Modular Verification of Order-Preserving Write-Back Caches [⋆]

Jörg Pfähler, Gidon Ernst, Stefan Bodenmüller, Gerhard Schellhorn, and Wolfgang Reif

Institute for Software and Systems Engineering
University of Augsburg, Germany
{pfaehler,ernst,bodenmueller,schellhorn,reif}@isse.de

Abstract. File systems not only have to be functionally correct, they also have to be crash-safe: a power cut while an operation is running must be guaranteed to lead to a consistent state after restart that loses as little information as possible. Specification and verification of crash-safety is particularly difficult for non-redundant write-back caches. This paper defines a novel crash-safety criterion that facilitates specification and verification of order-preserving caches. A power cut is basically observationally equivalent to a retraction of a few of the last executed operations. The approach is modular: It gives simple proof obligations for each individual component and for each refinement in the development. The theory is supported by our interactive theorem prover KIV and proof obligations for crash-safety have been verified for the Flashix flash file system.

Keywords: Write-Back Caching, Crash-Safe Refinement, Flash File Systems

1 Introduction

To be reliable, file systems have to be both functionally correct and crash-safe. Functional correctness is typically expressed in terms of a high-level specification of its operations, as given for example by the established POSIX standard [1]. Crash-safety is harder to prove, since it not only has to consider the states before and after operations. Instead, a power cut that interrupts an operation in any intermediate state must lead to a consistent state after reboot, where as little information as possible has been lost.

We develop Flashix [18], a file system for flash memory that is verified with the interactive theorem prover KIV [8] to be both functionally correct with respect to POSIX and crash-safe. Flashix is strongly modular: it is hierarchically composed of encapsulated components, which are formalized as data types [12,7] extended by a specification of the effect of a power cut and subsequent recovery.

[⋆] Supported by the Deutsche Forschungsgemeinschaft (DFG), “Verifikation von Flash-Dateisystemen” (grants RE828/13-1 and RE828/13-2).

Verifying crash-safety is critically affected by *caching* mechanisms employed by the implementation, as data structures in main memory are lost upon a power cut. Caches can be classified into *write-through* and *write-back* caches. The former can be reconstructed from persistent memory and are therefore fully redundant. Crash-safety is expressible by stating that the recovery operation restores the state from before the power cut. Losing a write-through cache due to a power cut is invisible to its clients (components that use the cache).

Write-back caches, on the other hand, lead to actual loss of data in the event of a power cut. The crash-safety of their clients then depends heavily on the exact nature of the data lost. Therefore the specification of the crash-safety of the cache needs to be propagated upwards through the component hierarchy.

Flashix is a log-structured file system. The log component appends log entries by using a write-back cache. It defers writes until the size of a page or sector is reached. This cache is *order-preserving*, i.e. the write operations to the storage device are in the same order as the writes to the cache. In our experience (Sec. 2 and [9]) if crash-safety is expressed as a *state-based* property, i.e. as a relation between the state before and after the power cut, it needs to be expressed on every level of abstraction, which complicates verification significantly.

The contribution of this paper is threefold. We propose a new correctness criterion for order-preserving caches called *quasi sequentially crash consistent*[1]. A storage system satisfies this criterion if a power cut takes the system's state backwards in time by retracting several system operations in order and by re-executing the earliest retracted operation. Secondly, we embed this *operations-based* property into the semantics of components (Sec. 3). This allows us to propagate it over component hierarchies implicitly via refinement (Sec. 4). The notion of refinement defined allows for substitution (Sec. 5). Sec. 6 shows that in practice considering the initial and final state of an operations execution is sufficient for the verification of crash-safety. Finally, we implemented support for the proof obligations of the theory in our interactive theorem prover KIV [8] and applied it to the Flashix file system. All models, proofs and the executable code are available online.[2]

2 Motivation

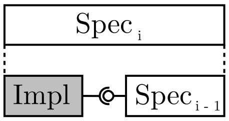

Fig. 1: Refinement

The formal development of a software system usually starts with a specification of its desired behavior and properties, e.g. POSIX in the case of a file system. The implementation then comprises a hierarchy of components, stacked as indicated by Fig. 1, i.e., each implementation refines (dotted lines) a specification $Spec_i$. It is a client of (−◎−) an abstraction of one or more subcomponents $Spec_{i-1}$. Either $Spec_{i-1}$ is refined further or serves as a specification for external components, e.g.

[1] The classification of Bornholt et al. [3] defines *sequential crash consistency.*

[2] `http://www.isse.de/flashix`

```
state $block$: Array⟨Byte⟩
append($in$ $buf$: Array⟨Byte⟩, $out$ $err$: Error)
  precondition
    $\# block + \# buf \leq$ BLOCK_SIZE
  atomic
    choose $n$: $\mathbb{N}$ with $n \leq \# buf$ in
      $block$ := $block$ ++ $buf[0 \ldots n]$
      $err$ := $(n = \# buf)$ ? ESUCCESS : EIO
crash
  $block' = block \downarrow$PAGE_SIZE
  with domain $true$
synchronized
  true
```

Fig. 3: Explicit Specification

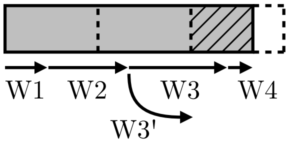

Fig. 4: Alternative Re-Execution

```
// same state & operations
crash
  $block' = block$
  with domain $page\text{-}aligned(block)$
synchronized
  $page\text{-}aligned(block)$
```

Fig. 5: Implicit Specification

the interface to flash hardware in the context of Flashix. Refinement guarantees that the final implementation has the properties of the top-level specification.

In [10] we have integrated the verification of crash-safety into this scheme: In addition to regular operations, each model is equipped with a specification of *resets*, which consist of the effect of crashes and their subsequent recovery, specified as a predicate over two states. We first illustrate this type of state-based specification with the Flashix write buffer [9] and then highlight a crucial problem with this approach and propose a different, operations-based specification.

The write buffer is visualized in Fig. 2. It alleviates the limitation that flash blocks can only be written sequentially and in page-sized chunks. The component keeps a page-sized buffer in RAM and writes it to flash as soon as the page-size is reached. A transactional log or journal uses the write buffer to record file system changes.

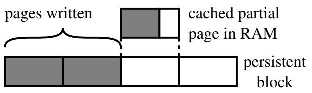

Fig. 2: Write Buffer [9]

Explicit Reset Specification. Fig. 3 depicts a first, natural abstraction of the write buffer that merges the cache and the contents of the flash block into one dynamically-sized array of bytes *block*. The `append` operation extends the contents of the abstract block by new data stored in *buf*. Since flash memory is inherently unreliable [20], the specification accounts for short writes that fail to persist the whole *buf* up to its size $\# buf$ and write just the subrange from 0 to n (n excluded) instead.[3] The specification is a program that cannot be interrupted by a power cut in an intermediate state, signified by the keyword `atomic`.

The effect of losing the cache is captured by the `crash` predicate. It restricts a transition from state s to s' of the system. In the case of the write buffer a prefix $block'$ of $block$ rounded down to the previous page boundary is taken, where $block \downarrow \texttt{PAGE\_SIZE} = block[0 ... (\# block) \downarrow \texttt{PAGE\_SIZE}]$ for $n \downarrow \texttt{PAGE\_SIZE} = n - (n \bmod \texttt{PAGE\_SIZE})$.

[3] Note that the POSIX specification [1] explicitly permits such short writes to surface at the system interface.

The problem with the approach is that the data lost by the reset of the cache component $Spec_{i-1}$ (in Fig. 1) must be propagated explicitly to the levels of abstraction given by its direct client $Spec_i$, and then to its client's client, until the top-level specification. This is particularly problematic if the hierarchy is deep or, as is the case in Flashix, higher levels of abstraction can not naturally express the property. Therefore, we split the specification of a reset into 1. a retraction transition followed by a re-execution transition and 2. a crash transition.

Implicit Reset Specifications. Figure 4 shows an example with write operations W1 to W4. The operations and how far they filled the block is denoted by the start/end position of the arrows. Page boundaries are depicted as dotted lines. A power failure in the explicit approach (i.e., what effectively happens) removes the hatched part at the end of the block up to the last dotted line.

The effect can be explained *alternatively* by first reversing the effect of the last two operations W3 and W4 and subsequently re-executing the W3 operation (denoted by W3') on the same inputs, choosing the error path that writes only $n = (\# \mathit{buf}) \downarrow \texttt{PAGE\_SIZE}$ bytes. This alternative specification requires to define *synchronized* states that are resilient against crashes, i.e., a retraction is not allowed in such a state: Fig. 5 shows how these can be captured in the write buffer by a `synchronized` predicate over the state, where $\mathit{page\text{-}aligned}(\mathit{block})$ holds iff $\# \mathit{block} \bmod \texttt{PAGE\_SIZE} = 0$. Clearly, the alternative trace that executes W1, W2 and W3' is possible by the specification of the `append` operation. In a synchronized state there is no additional effect of a crash that must be modeled explicitly, i.e., the crash predicate of the implicit specification of Fig. 5 is just identity. In states outside of the domain of the crash predicate a crash can not occur.

By making the crash predicate partial, we mark the states of interest in which we *want* to consider crash transitions. Here, we use just the synchronized states, while in the general theory it is also possible to use a superset. We have to ensure that retracting operations and re-executing one operation actually targets a state in the domain of the (now partial) crash predicate. Informally, this is guaranteed when it can be proved that operations fall into two classes: *retractable* operations like W4, where a crash has the same effect before and after the operation, and *completable* operations like W3, which have an execution that leads to a state in the domain of the crash predicate.

We use both the implicit as well as the explicit specification as shown by Fig. 6. First, we abstract the write buffer implementation to the explicit specification with a normal refinement step (Thm. 1, Sec. 4). Then we introduce the implicit specification in a separate refinement step (Thm. 2, Sec. 4). In the semantics we define next, the retraction transition is then implicitly propagated upwards through the remaining refinement hierarchy automatically (Thm. 3, Sec. 5). Therefore only the (now trivial) crash predicate needs to be expressed on each layer of abstraction.

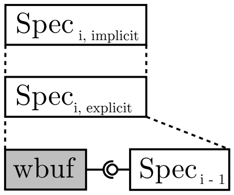

Fig. 6: Introducing Implicit Specifications

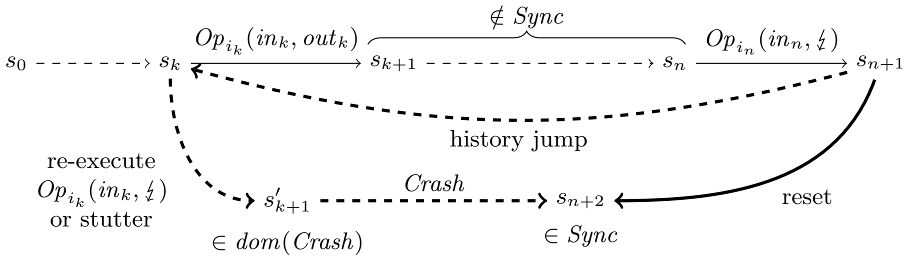

Fig. 7: Constructing a reset transition $s_{n+1} \longrightarrow s_{n+2}$.

3 Components with Power Cuts

Systems considered in this work are hierarchically composed of encapsulated components, which are formally presented in terms of data types [12,7] extended by a specification of the effect of a power cut and subsequent recovery.

Definition 1 (Component). *A component $C = (S, In, Out, Init, Sync, Crash, (Op_i)_{i \in I})$ consists of a set of states S, inputs In and outputs Out, initial states $Init$, synchronized states $Sync$ with $Init \subseteq Sync \subseteq S$, a relation $Crash \subseteq S \times Sync$ with $Sync \subseteq dom(Crash)$ describing the effect of a crash including its subsequent recovery,*[4] *and regular operations $Op_i \subseteq In \times S \times S \times (Out \uplus \{\lightning\})$. The value $\lightning$ signifies that the operation was interrupted by a power cut.*

Operations are defined by programs that modify the state and compute an output from an input. Implementation programs may call operations of another (sub-)component as detailed in Sec. 5. A small-step semantics of the programs is given in [10]. Here, we abstract program runs to a relation Op_i between initial and final states. The crash and synchronized relation are given syntactically by the `crash` and `synchronized` predicate in Fig. 3 and Fig. 5.

A complete program run starting in state s with input in, finishing in state s' with output out is written $s \xrightarrow{Op_i(in,out)} s'$, which is equivalent to $(in, s, s', out) \in Op_i$, using the induced relation $Op_i(in, out) \subseteq S \times S$ as a label. Abbreviation $(s, s') \in Op_i(in)$ holds if there is any $out \neq \lightning$ such that $(s, s') \in Op_i(in, out)$.

An incomplete run where the computation is interrupted by a power cut in an intermediate state s' (and the operation does not return a result) results in tuple $(in, s, s', \lightning) \in Op_i$, written as $s \xrightarrow{Op_i(in,\lightning)} s'$, again using $Op_i(in, \lightning) \subseteq S \times S$ as a label. It is reasonable to assume that a crash can happen in initial as well as final states, i.e., we assume $Id_S \subseteq Op_i(in, \lightning)$ for the identity relation resp. $Op_i(in) \subseteq Op_i(in, \lightning)$. Interrupted steps in a run are followed by steps $s' \xrightarrow{\text{reset}} s''$ (detailed below), that model the *effect* of a power cut and its subsequent recovery.

[4] In the actual models recovery is a separate *operation* that runs directly after a crash and tries to restore the state. To keep the presentation brief, we combine the crash and recovery into one transition here.

The semantics of components is a set of runs, which are finite or infinite sequences of labeled transitions of these three kinds, which generalizes data types as used in Z [22] that have regular transitions $s \xrightarrow{Op_i(in,out)} s'$ only.

Definition 2 (Runs). *A run of the component C is given by a sequence of labeled state transitions $s_0 \xrightarrow{l_0} s_1 \xrightarrow{l_1} \cdots$ that starts in an initial state with $s_0 \in Init$ and consist of fragments for each (non-interrupted) state s_n:*

$$s_n \xrightarrow{Op_{i_n}(in_n, out_n)} s_{n+1} \qquad \text{or} \qquad s_n \xrightarrow{Op_{i_n}(in_n, \lightning)} s_{n+1} \xrightarrow{\text{reset}} s_{n+2},$$

where $s_{n+1} \xrightarrow{\text{reset}} s_{n+2}$ picks an earlier state s_k from this run, optionally re-executes the corresponding k-th operation partially ($\lightning$ output), and applies the residual effect of crash & recovery, i.e., there is k with $k \leq n+1$ and s'_{k+1} s.t.:

- $s_{k'} \notin Sync$ for all k' with $k < k' \leq n+1$,
- $s_k \xrightarrow{Op_{i_k}(in, \lightning)} s'_{k+1}$ and $k < n+1$ or $s_k = s'_{k+1}$,
- $(s'_{k+1}, s_{n+2}) \in Crash$

Figure 7 depicts how a transition $s_{n+1} \xrightarrow{\text{reset}} s_{n+2}$ (arrow $\longrightarrow$) is constructed by these three constituents (arrows $\dashrightarrow$).

We point out some aspects of Def. 2: Re-execution is optional and only permitted when at least one operation had been retracted by the jump ($k \neq n+1$). The state s_{n+2} will be synchronized as $Crash \subseteq S \times Sync$, implying that another crash does not go back further in the history. State s'_{k+1} must fall into the domain of *Crash*. This corresponds to the intuition that a power cut can be *observed in* or needs to be *considered in* states in $dom(Crash)$. Expressing the *Crash*-predicate on a selected subset of states is easier for the given component and its clients as we have motivated with Fig. 5. Retracting operations implies the existence of a different run without a jump that ends in the same state s_{n+2}:

$$s_0 \to \cdots \to s_k \xrightarrow{Op_{i_k}(in_k, \lightning)} s_{k'+1} \xrightarrow{Crash} s_{n+2}. \tag{1}$$

A component where all states are synchronized ($Sync = S$) neither retracts nor re-executes operations. This view is used for the lowest level of specification, where the distinction between volatile and persistent memory is explicit, and the effect of a power cut is expressed as just forgetting data in volatile memory.

4 Crash-Safe Refinement

The observable behavior of a run is the sequence of its labels. Refinement is defined based on preserving observable behavior:

Definition 3 (Refinement). *A component C refines a component A, written $A \sqsubseteq C$, iff they have the same input and output set and the same index set of operations and for each run $cs_0 \xrightarrow{l_0} cs_1 \xrightarrow{l_1} \cdots$ of C there is a matching run $as_0 \xrightarrow{l_0} as_1 \xrightarrow{l_1} \cdots$ of A with the same labels.*

With these definitions it is now possible to express the correctness and crash-safety criterion we propose for file systems.

Definition 4 (Quasi Sequential Crash Consistency). *A file system is quasi sequentially crash consistent, iff it refines the POSIX component given in [11] augmented with synchronized states reached by successful calls to* `fsync` *or* `sync`*. The crash predicate discards open file handles and deletes orphaned files [10].*

Since our POSIX specification is a component as by Def. 1 its reset is allowed to retract operations, however, never across a successful call to `fsync` or `sync`. "Quasi" signifies that one re-execution is allowed, which is not allowed in Bornholt's definition of sequential crash consistency [3]. The Flashix file system is developed via incremental refinement of the POSIX component.

In general, a refinement step can change data representation as well as change the view of a crash, since only the observable behavior must be preserved. The generality of having both changes in abstraction is only needed for a uniform definition of refinement. In practice, refinements either change data representation, or the specification of a crash individually—several refinement steps can be combined transitively if needed. The following two subsections therefore consider the two types of refinement separately. Like in data refinement, refinement is typically proved using forward simulations. New proof obligations result from steps $s_n \xrightarrow{Op_j(in,\lightning)} s_{n+1} \xrightarrow{\text{reset}} s_{n+2}$, therefore the proofs focus on these transitions.

4.1 Data Refinement & Propagation of Jumps

The proof obligations for changing data representation are just slightly more complex than data refinement. We denote with $R_1 \fatsemi R_2$ the composition of two relations R_1 and R_2 and with $D \lhd R$ and $R \rhd D$ the domain resp. range restriction of the binary relation R to the set D.

Theorem 1 (Data Refinement by Forward Simulation). *A refinement $A \sqsubseteq C$ is implied by a forward simulation $R \subseteq AS \times CS$ satisfying (for all $i \in I, in \in In, out \in Out$)*

1. $CInit \subseteq AInit \fatsemi R$ (initialization)
2. $R \fatsemi COp_i(in, out) \subseteq AOp_i(in, out) \fatsemi R$ (correctness)
3. $ASync \fatsemi R \subseteq CSync$ (synchronization)
4. $R \fatsemi COp_i(in, \lightning) \fatsemi CCrash \subseteq AOp_i(in, \lightning) \fatsemi ACrash \fatsemi R$ (crash)

The synchronization condition states that fewer states of component A are synchronized and the crash condition abstracts the remaining effect of a power cut.

Proof (of Thm. 1). The proof composes commuting diagrams as usual, starting with two related initial states given by 1. For regular transitions, proof obligation 2. gives the relevant commuting diagram. A history jump of component C is mapped to a history jump over the same number of operations in A. Condition 3. ensures that each unsynchronized state retracted by C can be retracted by A as well. Condition 4. commutes either the interrupted operation (when the jump is empty) or the re-execution jointly with the subsequent crash and recovery. □

$$\begin{tikzcd}
s_0 \arrow[r, "Op(in, out)"] \arrow[rr, bend right, "Crash"'] & s_1 \arrow[r, "Crash"] & s_2
\end{tikzcd}$$

Fig. 8: Retractable before *Crash*

$$\begin{tikzcd}
s_0 \arrow[r, "Op(in, out)"] \arrow[dr, "Op(in, \lightning)"'] & s_1 \arrow[r, "Crash"] & s_2 \\
& \boldsymbol{s'_1} \arrow[ur, "Crash"'] & \\
& \in D &
\end{tikzcd}$$

Fig. 9: *D*-Completable before *Crash*

4.2 Crash Refinement & Introduction of Jumps

Incrementally introducing history jumps is the second kind of refinement. It assumes that the data structures and operations are the same on both levels. The basic idea is to move parts of a power cut from a *Crash* transition to the jump transition by looking at the history fragment $s_n \xrightarrow{Op_i(in,out)} s_{n+1} \xrightarrow{Crash} s_{n+2}$ of the component before a crash transition and construct a different explanation of how the component ended up in s_{n+2}. This construction yields an alternative intermediate state s'_{n+1} from a set $D \subseteq dom(Crash)$, allowing us to simplify the crash transition to the relation $D \lhd Crash$ as in Fig. 5 for $D = page\text{-}aligned$.

Definition 5 (Retractable before *Crash***).** *A transition $s_0 \to s_1$ is retractable before Crash, iff every state s_2 with $(s_1, s_2) \in Crash$, also satisfies $(s_0, s_2) \in Crash$.*

If an execution step is retractable before *Crash*, it did not have any immediate permanent effect and we can ignore that it ever took place directly before a crash happened. Figure 8 depicts this alternative execution in bold. This does not mean that the execution will never have a permanent effect. Any of the subsequent operations may very well persist the data of previous operations. In the example, the transition W4 in Fig. 4 is retractable before the crash of Fig. 3 that sets the state to $block \downarrow \texttt{PAGE\_SIZE}$.

Definition 6 (*D*-Completable before *Crash***).** *A transition $s_0 \xrightarrow{Op(in,out)} s_1$ with $out \in Out \uplus \{\lightning\}$ of an operation Op is called D-completable before Crash for some set $D \subseteq dom(Crash)$, iff for every state s_2 with $(s_1, s_2) \in Crash$ there is an execution $s_0 \xrightarrow{Op(in,\lightning)} s'_1$ with $s'_1 \in D$ and $(s'_1, s_2) \in Crash$.*

If a transition is *D*-completable before *Crash* it is possible to construct an alternative partial execution that ended in a *D*-state without any difference after a crash. Figure 9 also depicts this alternative execution in bold. Transition W3 in Fig. 4 for example is *page-aligned*-completable before the crash of Fig. 3 to $block \downarrow \texttt{PAGE\_SIZE}$ and the depicted re-execution W3' is the alternative.

Definition 7 (*D*-Retractable before *Crash***).** *An operation Op is D-retractable before Crash for some set $D \subseteq dom(Crash)$, iff every transition of Op is either retractable or D-completable before Crash, or equivalently:*

$$Op(in) \fatsemi Crash \subseteq (Id_S \cup (Op(in, \lightning) \rhd D)) \fatsemi Crash \qquad \textit{for all } in \in In$$

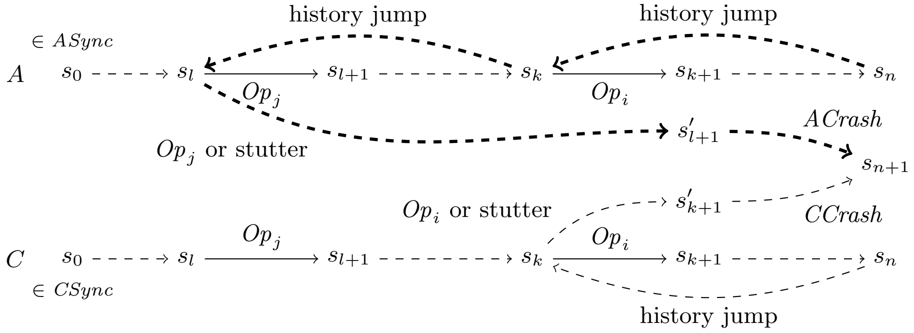

Fig. 10: From Implicit to the Explicit Reset Specification

This lifts Defs. 5 and 6 to the level of one operation. For example, `append` of the write buffer is *page*-*aligned*-retractable before *block* ↓ `PAGE_SIZE`, since one can either retract the operation if it did not cross a page boundary or execute it in such a way that it writes up to the last page boundary.

The following theorem can be used to abstract an explicit crash specification as part of C to an implicit crash specification by A.

Theorem 2 (Implicit to Explicit Refinement). *The refinement $A \sqsubseteq C$ for*

$$C = (S, \mathit{Init}, \mathit{In}, \mathit{Out},\ \mathit{CSync},\ \mathit{CCrash},\ (\mathit{Op}_i)_{i \in I}) \qquad \textit{and}$$
$$A = (S, \mathit{Init}, \mathit{In}, \mathit{Out},\ \mathit{ASync},\ \mathit{ACrash},\ (\mathit{Op}_i)_{i \in I})$$

with atomic operations Op_i for all $i \in I$ follows from

1. *$dom(ACrash) \lhd CCrash \subseteq ACrash$*
2. *$ASync \subseteq CSync$*
3. *Op_i is $dom(ACrash)$-retractable before $CCrash$ for all $i \in I$*

We usually apply Thm. 2 with crash predicates that satisfy the (stronger) condition $dom(ACrash) \subset dom(CCrash)$ to strengthen the crash transition, i.e., a crash can happen in fewer states of component A than of component C and is therefore simpler to express. This is compensated by farther jumps on the history of A in comparison to those of C as by 2. A has less synchronized states. These jumps are then easily propagated upwards over abstractions with Thm. 1.

Proof (of Thm. 2). We choose the run of C as the run of A and focus on the transitions of a power cut. Figure 10 depicts the situation before the power cut (omitting input and output labels), starting in a state s_0 where both C and A are synchronized. Such a state exists because at least in the initial state and after every power cut *both* components are synchronized. The three parts of the power cut transition of C are depicted in the figure starting in state s_n and ending in s_{n+1}. We construct a matching transition of A, depicted by arrows $\dashrightarrow$ in the figure. All operations that C retracts are also retracted by A (*history jump* from s_n to s_k). However, the history jump transition might be farther still (*history jump* from s_k to s_l). The idea is to determine a state $s'_{l+1} \in dom(ACrash)$ of component A with the properties shown in the figure: there is an additional second jump backwards to s_l and a re-execution that yields s'_{l+1}. The construction considers the *run* $s_0 \dashrightarrow s_k \xrightarrow{Op_i \text{ or stutter}} s'_{k+1} \xrightarrow{CCrash} s_{n+1}$ of C implied by (1) of Sec. 3 that yields s'_{k+1}.

The existence of A's history jump and re-execution is proven by induction over k. If the sequence is empty ($k = 0$ and the transition stutters), then there is only a transition $s_0 \xrightarrow{CCrash} s_{n+1}$ starting from state s_0 with $s_0 \in ASync$. By proof obligation 1., $s_0 \xrightarrow{ACrash} s_{n+1}$ is the matching run of A: the additional history jump and re-execution transitions stutter. Otherwise, Op_i is $dom(ACrash)$-retractable before $CCrash$ and therefore the transition $s_k \xrightarrow{Op_i} s'_{k+1}$ is either:

- Retractable before $CCrash$ and therefore $s_0 \dashrightarrow s_k \xrightarrow{CCrash} s_{n+1}$ is also a valid run. The induction hypothesis gives a matching run of A and a history jump over m operations for this sequence. The history jump for the original sequence then is over $m + 1$ operations and we take the re-execution from the induction hypothesis which ends in the state s_{n+1}.
- $dom(ACrash)$-completable before $CCrash$ and $s_k \xrightarrow{Op_i} s''_{k+1} \xrightarrow{CCrash} s_{n+1}$ holds for some state $s''_{k+1} \in dom(ACrash)$. We choose $s'_{l+1} = s''_{k+1}$ and by proof obligation 1., $s_0 \dashrightarrow s_k \xrightarrow{Op_i} s'_{l+1} \xrightarrow{ACrash} s_{n+1}$ is a re-execution of A with $s'_{l+1} \in dom(ACrash)$ and the history jump stutters. □

5 Component Hierarchies & Substitution

This section defines components $M(A)$ that use a subcomponent A (see Fig. 1), underlying several limitations to confine communication between M and A to the interface. Hierarchies allow us to split off a part M of the entire implementation and verify it based on a (possibly very abstract) component A. A can then be refined separately, without jeopardizing the correctness and crash-safety of M. This facilitates modular and incremental development of a large system.

Intuitively, M has volatile state only and the entire persisted state resides in its subcomponent A. Combined states of $M(A)$ are written $ms \oplus as$.

Definition 8 (Hierarchies). *The component* $M(A) = (MS \times AS, In, Out, MInit \times AInit, MS \times ASync, MCrash \times ACrash, (MOp_i)_{i \in I})$ *combines the state space of* M *and of its subcomponent* A*. The state of* A *is hidden from* M *(information hiding) and the interaction with* A *is accomplished via synchronous calls to* A*'s operations in the programs* MOp_i *of* M *and observation of their inputs and outputs. The crash on the* M *part of the state must be arbitrary, i.e.,* $MCrash = MS \times MS$*, and* $M(A)$ *is synchronized if and only if* A *is.*

Refinement is compatible with hierarchical composition, i.e., correctness and crash-safety of a component is preserved by substitution of its subcomponents.

Theorem 3 (Substitution). *If $CSync = CS$ and $A \sqsubseteq C$, then $M(A) \sqsubseteq M(C)$.*

The condition $CSync = CS$ states that every C state is synchronized, i.e., there are no backward jumps and no re-execution in C and likewise in the combined $M(C)$. The Theorem is applicable in practice, because we can substitute implementation machines, which are always synchronized, bottom-up.

Proof (of Thm. 3). Given a fixed, arbitrary run of $M(C)$ we derive a matching run of $M(A)$ with the same labels to satisfy Def. 3 in several steps: From each $M(C)$-transition, we extract the fine-grained steps of C corresponding to the calls of its operations. Concatenating these gives a run of C, which can be mapped to one of A *as a whole* by the assumption $A \sqsubseteq C$. Finally, the A run can be integrated back with M. The reset transitions reduces to two helper lemmas. Specifically, if C is an implementation component, then Lem. 1 maps the $M(C)$ crash to the C one.

Lemma 1. *If $ms \oplus cs \xrightarrow{\text{reset}} ms' \oplus cs'$, then $cs \xrightarrow{\text{reset}} cs'$ holds for all ms, ms'*

The converse (Lem. 2) lifts the matching A reset from as to as' back into the context.

Lemma 2. *If $as \xrightarrow{\text{reset}} as'$, then $ms \oplus as \xrightarrow{\text{reset}} ms' \oplus as'$ holds for all ms, ms'.*

Lemma 1 is trivial: the $M(C)$ reset transition has no back-jumps and is thus equivalent to $(cs, cs') \in CCrash$ by the restrictions of component composition of Def. 8. Note that in general, compositions $M(C)$ retract operations at a coarser granularity than C can do on its own, i.e., a reset of $M(C)$ cannot be explained with the help of a C-reset in the presence of back-jumps.

Lemma 2 guarantees that the back-jump induced by the reset transition of the abstract A is permitted by the semantics of $M(A)$. The proof can be followed alongside Fig. 11 to establish arrows $\longrightarrow$ from the given arrows $\dashrightarrow$. The first line shows the big-step semantics of $M(A)$ and the second line extracts the small-step semantics of the one operation MOp_i ($ms_0 \oplus as_0 = ms^k \oplus as^k$ and $ms_m \oplus as_m = ms^{k+1} \oplus as^{k+1}$). The history jump of A targets a state as_l in the middle of a previously execution operation MOp_i. The reset of $M(A)$ retracts to the state $ms^k \oplus as^k$ right before this call and then reaches the combined intermediate state $ms_l \oplus as_l$ by partial re-execution of the MOp_i potentially including a partial AOp_j from the A reset. In the resulting state $ms_l' \oplus as_l'$, the A crash is known and the M crash admits any desired successor anyway. □

6 *Crash*-neutrality

The proof obligations 4. of Thm. 1 consider intermediate states. In this section we adapt the criterion of *Crash*-neutrality from our previous work [10], allowing us to consider initial and final states of operations only. This reduces the difficulty and size of proofs by enabling standard techniques from sequential verification.

Informally, *Crash*-neutrality asserts that for any intermediate state a completion exists that does not modify the persistent memory any further. A component is *Crash*-neutral if all its operations are.

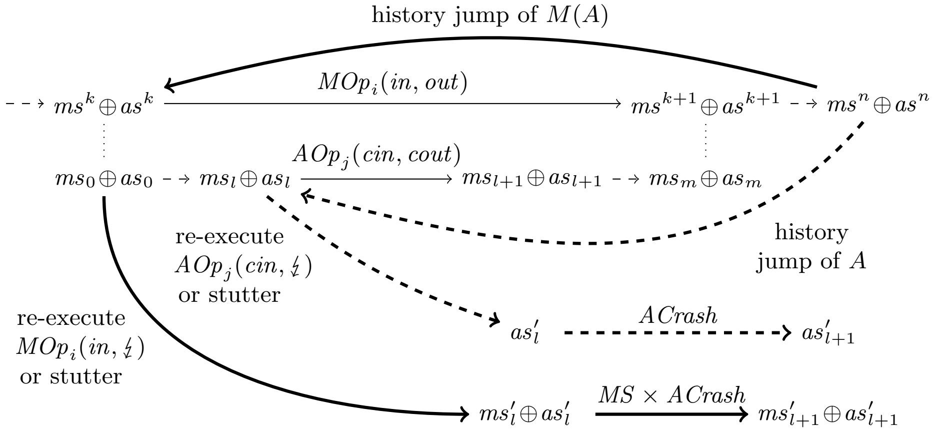

Fig. 11: Mapping a reset transition of a subcomponent A to one of $M(A)$.

Definition 9 (*Crash*-**Neutrality**). *An operation Op is Crash-neutral if every partial execution* $s \xrightarrow{Op(in, \lightning)} s'$ *with* $s' \in dom(Crash)$*, has a completion that terminates in a state* s'' *with the following property: for every state* s_0 *with* $(s', s_0) \in Crash$ *then* $(s'', s_0) \in Crash$ *holds, too.*

A useful shorthand to proving *Crash*-neutrality of $M(C)$ is given by the following lemmata. Its basic insight is that an operation is *Crash*-neutral if every small step of its program is *Crash*-neutral. Since all steps of M are either calls to C or just in-memory, it remains to ensure that C is *Crash*-neutral:

Lemma 3 (*Crash*-**Neutrality of** $M(C)$). *If C is Crash-neutral and all operation of M terminate, then $M(C)$ is Crash-neutral.* □

Lemma 4 (*Crash*-**Neutrality of atomic** C). *If every operation of C is atomic, then Crash-neutrality of C can be characterized by*

$$Crash \subseteq COp_i(in) \fatsemi Crash \qquad \textit{for all } i \in I \textit{ and } in \in In \qquad \square$$

The `append` operation of the write buffer (see Fig. 3) is *Crash*-neutral, we simply choose $n = 0$ as the number of bytes written. With *Crash*-neutrality Thm. 1 can be reformulated such that only reasoning about initial and final states is necessary.

Theorem 4 (*Crash*-**Neutral Data Refinement by Forward Simulation**). *A refinement $A \sqsubseteq C$ for a CCrash-neutral component C is implied by a forward simulation $R \subseteq AS \times CS$ satisfying 1.–3. of Thm. 1 and*

4′. $R \fatsemi CCrash \subseteq ACrash \fatsemi R$ (crash)

Proof. We consider the transition sequence $cs \xrightarrow{Op_i(in, \lightning)} cs' \xrightarrow{\text{reset}} cs''$. If the history jump transition (and therefore the re-execute) stutter, we complete the operation $Op_i(in, \lightning)$ by *CCrash*-neutrality and are still able to crash to cs'' afterwards. If we have a history jump, we complete the re-execution transition by *CCrash*-neutrality and are able to crash to cs'' afterwards. All relevant transitions for a forward simulation are explained by complete executions and we can use proof obligation 2. of Thm. 1 to find the matching abstract transition. □

7 Related Work

We focus on techniques for the verification of crash behavior, comparison of Flashix to related efforts can be found in [11,18,9] and Lali's summary [14].

Bornholt et al. [3] define crash consistency models for file systems, based on operations that produce (potentially many) update events. A crash is then expressed by taking a prefix of the update events. The difference between their definition of sequential crash consistency [3, Def. 5] and quasi sequential crash consistency (Def. 4) is that we allow a re-execution that might produce different events and not just (a reordering of) a prefix, and we allow an additional effect of the crash afterwards. Update events have the same drawback as the explicit specification provided in Sec. 2. Their notion of crash consistency also omits orphaned files. Follow-up work [19] integrates crash-safety with simulation conditions similar to the ones we have given previously in [10]. This paper clarifies the adequacy of the simulation conditions wrt. a component semantics, which is not discussed in [19]. In particular, hierarchical composition of components has subtle effects of how exactly a crash and recovery must be organized that substitution is possible (Thm. 3).

Write-back caches where a crash affects multiple operations is discussed in [2,5,19], too. The abstract model of [2] keeps an explicit history back to the most recent flush as a list of higher-order state transformers. It is proved that the implementation of `sync` correlates to reducing the history to produce a current state. Chen's thesis [5] discusses a specification methodology of write-back caches that are not order-preserving. It is based on explicitly rewriting histories, although he lacks modular conditions as in Thm. 2.

In this paper as well as in [19] the intermediate steps of operations are summarized at the semantic level (as $Op(in, \lightning)$ resp. $f(s, x, sync = false)$). Ntzik et al. [16] as well as Chen et al. [6,4] have developed Hoare-style proof rules that establish a user-provided invariant called "crash condition" over the intermediate states of a program that serves as the precondition of recovery. The latter work has produced the FSCQ file system that is verified with Coq. Marić and Sprenger [15] model crashes by exceptions that are triggered nondeterministically in the write operations of the hardware model to verify a redundant storage system. We have addressed this issue by a fine-grained semantics of programs in [10] which computes the crash condition symbolically.

Re-execution of operations underlies the "recoverability" criterion of Koskinen and Yang [13] at the level of entire programs. Their approach can be recast in our notation such that $Op(in, \lightning) \fatsemi Crash$ establishes the precondition of the program, which can then be re-run to recover the intermediate state without runtime errors. Here, the purpose of re-execution is different: We use it as a specification mechanism to reach certain intermediate states.

8 Conclusion

In this paper we have defined an approach that facilitates the integration of order-preserving write-back caches into the hierarchical development of file systems. It is possible to verify functional correctness and quasi sequential crash consistency modularly. This enables modular, large-scale verification, which would otherwise be unrealistic to perform and hard to maintain.

We have reinterpreted the behavior of a crash in terms of the system's operations, so that at each level of abstraction a backward jump (induced by a crash) does not need to be expressed as part of the state. This allowed us to propagate the reset specification implicitly upwards through a refinement hierarchy. Obviously, it is necessary to capture the effect semantically to do this.

We implemented support for component specifications of Def. 1 and generate the proof obligations in our interactive theorem prover KIV [8]. We mechanized the verification of the Flashix file system, which provides quasi sequential crash consistency. Previously, we performed a verification of write-back caching for the two components above the write buffer, where the sequence of operations is mostly part of the state. The second component then flushed the write-back cache at the end of its operations, which we can avoid now. With the theory of this paper, the verification of the write buffer itself requires just little extra effort, due to the switch from the implicit to the explicit reset specification. However, the *specifications* (not the implementations) of all components above the write buffer greatly benefited in terms of verification effort. For the two abstractions directly above the write buffer we report a decrease of 40% resp. 17% of user interactions in the proofs (from 500 to 300 and from 1270 to 1050). Flashix is now significantly faster and more space efficient, due to fewer flushes.

The theory in this paper should be applicable to other file systems and achieve similar results, since all journaling and log-structured file systems [17,21] feature comparable write-back caches.

In future work, we plan to extend the theory to non-order-preserving caches by allowing commutations of operations.

References

1. The Open Group Base Specifications Issue 7, IEEE Std 1003.1, 2013 Edition. The IEEE and The Open Group, 2013.
2. S. Amani and T. Murray. Specifying a Realistic File System. In Proc. Workshop on *Models for Formal Analysis of Real Systems*, volume 196 of *Electronic Proc. in Theoretical Computer Science*, pages 1–9. Open Publishing Association, 2015.
3. J. Bornholt, A. Kaufmann, J. Li, A. Krishnamurthy, E. Torlak, and X. Wang. Specifying and checking file system crash-consistency models. In *Proc. of ASPLOS*, pages 83–98. ACM, 2016.

4. T. Chajed, H. Chen, A. Chlipala, M. F. Kaashoek, N. Zeldovich, and D. Ziegler. Certifying a file system using crash hoare logic: correctness in the presence of crashes. *Communications of the ACM*, 60(4):75–84, 2017.
5. H. Chen. *Certifying a crash-safe file system.* PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, United States, 2016.
6. H. Chen, D. Ziegler, A. Chlipala, N. Zeldovich, and M. F. Kaashoek. Using Crash Hoare Logic for certifying the FSCQ file system. In *Proc. of the Symposium on Operating Systems Principles (SOSP).* ACM, 2015.
7. W.-P. de Roever and K. Engelhardt. *Data refinement: model-oriented proof methods and their comparison.* Cambridge University Press, 1998.
8. 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.
9. 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.
10. G. Ernst, J. Pfähler, G. Schellhorn, and W. Reif. Modular, crash-safe refinement for ASMs with submachines. *Science of Computer Programming (SCP)*, 2016.
11. 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, 2014.
12. J. He, C. A. R. Hoare, and J. W. Sanders. Data refinement refined. In *Proc. of the European Symposium on Programming (ESOP)*, pages 187–196. Springer, 1986.
13. E. Koskinen and J. Yang. Reducing crash recoverability to reachability. In *Proc. of Principles of Programming Languages (POPL)*, pages 97–108. ACM, 2016.
14. M. I. Lali. File system formalization: revisited. *International Journal of Advanced Computer Science*, 3(12):602–606, 2013.
15. O. Marić and C. Sprenger. Verification of a transactional memory manager under hardware failures and restarts. In *Proc. of Formal Methods (FM)*, volume 8442 of *LNCS*, pages 449–464. Springer, 2014.
16. G. Ntzik, P. da Rocha Pinto, and P. Gardner. Fault-tolerant resource reasoning. In *Proc. of the Asian Symposium on Programming Languages and Systems (APLAS)*, volume 9458 of *LNCS*, pages 169–188. Springer, 2015.
17. M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, 1992.
18. G. Schellhorn, G. Ernst, J. Pfähler, D. Haneberg, and W. Reif. Development of a verified flash file system. In *Proc. of Alloy, ASM, B, TLA, VDM, and Z (ABZ)*, volume 8477 of *LNCS*, pages 9–24. Springer, 2014. Invited Paper.
19. H. Sigurbjarnarson, J. Bornholt, E. Torlak, and X. Wang. Push-button verification of file systems via crash refinement. In *Symposium on Operating Systems Design and Implementation (OSDI).* USENIX Association, 2016.
20. H-W. Tseng, L. Grupp, and S. Swanson. Understanding the impact of power loss on flash memory. In *Proc. of the Design Automation Conference (DAC)*, pages 35–40. ACM, 2011.
21. S. C. Tweedie. Journaling the Linux ext2fs filesystem. In *The Fourth Annual Linux Expo*, 1998.
22. J. Woodcock and J. Davies. *Using Z: Specification, Proof and Refinement.* Prentice Hall, 1996.