Flashix: Results and Perspective

Jörg Pfähler, Stefan Bodenmüller, Gerhard Schellhorn, (Gidon Ernst)

Universität Augsburg

Overview

Institute for
Software & Systems
Engineering

1. Flash Memory and Flash File Systems
2. Results of Flashix I
3. Current Result: Integration of write-back Caches
4. Outlook: Concurrency

Motivation (I)

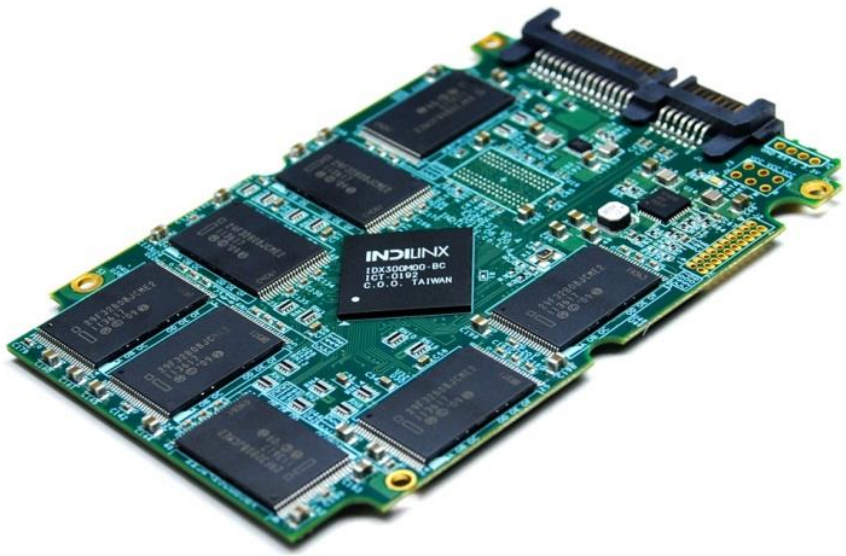
Flash Memory

- increasingly widespread use
- also in critical systems (server, aeronautics)

⊕ shock resistant
⊕ energy efficient
⊖ specific write characteristics
→ complex software

Motivation (II)

Institute for Software & Systems Engineering



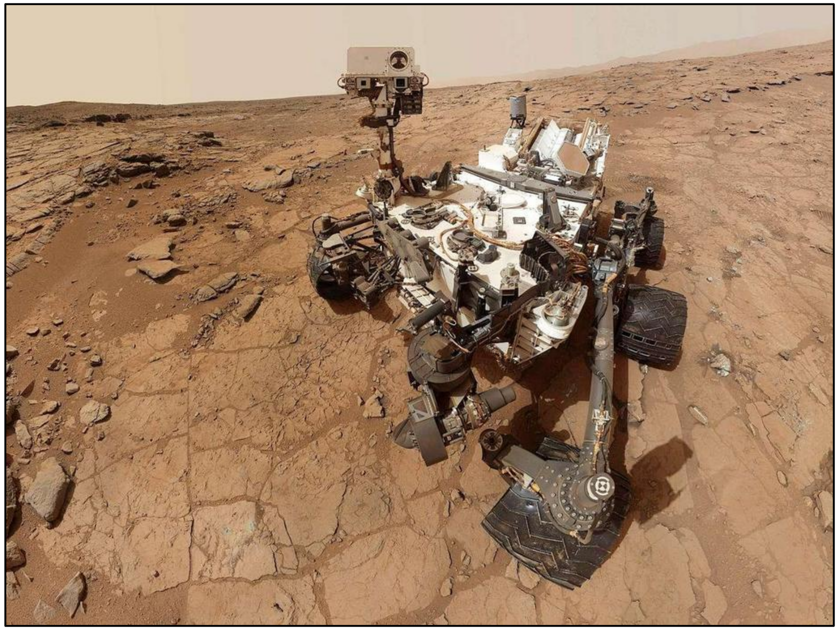
Indilinx Everest SATA 3.0 SSD platform specs:
- Dual core 400 MHz ARM
- 1 GB DDR3 RAM
- Up to 0,5 GB/s sequential read/write speed

Firmware errors

- Intel SSD 320: power loss leads to data corruption
- Crucial m4, Sandforce: drive not responding
- Samsung: crash during reactivation from sleep state

Motivation (III)

Institute for Software & Systems Engineering



Mars Rover *Spirit*
- Loss of communication
- Error in the file system implementation lead to repeated reboots
- [Reeves, Neilson 05]

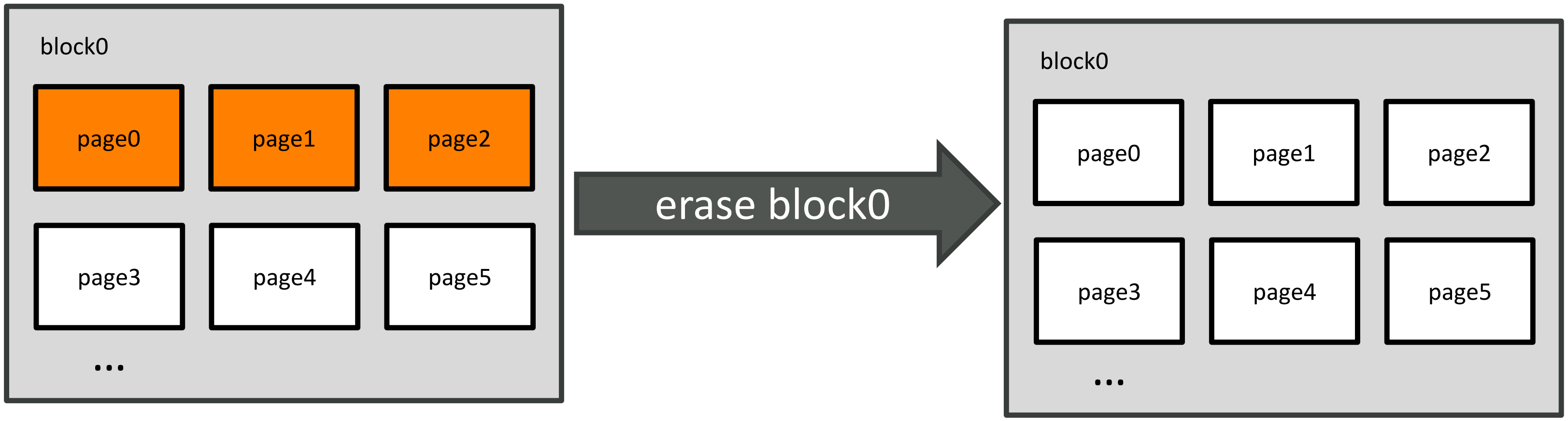
Mars Rover *Curiosity*
- Feb 27, March 16 2013: Safe Mode because of data corruption
- Switched to backup computer

- Pilot project of the Verification Grand Challenge:
  Develop a *formally verified state-of-the-art* flash file system
  [Rajeev Joshi und Gerard Holzmann 07]

Flash Memory (I)

- Operations
  - read page
  - write empty page (no in-place overwrite, only sequential)
  - erase block (expensive!)

Flash Memory (I)



- Operationen
  - read page
  - write empty page
  - erase block (expensive!)

Flash Memory (II)

Institute for
Software & Systems
Engineering

- Limited lifetime: 10^4 – 10^6 Erase-cycles
  - Distribute erase operations equally (Wear-Leveling)
- Out-of-place Updates
  - Mapping logical → physical erase blocks
  - Garbage collection
- SSDs, USB drives
  - Built-in Flash-Translation-Layer (FTL)
- Embedded
  - Specific filesystems (JFFS, YAFFS, UBIFS)

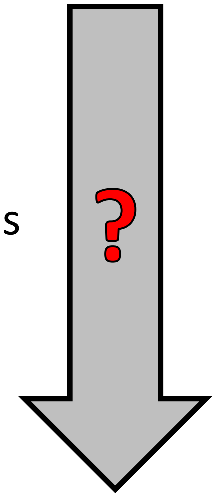
Flashix: System Boundaries

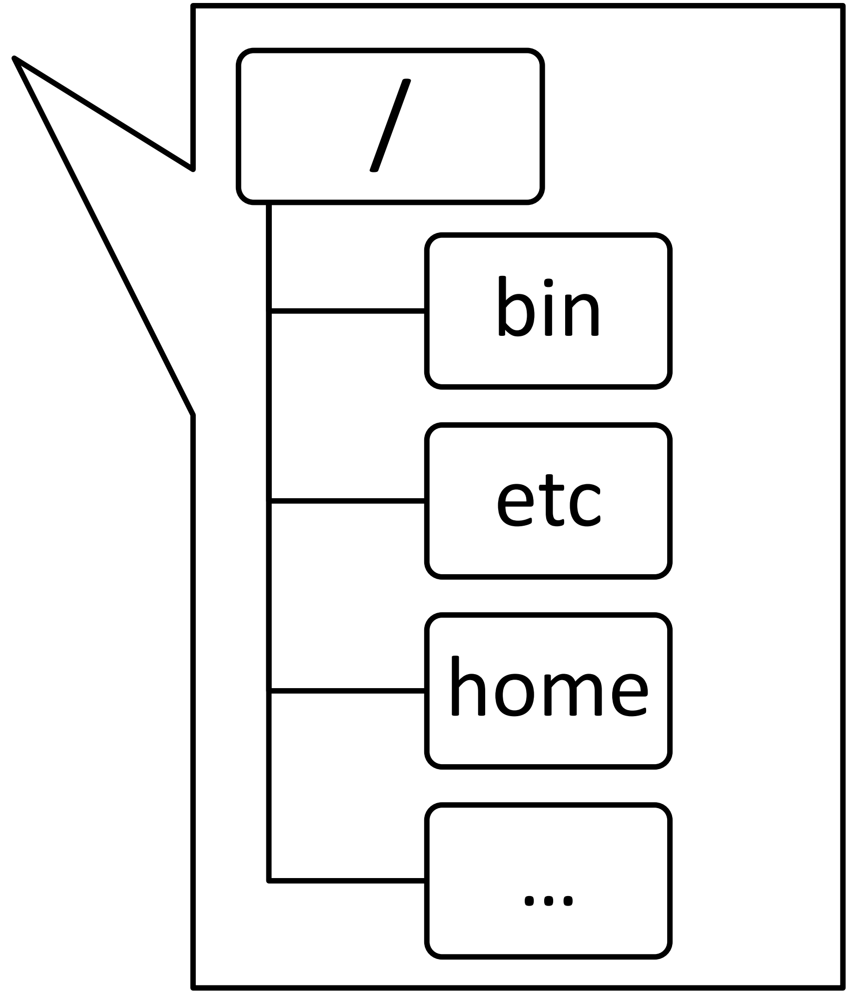
Institute for
Software & Systems
Engineering

POSIX

Flashix:

- Functional Correctness
- Crash-Safety



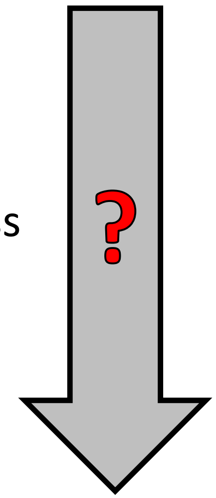


Flash driver

Flashix: System Boundaries

Institute for Software & Systems Engineering

POSIX

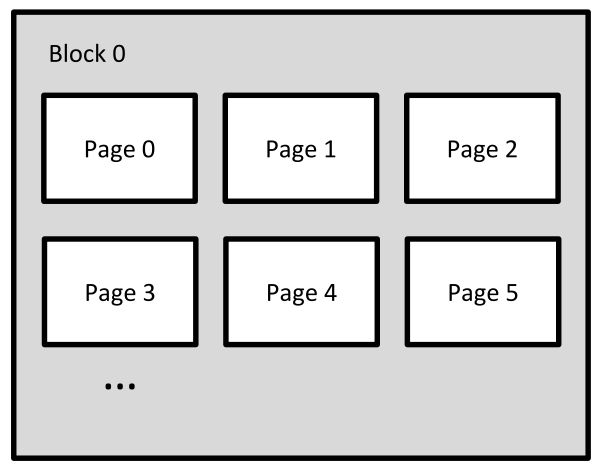


Flashix:

- Functional Correctness
- Crash-Safety

Flash driver

- Sequential writing of pages (no overwrite)
- Erasing whole blocks (slow, deteriorates memory)

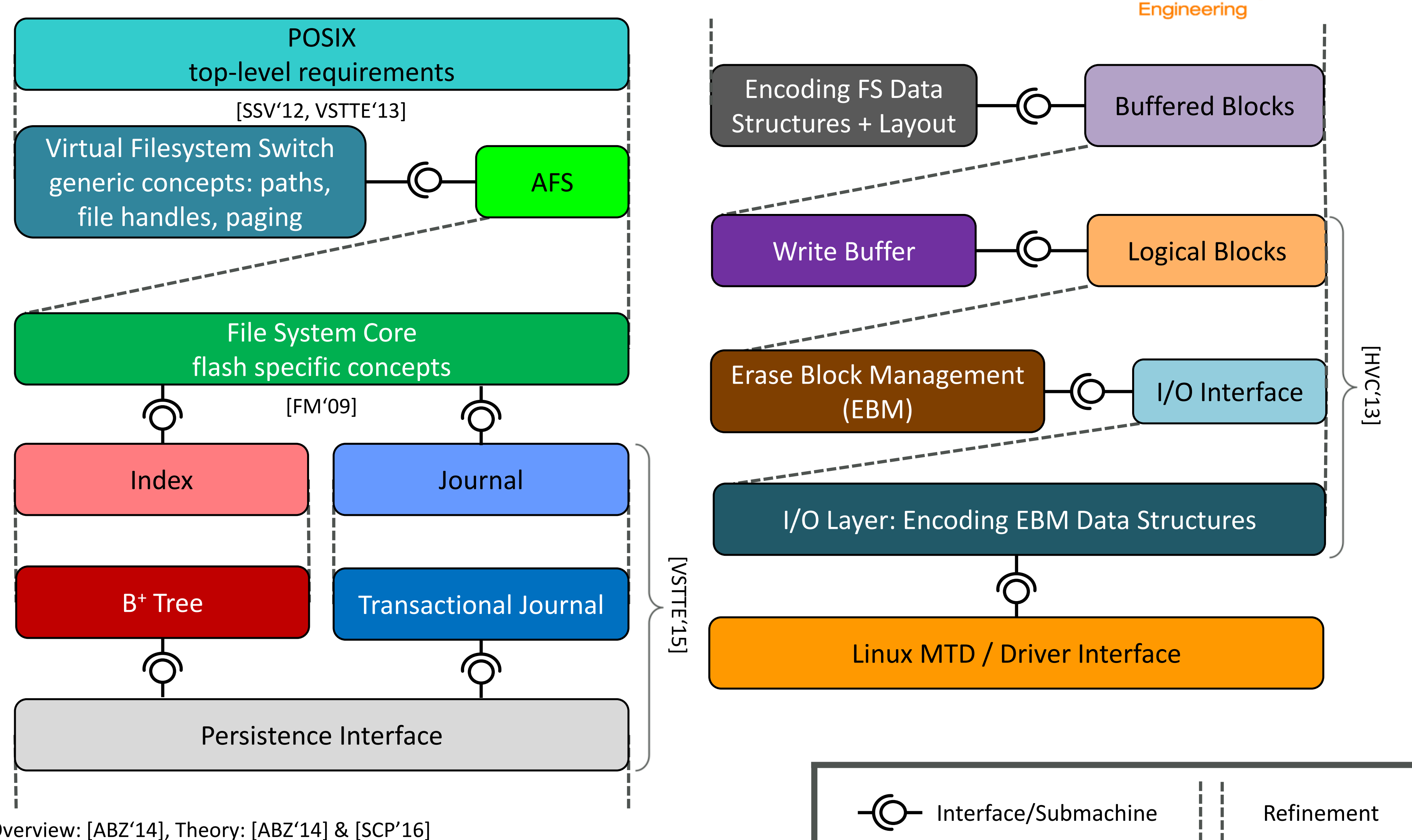


Overview

Institute for Software & Systems Engineering

1. Flash Memory and Flash File Systems
2. Results of Flashix I
3. Current Result: Integration of write-back Caches
4. Outlook: Concurrency

Models (simplified)



Overview: [ABZ'14], Theory: [ABZ'14] & [SCP'16]

Models: Highlights

- POSIX: very abstract, understandable specification (based on algebraic trees)
- Generic, filesystem-independent part similar to VFS in Linux
- Orphaned Files and Hardlinks are considered
- Journal-based implementation for crash-safety
- Garbage Collection and Wear-Leveling
- Efficient B^+-tree-based indexing
- Index on flash for efficient reboot
- Write-through Caches

Related:

- FSCQ [Chen et. al. 15]: no flash-specifics, generates Haskell code, verified with Coq
- Data61 (NICTA) [Keller eta al 14]: only middle part of the hierarchy considered, no crash-safety, verified code generator

Read: POSIX

Institute for Software & Systems Engineering

```
data asm specification

state variables
  root : tree[fid]
  fs   : fid ⇸ seq[byte]
  of   : fh  ⇸ (fid × pos)

operations
posix_read(fh; buf, len)
{ /* error handling omitted */
  let (fid, pos) = of[fh]

  choose n with n ≤ len ∧ pos + n ≤ # fs[fid] in
    len := n

  buf    := copy(fs[fid], pos, buf, 0, len)
  of[fh] := (fid, pos + len)
}

[…]
```

Read: VFS

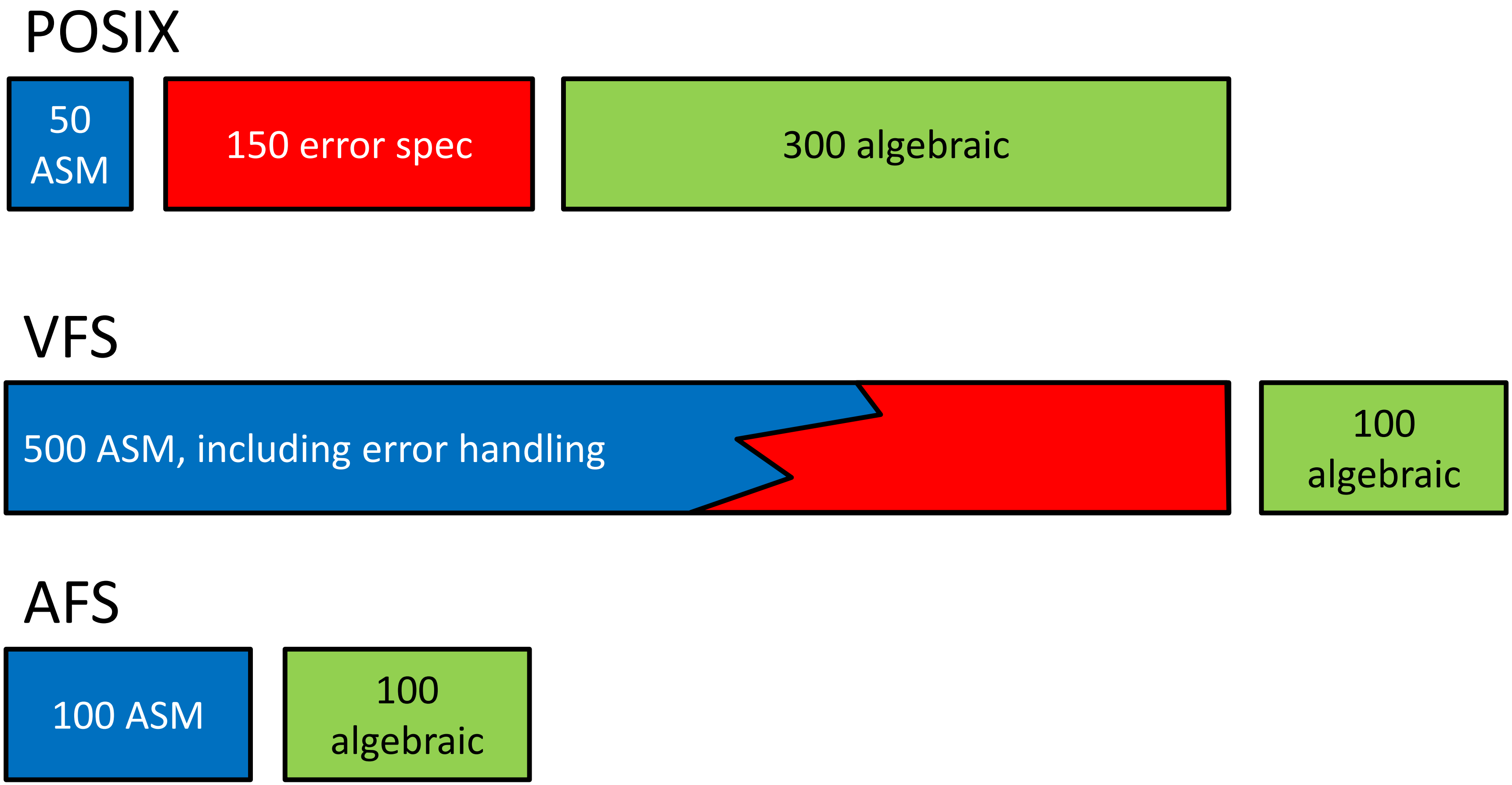
Institute for Software & Systems Engineering

```
vfs_read#(FD; BUF, N; ERR) {
 ERR := ESUCCESS;
 if ¬ FD ∈ OF
 then ERR := EBADFD
 else if   OF[FD].mode ≠ MODE_R
         ∧ OF[FD].mode ≠ MODE_RW
 then ERR := EBADFD
 else let INODE = [?] in {
  afs_iget#(OF[FD].ino; INODE, ERR);
  if ERR = ESUCCESS
  then {
   if INODE.directory
   then ERR := EISDIR
   else let START = OF[FD].pos,
            END   = OF[FD].pos + N,
            TOTAL = 0,
            DST   = 0 in
   if START ≤ INODE.size
   then {
    vfs_read_loop#;
    OF[FD].pos := START + TOTAL;
    N := TOTAL
   } else
    N := 0
  }
 }
}
```

```
vfs_read_loop# {
  let DONE = false, DST = DST in
  while ERR = ESUCCESS ∧ ¬ DONE do
    vfs_read_block#
}

vfs_read_block# {
 let PAGENO = (START + TOTAL) / PAGE_SIZE,
     OFFSET = (START + TOTAL) % PAGE_SIZE,
     PAGE   = emptypage
 in {
  let N = min(END         - (START + TOTAL),
              PAGE_SIZE   - OFFSET,
              INODE.size - (START + TOTAL))
  in
  if N ≠ 0 then {
   afs_readpage#(INODE.ino, PAGENO; PAGE, ERR);
   if ERR = ESUCCESS
   then {
    BUF := copy(load(PAGE),OFFSET,BUF,DST+TOTAL,N);
    TOTAL := TOTAL + N
    }
  } else {
    DONE := true
  }
 }
}
```

Size of Models (LOC)

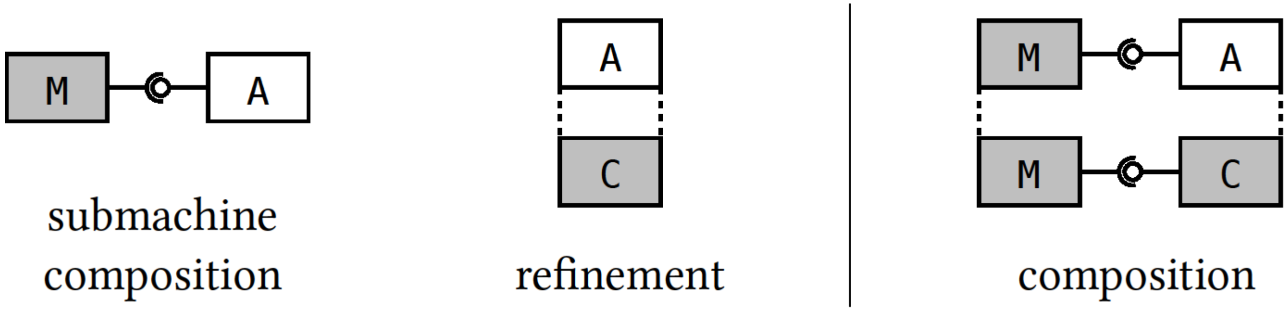


Theoretical Result: Submachines

Institute for
Software & Systems
Engineering

Theorem [SCP 16] : Submachine Refinement is compositional

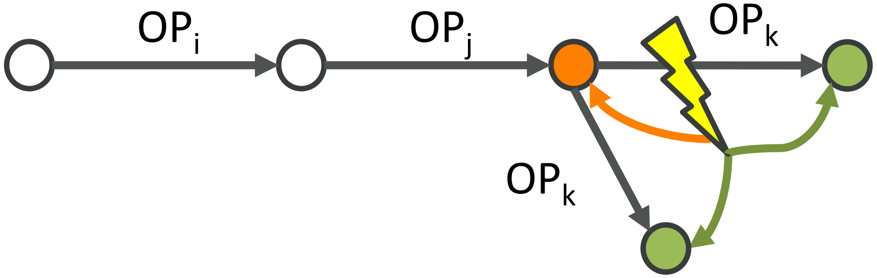
$$A \sqsubseteq C \rightarrow M(A) \sqsubseteq M(C)$$



Related:

- Simulations propagate [Engelhardt, deRoever]

Goal: Crash-Safety



Goal: A File System is **crash-safe** if a crash in the middle of an operation leads to a state that is *similar* to

a) the initial state of the operation

b) some final state of a run of the operation

where *similar* = equal after reboot.

Motivation for „similar“: open files handles are cleared = effect of reboot

Definition: Crash-Neutrality

Institute for Software & Systems Engineering

Definition: An atomic operation is **crash-neutral** if it has a („do nothing“) run such that a crash after the operation leads to the same state as the crash before the operation.

Motivation: operations on flash hardware always have a „do-nothing“ run, since the hardware can always refuse the operation

Proof Obligation:

$\quad pre(Op)(in, state)$

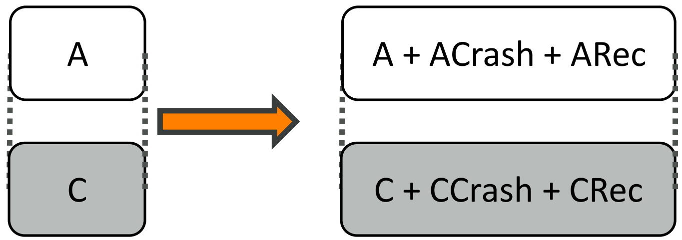
$\wedge\ Crash(state, state')$

$\rightarrow\ \langle\ Op\ (in; state; out)\ \rangle\ Crash(state, state')$

Crash-Safety: Refinement

Institute for Software & Systems Engineering

Refinement POs



Refinement + Crash POs

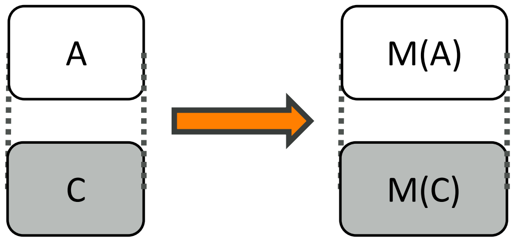
Theorem [Ernst et. al., SCP 16]:
If
- All operations of C are crash-neutral
- Refinement PO for each operation, including { Crash; Recovery }

then C is a crash-safe implementation of A, written $A \sqsubseteq_{cs} C$.

Main difficulties:
- Additional data structures and algorithms required for recovery (e.g. journals, persisted index structures, …)
- Additional Invariants for these data structures required
- Refinement proof for { Crash; Recovery } must ensure that the entire RAM state can be recovered

Crash-Safety: Submachines

Institute for Software & Systems Engineering



Theorem [Ernst et. al., SCP 16]:
Crash-Safe Submachine Refinement is compositional and transitive

- $A \sqsubseteq_{cs} C \rightarrow M(A) \sqsubseteq_{cs} M(C)$
- $A \sqsubseteq_{cs} B$ and $B \sqsubseteq_{cs} C \rightarrow A \sqsubseteq_{cs} C$

By transitivity of refinement we get:

$$\text{POSIX} \sqsubseteq_{cs} \text{VFS}(\ldots(\text{MTD}))$$

Related Work:

- Temporal extension of Hoare Logic to reason about all intermediate states [Chen et. al. 15]
- Model-checking all intermediate states [Koskinen et. al., POPL16]
- Crashes as exceptions [Maric and Sprenger, FM2014]

Models: Size & Effort

Institute for Software & Systems Engineering

- 21 models of 5 – 15 operations each
- 10 Refinements
- Models ASMs: 4k LoC
  algebraic: 10k LoC
- Ca. 3000 theorems to prove functional correctness, crash-safety and quality of wear-leveling

- Effort:
  - 2 PhDs
  - Σ individual problems < fully developed system
  - Good, stable interfaces are crucial, but difficult to achieve; in particular in the presence of errors and crashes

Design of Models (I)

Institute for Software & Systems Engineering

- Modularization is key to success
  - Design small abstract interfaces on many levels
  - Use extra refinement levels to capture key concepts
  - Horizontal structure: Use submachines!
- Middle-out strategy was key to bridge the wide gap between POSIX and Flash Interface

Design of Models (II)

- Use expressive data types + control constructs
  - (KIV's) version of ASMs allows abstract models as well as Code-like implementations
  - Do not use program counters for control structure
  - Expressive data types are helpful (various types of trees, streams, pointer structures with separation logic library in HOL).
  - Sometimes we would have liked even more expressiveness, e.g. dependent/predicative types.

Changing Models and Verification Support

Institute for Software & Systems Engineering

- Models are bound to change: modifications ripple through several models → great similarity to software refactoring
- Main reason for changes due to properly handling hardware failures and power cuts
- Do not verify too early: testing and simulation can help a lot! Better integration would help
- Support machines with crashes and generate VCs for crash-safe refinement -> less error-prone, faster refactoring
- Verification tool has to minimize redoing proofs:
  - Compute minimal set of affected proofs (Correctness Management)
  - Replaying proofs is common

Open issues and limitations of Flashix I

Institute for Software & Systems Engineering

- Verification of final C-code
  - Idea: Use VCC/VeriFast to prove 1:1-correspondence between C code and KIV-ASM annotated as ghost code
- Limitations:
  - Concurrency has not been considered
  - Limited use of write-back Caches
  - Special files (e.g. pipes, symbolic links) have been left out, but could be added orthogonally

Code Size & Performance

- C Code generated: 13k LoC
  manually: 1k LoC (integration)
- Runs on embedded board (with Linux)
- Scala Code available (requires Linux FUSE library):
  https://github.com/isse-augsburg/flashix

Overview

Institute for Software & Systems Engineering

1. Flash Memory and Flash File Systems
2. Results of Flashix I
3. Current Result: Integration of write-back Caches
4. Outlook: Concurrency

Caches in Flash File Systems

Institute for
Software & Systems
Engineering

- Flashix uses several caches: index, superblock, etc…
- Most are recoverable from data stored on flash
- These just need an invariant in proofs:

  Cache = recover(Flash)
- Invisible to the user of POSIX

- Other write-back Caches are visible to the user
  - Write-buffer
  - Inode/Page/Dentry-Cache in VFS (Future Work)

Flashix: Write Buffer (I)

Institute for
Software & Systems
Engineering

Flashix: Write Buffer (I)

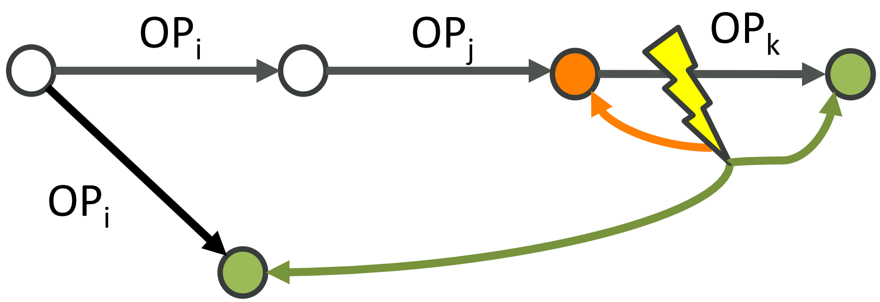
Institute for Software & Systems Engineering

- Low-Level View: Crash loses data in Cache
- Other higher-level Specifications (POSIX) cannot express this
- Therefore, Flashix I flushed the write buffer at the end of every AFS operation (wastes space, less efficient)

- High-Level View: Crash retracts several operations (blue and gray)

Weak Crash-Safety

Institute for Software & Systems Engineering



Definition: The implementation of a machine is **weak crash-safe** if a crash in the middle of an operation leads to a state that is *similar* to

a) the initial state of the operation

b) some final state of a run of **an earlier** operation

where *similar* = equal after reboot.

Flashix: Write Buffer

- High-Level View: Crash retracts several operations (blue and gray)

- Observation: Runs of operations are either
  - **retractable**: Crashing before or after the operation has the same effect (gray)
  - **completable**: there is an alternative run that leads to a synchronized state with empty cache (blue)

- **Synchronized States** are definable on abstract levels, e.g. POSIX: every state after fsync

Idea: Weak Crash-Safety by Refinement

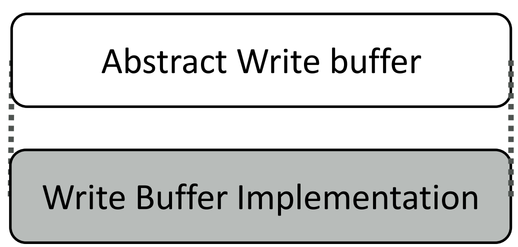
Institute for
Software & Systems
Engineering

- Machines with synchronized states *Sync* ⊆ *S*
  and *Crash* ⊆ *Sync x Sync*

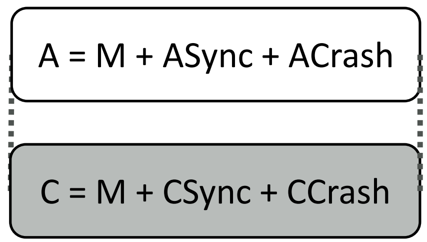
- The write buffer implementation has
  Sync = S and *Crash = „delete cache"*

- The abstract write buffer specification has
  Sync = „cache is empty" and *Crash = identity*

- Idea: Incrementally switch from low-level view to high-level view by refinement



Weak Crash-Safety: Refinement Type I



Theorem [Pfähler et. al., submitted to iFM17]:
If every run of every operation is either retractable or completable then C is a weak crash-safe implementation of A, written $A \sqsubseteq_{wcs} C$.

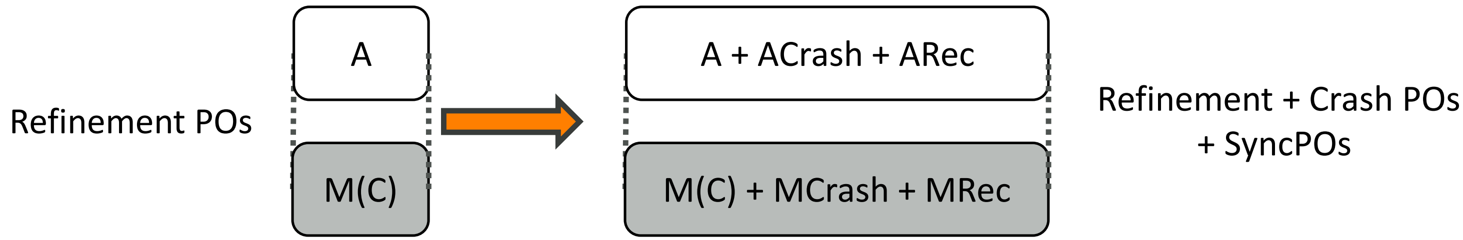
PO for Op retractable or completable:

$\langle Op(s) \rangle (CCrash(s, s'))$

$\rightarrow \quad CCrash(s, s')$

$\quad \vee \ \langle Op(s) \rangle (\ ASync \wedge CCrash(s, s')\ )$

Weak Crash-Safety: Refinement Type II



Theorem [Pfähler et. al., submitted to iFM17]:
If

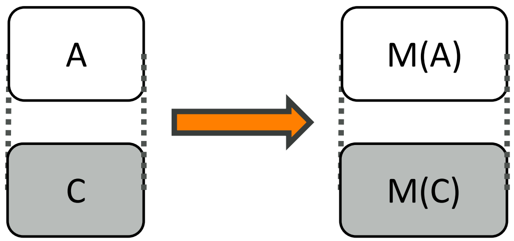
- C crash-neutral
- Refinement PO for each operation, including { Crash; Recovery } assuming we start in a synchronized state
- M has no additional persistent state
- ASync ∧ abs → CSync

then $A \sqsubseteq_{wcs} M(C)$

By transitivity of refinement we get:

$$POSIX \sqsubseteq_{wcs} VFS(\ldots(MTD))$$

Weak Crash-Safety: Submachines



Theorem [Pfähler et. al., submitted to iFM17]:
Weak Crash-Safe Submachine Refinement is compositional and transitive

- $A \sqsubseteq_{wcs} C \rightarrow M(A) \sqsubseteq_{wcs} M(C)$
- $A \sqsubseteq_{wcs} B$ and $C \sqsubseteq_{wcs} C \rightarrow A \sqsubseteq_{wcs} C$

By transitivity of refinement we get:

$$POSIX \sqsubseteq_{wcs} VFS(\ldots(WriteBuffer(\ldots(MTD))))$$

Summary & Related Work

Institute for Software & Systems Engineering

- Added KIV support for weak crash-safe machines
- Simplified Verification
  500 → 300, 1050 → 1270 (proof interactions)
  for the two specifications where we previously had proofs
- 30-40% less waste of space for padding

Related Work:

- Specifying and Checking File System Crash-Consistency Models [ASPLOS 16]
- Reducing Crash Recoverability to Reachability [POPL 16]

Overview

1. Flash Memory and Flash File Systems
2. Results of Flashix I
3. Current Result: Integration of write-back Caches
4. Outlook: Concurrency

Goals & Previous Research

Institute for Software & Systems Engineering

Goals for Flashix:

- Parallel operations
  - Garbage Collection, Wear-Leveling in background
  - Allow parallel access to POSIX
- No Dead/Livelocks

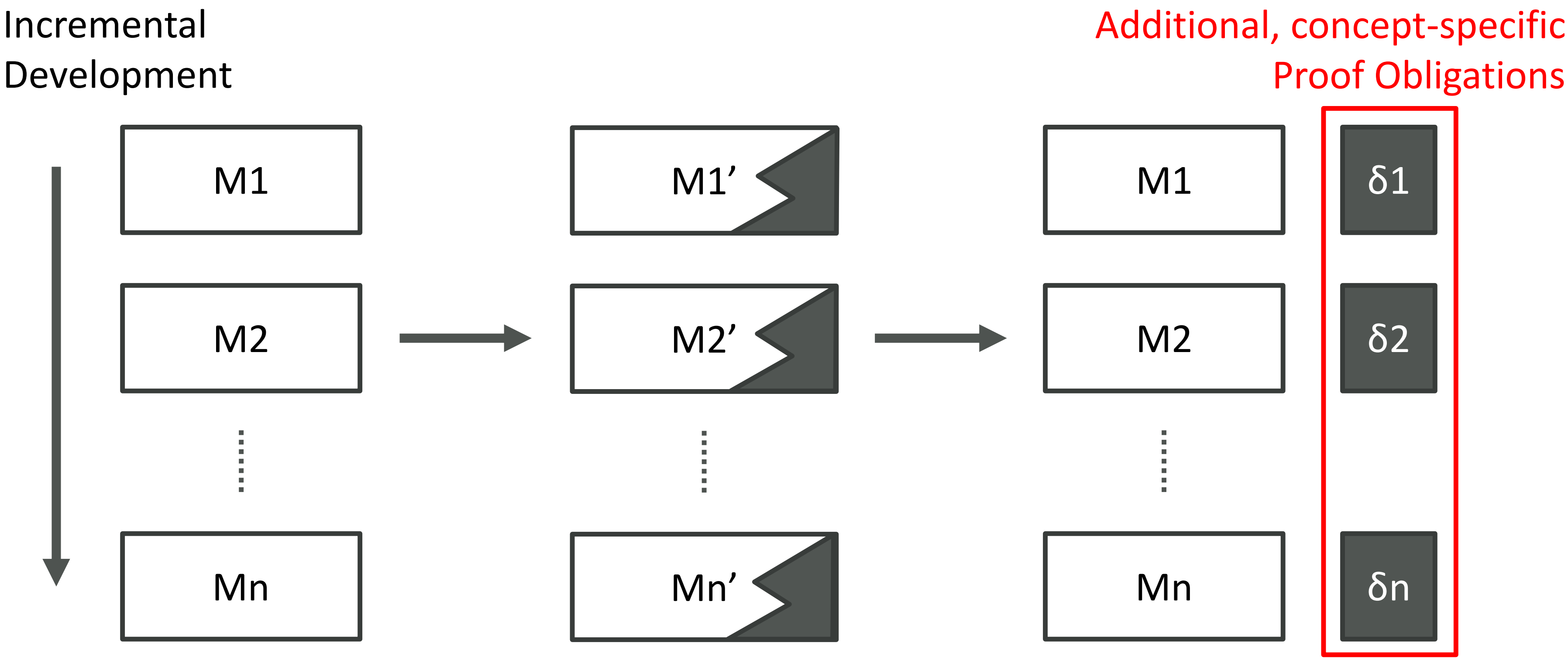
Previous Research:

- Rely/Guarantee & Temporal Logic
- Linearizability
- Lock-free & starvation-free algorithms / data structures

Challenge in Flashix:

- Scale verification to a large case study with deep hierarchy of refinements

Non-local Extension



Non-local Extension with an additional concept

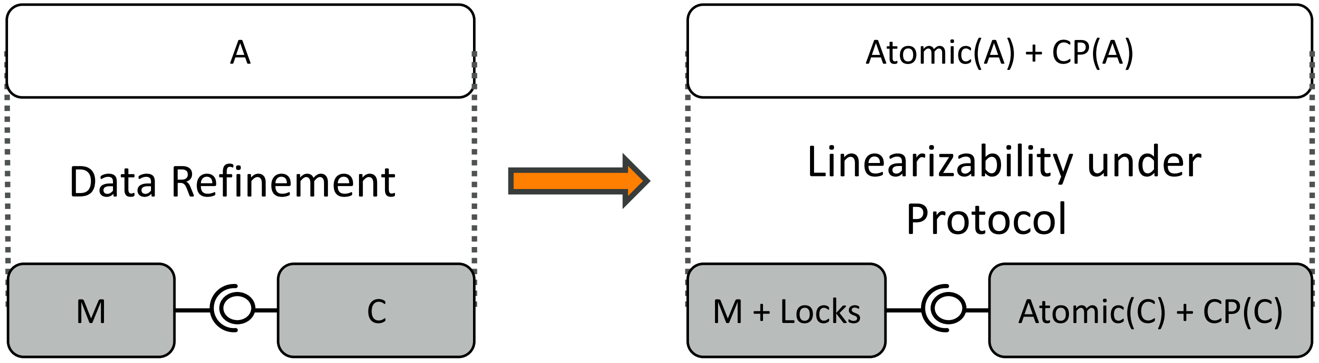
Modularization following the original refinements
Goal: Do not verify from scratch

Instances of Non-local Extensions

Institute for Software & Systems Engineering

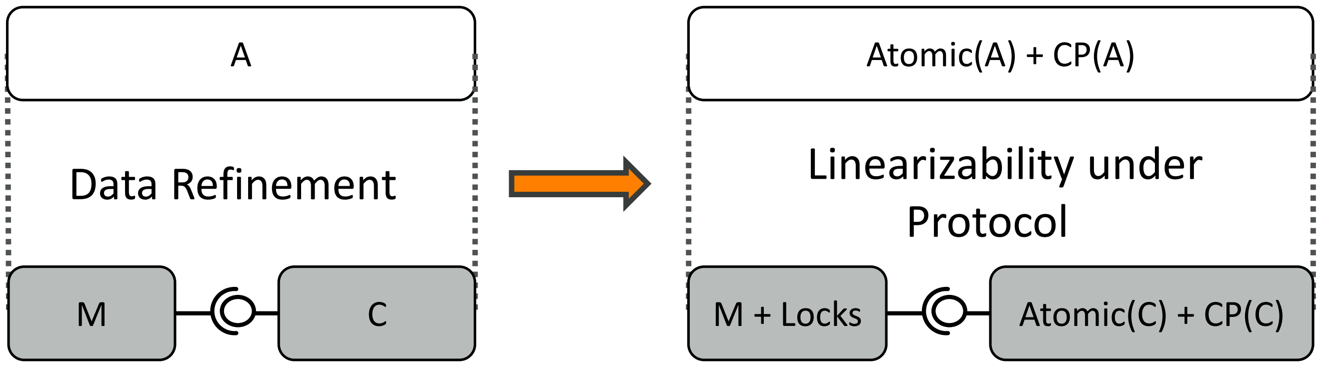
- Crash-Safety
  - Modularization resulting in additional, orthogonal proof obligations worked
- Write-back Caches and Weak Crash-Safety
- Concurrency?
  - Making expensive operations concurrent seems to be a standard problem in software engineering
  - Related formal theories or verified case studies?
    → Interested in Feedback

Linearizability under Protocol (I)



- Concurrency Protocol CP(A) specifies whether $AOp_i(in_i)$ || $AOp_j(in_j)$ is allowed
- Restricts possible concurrent histories
  => only these have to be linearizable
- Examples in Flashix:
  - Writing to the same block disallowed (only sequential writes)
  - Wear-Leveling or block erase is allowed in parallel
- Examples outside Flashix:
  - Iterators may not be used concurrent with modifications
- Difference to general linearizability: we have a single known client M for C, while linearizability requires C to work for any client

Linearizability under Protocol (II)



Open Issues:

- How to specify CP? Current assumption is that a predicate (AOp_i, $in_{i.}$ $AOp_{j,}$ in_j) is sufficient
- What proof obligations show that calls of C opertions follow protocol CP(C) assuming that calls to M(C) operations follow protcol CP(A)?
- Incrementally increase atomicity of M operations [Lipton 75], [Elmas, Qadeer, Tasiran 09] with ownership
- What granularity of atomic blocks remains and how do we then reuse the sequential verification?
  - Ideally, M(C) operations with locks are immediately atomic → nothing new must be proved