GoJournal: a verified, concurrent, crash-safe journaling system

Tej Chajed
MIT CSAIL

Joseph Tassarotti
Boston College

Mark Theng
MIT CSAIL

Ralf Jung
MPI-SWS

Frans Kaashoek
MIT CSAIL

Nickolai Zeldovich
MIT CSAIL

Suppose we want to write a correct file system

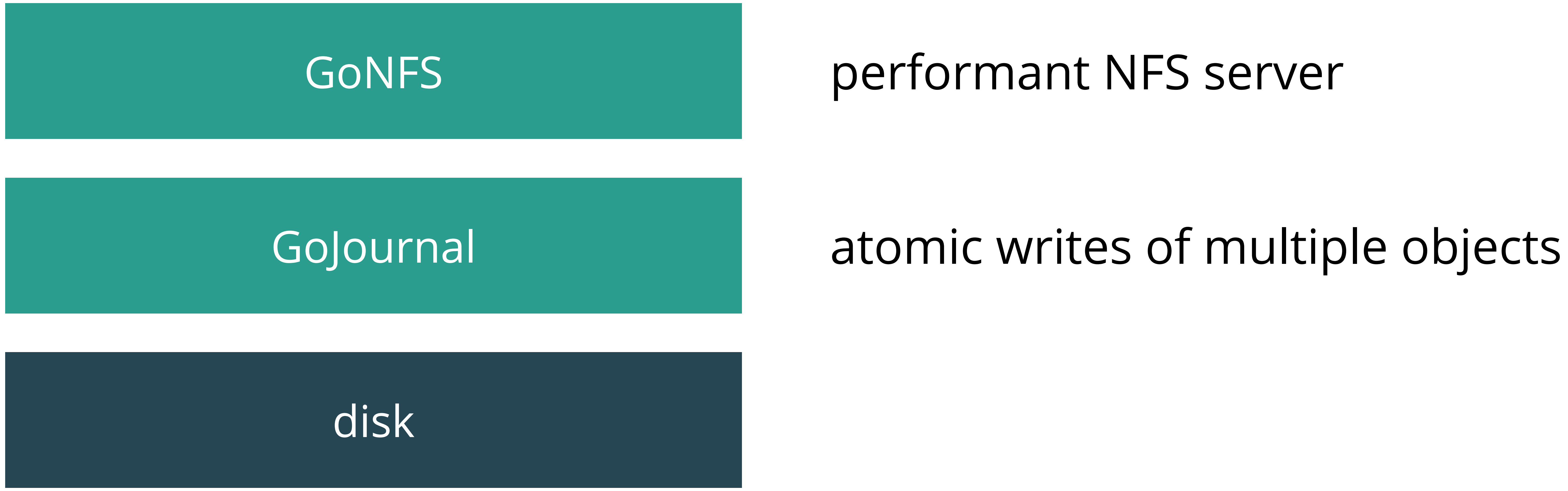
Correct: file-system operations atomically follow specification, even on crash

Performant: take advantage of concurrent operations to efficiently use CPU and I/O

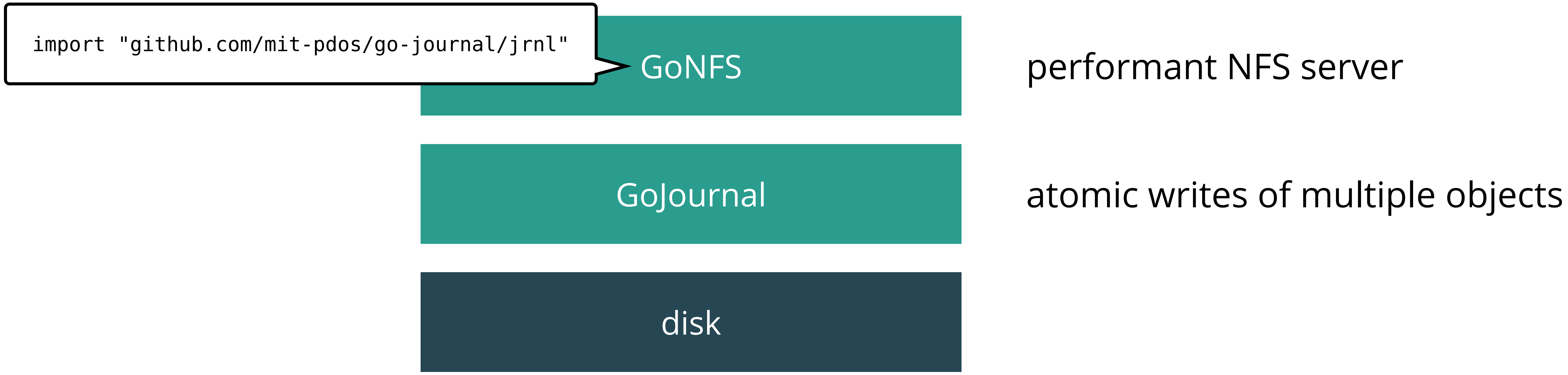
GoJournal gives a storage system efficient, atomic writes

atomic writes of multiple objects

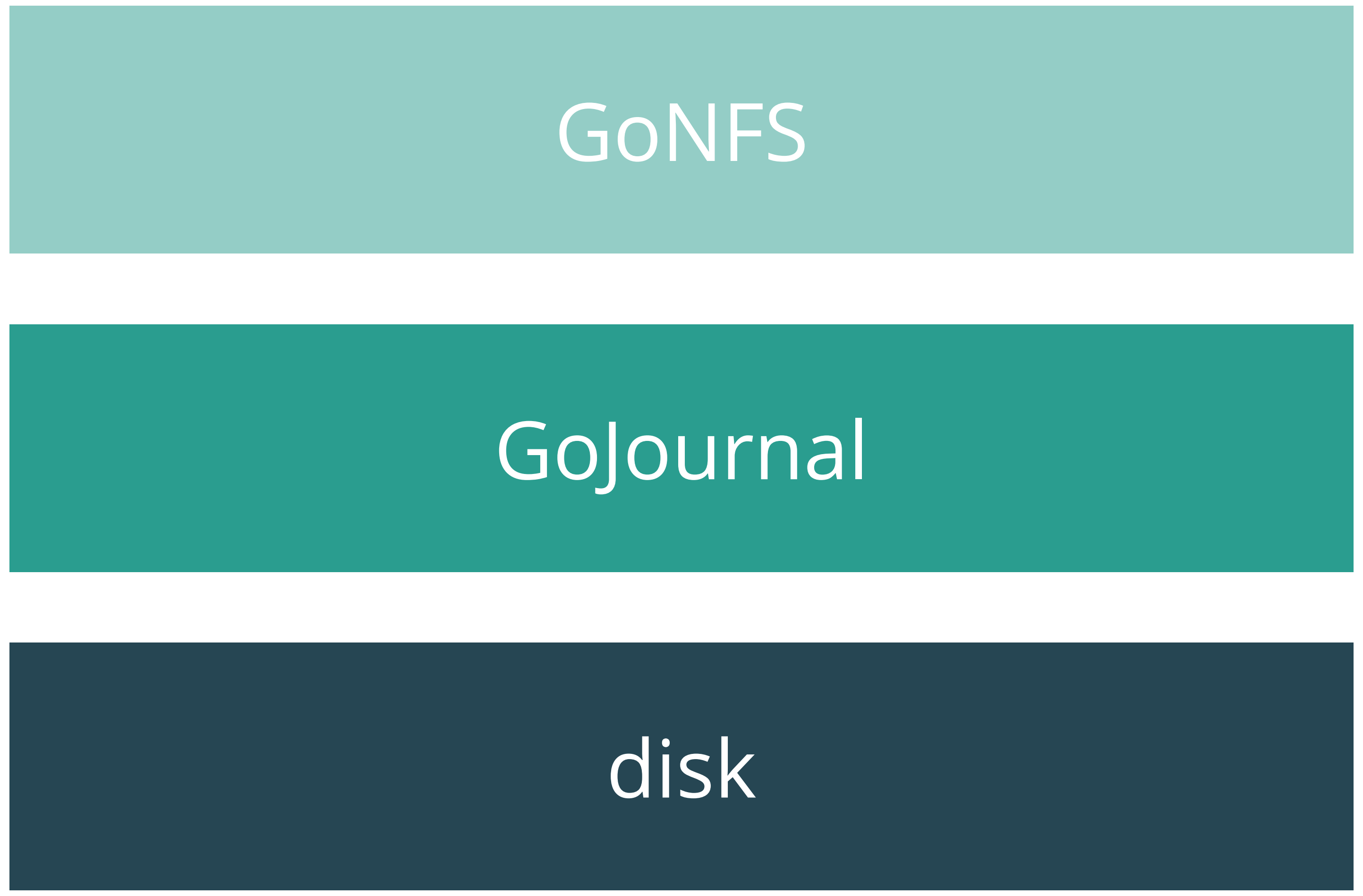
GoJournal gives a storage system efficient, atomic writes



GoJournal gives a storage system efficient, atomic writes



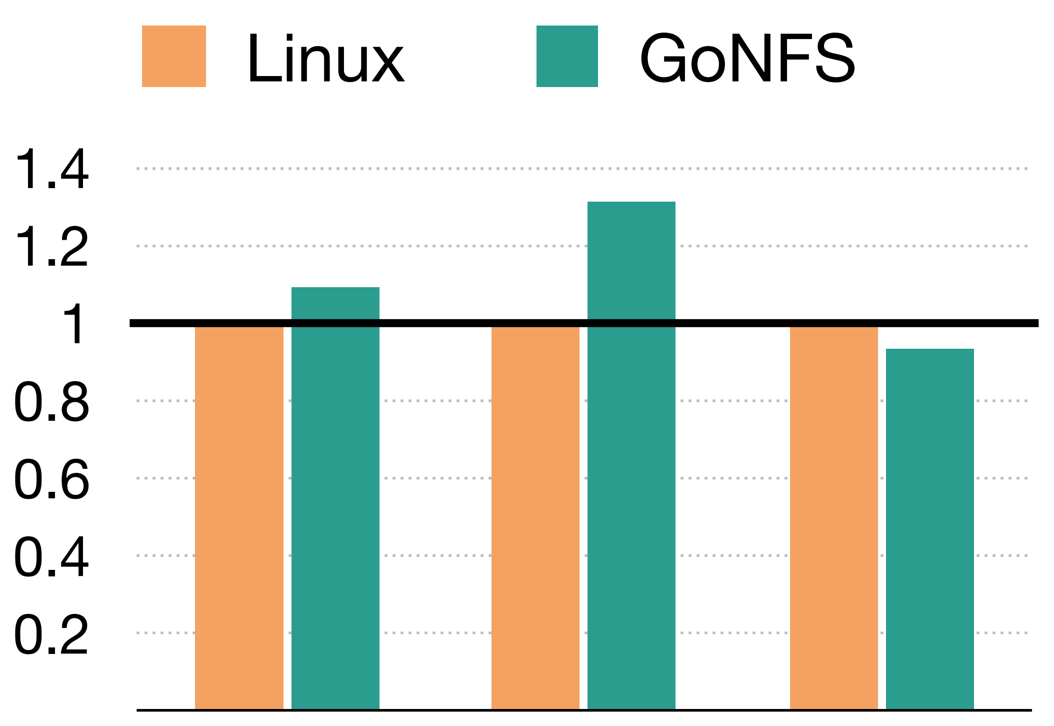
GoJournal is a verified journaling system



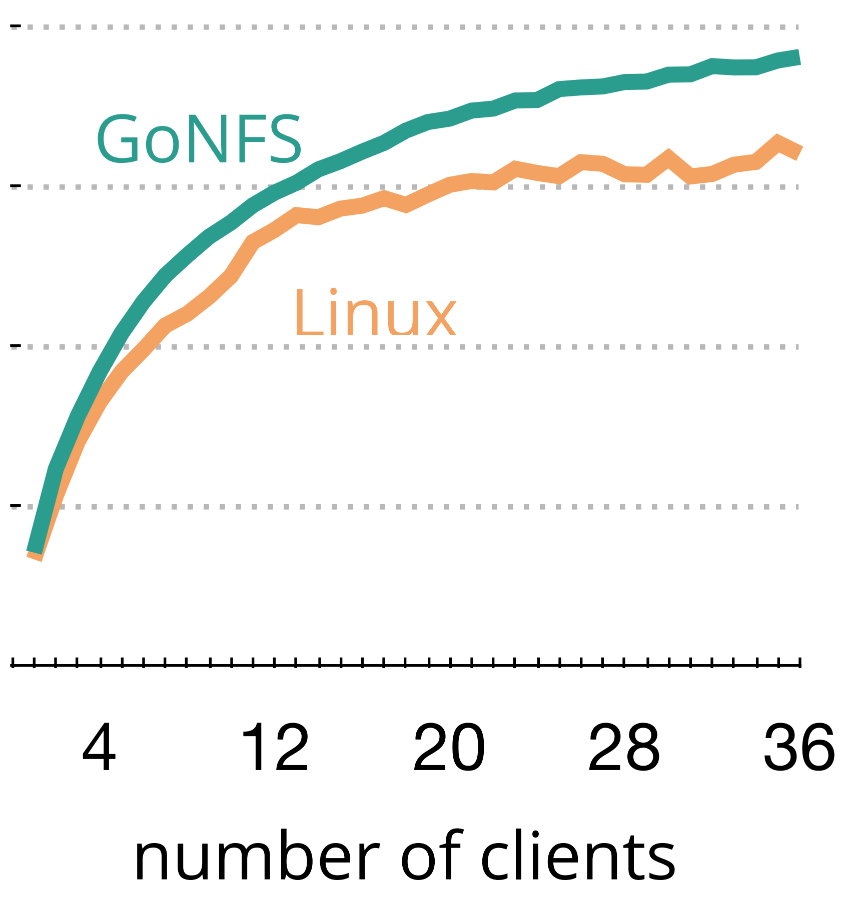
performant NFS server

atomic writes of multiple objects
comes with a machine-checked **proof**

GoJournal has a practical implementation



>95% throughput of Linux with a single client



Throughput scales with number of concurrent clients

Current approaches cannot handle a system of this complexity

Crash-safe but sequential file systems

FSCQ, Yggdrasil, VeriBetrFS

Concurrent systems

CertiKOS, AtomFS, ...

Current approaches cannot handle a system of this complexity

Crash-safe but sequential file systems
FSCQ, Yggdrasil, VeriBetrFS

Concurrent systems
CertiKOS, AtomFS, ...

Crash safety and concurrency
Perennial 1.0

Contributions

GoJournal, the first verified concurrent journal

Perennial 2.0, a new verification framework

SimpleNFS to evaluate specification

Evaluation showing GoJournal achieves good performance

Contributions

GoJournal, the first verified concurrent journal

Perennial 2.0, a new verification framework

SimpleNFS to evaluate specification

Evaluation showing GoJournal achieves good performance

GoJournal writes operations atomically to disk

```
// one-time init
var d Disk
jrnl := OpenJrnl(d)
```

GoJournal writes operations atomically to disk

```
// one-time init
var d Disk
jrnl := OpenJrnl(d)

// copy block at 0 to 1 and 2
op := jrnl.Begin()
buf := op.ReadBuf(0, blockSz)
op.OverWrite(1, buf.Data)
op.OverWrite(2, buf.Data)
op.Commit()
```

GoJournal writes operations atomically to disk

```
// one-time init
var d Disk
jrnl := OpenJrnl(d)

// copy block at 0 to 1 and 2
op := jrnl.Begin()
buf := op.ReadBuf(0, blockSz)
op.OverWrite(1, buf.Data)
op.OverWrite(2, buf.Data)
op.Commit()
```

concurrent operations are atomic

caller is responsible for locking

Operations can concurrently manipulate objects within a block

File system has 128-byte inodes

Sub-block access improves concurrency since caller only locks the required objects

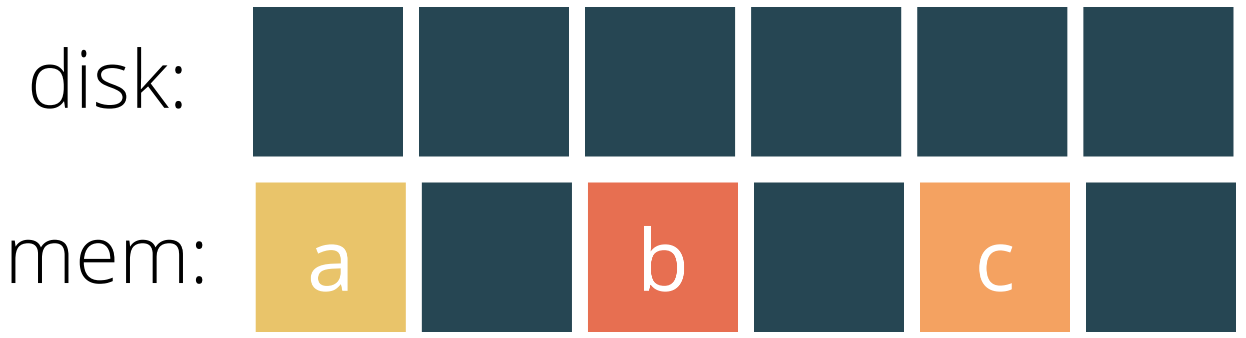
Specification challenge: what do concurrently committed operations do?

```
op := jrnl.Begin()
buf := op.ReadBuf(0, blockSz)
op.OverWrite(1, buf.Data)
op.OverWrite(2, buf.Data)
op.Commit()
```

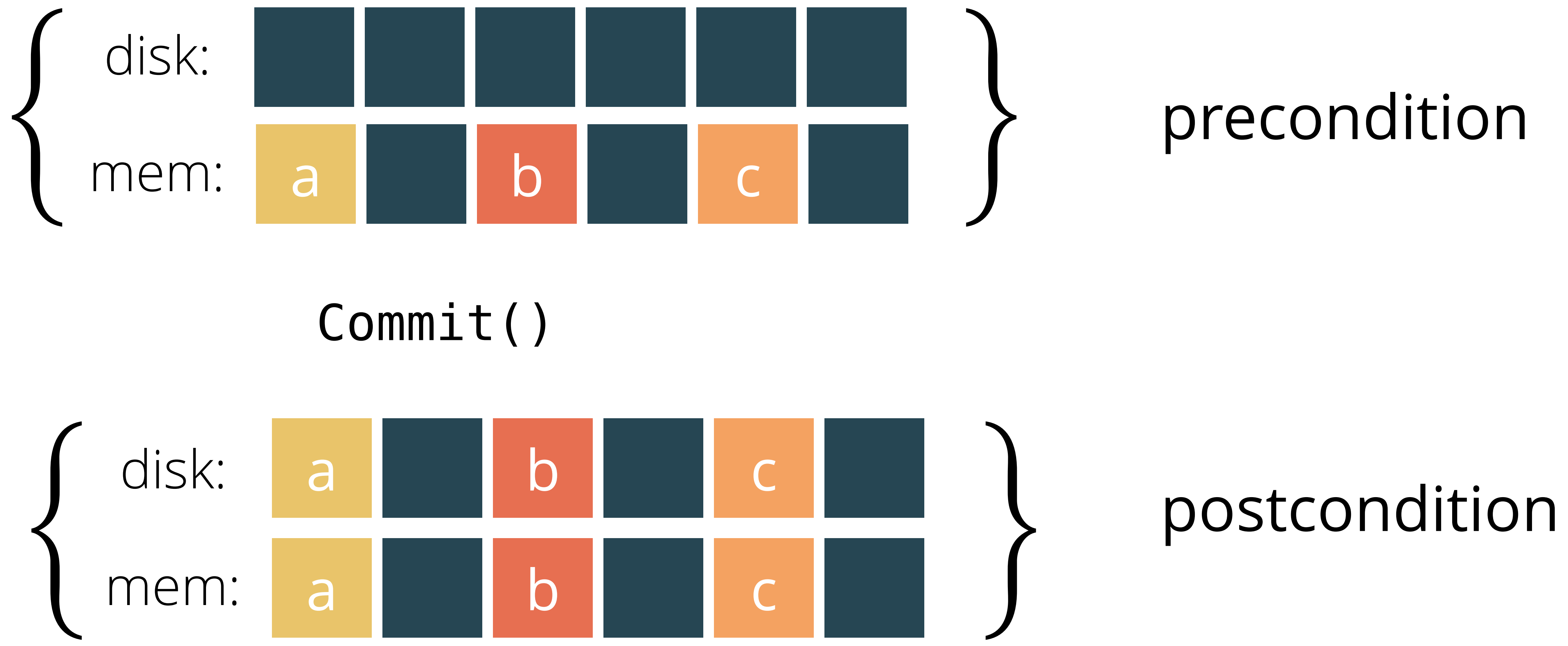
||

```
op := jrnl.Begin()
op.OverWrite(7, data)
op.Commit()
```

Sequential journaling only maintains old and next state



Sequential journaling only maintains old and next state



An operation's specification only refers to its disk footprint

says block at `0` has value b_0

`0` $\mapsto b_0$

disk

An operation's specification only refers to its disk footprint

$0 \mapsto b_0$ disk	$1 \mapsto -$ disk	$2 \mapsto -$ disk

An operation's specification only refers to its disk footprint

$$\left\{\ \boxed{0 \mapsto b_0 \ \ \text{disk}}\quad \boxed{1 \mapsto - \ \ \text{disk}}\quad \boxed{2 \mapsto - \ \ \text{disk}}\ \right\}$$

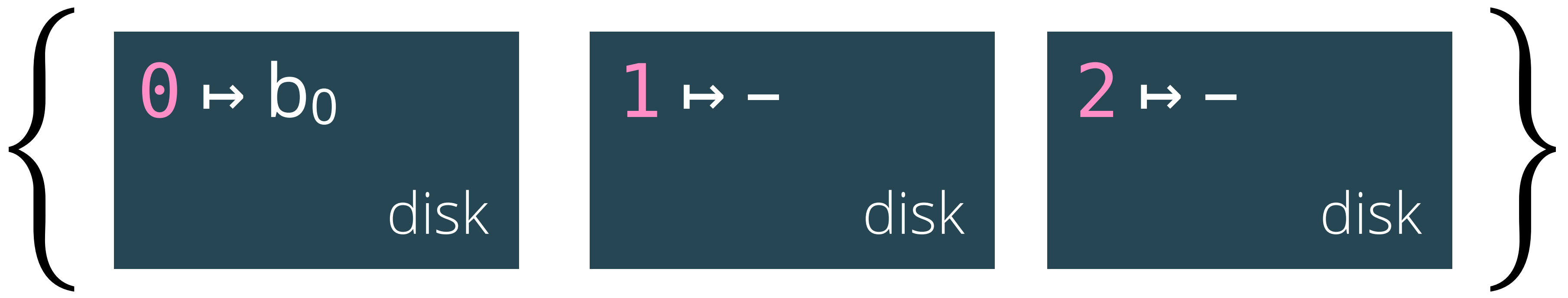
```
op := jrnl.Begin()

buf := op.ReadBuf(0, blockSz)
op.OverWrite(1, buf.Data)
op.OverWrite(2, buf.Data)

op.Commit()
```

$$\left\{\ \boxed{0 \mapsto b_0 \ \ \text{disk}}\quad \boxed{1 \mapsto b_0 \ \ \text{disk}}\quad \boxed{2 \mapsto b_0 \ \ \text{disk}}\ \right\}$$

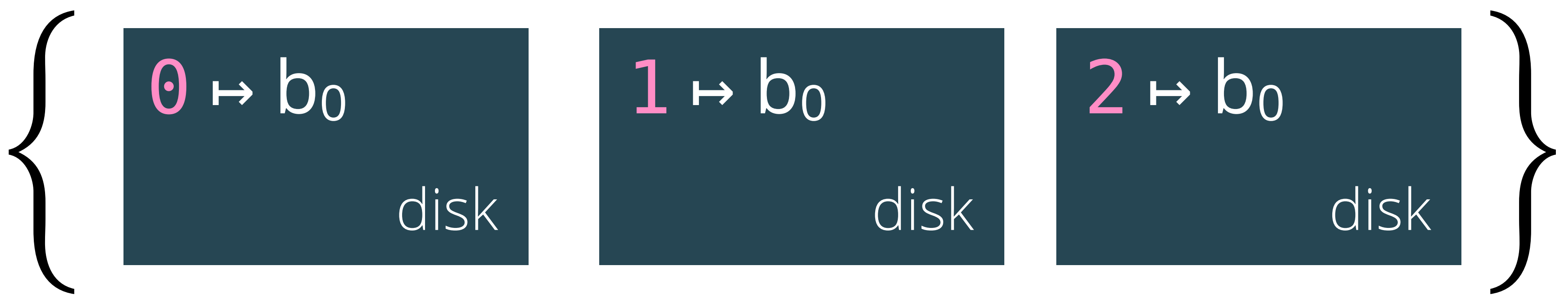
An operation's specification only refers to its disk footprint




```
op := jrnl.Begin()

buf := op.ReadBuf(0, blockSz)
op.OverWrite(1, buf.Data)
op.OverWrite(2, buf.Data)

op.Commit()
```



implies any other data is not involved

Introduce assertion for operation's view of disk

"disk points-to"

$1 \mapsto -$

disk

"operation points-to"

$1 \mapsto -$

op

Introduce assertion for operation’s view of disk

“disk points-to”

$1 \mapsto -$

disk

on-disk value of block 1

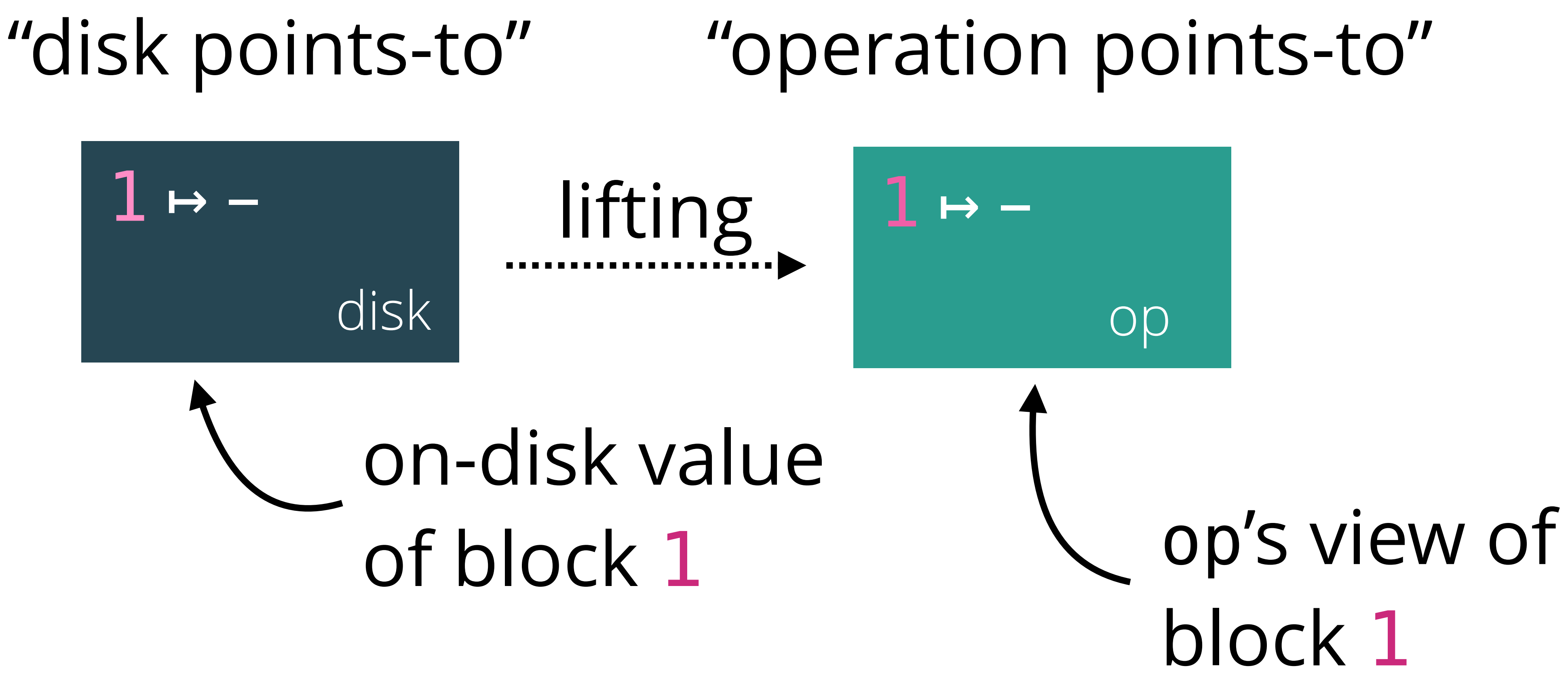
“operation points-to”

$1 \mapsto -$

op

op’s view of block 1

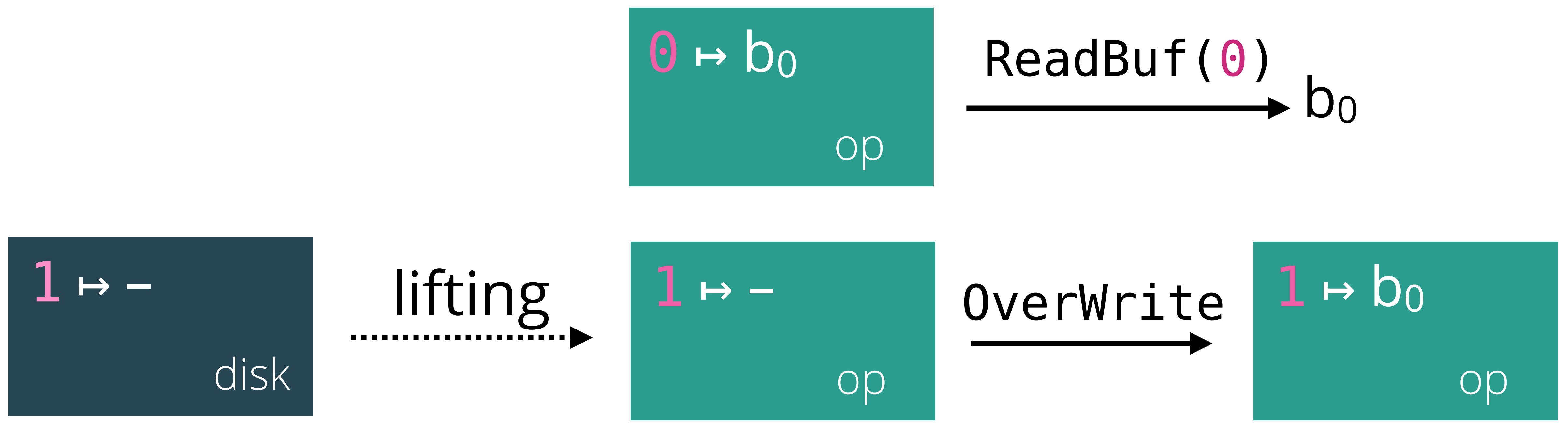
Introduce assertion for operation's view of disk



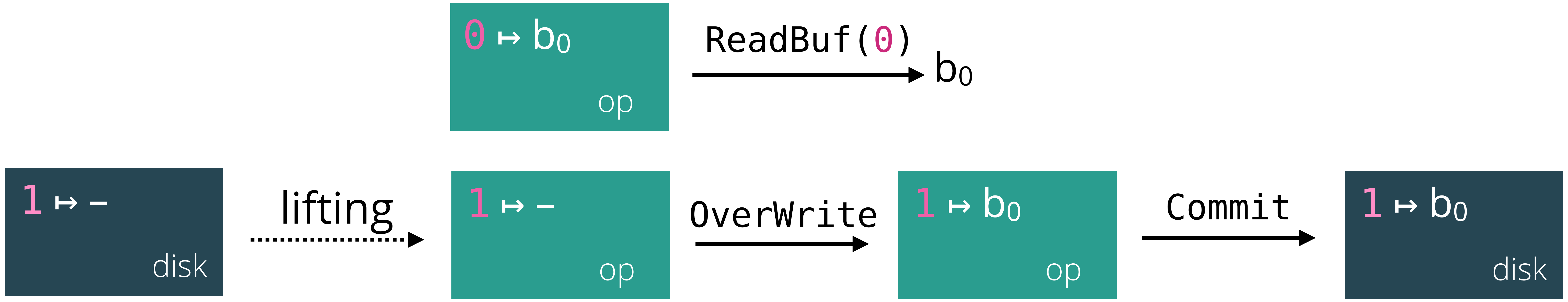
Key idea: operations manipulate an in-memory view of each object

Key idea: operations manipulate an in-memory view of each object

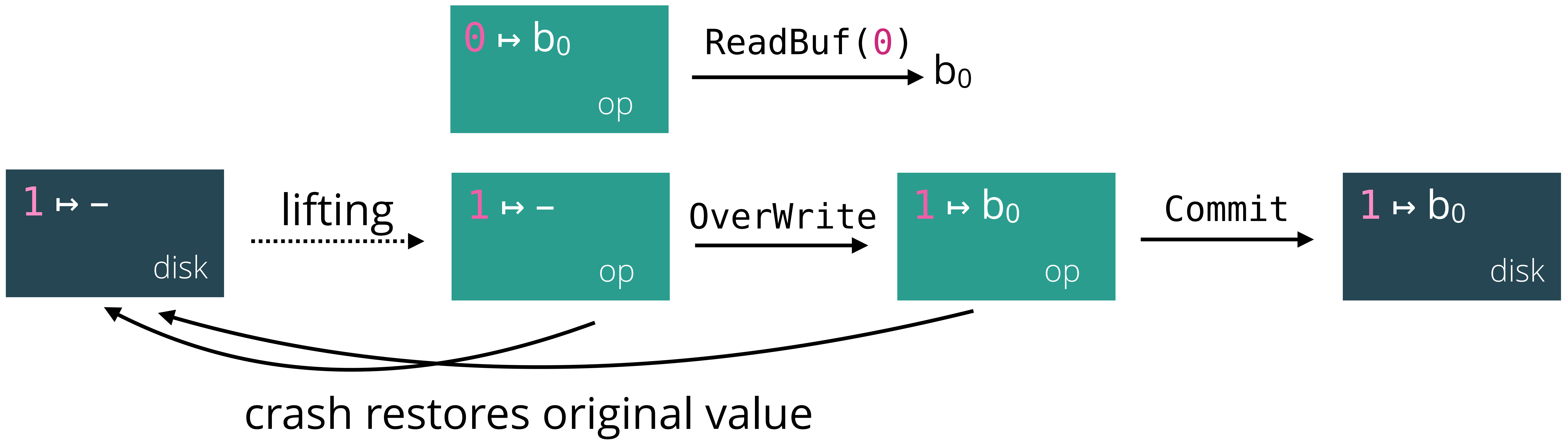
Key idea: operations manipulate an in-memory view of each object



Key idea: operations manipulate an in-memory view of each object



Key idea: operations manipulate an in-memory view of each object



Key idea: operations manipulate an in-memory view of each object

0 $\mapsto b_0$ disk	1 $\mapsto$ – disk	2 $\mapsto$ – disk

```
op := jrnl.Begin()
```

```
buf := op.ReadBuf(0, blockSz)
op.OverWrite(1, buf.Data)
op.OverWrite(2, buf.Data)
```

```
op.Commit()
```

0 $\mapsto b_0$ disk	1 $\mapsto b_0$ disk	2 $\mapsto b_0$ disk

Key idea: operations manipulate an in-memory view of each object

$0 \mapsto b_0$ disk	$1 \mapsto -$ disk	$2 \mapsto -$ disk

```
op := jrnl.Begin()
```

lifting

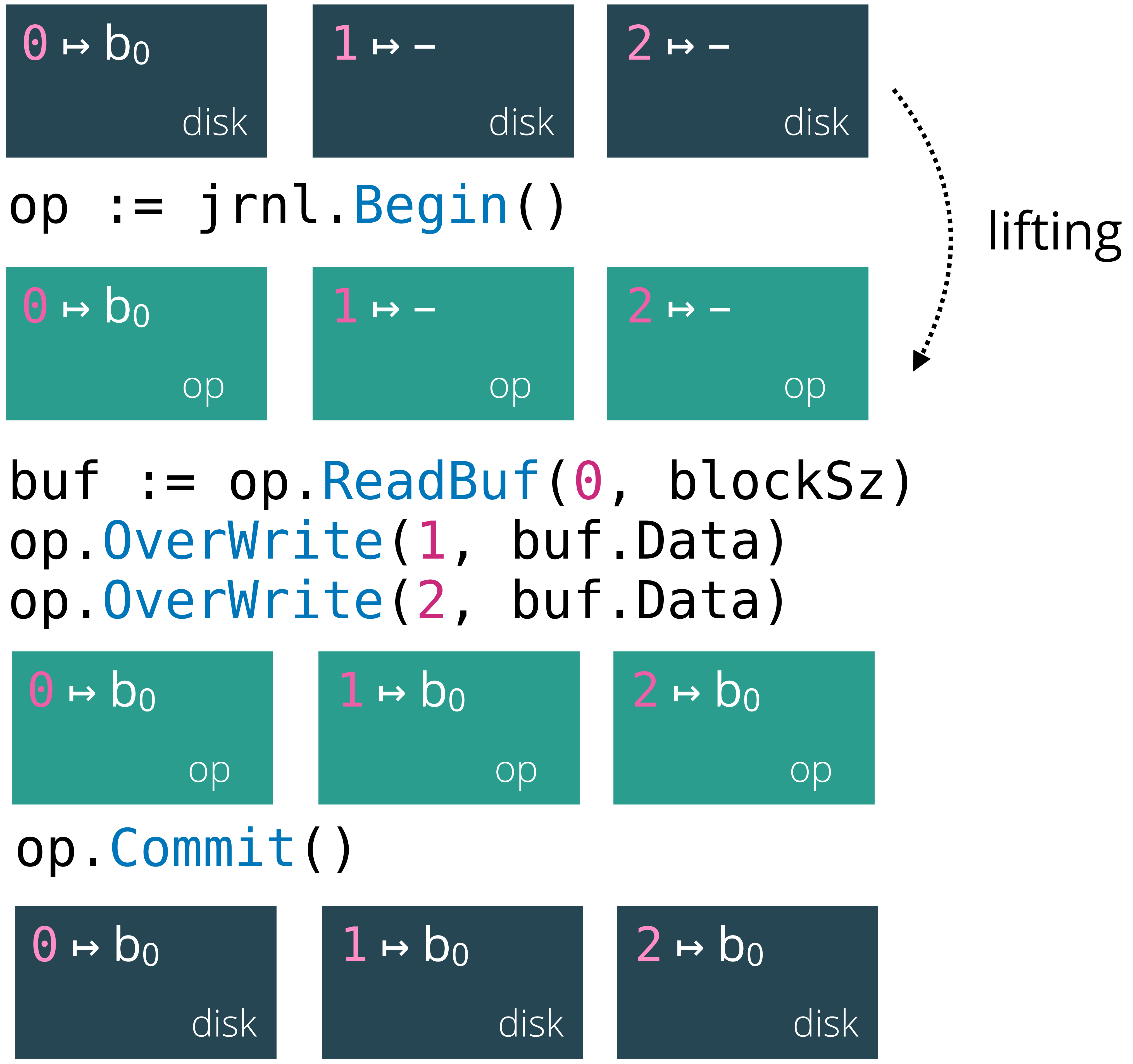
$0 \mapsto b_0$ op	$1 \mapsto -$ op	$2 \mapsto -$ op

```
buf := op.ReadBuf(0, blockSz)
op.OverWrite(1, buf.Data)
op.OverWrite(2, buf.Data)
```

```
op.Commit()
```

$0 \mapsto b_0$ disk	$1 \mapsto b_0$ disk	$2 \mapsto b_0$ disk

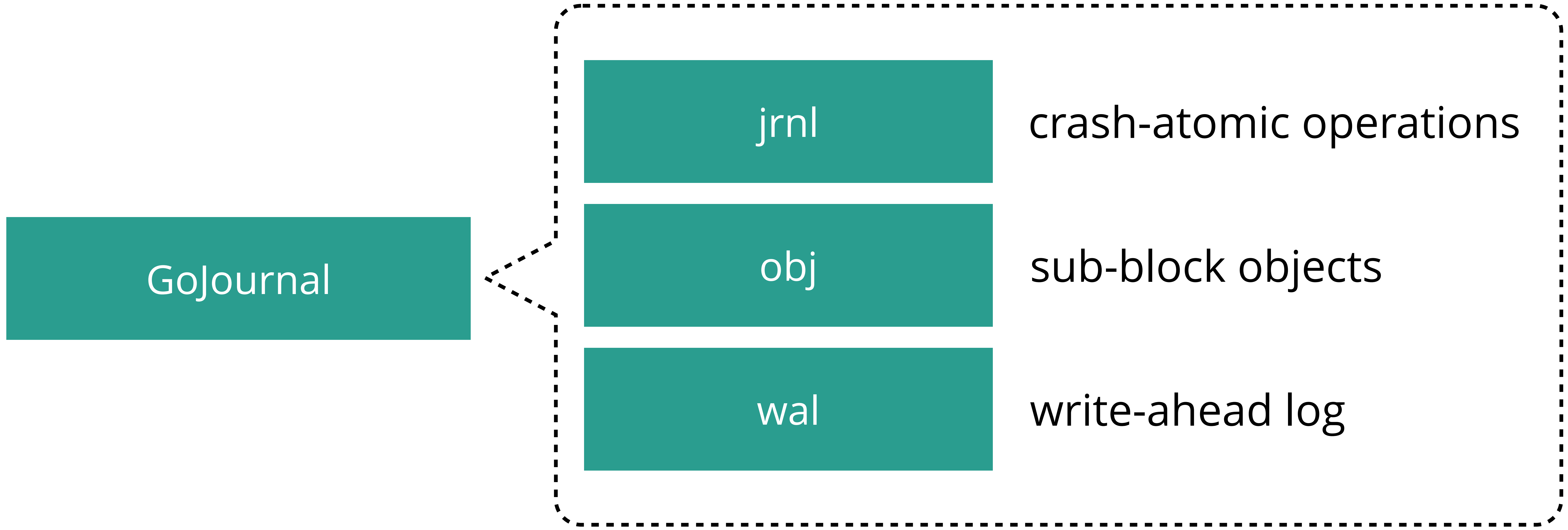
Key idea: operations manipulate an in-memory view of each object



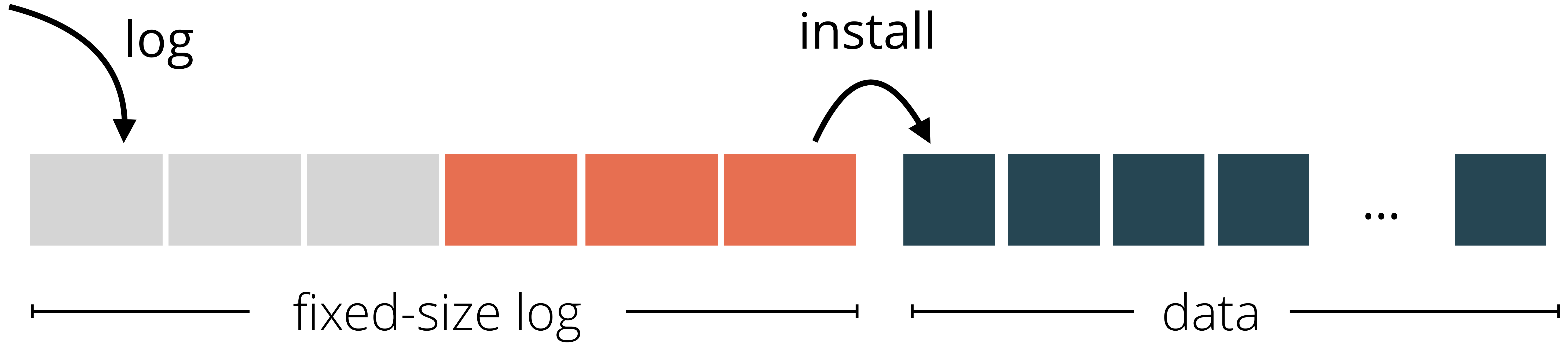
GoJournal has a modular implementation and proof

GoJournal

GoJournal has a modular implementation and proof



Write-ahead log implements the core atomicity of the journal



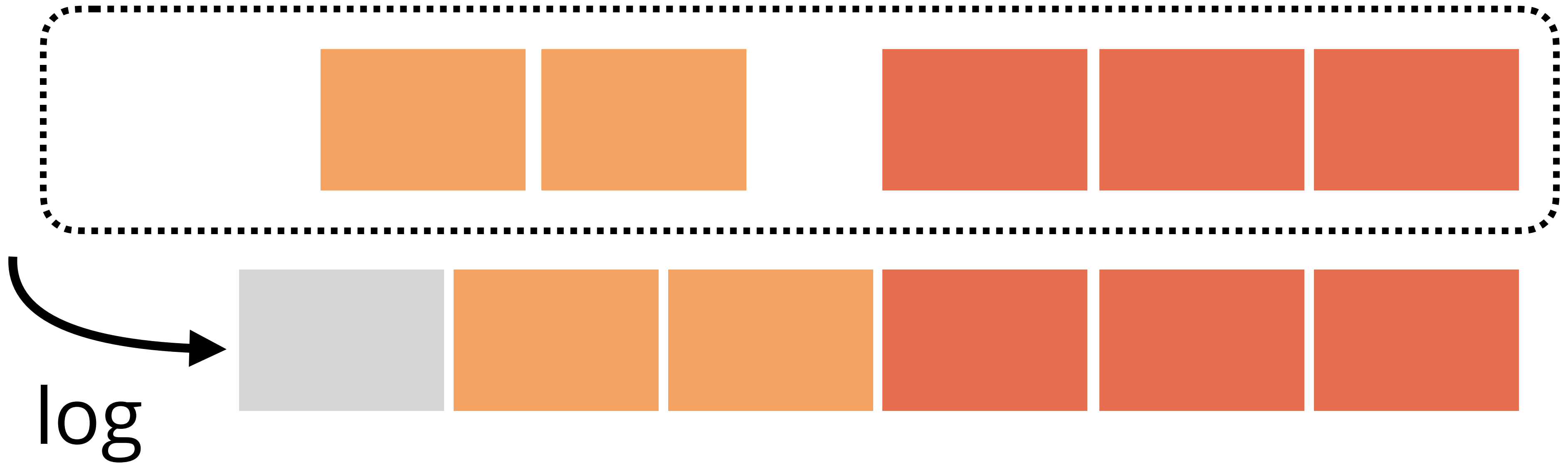
Writes are buffered before being logged

1 write gets buffered

Writes are buffered before being logged

1. write gets buffered

2. write gets logged



Challenge 1:
Reads can observe unstable writes

1 write gets buffered

2 read returns new data

system crashes here

Challenge 1:
Reads can observe unstable writes

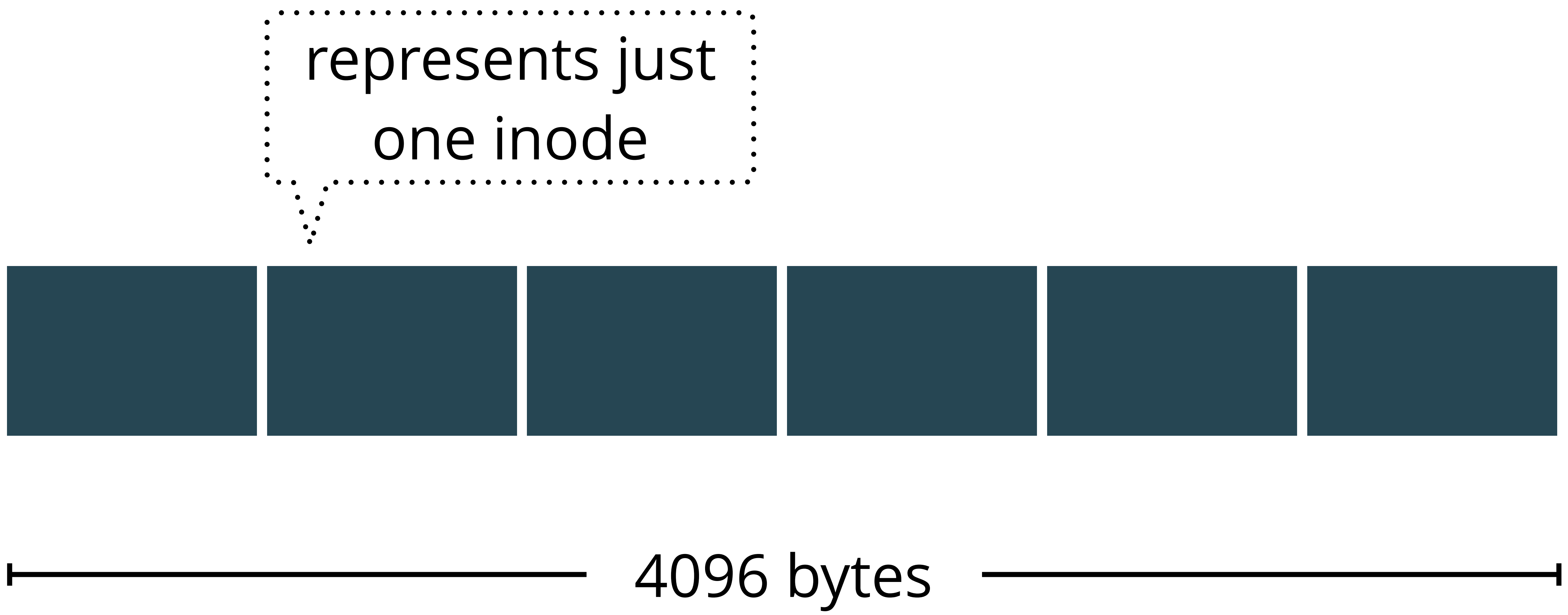
1 write gets buffered

2 read returns new data

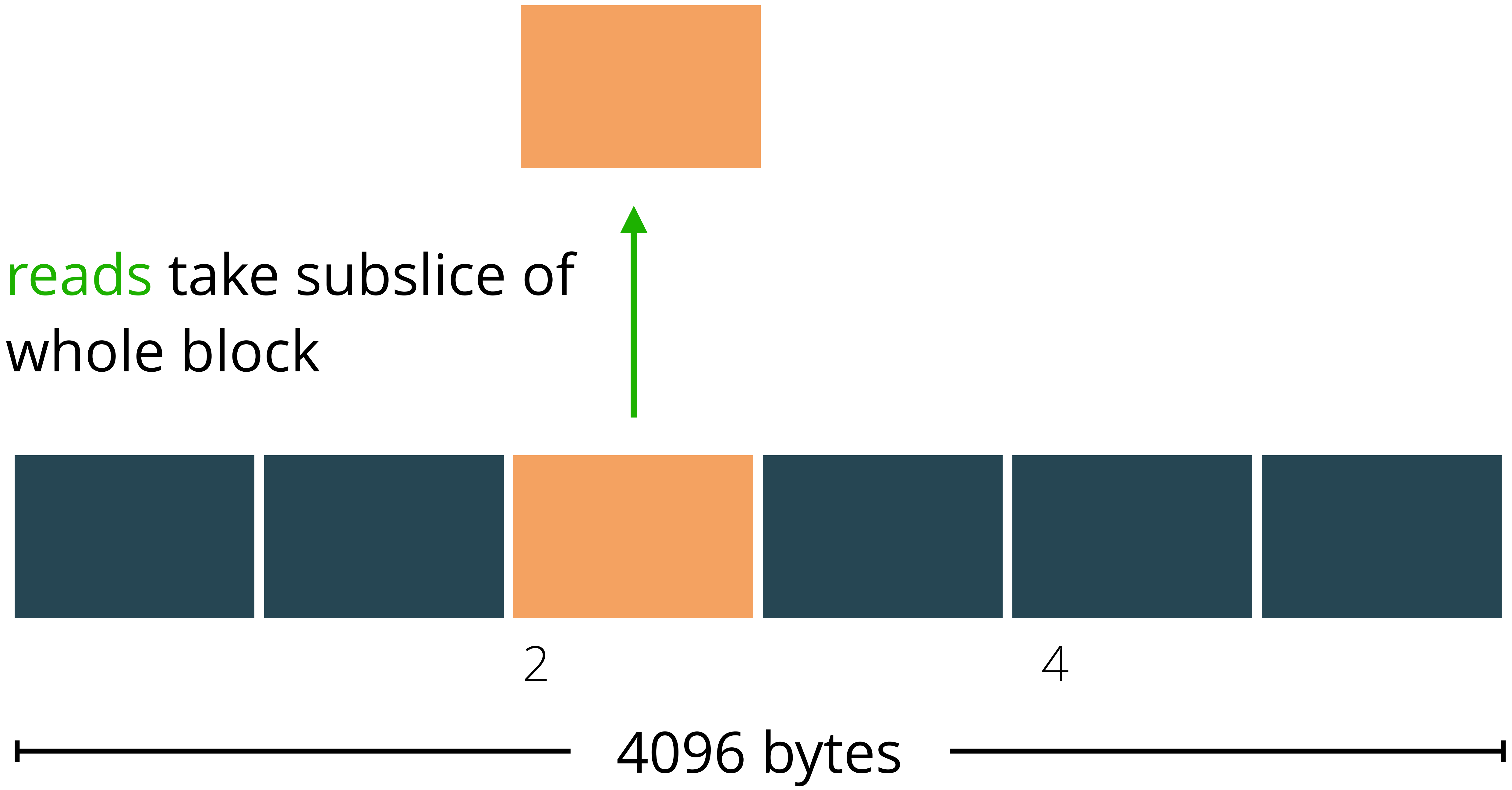
system crashes here

3 **read returns old data**

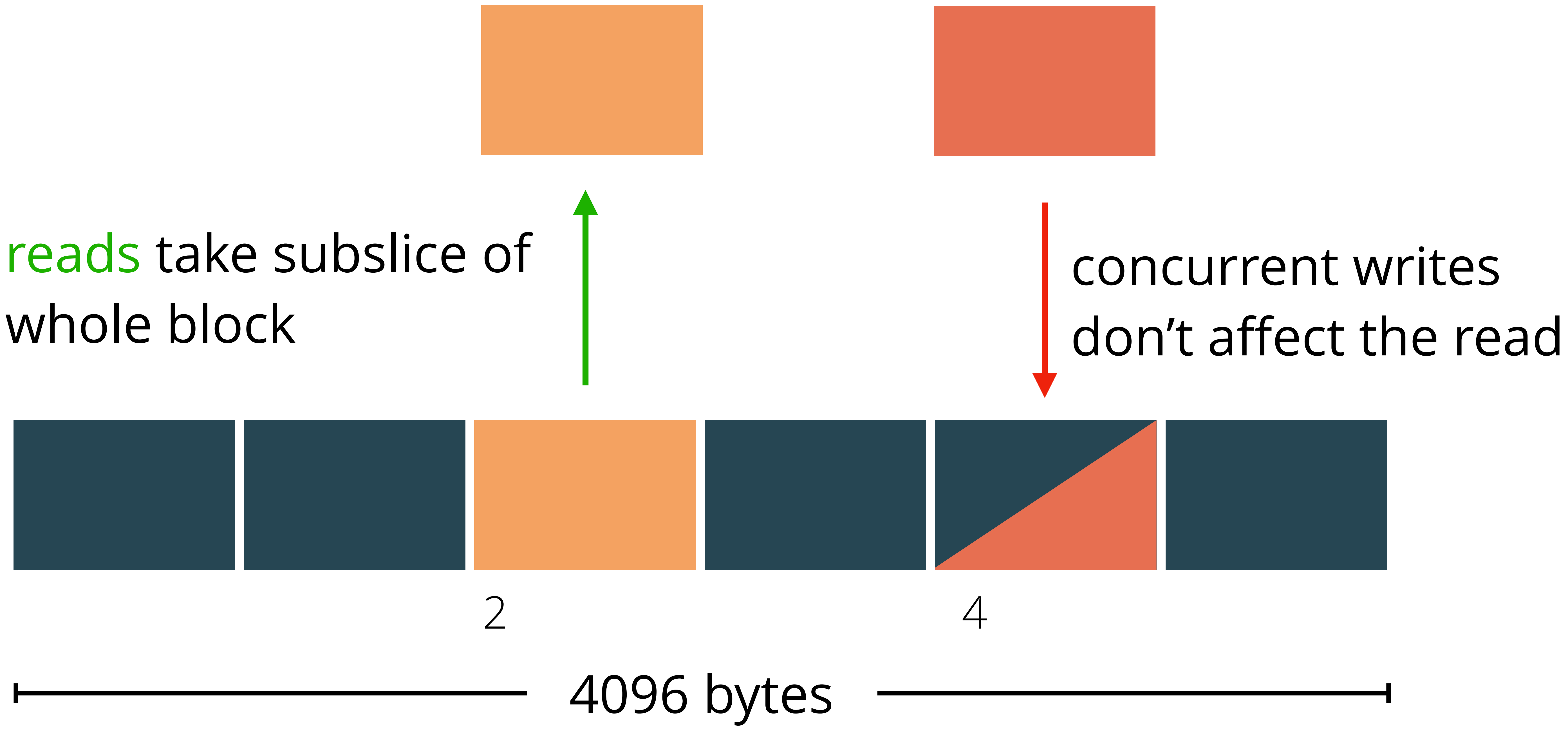
Object layer implements sub-block object access



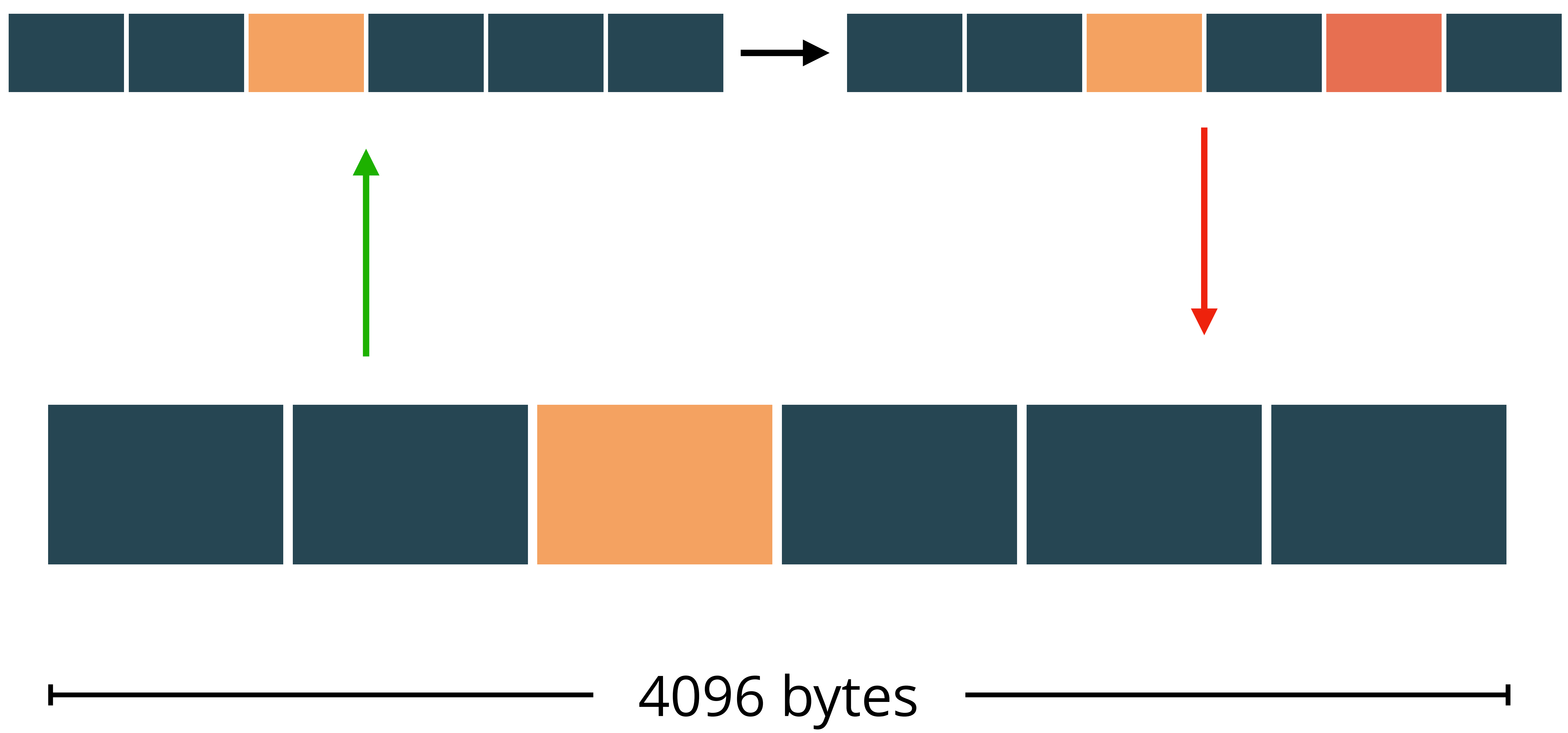
Challenge 2:
Reads and writes can proceed concurrently



Challenge 2:
Reads and writes can proceed concurrently



Concurrent writes are unsafe due to read-modify-write sequence



Verification techniques in Perennial 2.0

see paper for details

Logically atomic crash specifications

Lock-free reasoning with monotonic counters

Lifting to specify Commit

Crash-aware lock specification

Implementation overview

1,300 lines

GoJournal

code is available at
https://github.com/mit-pdos/go-journal

Implementation overview

1,300 lines

GoJournal

code is available at
https://github.com/mit-pdos/go-journal

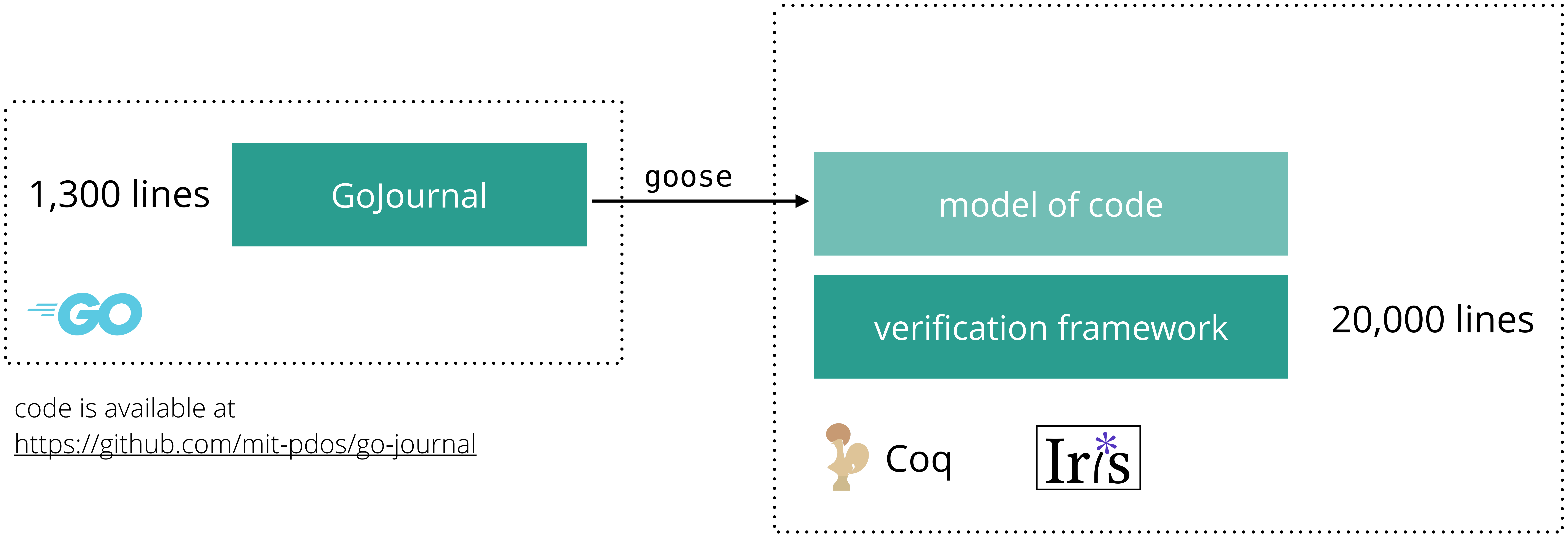
verification framework

20,000 lines

Coq

Iris

Implementation overview



code is available at
https://github.com/mit-pdos/go-journal

Implementation overview

1,300 lines GoJournal

goose

Proof of GoJournal — 26,000 lines

model of code

verification framework — 20,000 lines

Coq Iris

code is available at
https://github.com/mit-pdos/go-journal

Evaluating GoNFS's performance

Implemented GoNFS, an (unverified) NFS server, on top of GoJournal

Compare against Linux kernel NFS server exporting ext4 (with `data=journal` mode for fair comparison)

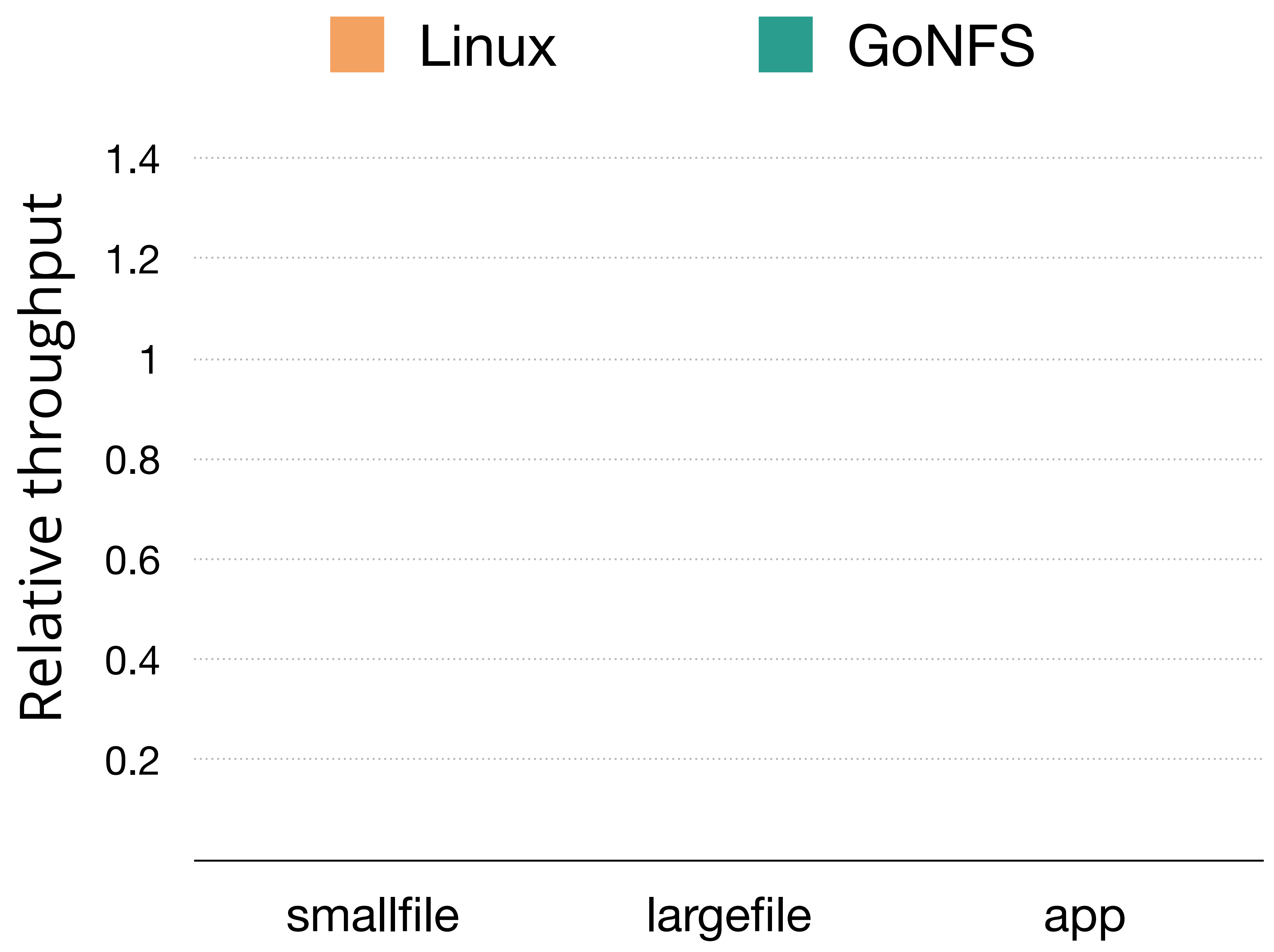
Experimental setup

Hardware: AWS i3.metal
36 cores at 2.3GHz, NVMe SSD

Benchmarks:

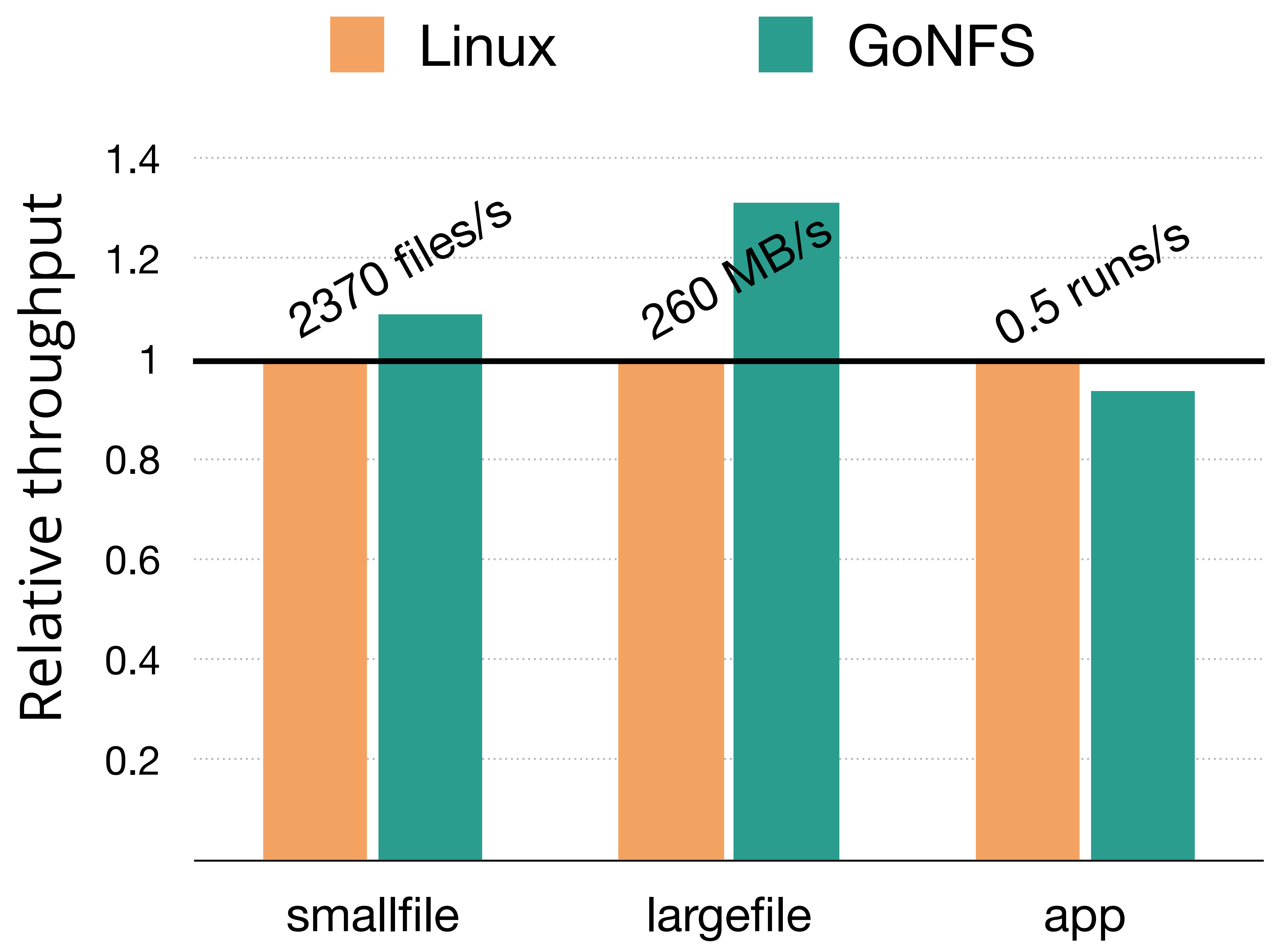
- smallfile: metadata heavy
- largefile: data heavy
- app: `git clone` + `make`

Run using Linux NFS client

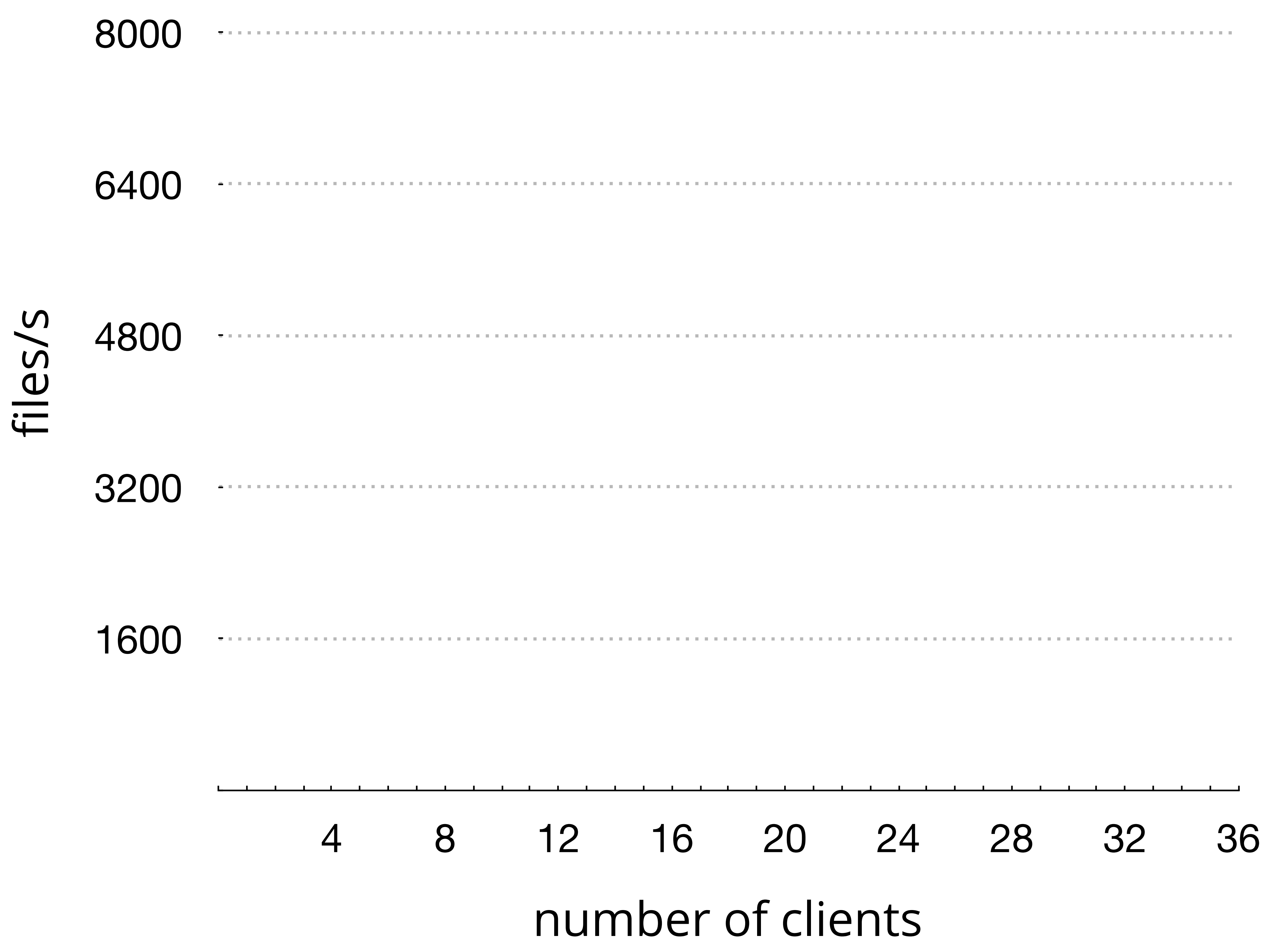


Compare GoNFS throughput to Linux, running on an in-memory disk

GoNFS gets comparable performance even with a single client

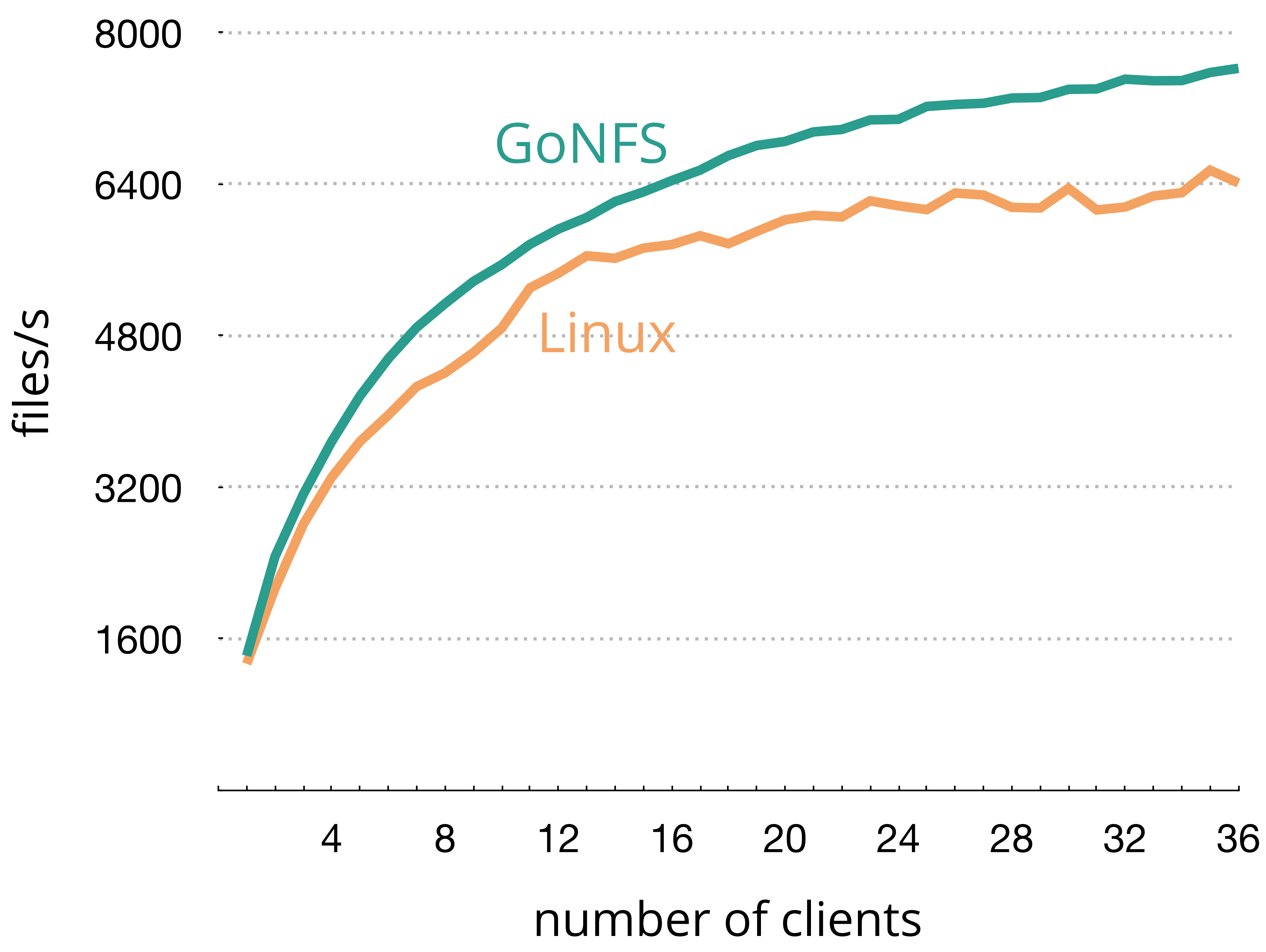


Compare GoNFS throughput to Linux, running on an in-memory disk



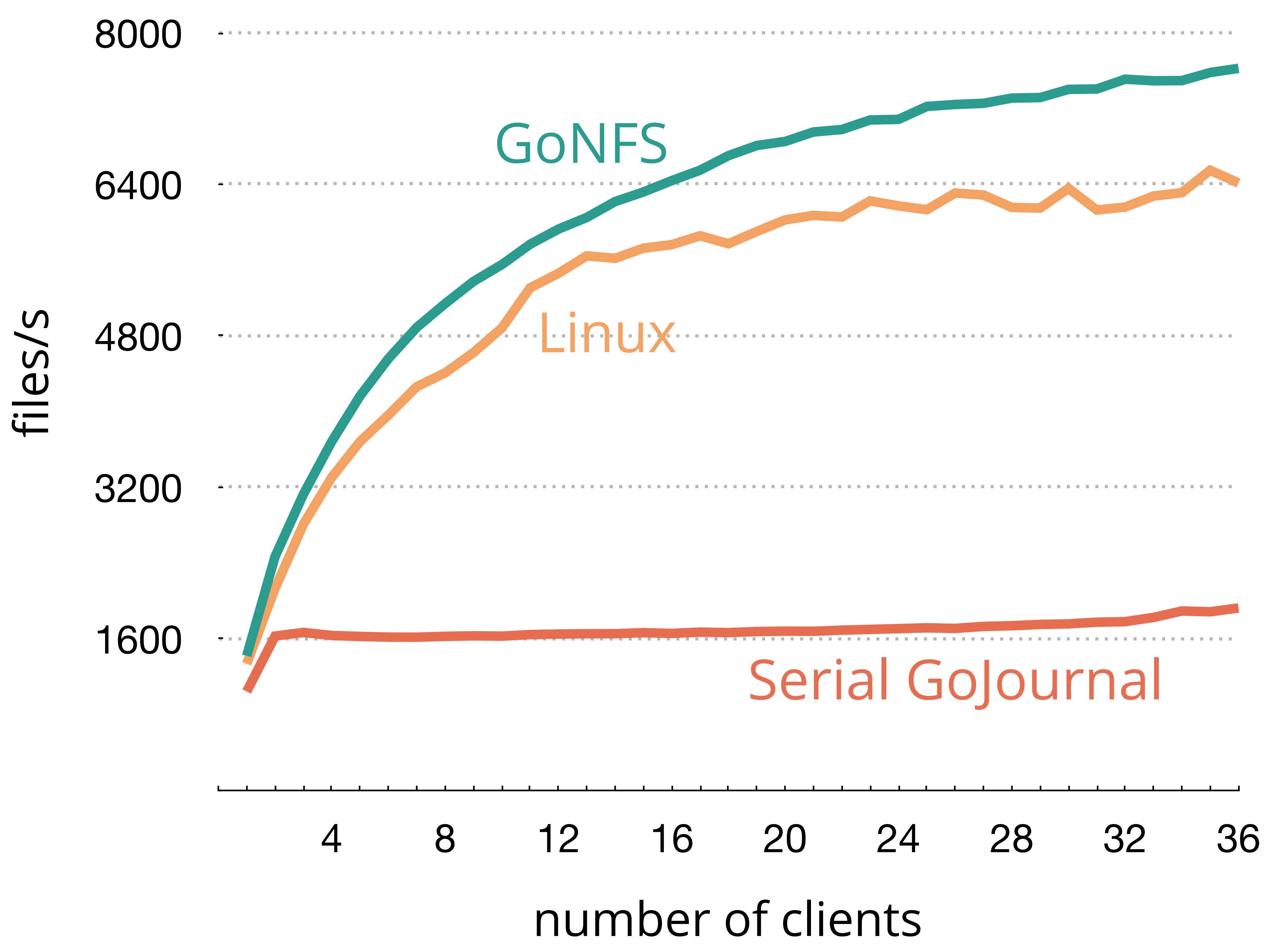
Run smallfile with many clients on an NVMe SSD

GoJournal allows GoNFS to scale with number of clients



Run smallfile with many clients on an NVMe SSD

Concurrency in the journal matters



Serial GoJournal has locks around tricky concurrent parts of WAL

Summary

GoJournal is a verified, concurrent, crash-safe journaling system

Many concurrency challenges in verification

Demonstrate good performance with GoNFS

for followup questions you can contact Tej (tchajed@mit.edu)