The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors

Austin T. Clements

Thesis advisors:
M. Frans Kaashoek
Nickolai Zeldovich
Robert Morris
Eddie Kohler

Presented by: Kostis Tsesmelis

Outline

Introduction
- Presenting the rule
- Definition of scalability
- Intuition/Examples

Applying the rule
- Commuter
- Evaluation

Formalizing the rule
- Definitions
- Proof

Current approach to scalable software development

2009 — **Corey**
OSDI '08

2010

2011 — **Linux scalability**
OSDI '10

2012 — **Bonsai VM**
ASPLOS '12

2013

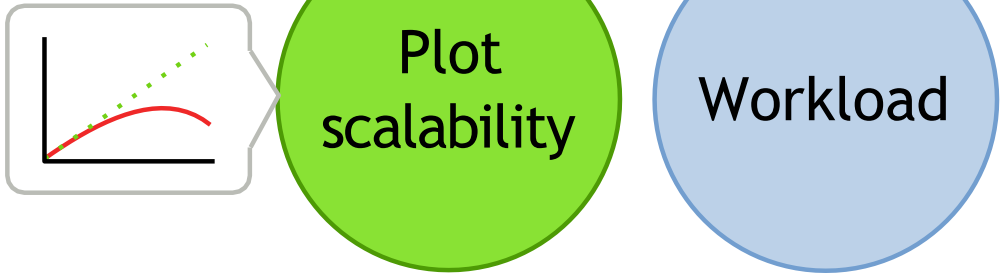
2014 — **RadixVM**
EuroSys '13

Current approach to scalable software development

2009 — Corey
OSDI '08

2010 — Linux scalability
OSDI '10

2011



2012 — Bonsai VM
ASPLOS '12

2013 — RadixVM
EuroSys '13

2014

Current approach to scalable software development

2009 — Corey
OSDI '08

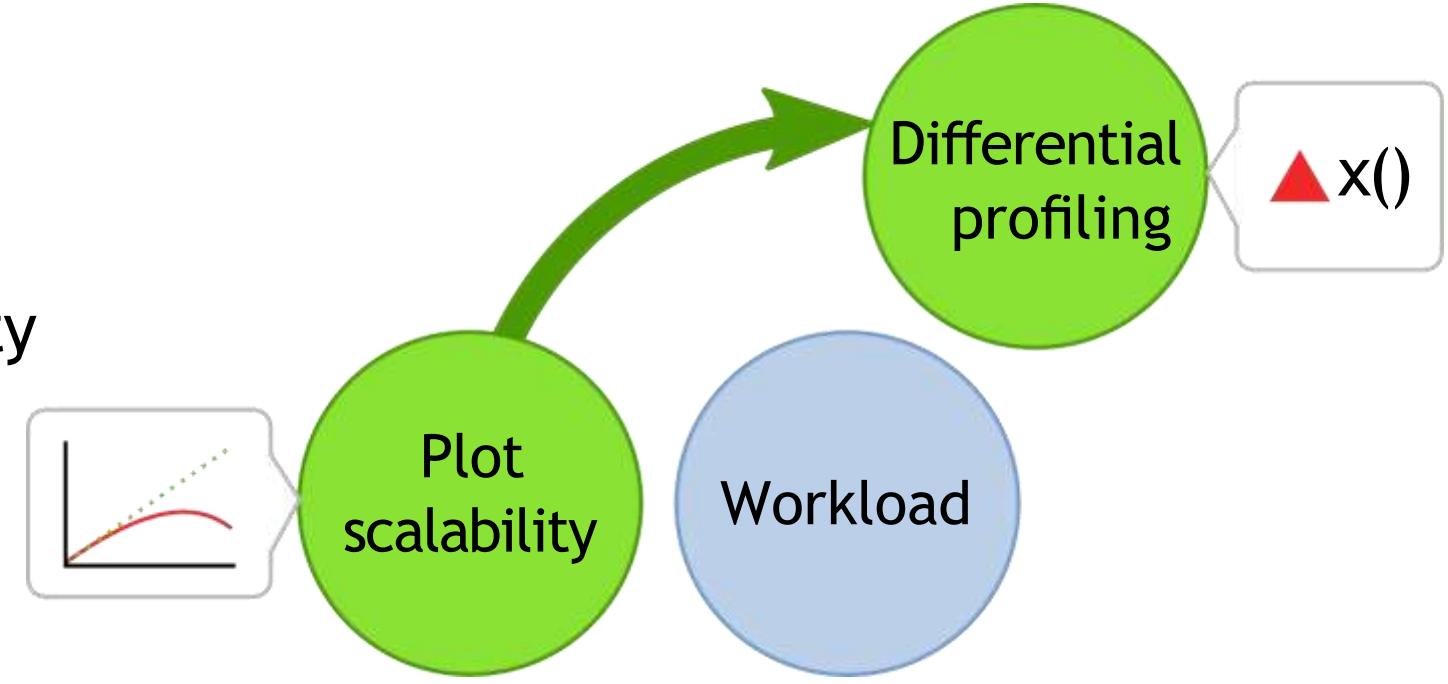
2010 — Linux scalability
OSDI '10

2011

2012 — Bonsai VM
ASPLOS '12

2013 — RadixVM
EuroSys '13

2014



Current approach to scalable software development

2009 — Corey
OSDI '08

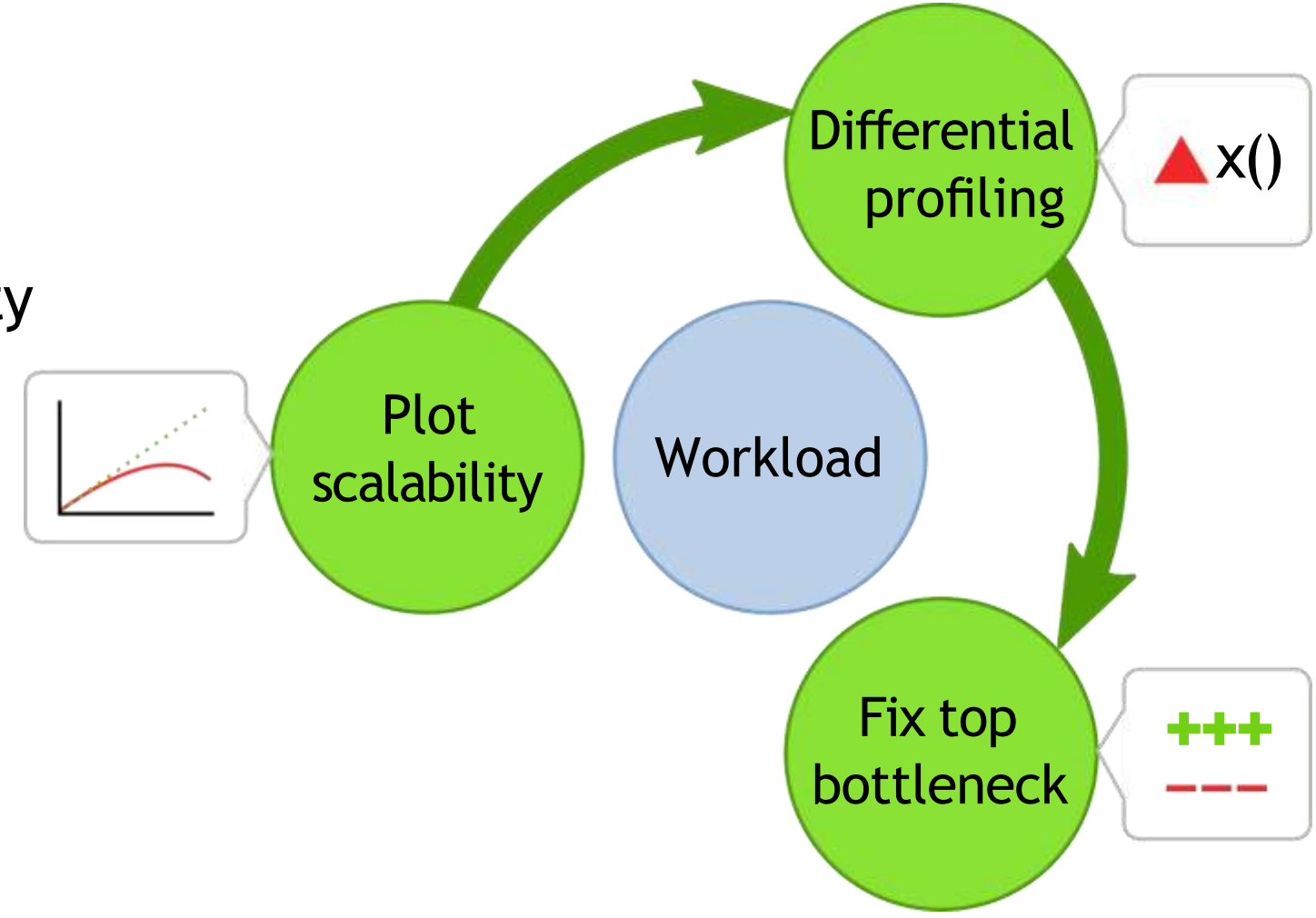
2010 — Linux scalability
OSDI '10

2011

2012 — Bonsai VM
ASPLOS '12

2013 — RadixVM
EuroSys '13

2014



Current approach to scalable software development

2009 — Corey
OSDI '08

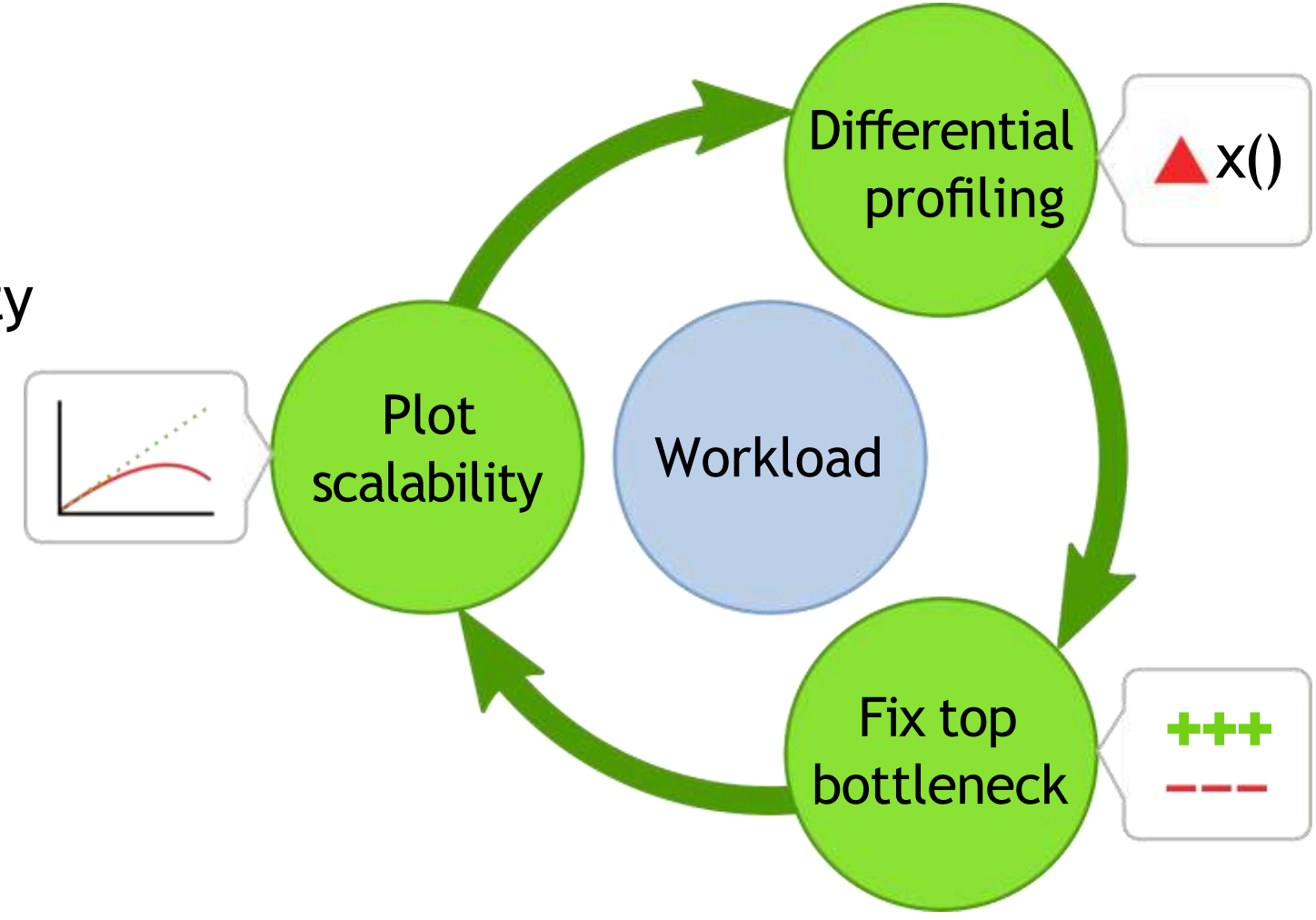
2010 — Linux scalability
OSDI '10

2011

2012 — Bonsai VM
ASPLOS '12

2013 — RadixVM
EuroSys '13

2014



Current approach to scalable software development

Disadvantages

- Requires huge amounts of effort
- New workloads expose new bottlenecks
- More cores expose new bottlenecks
- The real bottlenecks may be in the interface design

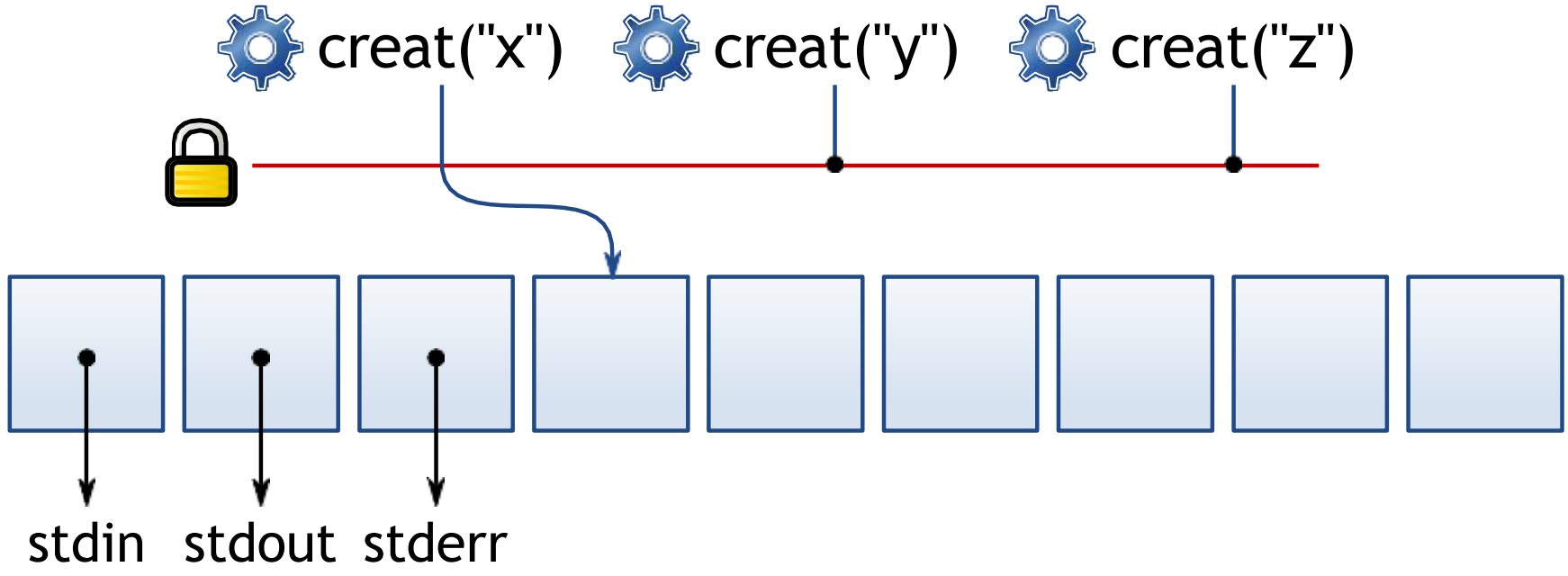
Current approach to scalable software development

Disadvantages

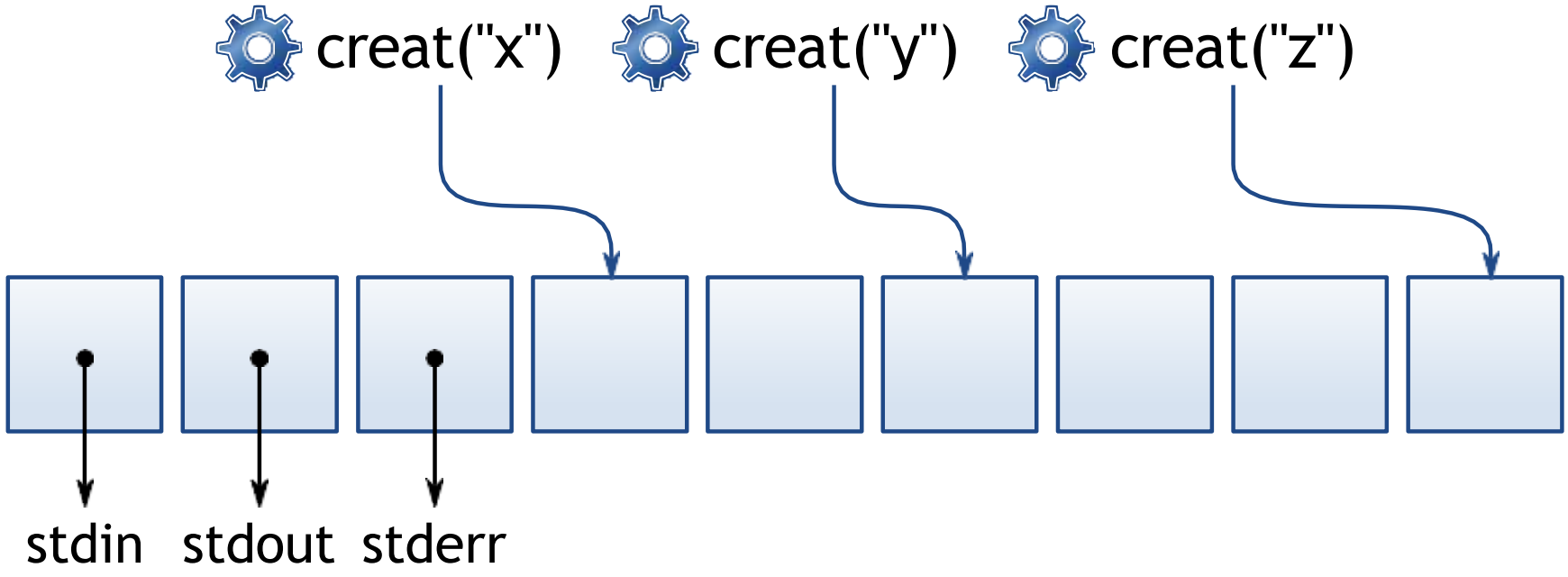
- Requires huge amounts of effort
- New workloads expose new bottlenecks
- More cores expose new bottlenecks
- The real bottlenecks may be in the interface design

Interface scalability example

Interface scalability example



Interface scalability example



Solution: Change the interface?

Approach: interface-driven scalability

The scalable commutativity rule

Whenever interface operations commute, they can be implemented in a way that scales.

Approach: interface-driven scalability

The scalable commutativity rule

Whenever interface operations commute, they can be implemented in a way that scales.

	Commutes	Scalable implementation exists
creat with lowest FD	?	

Approach: interface-driven scalability

The scalable commutativity rule

Whenever interface operations commute, they can be implemented in a way that scales.

	Commutes	Scalable implementation exists
creat with lowest FD	?	
	creat → 3 creat → 4	

Approach: interface-driven scalability

The scalable commutativity rule

Whenever interface operations commute, they can be implemented in a way that scales.

	Commutes	Scalable implementation exists
creat with lowest FD	✗	

Approach: interface-driven scalability

The scalable commutativity rule

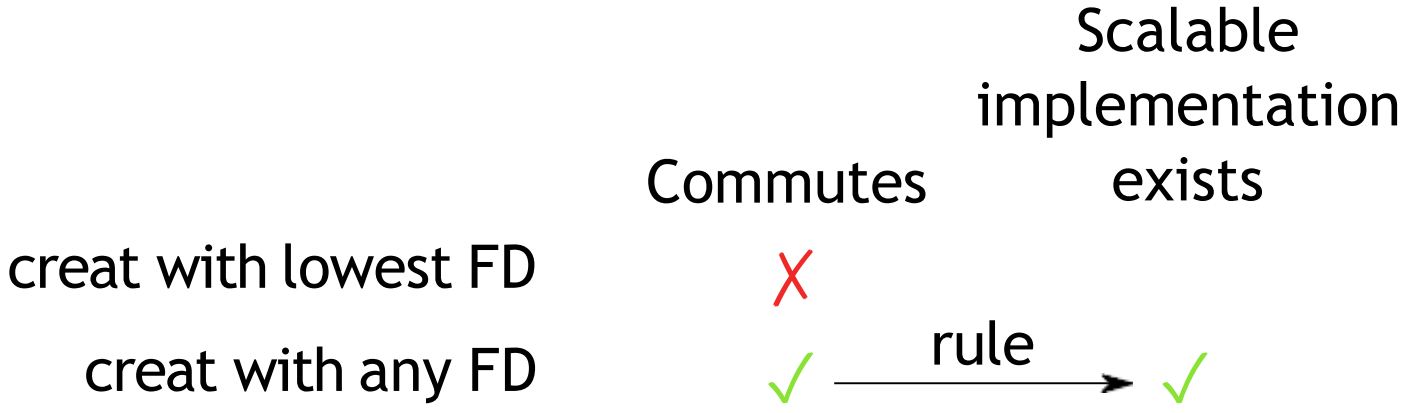
Whenever interface operations commute, they can be implemented in a way that scales.

	Commutes	Scalable implementation exists
creat with lowest FD	✗	
creat with any FD	?	
	creat → 42 creat → 17	

Approach: interface-driven scalability

The scalable commutativity rule

Whenever interface operations commute, they can be implemented in a way that scales.



Approach: interface-driven scalability

The scalable commutativity rule

Whenever interface operations commute*, they can be implemented in a way that scales.

*SIM Commutativity:

- State-Depended
- Interface-based
- Monotonic

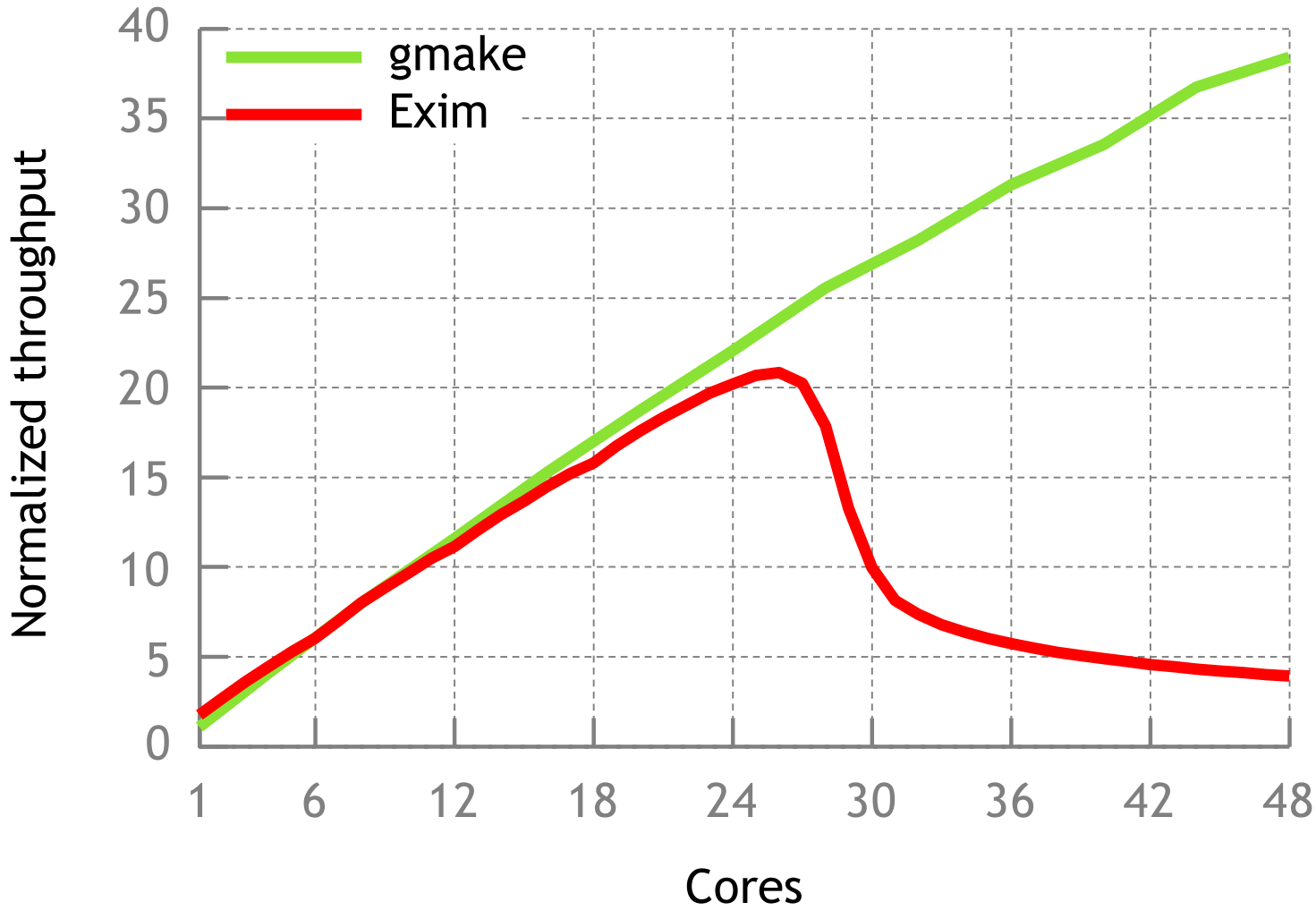
Approach: interface-driven scalability

The scalable commutativity rule

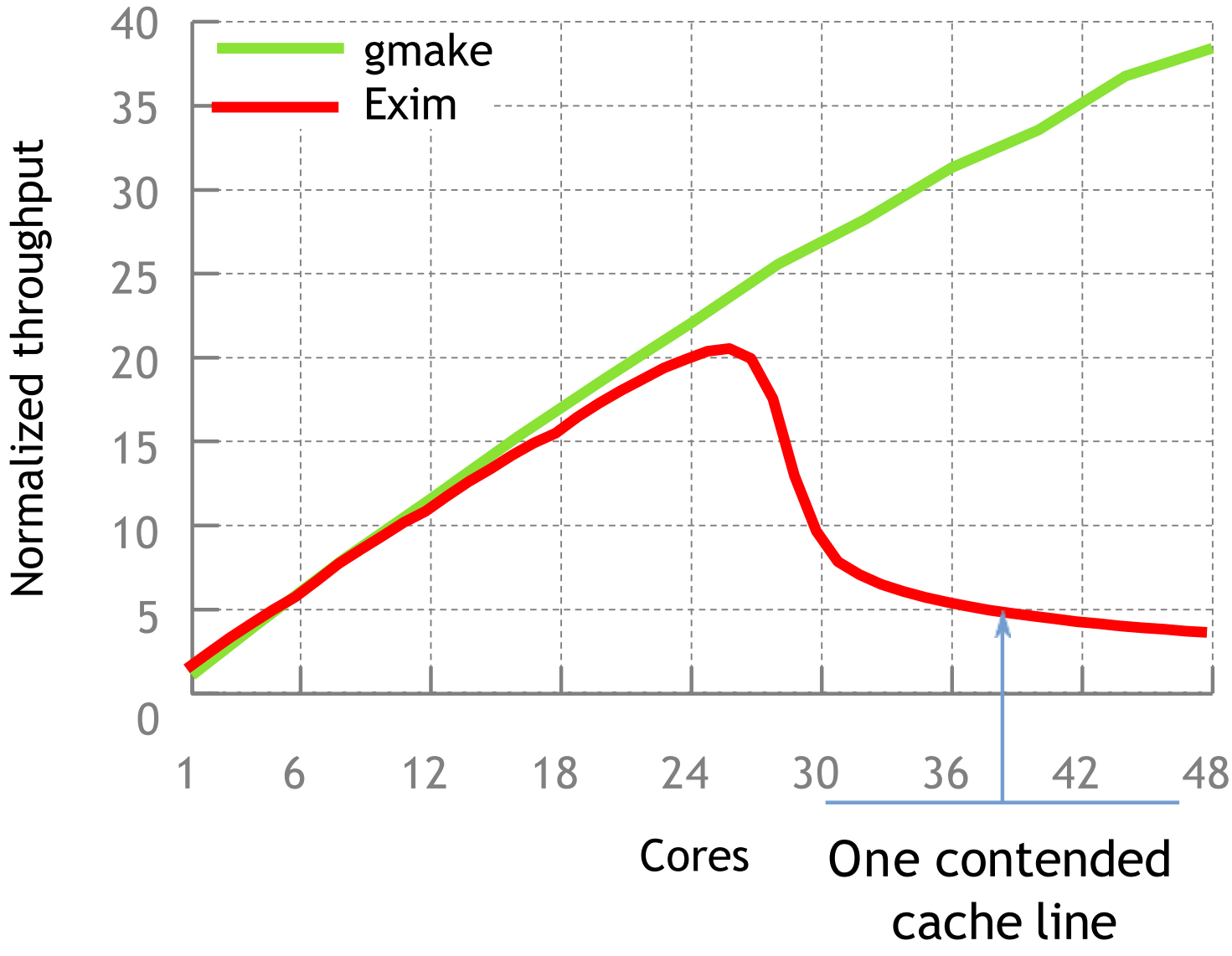
Whenever interface operations commute*, they can be implemented in a way that scales.

The rule lets developers reason about scalability during software design , even before an implementation exists or a workload can be run on it.

A Scalability bottleneck



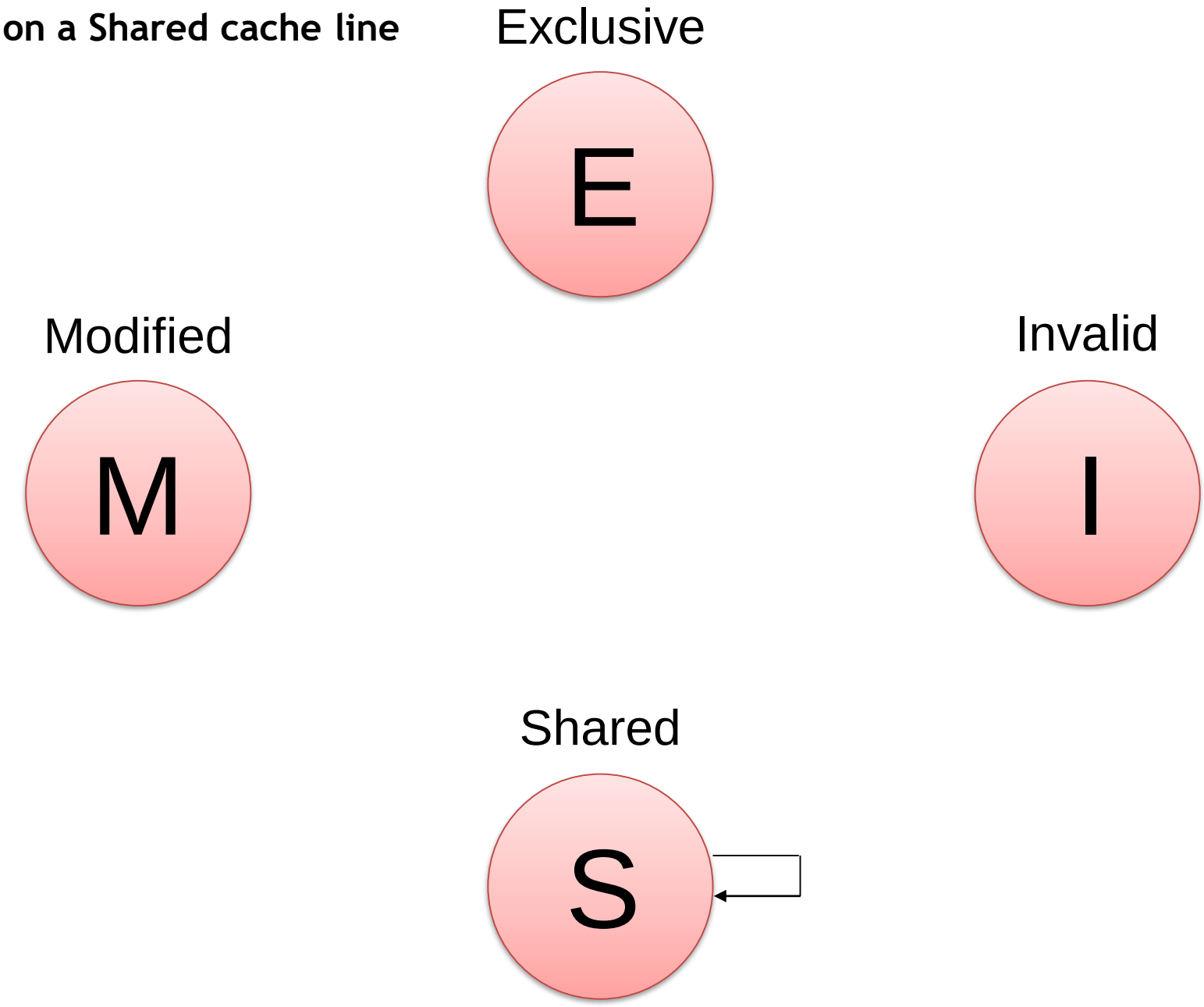
Approach: interface-driven scalability



A single contended cache line can wreck scalability

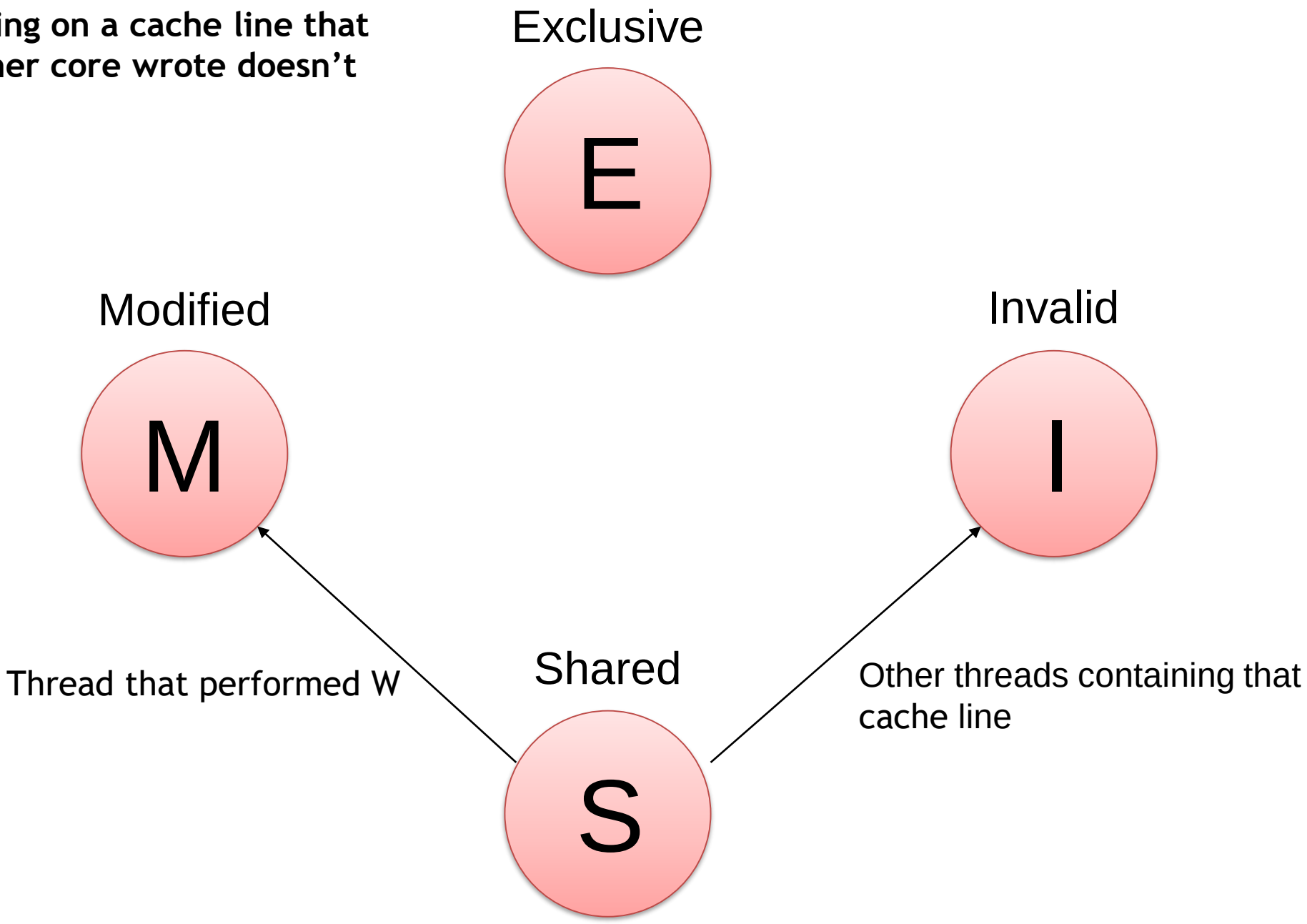
MESI-like cache coherence protocol

Reading on a Shared cache line scales



MESI-like cache coherence protocol

Reading on a cache line that another core wrote doesn't



What scales on today's multicores

Core *Y* \ Core *X*	W	R	-
W	✗	✗	✓
R	✗	✓	✓
-	✓	✓	-

We say two or more operations are *scalable* if they are *conflict-free*.

The intuition behind the rule

Whenever interface operations commute, they can be implemented in a way that scales.

Operations commute
⇒ results independent of order
⇒ communication is unnecessary
⇒ without communication, no conflicts

Outline

Introduction
- Presenting the rule
- Definition of scalability
- Intuition

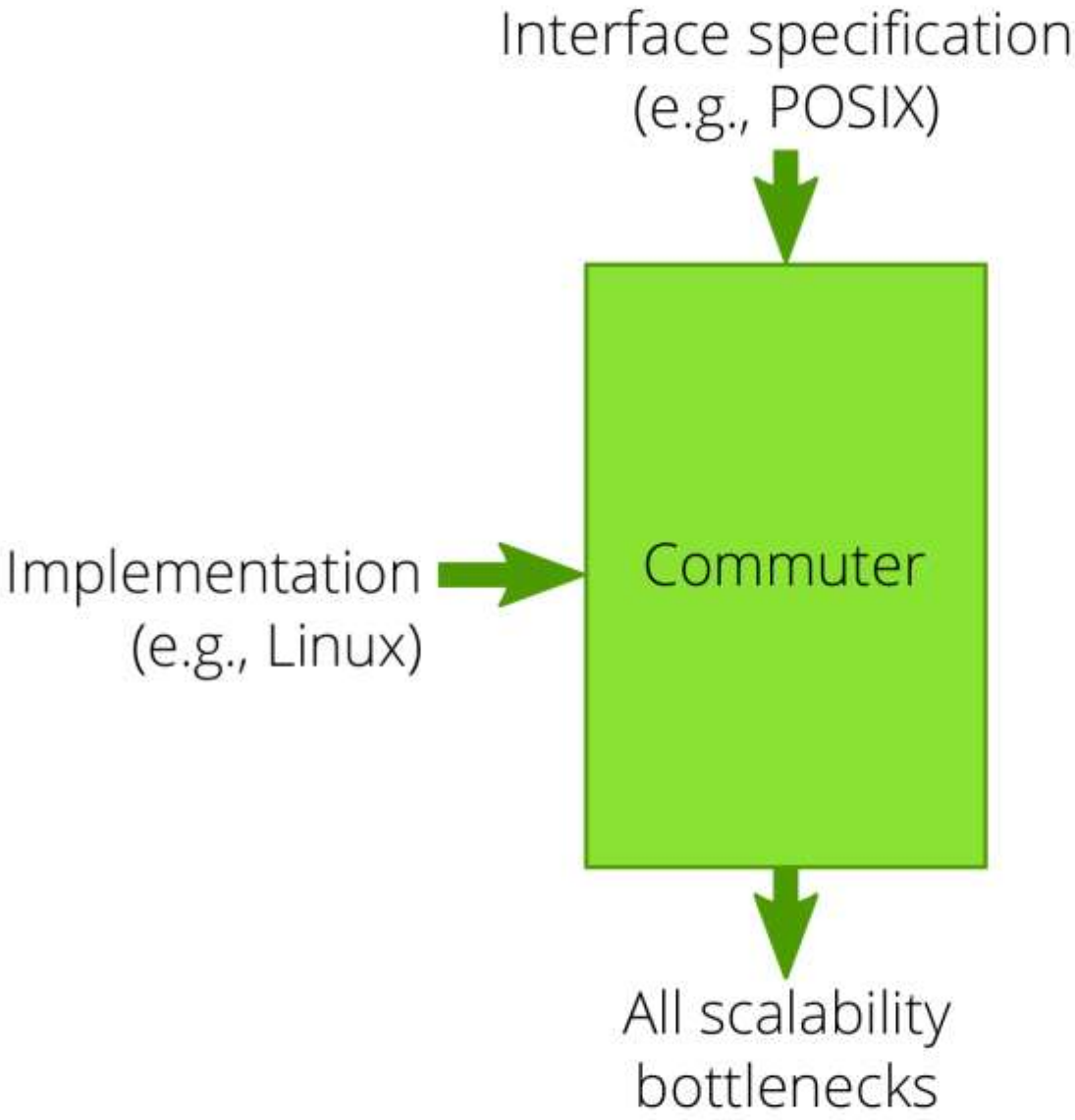
Applying the rule
- Commuter
- Evaluation

Formalizing the rule
- Definitions
- Proof

Applying the rule to real systems

Commuter

Applying the rule to real systems



Input: Symbolic model

```
SymInode    = tstruct(data  = tlist(SymByte),
                      nlink = SymInt)
SymIMap     = tdict(SymInt, SymInode)
SymFilename = tuninterpreted('Filename')
SymDir      = tdict(SymFilename, SymInt)

class POSIX:
  def __init__(self):
    self.fname_to_inum = SymDir.any()
    self.inodes = SymIMap.any()

  @symargs(src=SymFilename, dst=SymFilename)
  def rename(self, src, dst):
    if src not in self.fname_to_inum:
      return (-1, errno.ENOENT)
    if src == dst:
      return 0
    if dst in self.fname_to_inum:
      self.inodes[self.fname_to_inum[dst]].nlink -= 1
    self.fname_to_inum[dst] = self.fname_to_inum[src]
    del self.fname_to_inum[src]
    return 0
```

Commutativity conditions

```
@symargs(src=SymFilename, dst=SymFilename)
def rename(self, src, dst):
  if src not in self.fname_to_inum:
    return (-1, errno.ENOENT)
  if src == dst:
    return 0
  if dst in self.fname_to_inum:
    self.inodes[self.fname_to_inum[dst]].nlink -= 1
  self.fname_to_inum[dst] = self.fname_to_inum[src]
  del self.fname_to_inum[src]
  return 0
```

rename(a, b) and rename(c, d) commute if:

- Both source files exist and all names are different
- Neither source file exists
- a xor c exists, and it is not the other rename's destination
- Both calls are self-renames
- One call is a self-rename of an existing file and a ≠ c
- a and c are hard links to the same inode, a ≠ c, and b = d

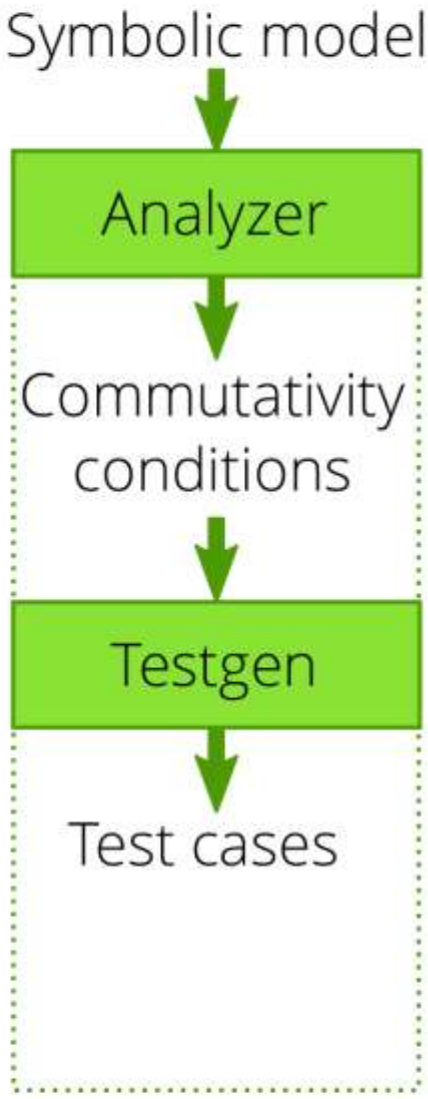
Test cases

rename(a, b) and rename(c, d) commute if:

- Both source files exist and all names are different
- Neither source file exists
- a xor c exists, and it is not the other rename's destination
- Both calls are self-renames
- One call is a self-rename of an existing file and a ≠ c
- a and c are hard links to the same inode, a ≠ c, and b = d

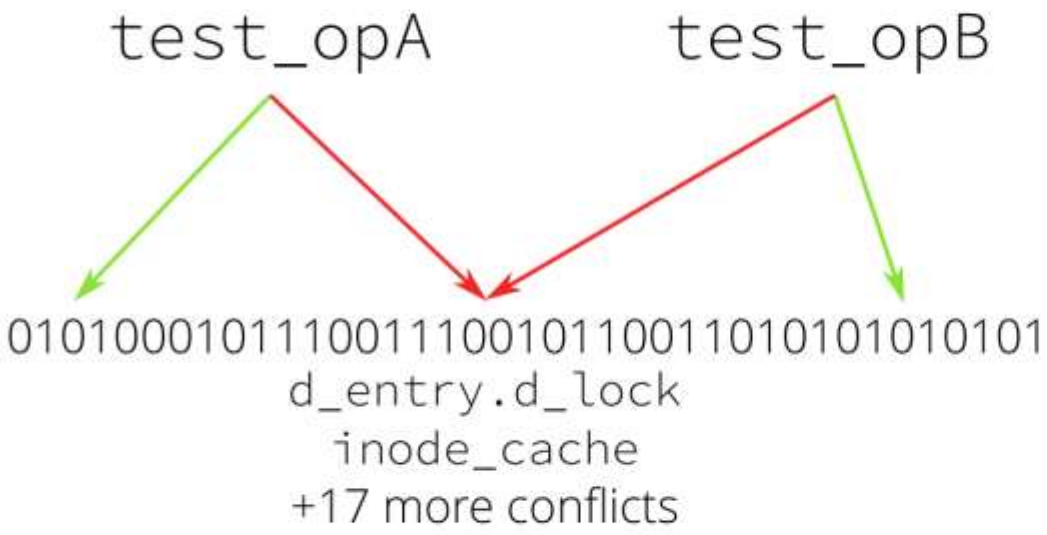
```
void setup() {
    close(creat("f0", 0666));
    close(creat("f2", 0666));
}
void test_opA() { rename("f0", "f1"); }
void test_opB() { rename("f2", "f3"); }
```

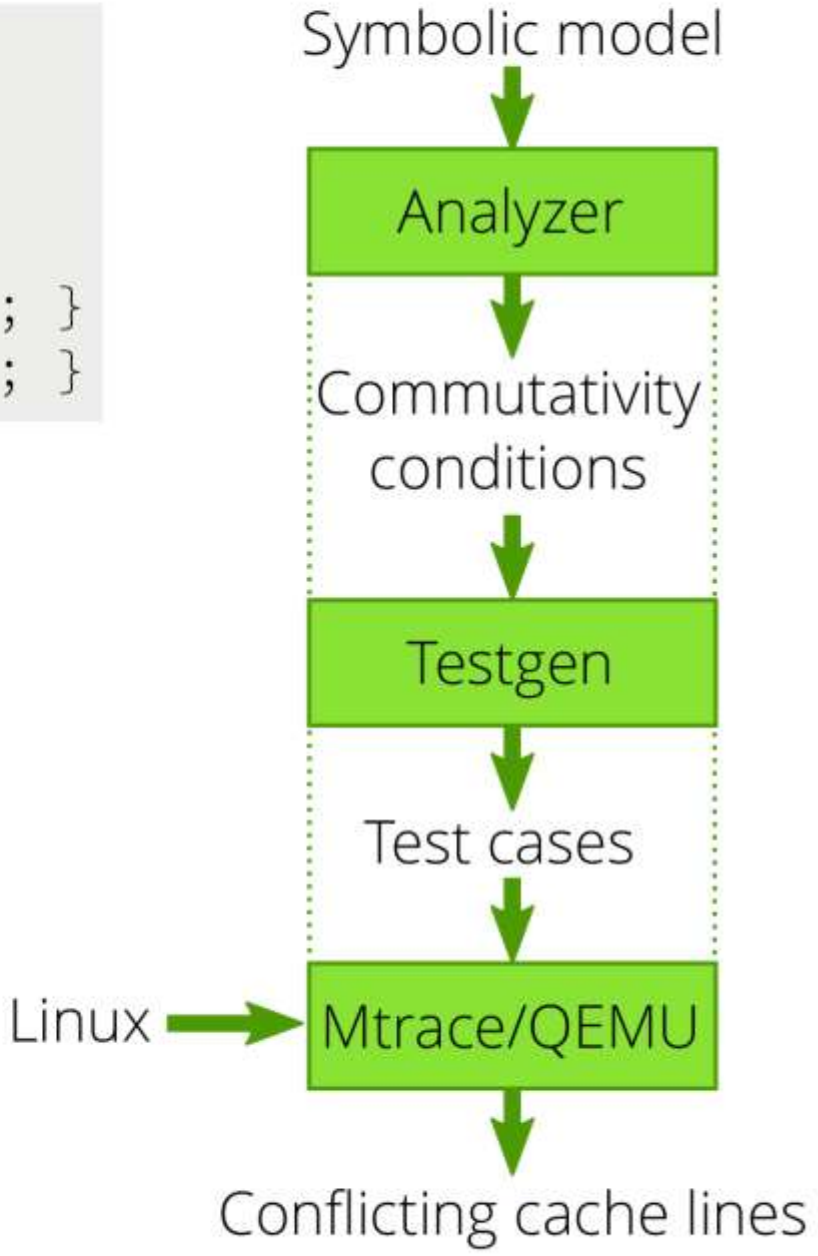
+ 26 more



Output: Conflicting cache lines

```
void setup() {
    close(creat("f0", 0666));
    close(creat("f2", 0666));
}
void test_opA() { rename("f0", "f1"); }
void test_opB() { rename("f2", "f3"); }
```

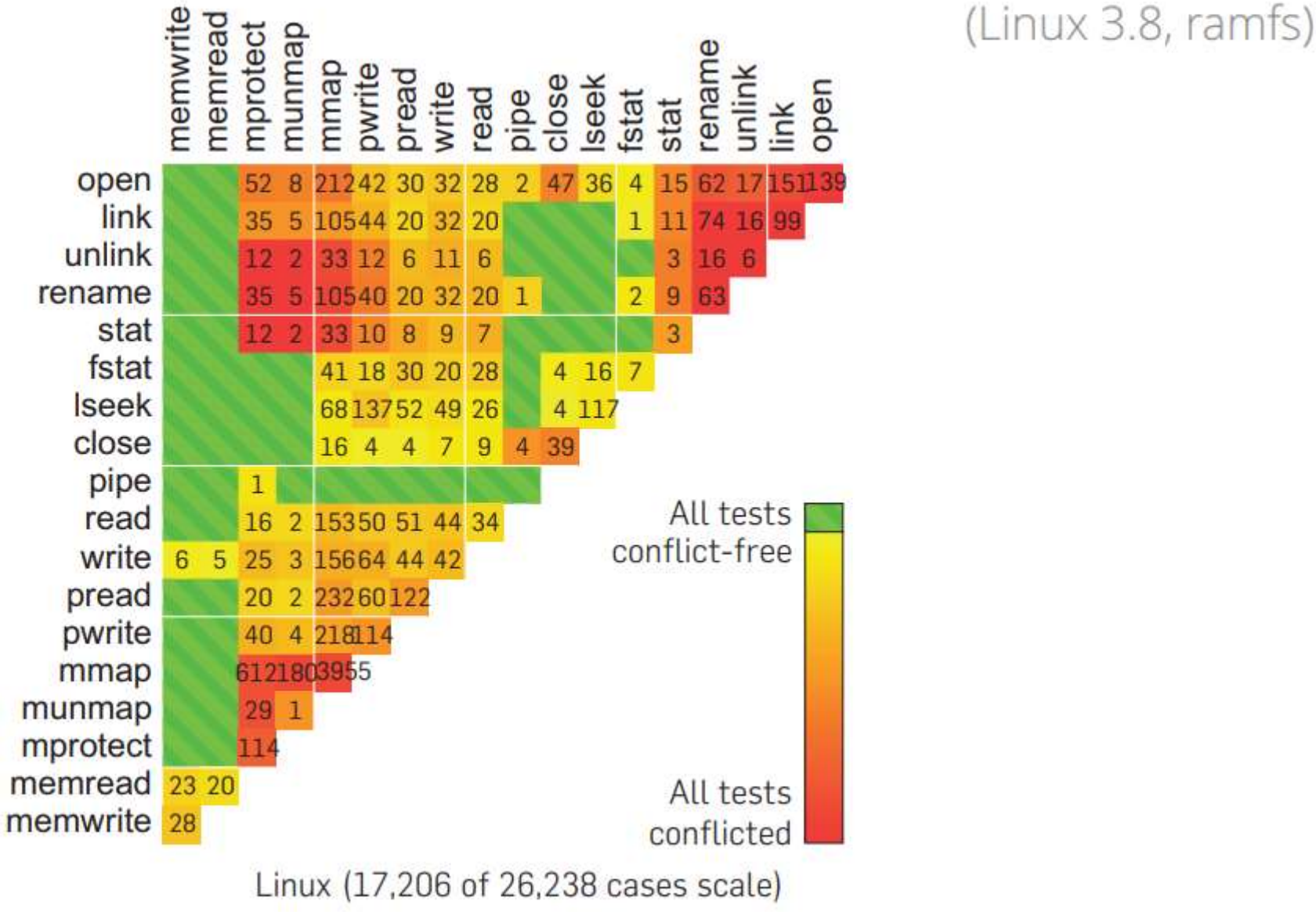




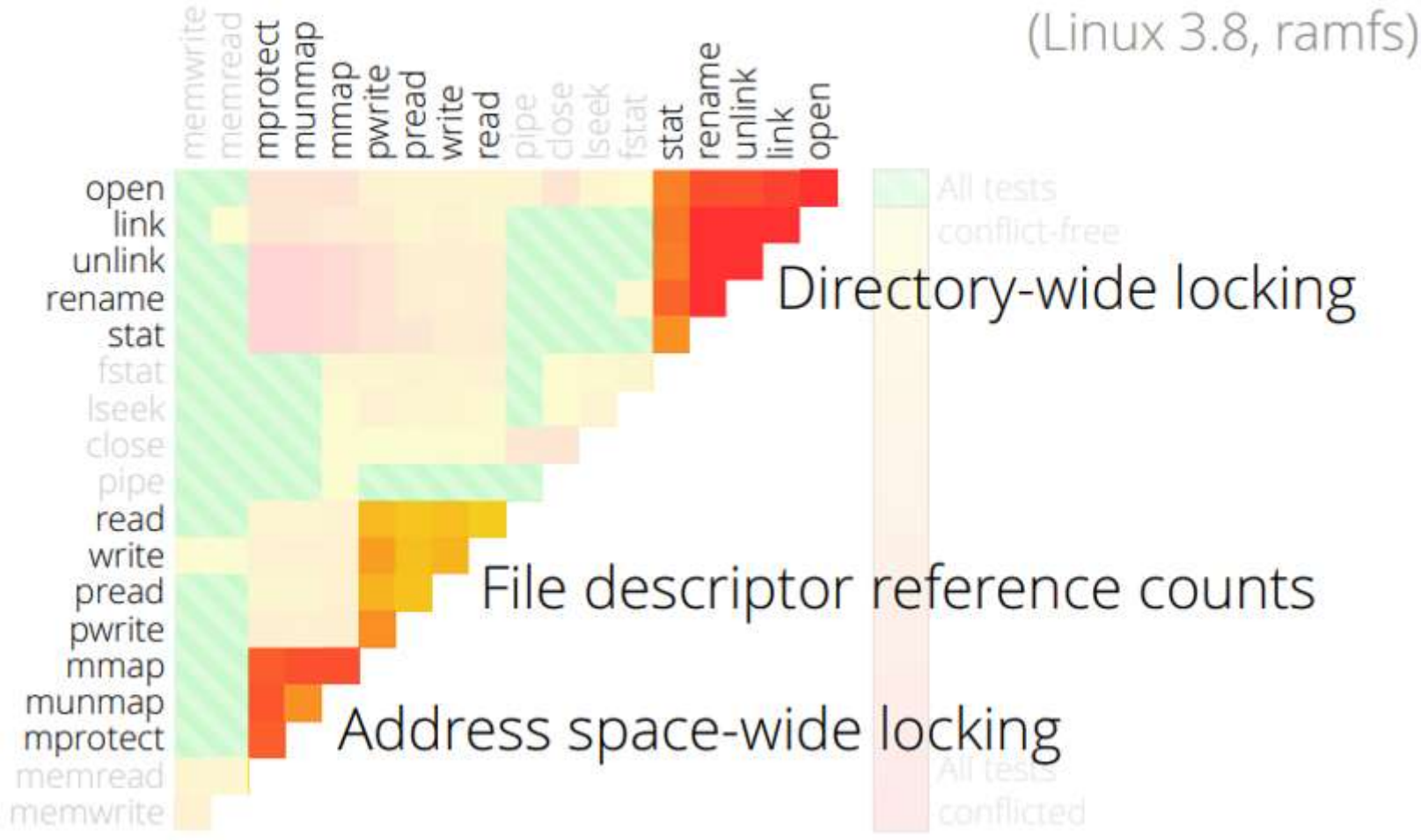
Evaluation

Does the rule help build scalable systems?

Commuter finds non-scalable cases in Linux



Commuter finds non-scalable cases in Linux



sv6 : A scalable OS

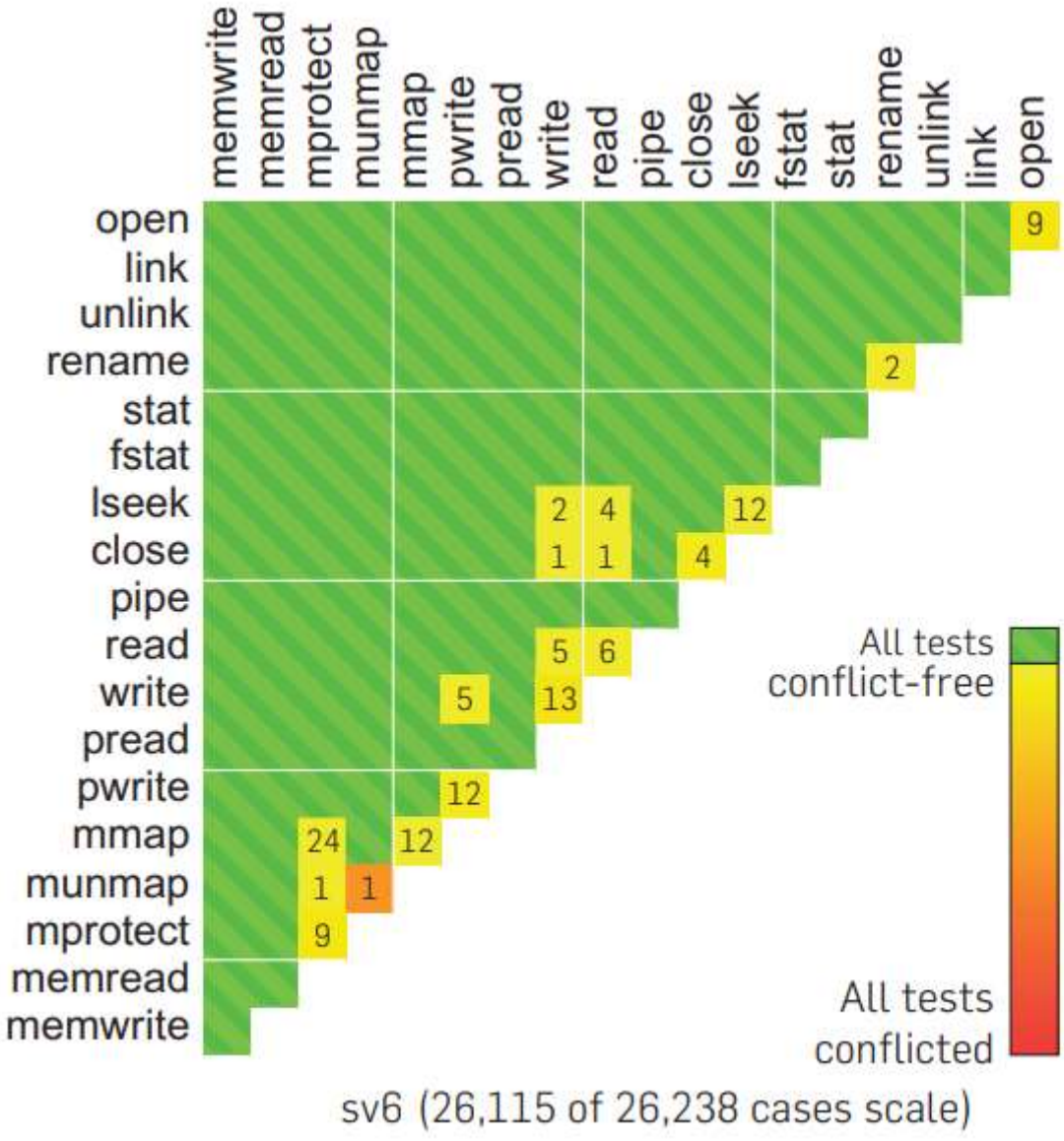
POSIX-like operating system

File system and virtual memory system follow commutativity rule (ScaleFS , RadixVM)

Implementation using standard parallel programming techniques, but guided by Commuter

- Scalable data structures (linear arrays, radix arrays, hash tables)
- Postponing resource deallocation (Refcache)
- Optimistic checks and pessimistic updates before operations
- Avoiding unnecessary reads

Commutative operations can be made to scale



sv6 (26,115 of 26,238 cases scale)

Refining POSIX with the rule

Avoid unnecessary return-value dependencies
- Lowest FD versus any FD
- stat versus xstat

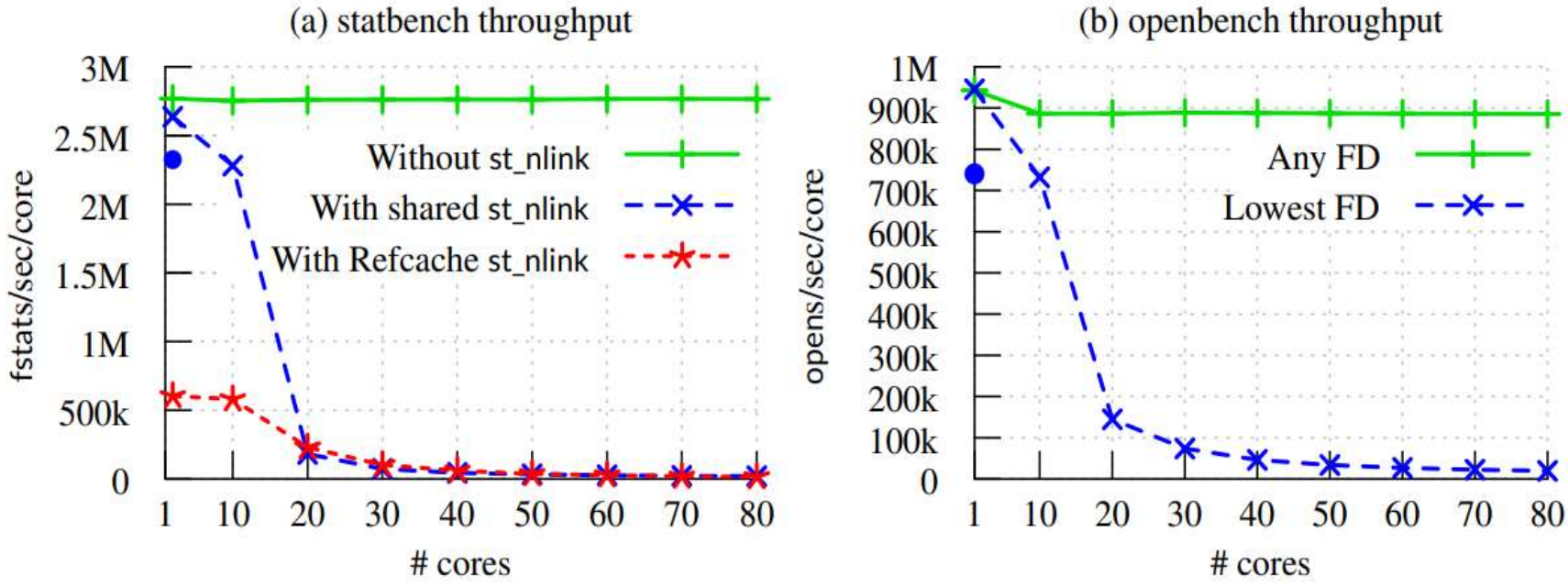
Permit weak ordering
- Unordered sockets

Permit asynchronous / delayed effects
- Delayed munmap

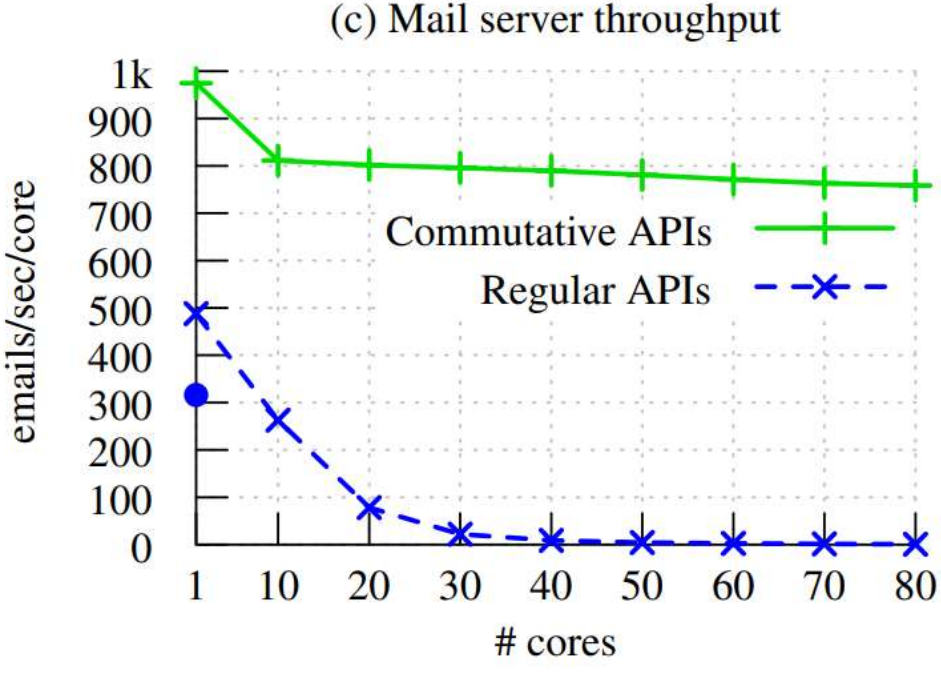
Decomposing compound operations.
- fork+exec versus posix_spawn

Evaluation

Microbenchmarks:



Custom mail server application:



Outline

Introduction
- Presenting the rule
- Definition of scalability
- Intuition

Applying the rule
- Commuter
- Evaluation

Formalizing the rule
- Definitions
- Proof

Example: reference counter

T1 iszero() → F

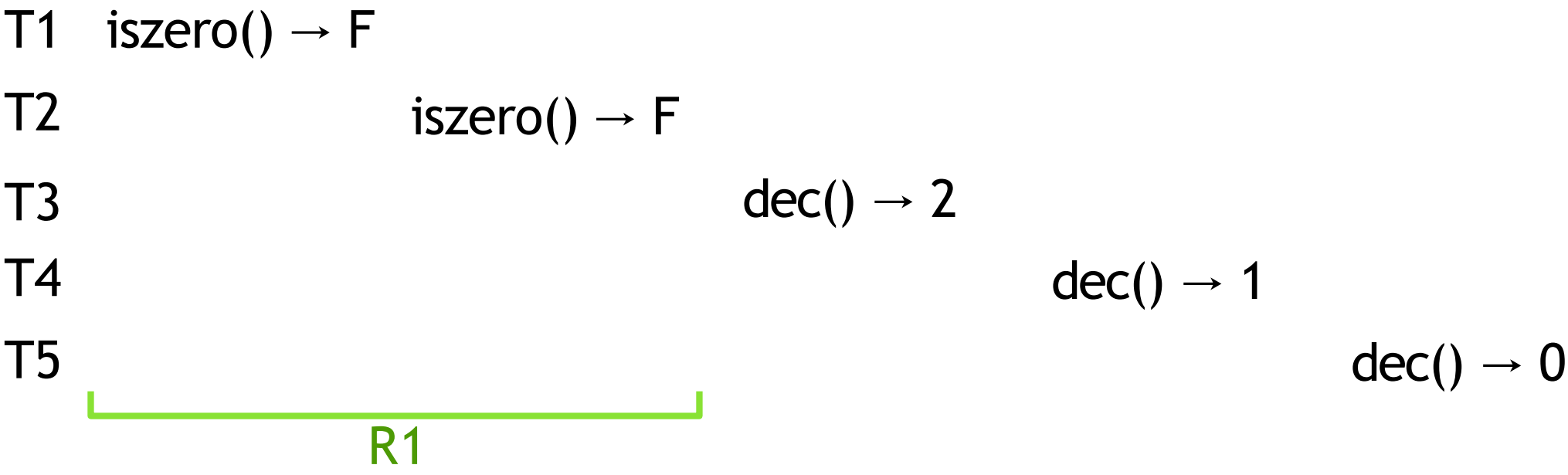
T2 iszero() → F

T3 dec() → 2

T4 dec() → 1

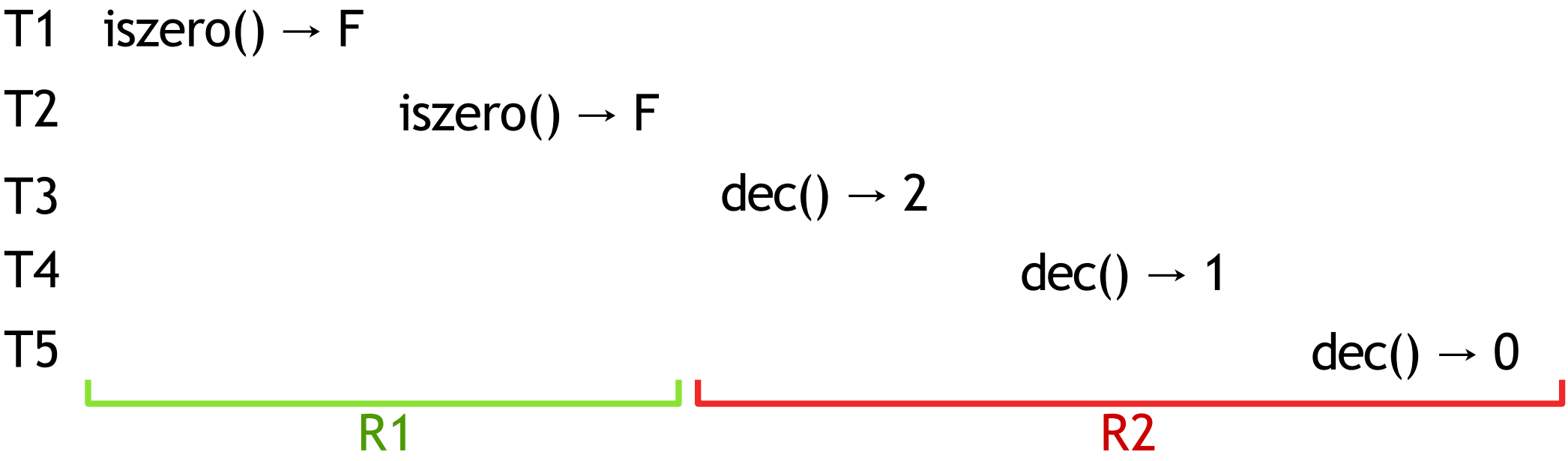
T5 dec() → 0

Example: reference counter



✓ R1 commutes; conflict-free implementation: shared counter

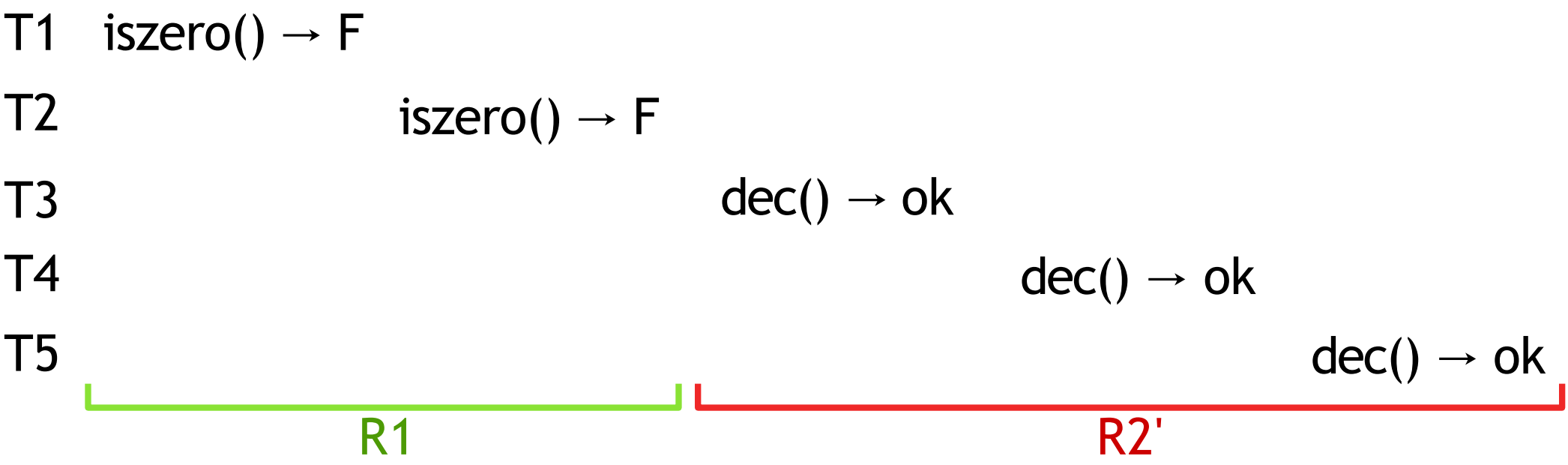
Example: reference counter



✓ R1 commutes; conflict-free implementation: shared counter

✗ R2 does not commute because dec() returns counter value

Example: reference counter

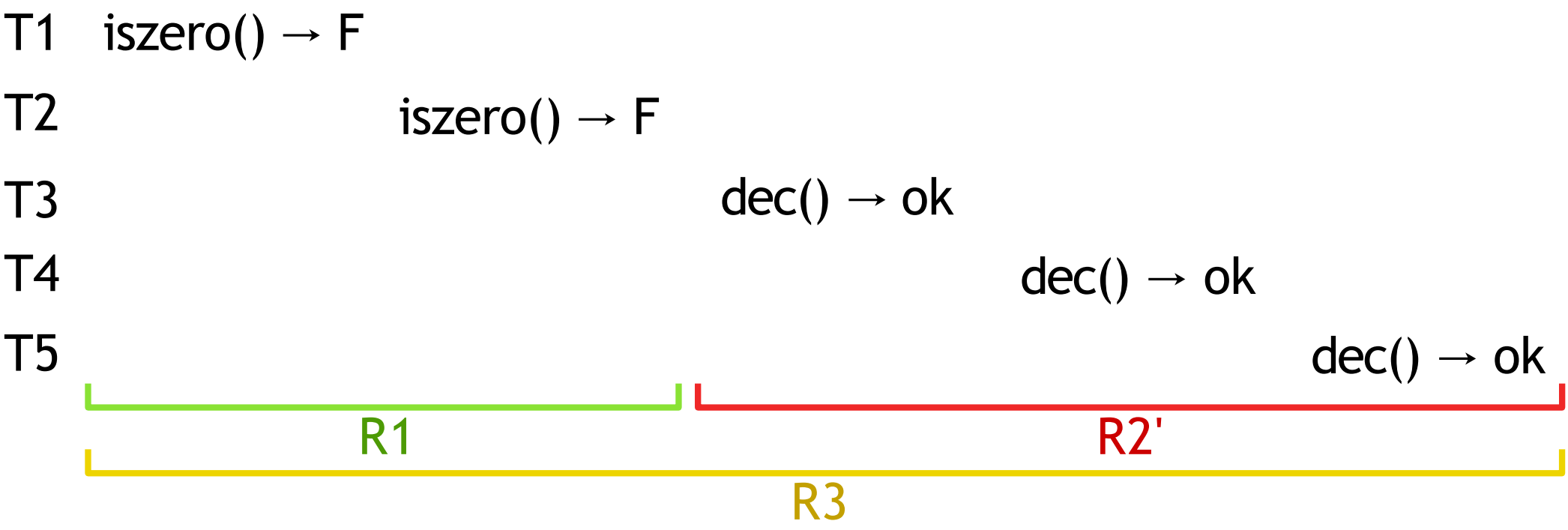


✓ R1 commutes; conflict-free implementation: shared counter

✗ R2 does not commute because dec() returns counter value

✓ R2' does commute; conflict-free implementation: per-core counter

Example: reference counter



✓ R1 commutes; conflict-free implementation: shared counter

✗ R2 does not commute because dec() returns counter value

✓ R2' does commute; conflict-free implementation: per-core counter

R3 depends on state

✓ Initial value > 3 ✗ Initial value ≤ 3

Formalizing the rule

Definitions :

- Actions
- History
- Specification
- Reordering
- SI/SIM Commutativity
- Implementation
- Conflict freedom

Formal scalable commutativity rule

Definitions

Actions:
Actions are pairs of an invocation and a response

Definitions

Actions:
Actions are pairs of an invocation and a response

History:
A sequence of actions on one or multiple threads.

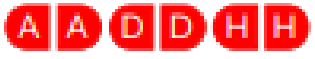
Definitions

Actions:
Actions are pairs of an invocation and a response

History:
A sequence of actions on one or multiple threads.

$H =$ 

Thread-restricted sub-history $\mathrm{H}|\mathrm{t}$:
The actions of a specific thread t in a history

$H|t =$ 

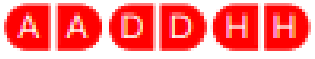
Definitions

Actions:
Actions are pairs of an invocation and a response

History:
A sequence of actions on one or multiple threads.

$H =$ 

Thread-restricted sub-history $\mathrm{H}|\mathrm{t}$:
The actions of a specific thread t in a history

$H|_t =$ 

Specification $\mathcal{S}$:
A **specification** is the set of **legal** histories
giving the allowed behavior of an interface.
(e.g $[\mathrm{A} = \mathrm{getpid}(), \mathrm{A} = 0] \notin \mathcal{S}$)

Definitions

Reordering:
H′ is a reordering of a history H when H|t = H′|t for every thread.

Definitions

Reordering:

H′ is a reordering of a history H when H|t = H′|t for every thread. *e.g*

H = 

H′ = 

Valid Reordering

H′ = 

Invalid Reordering

Definitions

Reordering:

H′ is a reordering of a history H when H|t = H′|t for every thread. *e.g*

H = 

H′ = 

Valid Reordering

H′ = 

Invalid Reordering

SI-Commutativity:

H = X || Y || Z , X and Y are regions of H

Y **SI-commutes** in H when given any reordering Y′ of Y, and any action sequence Z,

$X \,||\, Y \,||\, Z \in \mathcal{S}$ *iff* $X \,||\, Y' \,||\, Z \in \mathcal{S}$.

Definitions

Reordering:

H′ is a reordering of a history H when H|t = H′|t for every thread. *e.g*

H = 

H′ = 

Valid Reordering

H′ = 

Invalid Reordering

SI-Commutativity:

H = X || Y || Z , X and Y are regions of H

Y **SI-commutes** in H when given any reordering Y′ of Y, and any action sequence Z,

$X \,||\, Y \,||\, Z \in \mathcal{S}$ *iff* $X \,||\, Y' \,||\, Z \in \mathcal{S}$.

$Y = [A = \text{set}(1), A, B = \text{set}(2), B, C = \text{set}(2), C]$,

SI-commutes (in every order, value is set to 2)

$Y_1 = [A = \text{set}(1), A, B = \text{set}(2), B]$,

Does not commute (value can be set to 1 or 2)

Definitions

Reordering:

H′ is a reordering of a history H when H|t = H′|t for every thread. *e.g*

H = 

H′ = 

Valid Reordering

H′ = 

Invalid Reordering

SI-Commutativity:

H = X || Y || Z , X and Y are regions of H

Y **SI-commutes** in H when given any reordering Y′ of Y, and any action sequence Z,

$X \,||\, Y \,||\, Z \in \mathcal{S}$ *iff* $X \,||\, Y' \,||\, Z \in \mathcal{S}$.

SIM-Commutativity:

for any prefix P of some reordering of Y (including P = Y),

P SI-commutes in X || P

Definitions

Implementation:

Given an old state and an invocation, the implementation produces a new state and a response.

M(state , invocation) = *⟨new_state, response⟩*

- **States** are comprised of i **state components**
  [s.0, ..., s.i]

- **Conflict freedom** is achieved when a thread **does not** read/write a state component that a different thread has written.

The formal scalable commutativity rule

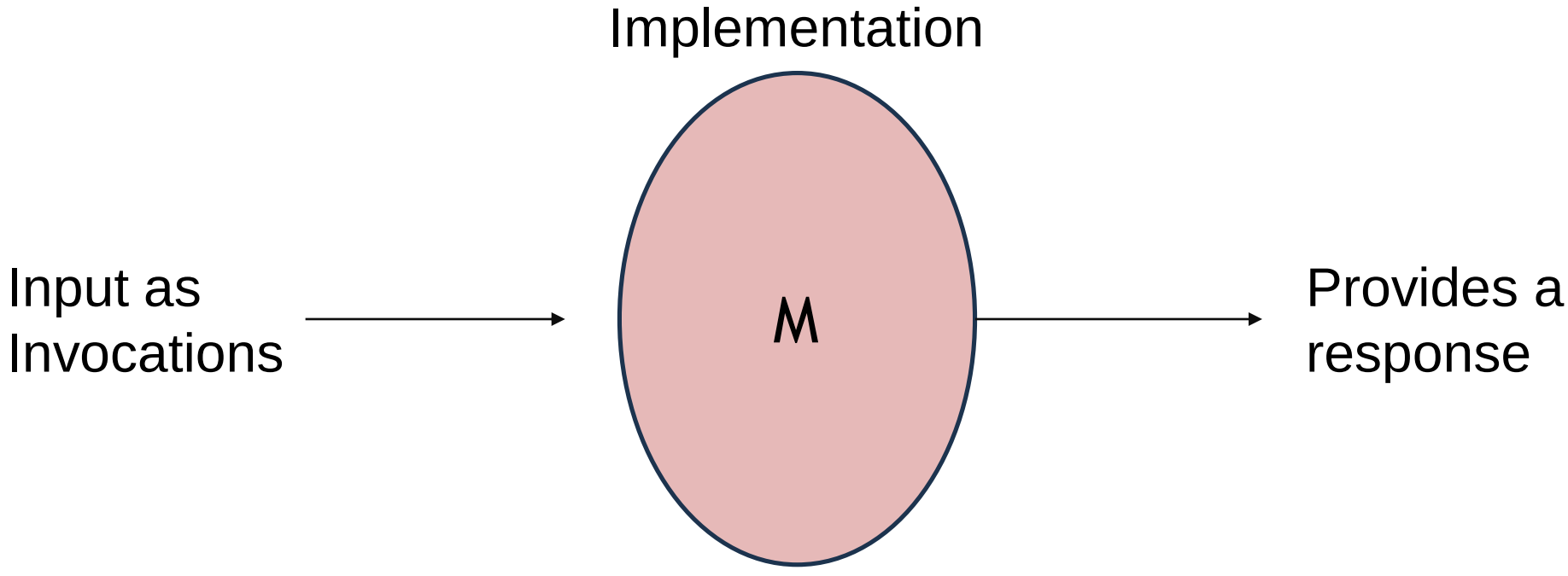
Let $\mathcal{S}$ be a specification with a reference implementation *M*. Consider a history *X* || *Y* where *Y* *SIM*-commutes in *X* || *Y* and *M* can generate *X* || *Y*.

There exists a correct implementation m of $\mathcal{S}$ whose execution of *X* || *Y* is conflict-free in the commutative region *Y*.

The formal scalable commutativity rule

Let $\mathcal{S}$ be a specification with a reference implementation *M*. Consider a history *X* || *Y* where *Y* *SIM*-commutes in *X* || *Y* and *M* can generate *X* || *Y*.

There exists a correct implementation m of $\mathcal{S}$ whose execution of *X* || *Y* is conflict-free in the commutative region *Y*.

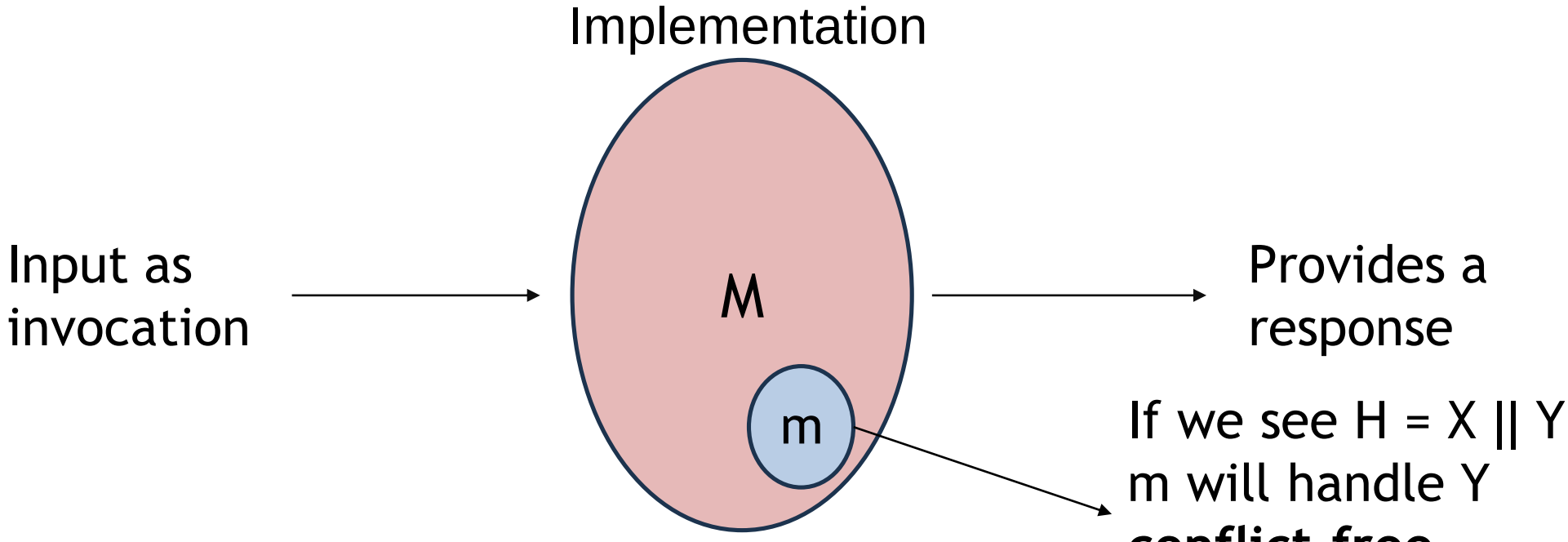


(Not so formal diagram !)

The formal scalable commutativity rule

Let $\mathcal{S}$ be a specification with a reference implementation *M*. Consider a history *X* || *Y* where *Y* *SIM*-commutes in *X* || *Y* and *M* can generate *X* || *Y*.

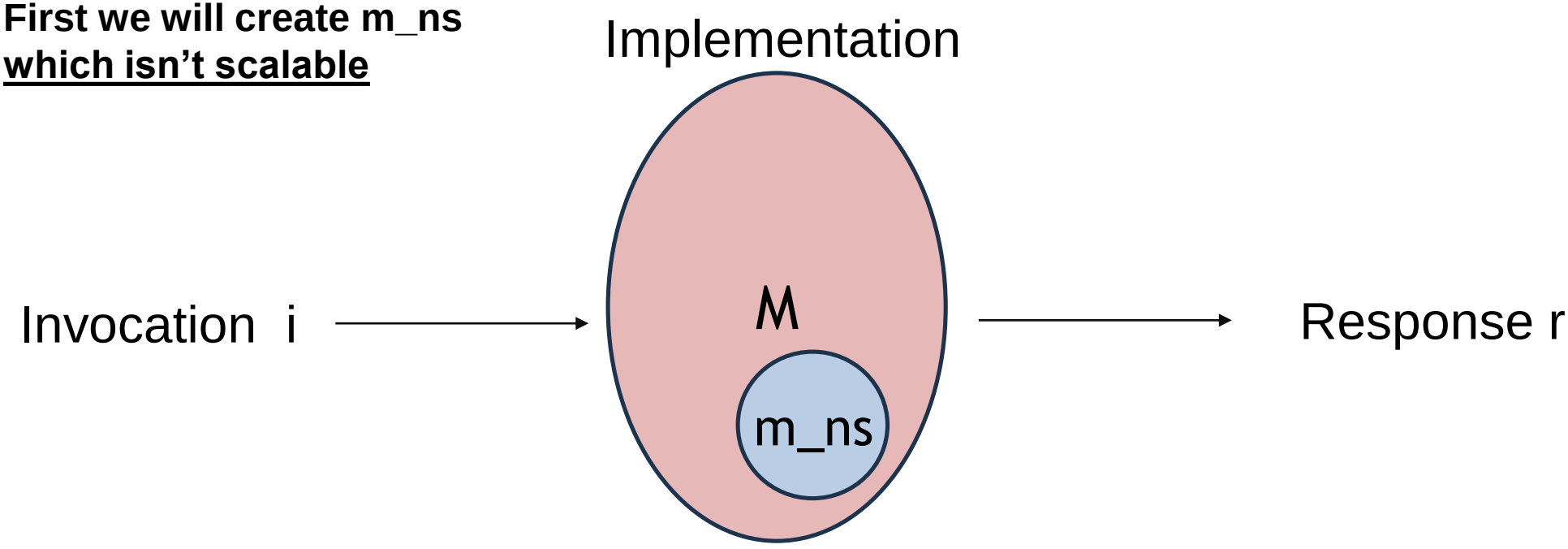
There exists a correct implementation m of $\mathcal{S}$ whose execution of *X* || *Y* is conflict-free in the commutative region *Y*.



(Not so formal diagram !)

non-scalable implementation m

First we will create m_ns
<u>which isn't scalable</u>

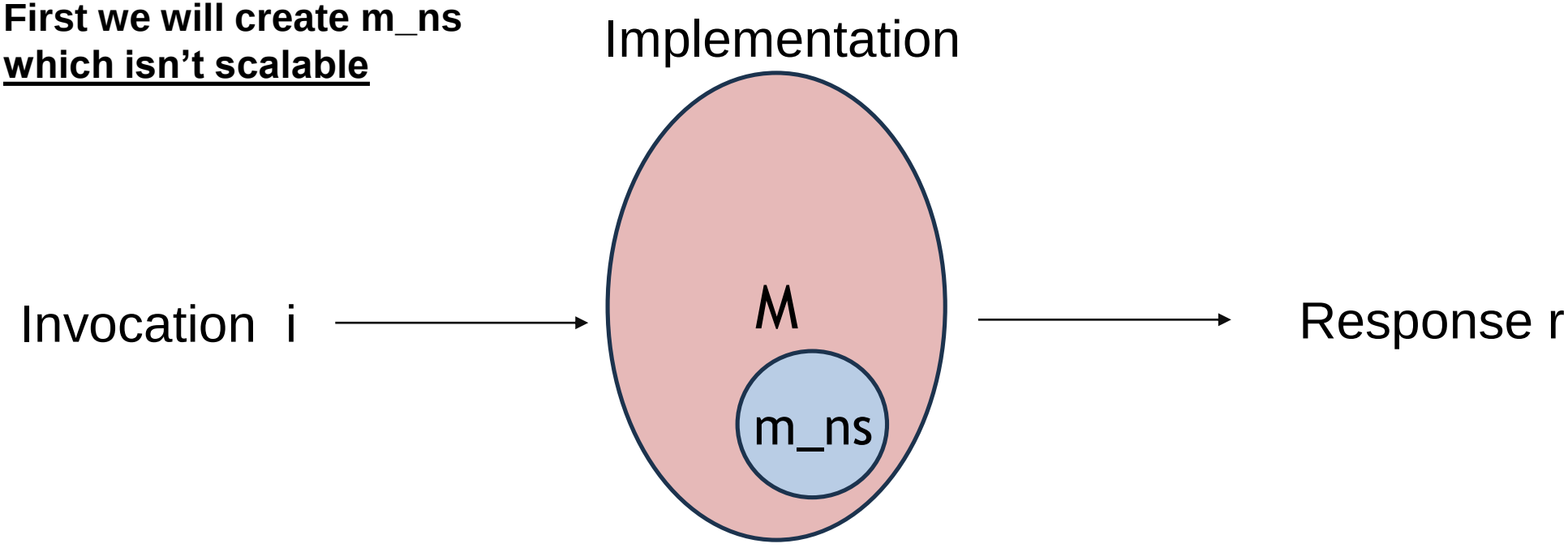


$$H = [\underbrace{\text{reset}(2)_3, \overline{\text{OK}}_3}_{X}, \underbrace{\text{isz}_1, \overline{\text{NZ}}_1, \text{isz}_2, \overline{\text{NZ}}_2, \text{dec}_3, \overline{\text{OK}}_3}_{Y}].$$

Y <u>SIM-Commutes</u> in H

non-scalable implementation m

First we will create m_ns which isn't scalable



$$H = [\underbrace{\text{reset}(2)_3, \overline{\text{OK}}_3}_{X}, \underbrace{\text{isz}_1, \overline{\text{NZ}}_1, \text{isz}_2, \overline{\text{NZ}}_2, \text{dec}_3, \overline{\text{OK}}_3}_{Y}].$$

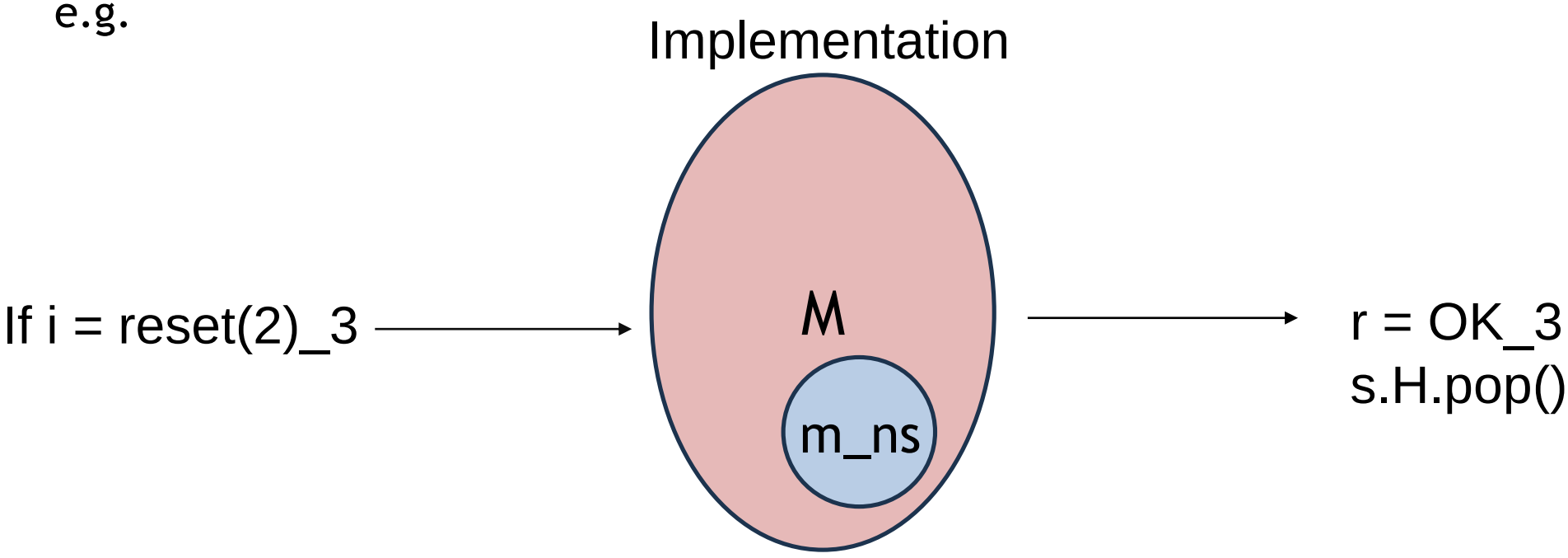
Y SIM-Commutes in H

m stores the history as a "queue" s.H of pairs of **invocations** and **pre-recorded responses.**

Since its pre-recorded, it will be the only state component that we will access.

Replay mode

e.g.

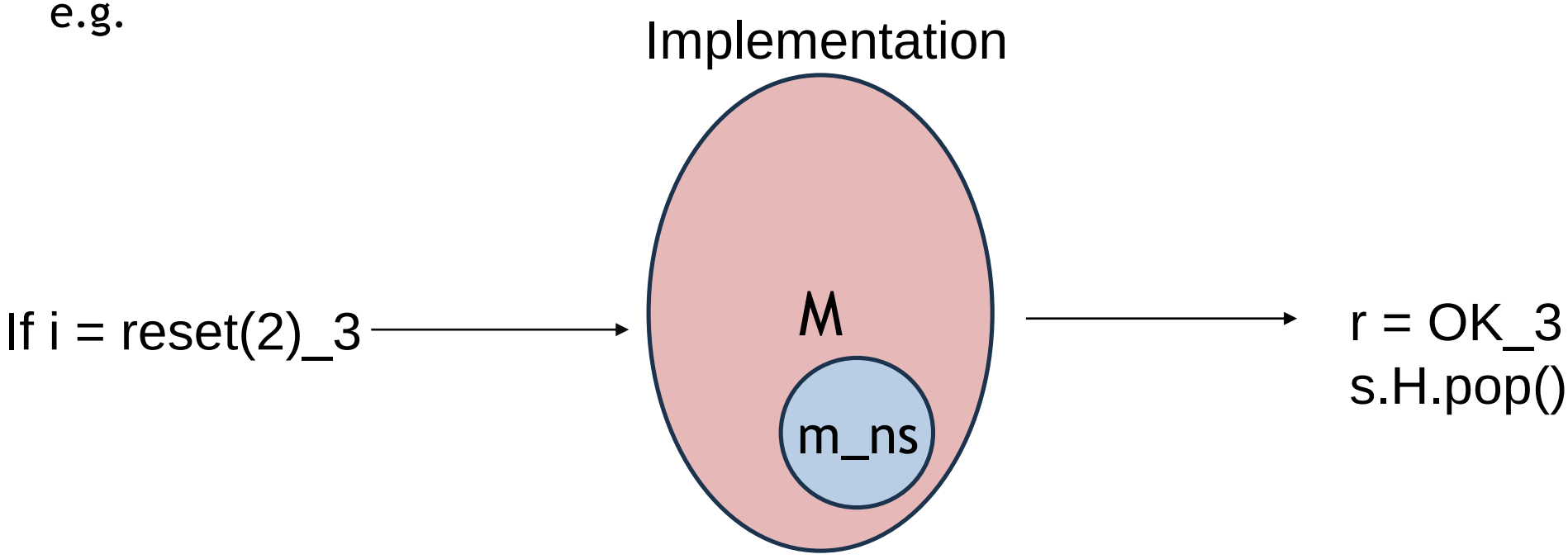


If the invocation <u>matches</u> the invocation in H, m returns the matching stored response.

$$H = [\underbrace{\text{reset}(2)_3, \overline{\text{OK}}_3}_{X}, \underbrace{\text{isz}_1, \overline{\text{NZ}}_1, \text{isz}_2, \overline{\text{NZ}}_2, \text{dec}_3, \overline{\text{OK}}_3}_{Y}].$$

Replay mode

e.g.

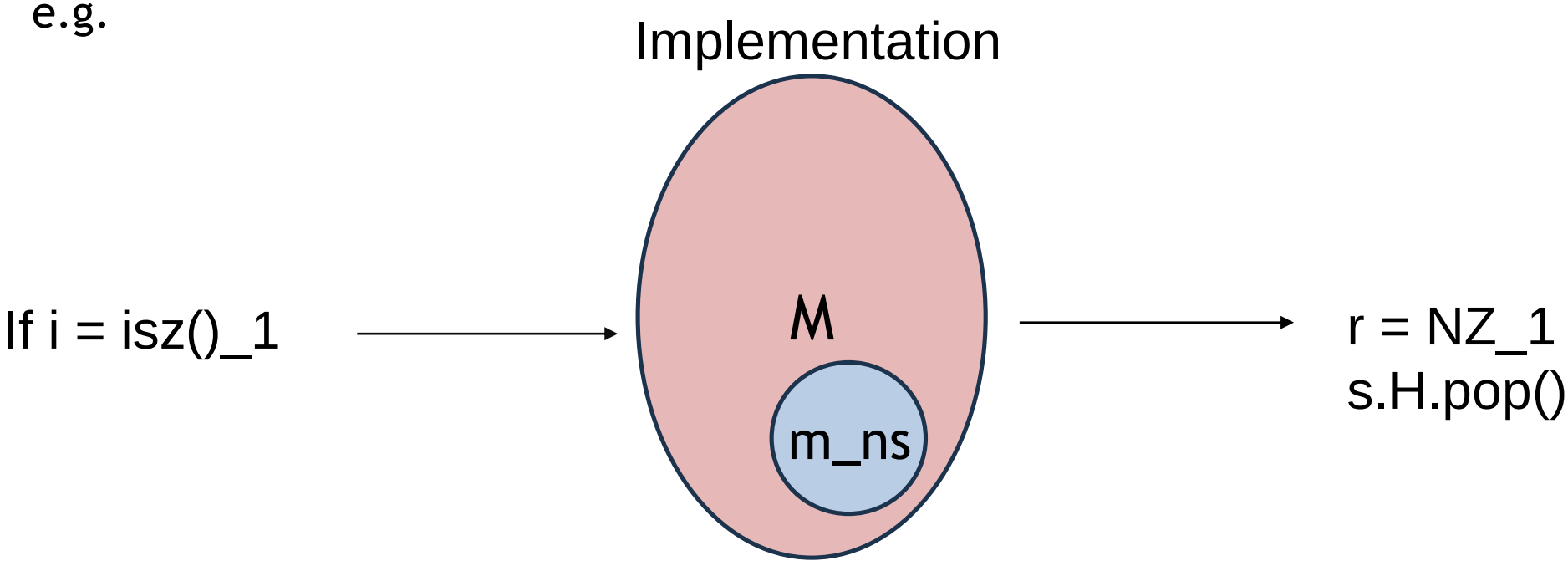


If the invocation <u>matches</u> the invocation in H, m returns the matching stored response.

s.H = $[\underbrace{isz_1, \overline{NZ}_1, isz_2, \overline{NZ}_2, dec_3, \overline{OK}_3}_{Y}]$.

Replay mode

e.g.



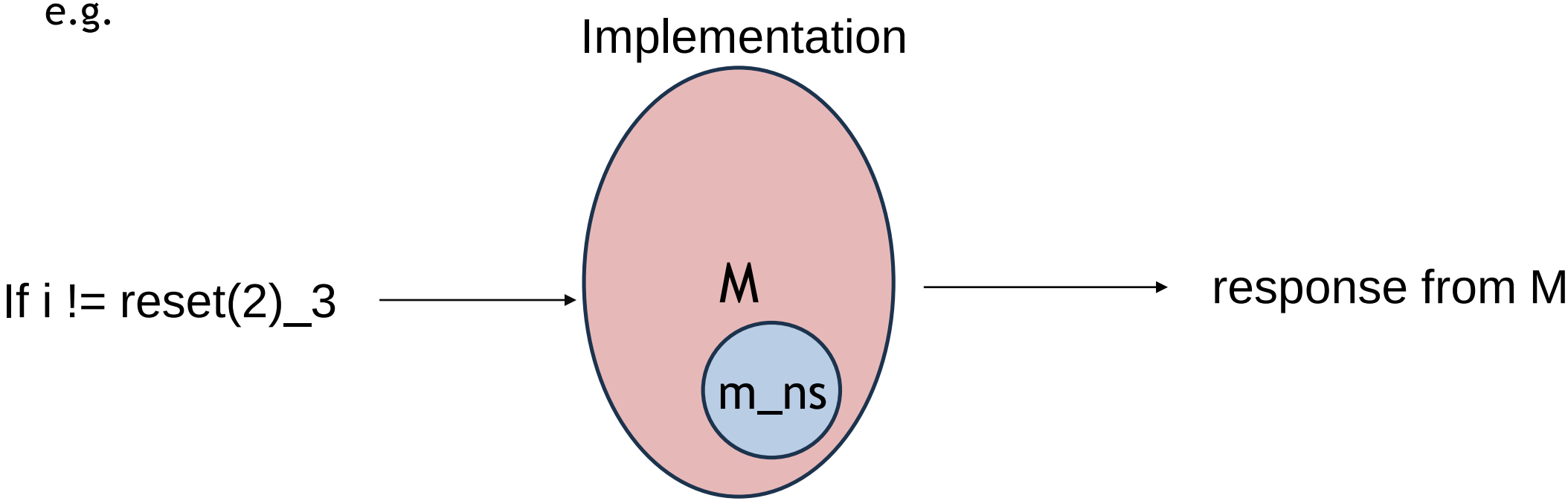
If the invocation <u>matches</u> the invocation in H, m returns the matching stored response.

...etc

s.H = $[\underbrace{isz_1, \overline{NZ}_1, isz_2, \overline{NZ}_2, dec_3, \overline{OK}_3}]$.

Emulation mode

e.g.



If the invocation <u>does not</u> match the expected invocation of H, m can no longer respond correctly.
So M takes over.

s.H = $[\overline{\text{reset}(2)_3}, \overline{\text{OK}_3}, \text{isz}_1, \overline{\text{NZ}_1}, \text{isz}_2, \overline{\text{NZ}_2}, \text{dec}_3, \overline{\text{OK}_3}]$.

X Y

Conflict-free implementation m

To make m_ns conflict-free we will:

Store s.H as different state components per-thread:

s.H[t] = X || COMMUTE ||(Y | t)

(COMMUTE is a flag that indicates we have reached Y)

Conflict-free implementation m

To make m_ns conflict-free we will:

Store s.H as different state components per-thread:

$s.H[t] = X \,||\, COMMUTE \,||\, (Y \mid t)$

(COMMUTE is a flag that indicates we have reached Y)

So

$$H = [\underbrace{\text{reset}(2)_3, \overline{\text{OK}}_3}_{X}, \underbrace{\text{isz}_1, \overline{\text{NZ}}_1, \text{isz}_2, \overline{\text{NZ}}_2, \text{dec}_3, \overline{\text{OK}}_3}_{Y}].$$

Will become:

$$s.H[1] = [\text{reset}(2)_3, \overline{\text{OK}}_3, \text{COMMUTE}, \text{isz}_1, \overline{\text{NZ}}_1].$$

$$s.H[2] = [\text{reset}(2)_3, \overline{\text{OK}}_3, \text{COMMUTE}, \text{isz}_2, \overline{\text{NZ}}_2].$$

$$s.H[3] = [\underbrace{\text{reset}(2)_3, \overline{\text{OK}}_3}_{X}, \text{COMMUTE}, \underbrace{\text{dec}_3, \overline{\text{OK}}_3}_{Y \mid t}].$$

Conflict-free mode

Idea is:

After we reach COMMUTE
For an invocation i from **thread** t:

If i ***matches invocation with*** head(s.H[t]) **:**
    respond with r
    pop(head(s.H[t]))
else:
    emulation mode

Conflict-free mode

Idea is:

After we reach COMMUTE
For an invocation i from **thread** t:

If i ***matches invocation with*** head(s.H[t]) **:**
    respond with r
    pop(head(s.H[t]))
else:
    emulation mode

s.H[1] = $[\text{reset}(2)_3, \overline{\text{OK}}_3, \text{COMMUTE}, \text{isz}_1, \overline{\text{NZ}}_1]$.

s.H[2] = $[\text{reset}(2)_3, \overline{\text{OK}}_3, \text{COMMUTE}, \text{isz}_2, \overline{\text{NZ}}_2]$.

s.H[3] = $[\text{reset}(2)_3, \overline{\text{OK}}_3, \text{COMMUTE}, \text{dec}_3, \overline{\text{OK}}_3]$.

Each thread only accesses its **own unique state components,**
therefore conflict free.
Since it **SIM-Commutes** , order doesn't matter

Conflict-free mode

After we reach COMMUTE
For an invocation i from **thread** t:

If i ***matches invocation with*** head(s.H[t]) **:**
    respond with pre-recorded r
    pop(head(s.H[t]))
else:
    emulation mode

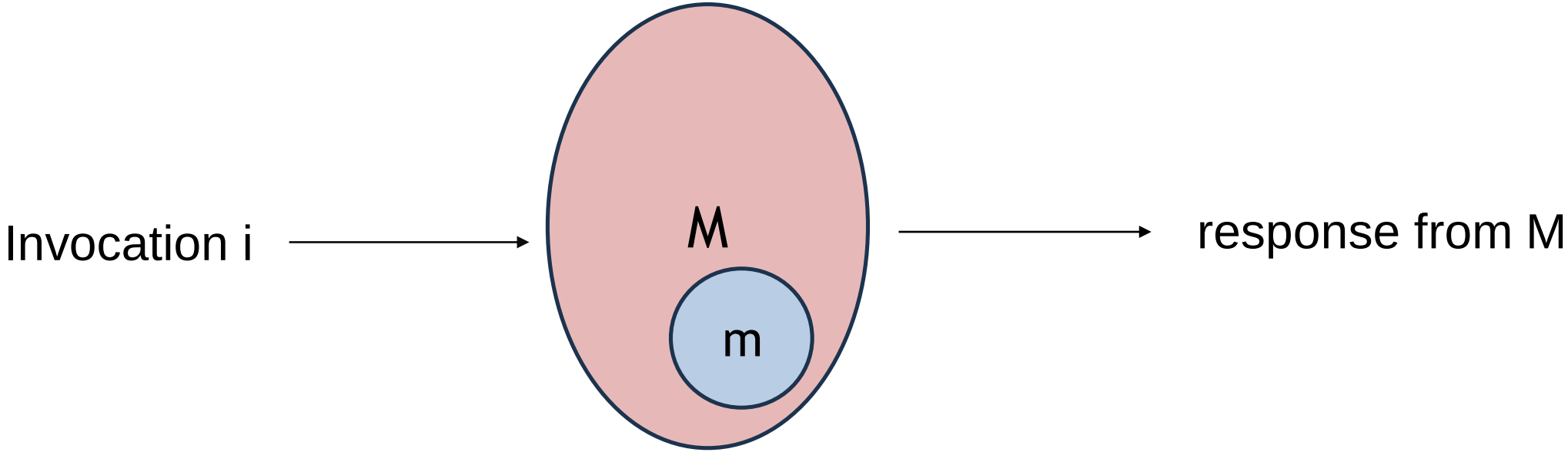
Conflict-free mode

s.H[1] = $[\text{reset}(2)_3, \overline{\text{OK}}_3,$ COMMUTE , $\text{isz}_1, \overline{\text{NZ}}_1]$.

s.H[2] = $[\text{reset}(2)_3, \overline{\text{OK}}_3,$ COMMUTE , $\text{isz}_2, \overline{\text{NZ}}_2]$.

s.H[3] = $[\text{reset}(2)_3, \overline{\text{OK}}_3,$ COMMUTE , $\text{dec}_3, \overline{\text{OK}}_3]$.

Implementation



Conflict-free mode

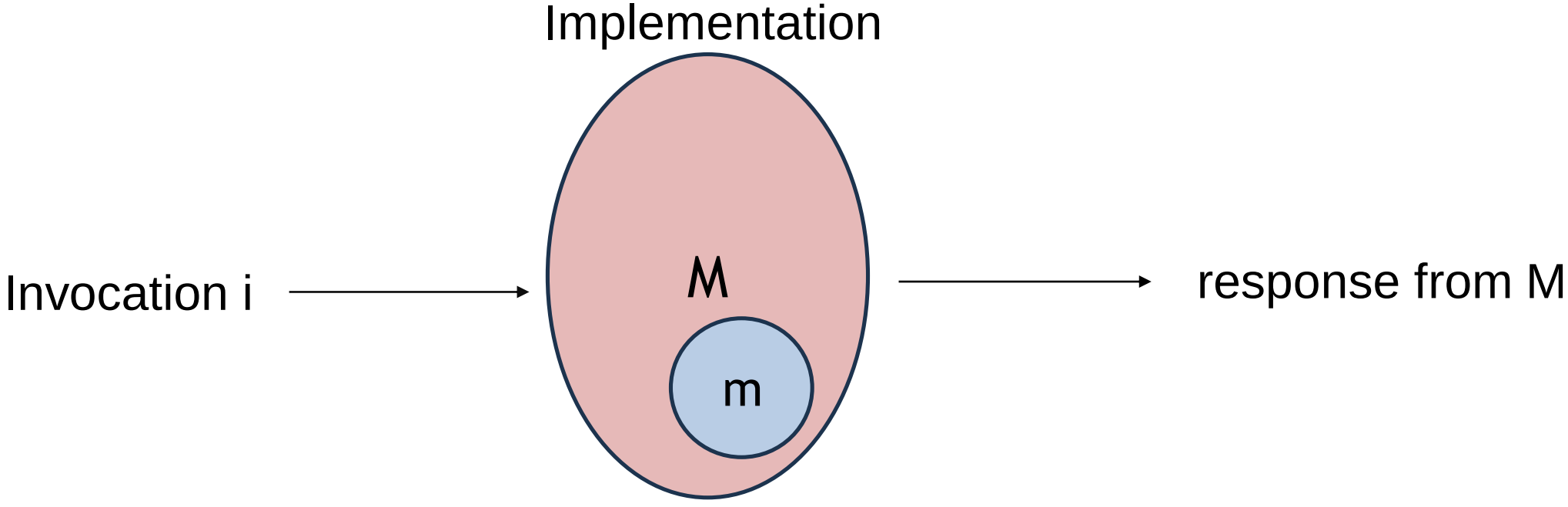
After we reach COMMUTE
For an invocation i from **thread** t:

If i ***matches invocation with*** head(s.H[t]) *:*
    respond with pre-recorded r
    pop(head(s.H[t]))
else:
    emulation mode

$s.H[1] = [\ isz_1, \overline{NZ}_1\ ].$

$s.H[2] = [\ isz_2, \overline{NZ}_2\ ].$

$s.H[3] = [\ dec_3, \overline{OK}_3\ ].$



Conflict-free mode

After we reach COMMUTE
For an invocation i from **thread** t:

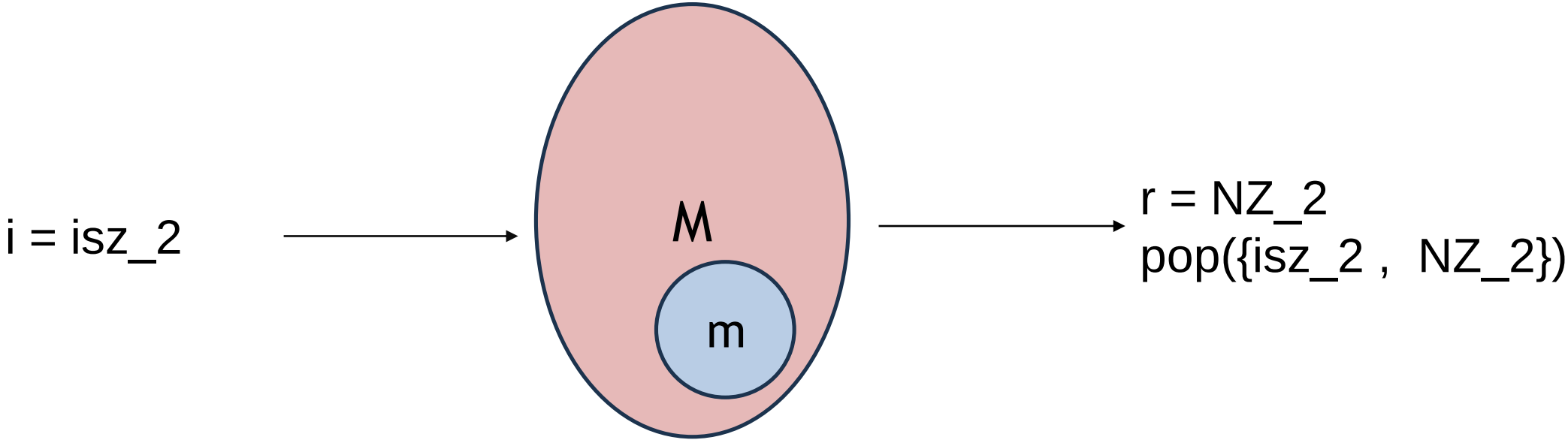
If i ***matches invocation with*** head(s.H[t]) *:*
    respond with pre-recorded r
    pop(head(s.H[t]))
else:
    emulation mode

s.H[1] = [$isz_1, \overline{NZ}_1$].

s.H[2] = [$isz_2, \overline{NZ}_2$]. ← **i matches this invocation**

s.H[3] = [$dec_3, \overline{OK}_3$].

Implementation



Conflict-free mode

After we reach COMMUTE
For an invocation i from **thread** t:

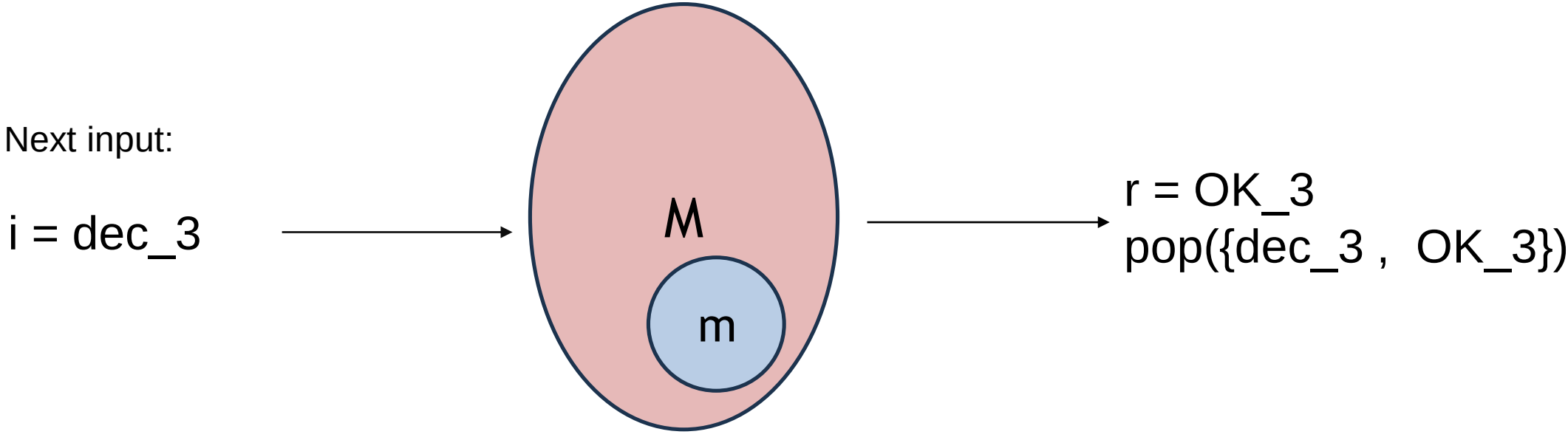
If i ***matches invocation with*** head(s.H[t]) *:*
    respond with pre-recorded r
    pop(head(s.H[t]))
else:
    emulation mode

s.H[1] = [$isz_1, \overline{NZ}_1$].

s.H[2] = [ ].

s.H[3] = [$dec_3, \overline{OK}_3$]. ← **i matches this invocation**

Implementation



Conflict-free mode

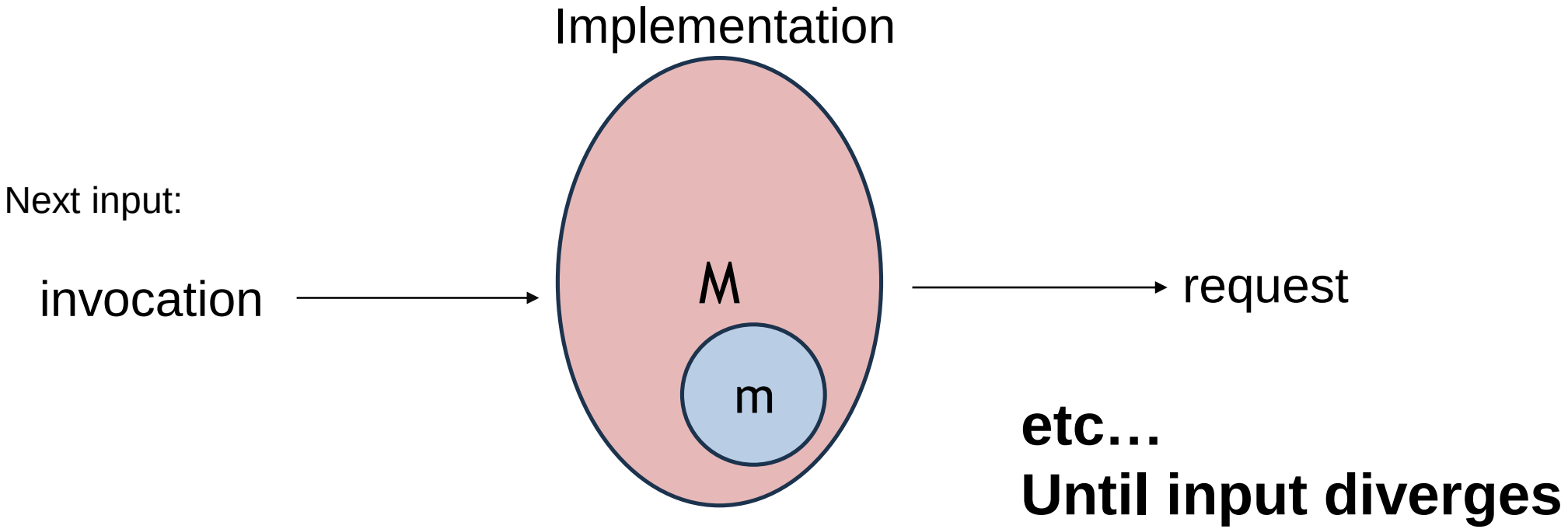
After we reach COMMUTE
For an invocation i from **thread** t:

If i *matches invocation with* head(s.H[t]) **:**
    respond with pre-recorded r
    pop(head(s.H[t]))
else:
    emulation mode

$s.H[1] = [\ isz_1, \overline{NZ}_1\ ].$

$s.H[2] = [\,].$

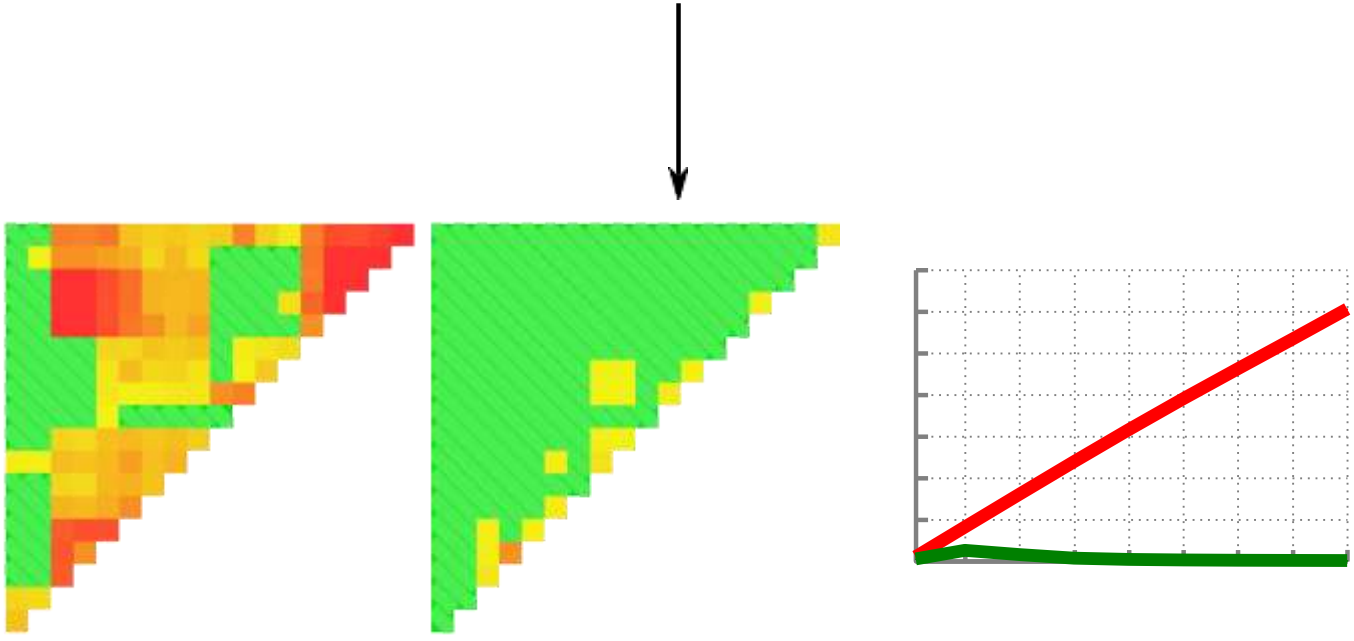
$s.H[3] = [\,].$



etc…
Until input diverges

Conclusion

Whenever interface operations commute,
they can be implemented in a way that scales.



Check out the code at http://pdos.csail.mit.edu/commuter

Thank you for listening !
Questions?