (write 2). To archive this behavior it is essential that VFS writes as well as synchronizations are performed bottom up and that writes can fail after writing a arbitrary number of bytes.

Informally, the criterion describes the effects of a crash by finding an alternative run where loosing cached data has no noticeable effect. These alternative runs may differ at most from their original runs in that writes since the last synchronization have written prefixes of their original runs. Because such an alternative run is a valid run and hence results in a consistent state, the original crashing run yields a consistent state as well.

The main effort for proving crash-safety is to show that such alternative runs exist for any possible occurrence of a crash. While finding suitable runs for crashes outside of **fsync** is unproblematic (if nothing was persisted, failed executions of cached operations can be chosen), this is especially hard for crashes within **fsync**. One particular challenge is to show that the aggregation of multiple truncate operations matches WPCC if the minimal truncation was executed but the final file size was not yet synchronized at the event of the crash. This can lead to slightly different junk data in the write-prefix run such that on the level of AFS the contents of the crashed run and the write-prefix run differ beyond the file size. However, this junk data is only visible in AFS as the abstraction to POSIX ignores all bytes beyond the file size. Hence an alternative run can be found on the level of POSIX, but this required to extend the proof work to another layer of abstraction. More details on the difficulties and the concrete proof strategy involving multiple commuting diagrams can be found in [4].

5 Concurrent Garbage Collection

Besides allowing concurrent calls to the file system interface as briefly outlined in Sec. 3, moving certain internal mechanisms into separate threads also introduces additional concurrency to the file system. Hence the affected models have to be modified in order to avoid conflicts resulting from parallel executions of operations.

The expansion with concurrent garbage collection ranges from the FFS layer to the Journal layer. In the FFS the concurrent operation for garbage collection is introduced (Fig. 8). This operation is not part of the interface (it refines skip, i.e. it has no visible effect for clients). Hence, it can not be called by any client components. Instead it will be repeated infinitely within its own thread. To ensure that garbage collection is not performed continuously, especially when no more space can be regained, a condition variable *gc_cond* is used [1]. At the beginning of each iteration the thread blocks at the **condition_wait** call until it is signaled by another thread to start. The concrete garbage collection algorithm is specified in the FFS-Core and implemented in the Journal component, so after being signaled the operation **ffs_core_gc** is called.

Signaling takes place in all FFS operations that may modify the file system state in the sense that either entries are written to the log and hence space on the flash device is allocated or garbage is introduced by invalidating allocated space. Such operations, as shown generically in Fig. 7, emit a signal to *gc_cond* after they have updated the index.

The implementation of **ffs_core_gc** in the Journal component then first checks whether there is a block which is suitable for garbage collection. If that is the case, all still referenced nodes of this block are collected, these nodes are then written to the journal, and their new addresses are updated in the index accordingly. Finally, if the referenced data was successfully copied, the block can be marked for erasure.

As an additional thread is introduced in the FFS, established ownerships in the VFS/AFS layer are not sufficient to prevent data races between the garbage collection thread and other threads. For this reason the reader-writer lock *core_lock* is added to the FFS component. It is used to acquire exclusive

[1] Note that condition variables are always coupled to a mutex. Here *gc_cond* is coupled to *gc_mutex*. Signaling a condition requires to hold the corresponding mutex. Starting to wait for a signal requires to hold the mutex as well, however, the mutex is released during waiting. As soon as a signal was emitted and the mutex is free, the waiting thread acquires the mutex and continues its execution.

```
**ffs_operation**(...) {
  ...
  $nd_1$ := inodenode($key_1$, ...);
  ...
  **rwlock_wlock**( ; *core_lock*);
  **ffs_core_wacquire**();
  **ffs_core_journal_add**($nd_1$, ... ; $adr_1$, ... ; *err*);
  **if** *err* = ESUCCESS **then** {
    **ffs_core_index_store**($key_1$; $adr_1$);
    ...
    **mutex_lock**( ; *gc_mutex*);
    **condition_signal**( ; *gc_cond*, *gc_mutex*);
    **mutex_unlock**( ; *gc_mutex*);
  };
  **ffs_core_release**();
  **rwlock_wunlock**( ; *core_lock*);
  ...
}
```

Fig. 7: General operation scheme of modifying FFS operations.

```
**ffs_gc**() {
  **mutex_lock**( ; *gc_mutex*);
  **condition_wait**( ; *gc_cond*, *gc_mutex*);
  **mutex_unlock**( ; *gc_mutex*);

  **rwlock_wlock**( ; *core_lock*);
  **ffs_core_wacquire**();
  **ffs_core_gc**();
  **ffs_core_release**();
  **rwlock_wunlock**( ; *core_lock*);
}
```

Fig. 8: FFS garbage collection operation.

or shared ownership for the journal and index data structures. We did not head for a more fine-grained locking approach since usually updates affect nearly all parts of the state of FFS-Core anyway. However, using reader-writer locks still allows for concurrent read accesses to the Journal.

Modifying operations in the FFS as shown in Fig. 7 always follow the same scheme. First, all new or updated data objects are wrappend into nodes ($nd_1, ...$) with an unique key ($key_1, ...$). Depending on the concrete operation, nodes for inodes, dentries, and pages are created. Then these nodes are grouped into transactions and appended to the log using the **ffs_core_journal_add** operation. If successful, i.e. the operation returns the code ESUCCESS, the operation returns the addresses ($adr_1, ...$) where the passed nodes have been written to. Finally, the index is updated by storing the new addresses of the affected keys via the operation **ffs_core_index_store**[2]. It is crucial that garbage collection is never performed between these calls since this could result in a loss of updates (e.g. if garbage collection moves nodes updated by the operation), potentially yielding an inconsistent file system state. Hence, the locking range must include the **ffs_core_journal_add** as well as all **ffs_core_index_store** calls.

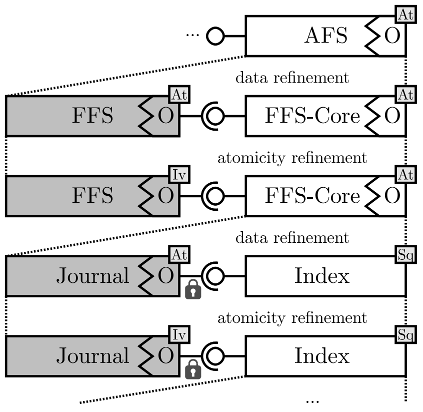


Fig. 9: Refinement hierarchy extended by concurrent garbage collection.

To prove that this locking strategy is in fact correct, i.e. that the interleaved components are linearizable, we again apply *atomicity refinement*. This results

[2] Some operations also update the index by removing entries from it.

```
journal_operation(...) {
  ...
  mutex_lock(; idx_lock);
  index_operation_i(...);
  mutex_unlock(; idx_lock);
  ...
  mutex_lock(; idx_lock);
  index_operation_j(...);
  mutex_unlock(; idx_lock);
  ...
}
```

```
journal_operation(...) {
  ...
  atomic {
    mutex_lock(; idx_lock);
    index_operation_i(...);
    mutex_unlock(; idx_lock);
  }
  ...
  atomic {
    mutex_lock(; idx_lock);
    index_operation_j(...);
    mutex_unlock(; idx_lock);
  }
  ...
}
```

```
journal_operation(...) {
  atomic {
    ...
    mutex_lock(; idx_lock);
    index_operation_i(...);
    mutex_unlock(; idx_lock);
    ...
    mutex_lock(; idx_lock);
    index_operation_j(...);
    mutex_unlock(; idx_lock);
    ...
  }
}
```

Fig. 10: Reduction steps of a Journal operation (from left to right).

in the expansion of the refinement hierarchy shown in Fig. 9. Usually, atomicity refinement would have to be applied to all layers below Index, too, but we did not put any effort in making a interleaved version of the B$^+$-Tree yet. Instead we locked the interface of the Index (depicted by 🔒). This means that each call to an Index operations **index_operation** requires the current thread to be an exclusive owner of the Index component. In the Journal this is realized by surrounding these calls with a mutex *idx_lock* as shown in Fig. 10 on the left. Owning a subcomponent exclusively ensures that the subcomponent is only called sequentially and hence allows to directly use the unaltered sequential version of the subcomponent and its refinements (denoted by Sq in Fig. 9).

FFS-Core is augmented with ownership ghost state matching the reader-writer lock *core_lock* of FFS. The FFS operations acquire and release this ownership according to the locking ranges (see Fig. 7 and Fig. 8). While the ownership granularity of AFS (owned directory tree, owned files, ...) does not match the state of the FFS-Core or the Journal, the information about which files etc. are owned when an operation is called (encoded in the preconditions) is still relevant for the FFS in order to preserve functional correctness. For example, an owned file must not be removed from the index while its metadata is updated. Therefore, ownership ghost state is added to FFS analogously to AFS and corresponding ownership properties are established. This is sufficient to prove that the interleaved FFS can be reduced to an atomic FFS via atomicity refinement. The data refinement of the atomic AFS to the atomic FFS is basically identical to the original sequential refinement, in addition, it must only be shown that their respective ownerships match.

When proving the atomicity refinement of the Journal, it is apparent that Index operations together with their surrounding lock calls form atomic blocks like in the center of Fig. 10. But as most operations have multiple calls to the Index, this is not sufficient to reduce these operations to completely atomic ones. It remains to show that these blocks as well as statements that access the local state of the Journal move appropriately (usually they have to be both mover). To prove this, the ownership information of the FFS-Core component can be used. The Journal is augmented with ownership properties, operations and preconditions that match those of the FFS-Core and so accesses to the local state can be inferred to be both movers. The information that a certain ownership is acquired at the calls of Index operations and their associated locking operations allows to prove that these blocks in fact are movers and hence to further reduce the operations to be atomic (Fig. 10 on the right). Although the proofs are simple, this is quite elaborate since many commutations have to be considered. The data refinement of FFS-Core to the atomic Journal then again is basically identical to the sequential refinement.

6 Evaluation

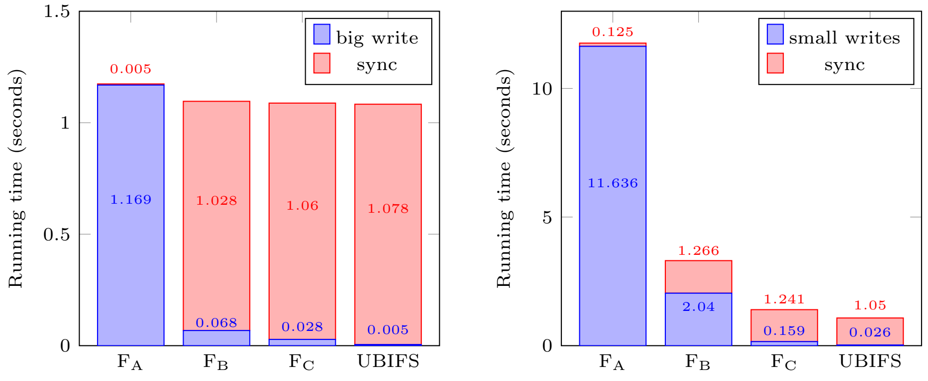


Fig. 11: Nano write benchmarks on Flashix and UBIFS: big write (left) and small writes (right). Flashix was used in three different configurations: sequentially without VFS cache (F_A), sequentially with VFS cache (F_B), and with VFS cache and concurrent wear leveling and garbage collection (F_C).

To evaluate the performance of the Flashix file system we perform a collection of microbenchmarks. This gives us some insight in whether the expansions we have made, especially those described in Sec. 4 and Sec. 5 or in [4,36], have an impact on the performance. Furthermore, we want to compare the performance of Flashix with state-of-the-art flash file systems like UBIFS [23].

All benchmarks were run within a virtualized Linux Mint 19.3 distribution, using 3 Cores of a Intel Core i5-7300HQ CPU and 4,8 GB of RAM. The flash device was simulated in RAM using the NAND simulator (nandsim) integrated into the Linux kernel [27]. The numbers shown in the following represent the mean of 5 benchmark runs in which the mean standard deviation across all runs is below 4.5% (this translates to a mean deviation in runtime of less than 0.16 seconds).

We chose some small workloads that represent everyday usage of file systems: copying and creating/extracting archives. Copying an archive to the file system

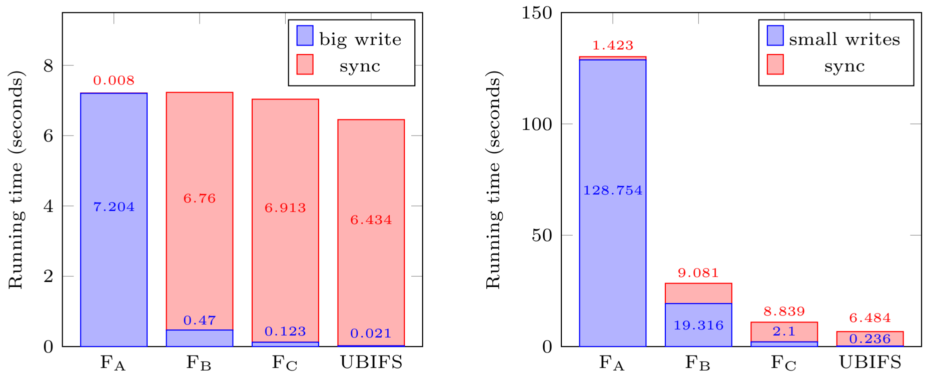


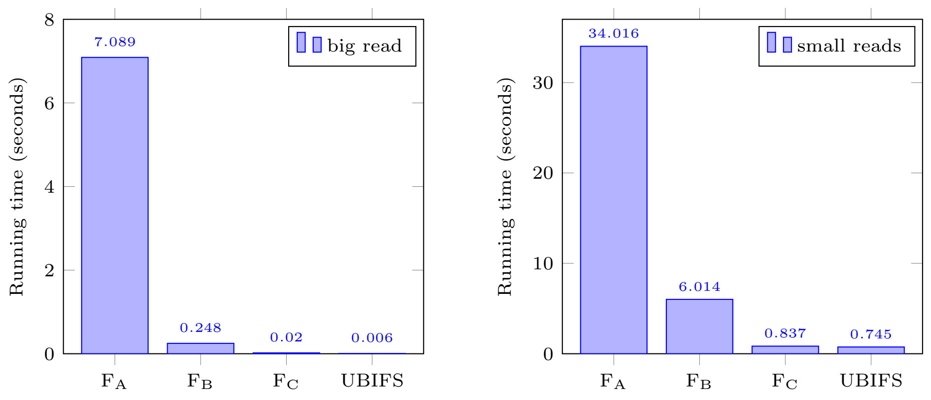
Fig. 12: Vim write benchmarks on Flashix and UBIFS: big write (left) and small writes (right). Flashix was used in three different configurations: sequentially without VFS cache (F_A), sequentially with VFS cache (F_B), and with VFS cache and concurrent wear leveling and garbage collection (F_C).

results in the creation of a file and writing the content of that one file. Analogously, copying an archive from the file system yields in reading the content of the file. Hence, we call these workloads *big write* and *big read* respectively. On the other hand, extracting an archive results in the creation of a directory structure containing many files. The contents of the created files are written as well, however, these are multiple smaller writes compared to the single big write when copying. Creating an archive from a directory structure on the file system requires to read all directories and files. Hence, we call such workloads *small writes* and *small reads* respectively. As sample data we used archives of the text editors Nano[3] and Vim[4].

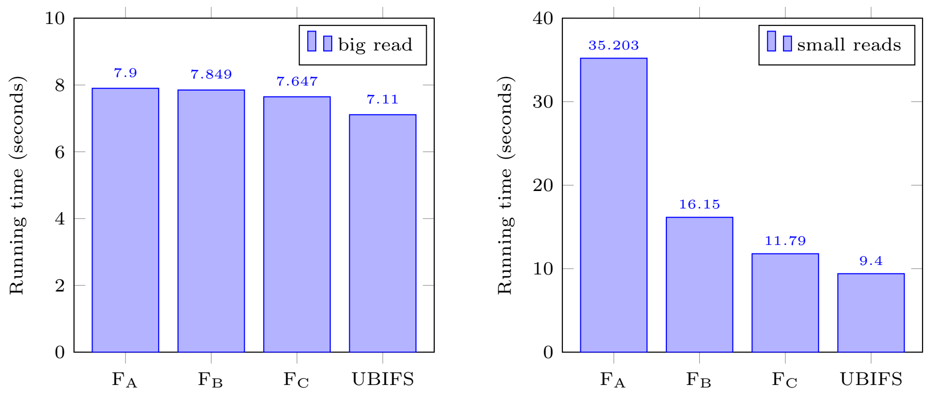
Fig. 11 shows the results of the write benchmarks with Nano. When comparing the uncached configuration (F_A) with the cached configuration (F_B) of Flashix, one can see that adding the VFS cache has indeed a significant impact on write times (depicted in blue). But as these times do not include persisting the cached data to flash, we enforced synchronization directly afterwards via *sync* calls (depicted in red). For big writes the combined runtime of the cached configuration is similar to the uncached one. For small writes though, the combined runtime of F_B is substantially faster since repeated reads to directory and file nodes during path traversal can be handled by the cache.

Moving wear leveling and garbage collection into separate threads (F_C) further improves the performance. This is especially noticeable in the small writes workload where the write time can be reduced by about one order of magnitude. In the sequential configurations (F_A and F_B), after each toplevel operation it was checked whether garbage collection or wear leveling should be performed.

[3] nano-2.4.2.tar: approx. 220 elements, 6.7 MB
[4] vim-7.4.tar: approx. 2570 elements, 40.9 MB



(a) Vim read benchmarks with with *hot* caches.



(b) Vim read benchmarks with with *cold* caches.

Fig. 13: Vim read benchmarks on Flashix and UBIFS: big read (left) and small reads (right). Flashix was used in three different configurations: sequentially without VFS cache (F_A), sequentially with VFS cache (F_B), and with VFS cache and concurrent wear leveling and garbage collection (F_C).

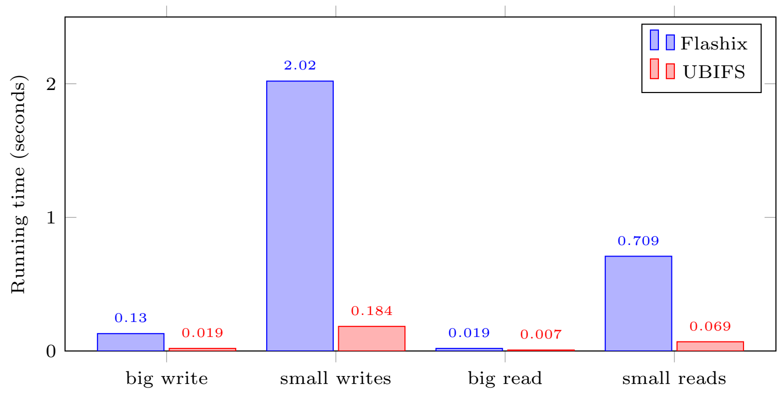


Fig. 14: Vim benchmarks on Flashix and UBIFS without flash delays.

During these checks and potential subsequent executions of the algorithms, other POSIX operation calls were blocked. In the concurrent configuration (F_C) these blocked time can be eliminated for the most part since writing to the cache does not interfere with garbage collection or wear leveling. As small writes trigger considerably more toplevel operation calls, this effect is much more noticeable than with big write workloads.

Compared to UBIFS, the current version of Flashix performs as expected. Runtimes of the F_C configuration are always within the same order of magnitude of those of UBIFS. This also applies for running the benchmarks with a larger archive like Vim shown in Fig. 12.

Similar effects can be observed when considering read workloads as shown in Fig. 13. Adding caches significantly speeds up both reading a single big file and reading many small files when the caches are *hot* (Fig. 13a), i.e. when the requested data is present in the caches. Likewise, moving wear leveling and garbage collection to background processes brings down the runtime by an order of magnitude. When reading from *cold* caches, i.e. when no requested data is present in the caches, the speed up is much more subtle since the main delay results from reading data from flash. As shown in Fig. 13b, for big reads there is hardly any improvement from F_A to F_B or F_C. However, both expansions have an impact on the runtime of small read workloads for the same reasons as for small write workloads: repeated reads to the directory structure can be handled by the cache and blocked time for garbage collection and wear leveling can be eliminated. With these additions one can see that Flashix is competitive with UBIFS regarding read performance, too.

In future work we plan to further improve the performance of Flashix by improving our code generator as the generated code is not optimal in terms of allocating/deallocating and copying data structures. The optimization potential becomes apparent when comparing the raw in-memory runtimes of Flashix and UBIFS like in Fig. 14. Here we instructed nandsim to not simulate any delays for accessing the simulated NAND memory. The results show that UBIFS is still up to a factor of 10 faster than Flashix for the Vim microbenchmarks (the Nano benchmarks yield similar results). First experiments show, that even simple routines can affect performance noticeably if they are generated inefficiently. For example, we found out that a simple optimization of a routine used in the Journal for calculating the required space of a node-list on flash improved the runtime by up to 30% compared to the generated code. Hence we plan to apply data flow analysis to identify this and other locations where such optimizations can be performed. We are optimistic that this will further close the gap to state-of-the-art handwritten file systems.

7 Related Work

There are some other projects related to verified file systems.

Damchoom et al. [11] develop a flash file system by using incremental refinement. Concurrency is verified on a similar level as AFS for reading and writing of file content and for wear-leveling as well. Synchronization between threads is implicit by semantics of Event-B [1] models. But this makes it difficult to derive executable code. Amani et al. [2] design the flash file system BilbyFS to research their tool Cogent for generating verified C code. The system can also derive specifications for Isabelle/HOL. BilbyFS has a similar but simpler structure as Flashix. For instance it builds on top of the EBM instead of MTD. It supports caching mechanism but not on the level of VFS. Crash-safety has not been considered so far.

(D)FSCQ [8, 9] is a sequential implemented file system developed by Chen et al., which is targeted for regular disks with random access, not 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. Verification is done by using Crash Hoare Logic and Haskell code is derived from the specification.

Our crash-safety criterion for order-preserving caches is similar to buffered durable linearizability [24], though there are some differences: the criterion is purely history based, it allows to construct a prefix of the history, where pending operations can be completed anew, similar to our approach. It however allows to complete pending operations in the shortened history even after operations that have started after the crash. This is useful for a concurrent recovery routine, that may restart operations that crashed. It is disallowed in our approach, since not relevant for file systems. Buffered durable lineariability also disallows the effect of closing open files that we specify separately with a Crash predicate.

Verification of a sophisticated locking scheme that locks inodes hand-over-hand (lock coupling) has recently been done in the theorem prover Coq for a file system prototype called AtomFS that is directly programmed in C and stores data in RAM [44]. A particular challenge for the proof of linearizability solved there was the rename operation, that moves directories (whole subtrees). The operation has to lock both the source and target directory, but has to avoid deadlocks. It should be possible to port this locking scheme to our file system.

Other, older related work can be found in our prior work [4, 36].

8 Conclusion

The Flashix project has developed the first realistic verified file system using a refinement- and component-based approach that generates code at the end.

Being realistic however had the price that the individual components had to be intertwined carefully, which caused lots of effort and was therefore substantially harder than analyzing concepts individually.

We think that developing such a large system without suitable modularization and abstraction by verifying concrete C-code directly would have been an almost impossible task.

The use of abstract data types and components, that allows efficient verification comes at a price, however. Since abstract data types have the semantics of predicate logic, which is a value semantics that does not take sharing, allocation, or destructive updates into consideration, generating correct, efficient C-code is still a challenge.

Generating functional (non-destructive) code instead has long been done by theorem provers, but this would be hopelessly inefficient for a file system, where destructively updated arrays (buffers, pages, blocks) are crucial for efficiency: we tried using Scala's immutable type Vector once, but the generated code is slower by at least one order of magnitude.

It is possible to refine individual pieces of the abstract code to heap-based destructive code, and this is occasionally necessary (e.g. to represent search trees efficiently as pointer structures), and the verification task can be supported by using a library for separation logic, however refining *all* abstract data structures with pointer-structures would mean to analyze sharing manually, and to duplicate all code.

Another alternative is to enforce a linear type system on abstract specifications, for which a code generator could be proven correct [28].

Our current code generator follows the principle of *not* sharing data structures in the resulting C code to have definite allocation and deallocation points, and to allow destructive updates. This however, enforces copying x and all its substructures to y, when executing an assignment x := y. We already do a simple liveness check (if y is no longer used, then copying can be avoided), and some more ad-hoc optimizations to avoid unnecessary copying.

Still, a systematic data flow analysis, that allows sharing in places where it is harmless, and avoids copying wherever possible, should be able to close a large part of the still existing gap between the efficiency of our generated code and the hand-written code of UBIFS.

Implementation of such a data flow analysis is still future work, and we also want to tackle formalization and verification of such an approach, thereby establishing the correctness of the code generator.

References

1. J.-R. Abrial. *Modeling in Event-B - System and Software Engineering.* Cambridge University Press, 2010. `doi:10.1017/CBO9781139195881`.
2. 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.
3. E. Bila, J. Derrick, S. Doherty, B. Dongol, G. Schellhorn, and H. Wehrheim. Modularising Verification of Durable Opacity. *Logical Methods in Computer Science*, 2021. submitted, draft available from the authors.
4. S. Bodenmüller, G. Schellhorn, and W. Reif. Modular Integration of Crashsafe Caching into a Verified Virtual File System Switch. In *Proc. of 16th International Conference on Integrated Formal Methods (IFM)*, volume 12546 of *LNCS*, pages 218 – 236. Springer, 2020.
5. E. Börger. The ASM Refinement Method. *Formal Aspects of Computing*, 15(1–2):237–257, 2003.
6. E. Börger and D. Rosenzweig. The WAM—definition and compiler correctness. In C. Beierle and L. Plümer, editors, *Logic Programming: Formal Methods and Practical Applications*, Studies in Computer Science and Artificial Intelligence 11, pages 20–90. Elsevier, Amsterdam, 1995.
7. E. Börger and R. F. Stärk. *Abstract State Machines — A Method for High-Level System Design and Analysis.* Springer, 2003.
8. H. Chen. *Certifying a crash-safe file system.* PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, United States, 2016.
9. 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.
10. D. Cooper, S. Stepney, and J. Woodcock. Derivation of Z Refinement Proof Rules: forwards and backwards rules incorporating input/output refinement. Technical Report YCS-2002-347, University of York, 2002. URL: `http://www-users.cs.york.ac.uk/susan/bib/ss/z/zrules.htm`.
11. K. Damchoom and M. Butler. Applying Event and Machine Decomposition to a Flash-Based Filestore in Event-B. In *Proc. of the Brazilian Symposium on Formal Methods (SBMF)*, volume 5902 of *LNCS*, pages 134–152. Springer, 2009.
12. W. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*, volume 47 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1998.
13. J. Derrick and E. Boiten. *Refinement in Z and in Object-Z : Foundations and Advanced Applications.* FACIT. Springer, 2001. second, revised edition 2014.
14. T. Elmas, S. Qadeer, and S. Tasiran. A Calculus of Atomic Actions. In *Proceeding POPL 2009*, pages 2–15. ACM, ACM, 2009.
15. 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.

16. 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.
17. 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).
18. 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.
19. H. Grandy, M. Bischof, G. Schellhorn, W. Reif, and K. Stenzel. Verification of Mondex Electronic Purses with KIV: From a Security Protocol to Verified Code. In *FM 2008: 15th Int. Symposium on Formal Methods*. Springer LNCS 5014, 2008.
20. D. Haneberg, N. Moebius, W. Reif, G. Schellhorn, and K. Stenzel. Mondex: Engineering a provable secure electronic purse. *International Journal of Software and Informatics*, 5(1):159–184, 2011. `http://www.ijsi.org`.
21. D. Haneberg, G. Schellhorn, H. Grandy, and W. Reif. Verification of Mondex Electronic Purses with KIV: From Transactions to a Security Protocol. *Formal Aspects of Computing*, 20(1), January 2008.
22. M.P. Herlihy and J.M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.
23. A. Hunter. A brief introduction to the design of UBIFS. URL: `http://www.linux-mtd.infradead.org/doc/ubifs_whitepaper.pdf`, 2008.
24. 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.
25. C. Jones and J. Woodcock, editors. *Formal Aspects of Computing*, volume 20 (1). Springer, January 2008.
26. R. J. Lipton. Reduction: A Method of Proving Properties of Parallel Programs. *Communications of the ACM*, 18(12):717–721, 1975.
27. Linux MTD: NAND and NAND simulator. `http://www.linux-mtd.infradead.org/faq/nand.html`.
28. L. O'Connor, Z. Chen, C. Rizkallah, S. Amani, J. Lim, T. Murray, Y. Nagashima, T. Sewell, and G. Klein. Refinement through Restraint: Bringing down the Cost of Verification. In *Proc. of the 21st Int. Conf. on Funct. Prog. Languages (ICFP)*, ICFP 2016, page 89–102, New York, NY, USA, 2016. Association for Computing Machinery.
29. J. Pfähler, G. Ernst, S. Bodenmüller, G. Schellhorn, and W. Reif. Modular Verification of Order-Preserving Write-Back Caches. In *Proc. of 13th International Conference on Integrated Formal Methods (IFM)*, volume 10510 of *LNCS*, pages 375–390. Springer, 2017.
30. 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.
31. The Open Group Base Specifications Issue 7, IEEE Std 1003.1, 2018 Edition. The IEEE and The Open Group, 2017.
32. G. Schellhorn. *Verification of Abstract State Machines*. PhD thesis, Universität Ulm, Fakultät für Informatik, 1999. URL: `https://www.uni-augsburg.de/en/fakultaet/fai/isse/prof/swtse/team/schellhorn/`.

33. G. Schellhorn. Verification of ASM Refinements Using Generalized Forward Simulation. *Journal of Universal Computer Science (J.UCS)*, 7(11):952–979, 2001. URL: `http://www.jucs.org`.
34. G. Schellhorn. Completeness of ASM Refinement. *Electron. Notes Theor. Comput. Sci.*, 214:25–49, 2008.
35. G. Schellhorn and W. Ahrendt. The WAM Case Study: Verifying Compiler Correctness for Prolog with KIV. In W. Bibel and P. Schmitt, editors, *Automated Deduction — A Basis for Applications*, volume III: Applications, chapter 3: Automated Theorem Proving in Software Engineering, pages 165 – 194. Kluwer Academic Publishers, 1998.
36. G. Schellhorn, S. Bodenmüller, J. Pfähler, and W. Reif. Adding Concurrency to a Sequential Refinement Tower. In *Proc. of International Conference on Rigorous State-Based Methods (ABZ)*, volume 12071 of *LNCS*, pages 6–23. Springer, 2020.
37. G. Schellhorn, B. Tofan, G. Ernst, J. Pfähler, and W. Reif. Rgitl: A temporal logic framework for compositional reasoning about interleaved programs. *Annals of Mathematics and Artificial Intelligence*, 71(1):131–174, 2014.
38. A. Schierl, G. Schellhorn, D. Haneberg, and W. Reif. Abstract Specification of the UBIFS File System for Flash Memory. In *Proc. of International Symposium on Formal Methods (FM)*, volume 5850 of *LNCS*, pages 190–206. Springer, 2009.
39. R. F. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine—Definition, Verification, Validation.* Springer-Verlag, 2001.
40. K. Stenzel. A formally verified calculus for full Java Card. In *Algebraic Methodology and Software Technology (AMAST) 2004, Proceedings.* Springer LNCS 3116, 2004.
41. S. Stepney, D. Cooper, and J. Woodcock. AN ELECTRONIC PURSE Specification, Refinement, and Proof. Technical monograph PRG-126, Oxford University Computing Laboratory, July 2000. URL: `http://www-users.cs.york.ac.uk/susan/bib/ss/z/monog.htm`.
42. M. Szeredi. File system in user space. `http://fuse.sourceforge.net`.
43. Q. Xu, W.-P. de Roever, and J. He. The Rely-Guarantee Method for Verifying Shared Variable Concurrent Programs. *Formal Aspects of Computing*, 9(2):149–174, Mar 1997. `doi:10.1007/BF01211617`.
44. M. Zou, H. Ding, D. Du, M. Fu, R. Gu, and H. Chen. Using concurrent relational logic with helpers for verifying the atomfs file system. In *Proc. of SOSP*, SOSP '19, page 259–274. ACM, 2019.