

Verifying a high-performance crash-safe file system using a tree specification

Haogang Chen, Tej Chajed, Stephanie Wang, Alex Konradi, Atalay İleri, Adam Chlipala, M. Frans Kaashoek, Nickolai Zeldovich



File systems are difficult to make correct

- Complicated implementations
 - on-disk layout
 - in-memory data structures
- Computer can crash at any time

Despite much effort, file systems have bugs

- File systems still have subtle bugs
 - Well documented [Lu, TOS '14] [Min, SOSPP '15]
- Example from ext4:
combination of two optimizations allows data to leak from one file to another on crash
 - Discovered after 6 years [Kara 2014]

Approach: formal verification

- Write a specification
- Prove implementation meets the specification
 - Ensures implementation handles all corner cases
 - Proof assistant (Coq) ensures proof is correct
- Avoid large class of bugs



Existing verified file systems

correctness

FSCQ [SOSP '15]
BilbyFS [ASPLOS '16]
Yggdrasil [OSDI '16]

verified file systems

ext4
btrfs
ZFS

performance

Goal: verified high-performance file system

correctness

FSCQ [SOSP '15]
BilbyFS [ASPLOS '16]
Yggdrasil [OSDI '16]

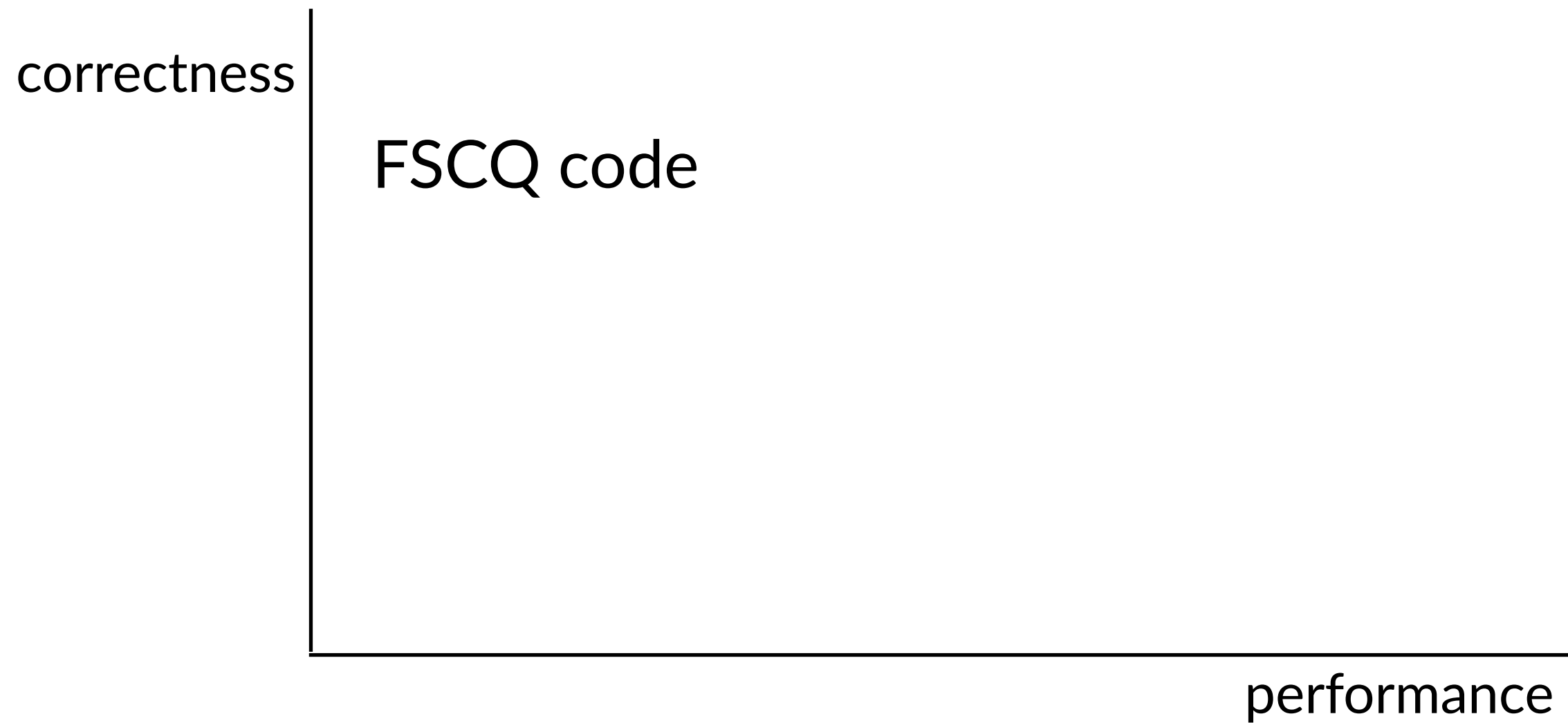
verified file systems

?

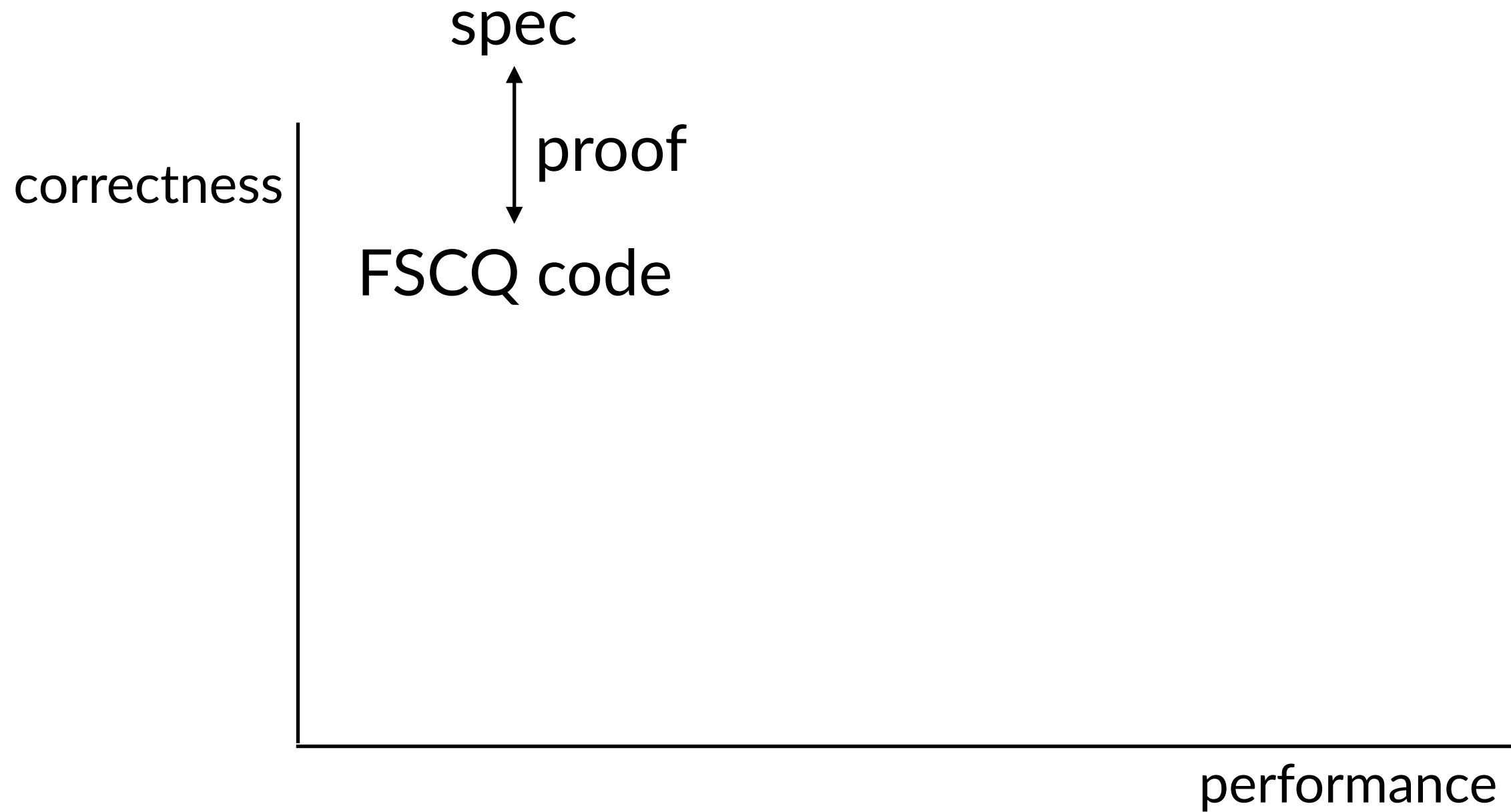
ext4
btrfs
ZFS

performance

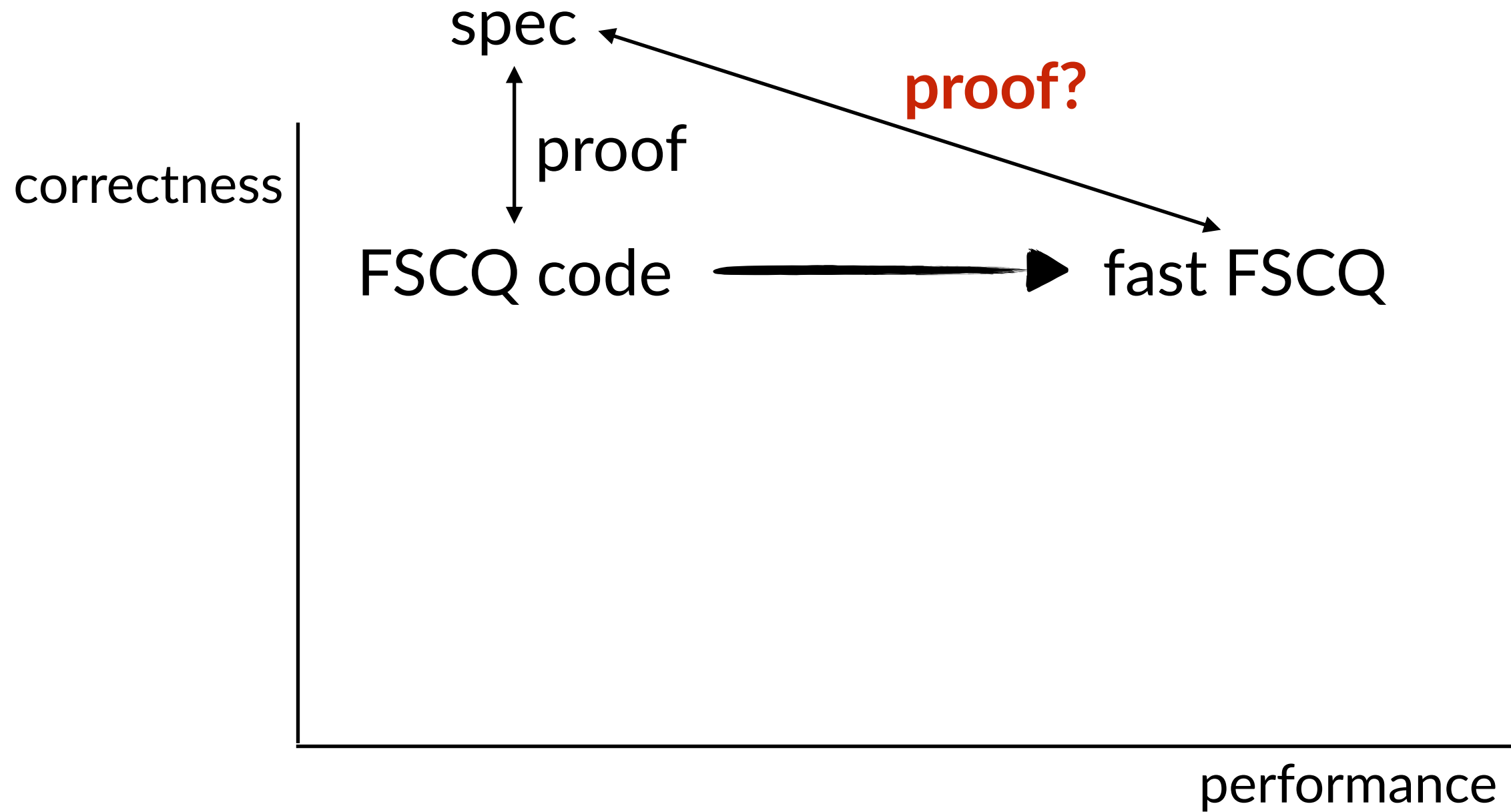
Strawman: optimize FSCQ



Strawman: optimize FSCQ



Strawman: optimize FSCQ



Problem: specification incompatible with high performance

- Achieving high performance requires optimizations
- Some optimizations change file-system behavior
- Requires changes to specification

Example optimization: deferred commit

- Deferred commit: buffer system calls until `fsync`
- FSCQ's specification: "if `create(f)` has returned and computer crashes, `f` exists"
- Deferred commit requires a new specification

Optimizations that change crash behavior

- Deferred commit: buffer system calls until fsync
- Log-bypass writes: skip log for data writes
- Buffer cache: cache data until fdatasync
- **Existing specifications do not support these optimizations**

Contribution: **DFSCQ** file system

- Precise specification for a subset of POSIX
 - supports deferred commit and log-bypass writes
- Verified, crash-safe file system
 - Traditional journalling file-system design
 - Implements most of ext4's optimizations
 - Machine-checked proof that implementation meets specification
 - Performance on par with ext4 (but DFSCQ has fewer features)

Specifying a file system

- Design abstract state

Specifying a file system

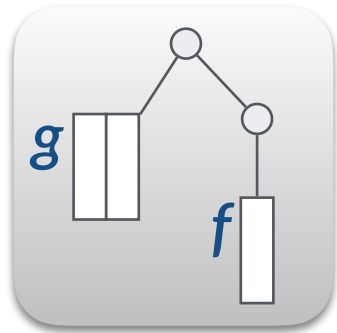
- Design abstract state
- Describe how system calls execute

Specifying a file system

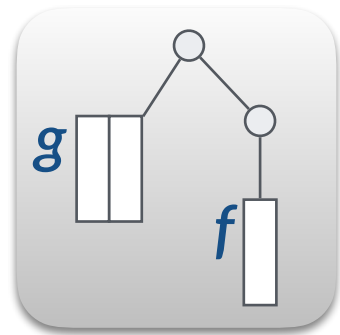
- Design abstract state
- Describe how system calls execute
- Describe effect of crashes

Starting point: tree as abstract state

Trees are a simplified abstraction of a file system



Specification abstracts implementation details



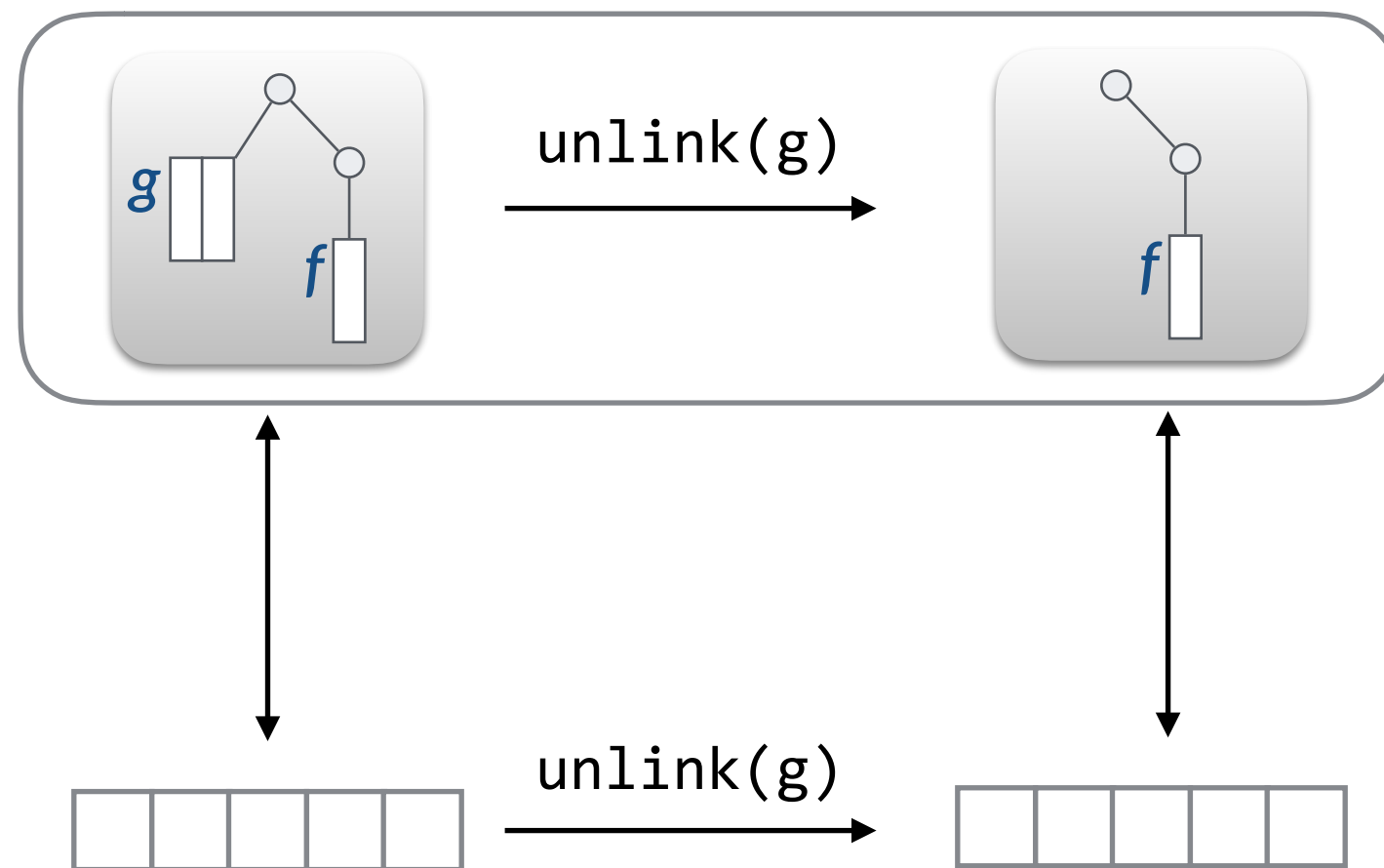
abstract state



implementation's
state

Specify how system calls affect abstract state

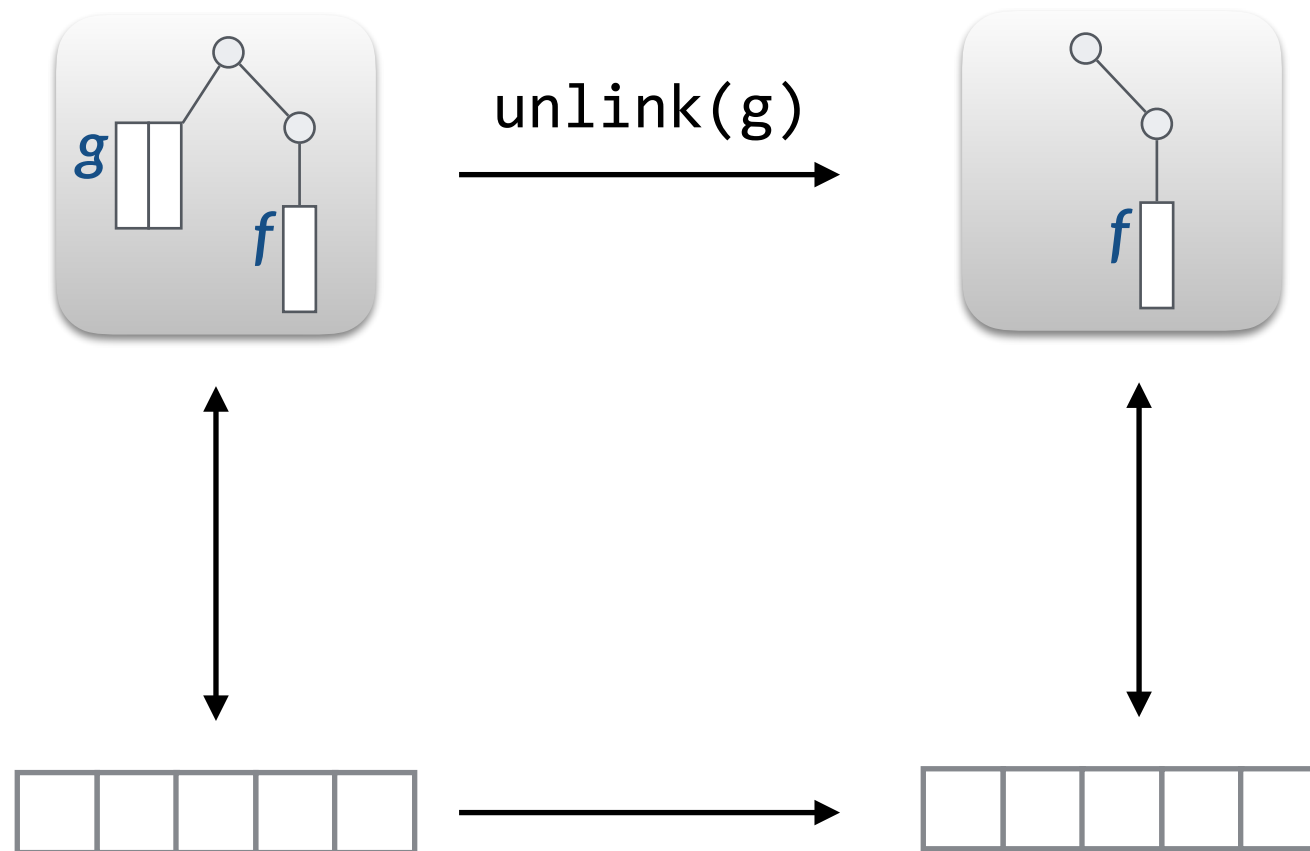
specification describes transition



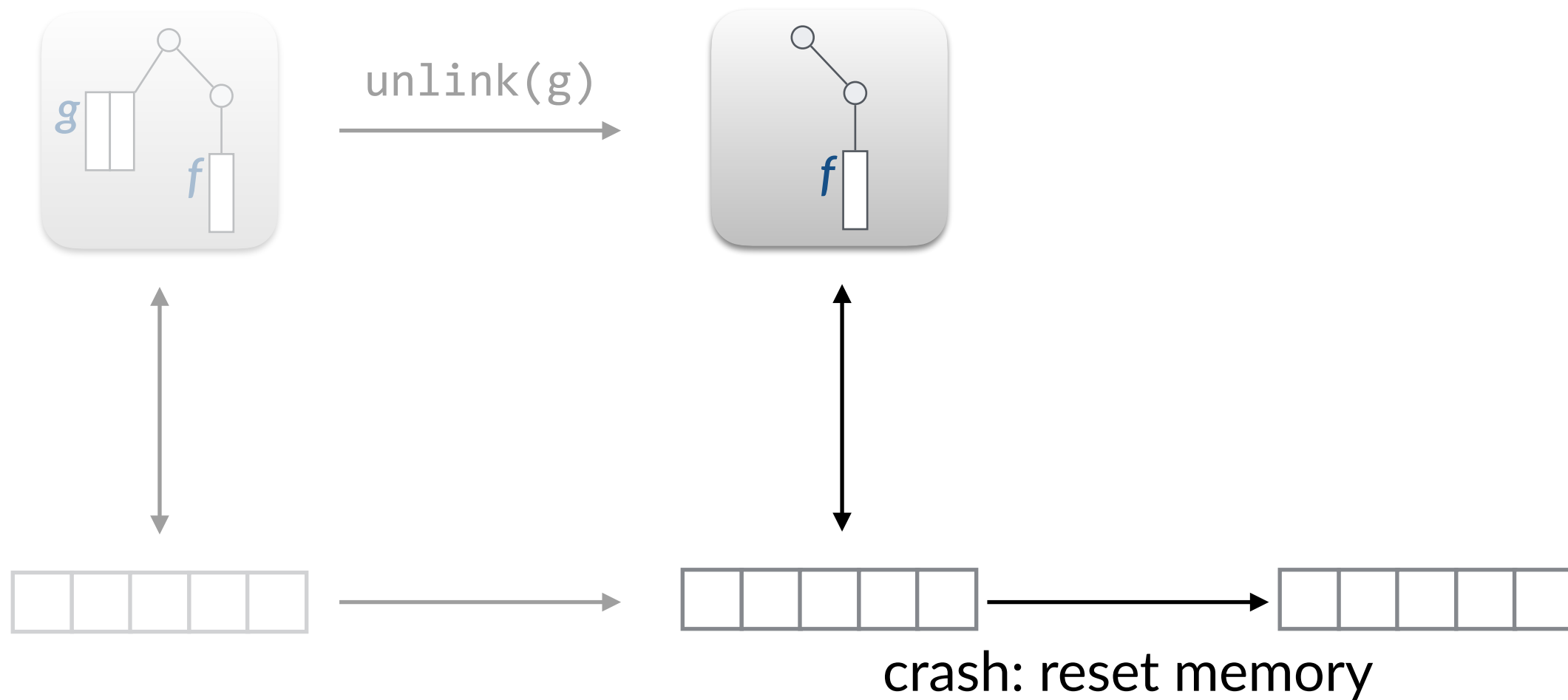
Challenges in specifying crash behavior

- Optimizations mean crashes can be complex
- Problem 1: deferred commit
- Problem 2: log-bypass writes
- Problem 3: caching

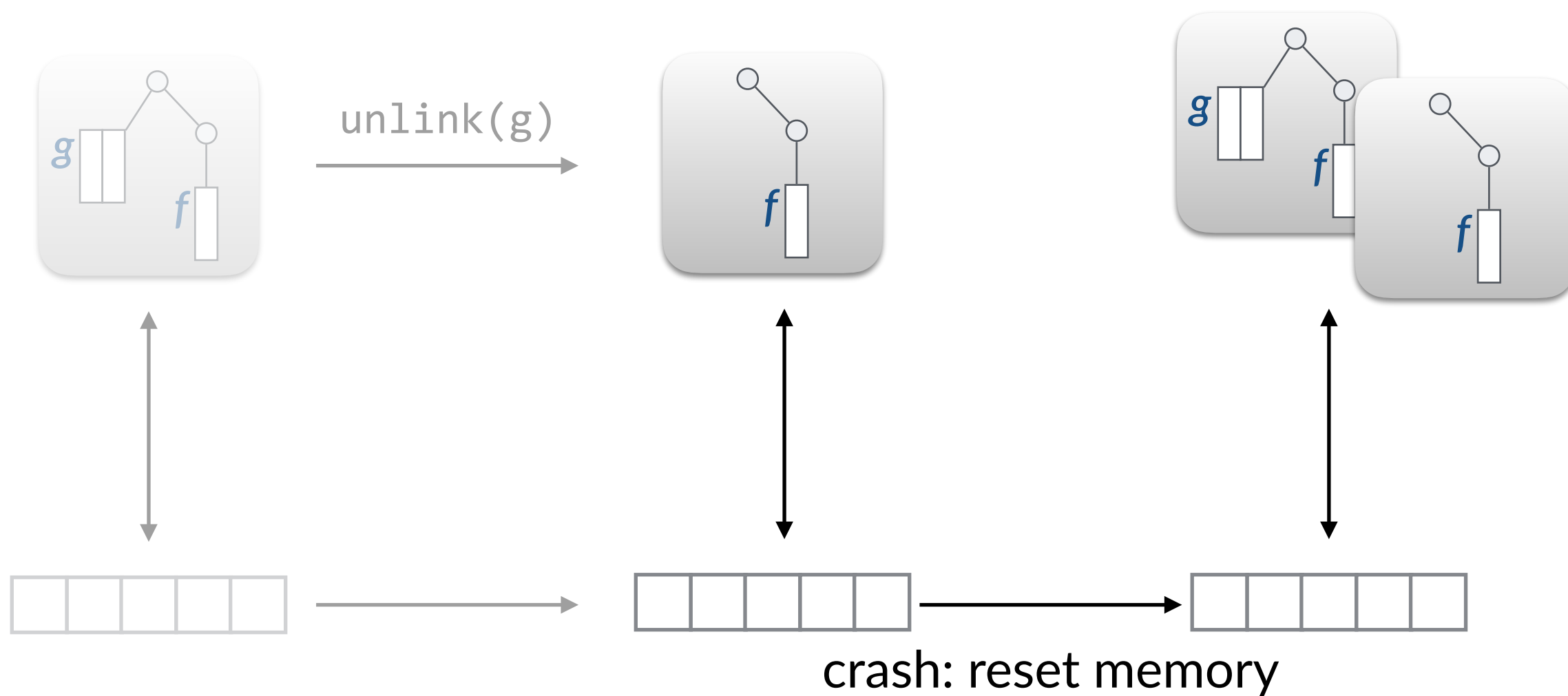
Problem 1: deferred commit leads to many crash states



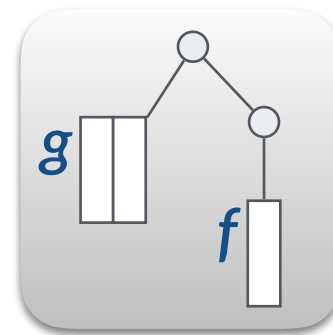
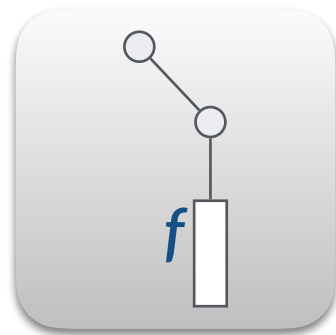
Problem 1: deferred commit leads to many crash states



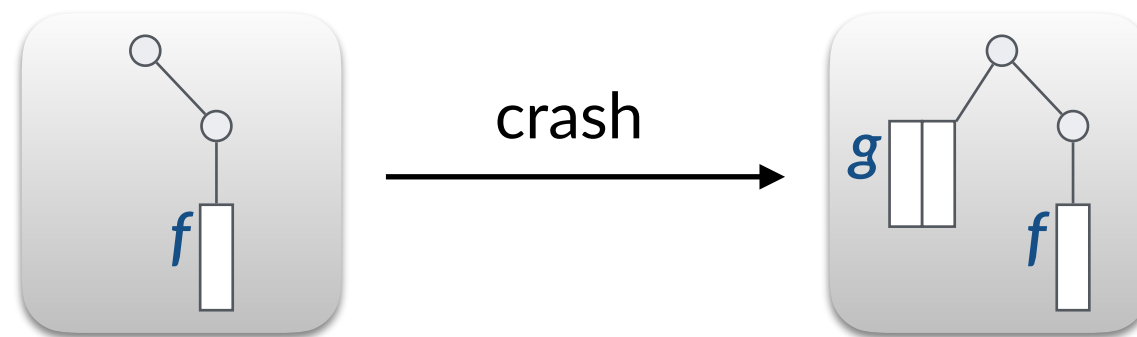
Problem 1: deferred commit leads to many crash states



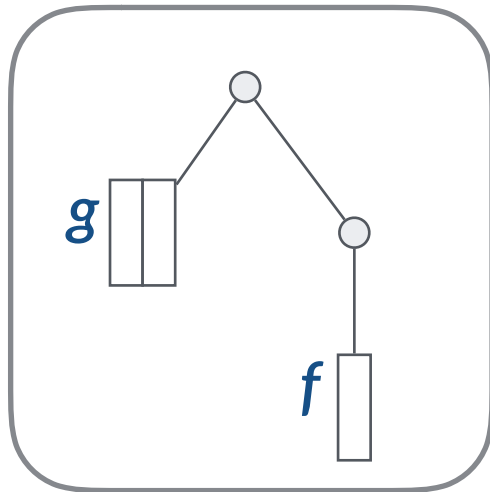
How do we specify crash outcomes with deferred commit?



How do we specify crash outcomes with deferred commit?



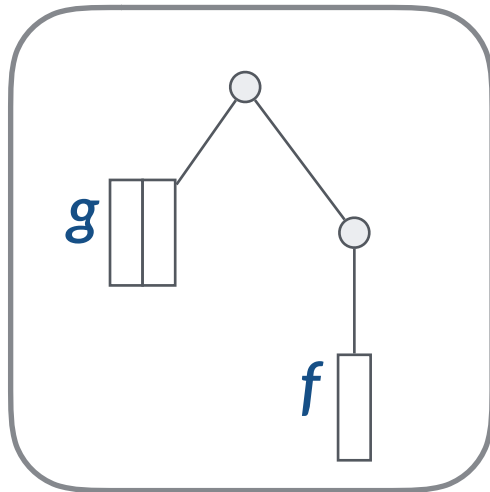
Specify deferred commit using **tree sequences**



tree sequence

Specify deferred commit using **tree sequences**

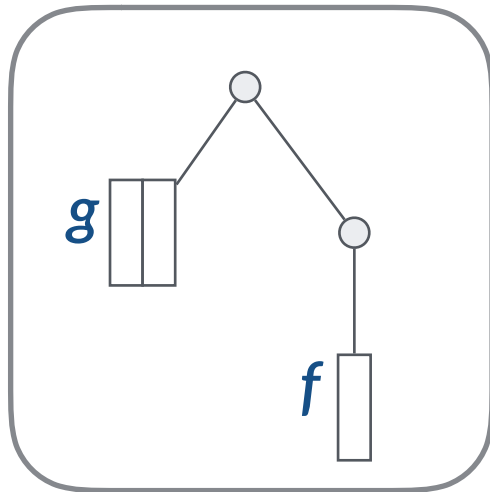
- Abstract state is a **sequence of trees**



tree sequence

Specify deferred commit using **tree sequences**

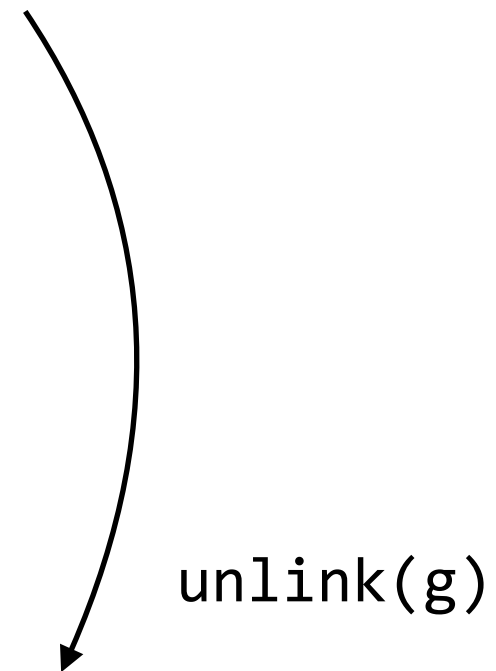
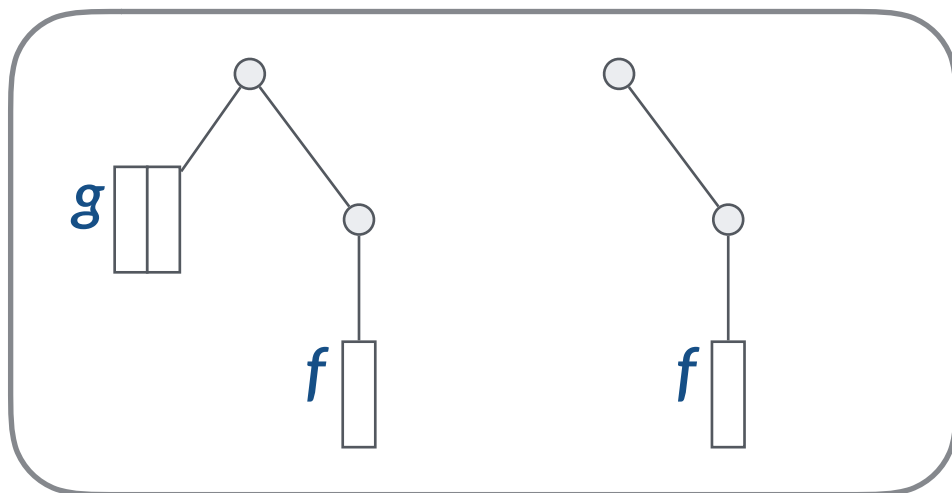
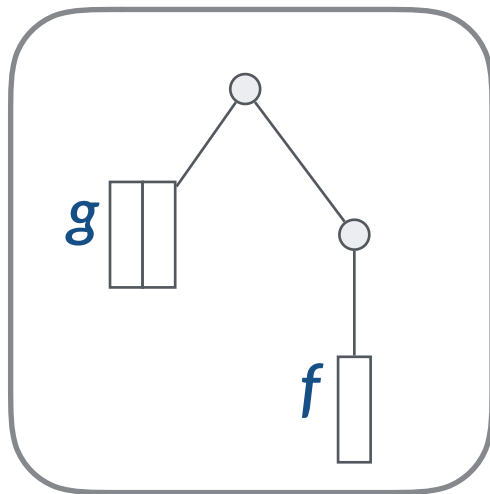
- Abstract state is a **sequence of trees**
- Always read from the latest tree



tree sequence

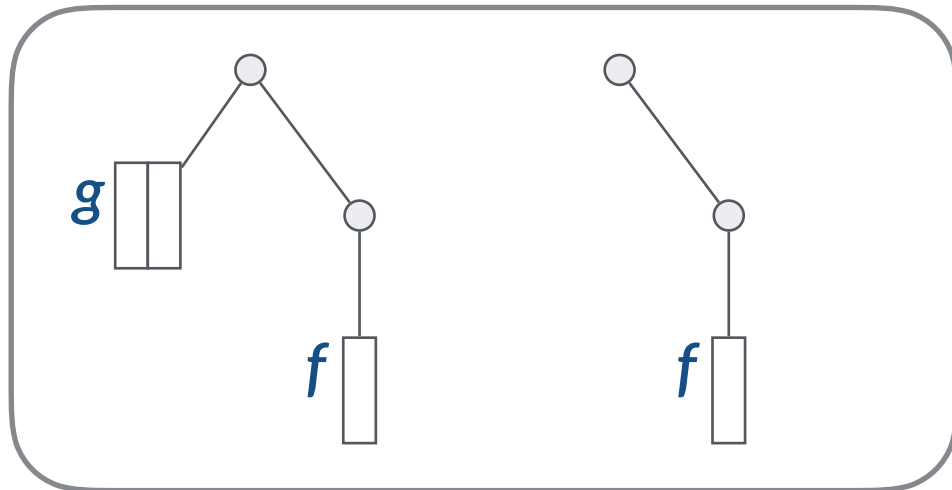
Specify deferred commit using **tree sequences**

- Metadata updates add new trees in the specification
- Always read from the latest tree



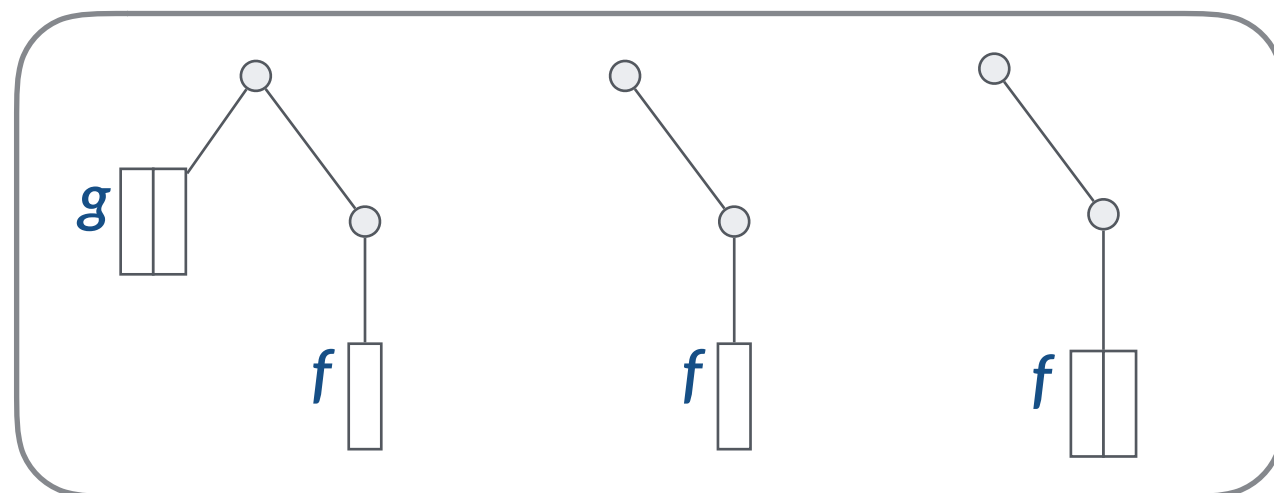
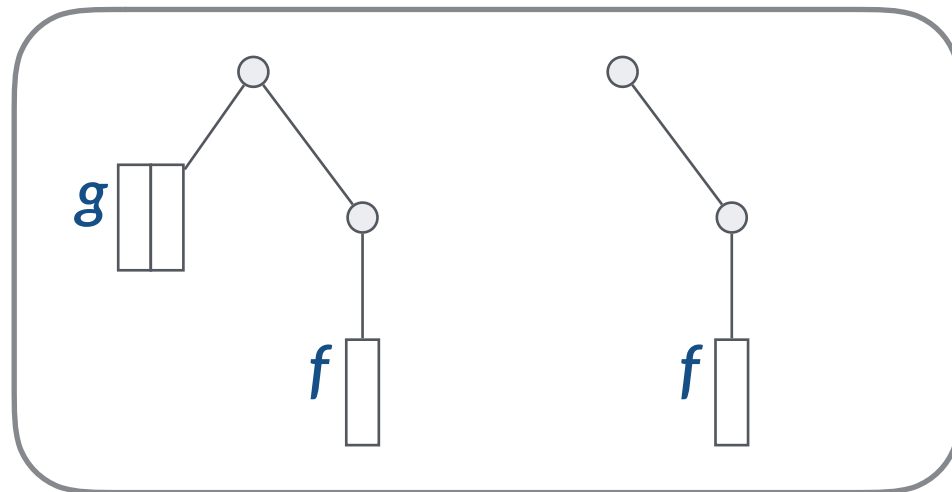
Specify deferred commit using **tree sequences**

- Metadata updates add new trees in the specification
- Always read from the latest tree



Specify deferred commit using **tree sequences**

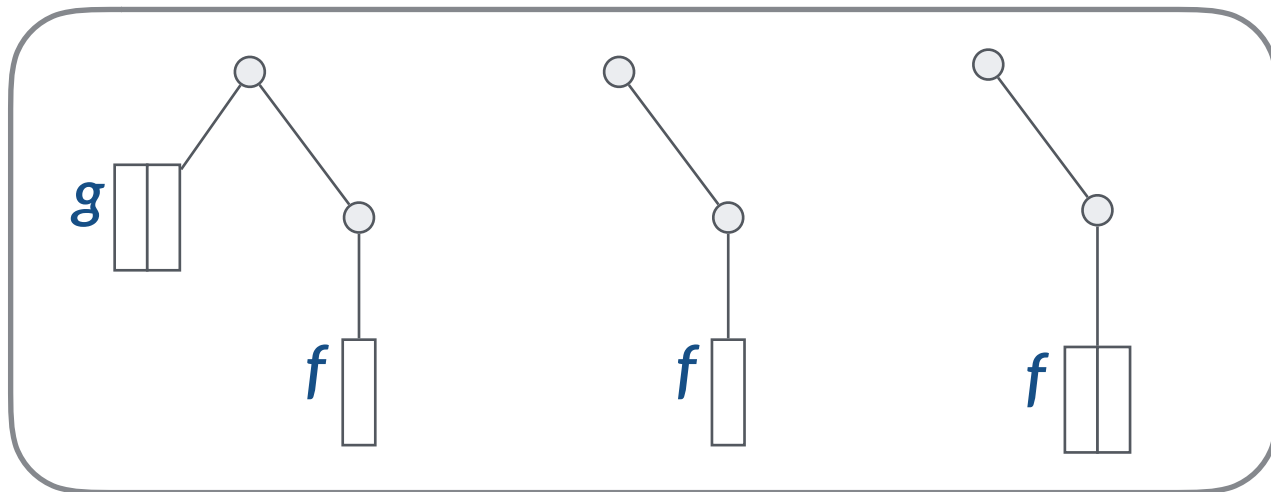
- Metadata updates add new trees in the specification
- Always read from the latest tree



truncate(f, 2)

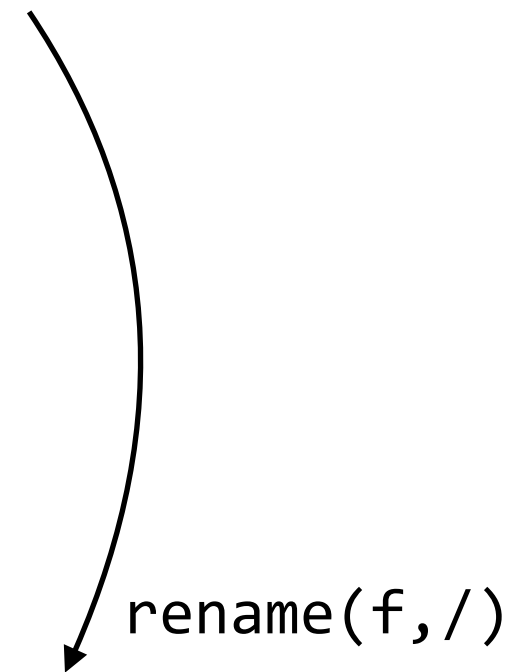
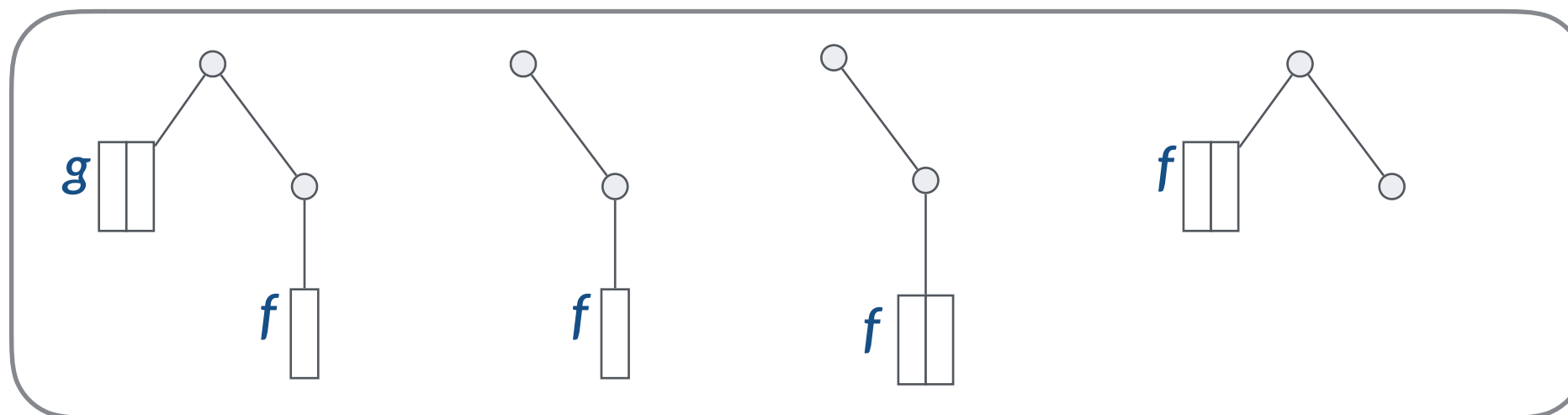
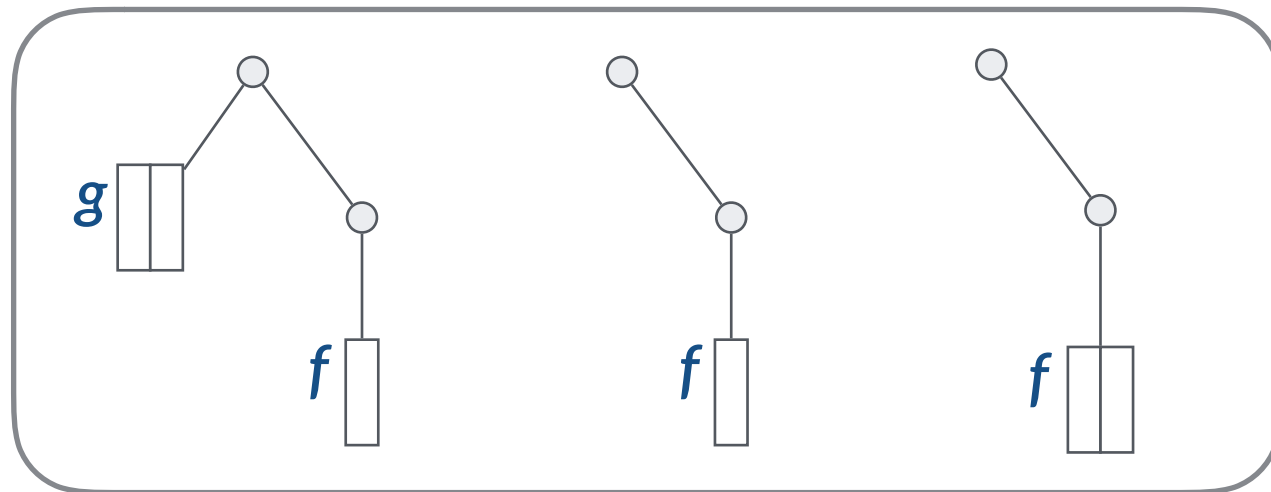
Specify deferred commit using **tree sequences**

- Metadata updates add new trees in the specification
- Always read from the latest tree



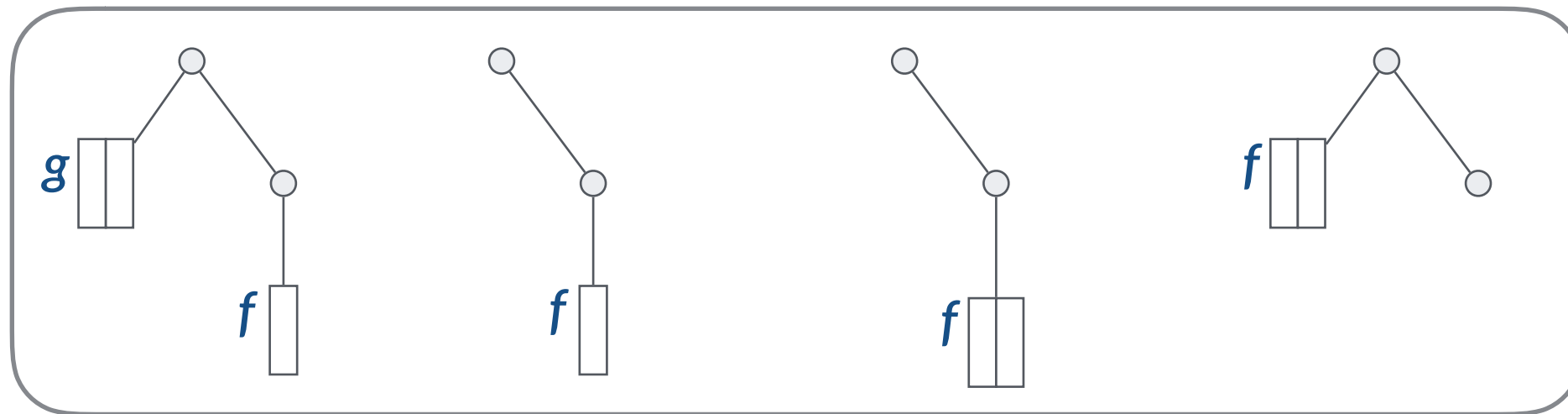
Specify deferred commit using **tree sequences**

- Metadata updates add new trees in the specification
- Always read from the latest tree



Behavior of tree sequences on crash

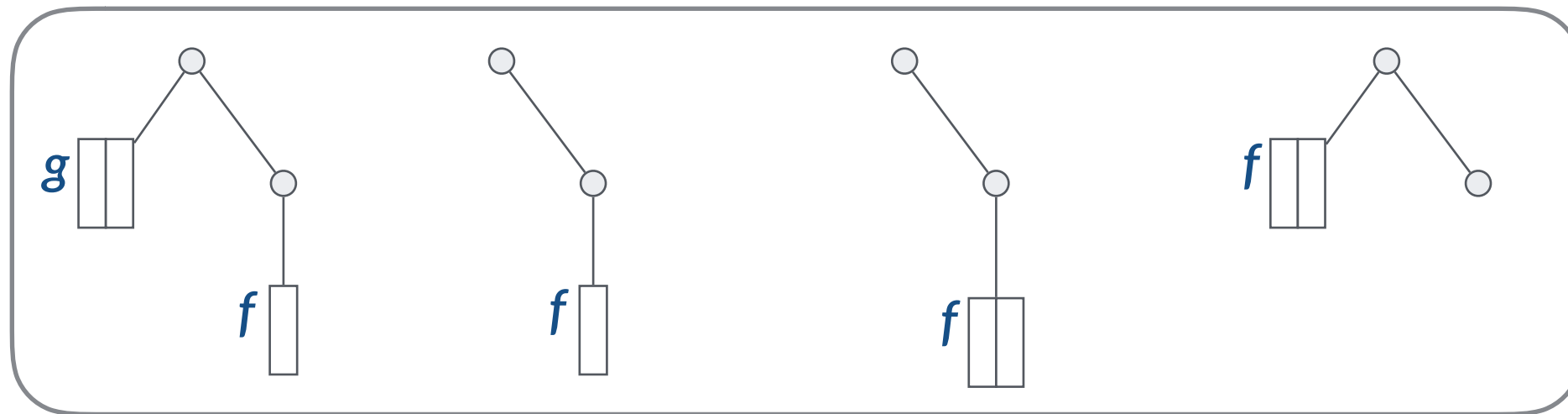
- What about crash behavior?



tree sequence

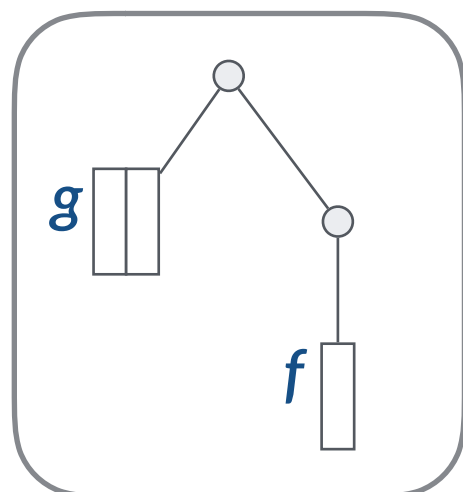
Behavior of tree sequences on crash

- What about crash behavior?



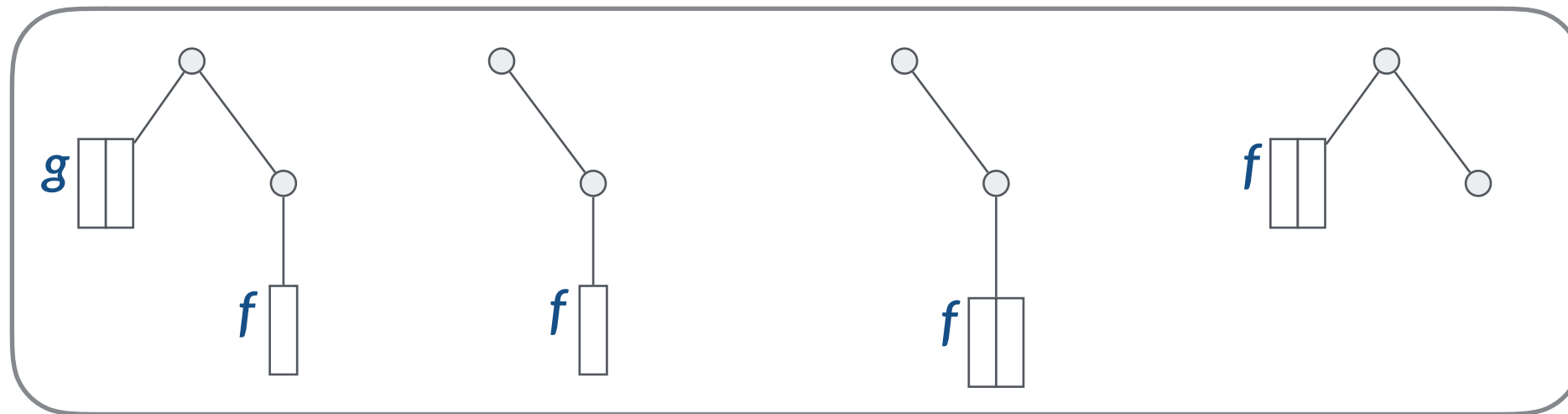
tree sequence

↓ crash



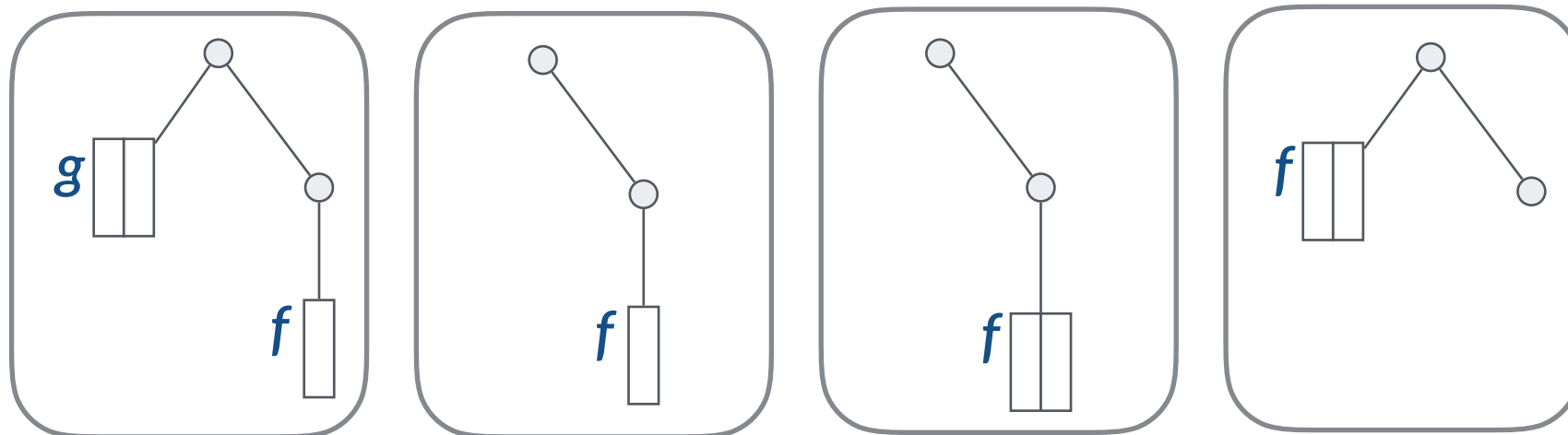
post-crash
tree sequence

Crash specification allows background commits



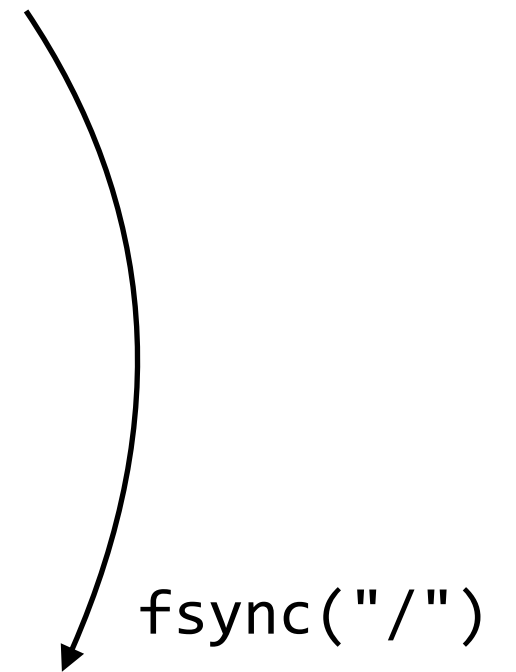
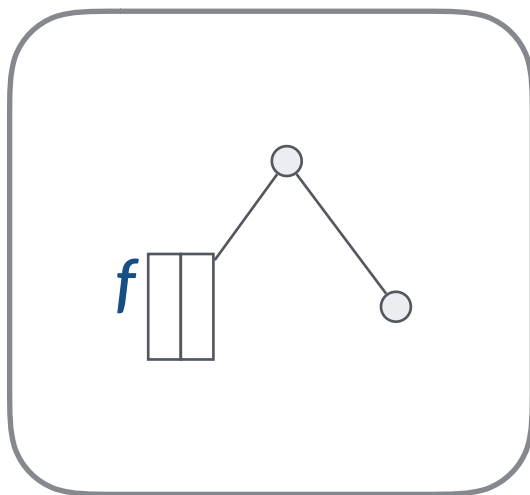
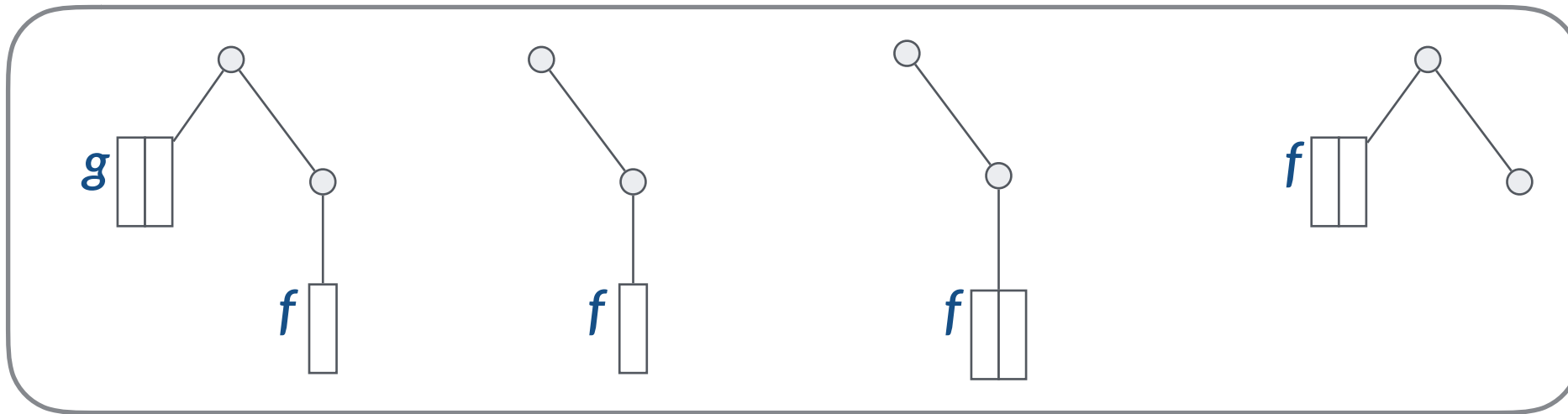
tree sequence

post-crash states:



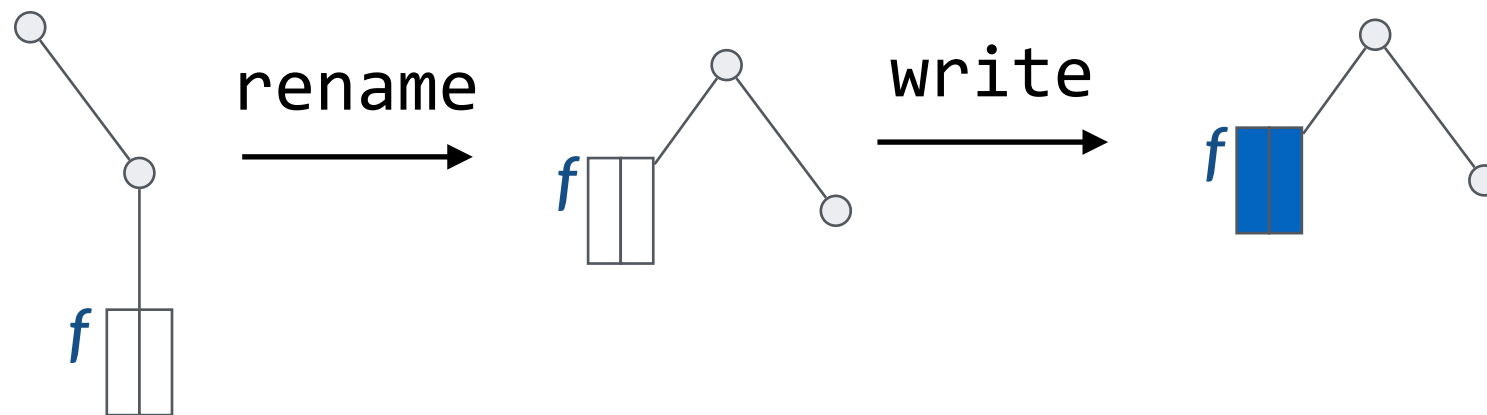
crash

Specification for **fsync**



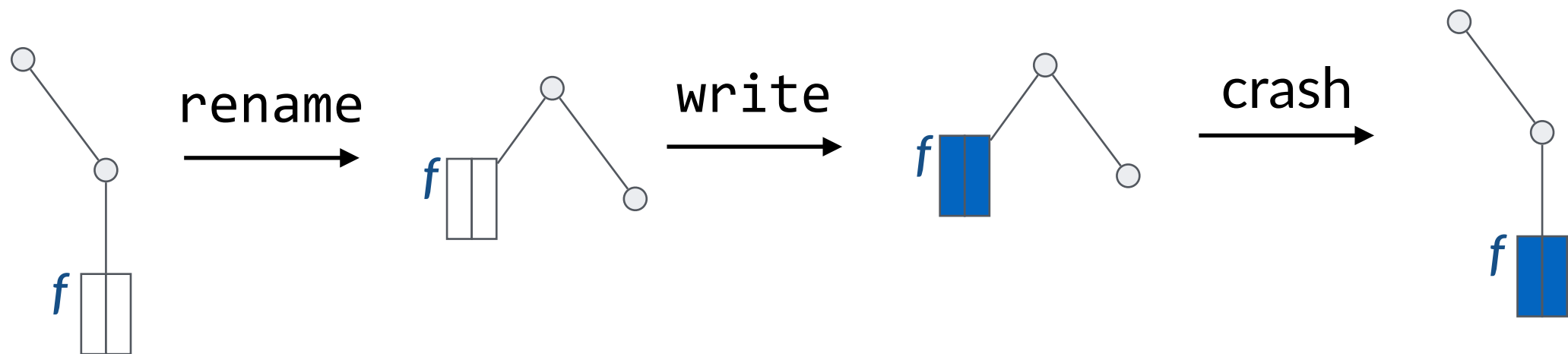
Problem 2: log-bypass writes may reorder updates

- Log-bypass writes: update file data blocks in place, skipping log

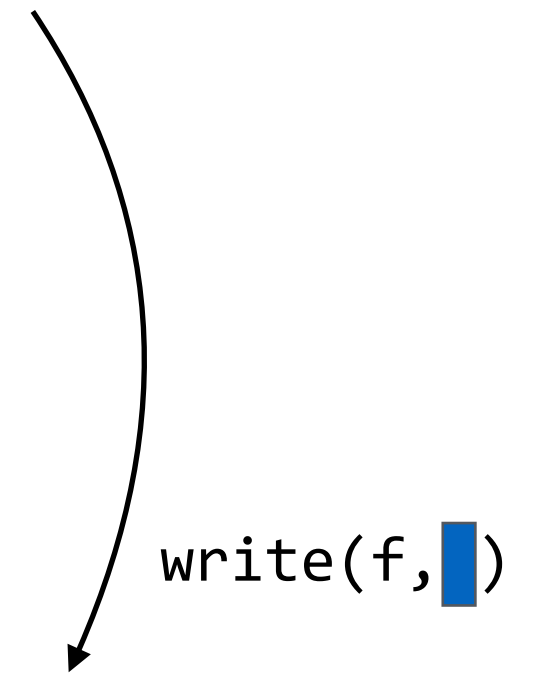
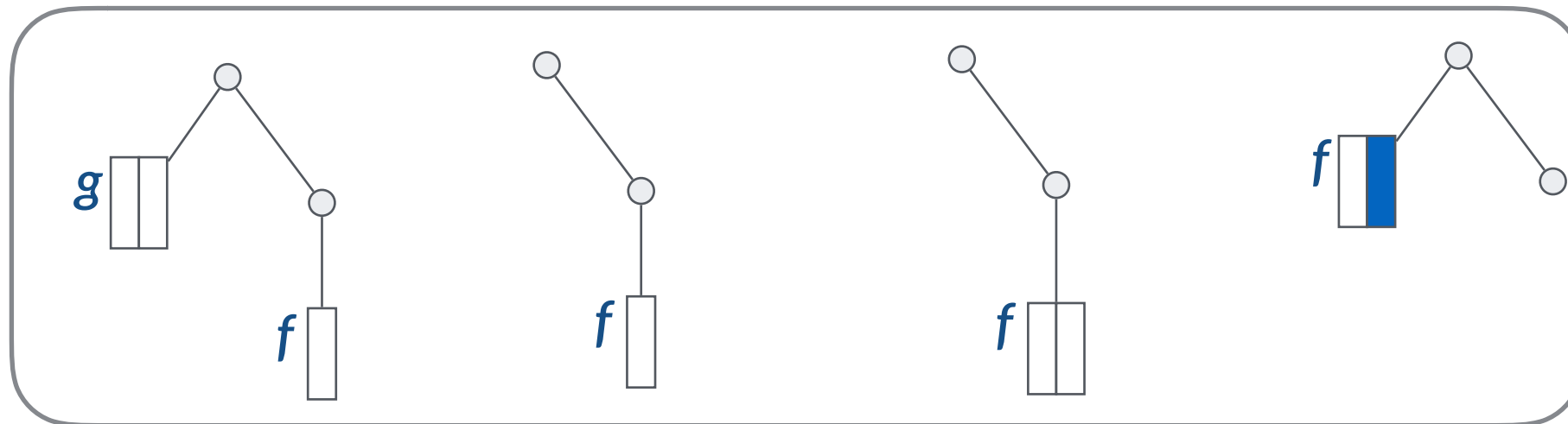
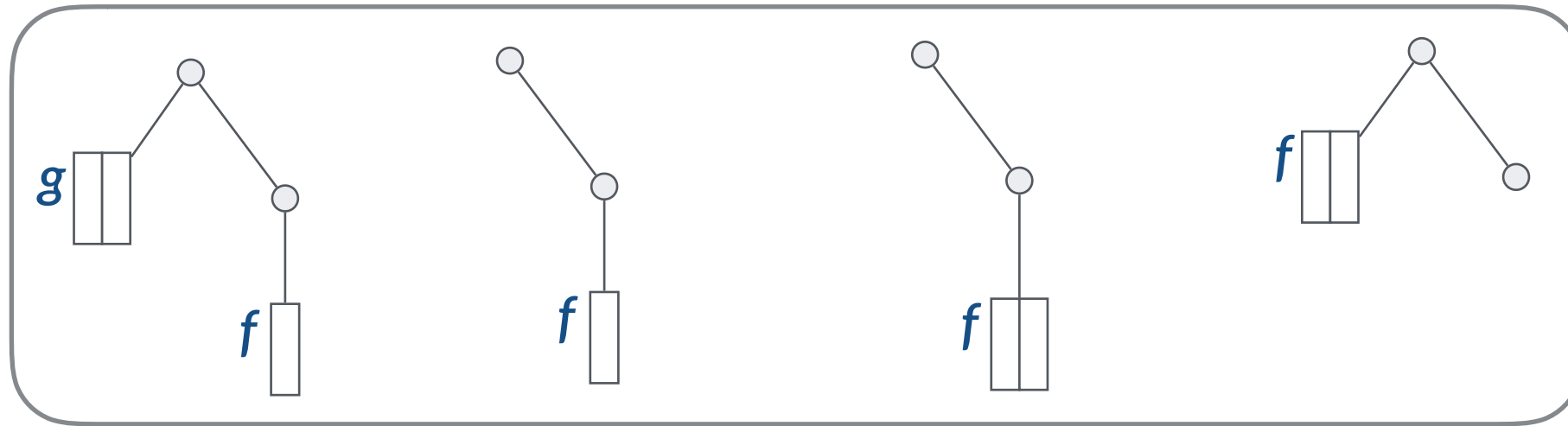


Problem 2: log-bypass writes may reorder updates

- Log-bypass writes: update file data blocks in place, skipping log
- Effect: data writes and metadata updates can be reordered on crash

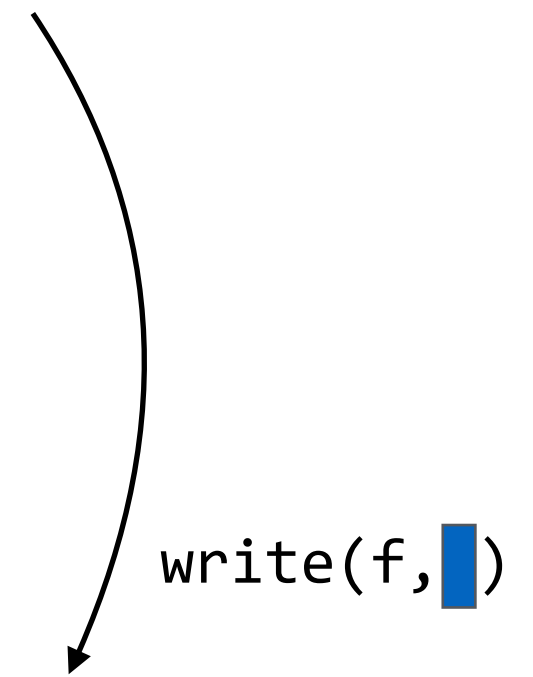
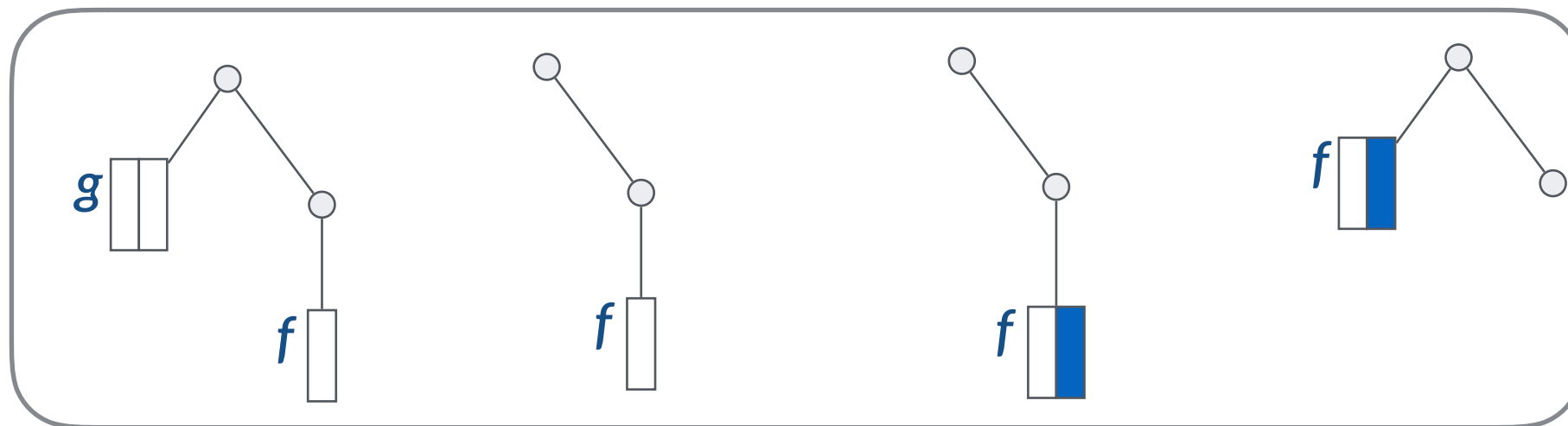
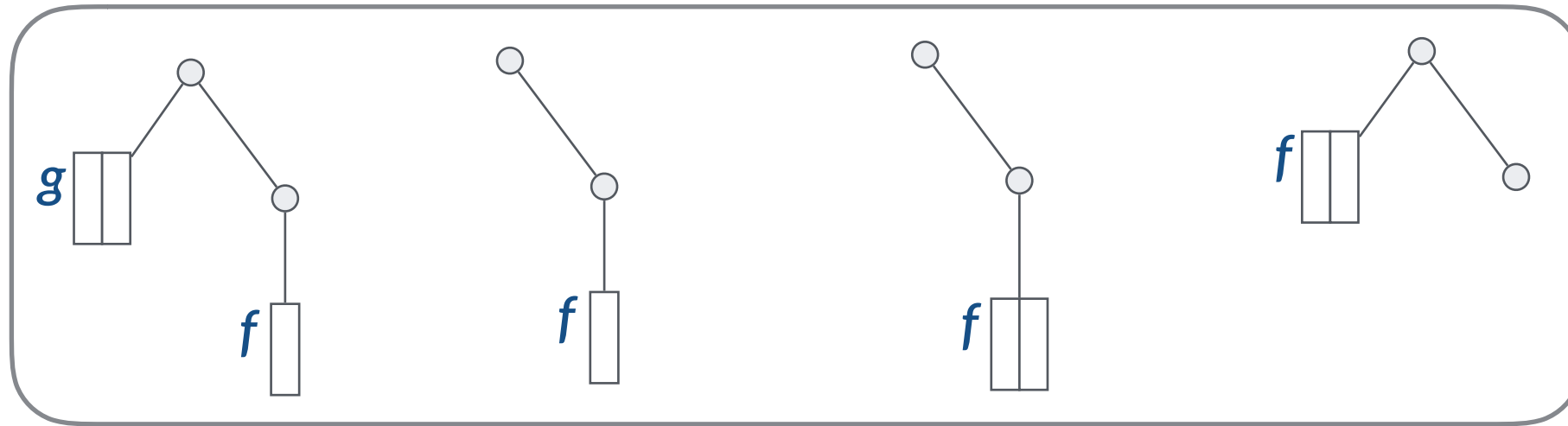


Log-bypass writes



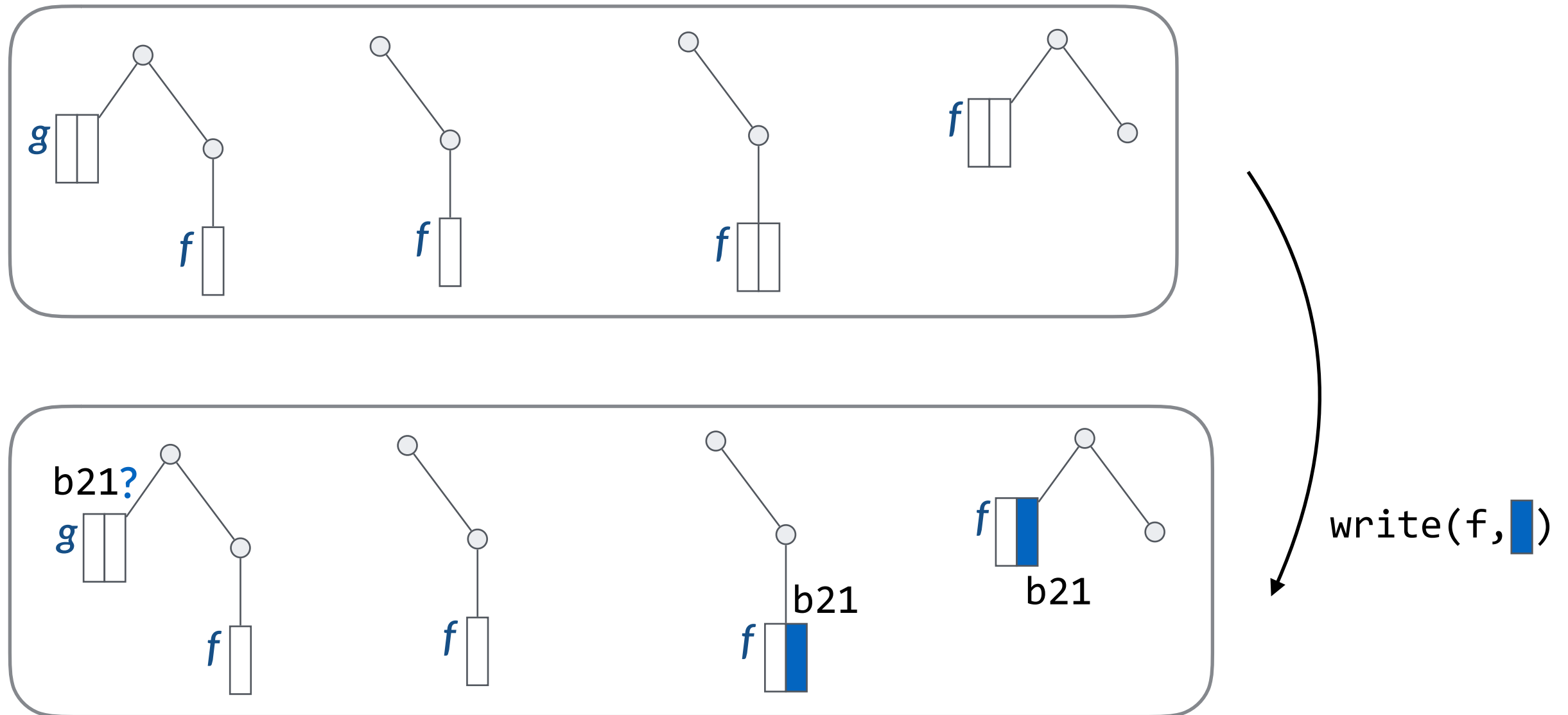
At minimum, writes to latest tree

Log-bypass writes



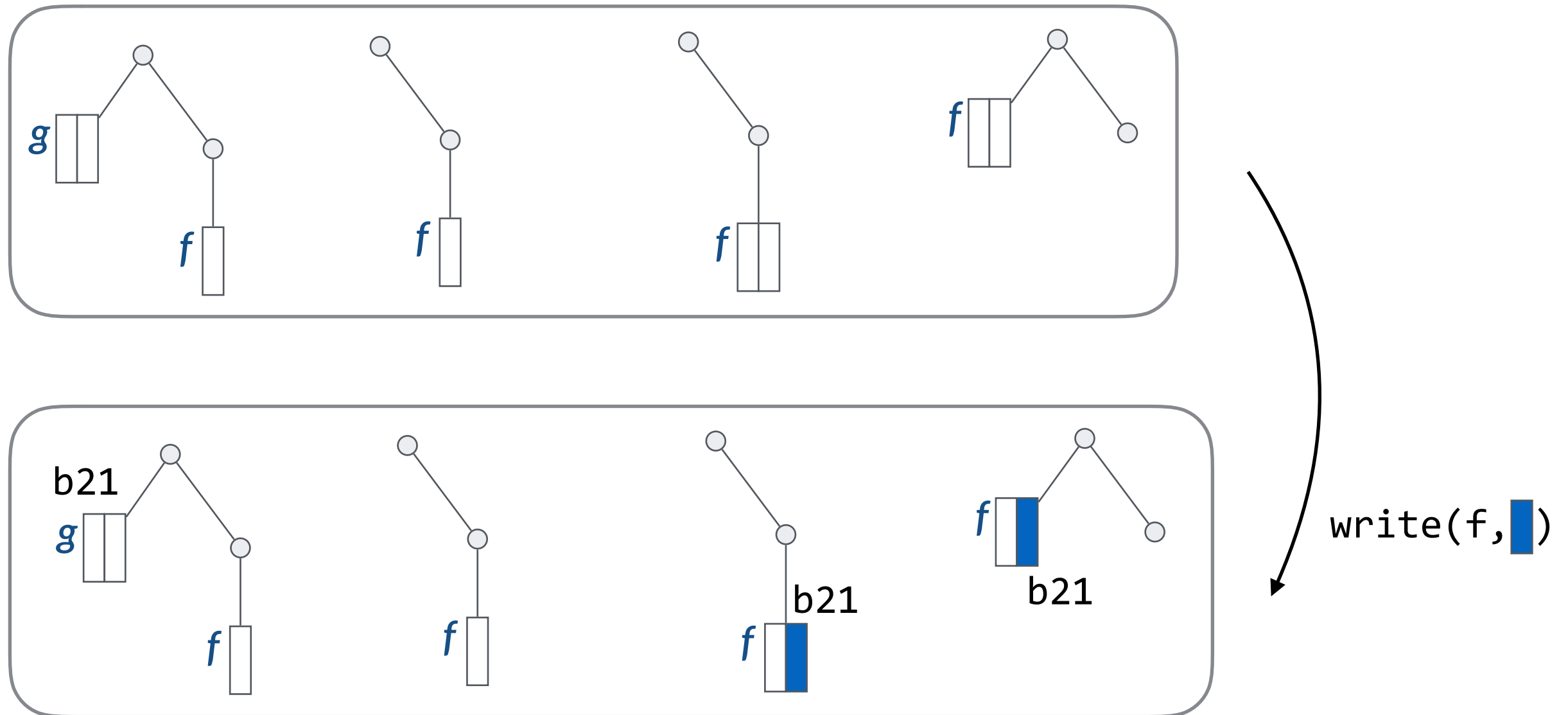
Affects the **same file** in earlier trees

Specify that other files are unaffected



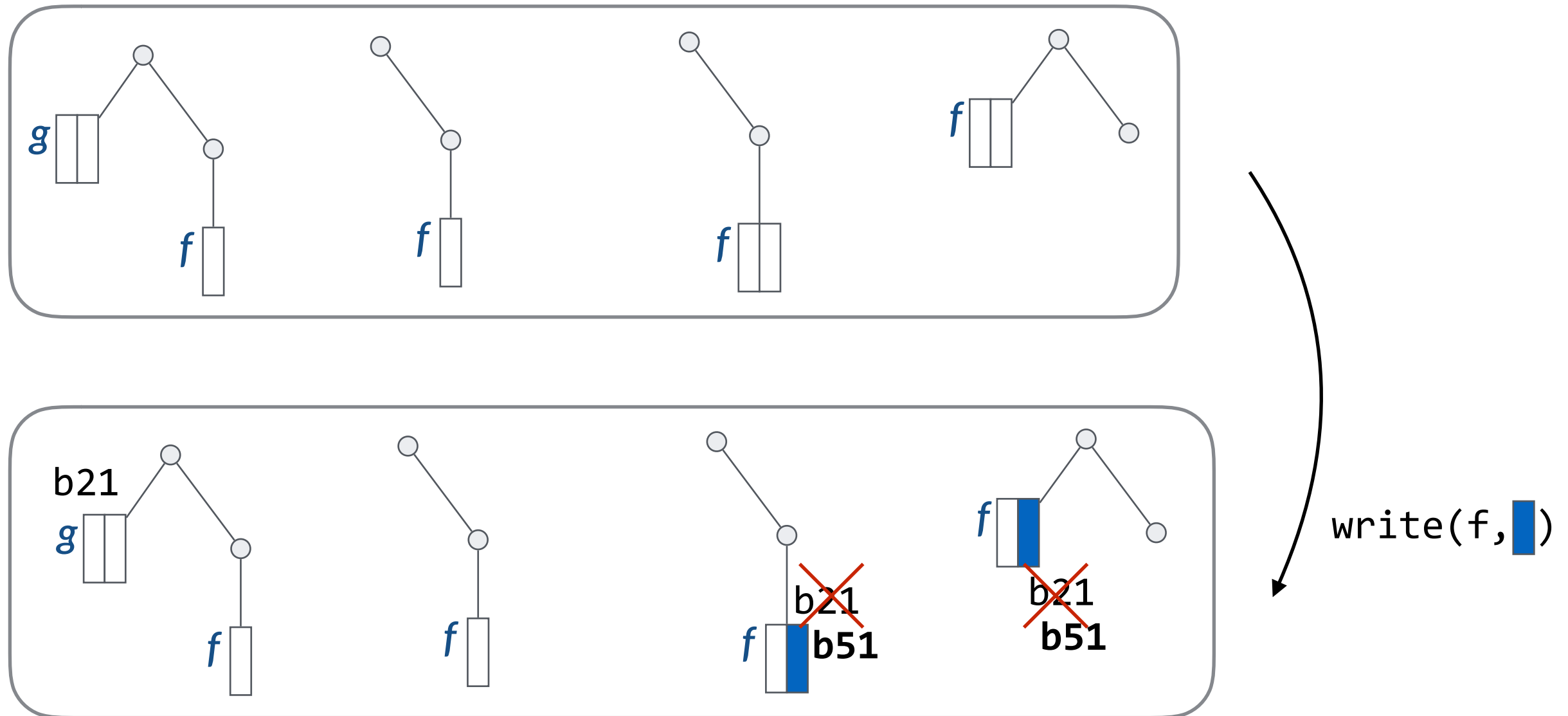
Puts an obligation on the implementation to avoid block re-use within a tree sequence

Specify that other files are unaffected



Puts an obligation on the implementation to avoid block re-use within a tree sequence

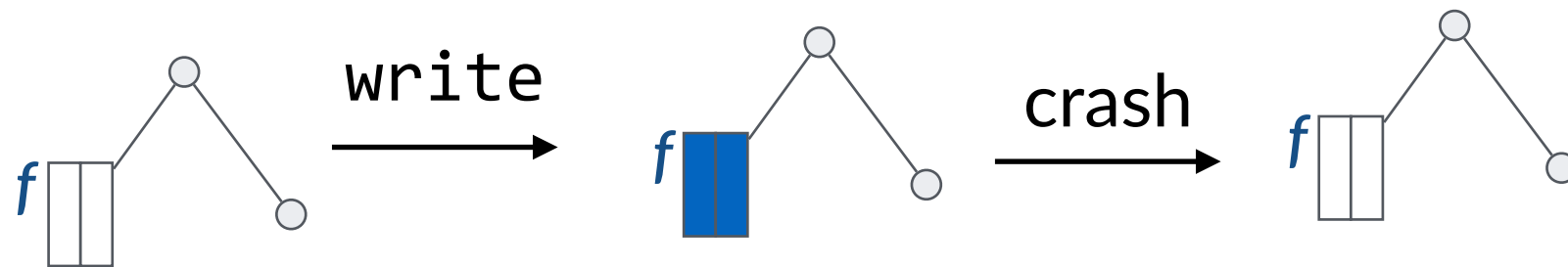
Specify that other files are unaffected



Puts an obligation on the implementation to avoid block re-use within a tree sequence

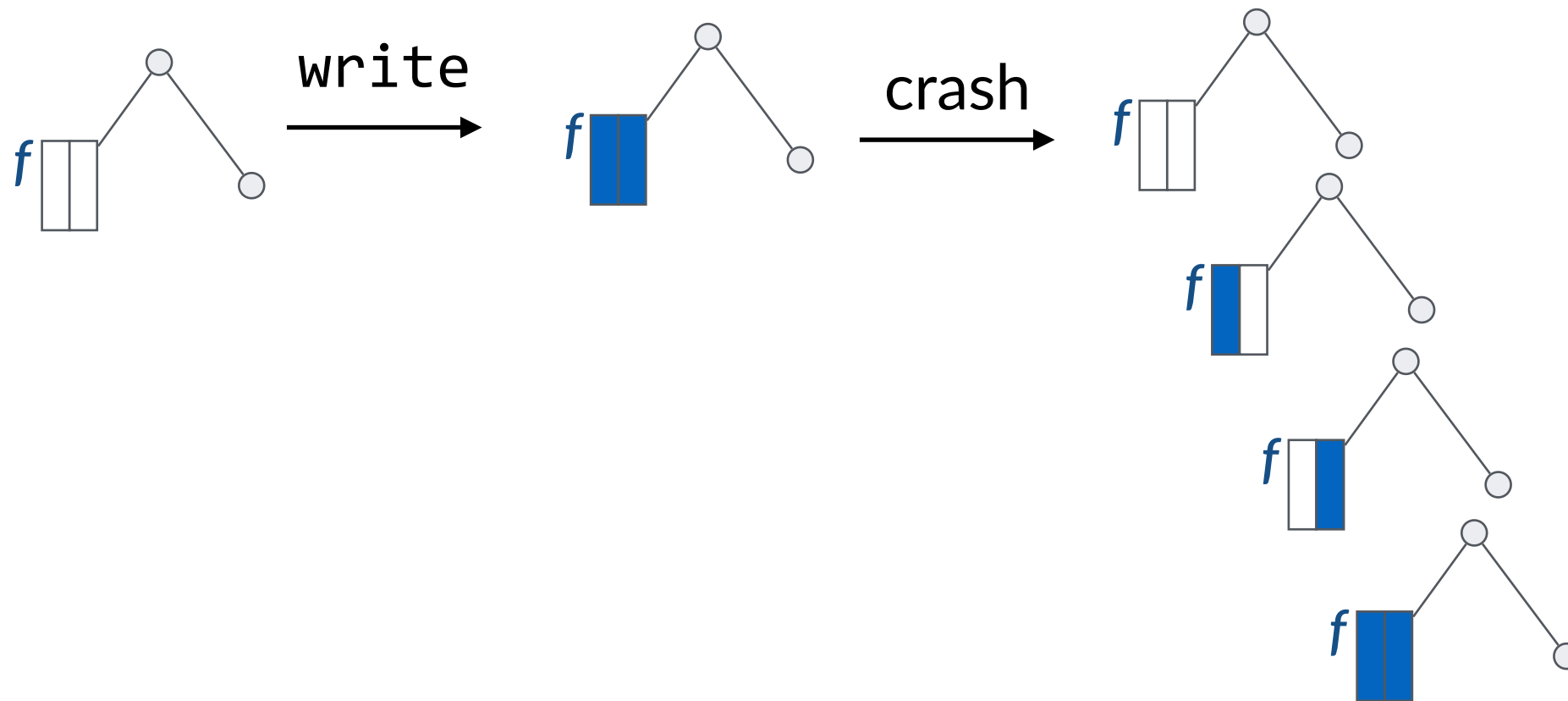
Problem 3: data writes are cached

- Write-back buffer cache

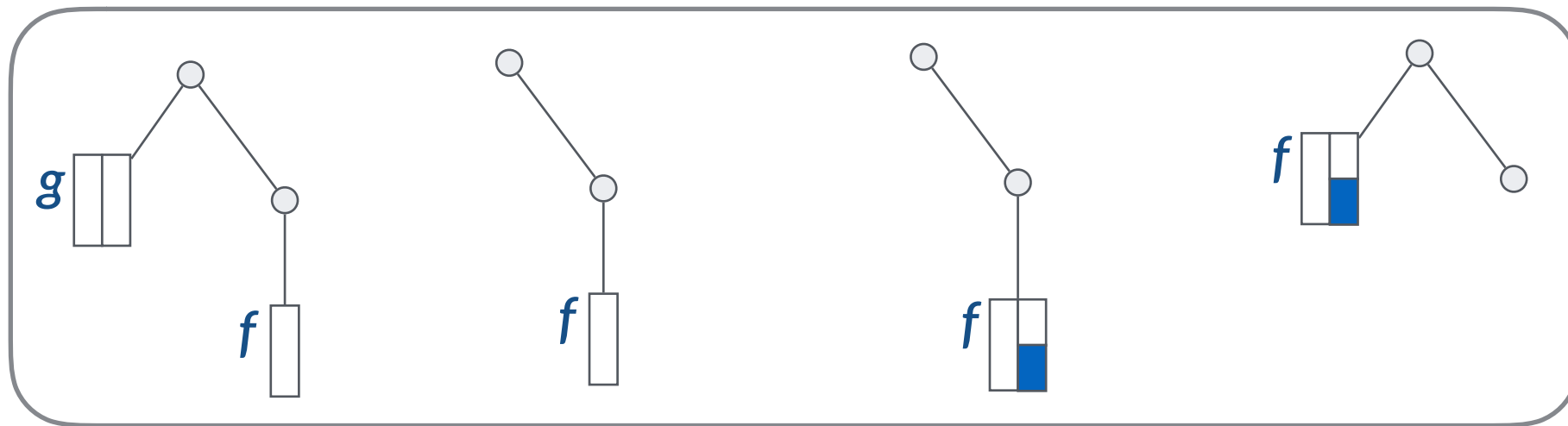


Problem 3: data writes are cached

- Write-back buffer cache
- Data can be persisted in any order



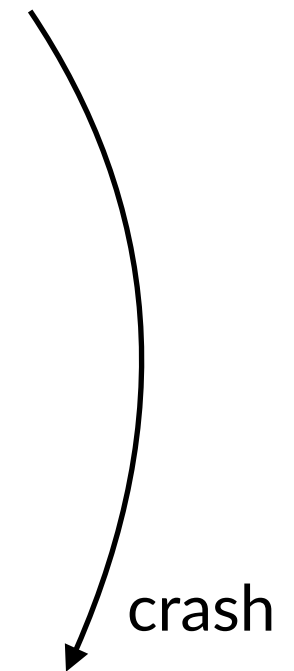
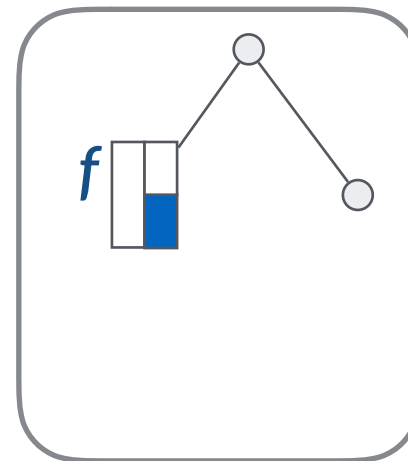
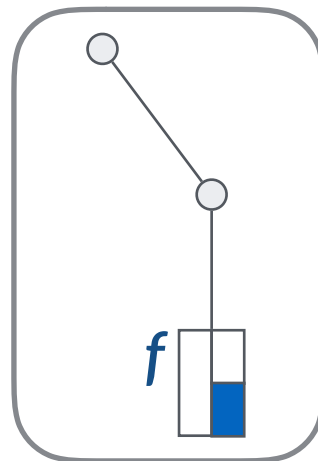
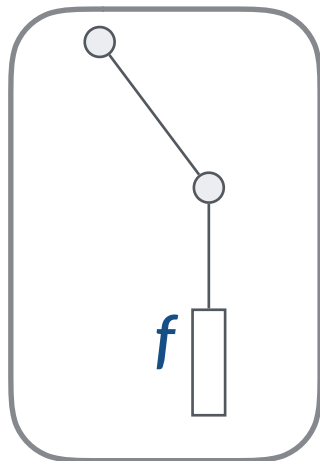
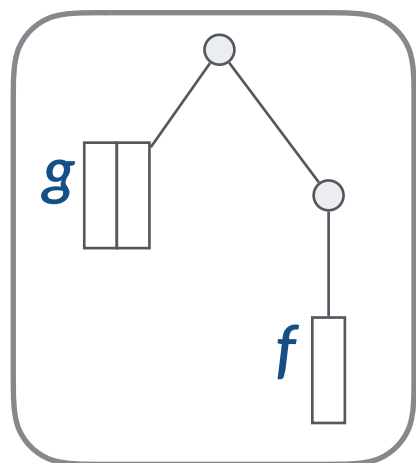
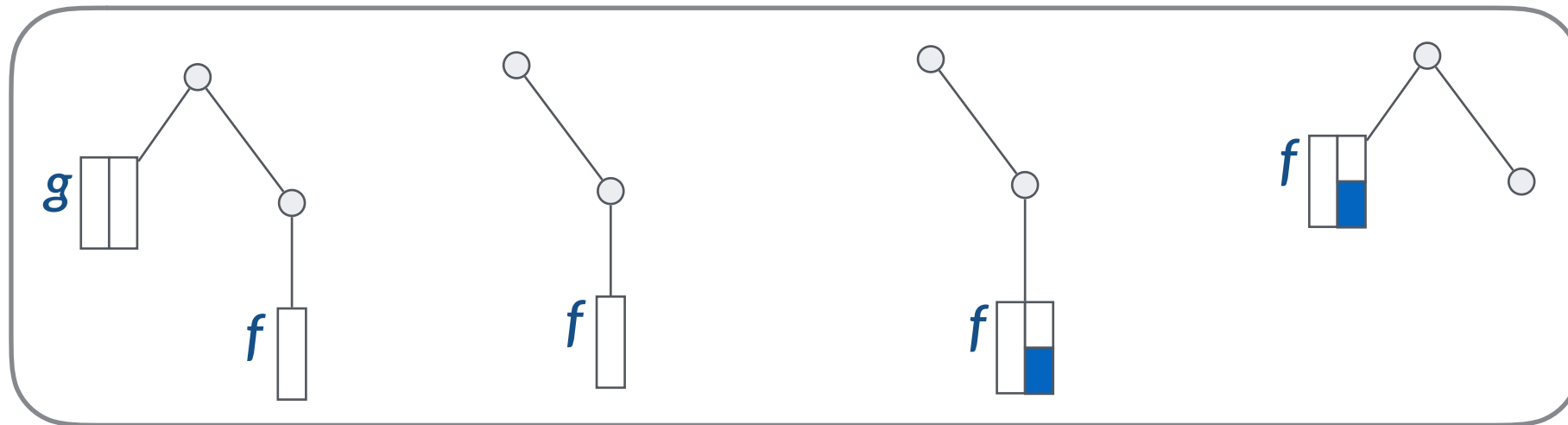
Specifying data caching: **block sets**



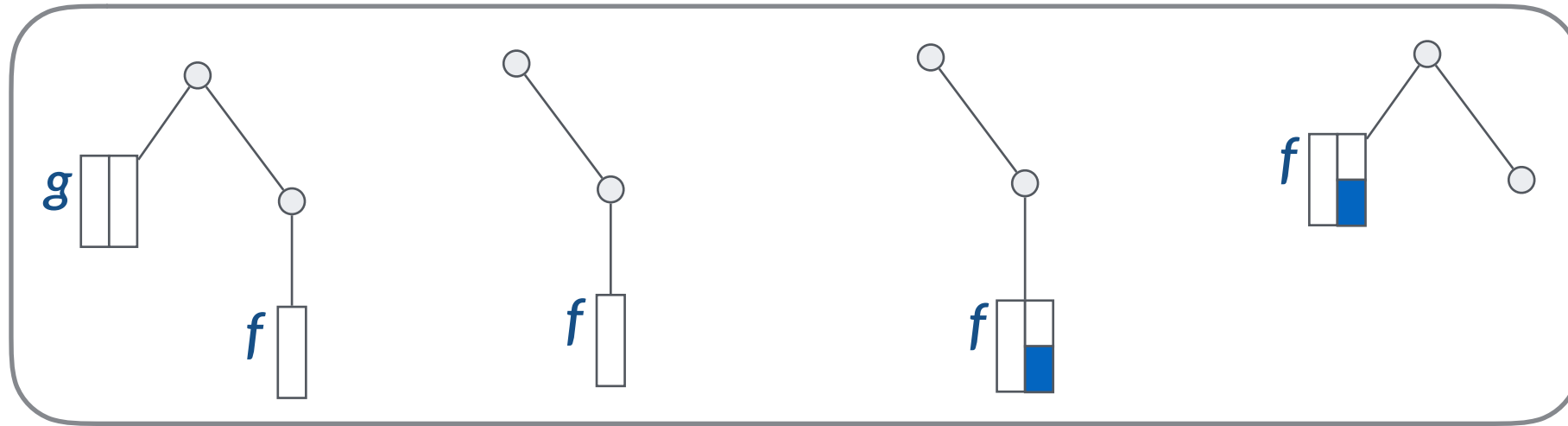
 uncached

 two possible values: old () and new ()

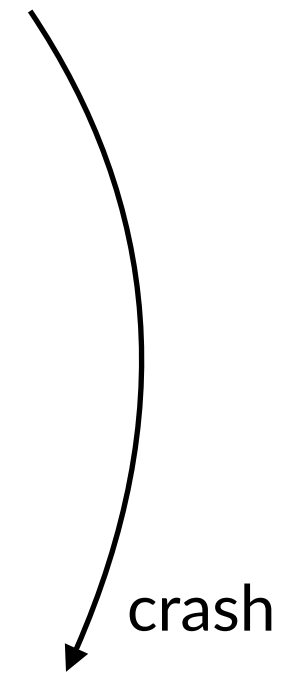
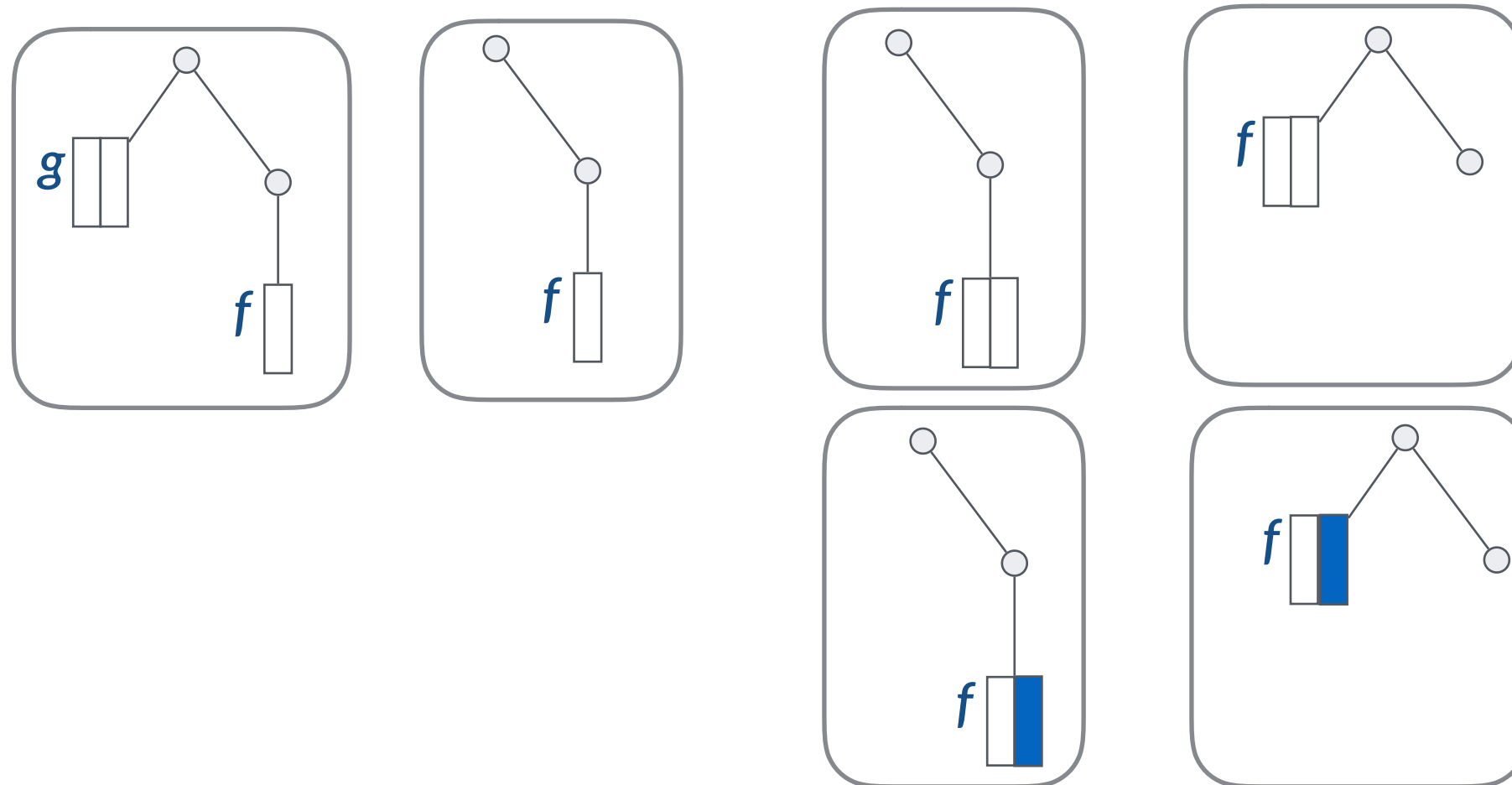
Behavior of block sets on crash



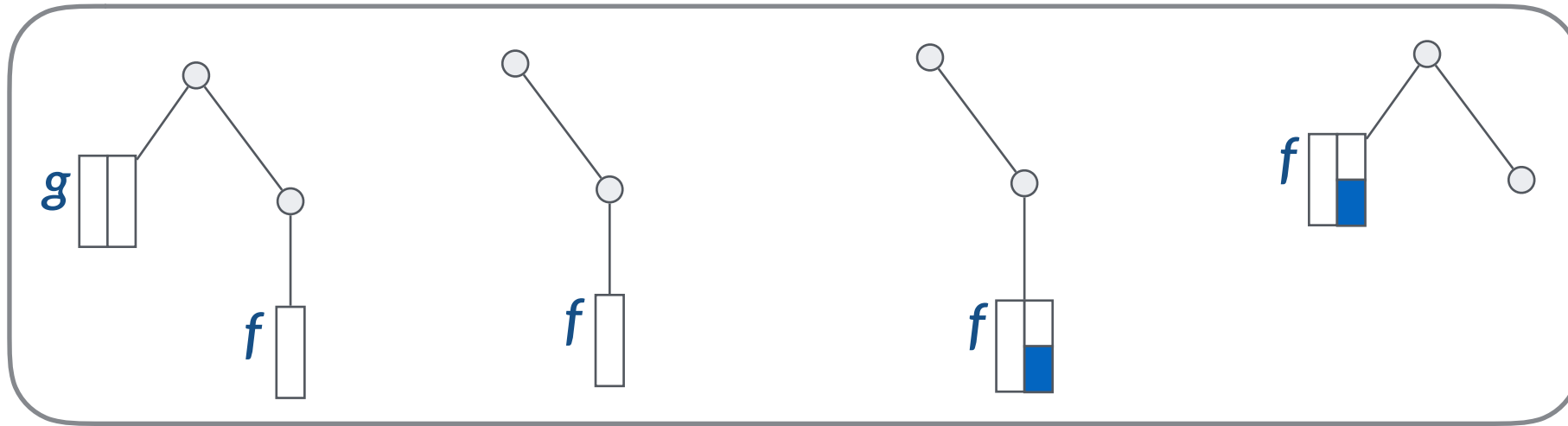
Behavior of block sets on crash



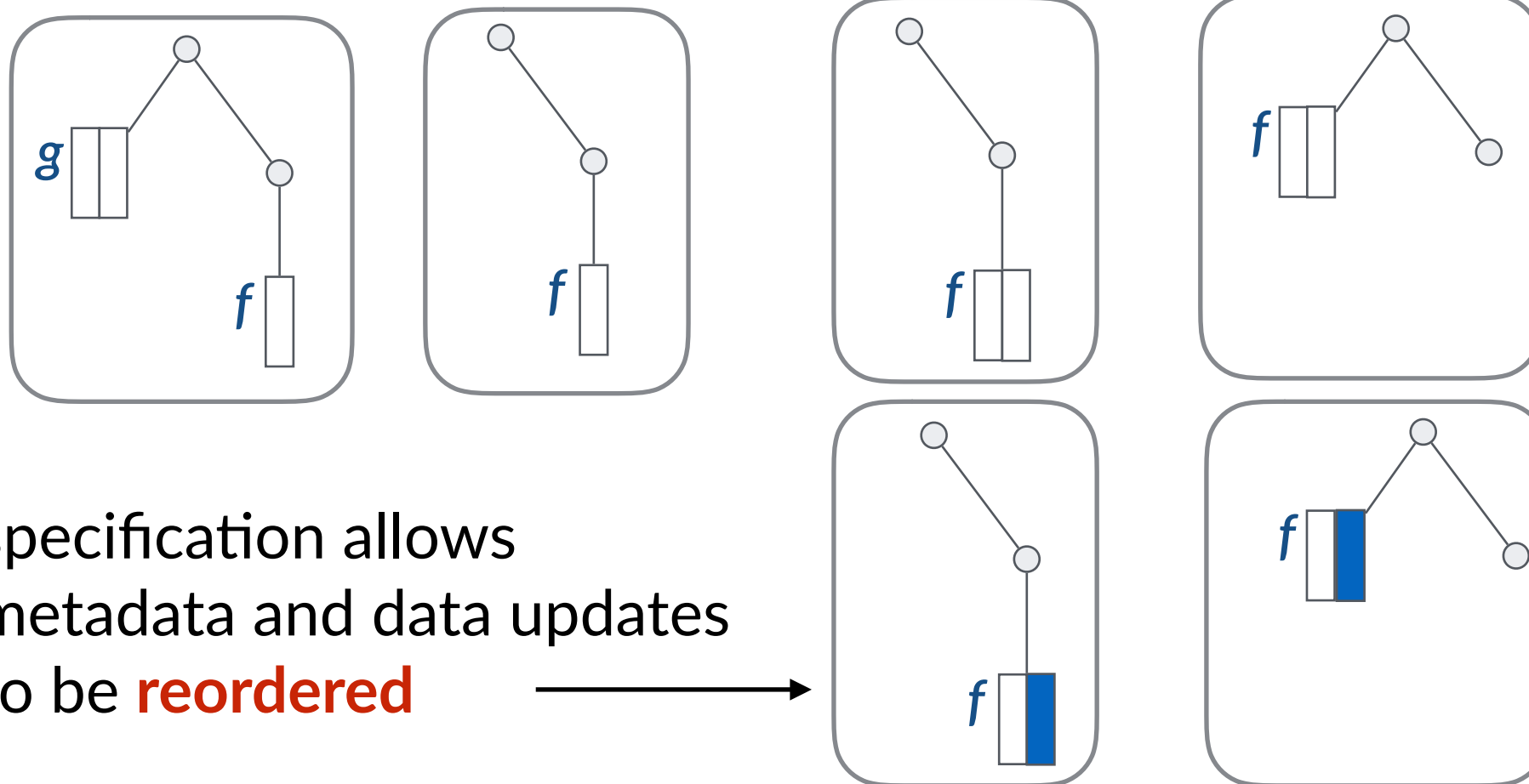
two degrees of non-determinism in crash states:



Behavior of block sets on crash



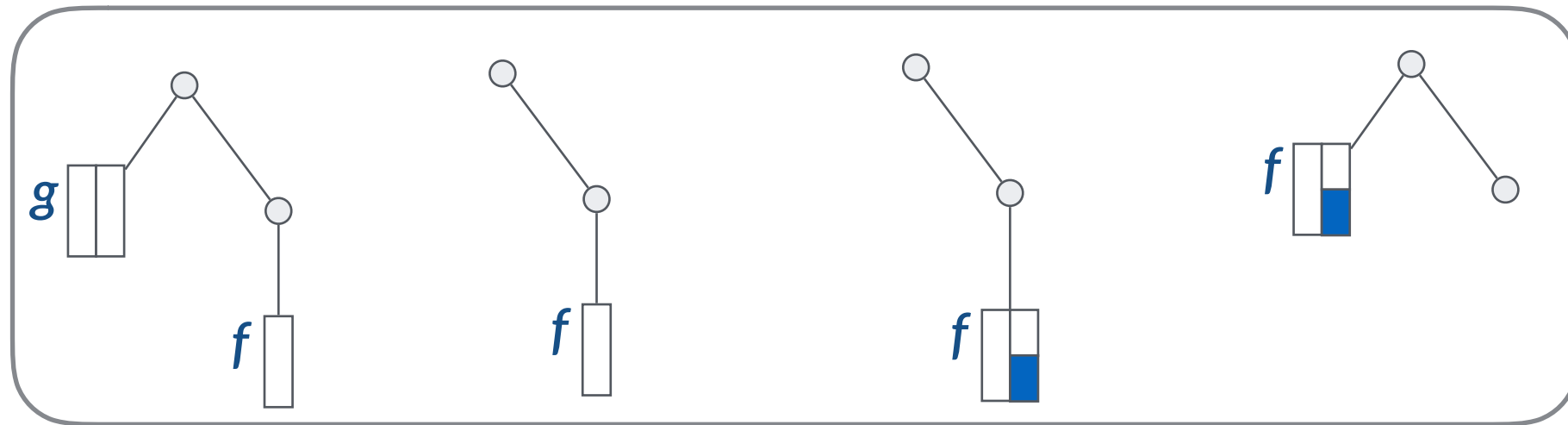
two degrees of non-determinism in crash states:



specification allows metadata and data updates to be **reordered**

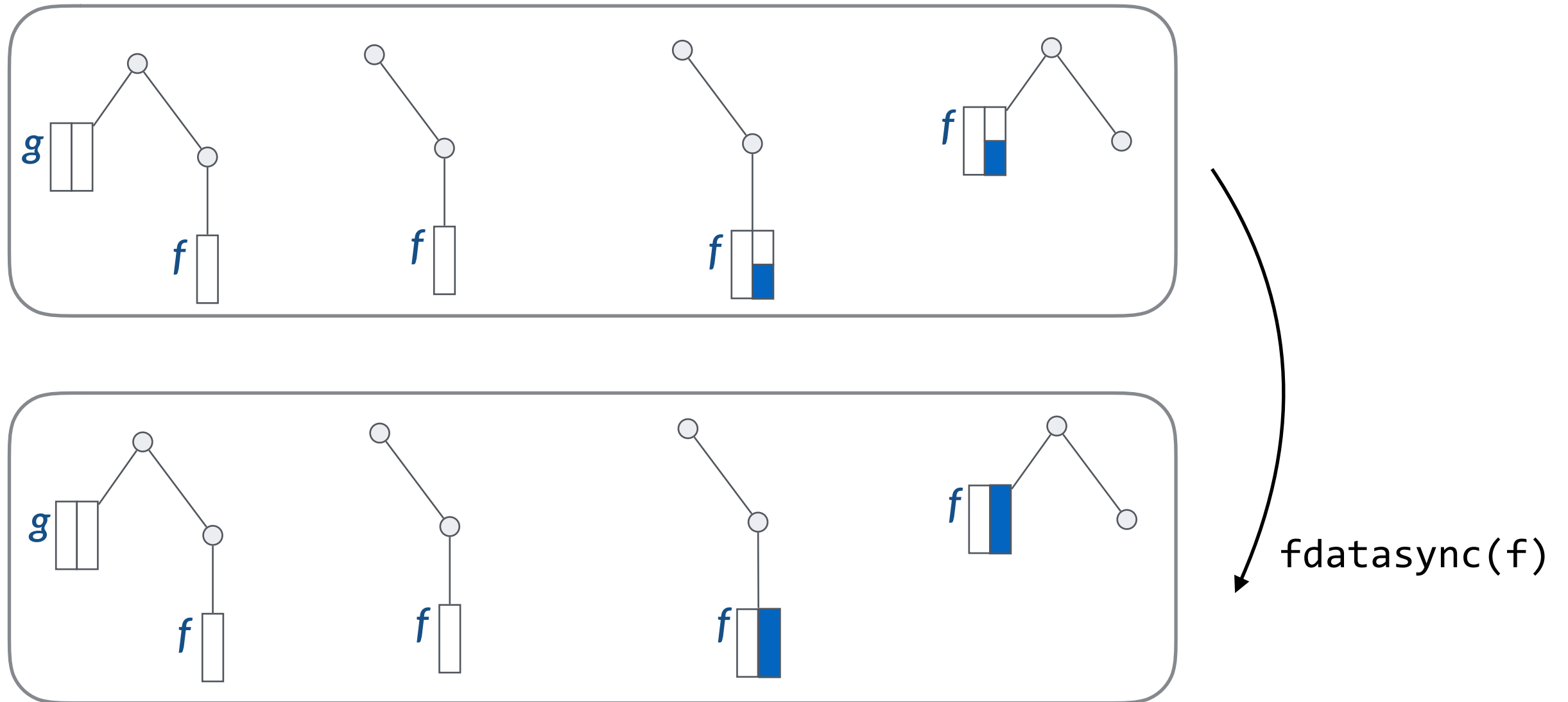
crash

Specification for **fdatasync**



↘
fdatasync(f)

Specification for **fdatasync**

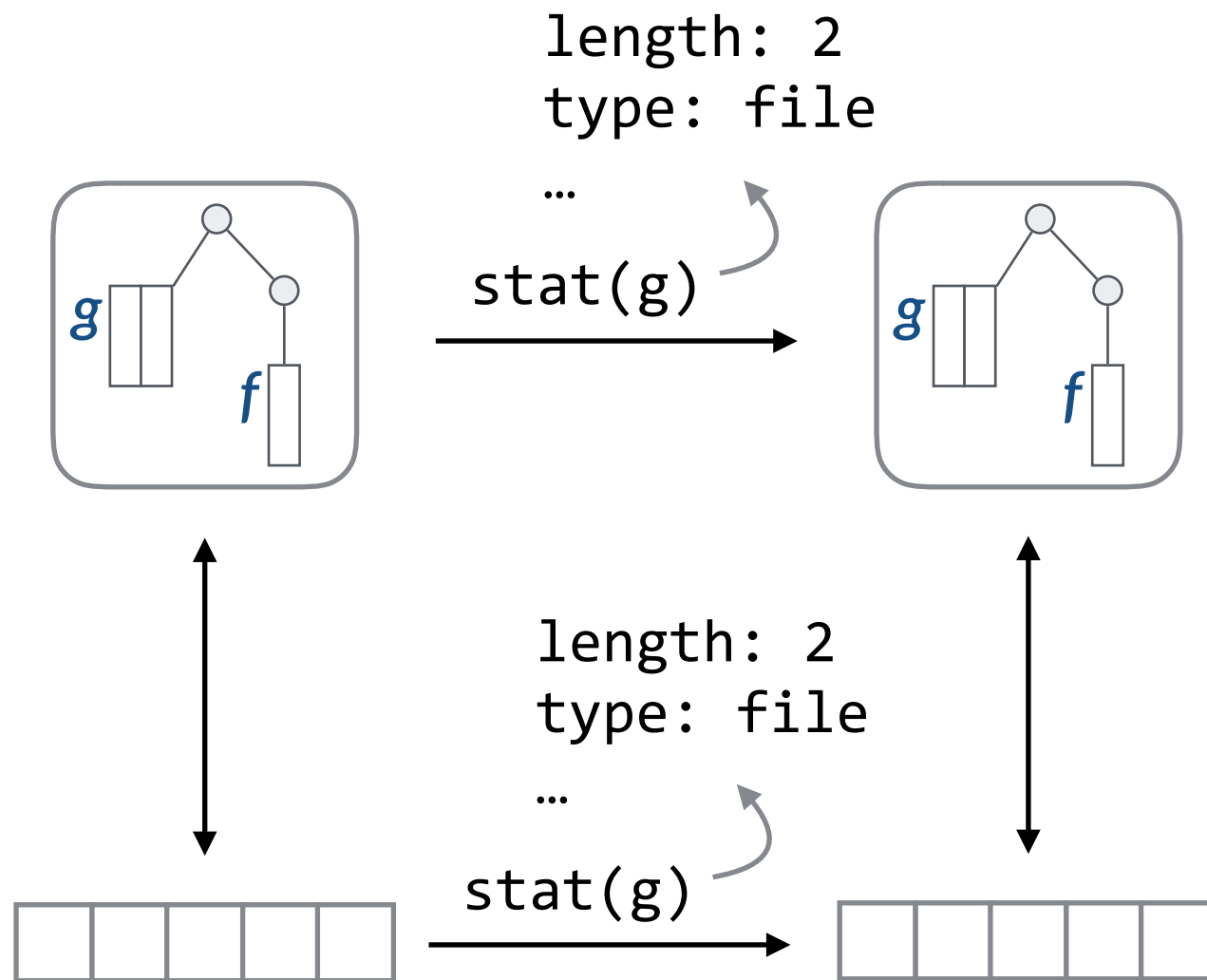


fdatasync specification says block sets collapse in every tree

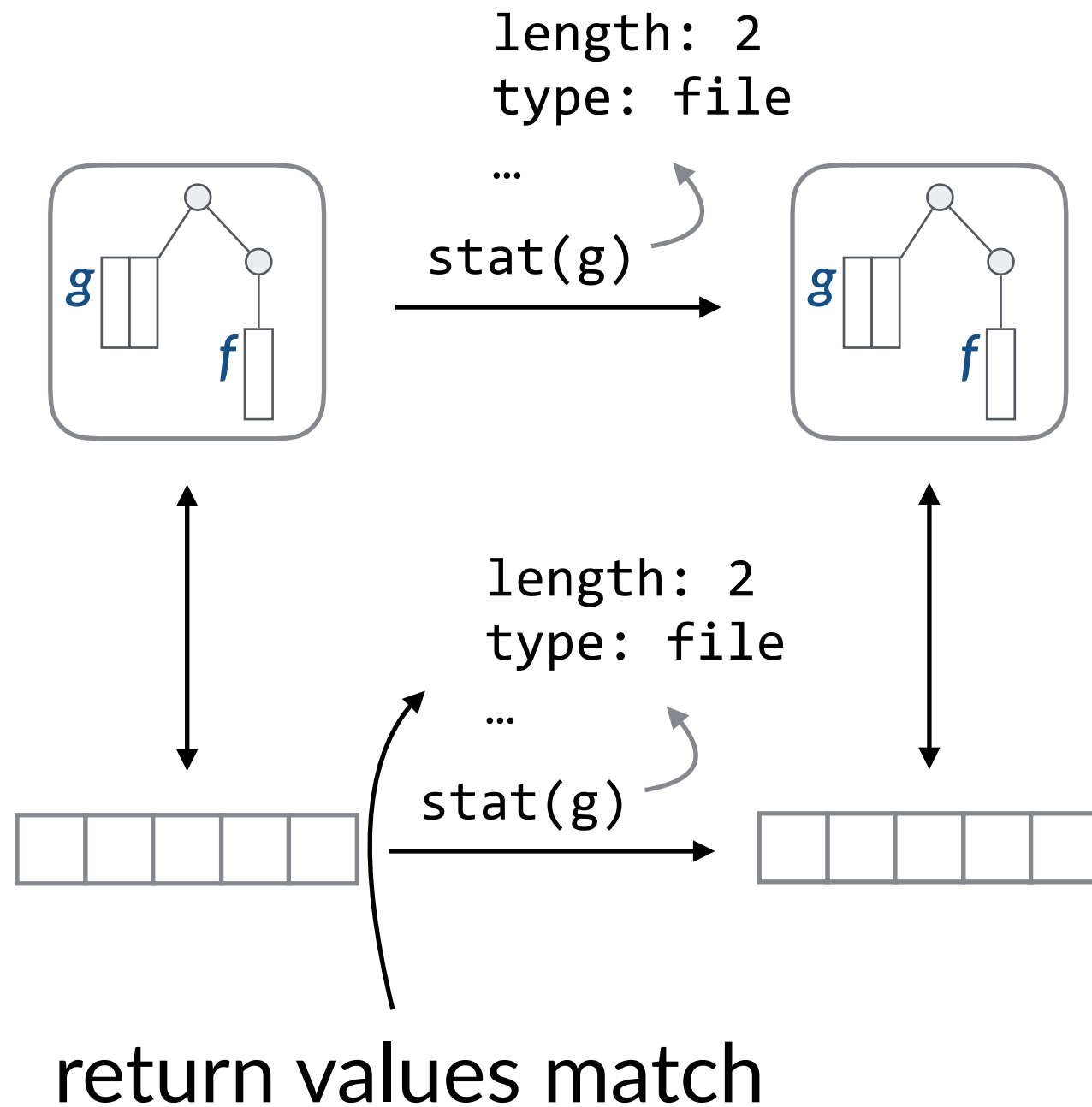
Summary: DFSCQ's tree-based specification

- metadata operations add a new tree
 - fsync collapses to latest tree
- writes update blocksets in every tree
 - fdatasync collapses blocksets in every tree

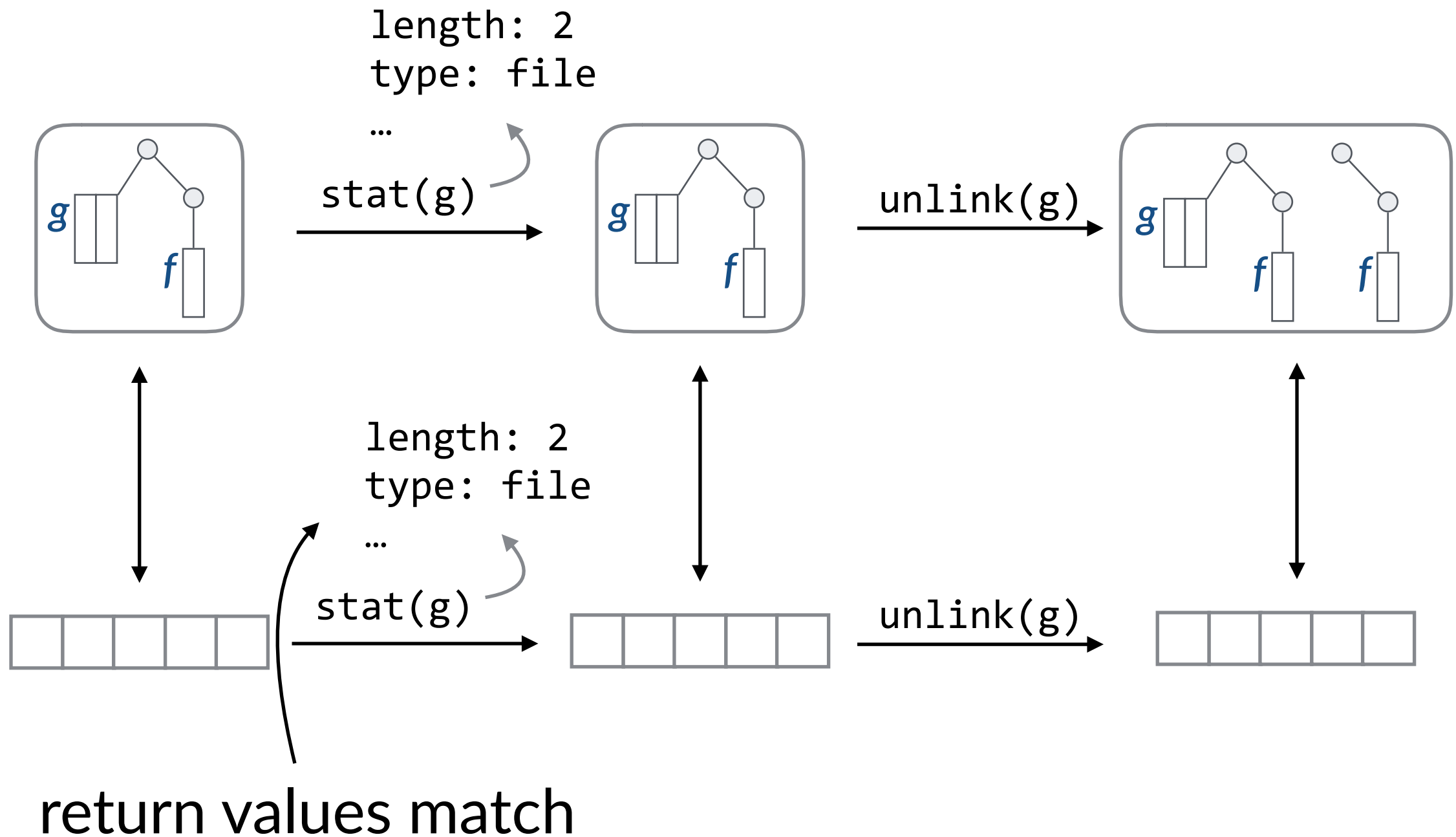
Prove implementation meets specification



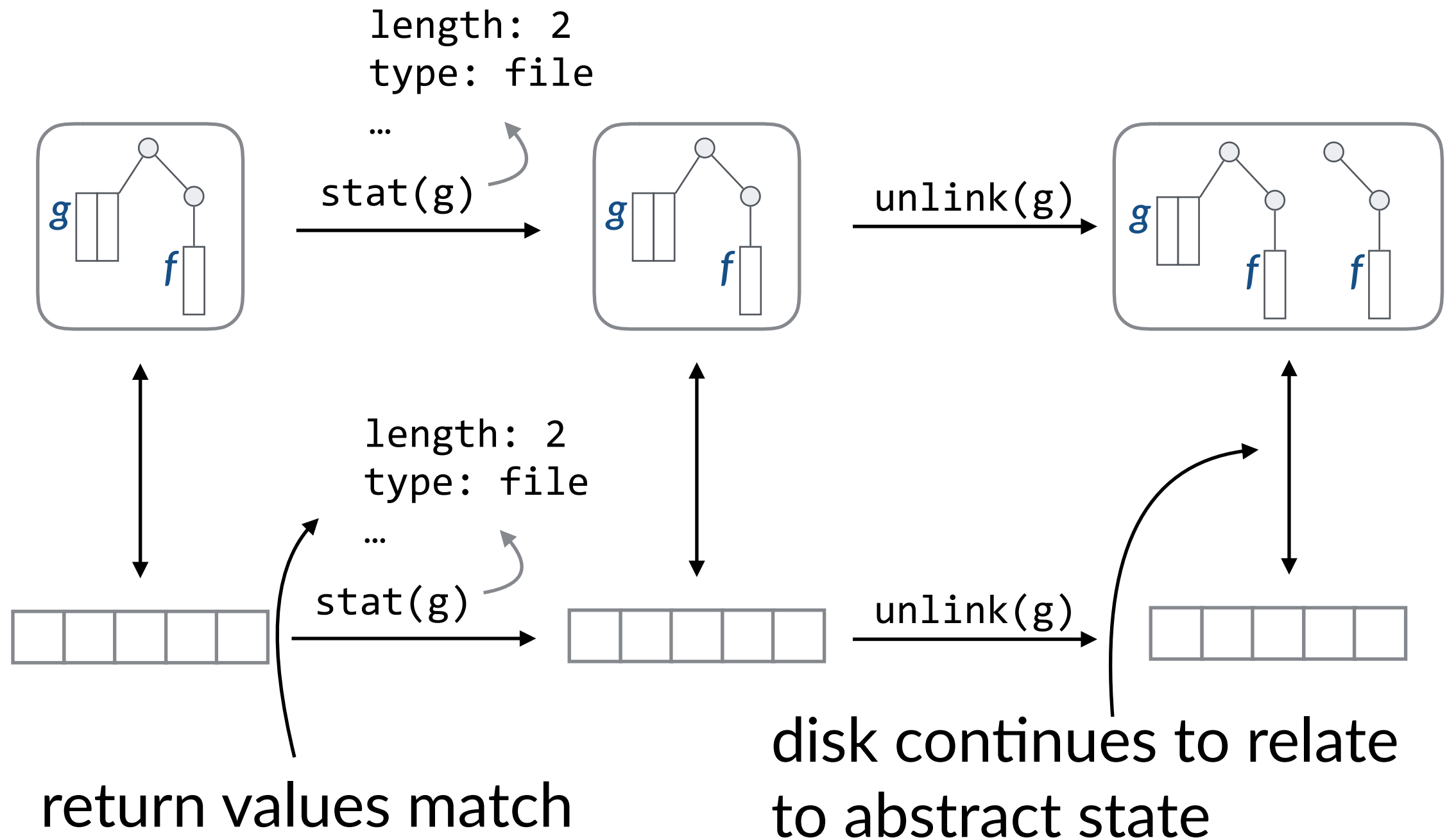
Prove implementation meets specification



Prove implementation meets specification



Prove implementation meets specification



DFSCQ Design

directory
name cache

inode
k-indirect blocks
dirty blocks

block allocator
free-bit cache
avoid re-use

logging
checksums
deferred commit
log-bypass API

buffer cache

Many single-layer optimizations

- Affect only proof of single layer

directory
name cache

inode
k-indirect blocks
dirty blocks

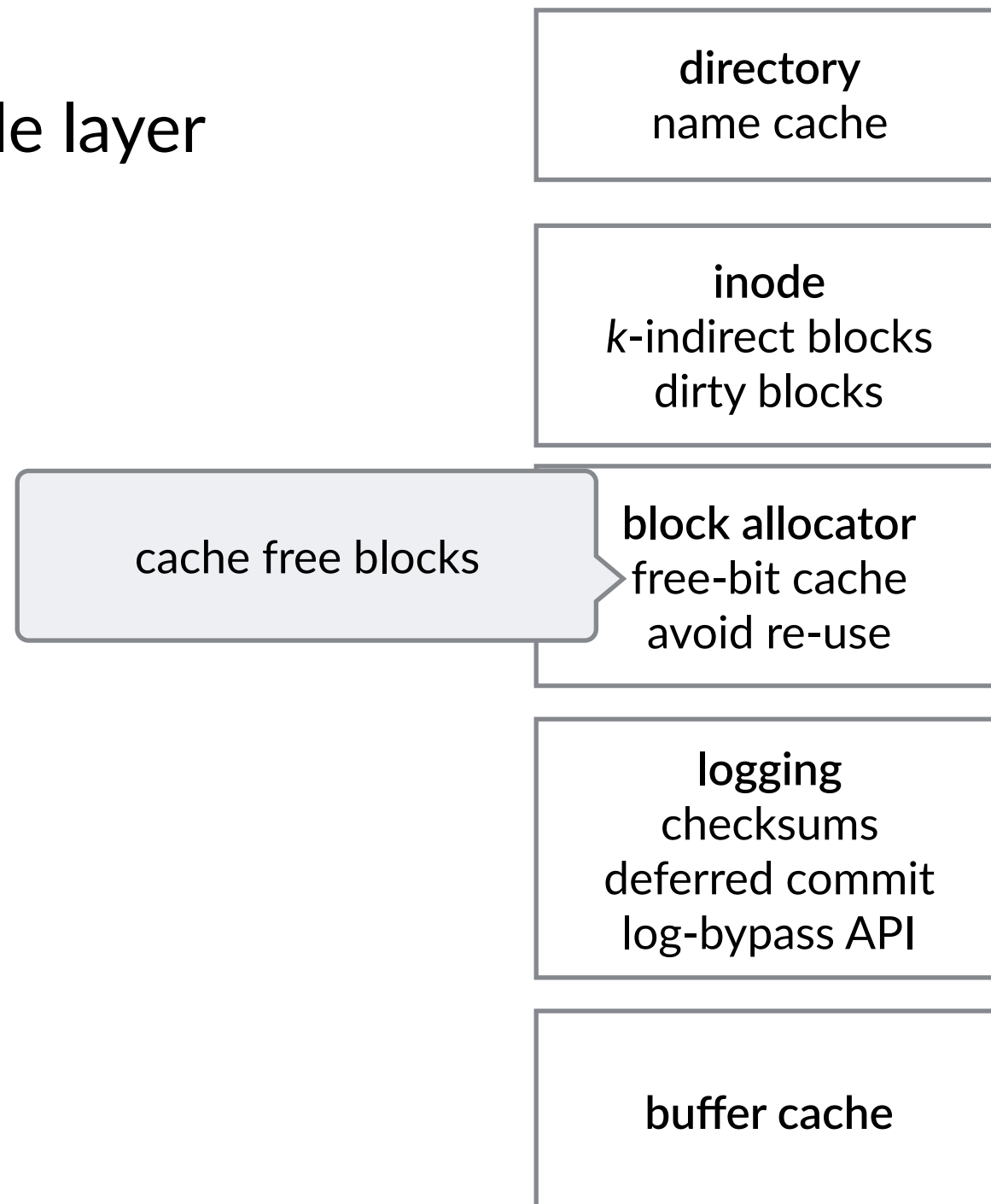
block allocator
free-bit cache
avoid re-use

logging
checksums
deferred commit
log-bypass API

buffer cache

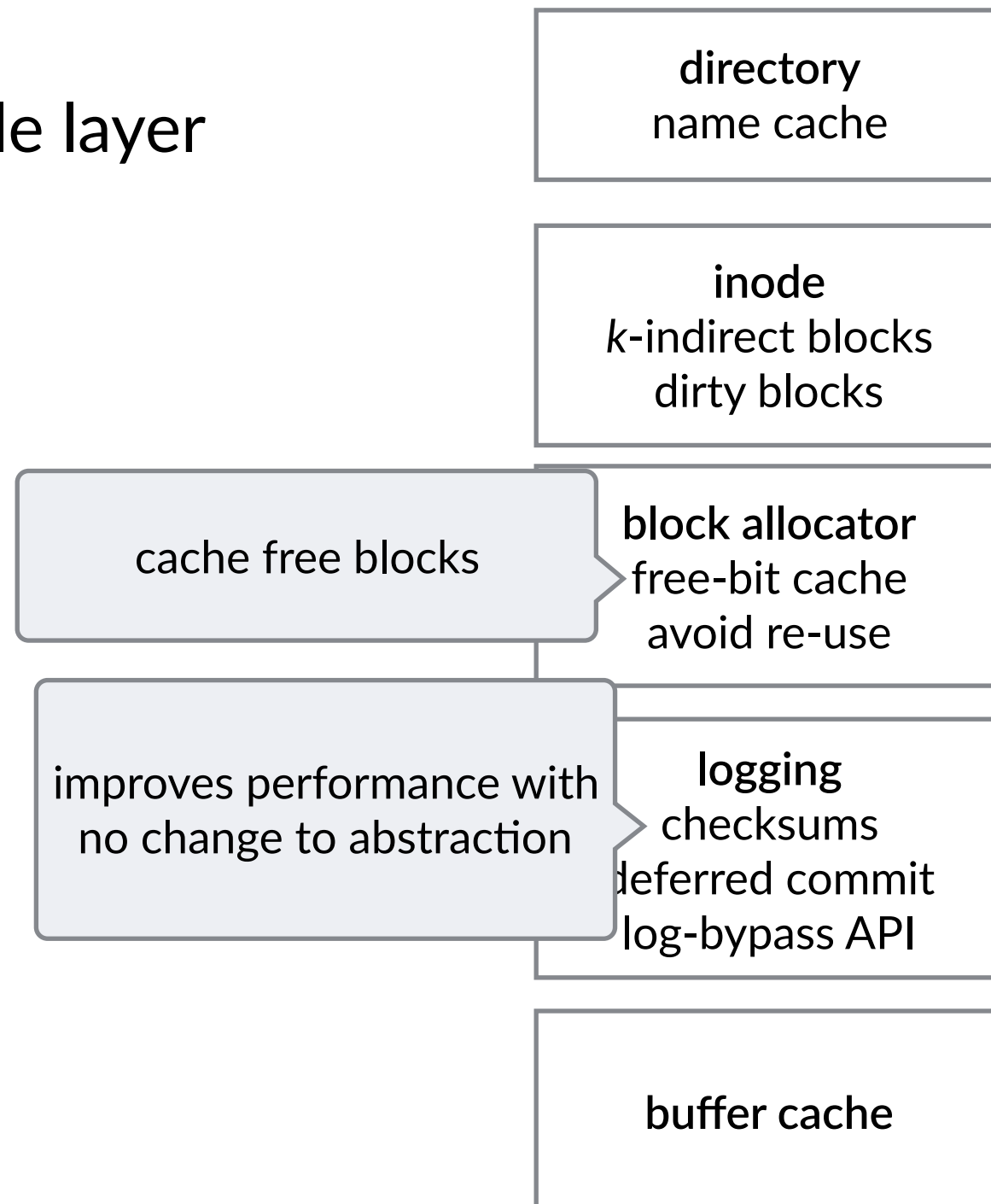
Many single-layer optimizations

- Affect only proof of single layer



Many single-layer optimizations

- Affect only proof of single layer



Cross-layer optimizations

- Break abstraction boundaries
- Complicate proofs
- Good for performance

directory
name cache

inode
k-indirect blocks
dirty blocks

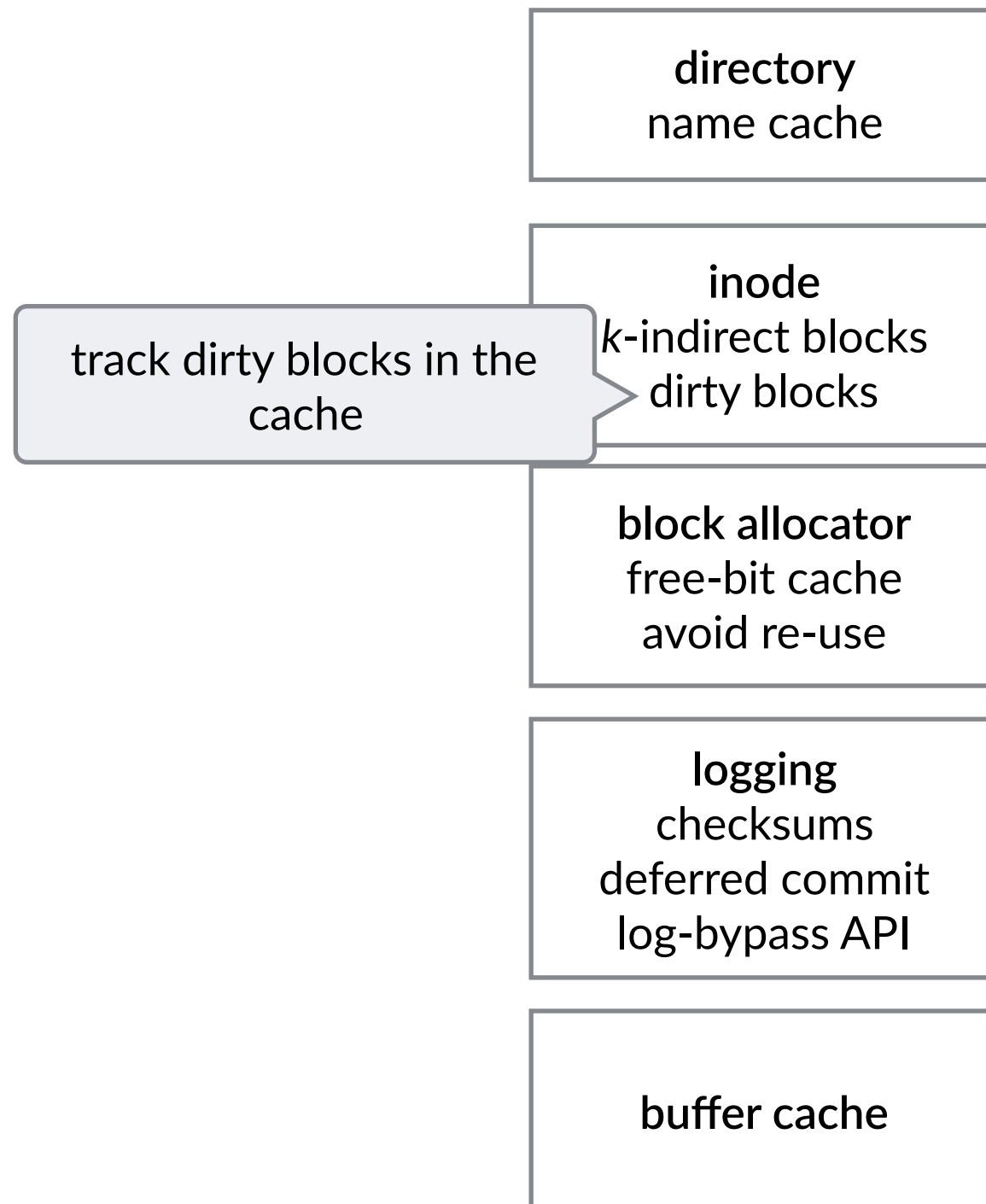
block allocator
free-bit cache
avoid re-use

logging
checksums
deferred commit
log-bypass API

buffer cache

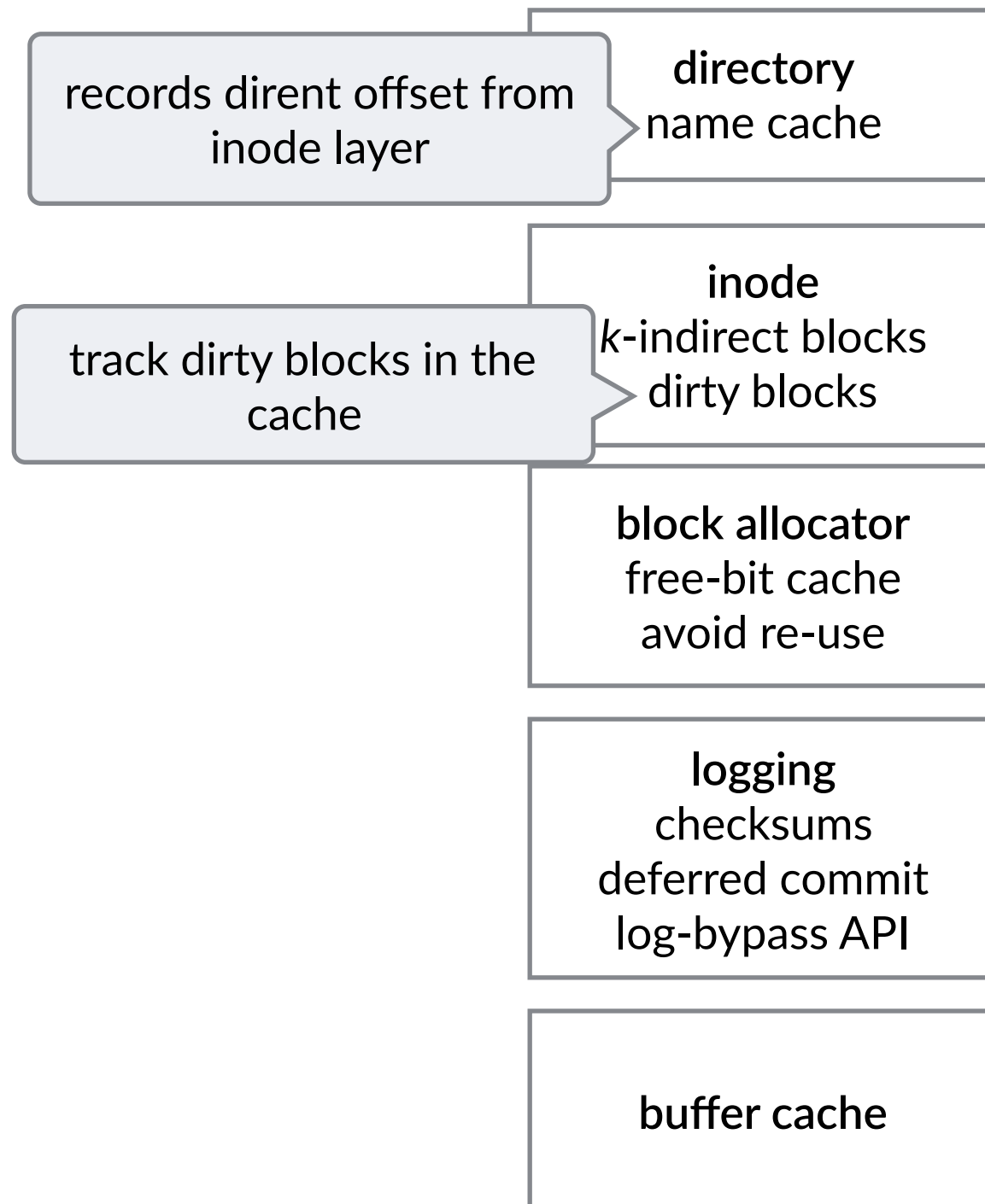
Cross-layer optimizations

- Break abstraction boundaries
- Complicate proofs
- Good for performance



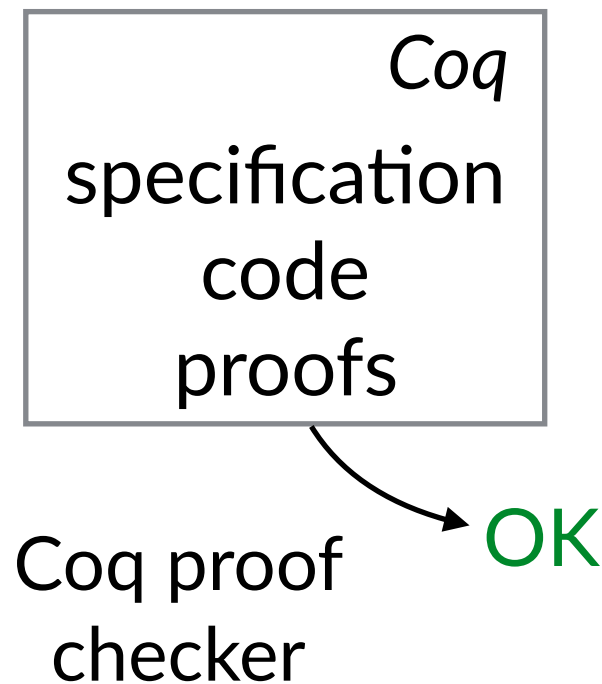
Cross-layer optimizations

- Break abstraction boundaries
- Complicate proofs
- Good for performance

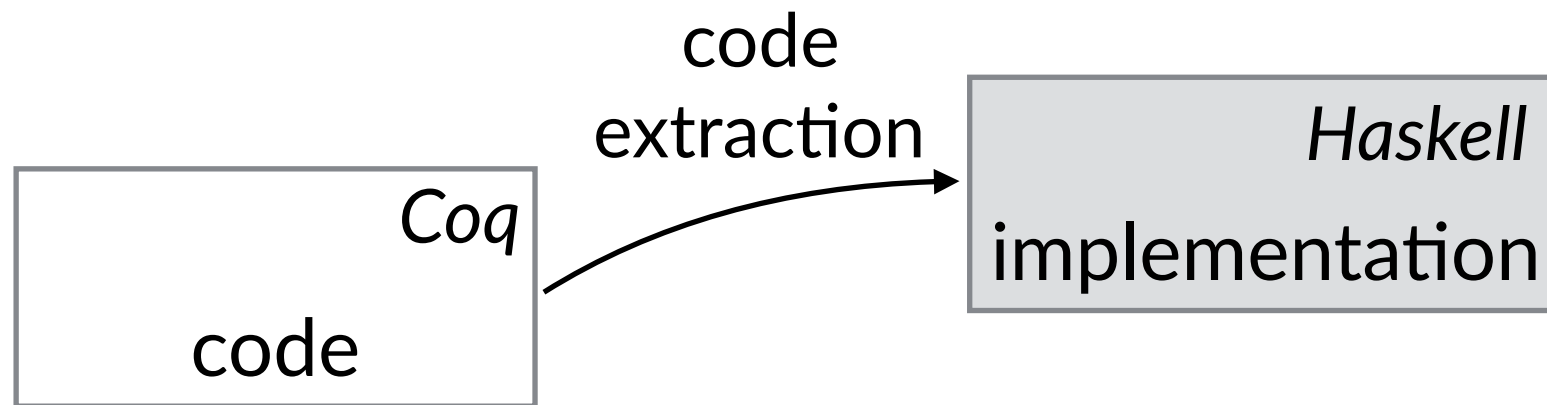


Implementation and proof

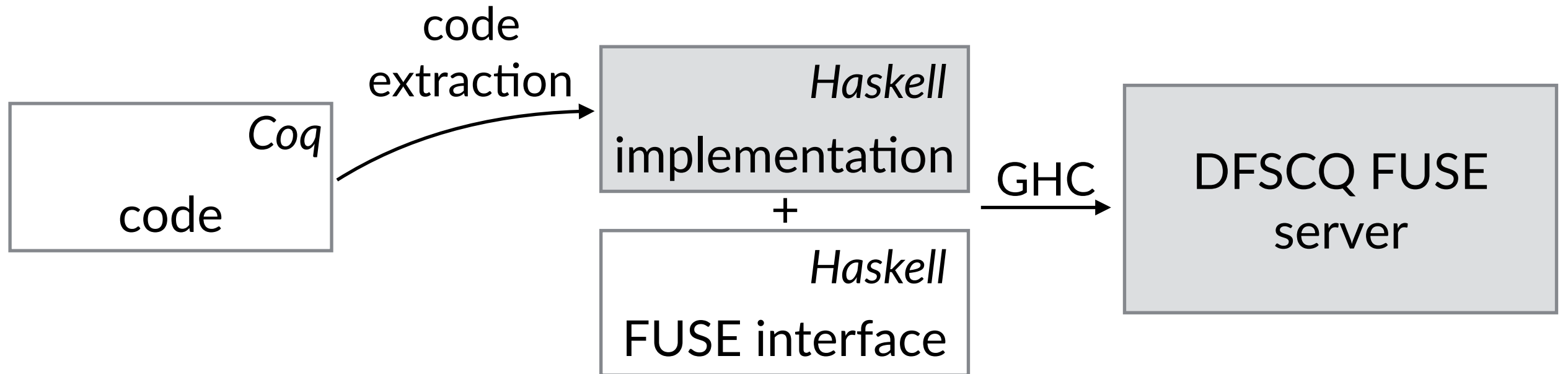
- Extend FSCQ [SOSP '15]
- 75,000 lines of Coq (compared to 31,000 in FSCQ)



Running DFSCQ



Running DFSCQ

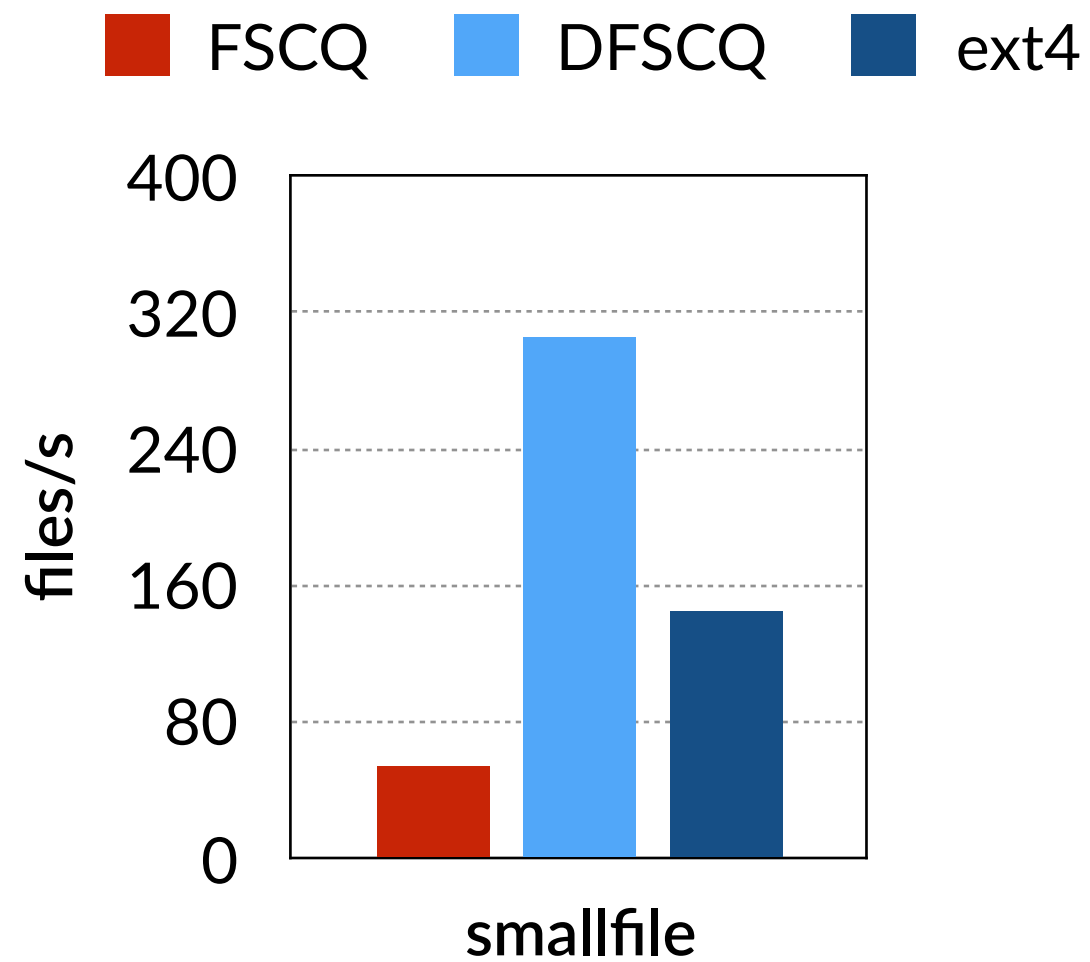


Performance evaluation

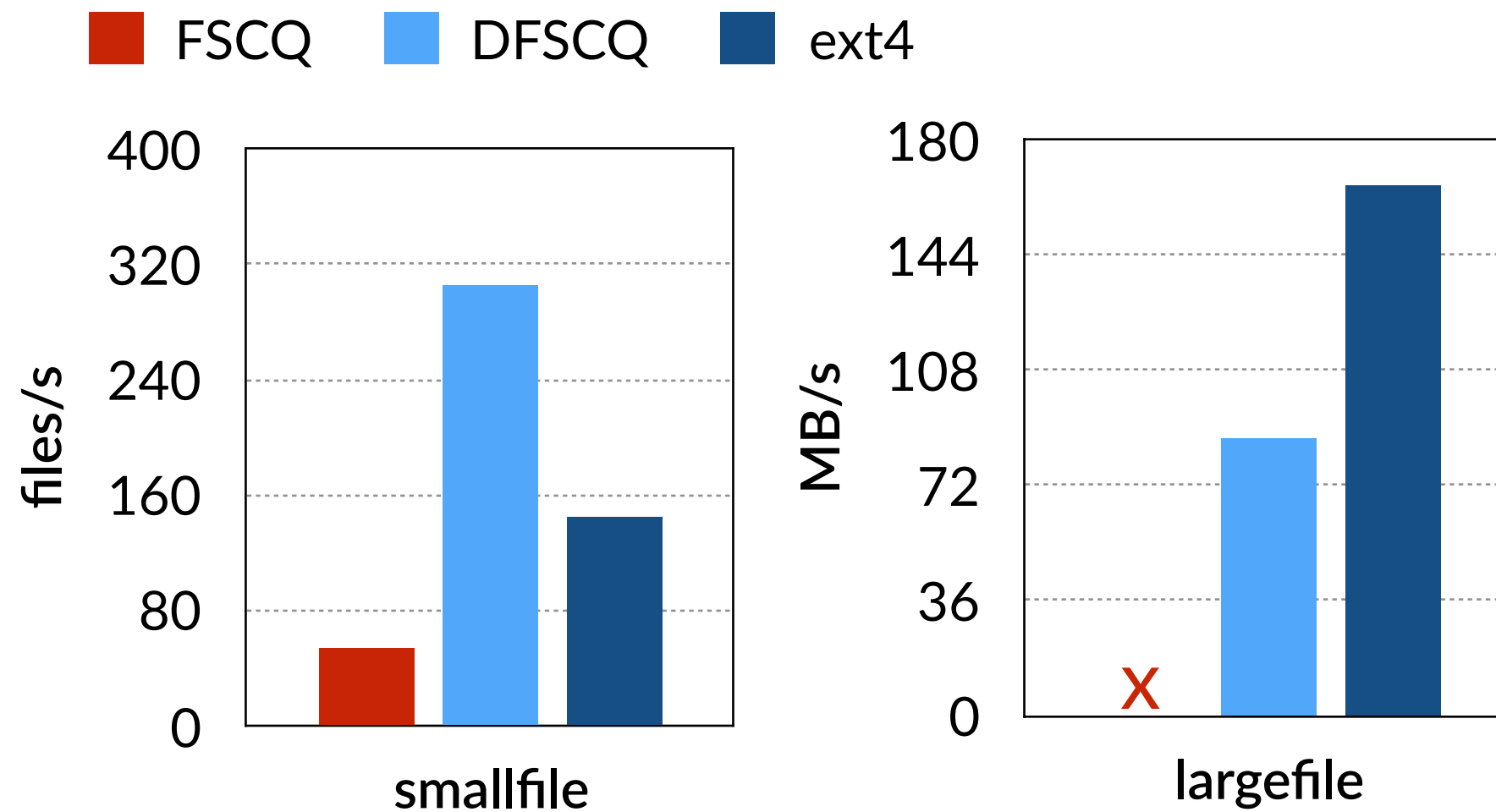
- Several workloads
 - micro benchmarks
 - application workloads
- Compare with ext4 in default mode
- Running on an SSD on a desktop

(see paper for more results)

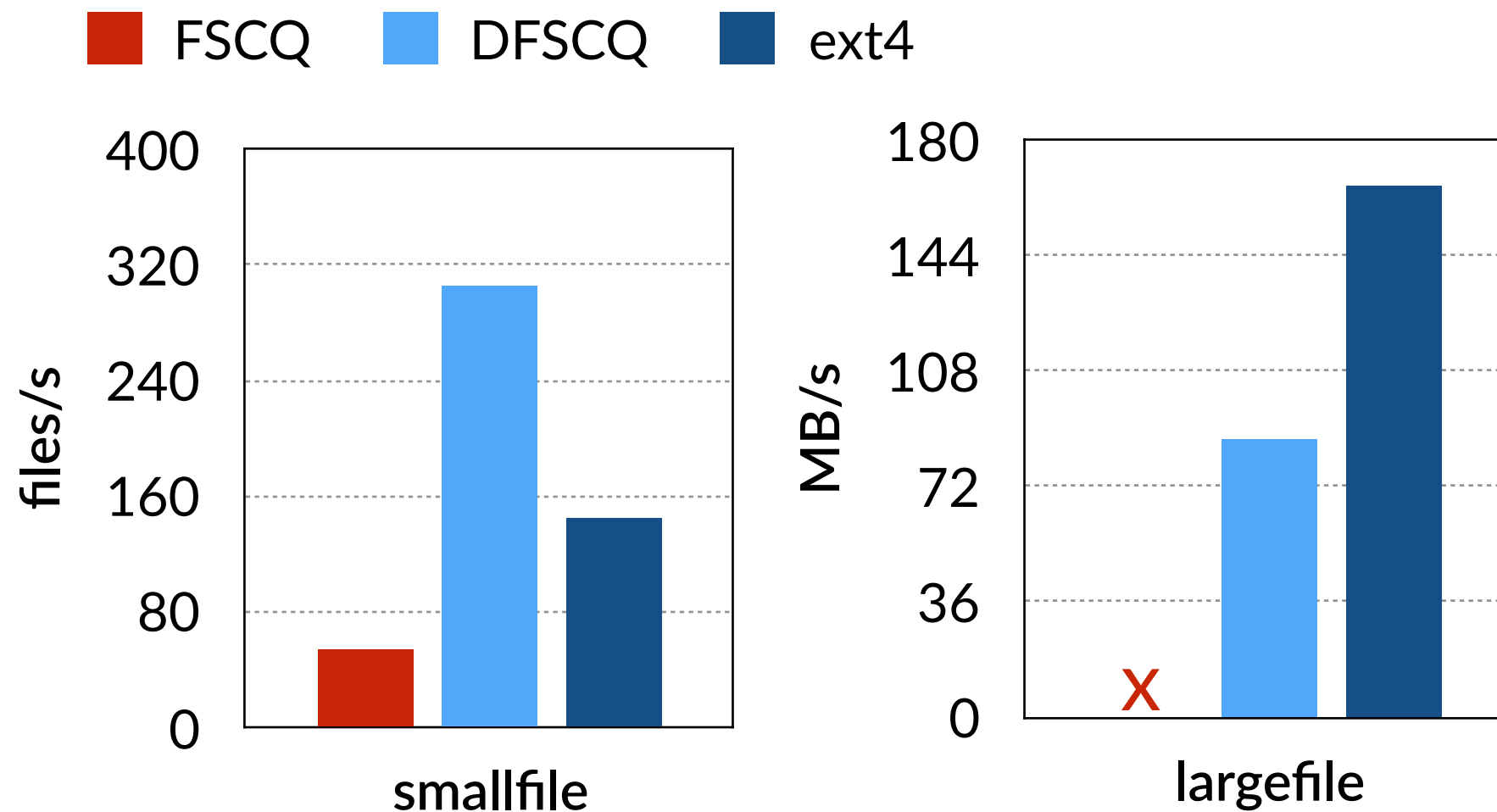
DFSCQ is competitive with ext4



DFSCQ is competitive with ext4

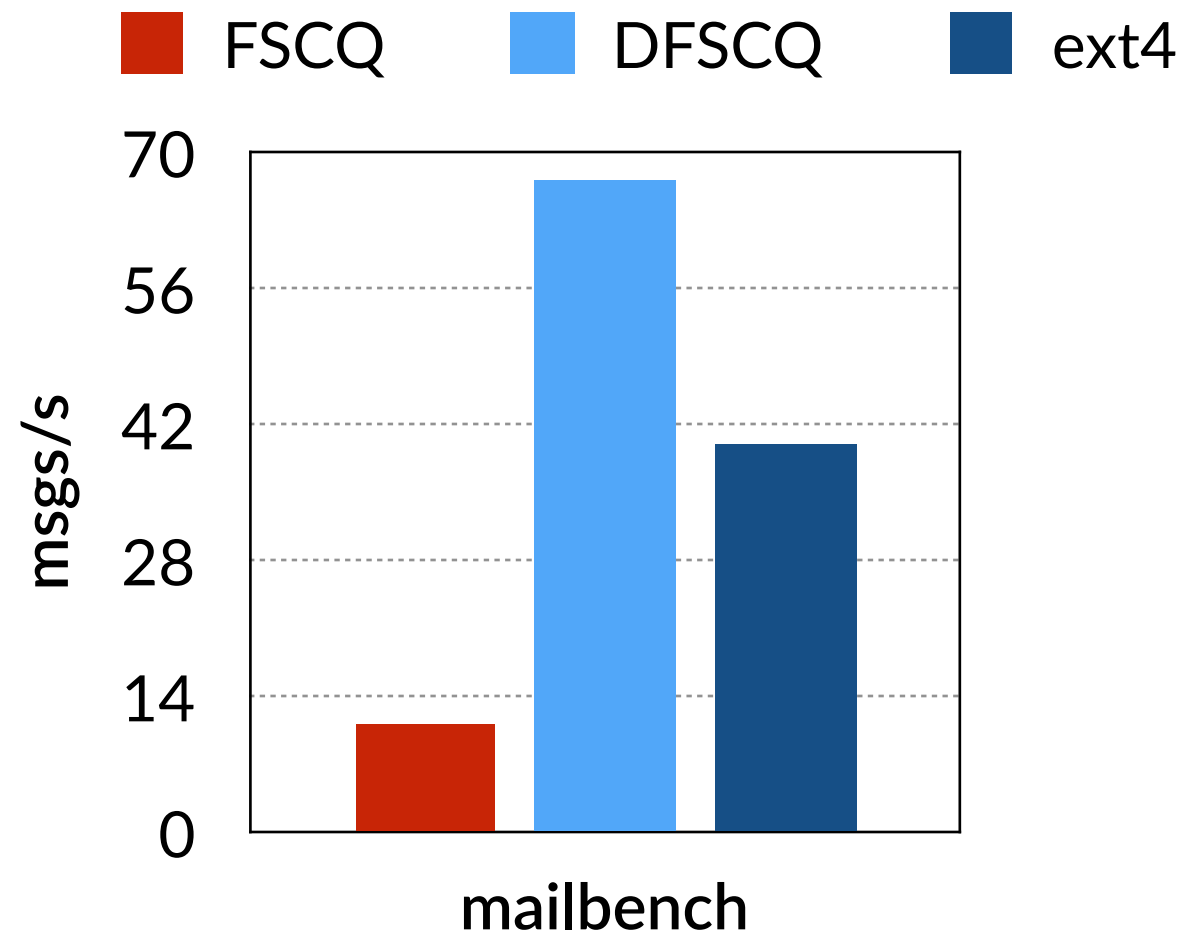


DFSCQ is competitive with ext4

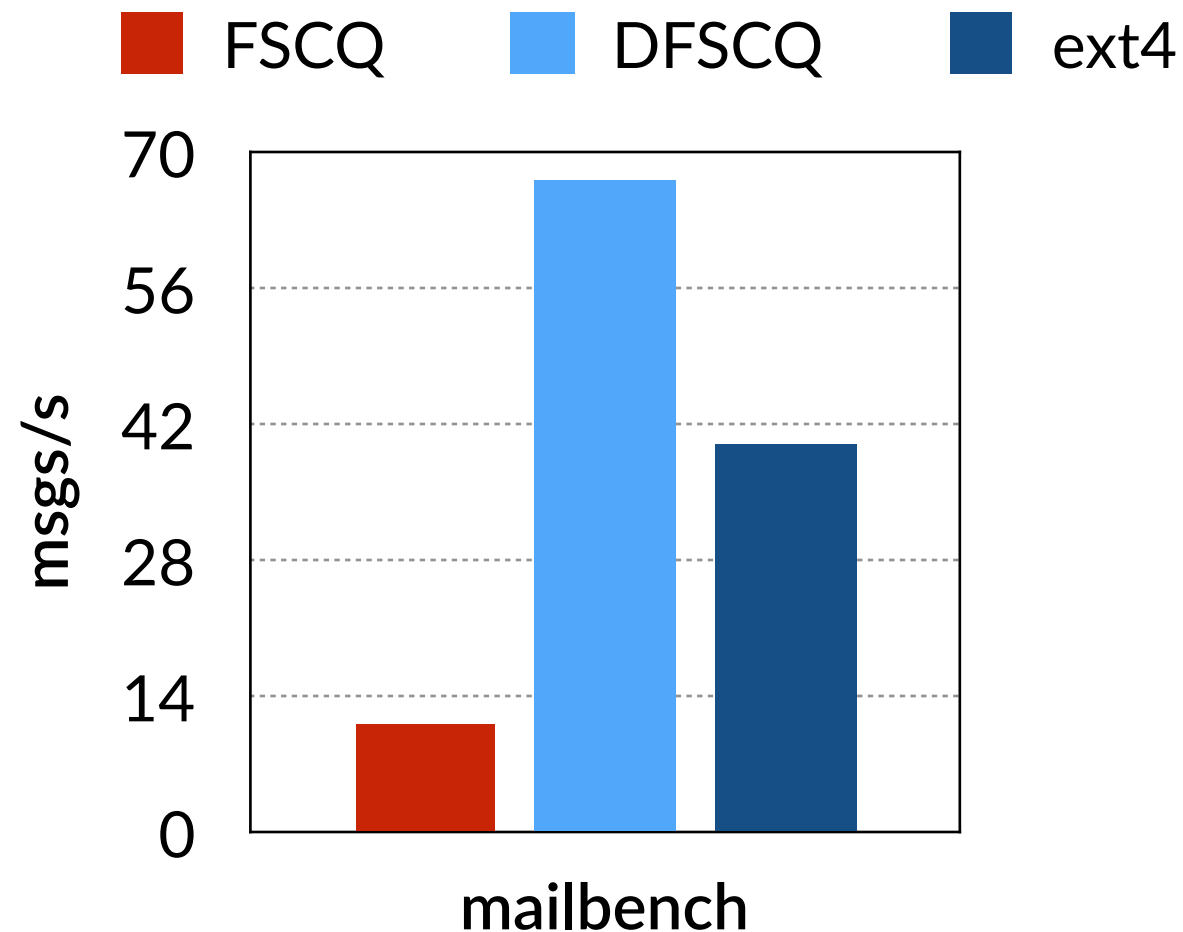


- DFSCQ still has high CPU overhead compared to ext4
 - Haskell code allocates large amounts of memory

DFSCQ outperforms ext4 on mailbench

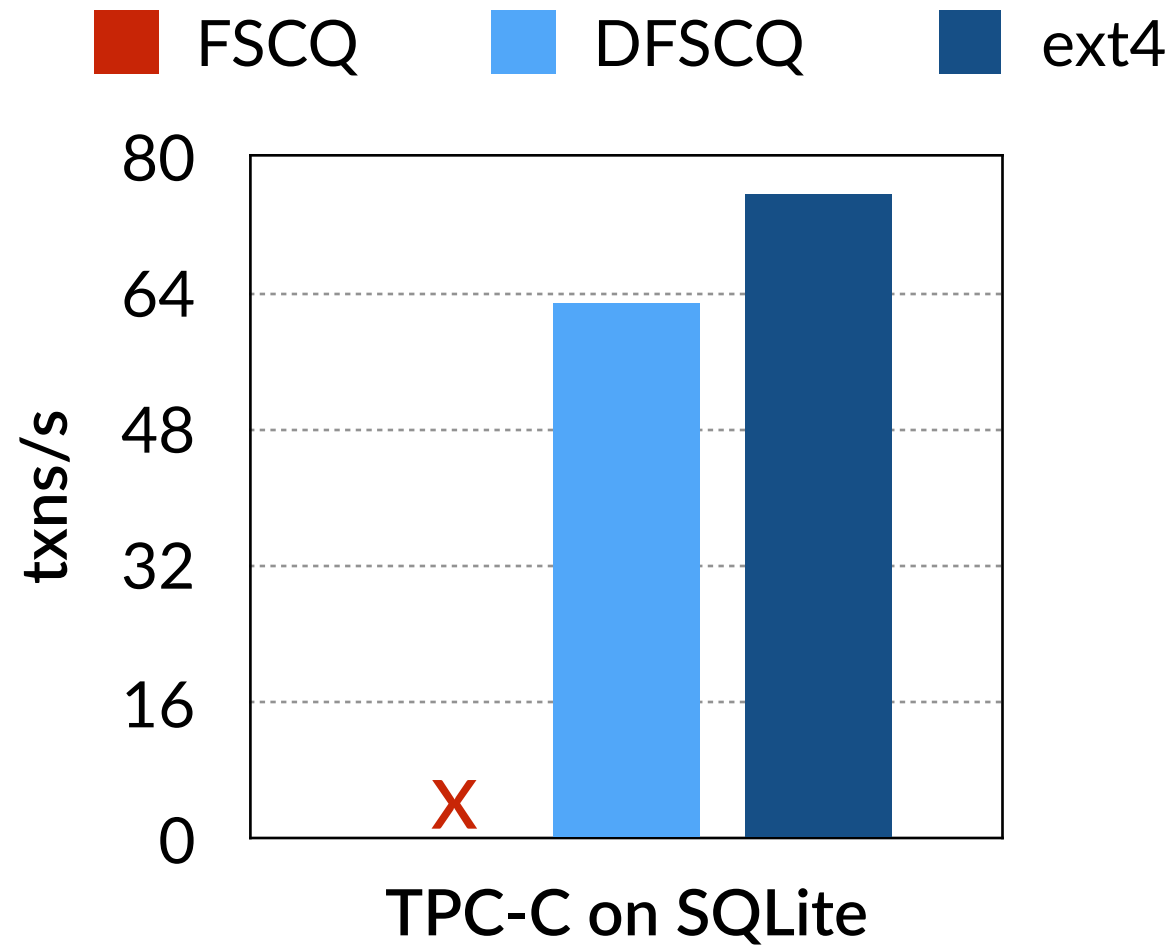


DFSCQ outperforms ext4 on mailbench

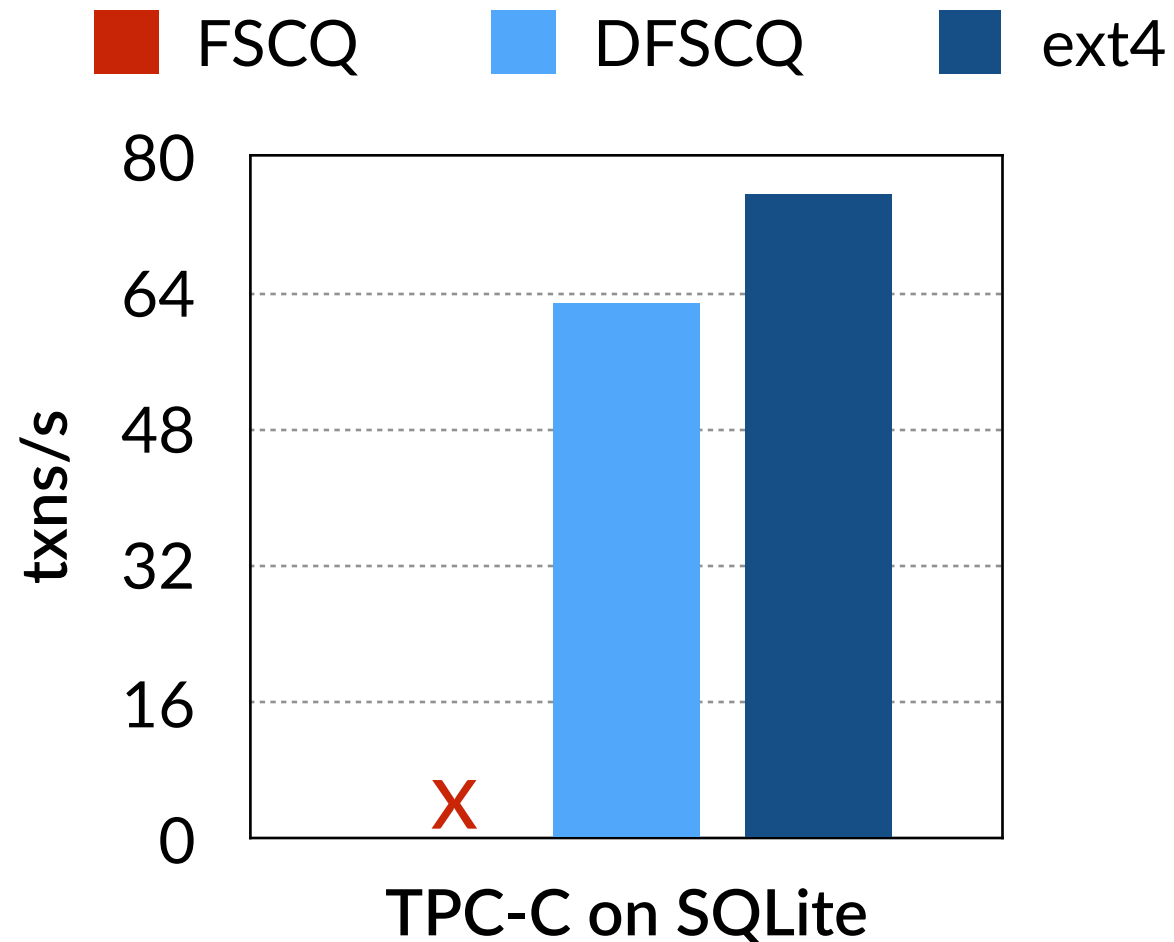


- mailbench simulates a gmail-like mail server
- metadata and fsync-heavy workload

SQLite on DFSCQ is competitive with ext4



SQLite on DFSCQ is competitive with ext4



- Write-heavy database workload
- DFSCQ issues less I/O, but has higher CPU overhead

Future work

- Reduce CPU overhead
- Concurrency

Summary

- **DFSCQ**: verified, efficient, crash-safe file system
 - **Precise tree-based specification** of deferred commit and log-bypass writes
 - Proof that implementation meets specification
 - Performance on par with Linux ext4



<https://github.com/mit-pdos/fscq>

Backup slides

- ext4 async commit + log-bypass bug
- verification architecture diagram
- write-ahead logging
- group commit
- log-bypass writes
- deferred commit and log-bypass perf
- spec example
- fsync(2)
- atomic write
- FUSE architecture
- LOC

Optimizations are hard to implement correctly

Subtle interaction between optimizations

- bug where crash could leak data in Linux ext4
- discovered after 6 years

ext4 now forbids both optimizations

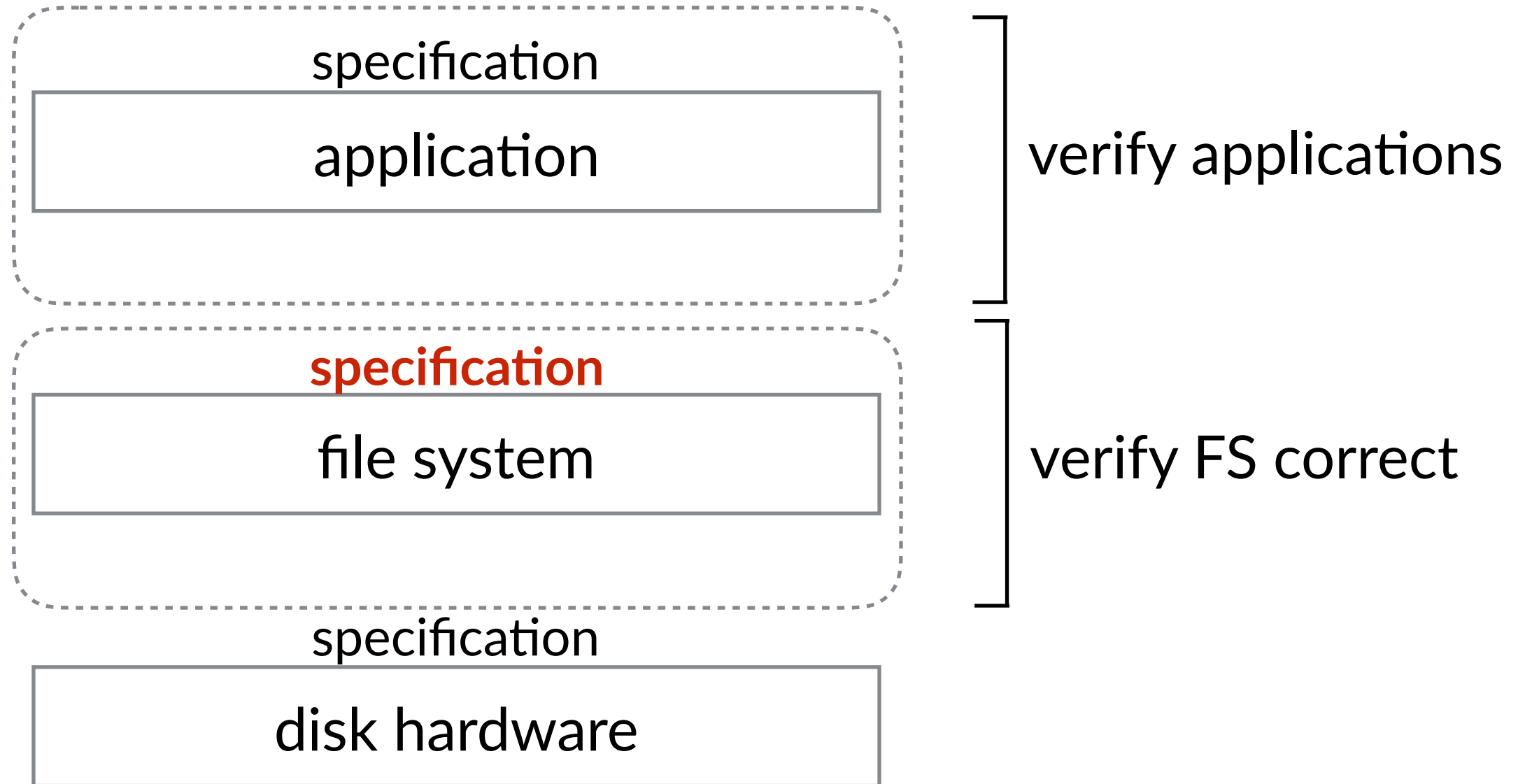
Author: Jan Kara <jack@suse.cz>

Date: Tue Nov 25 20:19:17 2014 -0500

```
ext4: forbid journal_async_commit in data=ordered mode
```

```
[...]
```

Approach to avoid bugs: verification



Write-ahead logging

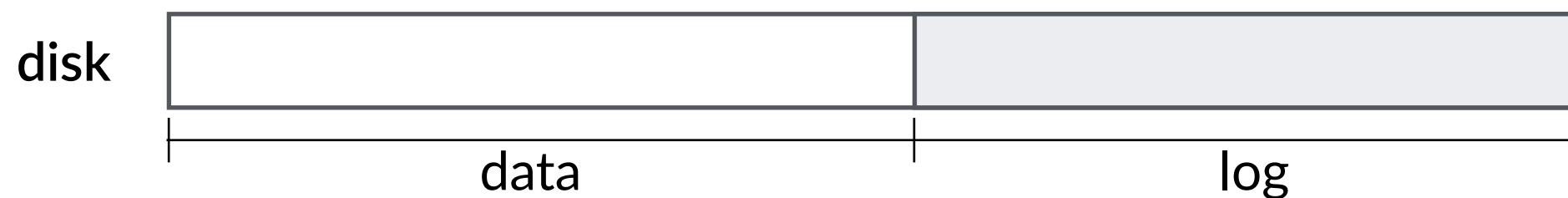


- System calls can update multiple disk blocks

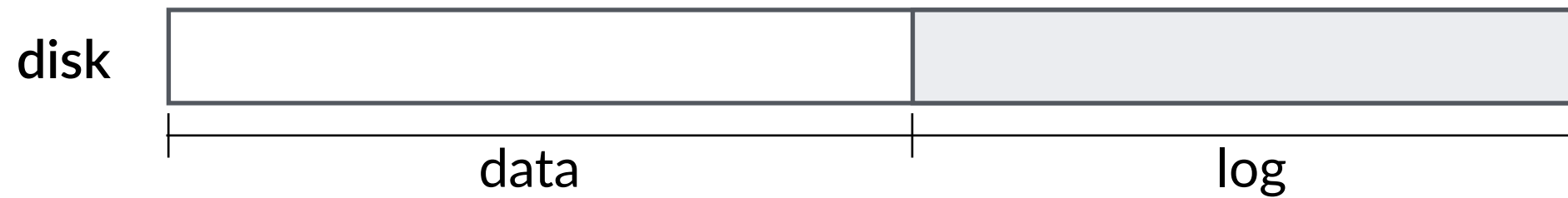
Write-ahead logging



- System calls can update multiple disk blocks
- Logging ensures all updates are persisted or none even if computer crashes

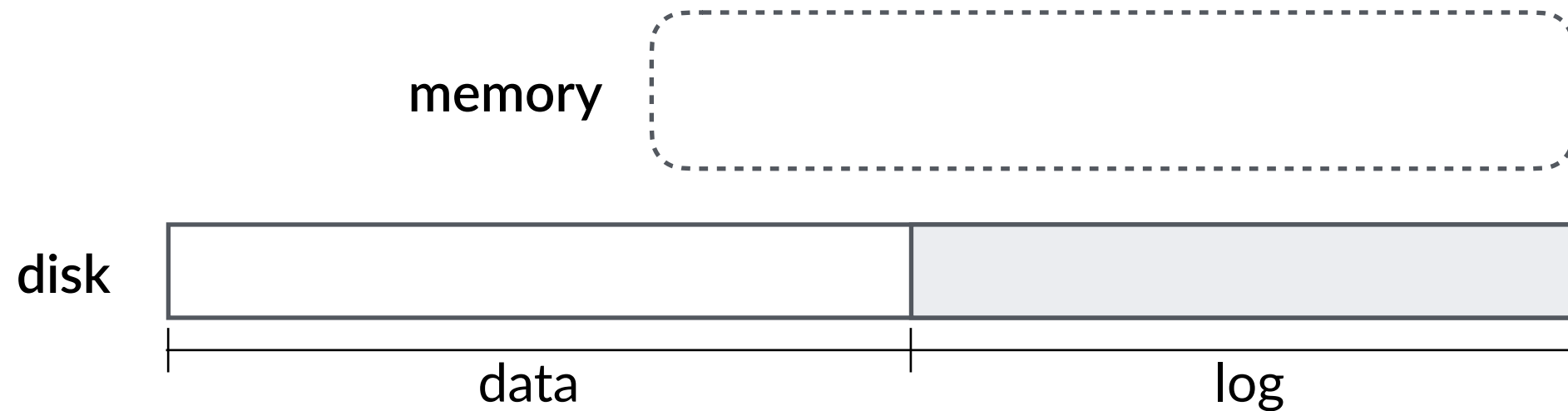


Deferred commit enables high throughput



Deferred commit enables high throughput

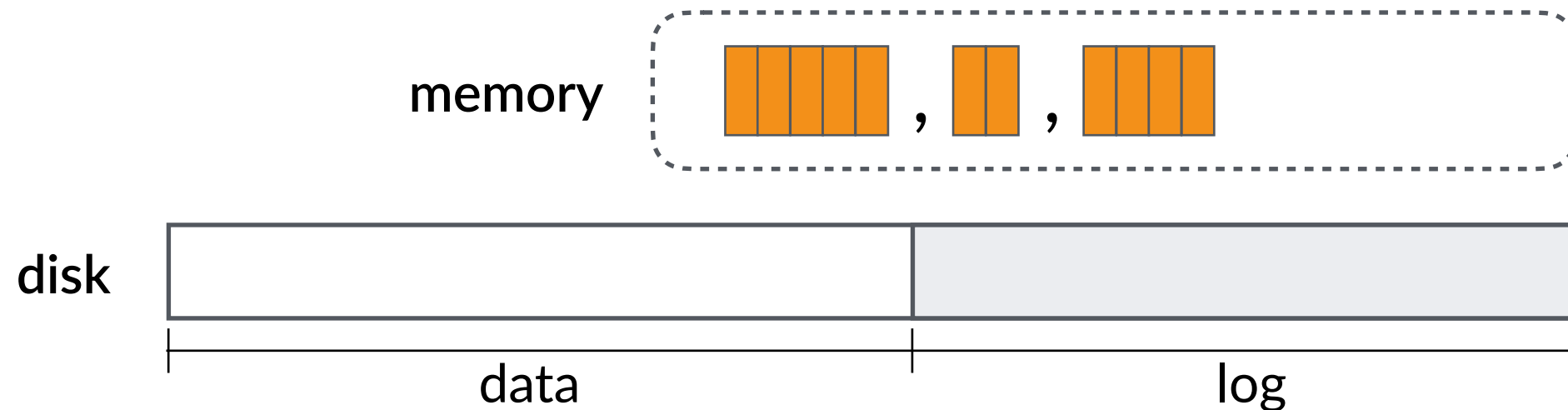
1. Buffer system calls in memory



Deferred commit enables high throughput

```
➔ mkdir('d')  
➔ create('d/a')  
➔ rename('d/a', 'd/b')
```

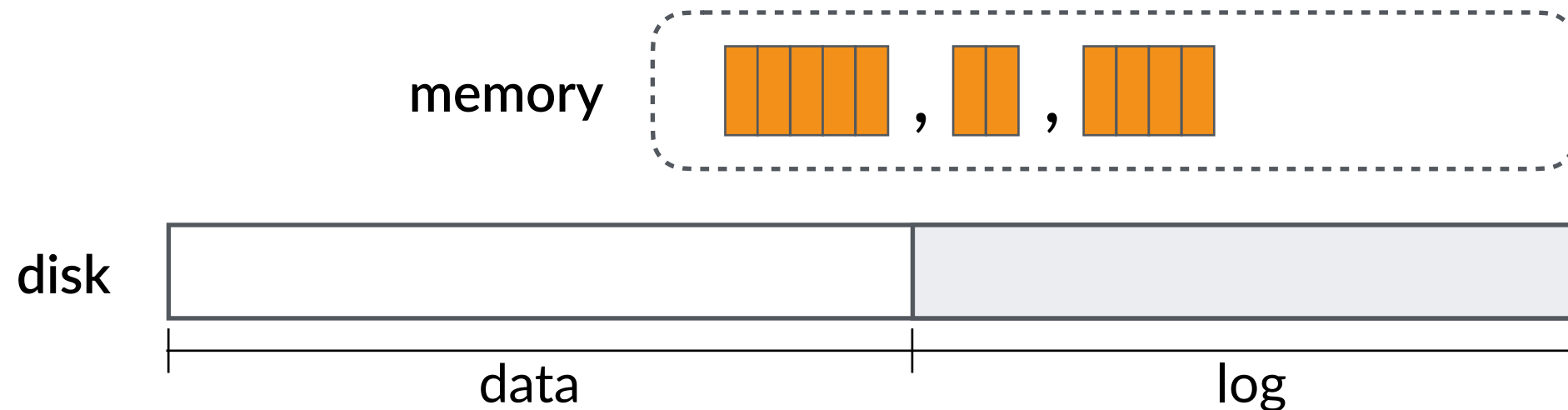
1. Buffer system calls in memory



Deferred commit enables high throughput

```
➔ mkdir('d')  
➔ create('d/a')  
➔ rename('d/a', 'd/b')  
➔ fsync('d')
```

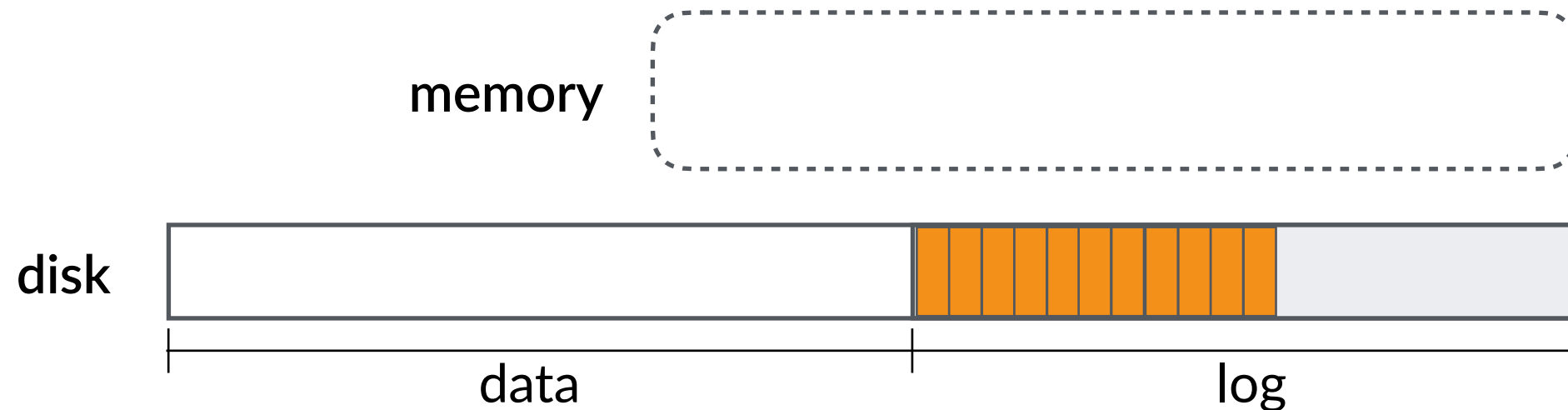
1. Buffer system calls in memory
2. fsync() flushes cached transactions to the on-disk log in a batch



Deferred commit enables high throughput

```
➔ mkdir('d')  
➔ create('d/a')  
➔ rename('d/a', 'd/b')  
➔ fsync('d')
```

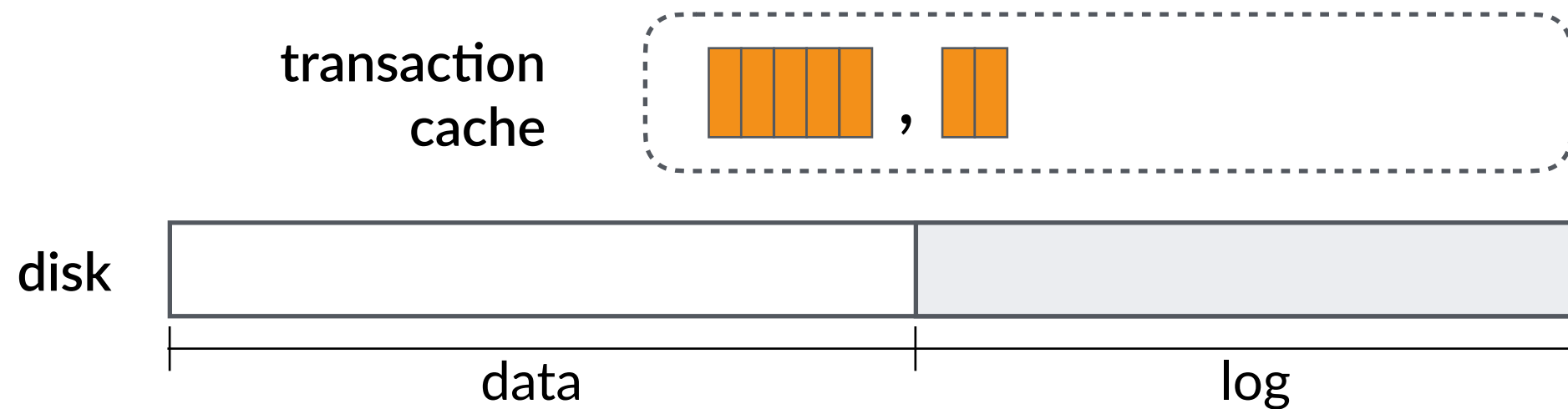
1. Buffer system calls in memory
2. fsync() flushes cached transactions to the on-disk log in a batch




Log-bypass writes avoid doubling data writes

```
→ mkdir('d')  
→ create('d/a')
```

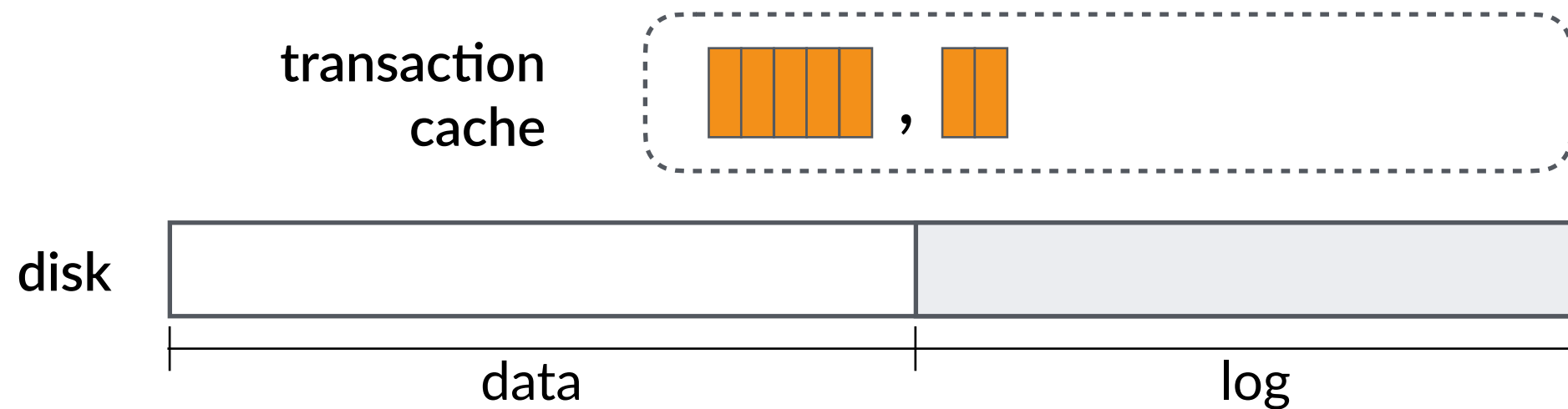
1. Record metadata updates in log as usual




Log-bypass writes avoid doubling data writes

```
→ mkdir('d')  
→ create('d/a')  
➔ write('d/a', )
```

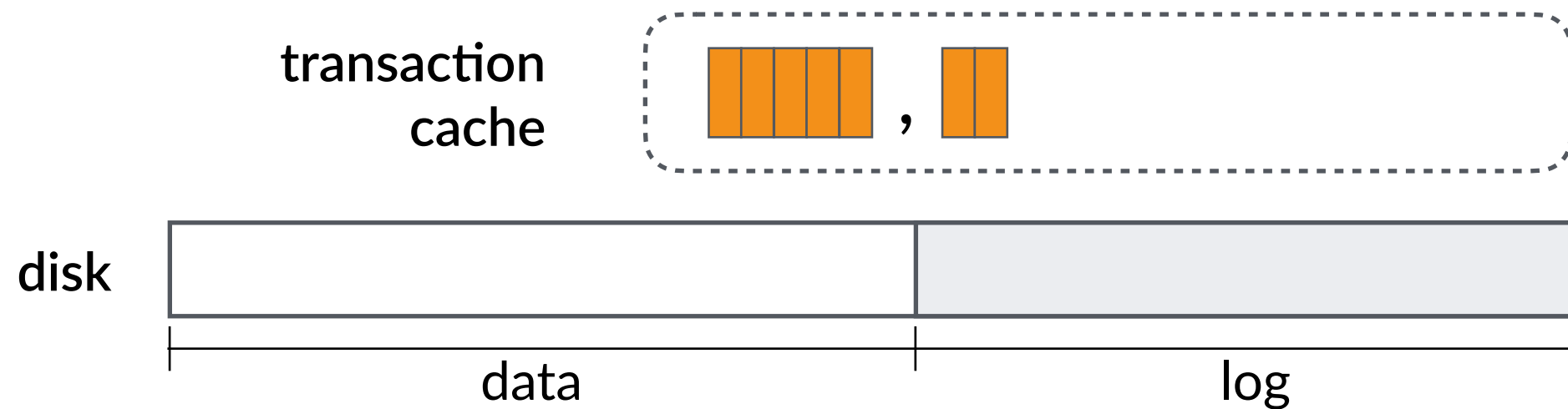
1. Record metadata updates in log as usual



Log-bypass writes avoid doubling data writes

```
→ mkdir('d')  
→ create('d/a')  
→ write('d/a', )
```

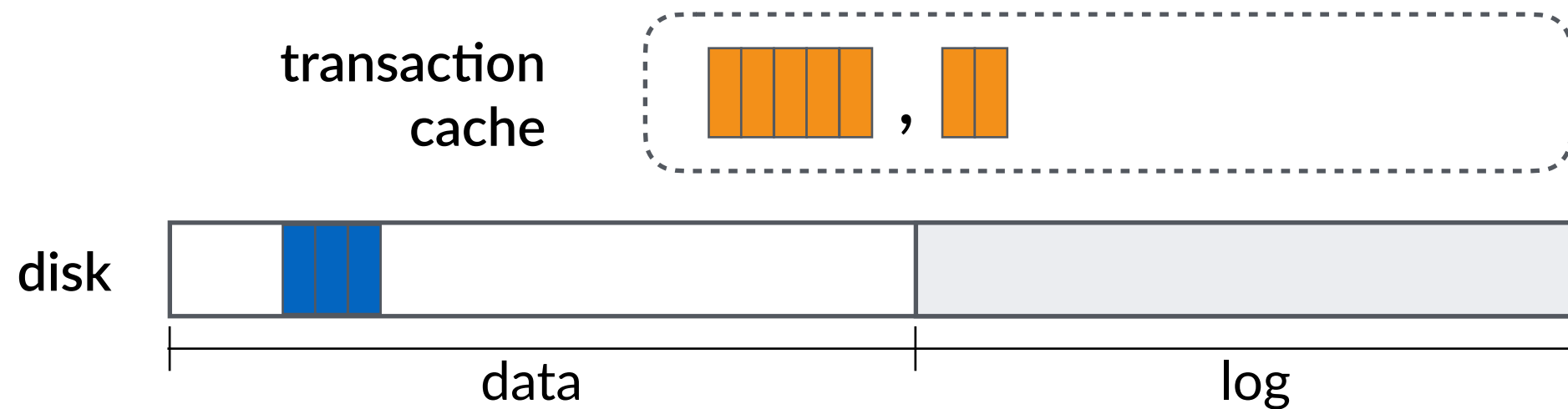
1. Record metadata updates in log as usual
2. **Bypass log** for file data



Log-bypass writes avoid doubling data writes

```
→ mkdir('d')  
→ create('d/a')  
➔ write('d/a',...)
```

1. Record metadata updates in log as usual
2. **Bypass log** for file data



Deferred commit and log bypass matter in practice

fdatasync every 10 MB
to an SSD

| ext4 performance | largefile |
|-------------------|-----------------|
| synchronous | 120 MB/s |
| + deferred commit | 150 MB/s |
| + log-bypass | 300 MB/s |

Specifications

```
SPEC  unlink(cwd_ino, pathname)
PRE    disk: tree_rep(tree_seq)
POST   disk: tree_rep(tree_seq ++ [new_tree]) /\
        new_tree = tree_prune(tree_seq.latest, cwd_ino, pathname)
CRASH  disk: tree_intact(tree_seq ++ [new_tree])
```

POSIX manual gives complicated specification

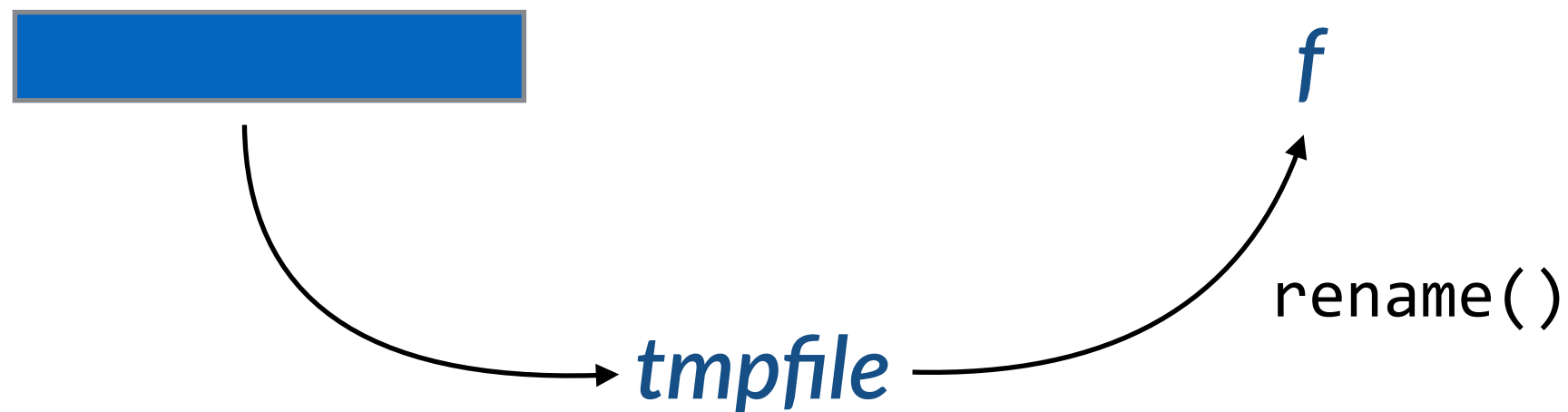
fsync() flushes modified buffer cache pages for fd to the disk device so that all changed information can be retrieved even after the system crashes or is rebooted. fsync() also flushes metadata information associated with the file (see inode(7)).

fdatasync() is similar to fsync(), but does not flush modified metadata unless that metadata is needed in order to allow a subsequent data retrieval to be correctly handled.

paraphrase of fsync(2) manpage

- not clear enough about crash behavior

Evaluating the specification: atomic write pattern



Goal: on crash *f* either:

- doesn't exist
- or contains 

Proved atomic write pattern crash safe

```
def atomic_write(data, name):  
    with open(tmpfile, "cw") as f:  
        ftruncate(f, len(data))  
        write(f, data)  
        fdatasync(f)  
    rename(tmpfile, name)  
    fsync(dirname(name))
```

prepare tmpfile
persist data
move to destination
persist metadata

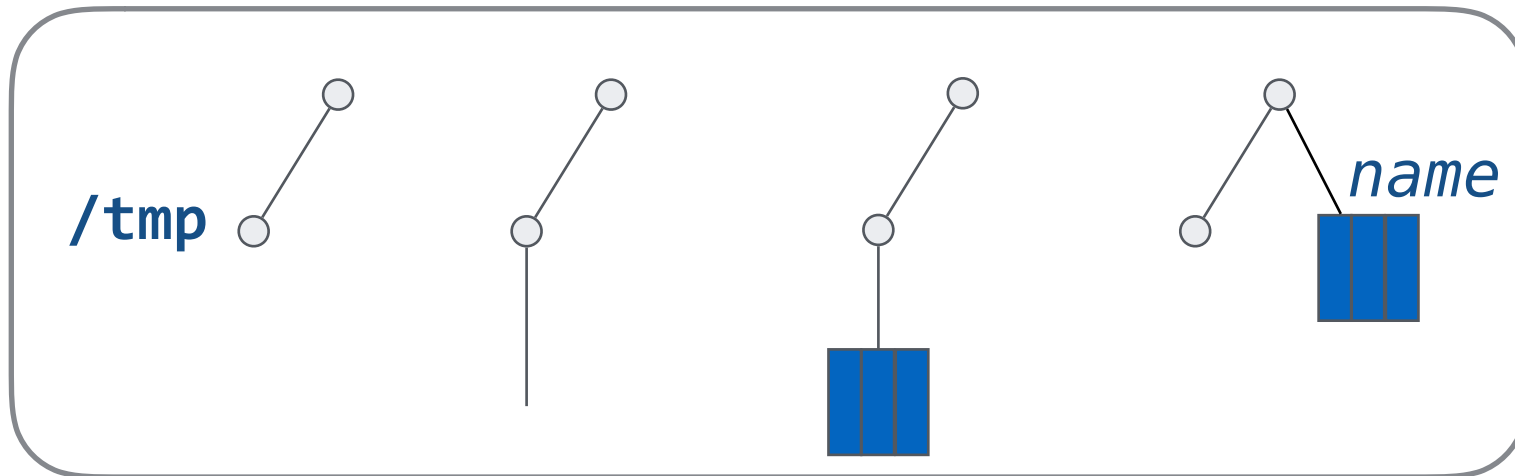
Proved atomic write pattern crash safe

```
def atomic_write(data, name):  
    with open(tmpfile, "cw") as f:  
        ftruncate(f, len(data))  
        write(f, data)  
        fdatasync(f)  
    rename(tmpfile, name)  
    fsync(dirname(name))
```

prepare tmpfile
persist data
move to destination
persist metadata

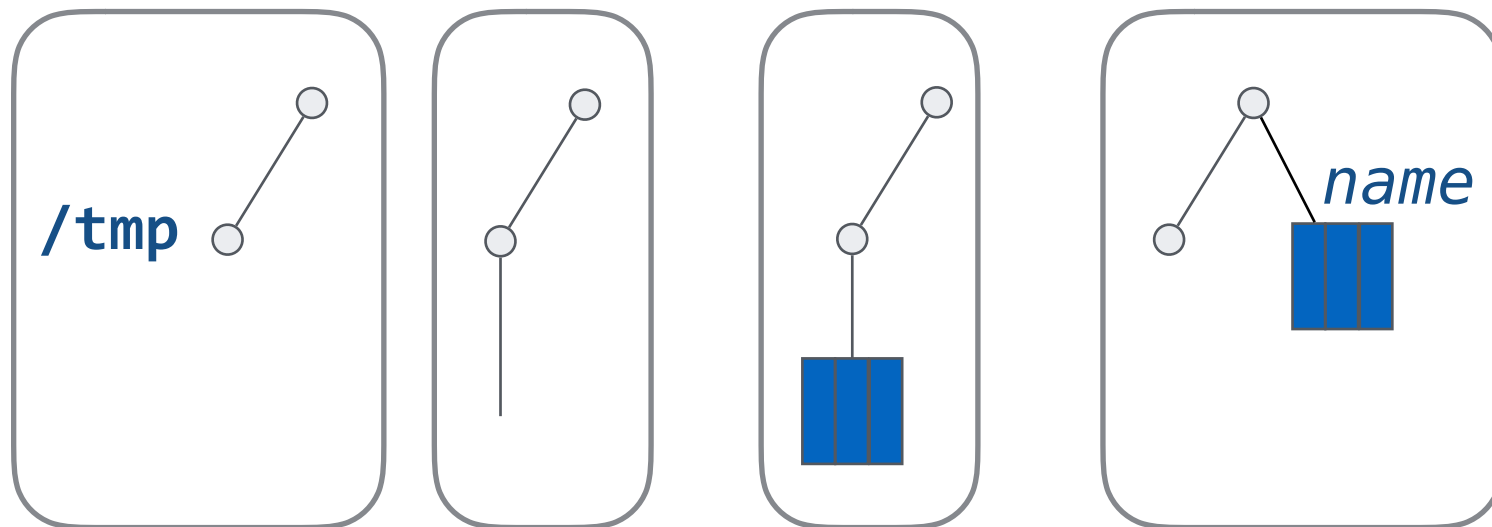
Specification is sufficient to prove
application-level properties

Atomic write is correct



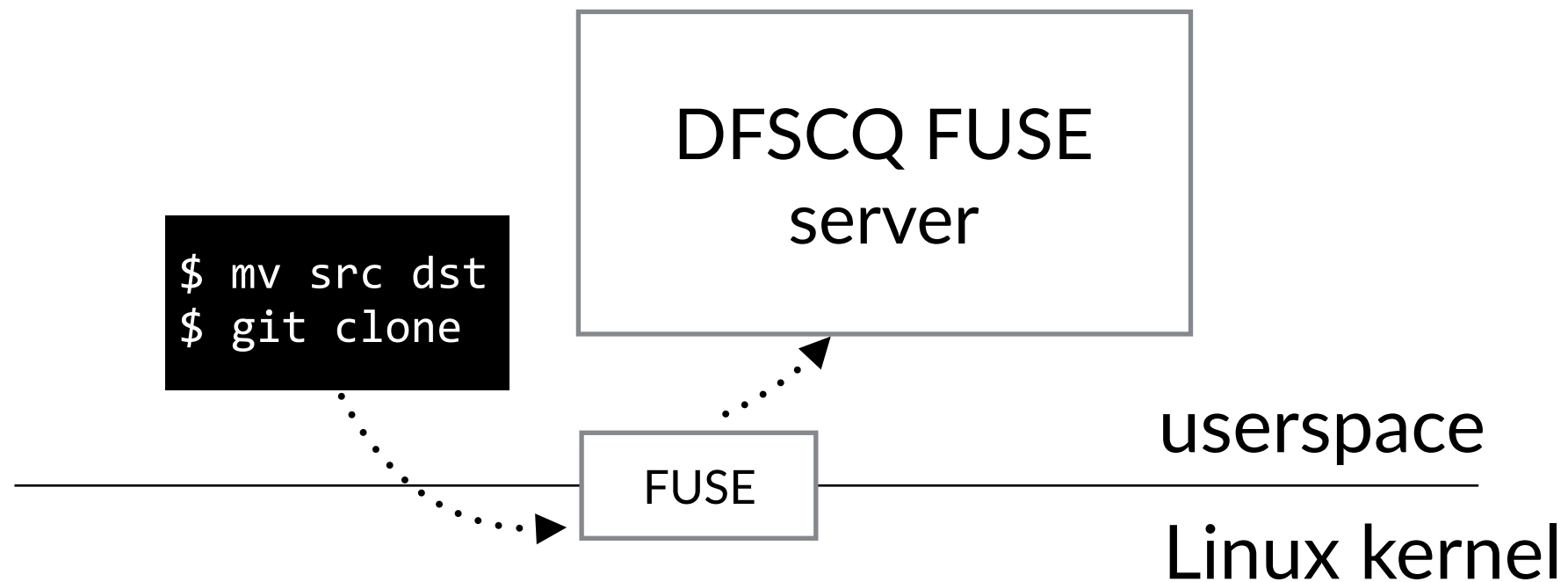
(just after rename)

crash states:



Specification: on crash, **name** either does not exist or contains **data**

DFSCQ runs ordinary Linux programs using FUSE



Effort to implement DFSCQ

- Total of 75,000 lines of **verified** code, specs, and proofs in Coq
 - Compared to FSCQ's 31,000 lines
 - 4,800 lines of implementation
- Took 5 authors 2 years (but less than 10 person years)

