The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors

Austin T. Clements
M. Frans Kaashoek
Nickolai Zeldovich
Robert Morris
Eddie Kohler †

MIT CSAIL and † Harvard

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

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

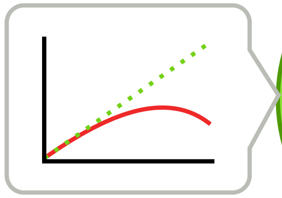
Current approach to scalable software development

2009 — **Corey**
OSDI '08

2010

Linux scalability
OSDI '10

2011



Plot scalability

Workload

Bonsai VM
ASPLOS '12

2012

2013

RadixVM
EuroSys '13

Current approach to scalable software development

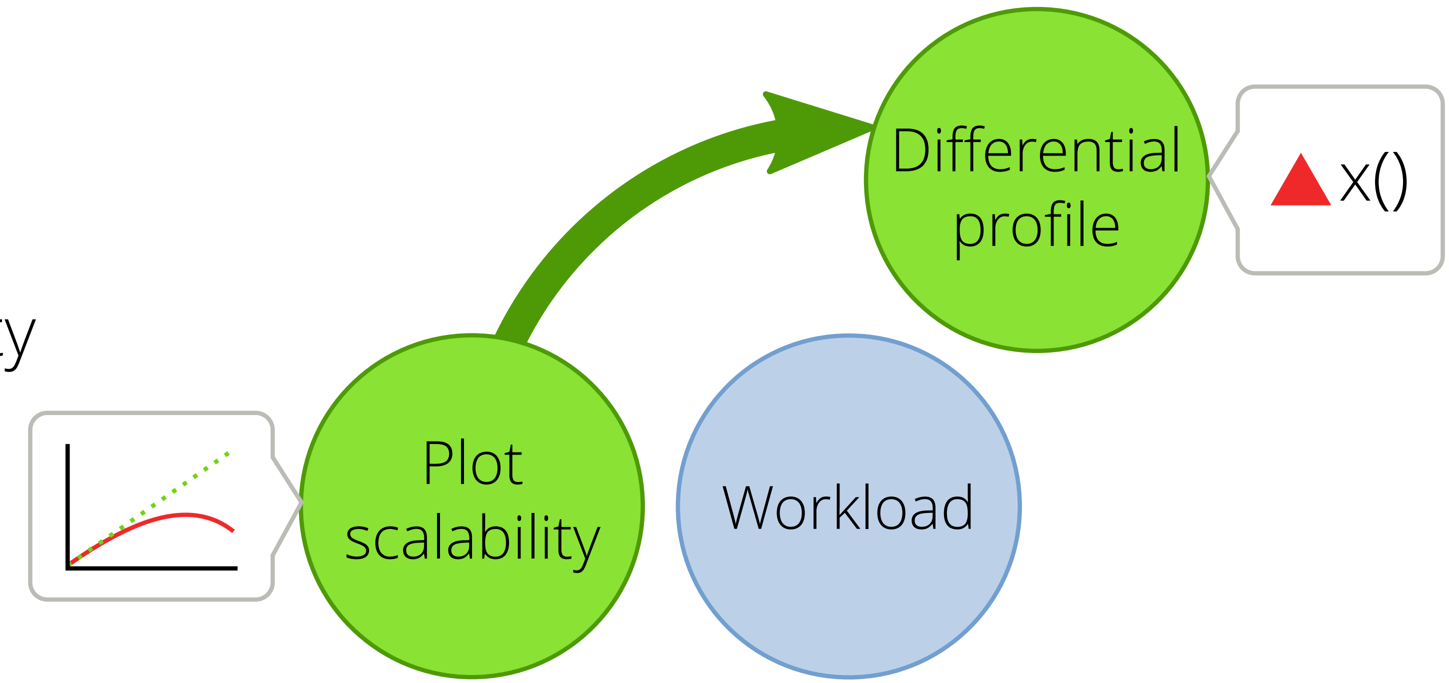
2009 — Corey, OSDI '08

2010 — Linux scalability, OSDI '10

2011

2012 — Bonsai VM, ASPLOS '12

2013 — RadixVM, EuroSys '13



Current approach to scalable software development

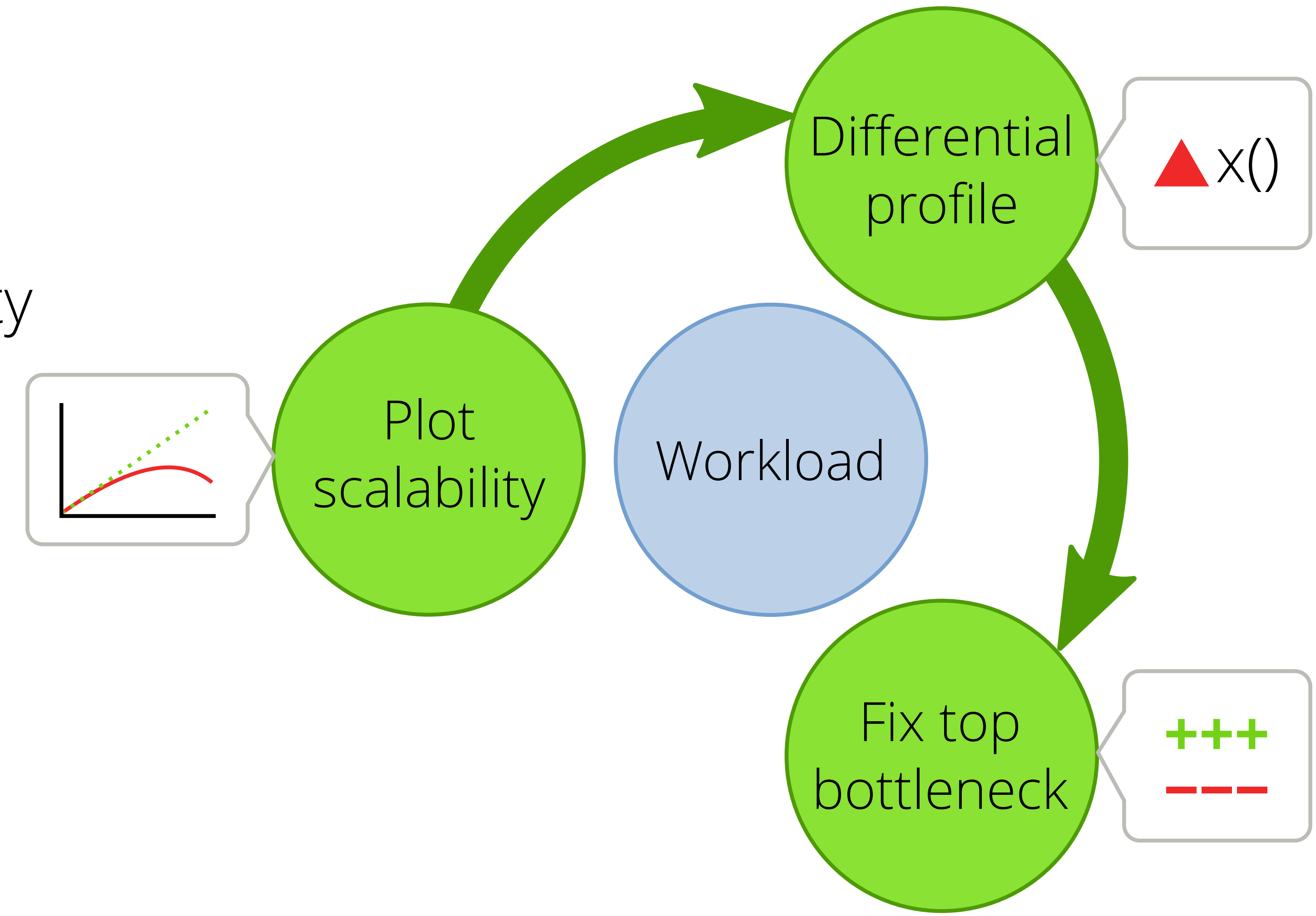
2009 — Corey
OSDI '08

2010 — Linux scalability
OSDI '10

2011

2012 — Bonsai VM
ASPLOS '12

2013 — RadixVM
EuroSys '13



Current approach to scalable software development

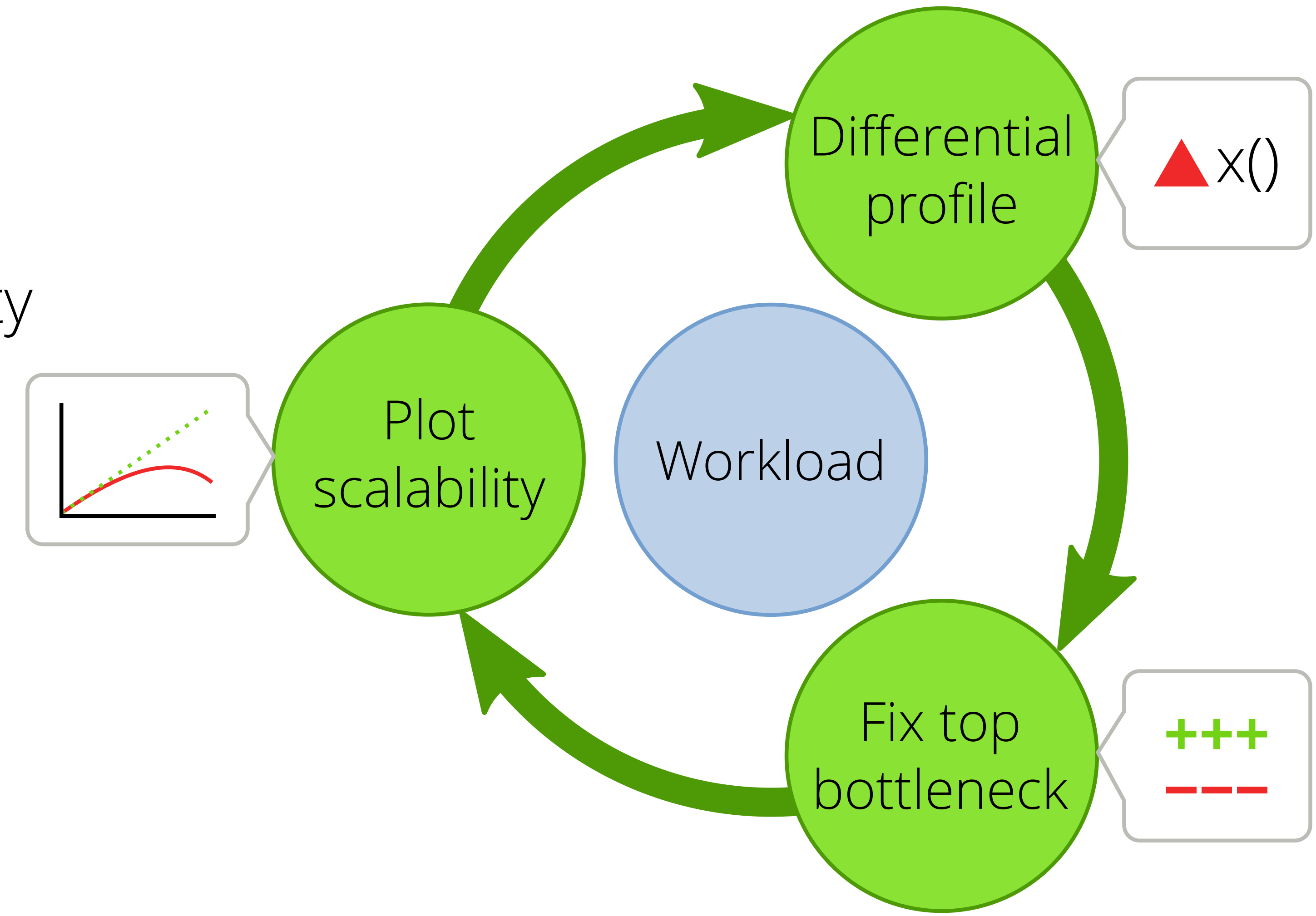
2009 — Corey
OSDI '08

2010 — Linux scalability
OSDI '10

2011

2012 — Bonsai VM
ASPLOS '12

2013 — RadixVM
EuroSys '13



Current approach to scalable software development

Successful in practice because it focuses developer effort

Disadvantages

- 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

Successful in practice because it focuses developer effort

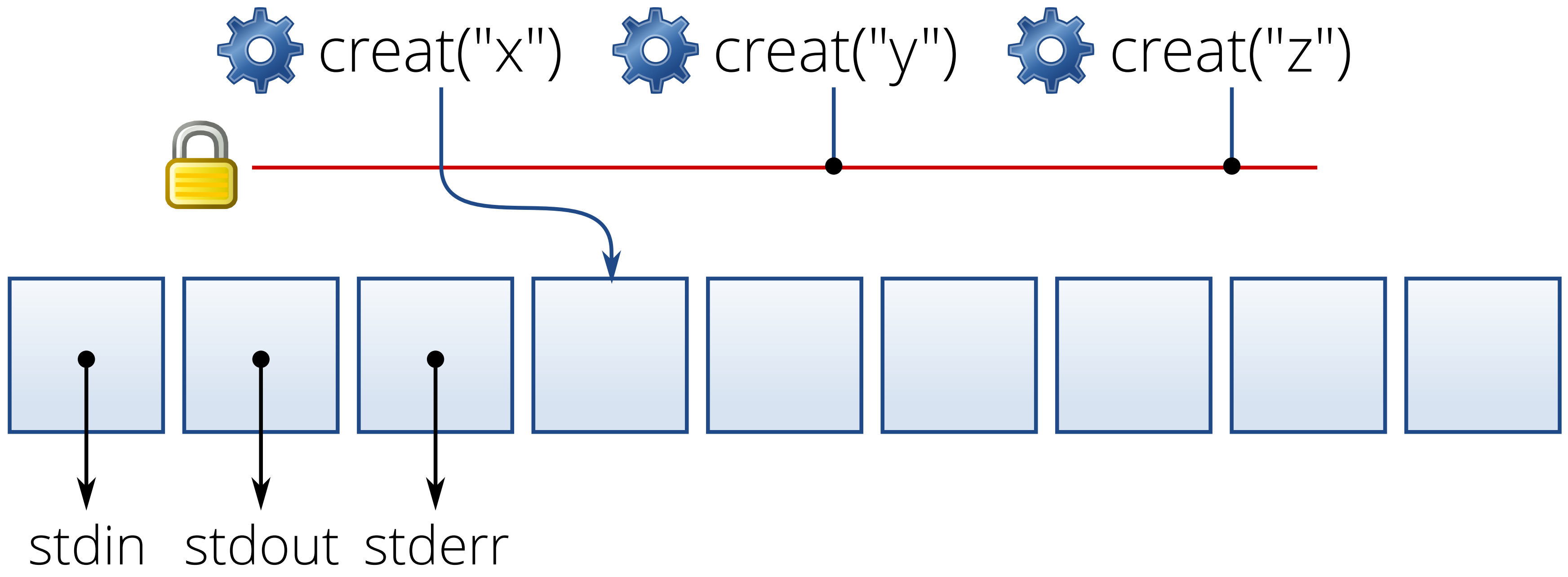
Disadvantages

- New workloads expose new bottlenecks
- More cores expose new bottlenecks
- The real bottlenecks may be in the interface design

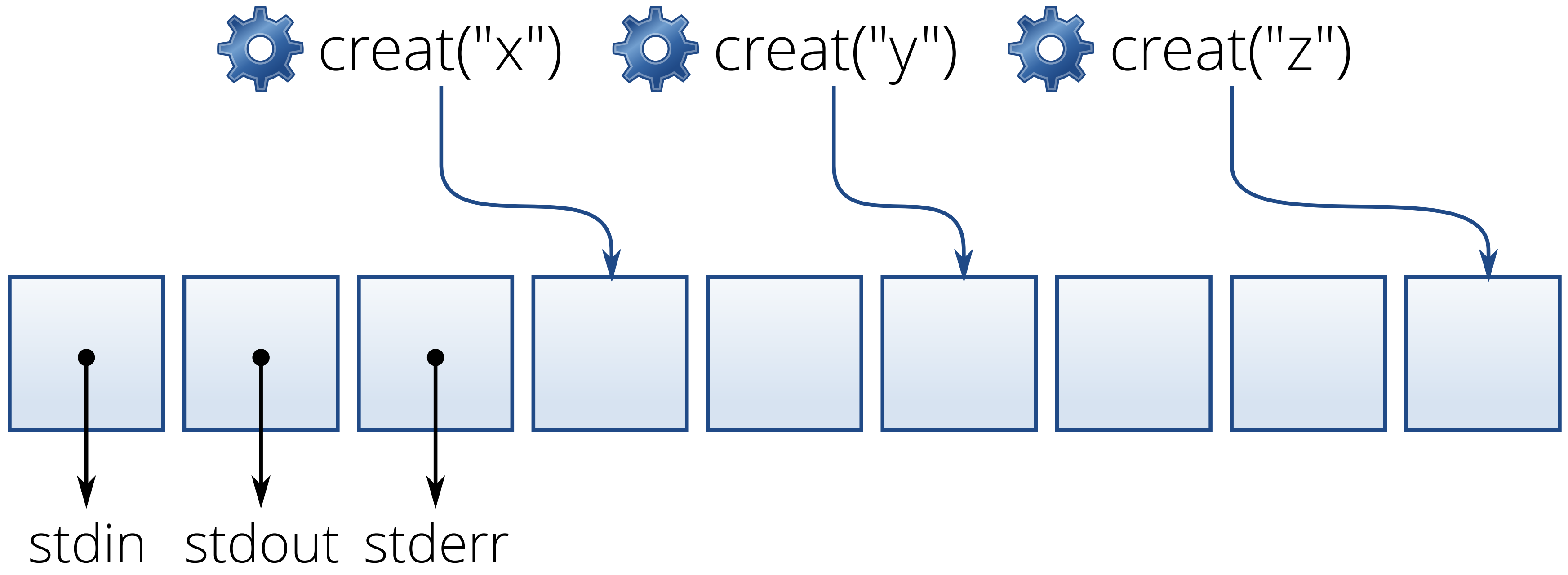
Interface scalability example

creat("x") creat("y") creat("z")

Interface scalability example



Interface scalability example



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.

	Commutes		Scalable implementation exists
creat with lowest FD	✗		
creat with any FD	✓	rule →	✓

Advantages of interface-driven scalability

The rule enables reasoning about scalability
throughout the software design process

Design	Guides design of scalable interfaces
Implement	Sets a clear implementation target
Test	Systematic, workload-independent scalability testing

Contributions

The scalable commutativity rule
- Formalization of the rule and proof of its correctness
- State-dependent, interface-based commutativity

Commuter: An automated scalability testing tool

sv6: A scalable POSIX-like kernel

Outline

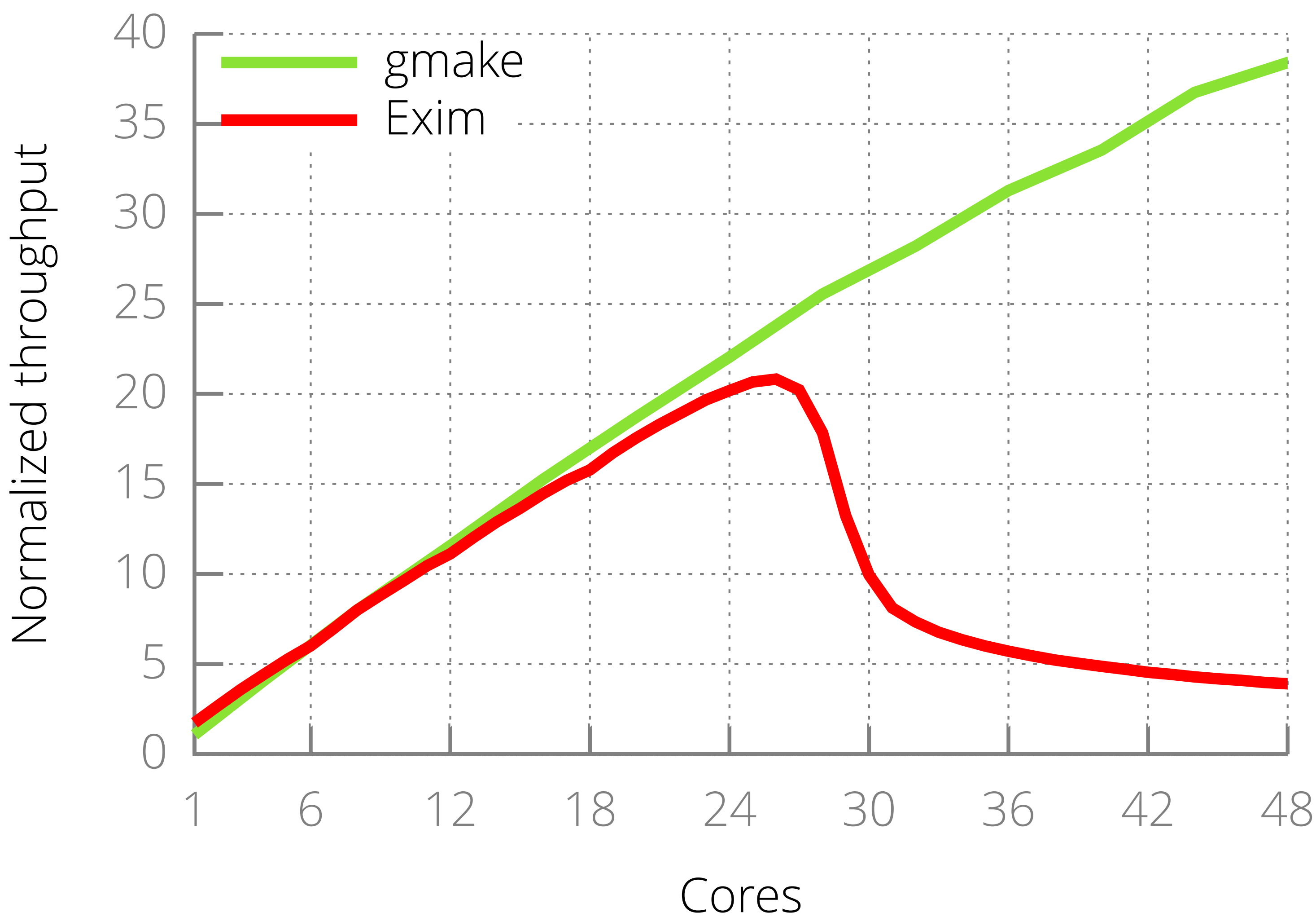
Defining the rule

- Definition of scalability
- Intuition
- Formalization

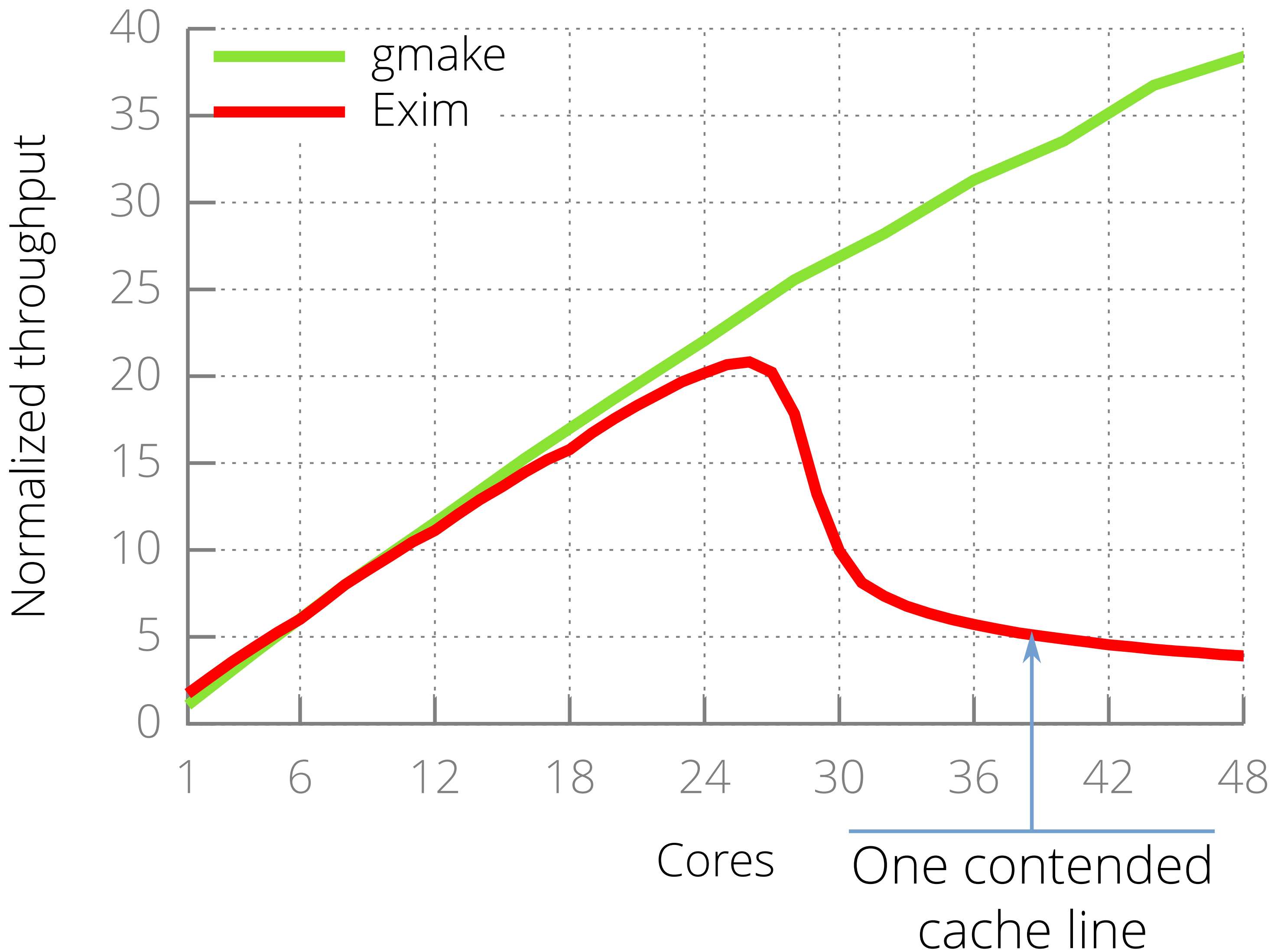
Applying the rule

- Commuter
- Evaluation

A scalability bottleneck

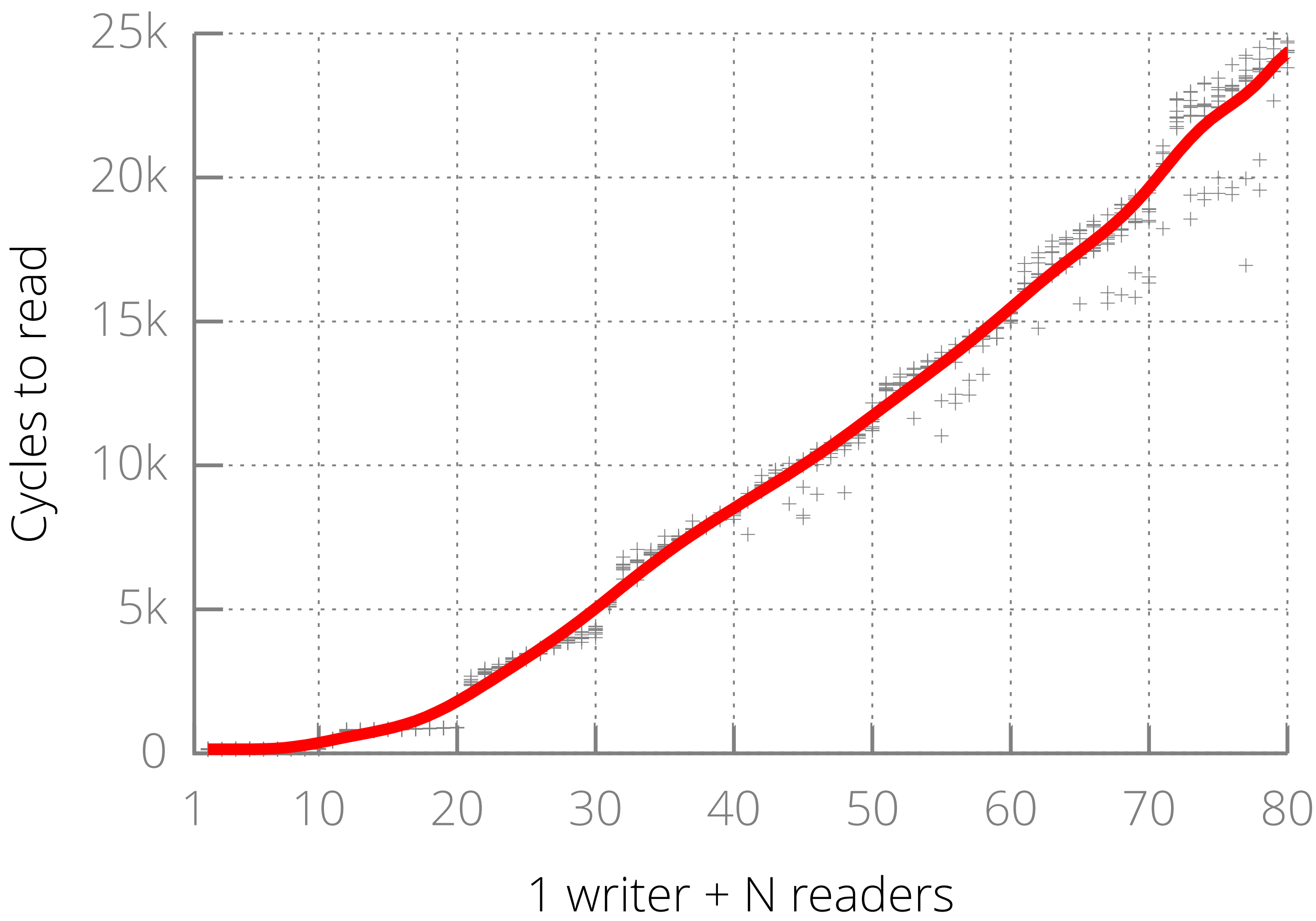


A scalability bottleneck

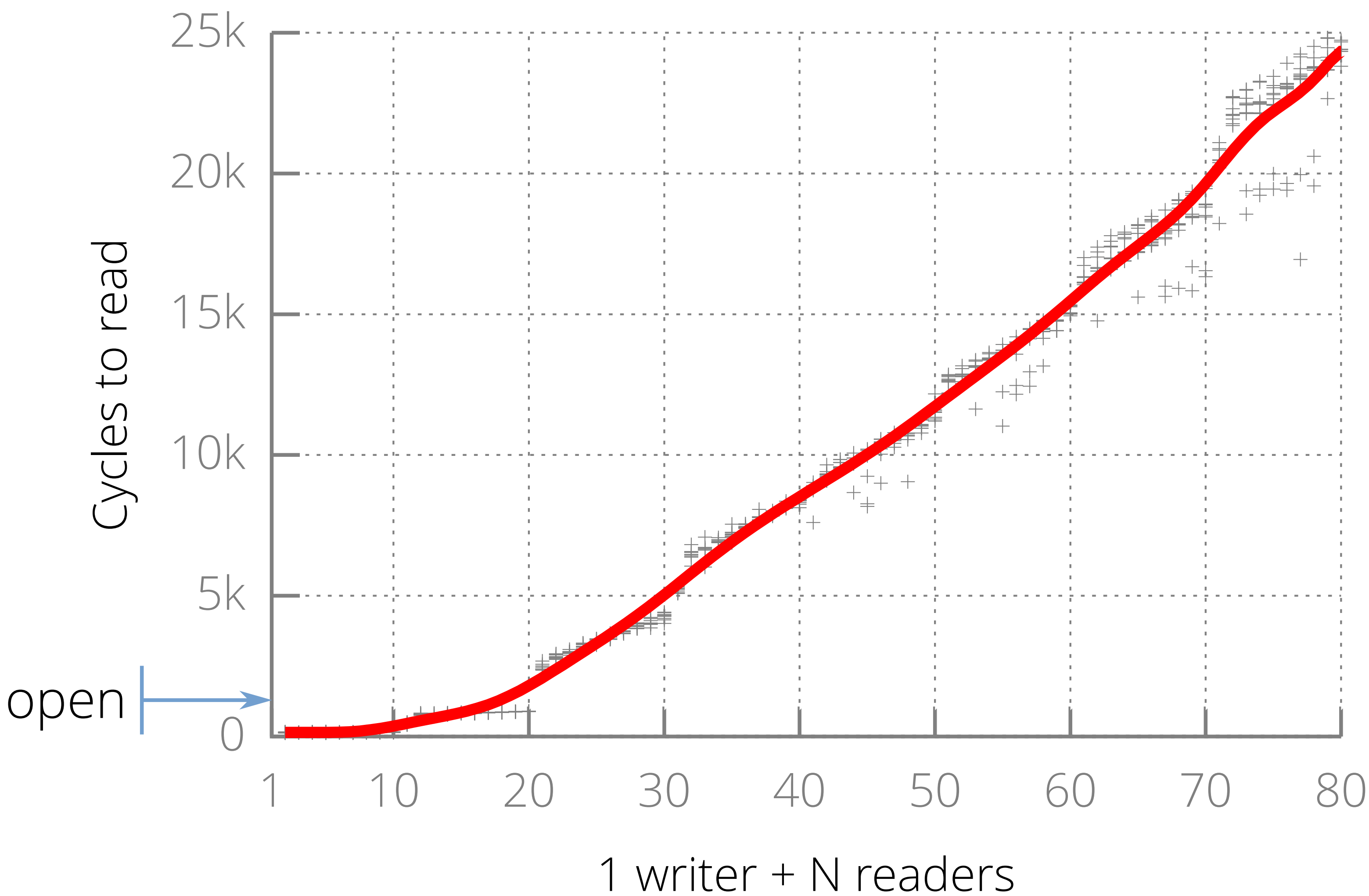


A single contended cache line can wreck scalability

Cost of a contended cache line



Cost of a contended cache line



What scales on today's multicores?

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

What scales on today's multicores?

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

What scales on today's multicores?

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

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
$\Rightarrow$ results independent of order
$\Rightarrow$ communication is unnecessary
$\Rightarrow$ without communication, no conflicts

Formalizing the rule

Y SI-commutes in $X \parallel Y$ ≔

$$\forall\, Y' \in \text{reorderings}(Y),\, Z\text{: } X \parallel Y \parallel Z \in \mathscr{S} \Leftrightarrow X \parallel Y' \parallel Z \in \mathscr{S}.$$

Y SIM-commutes in $X \parallel Y$ ≔

$$\forall\, P \in \text{prefixes}(\text{reorderings}(Y))\text{: } P \text{ SI-commutes in } X \parallel P.$$

An *implementation m* is a step function: $state \times inv \mapsto state \times resp$.

Given a specification $\mathscr{S}$,
a history $X \parallel Y$ in which *Y* SIM-commutes,
and a reference implementation M that can generate $X \parallel Y$,
∃ an implementation *m* of $\mathscr{S}$ whose steps in *Y* are conflict-free.

Proof by simulation construction.

Formalizing the rule

Commutativity is sensitive to
operations, arguments, *and state*

Example of using the rule

	Commutes	Scalable implementation exists
P1: creat P1: creat	✗	

Example of using the rule

	Commutes	Scalable implementation exists
P1: creat P1: creat	✗	
P1: creat("/tmp/x") P2: creat("/etc/y")		

Example of using the rule

	Commutes	Scalable implementation exists
P1: creat P1: creat	✗	
P1: creat("/tmp/x") P2: creat("/etc/y")	✓	✓ (Linux)

Example of using the rule

	Commutes	Scalable implementation exists
P1: creat P1: creat	✗	
P1: creat("/tmp/x") P2: creat("/etc/y")	✓	✓ (Linux)
P1: creat("/x") P2: creat("/y")		

Example of using the rule

	Commutes	Scalable implementation exists
P1: creat P1: creat	✗	
P1: creat("/tmp/x") P2: creat("/etc/y")	✓	✓ (Linux)
P1: creat("/x") P2: creat("/y")	✓	✓

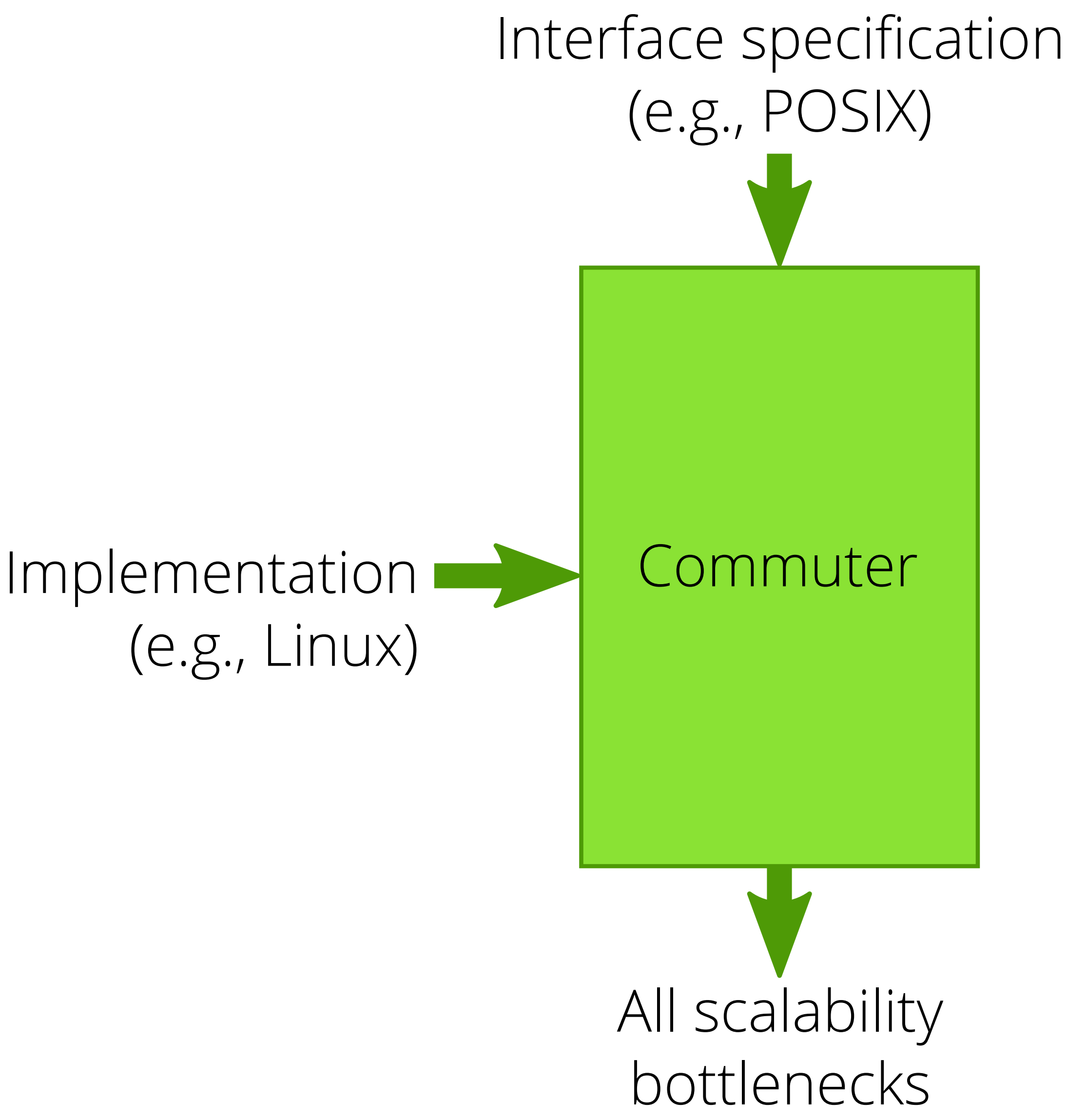
Example of using the rule

	Commutes	Scalable implementation exists
P1: creat P1: creat	✗	
P1: creat("/tmp/x") P2: creat("/etc/y")	✓	✓ (Linux)
P1: creat("/x") P2: creat("/y")	✓	✓
P1: creat("x", O_EXCL) P2: creat("x", O_EXCL)		

Example of using the rule

	Commutes	Scalable implementation exists
P1: creat P1: creat	✗	
P1: creat("/tmp/x") P2: creat("/etc/y")	✓	✓ (Linux)
P1: creat("/x") P2: creat("/y")	✓	✓
P1: creat("x", O_EXCL) P2: creat("x", O_EXCL)		
Same CWD	✗	
Different CWD	✓	✓

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:

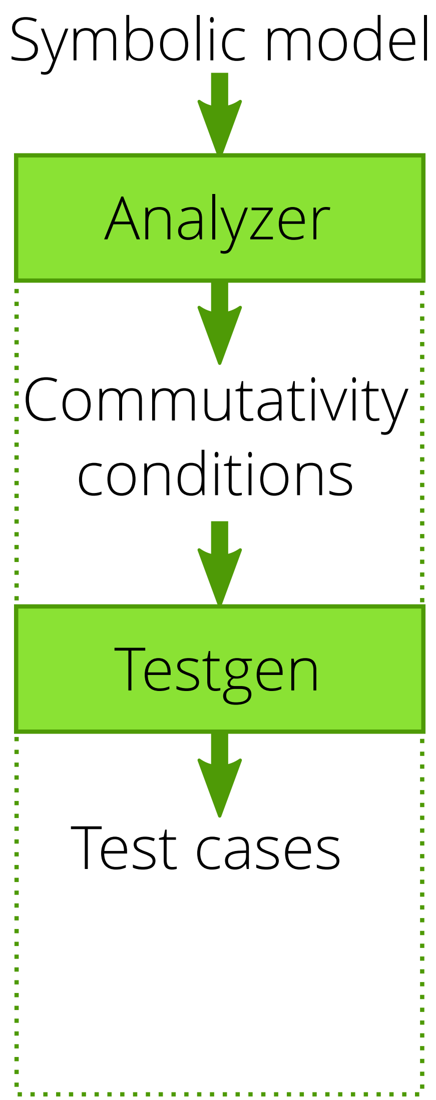
- Both source files exist and all names are different
- Neither source file exists
- a xor c exists, and it is not the other rename's destination
- Both calls are self-renames
- One call is a self-rename of an existing file and a != c
- a & c are hard links to the same inode, a != c, and b == d

Test cases

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

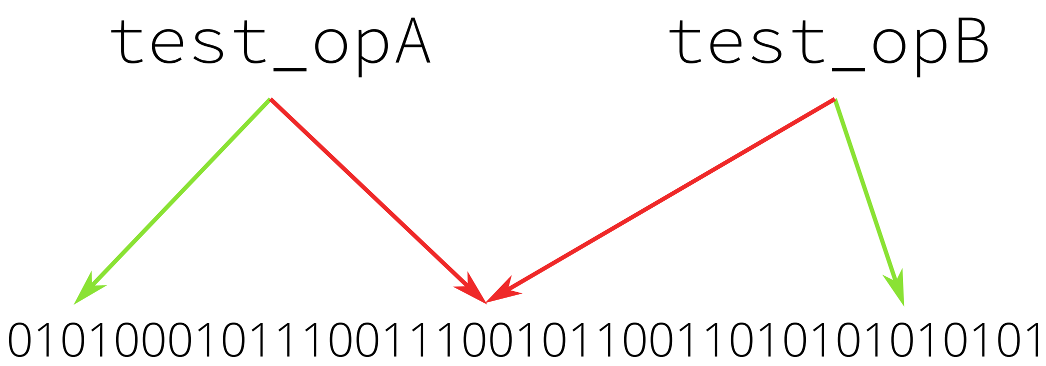
- Both source files exist and all names are different
- Neither source file exists
- a xor c exists, and it is not the other rename's destination
- Both calls are self-renames
- One call is a self-rename of an existing file and a != c
- a & c are hard links to the same inode, a != c, and b == d

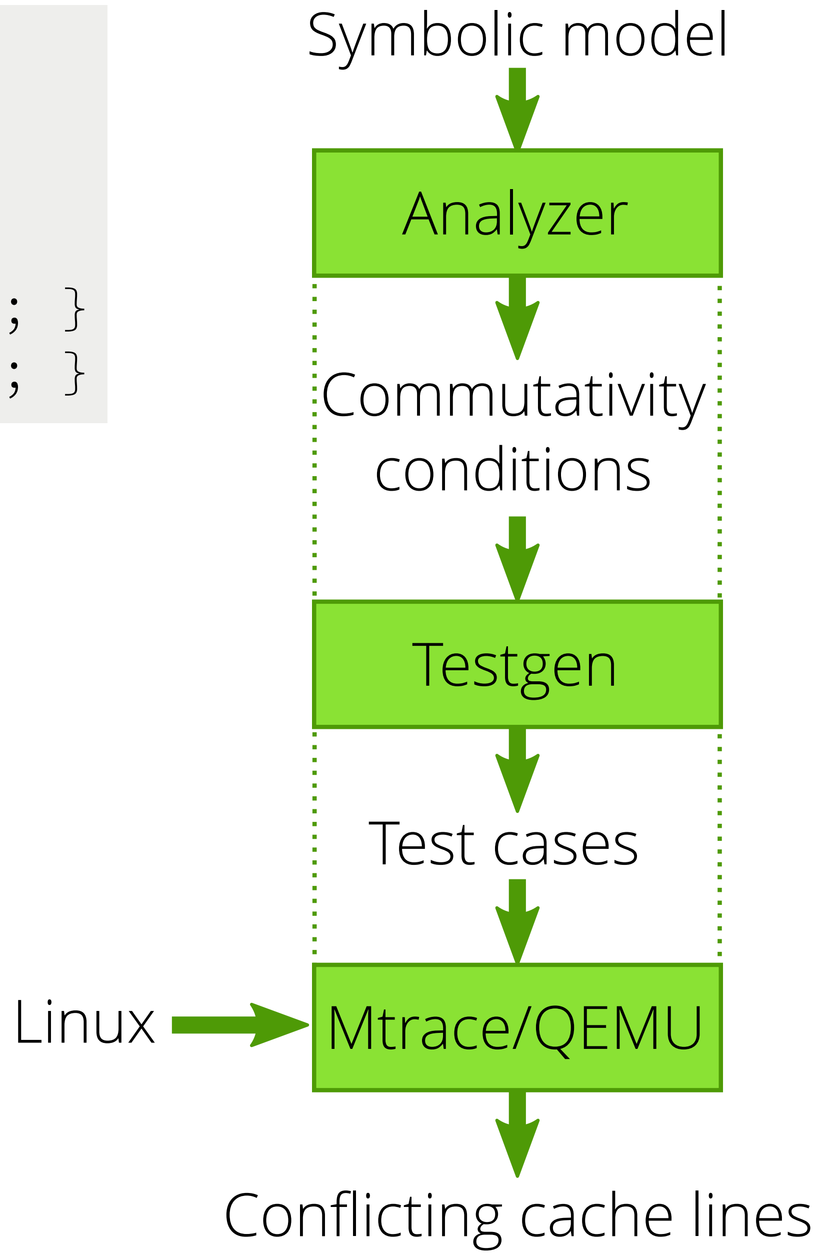
```
void setup() {
    close(creat("f0", 0666));
    close(creat("f2", 0666));
}
void test_opA() { rename("f0", "f1"); }
void test_opB() { rename("f2", "f3"); }
```



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"); }
```



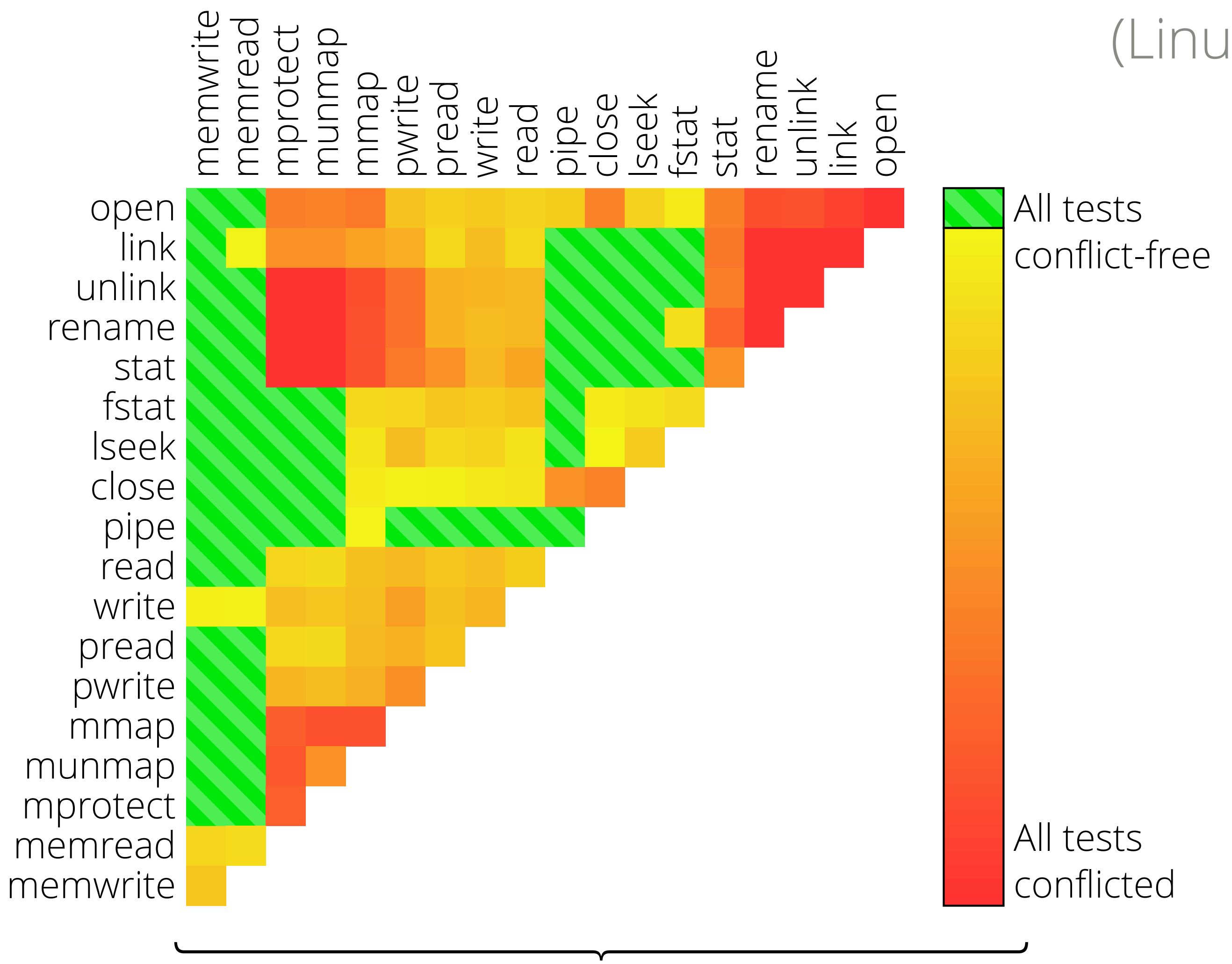


Evaluation

Does the rule help build scalable systems?

Commuter finds non-scalable cases in Linux

(Linux 3.8, ramfs)

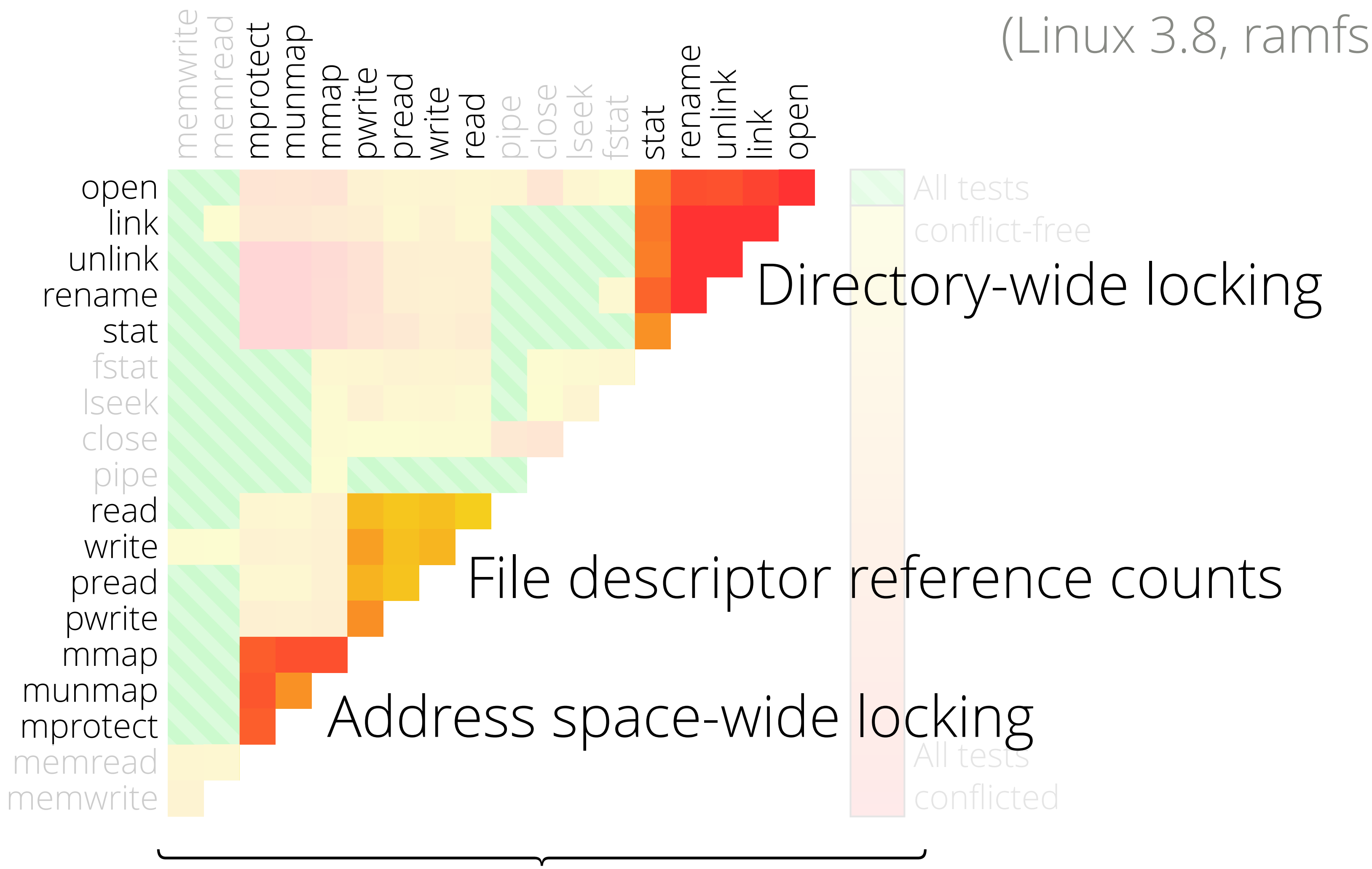


13,664 total test cases

68% are conflict-free

Many are "corner cases," many are not.

Commuter finds non-scalable cases in Linux



13,664 total test cases

68% are conflict-free

Many are "corner cases," many are not.

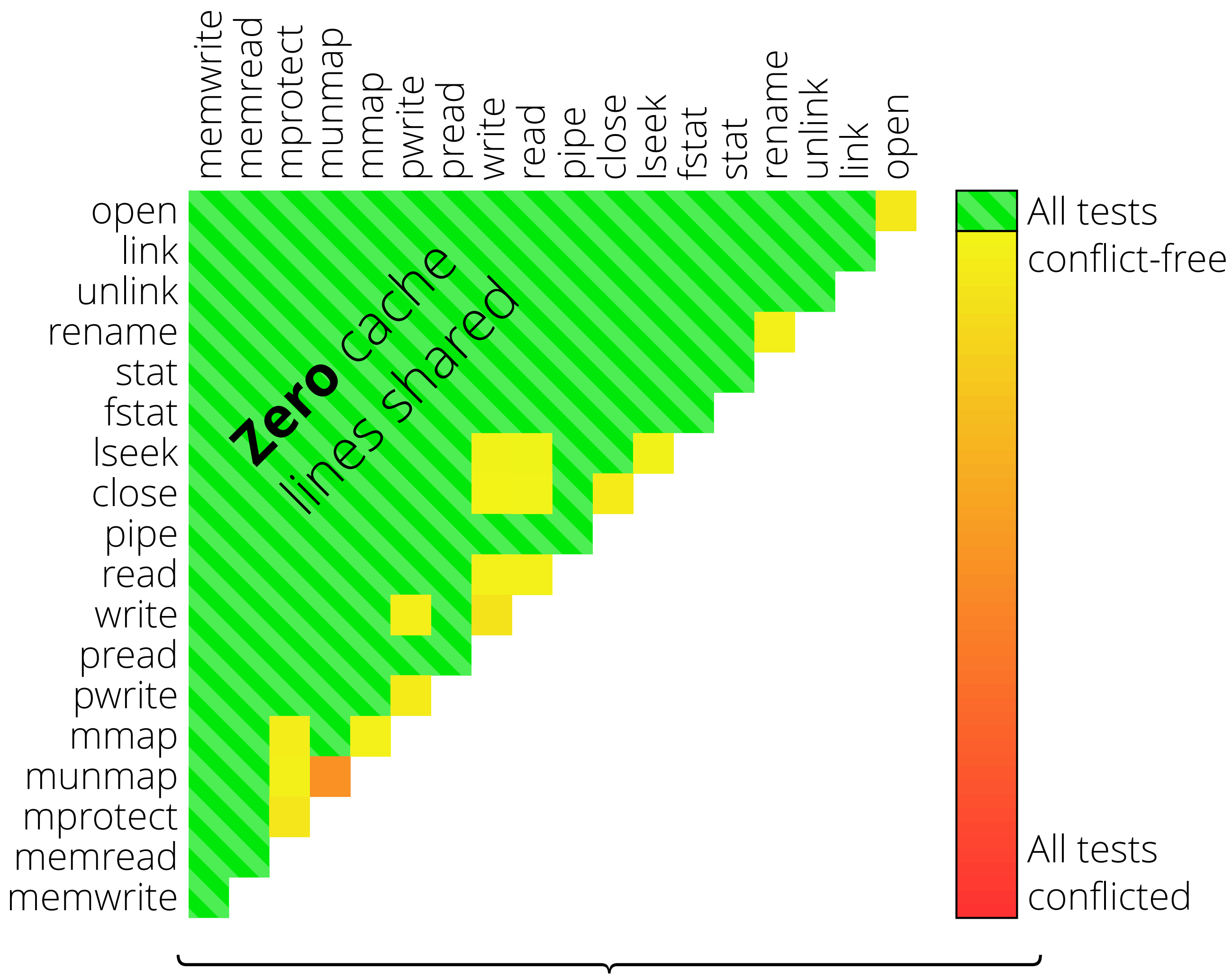
sv6: A scalable OS

POSIX-like operating system

File system and virtual memory system follow commutativity rule

Implementation using standard parallel programming techniques, but guided by Commuter

Commutative operations can be made to scale

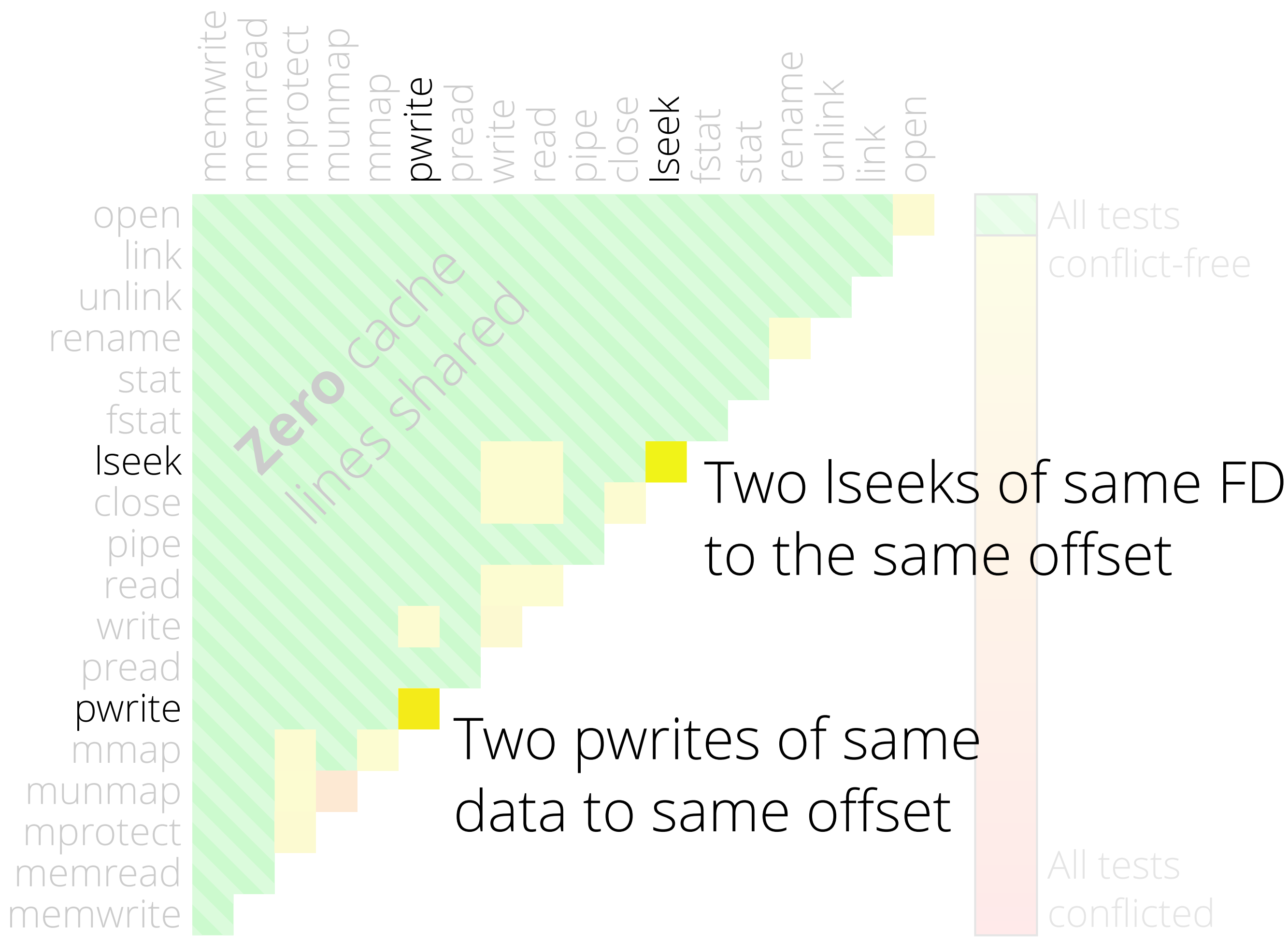


13,664 total test cases

99% are conflict-free

Remaining 1% are mostly "idempotent updates"

Commutative operations can be made to scale



13,664 total test cases

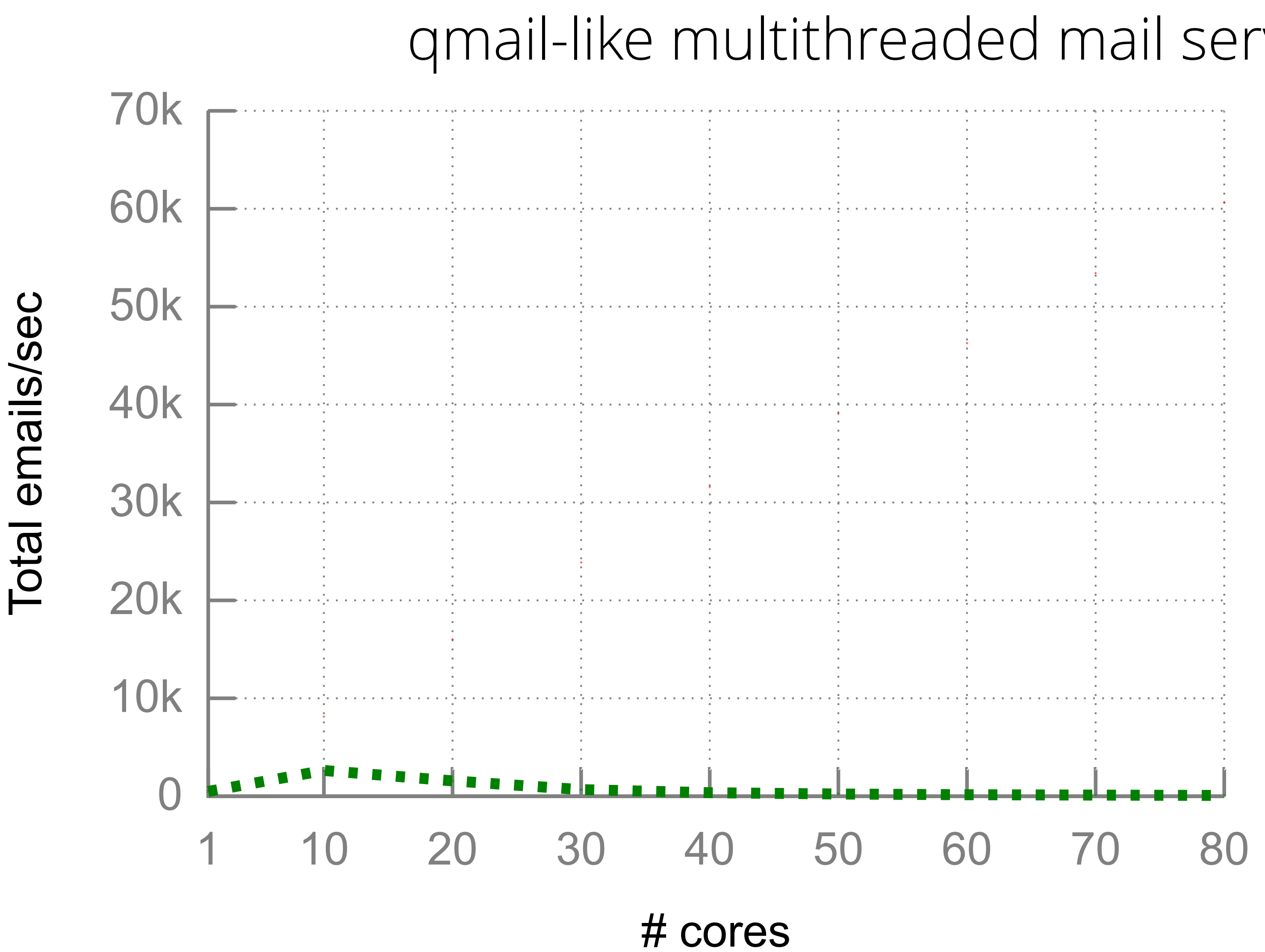
99% are conflict-free

Remaining 1% are mostly "idempotent updates"

Refining POSIX with the rule

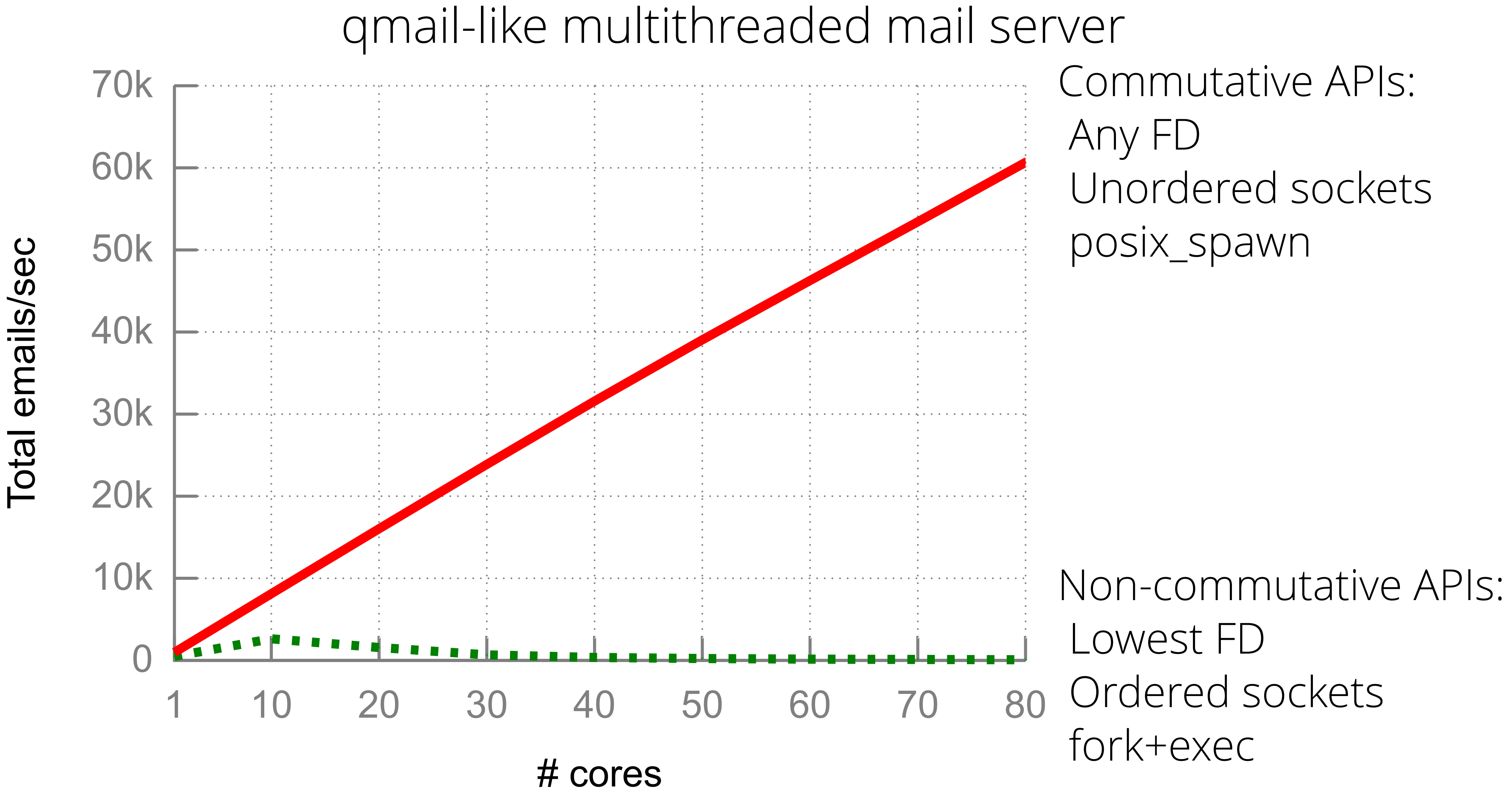
- Lowest FD versus any FD
- stat versus xstat
- Unordered sockets
- Delayed munmap
- fork+exec versus posix_spawn

Commutative operations matter to app scalabiliy



Non-commutative APIs:
Lowest FD
Ordered sockets
fork+exec

Commutative operations matter to app scalabiliy



Related work

Commutativity and concurrency
- [Bernstein '81]
- [Weihl '88]
- [Steele '90]
- [Rinard '97]
- [Shapiro '11]

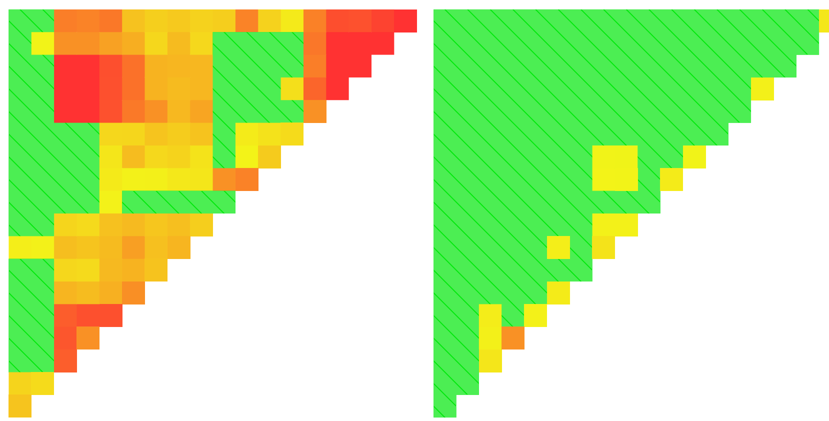
Laws of Order [Attiya '11]

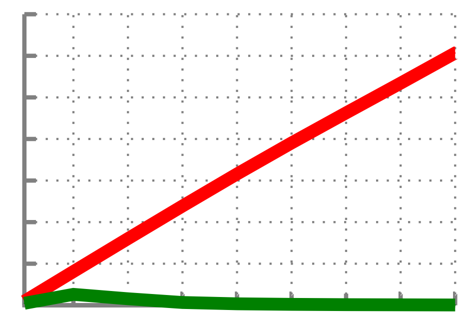
Disjoint-access parallelism [Israeli '94]
Scalable locks [MCS '91]
Scalable reference counting [Ellen '07, Corbet '10]

Conclusion

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

Design
Implement
Test





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