Verifying concurrent, crash-safe systems with **Perennial**

Tej Chajed, Joseph Tassarotti*, Frans Kaashoek, Nickolai Zeldovich

MIT and *Boston College

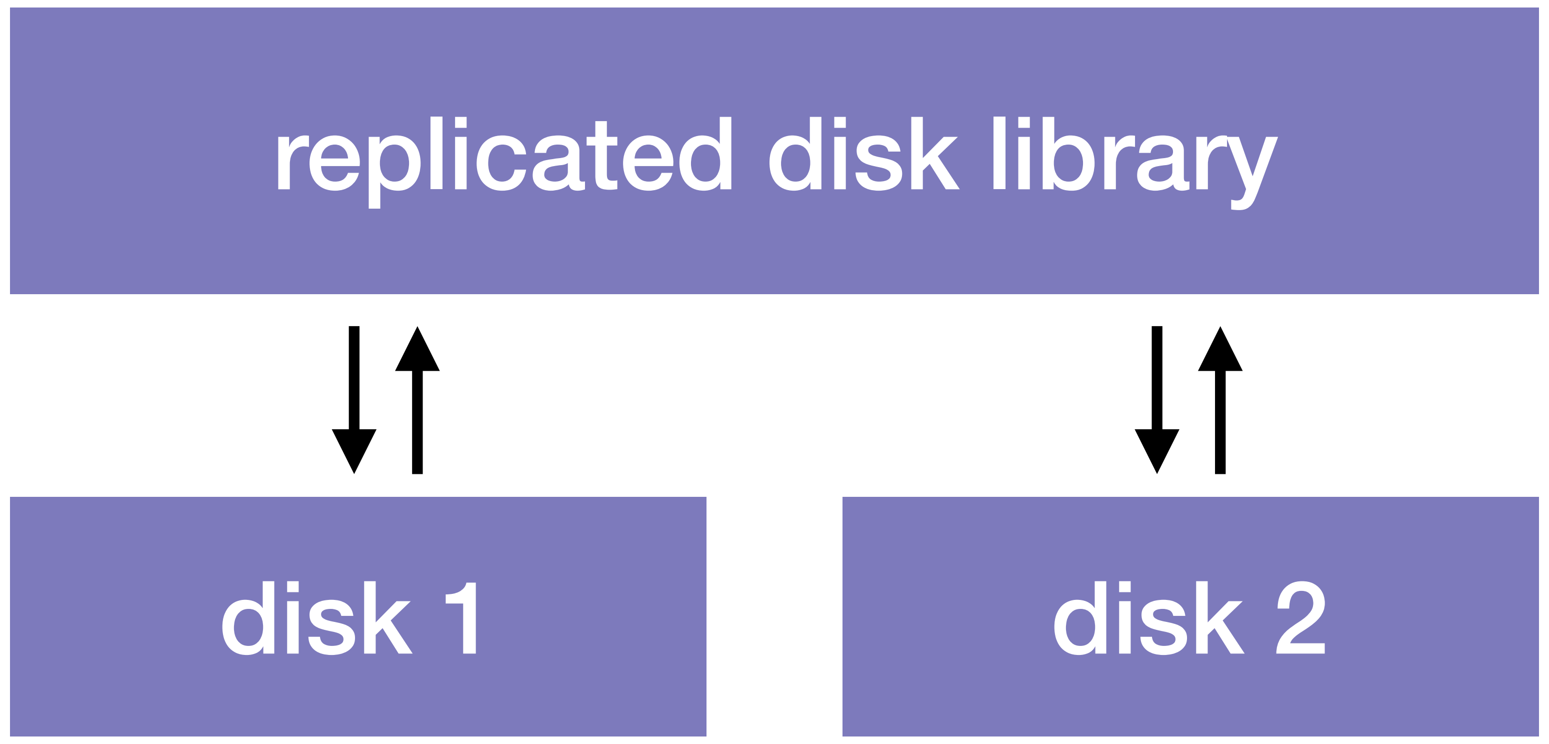
Many systems need concurrency and crash safety

Examples: file systems, databases, and key-value stores

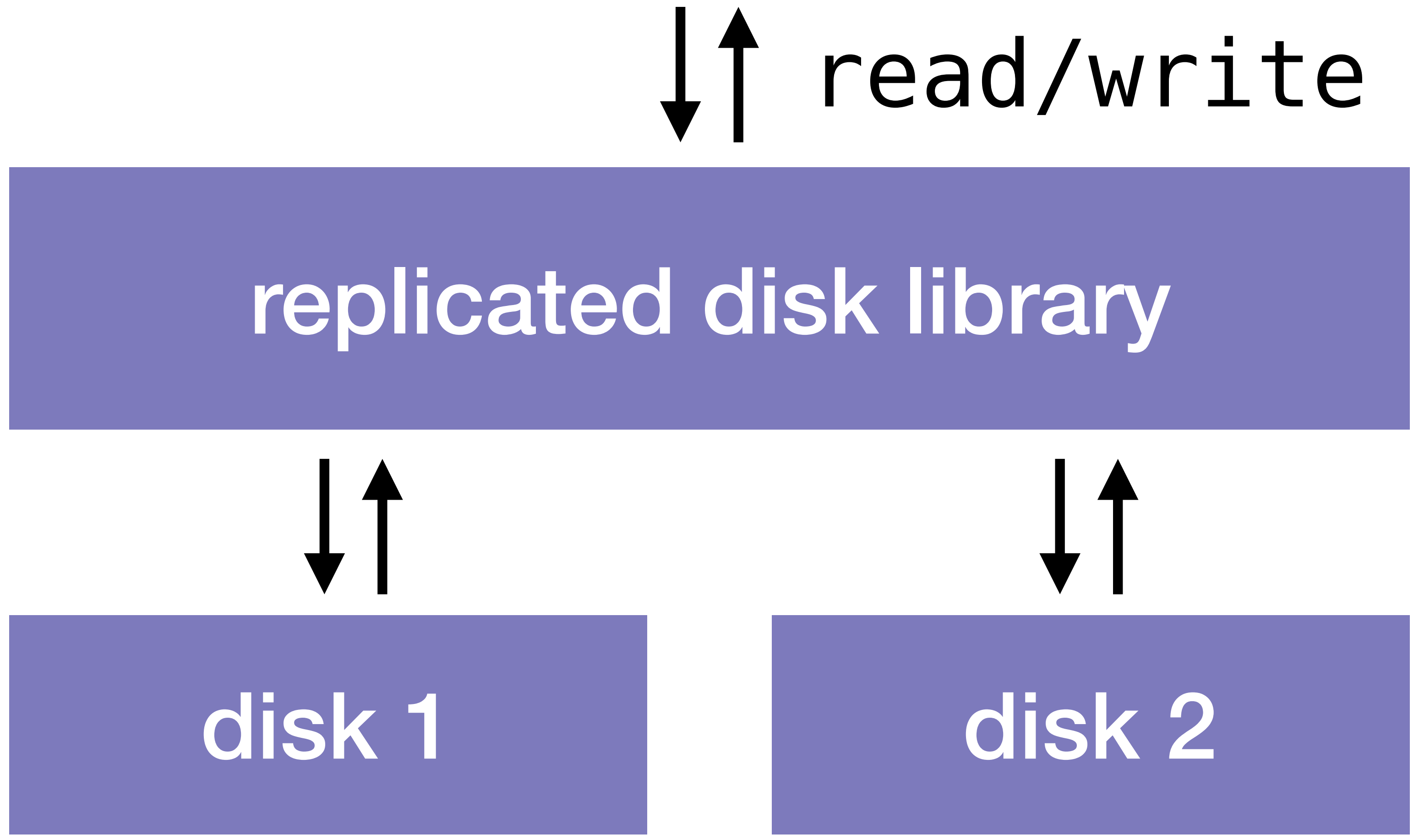
Make strong guarantees about keeping your data safe

Achieve high performance with concurrency

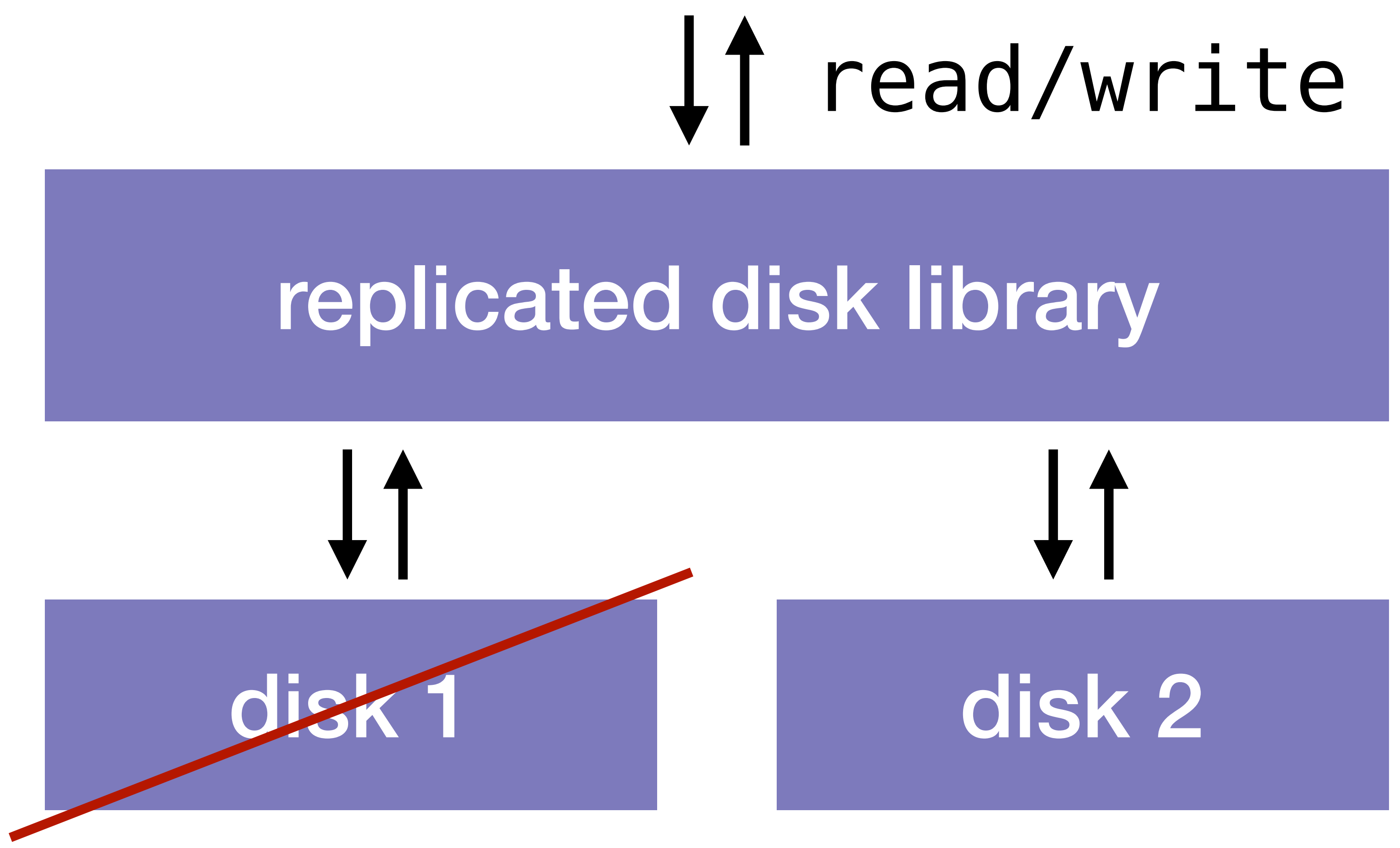
Simple example: replicated disk



Simple example: replicated disk



Simple example: replicated disk



Replicated disk is subtle

```
func write(a: addr, v: block) {
  lock_address(a)
  d1.write(a, v)
  d2.write(a, v)
  unlock_address(a)
}
```

Replicated disk is subtle

```
func write(a: addr, v: block) {
  lock_address(a)
  d1.write(a, v)
  d2.write(a, v)
  unlock_address(a)
}
```

what if system crashes here?
what if disk 1 fails?

Replicated disk is subtle

```
func write(a: addr, v: block) {
  lock_address(a)
  d1.write(a, v)
  d2.write(a, v)
  unlock_address(a)
}
```

what if system crashes here?

what if disk 1 fails?

```
// runs on reboot
func recover() {
  for a in … {
    // copy from d1 to d2
  }
}
```

Replicated disk is subtle

```
func write(a: addr, v: block) {
  lock_address(a)
  d1.write(a, v)
  d2.write(a, v)
  unlock_address(a)
}
```

what if system crashes here?
what if disk 1 fails?

```
// runs on reboot
func recover() {
  for a in … {
    // copy from d1 to d2
  }
}
```

```
func read(a: addr): block {
  lock_address(a)
  v, ok := d1.read(a)
  if !ok {
    v, _ = d2.read(a)
  }
  unlock_address(a)
  return v
}
```

Goal: systematically reason about all executions with formal verification

Existing verification frameworks do not support concurrency and crash safety

verified crash safety

FSCQ [SOSP ’15]
Yggdrasil [OSDI ’16]
DFSCQ [SOSP ’17]
…

verified concurrency

CertiKOS [OSDI ’16]
CSPEC [OSDI ’18]
AtomFS [SOSP ’19]
…

no system can do both

Combining verified crash safety and concurrency is challenging

Crash and recovery can interrupt a critical section

Crash wipes in-memory state

Recovery logically completes crashed threads' operations

Perennial's techniques address challenges integrating crash safety into concurrency reasoning

Crash and recovery can interrupt a critical section

➡ **leases**

Crash wipes in-memory state

➡ **memory versioning**

Recovery logically completes crashed threads' operations

➡ **recovery helping**

Perennial's techniques address challenges integrating crash safety into concurrency reasoning

see paper

Crash and recovery can interrupt a critical section

➡ **leases**

Crash wipes in-memory state

➡ **memory versioning**

this talk

Recovery logically completes crashed threads' operations

➡ **recovery helping**

Contributions

Perennial: framework for reasoning about crashes and concurrency

see paper Goose: reasoning about Go implementations

Evaluation: verified mail server written in Go with Perennial

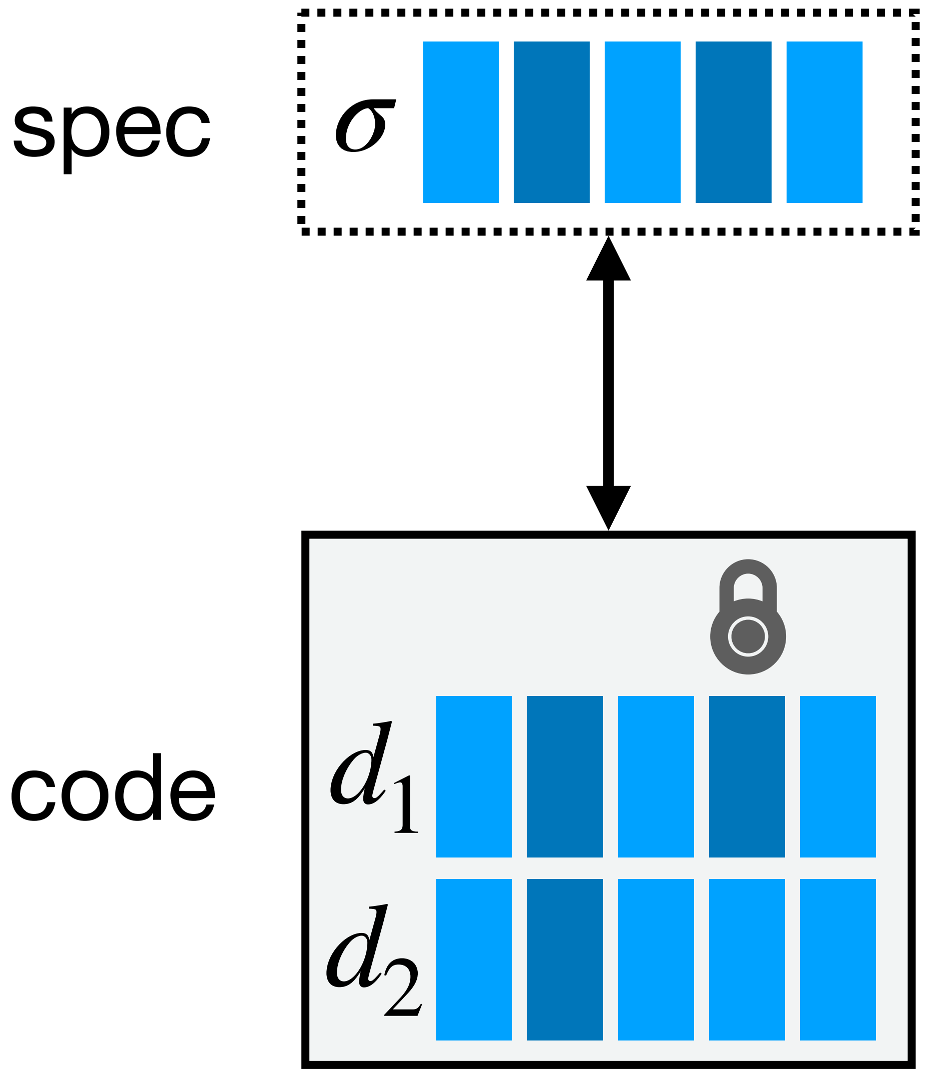
Specifying correctness: concurrent recovery refinement

All operations are **correct** and **atomic wrt concurrency and crashes**

Recovery repairs system after reboot

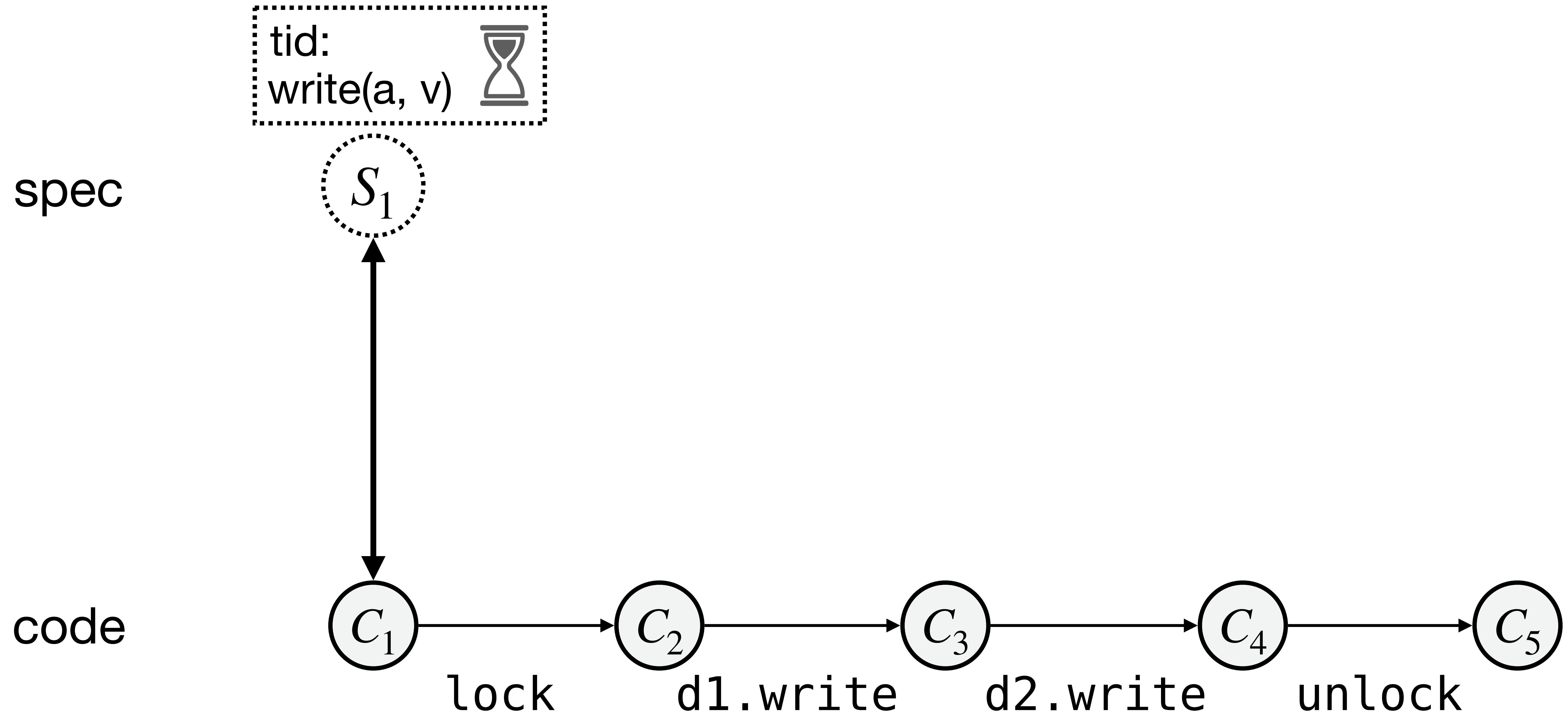
Proving the replicated disk correct

Proving refinement with forward simulation: relate code and spec states



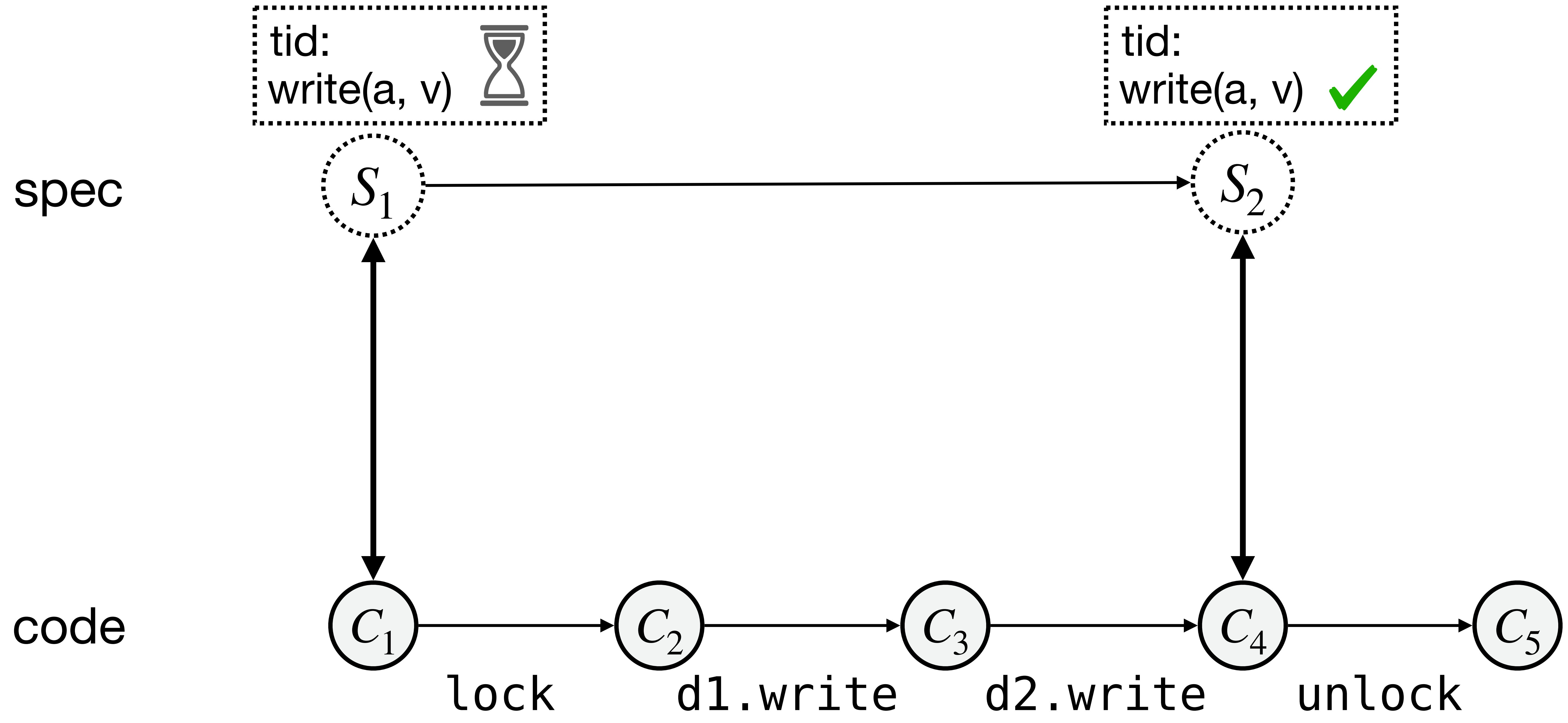
Proving refinement with forward simulation: prove every operation has a commit point

Background



1. Write down abstraction relation between code and spec states

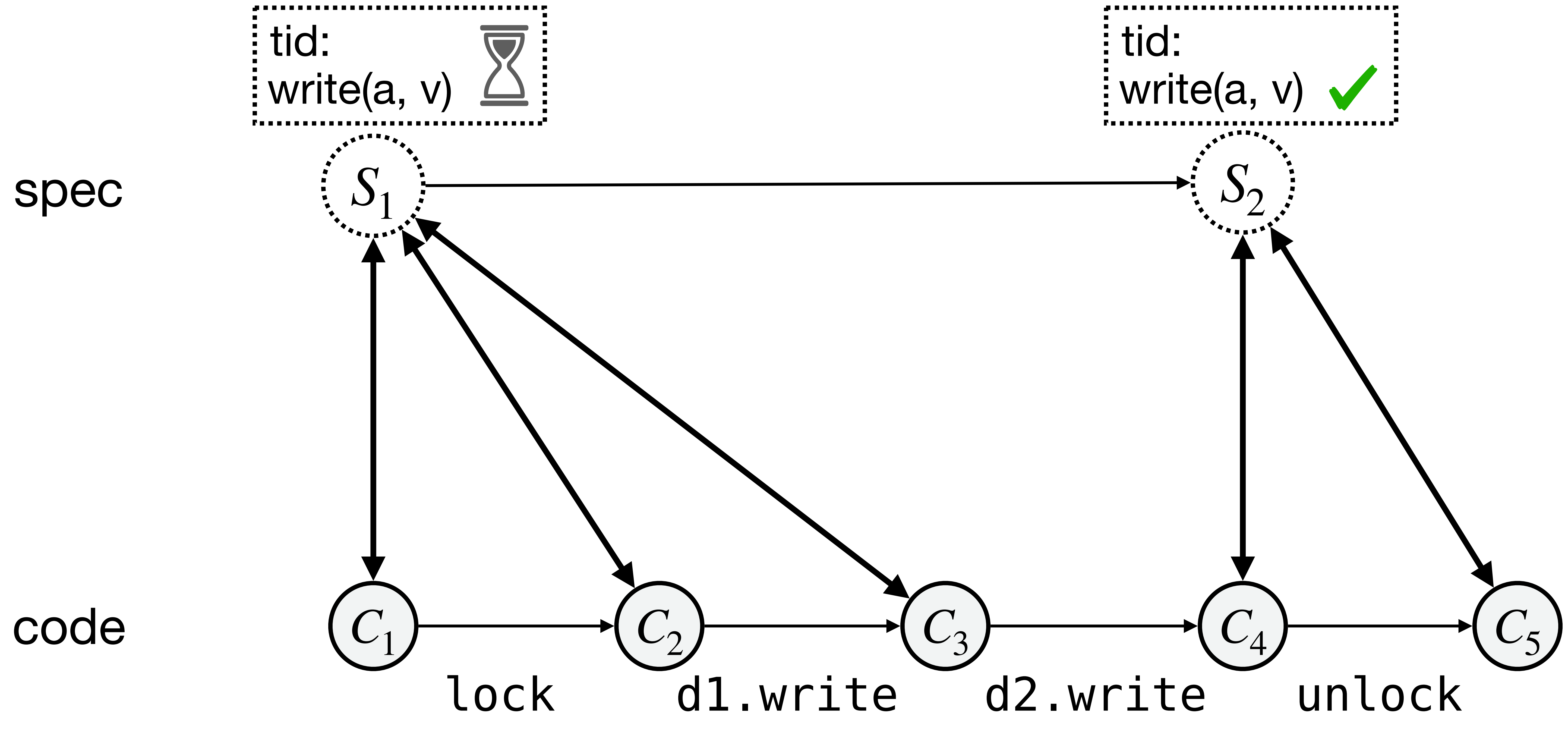
Proving refinement with forward simulation: prove every operation has a commit point



1. Write down abstraction relation between code and spec states
2. Prove every operation commits

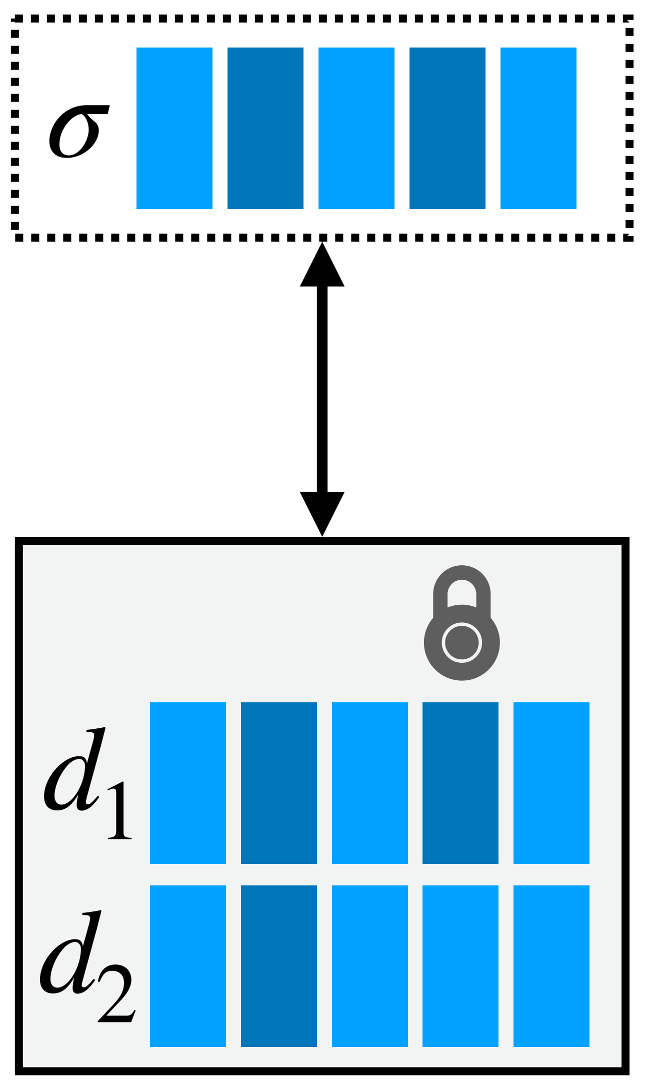
Proving refinement with forward simulation: prove every operation has a commit point

Background



1. Write down abstraction relation between code and spec states
2. Prove every operation commits
3. Prove abstraction relation is preserved

Abstraction relation for the replicated disk



abstraction relation:

$$\textbf{!locked}(a) \implies \begin{array}{r} \sigma[a] = d_1[a] \\ \wedge\ \sigma[a] = d_2[a] \end{array}$$

(if the disk has not failed)

Crashing breaks the abstraction relation

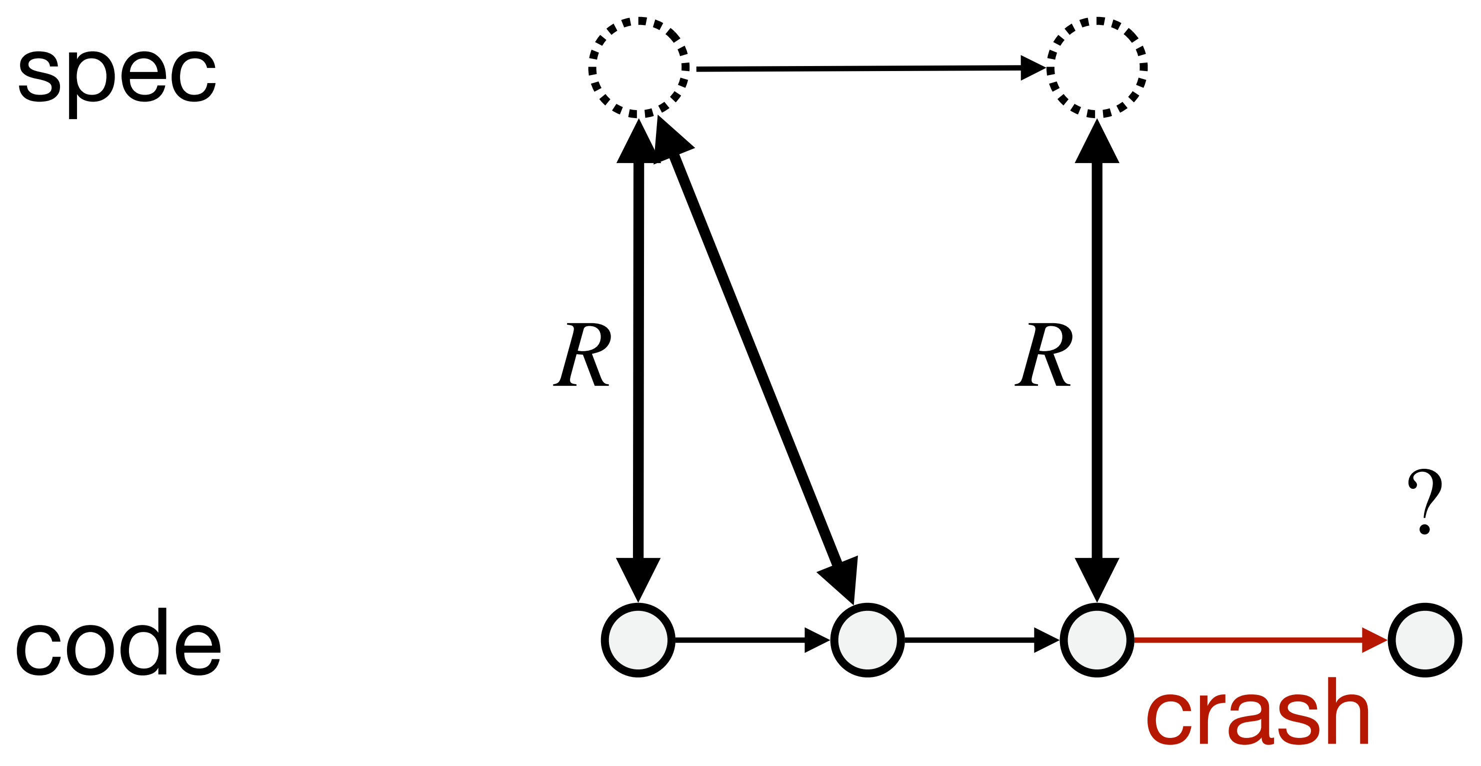
```
func write(a: addr,
           v: block) {

  lock(a)
  d1.write(a, v)
```

lock reverts to being free,
but disks are not in-sync

abstraction relation:

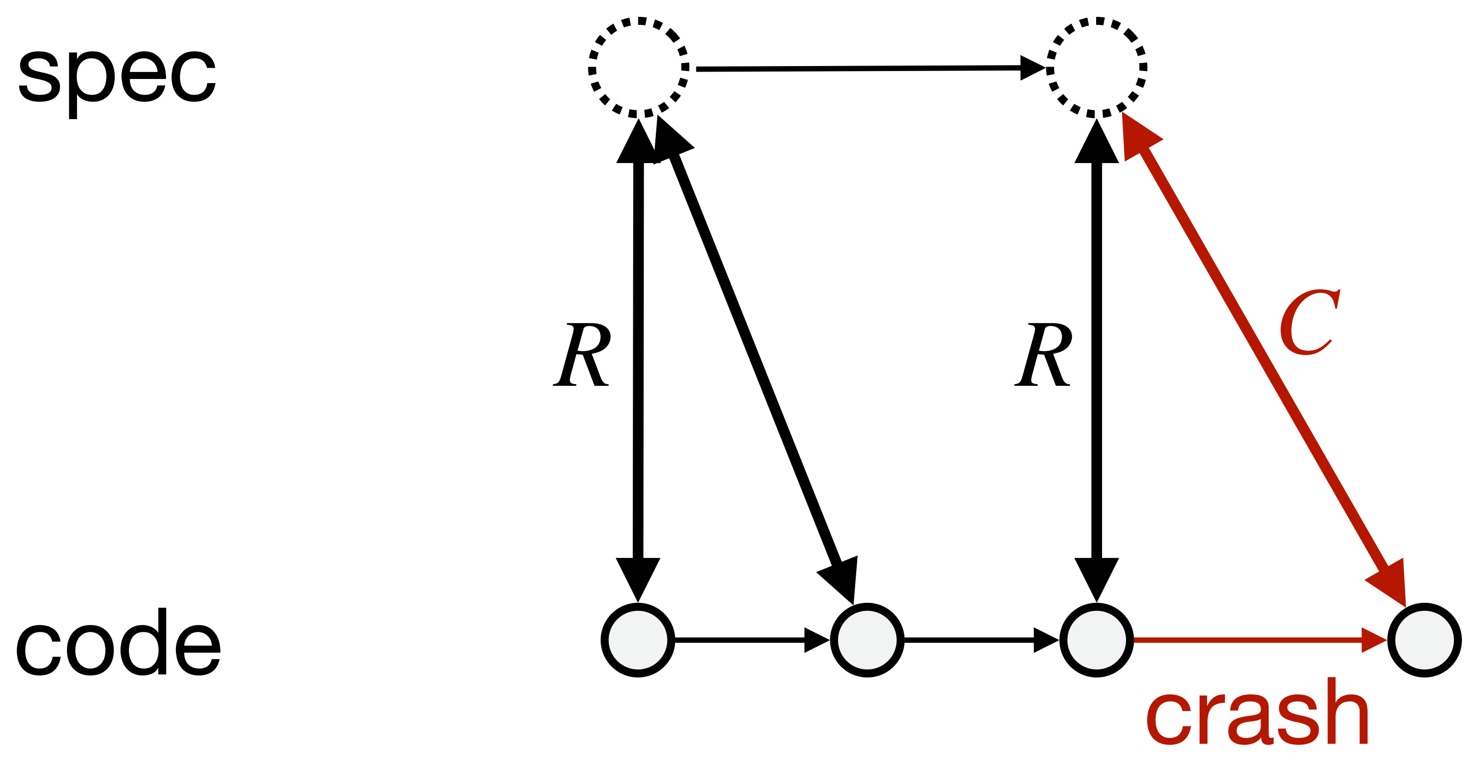
$$\textbf{!locked}(a) \implies \begin{array}{r} \sigma[a] = d_1[a] \\ \wedge\ \sigma[a] = d_2[a] \end{array}$$

So far: abstraction relation always holds



R abstraction relation

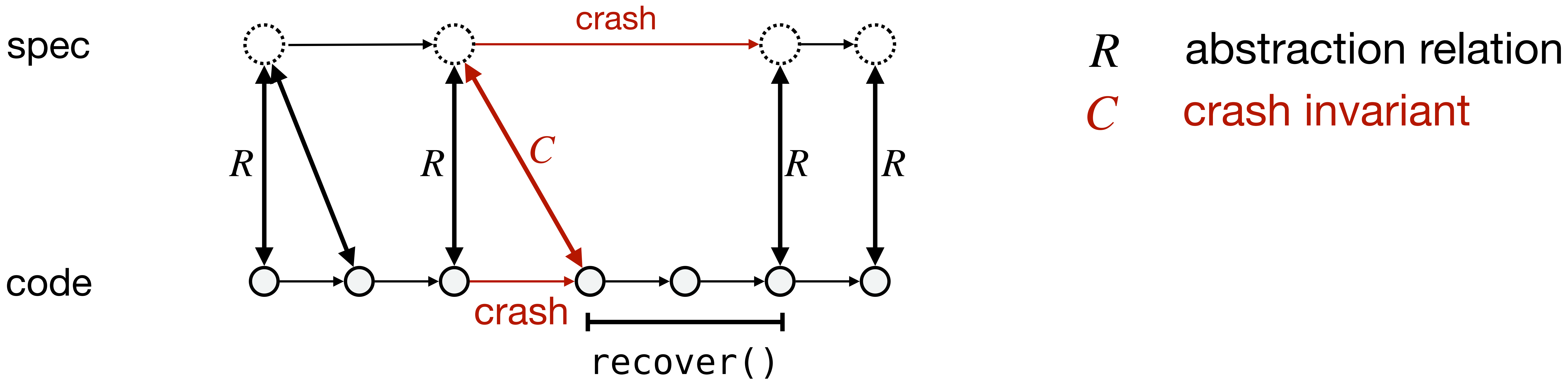
Separate a crash invariant from the abstraction relation



R abstraction relation

C crash invariant

Recovery proof uses the crash invariant to restore the abstraction relation



Proving recovery correct: makes writes atomic

```
func write(a: addr,
           v: block) {

  lock_address(a)
  d1.write(a, v)


func recover() {
  for a in … {
    v, ok := d1.read(a)
    if !ok { … }
    d2.write(a, v)
  }

}
```

Recovery helping: recovery can commit writes from before the crash

```
func write(a: addr,
           v: block) {

  lock_address(a)
  d1.write(a, v)
```

```
func recover() {
  for a in … {
    v, ok := d1.read(a)
    if !ok { … }
    d2.write(a, v)
  }

}
```

Recovery helping: recovery can commit writes from before the crash

```
func write(a: addr,
           v: block) {

  lock_address(a)
  d1.write(a, v)
```

```
func recover() {
  for a in … {
    v, ok := d1.read(a)
    if !ok { … }
    d2.write(a, v)
  }

}
```

tid:
write(a, v)

a pending spec operation

Recovery helping: recovery can commit writes from before the crash

```
func write(a: addr,
           v: block) {

  lock_address(a)
  d1.write(a, v)
```

tid:
write(a, v)

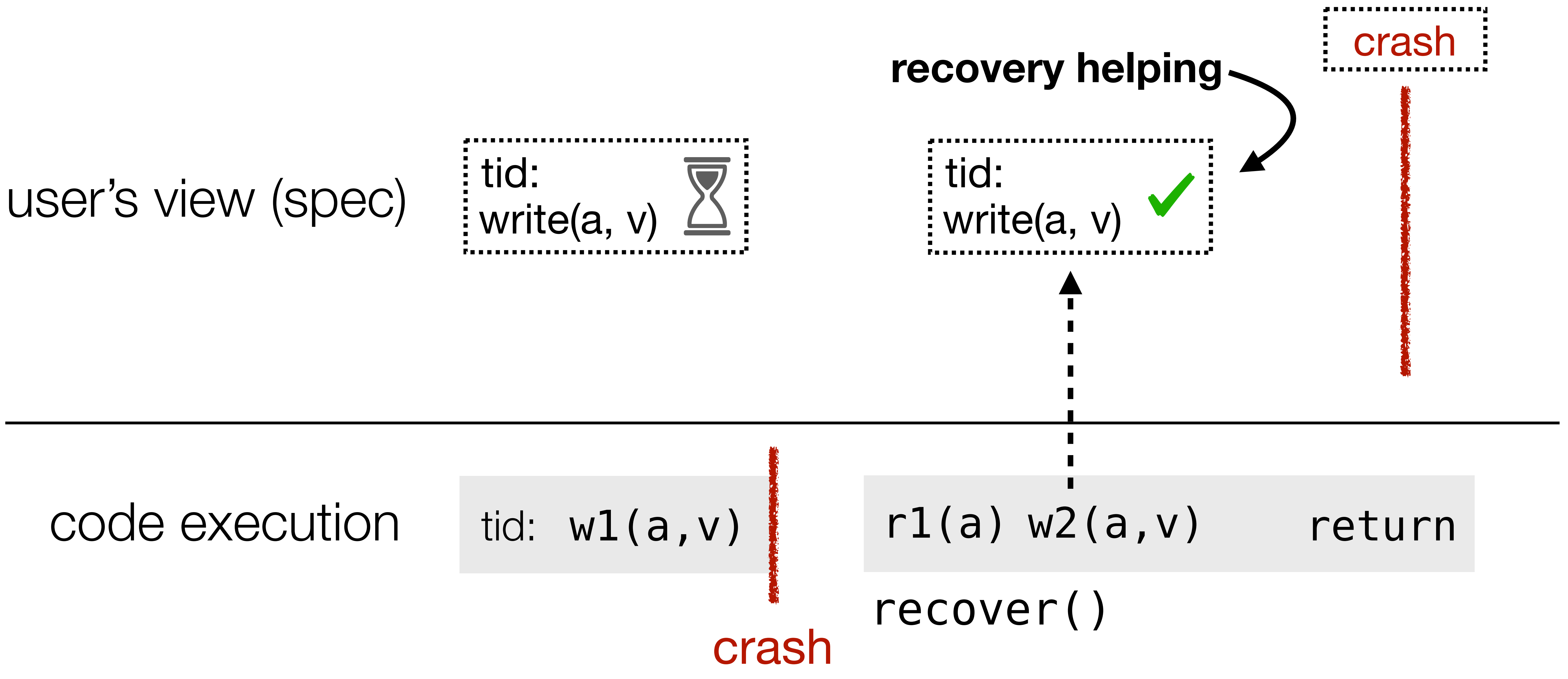
a pending spec operation

```
func recover() {
  for a in … {
    v, ok := d1.read(a)
    if !ok { … }
    d2.write(a, v)
  }

}
```

tid:
write(a, v)

recovery commits the interrupted operation

User sees an atomic write even with recovery helping



Crash invariant says "if disks disagree, some thread was writing the value on the first disk"

```
func write(a: addr,
           v: block) {

  lock_address(a)
  d1.write(a, v)
```

tid:
write(a, v)

```
func recover() {
  for a in … {
    v, ok := d1.read(a)
    if !ok { … }
    d2.write(a, v)   ------>
  }

}
```

tid:
write(a, v)

crash invariant:

$d_1[a] \neq d_2[a] \implies \exists \text{tid.}$ tid: write(a, $d_1[a]$)

Crash invariant says "if disks disagree, some thread was writing the value on the first disk"

```
func write(a: addr,
           v: block) {

  lock_address(a)
  d1.write(a, v)
```

tid:
write(a, v)

crash invariant:

$d_1[a] \neq d_2[a] \implies$

$\exists \text{tid.}$ tid:
write(a, $d_1[a]$)

```
func recover() {
  for a in … {
    v, ok := d1.read(a)
    if !ok { … }
    d2.write(a, v)
  }

}
```

tid:
write(a, v)

Key idea: crash invariant can refer to interrupted spec operations

```
func write(a: addr,
           v: block) {

  lock_address(a)
  d1.write(a, v)

func recover() {
  for a in … {
    v, ok := d1.read(a)
    if !ok { … }
    d2.write(a, v)
  }

}
```

tid:
write(a, v)

tid:
write(a, v) ✓

crash invariant:

$d_1[a] \neq d_2[a] \implies$
$\exists$tid. tid: write(a, $d_1[a]$)

Recovery proof shows code restores the abstraction relation by completing all interrupted writes

```
func write(a: addr,
           v: block) {

  lock_address(a)
  d1.write(a, v)
```

tid:
write(a, v)

```
func recover() {
  for a in … {
    v, ok := d1.read(a)
    if !ok { … }
    d2.write(a, v)
  }

}
```

tid:
write(a, v)

crash

abstraction relation:

$$\textbf{!locked}(a) \implies \begin{array}{r} \sigma[a] = d_1[a] \\ \land\ \sigma[a] = d_2[a] \end{array}$$

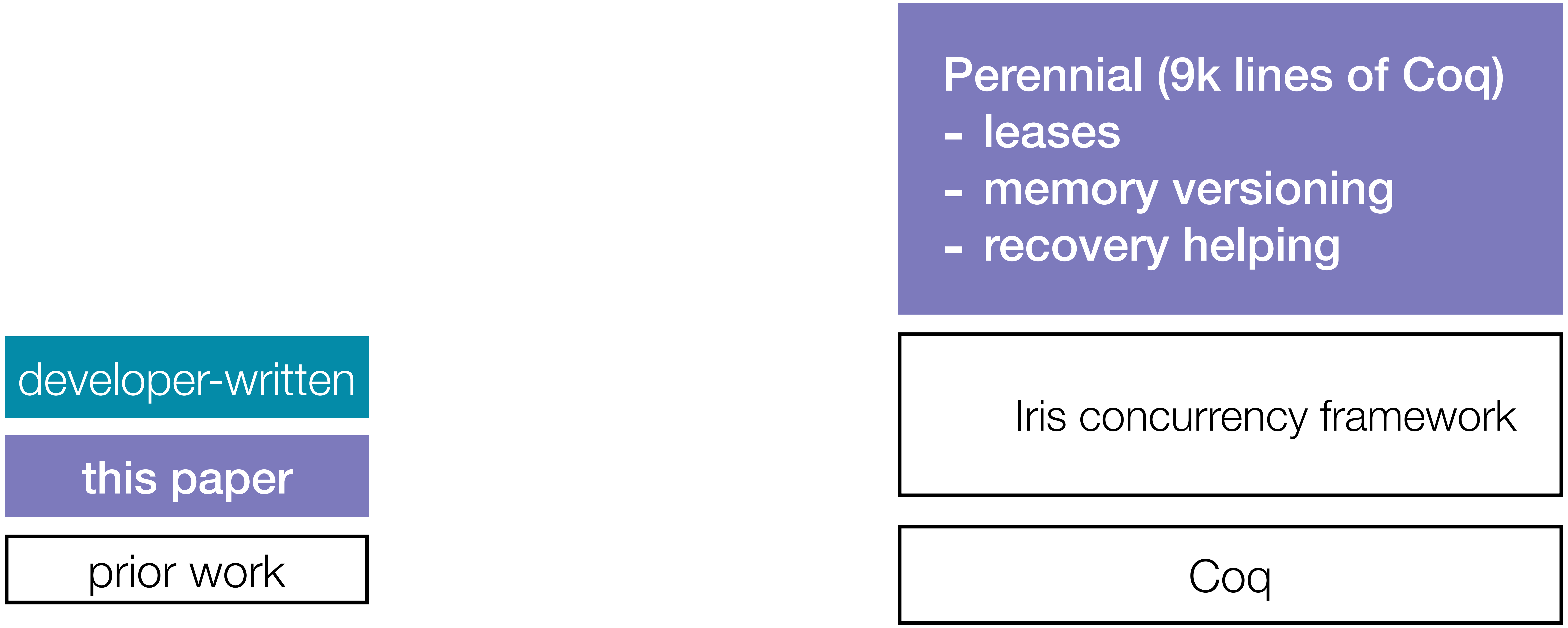
Proving concurrent recovery refinement

Recovery proof uses **crash invariant** to restore abstraction relation

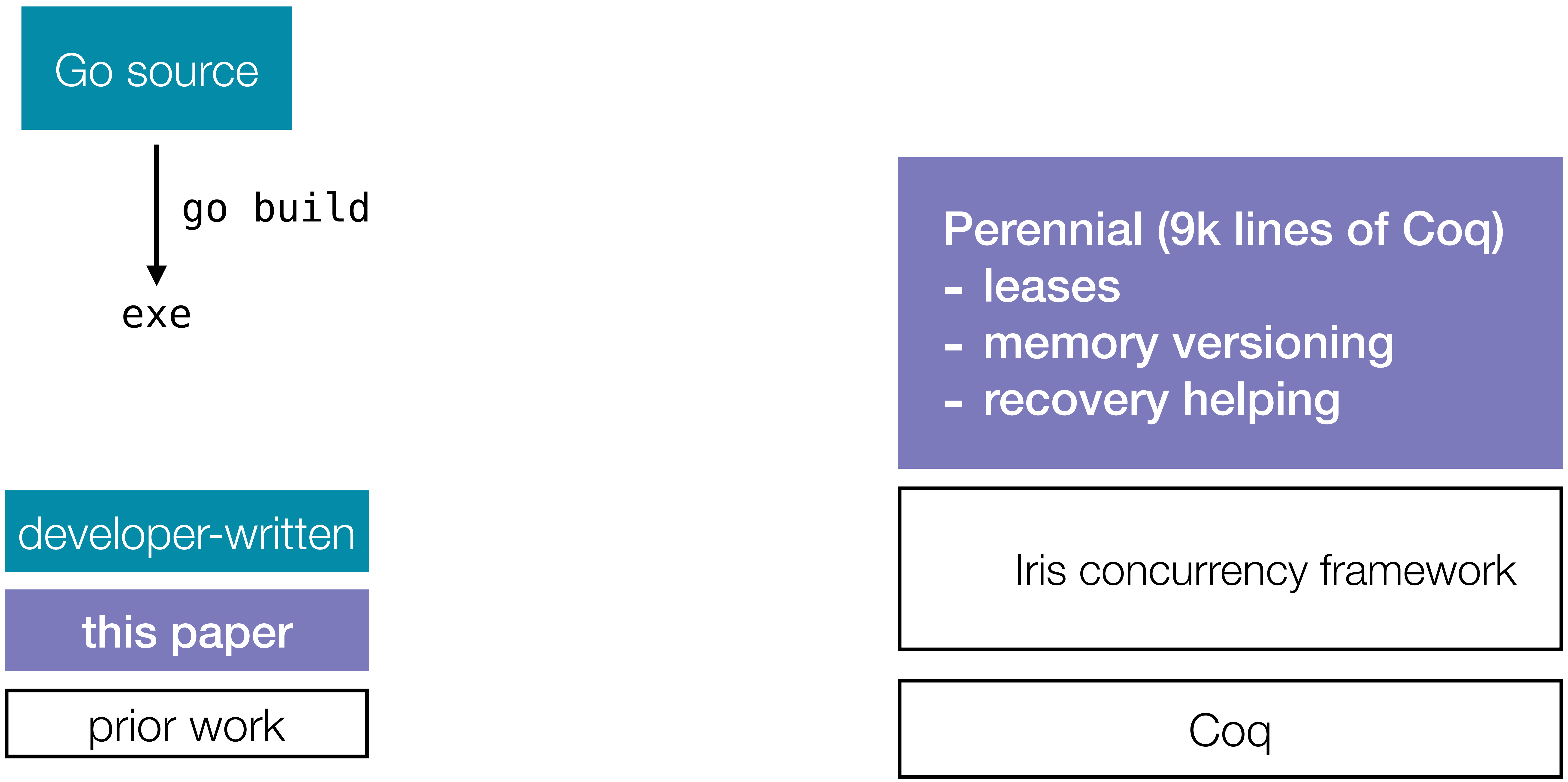
Proof can refer to interrupted operations, enabling **recovery helping** reasoning

Users get **correct behavior** and **atomicity**

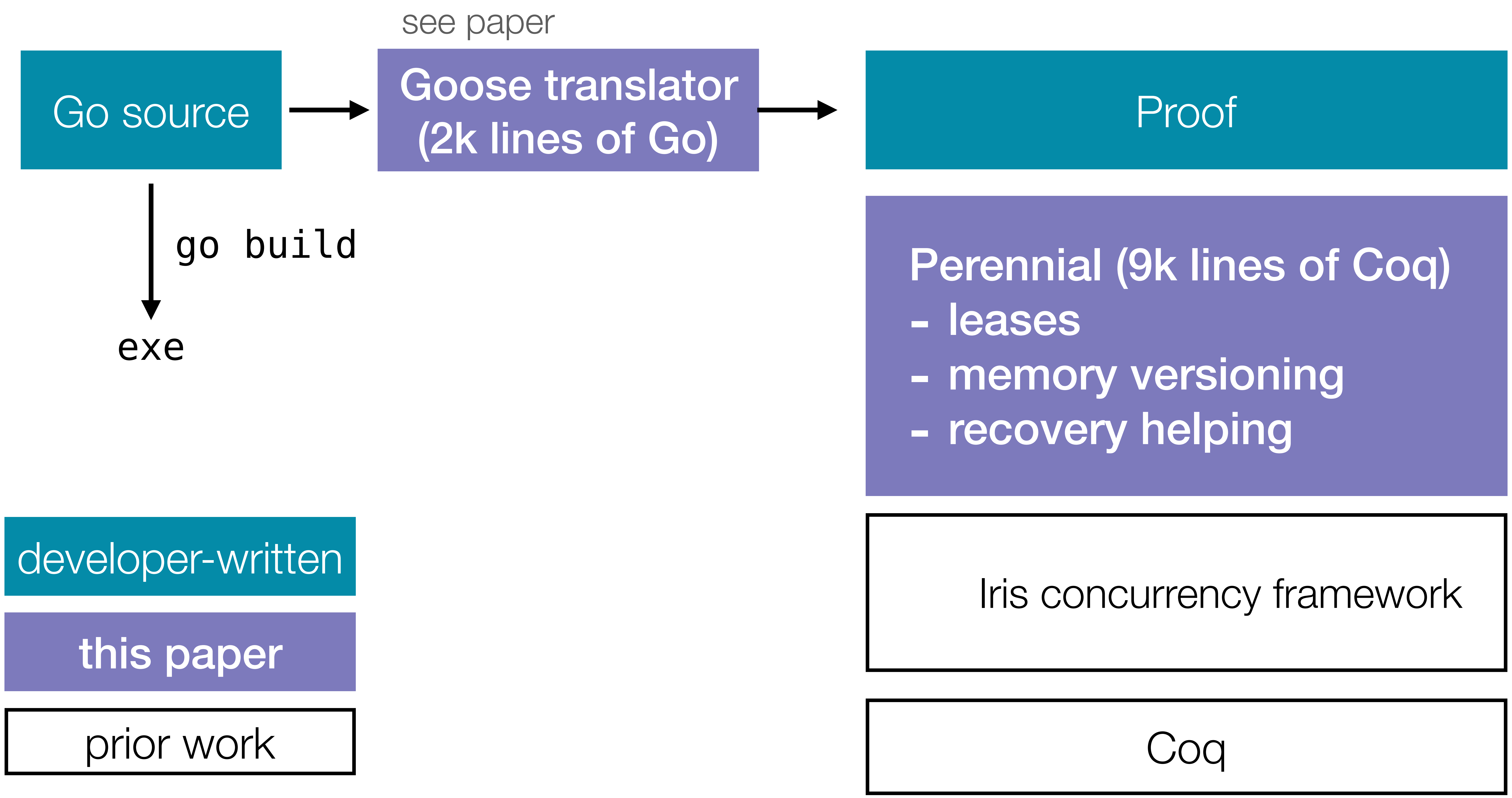
Implementation



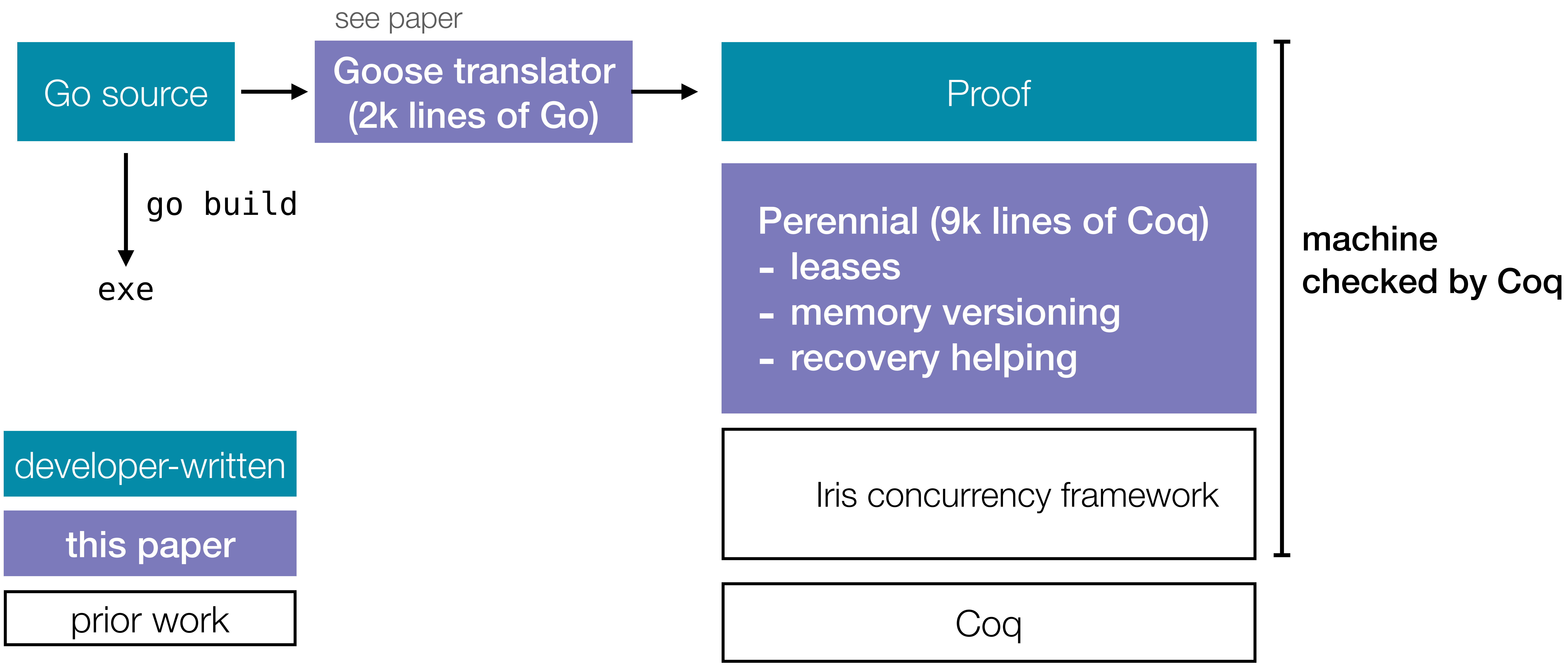
Implementation



Implementation



Implementation



Evaluation

This talk:

- proof-effort comparison

See paper:

- verified examples
- TCB
- bug discussion

Methodology:
Verify the same mail server as previous work, **CSPEC** [OSDI ’18]

Users can read, deliver, and delete mail

Implemented on top of a file system

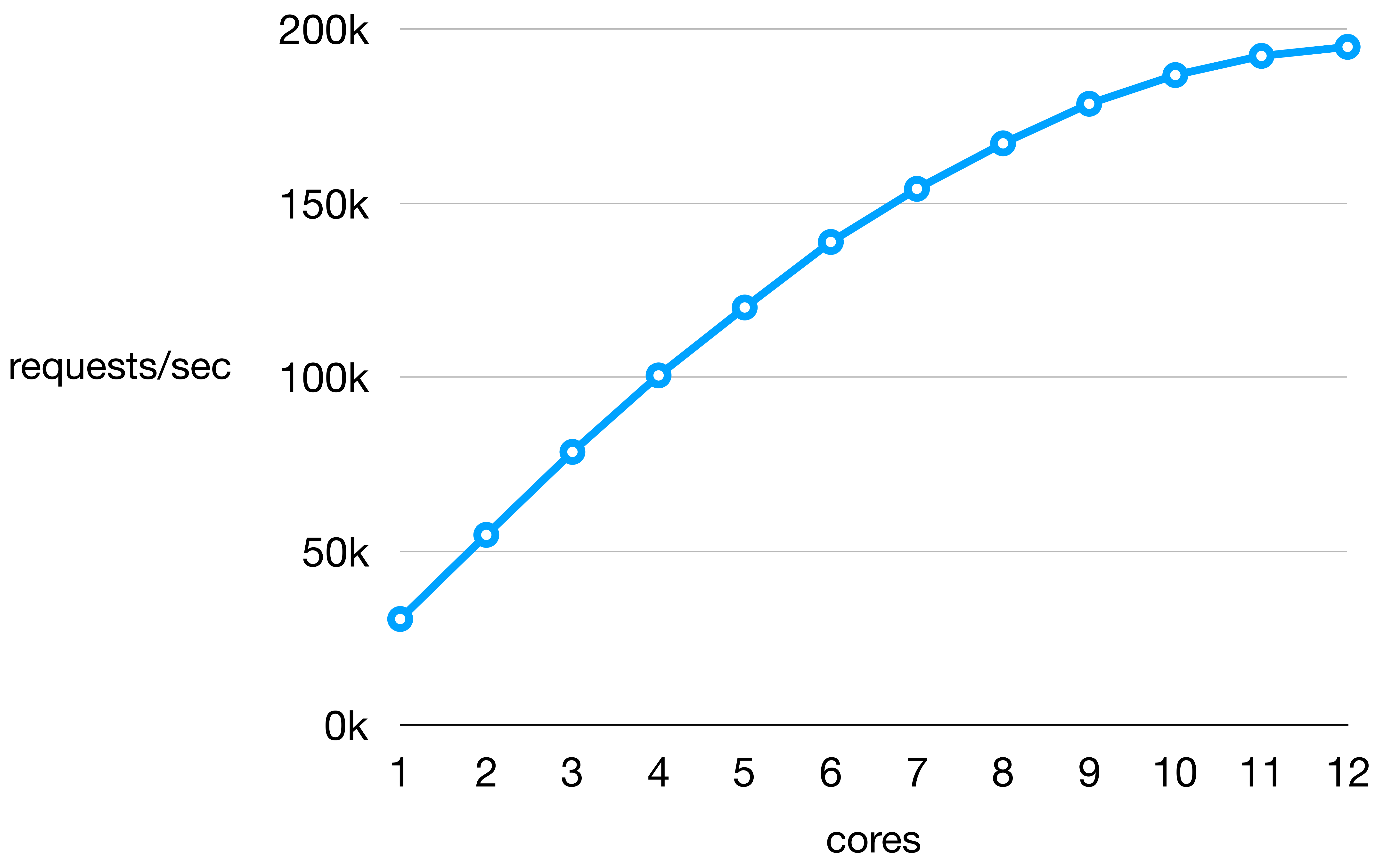
Operations are **atomic** (and **crash safe** in Perennial)

Perennial mail server was easier to verify and proves crash safety

	Perennial	**CSPEC** [OSDI '18]
mail server proof	3,200	4,000
time	2 weeks (**after** framework)	6 months (**with** framework)
code	159 (Go)	215 (Coq)

Perennial mail server really is concurrent

(see the paper for details)



Conclusion

Perennial introduces **crash-safety techniques** that extend concurrent verification in Iris

Goose lets us reason about **Go implementations**

Verified a Go mail server with **less effort** than previous work and proved crash safety

chajed.io/perennial