The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors

Austin T. Clements

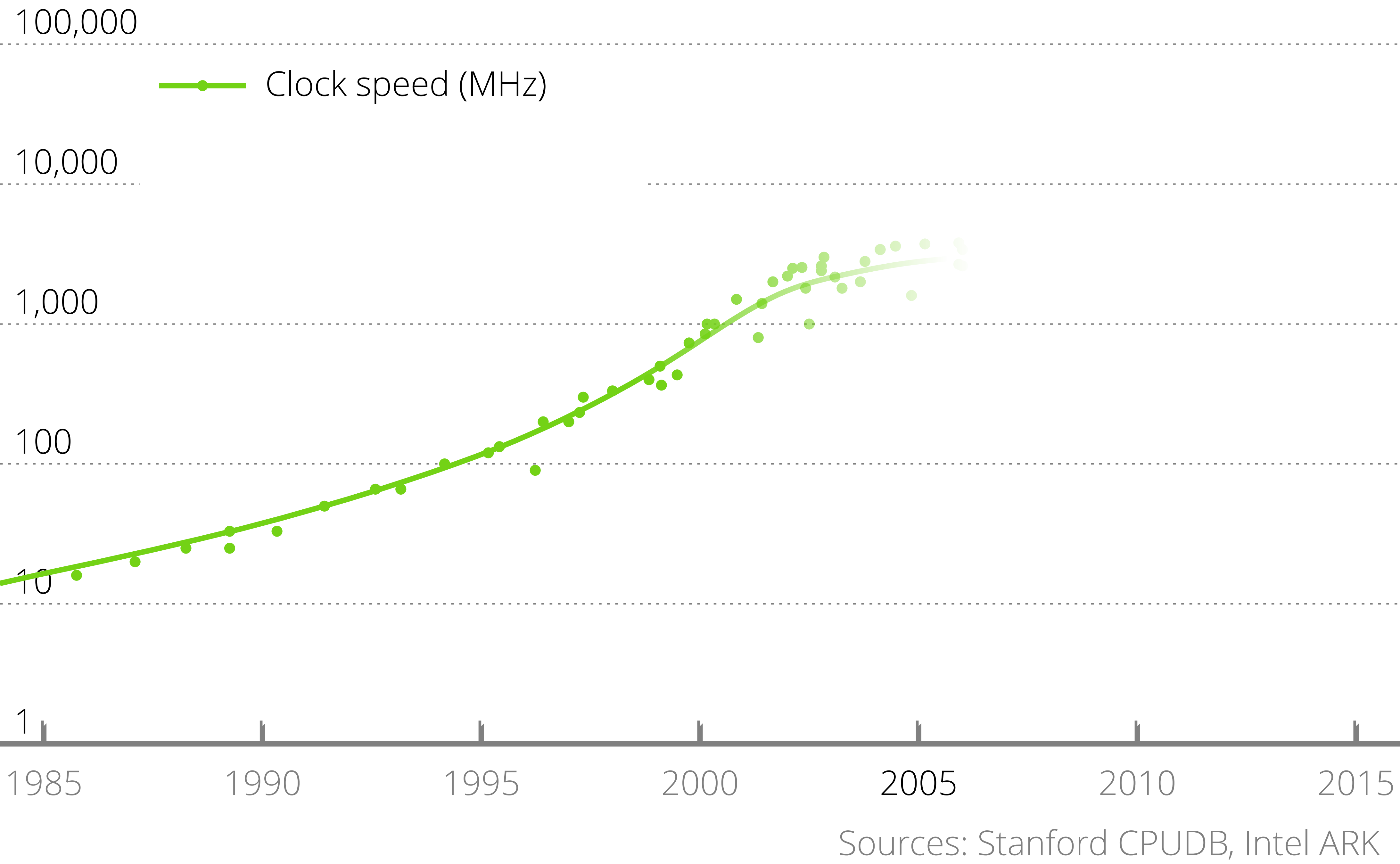
Thesis advisors:
M. Frans Kaashoek
Nickolai Zeldovich
Robert Morris
Eddie Kohler

x86 CPU trends

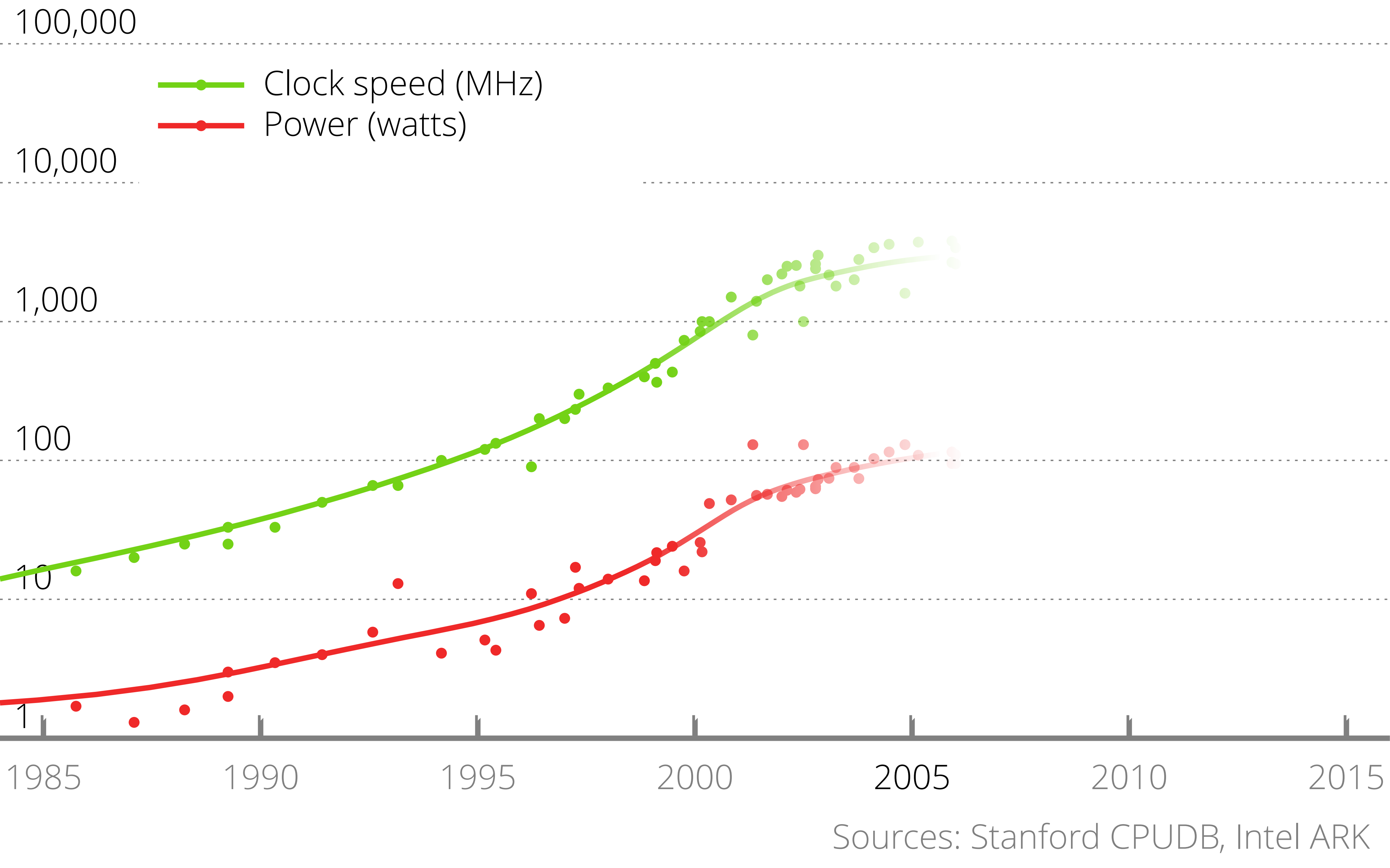
x86 CPU trends

2005

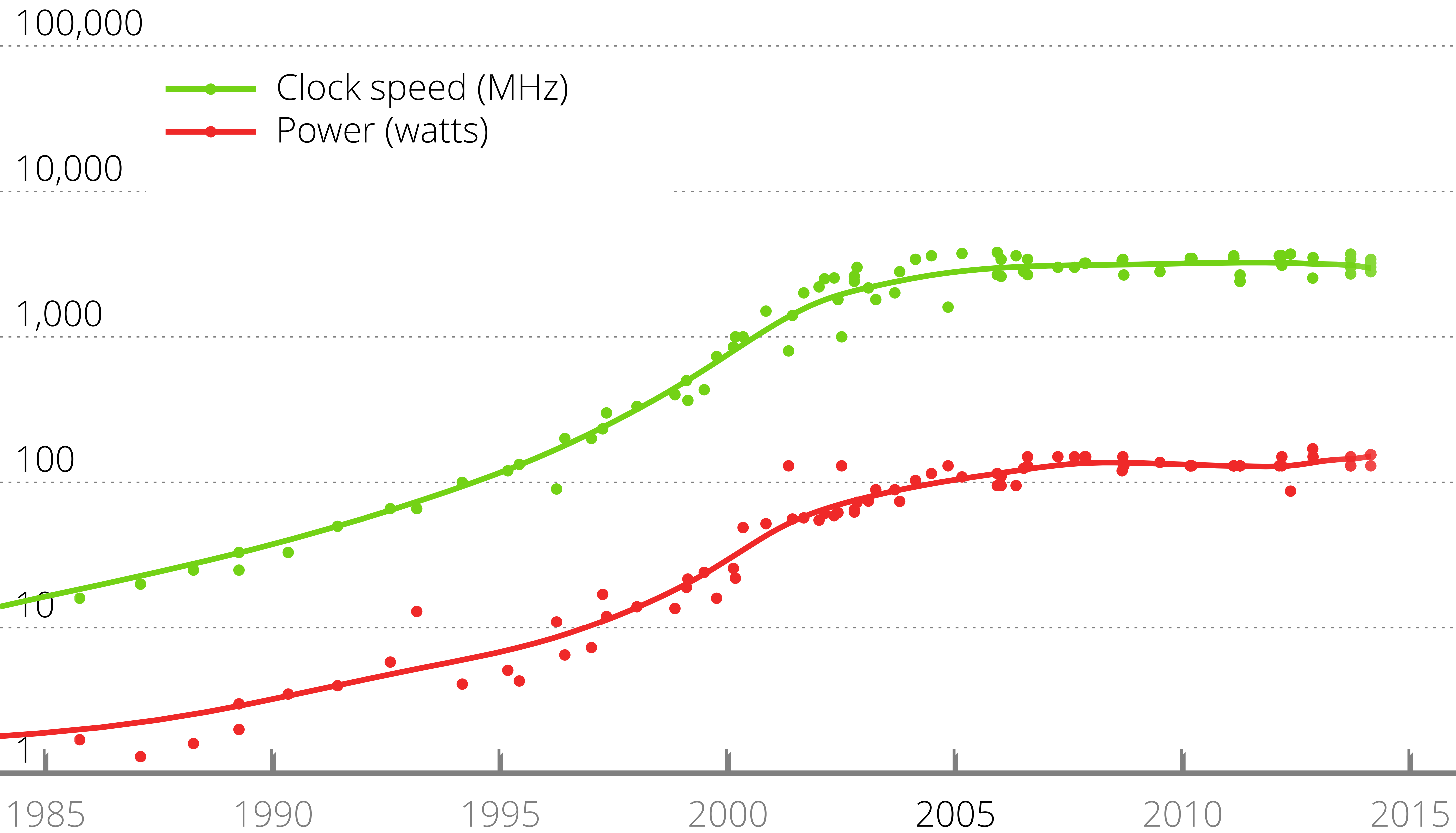
x86 CPU trends



x86 CPU trends

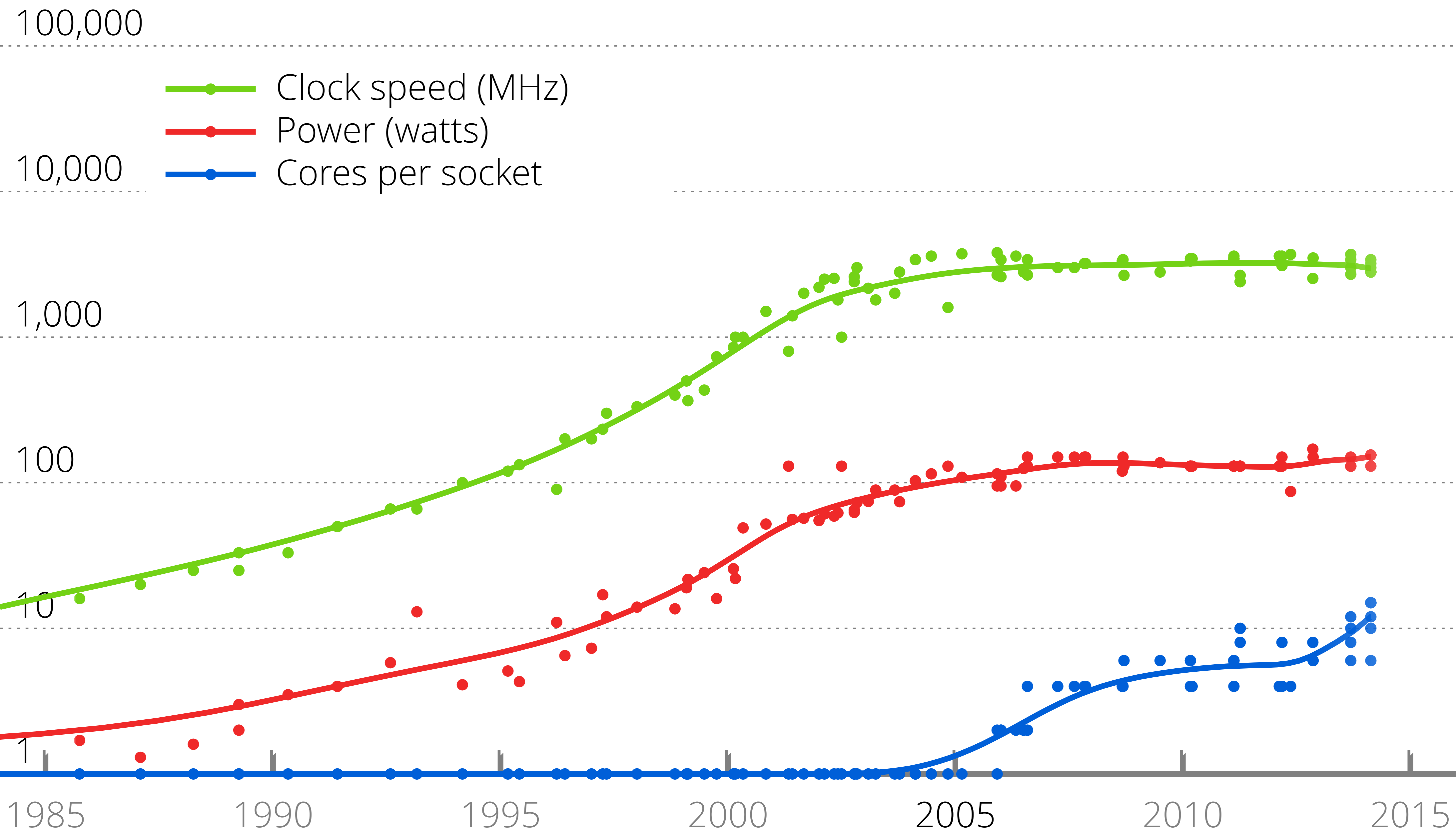


x86 CPU trends



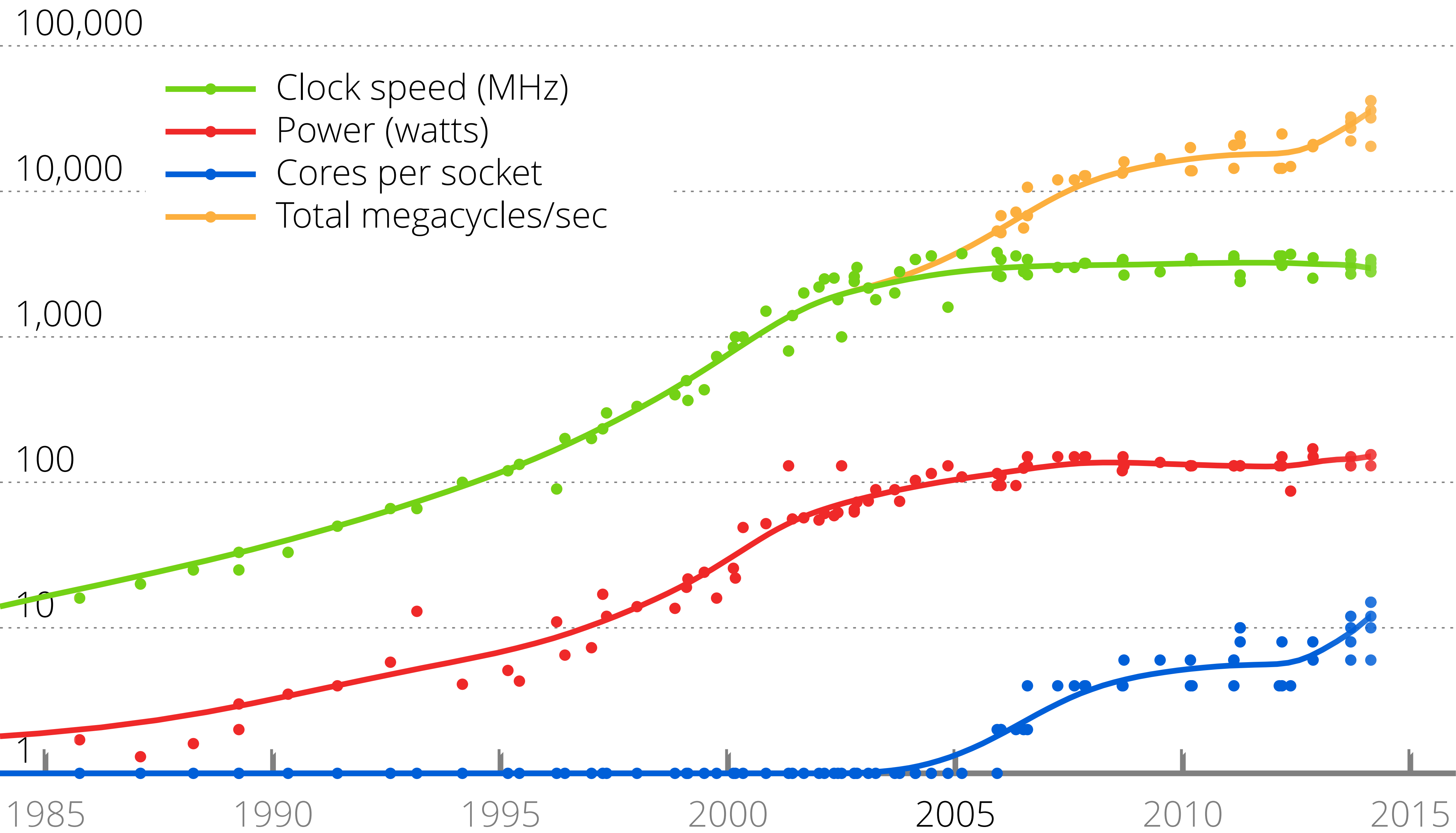
Sources: Stanford CPUDB, Intel ARK

x86 CPU trends



Sources: Stanford CPUDB, Intel ARK

x86 CPU trends



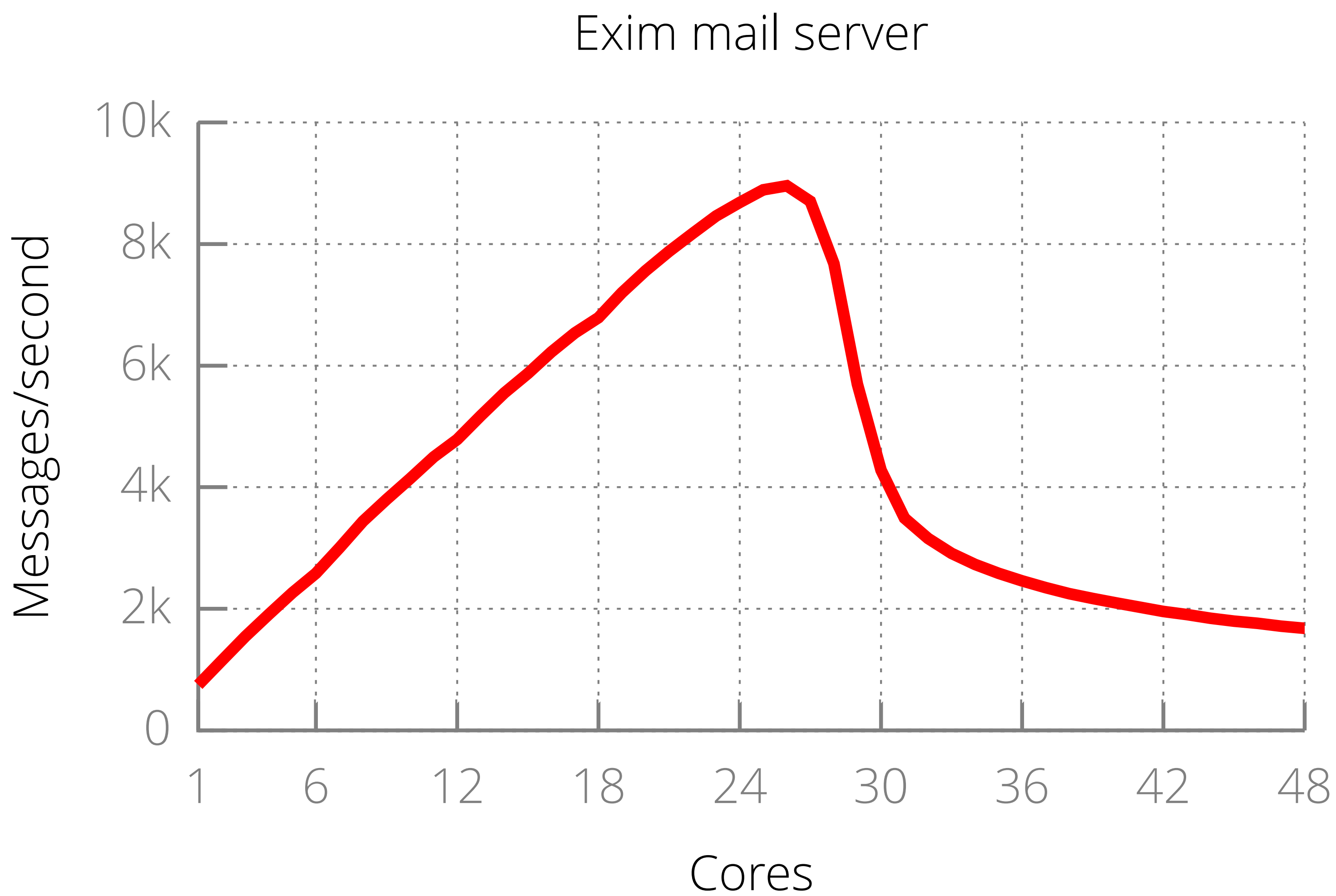
Sources: Stanford CPUDB, Intel ARK

Parallelize or perish

Software must be increasingly parallel to keep up with hardware, but scaling with parallelism is notoriously hard

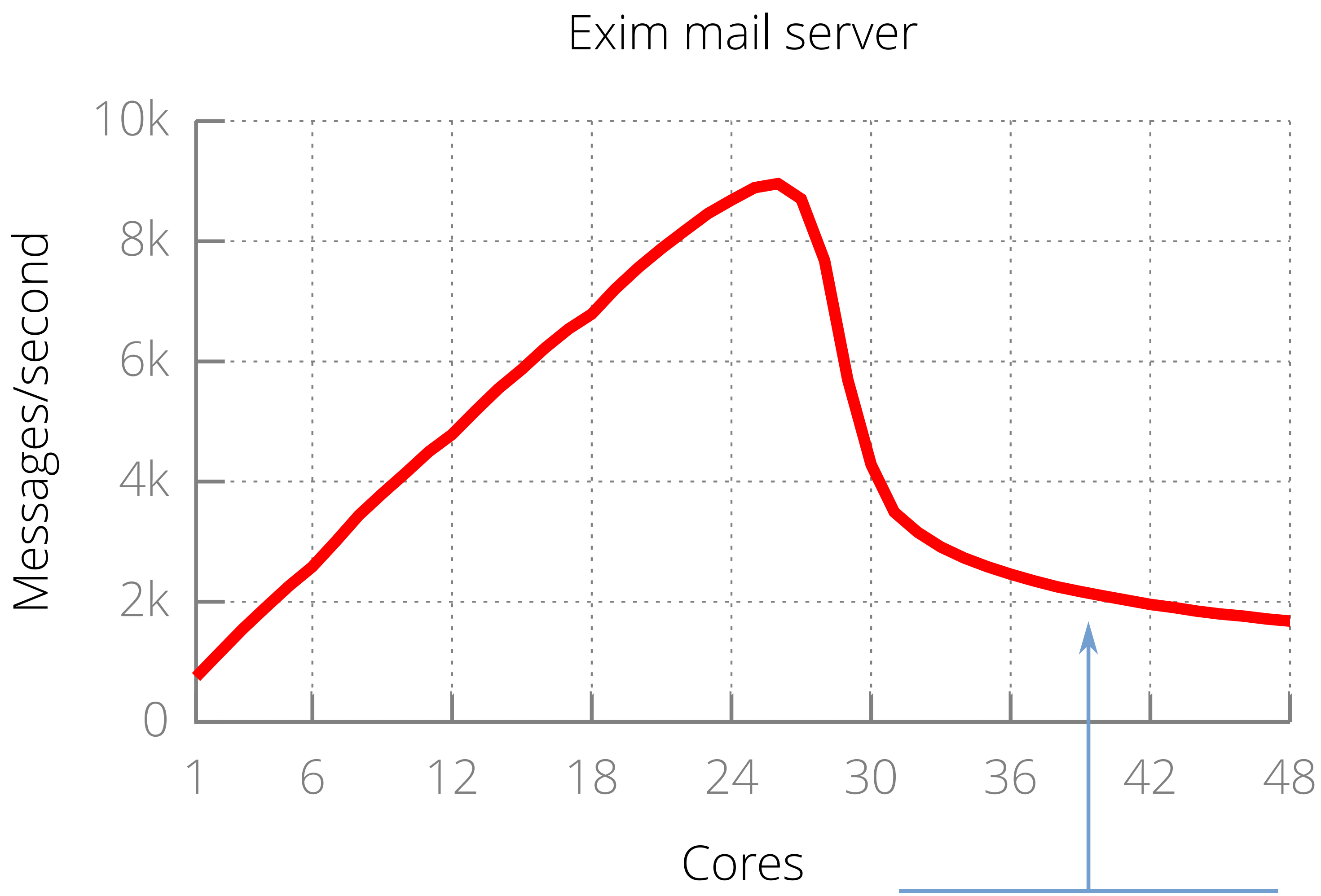
Parallelize or perish

Software must be increasingly parallel to keep up with hardware, but scaling with parallelism is notoriously hard



Parallelize or perish

Software must be increasingly parallel to keep up with hardware, but scaling with parallelism is notoriously hard



Problem lies in the OS kernel

OS kernel scalability

Kernel scalability is important

- Many applications depend on the OS kernel
- If the kernel doesn't scale, many applications won't scale

And hard

- $|\text{kernel threads}| > \sum |\text{application threads}|$
- Diverse and unknown workloads

Current approach to scalable software development

Corey
2009 — OSDI '08

2010

Linux scalability
OSDI '10

2011

Bonsai VM
2012 — ASPLOS '12

2013

RadixVM
EuroSys '13

Current approach to scalable software development

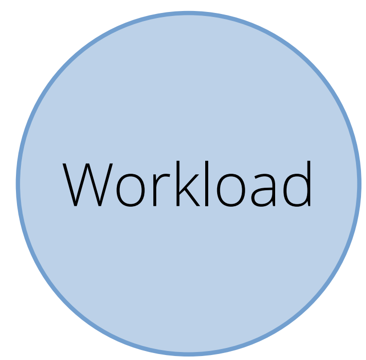
2009 — Corey
OSDI '08

2010 — Linux scalability
OSDI '10

2011

2012 — Bonsai VM
ASPLOS '12

2013 — RadixVM
EuroSys '13



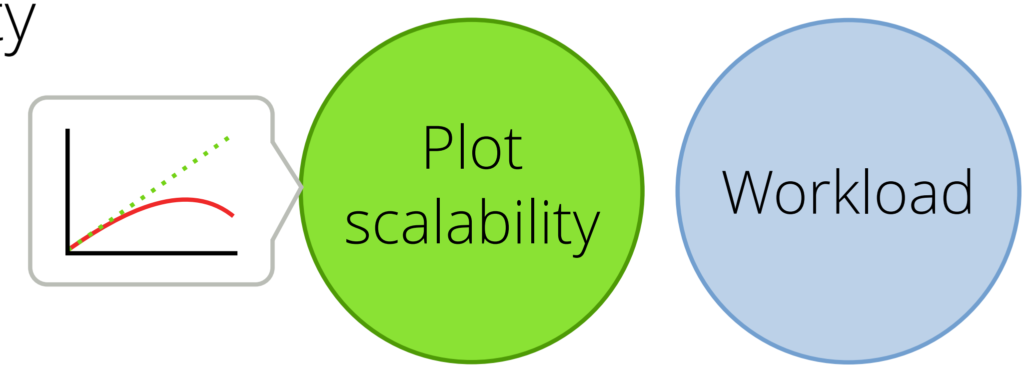
Current approach to scalable software development

2009 — Corey
OSDI '08

2010

Linux scalability
OSDI '10

2011



Bonsai VM
ASPLOS '12

2012

2013

RadixVM
EuroSys '13

Current approach to scalable software development

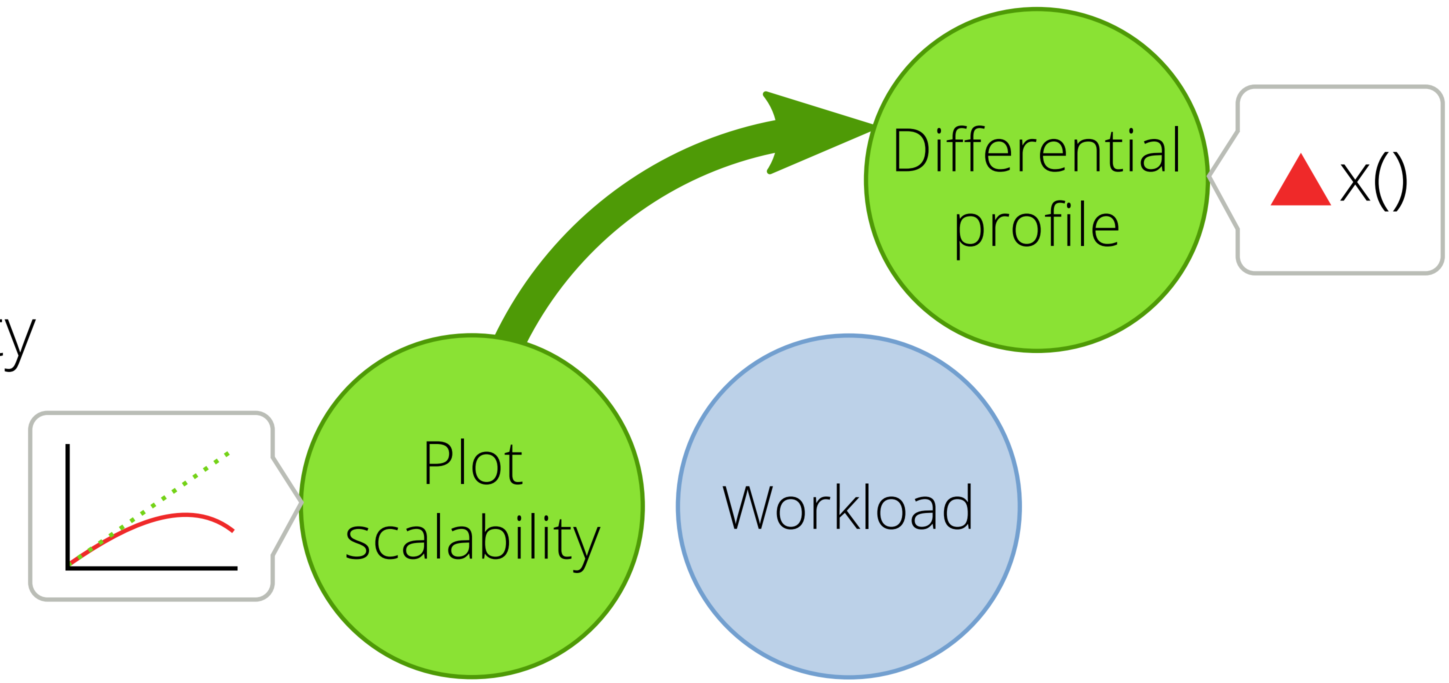
2009 — Corey, OSDI '08

2010 — Linux scalability, OSDI '10

2011

2012 — Bonsai VM, ASPLOS '12

2013 — RadixVM, EuroSys '13



Current approach to scalable software development

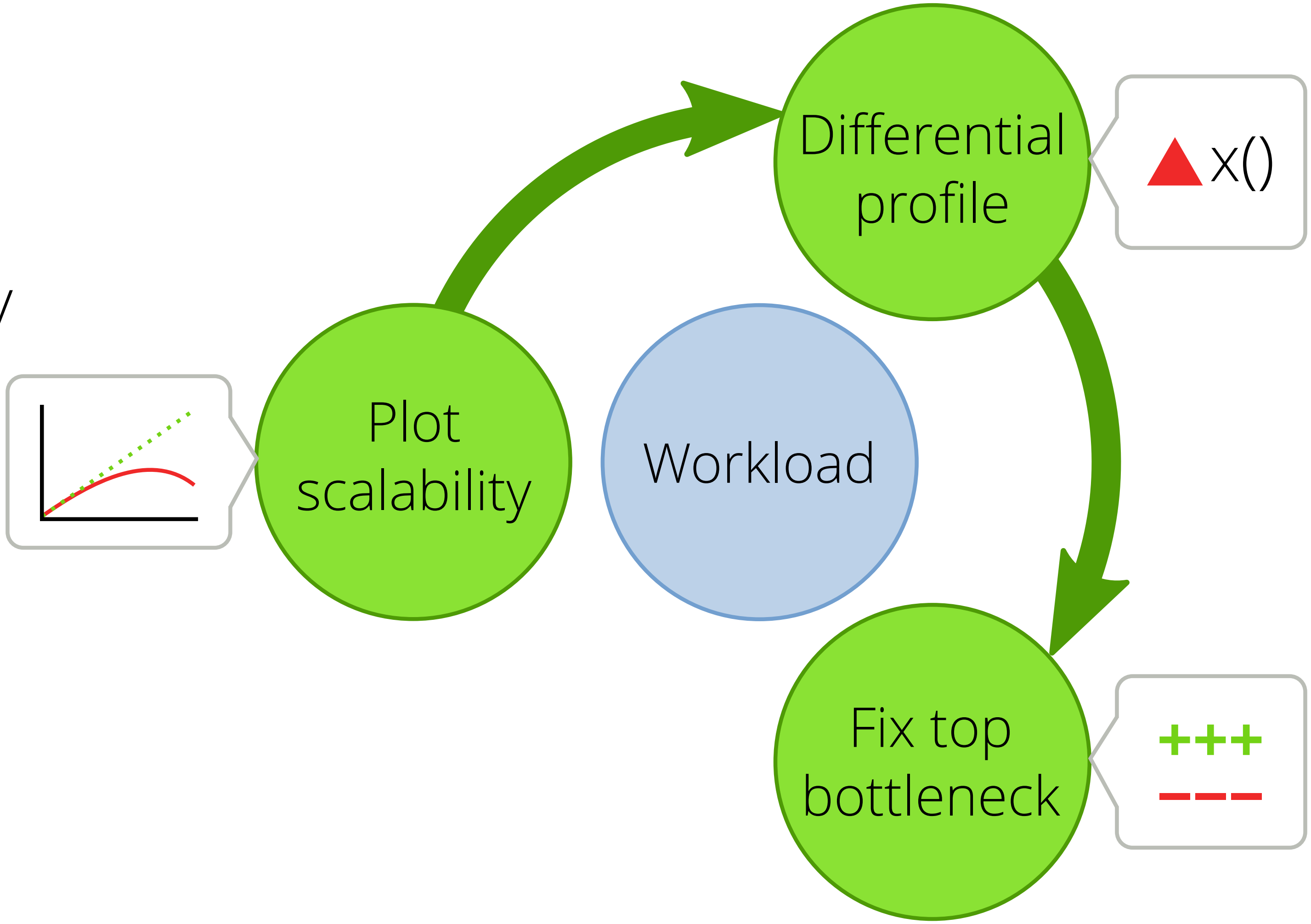
2009 — Corey, OSDI '08

2010 — Linux scalability, OSDI '10

2011

2012 — Bonsai VM, ASPLOS '12

2013 — RadixVM, EuroSys '13



Current approach to scalable software development

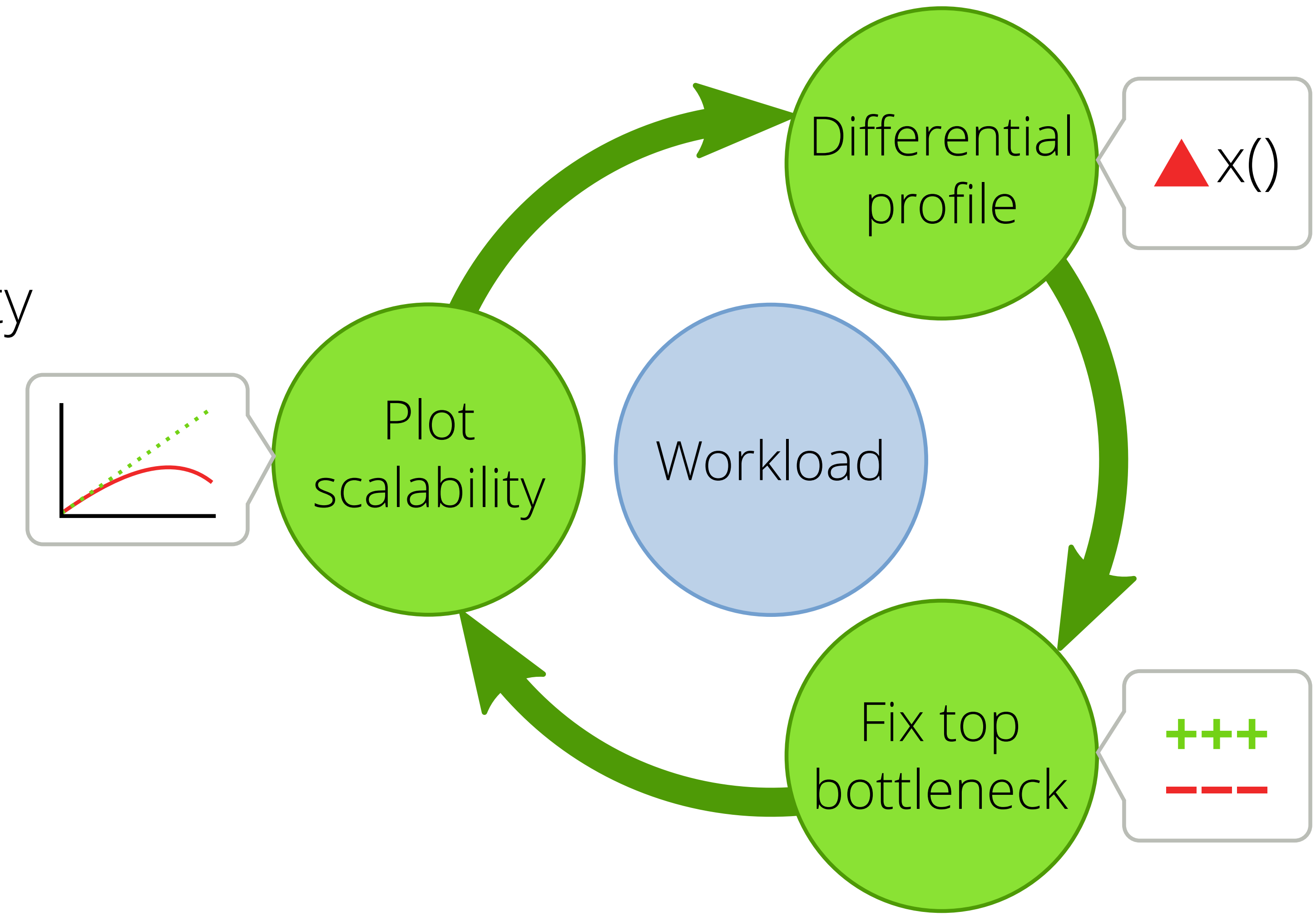
2009 — Corey, OSDI '08

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

- 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

Successful in practice because it focuses developer effort

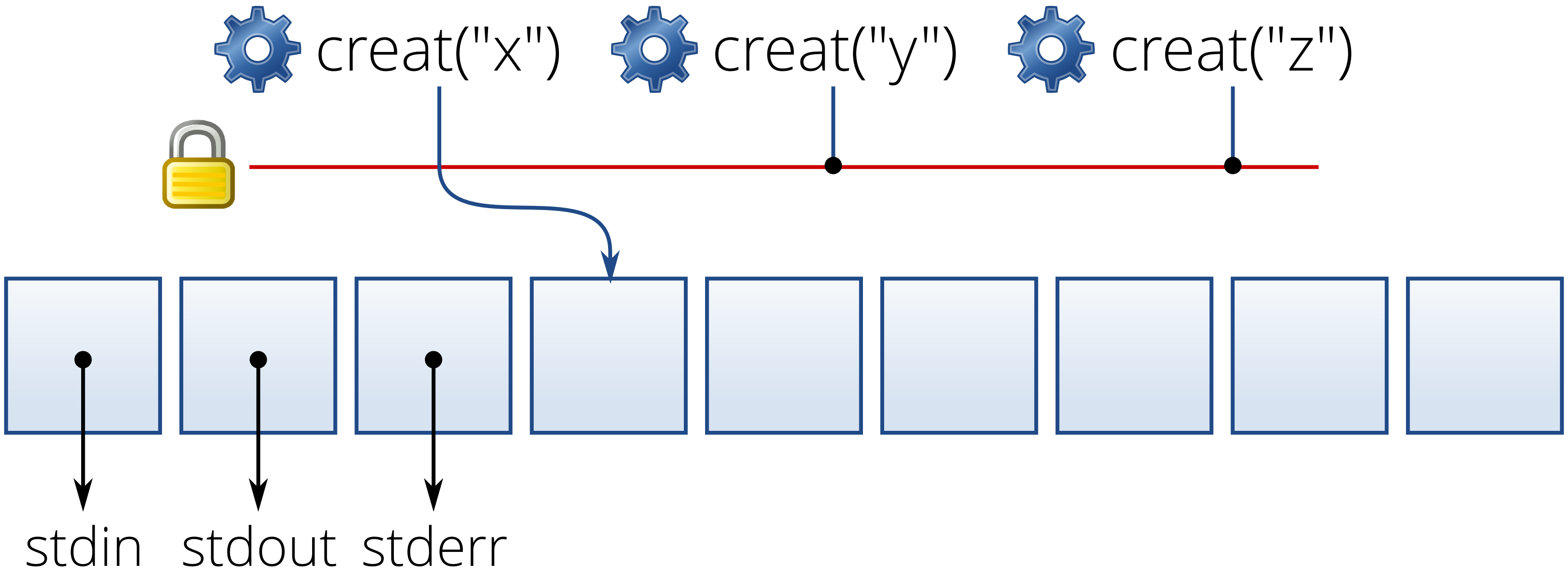
Disadvantages

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

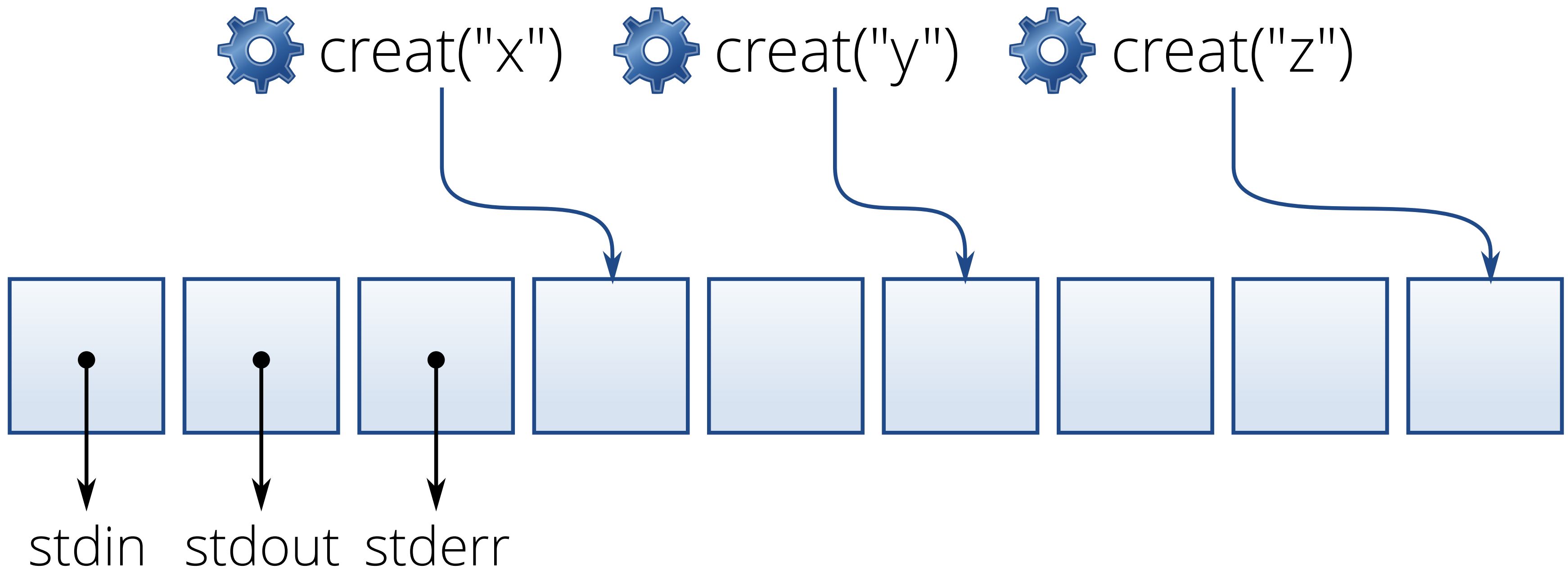
Interface scalability example

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

Interface scalability example

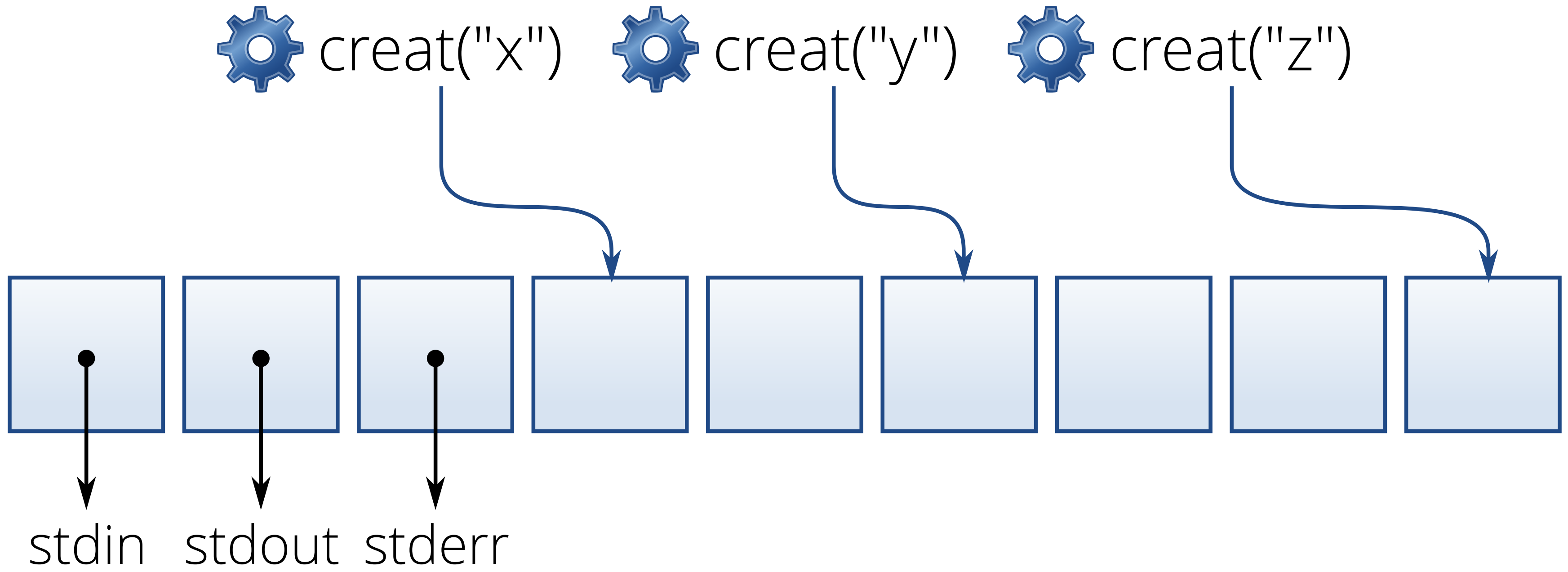


Interface scalability example



Solution: Change the interface?

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.**

	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

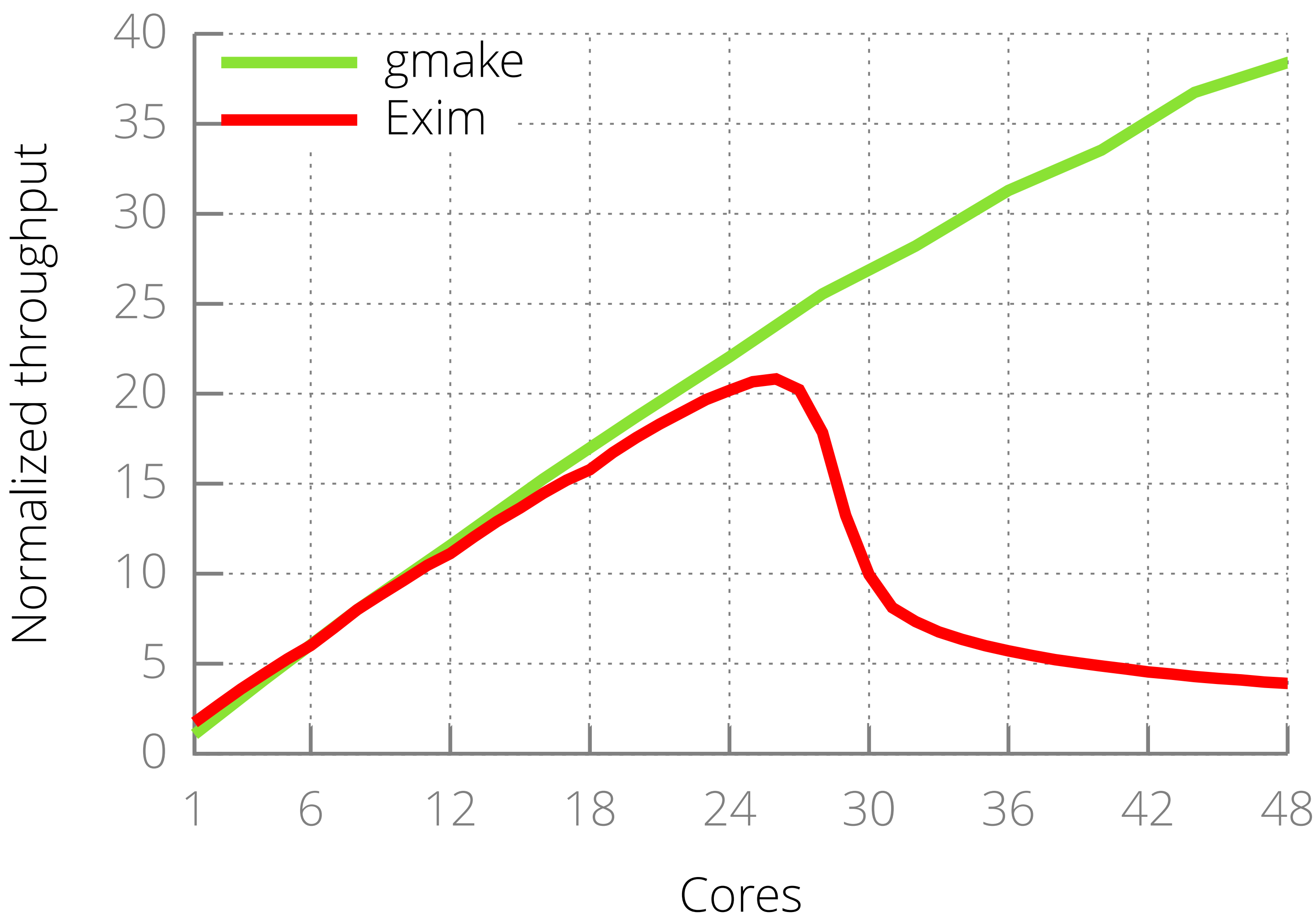
Defining the rule

- Definition of scalability
- Intuition
- Formalization

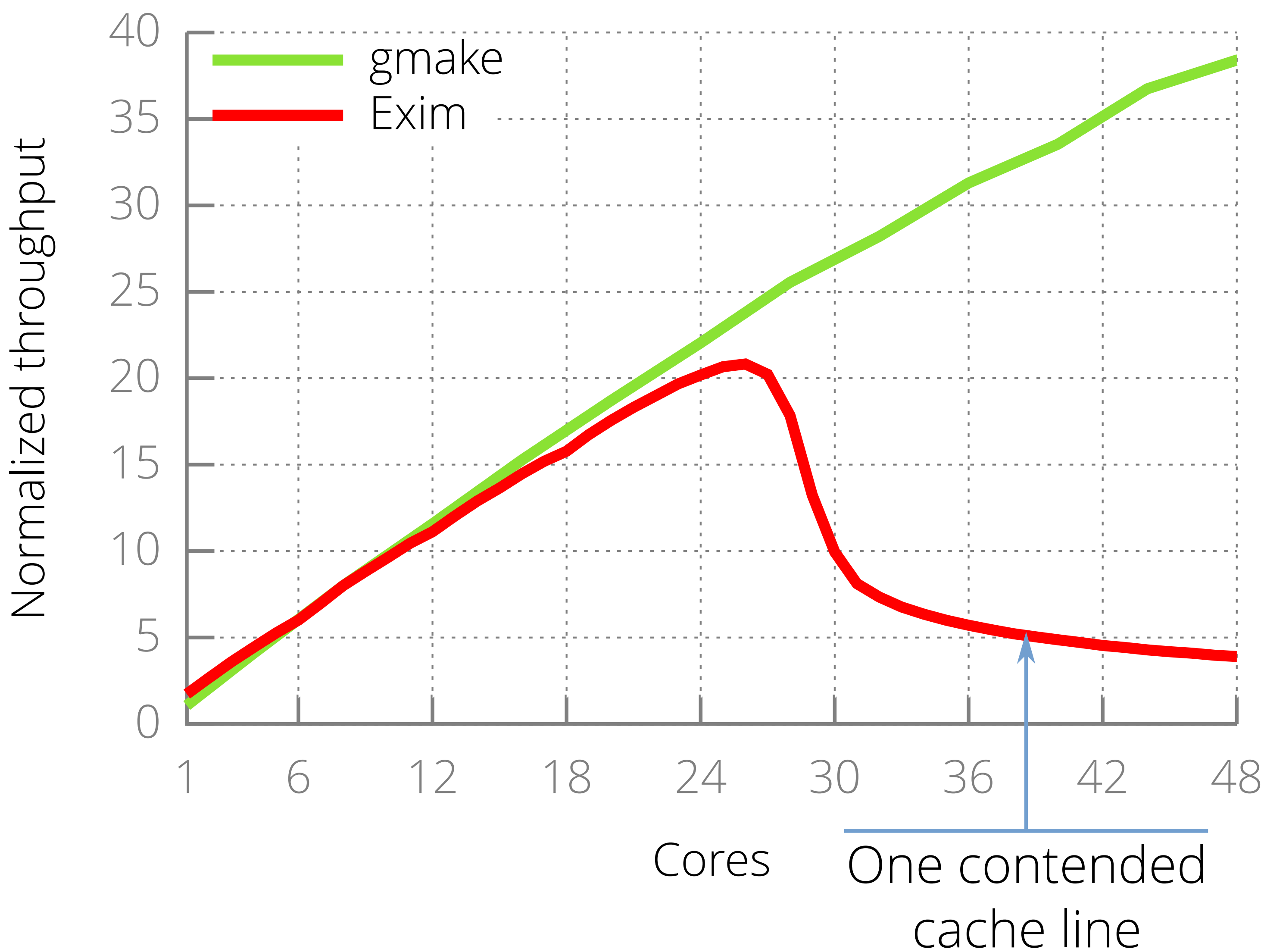
Applying the rule

- Commuter
- Evaluation

A scalability bottleneck

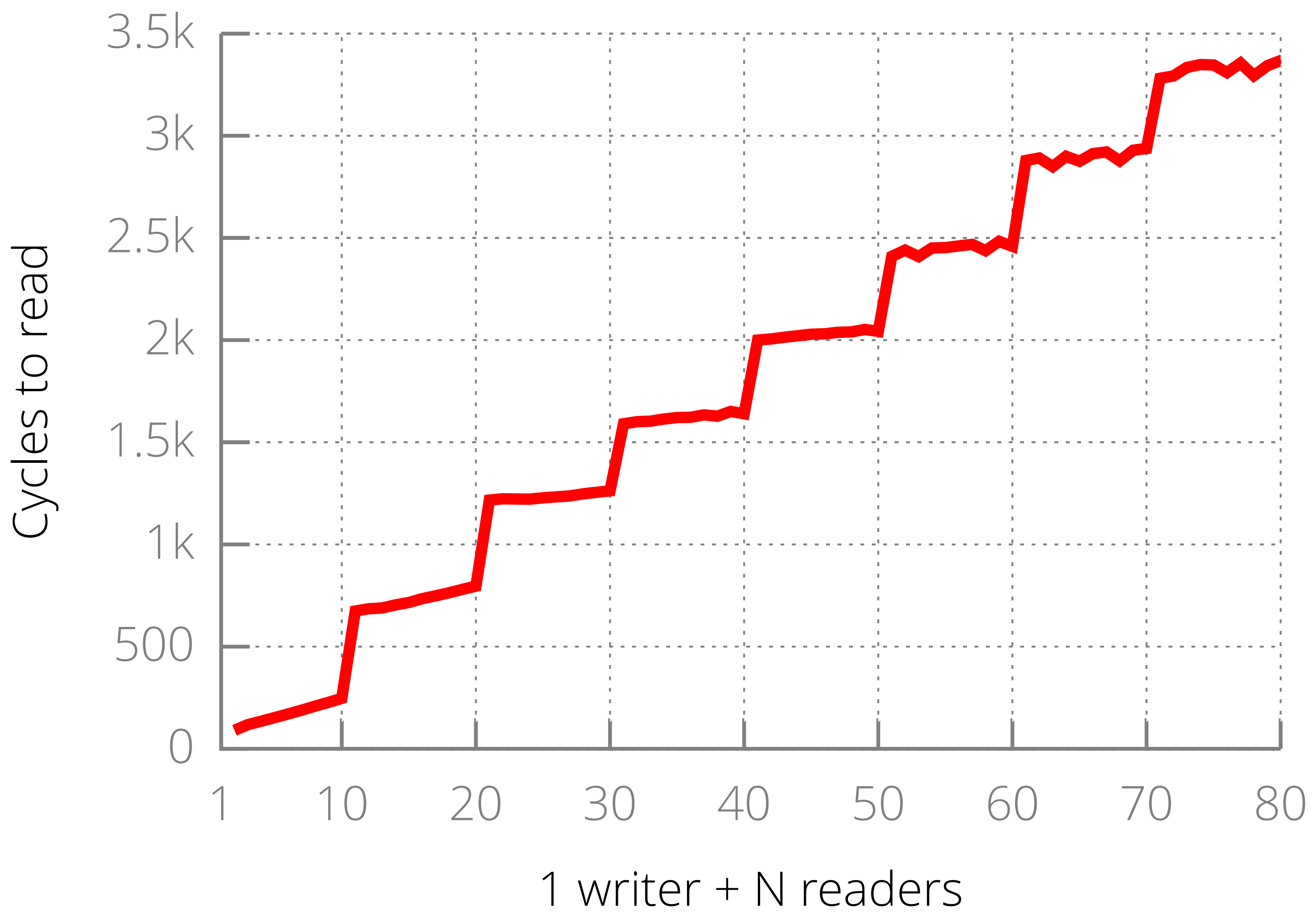


A scalability bottleneck

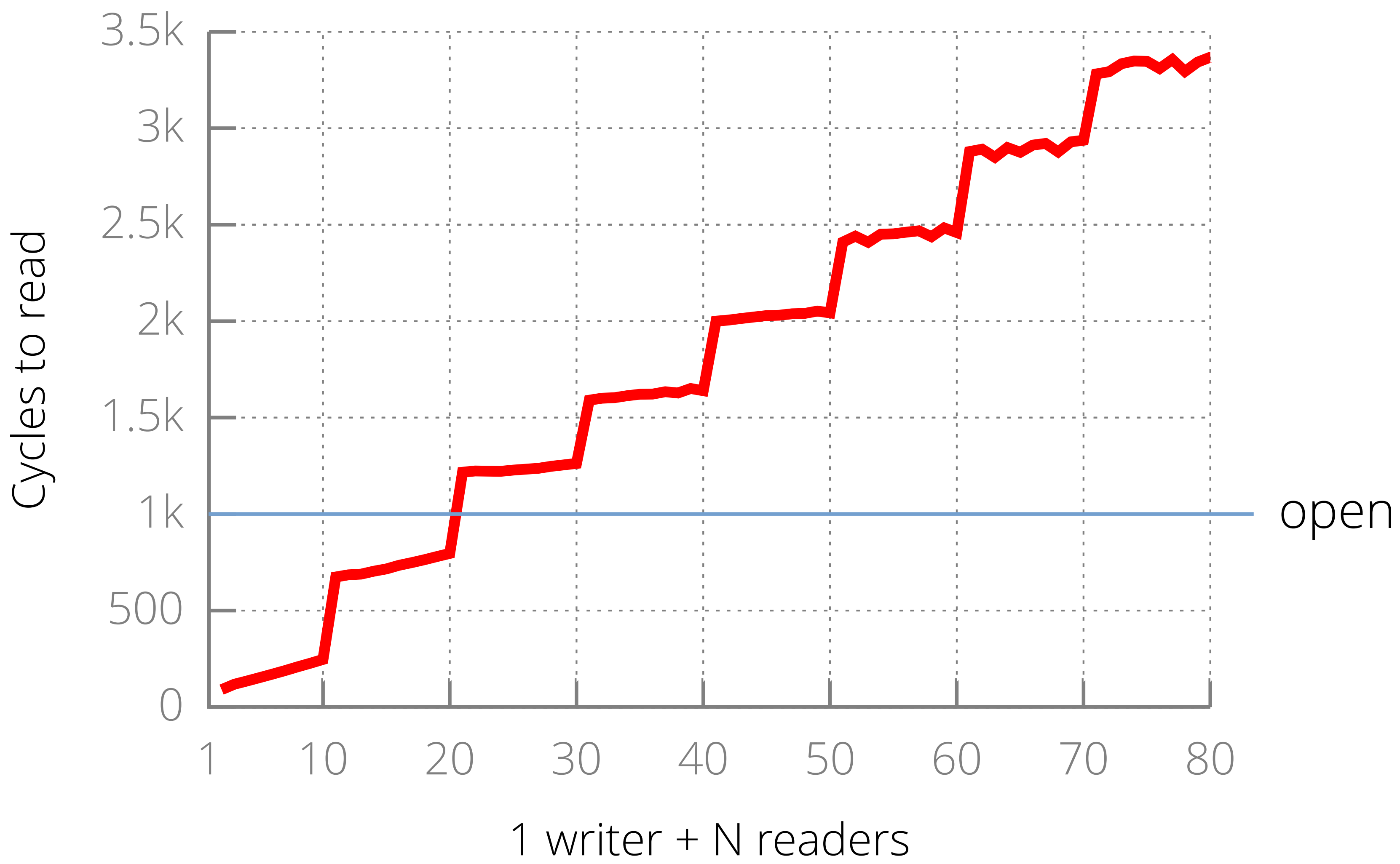


A single contended cache line can wreck scalability

Cost of a contended cache line



Cost of a contended cache line



What scales on today's multicores?

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

What scales on today's multicores?

Core *X*

Core Y		W	R	-
	W	✗	✗	✓
	R	✗	✓	✓
	-	✓	✓	-

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

Good approximation of current hardware.

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

Example: Reference counter

T1	iszero() → F				
T2		iszero() → F			
T3			dec() → 2		
T4				dec() → 1	
T5					dec() → 0

Example: Reference counter

T1 iszero() → F

T2 iszero() → F

T3 dec() → 2

T4 dec() → 1

T5 dec() → 0

R1

Example: Reference counter

T1	iszero() → F				
T2		iszero() → F			
T3			dec() → 2		
T4				dec() → 1	
T5					dec() → 0

R1

✓ R1 commutes; conflict-free implementation: shared counter

Example: Reference counter

T1 iszero() → F

T2 iszero() → F

T3 dec() → 2

T4 dec() → 1

T5 dec() → 0

R1 R2

✓ R1 commutes; conflict-free implementation: shared counter

Example: Reference counter

T1 iszero() → F

T2 iszero() → F

T3 dec() → 2

T4 dec() → 1

T5 dec() → 0

R1 R2

✓ R1 commutes; conflict-free implementation: shared counter

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

Example: Reference counter

T1 iszero() → F

T2 iszero() → F

T3 dec() → ok

T4 dec() → ok

T5 dec() → ok

R1 R2'

✓ R1 commutes; conflict-free implementation: shared counter

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

Example: Reference counter

T1	iszero() → F				
T2		iszero() → F			
T3			dec() → ok		
T4				dec() → ok	
T5					dec() → ok
	R1	R1	R2'	R2'	R2'

✓ 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

T1 iszero() → F

T2 iszero() → F

T3 dec() → ok

T4 dec() → ok

T5 dec() → ok

R1 R2'

R3

- ✓ 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

T1 iszero() → F

T2 iszero() → F

T3 dec() → ok

T4 dec() → ok

T5 dec() → ok

R1 R2'

R3

✓ 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

Example: Reference counter

T1 iszero() → F

T2 iszero() → F

T3 dec() → ok

T4 dec() → ok

T5 dec() → ok

R1 R2'

R3

✓ 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

- History
- Reordering
- Commutativity

Formal scalable commutativty rule

Histories capture state and arguments

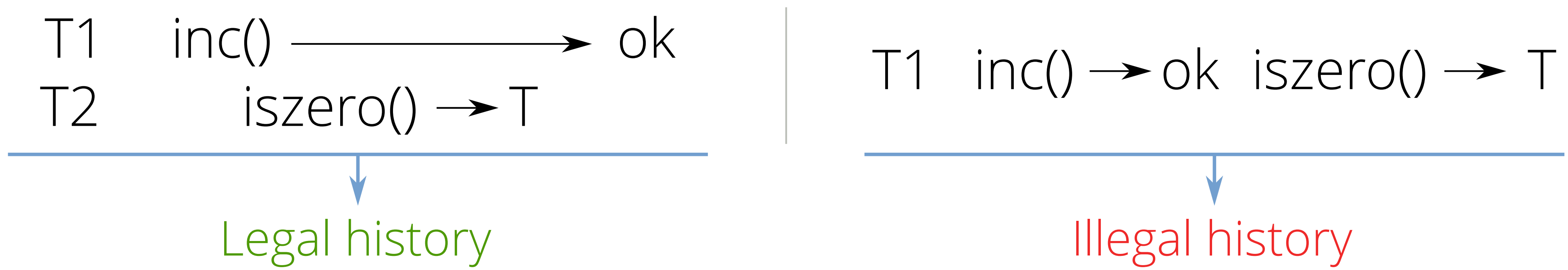
A **history** *H* is a sequence of invocations and responses on threads.

T1 inc() ——————→ ok
T2 iszero() → T

T1 inc() → ok iszero() → T

Histories capture state and arguments

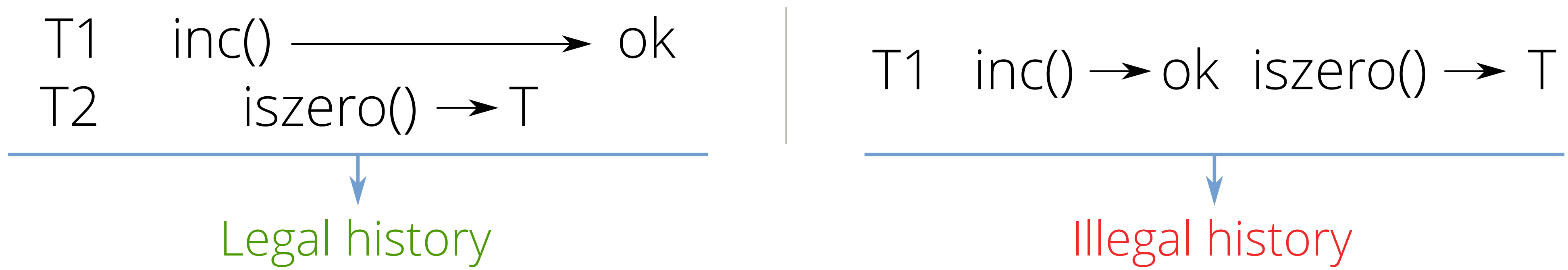
A **history** *H* is a sequence of invocations and responses on threads.



A **specification** $\mathscr{S}$ defines an interface. $\mathscr{S}$ is the set of **legal** histories giving the allowed behavior of an interface. [Herlihy & Wing, '90]

Histories capture state and arguments

A **history** *H* is a sequence of invocations and responses on threads.



A **specification** $\mathscr{S}$ defines an interface. $\mathscr{S}$ is the set of **legal** histories giving the allowed behavior of an interface. [Herlihy & Wing, '90]

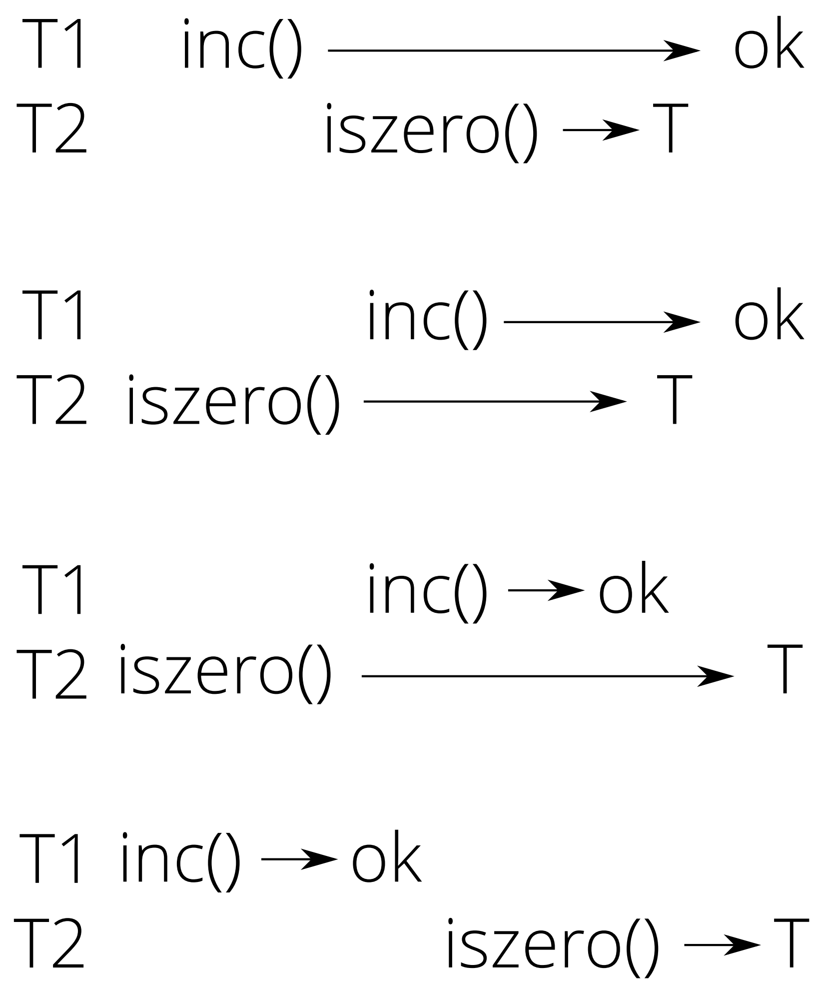
Lets us talk about interfaces, arguments, and state without specifying an implementation or a state representation.

Reorderings

A **reordering** H' is a permutation of H that maintains operation order for each individual thread ($H|t = H'|t$ for all t).

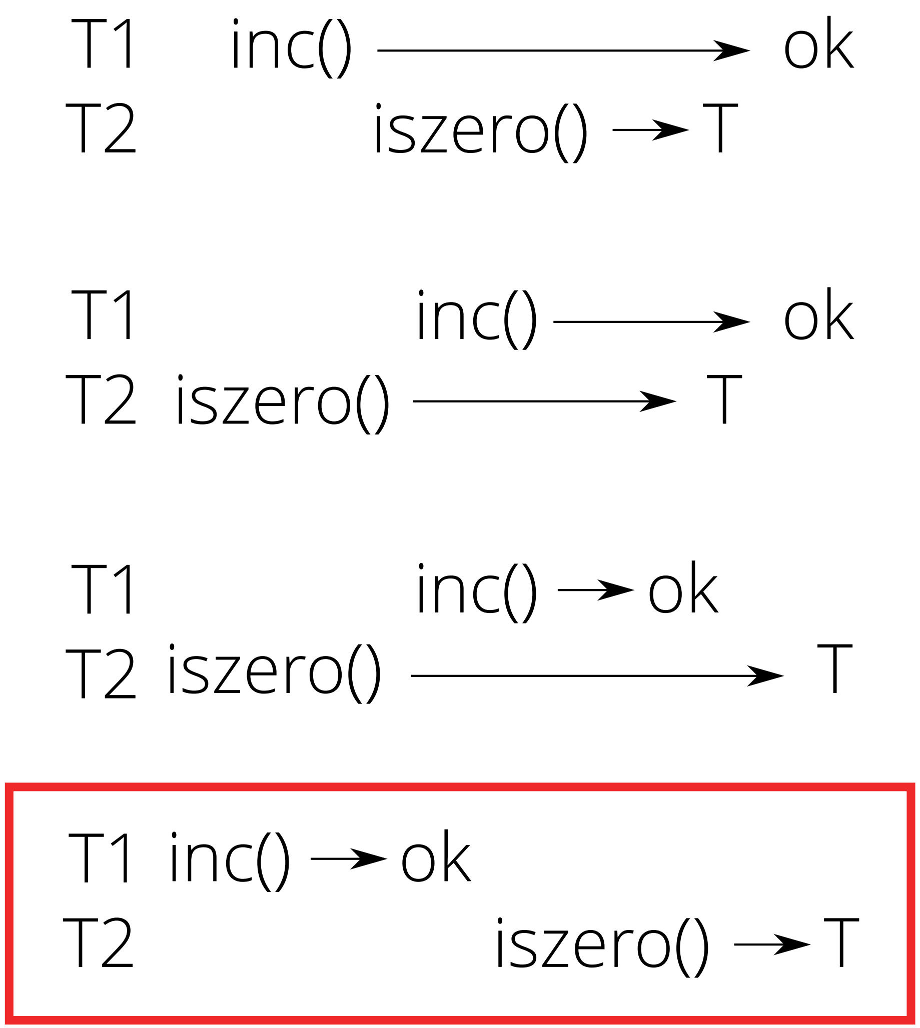
Reorderings

A **reordering** *H′* is a permutation of *H* that maintains operation order for each individual thread ($H|t = H'|t$ for all t).



Reorderings

A **reordering** *H′* is a permutation of *H* that maintains operation order for each individual thread ($H|t = H'|t$ for all t).



Commutativity

A region *Y* of a legal history *XY* **SIM-commutes** if every reordering *Y'* of *Y* also yields a legal history and every legal extension *Z* of *XY* is also a legal extension of *XY'*.

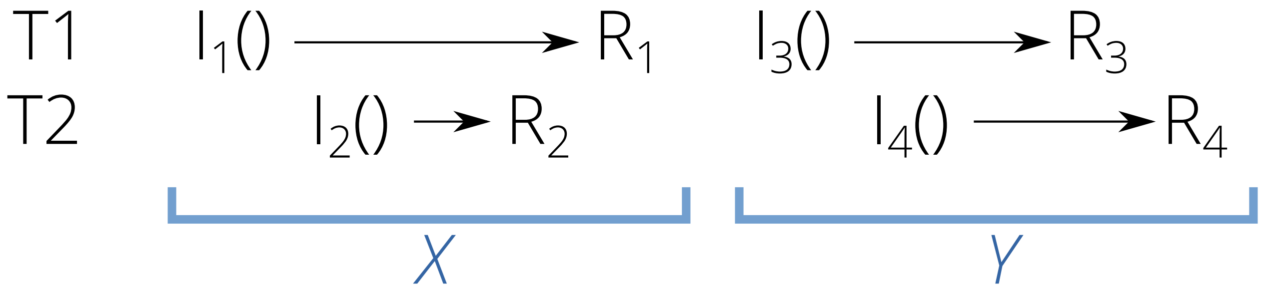
(And this must be true for every prefix of every reordering of *Y*.)

Commutativity

A region *Y* of a legal history *XY* **SIM-commutes** if every reordering *Y'* of *Y* also yields a legal history and every legal extension *Z* of *XY* is also a legal extension of *XY'*.

(And this must be true for every prefix of every reordering of *Y*.)

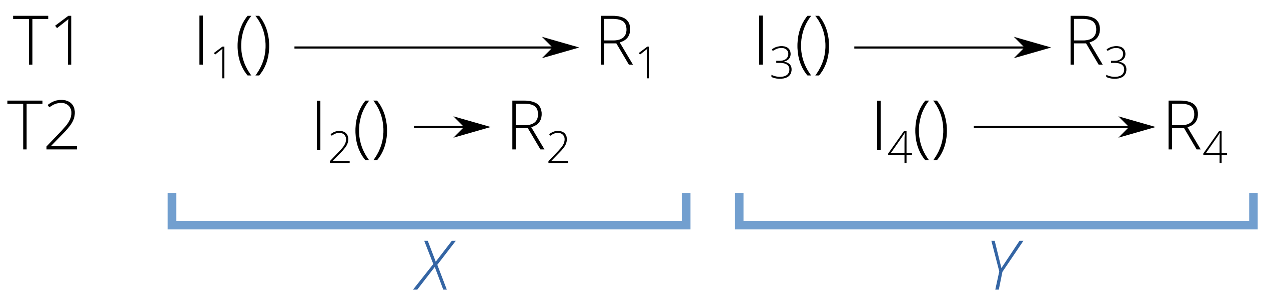
Commutativity



A region *Y* of a legal history *XY* **SIM-commutes** if every reordering *Y′* of *Y* also yields a legal history and every legal extension *Z* of *XY* is also a legal extension of *XY′*.

(And this must be true for every prefix of every reordering of *Y*.)

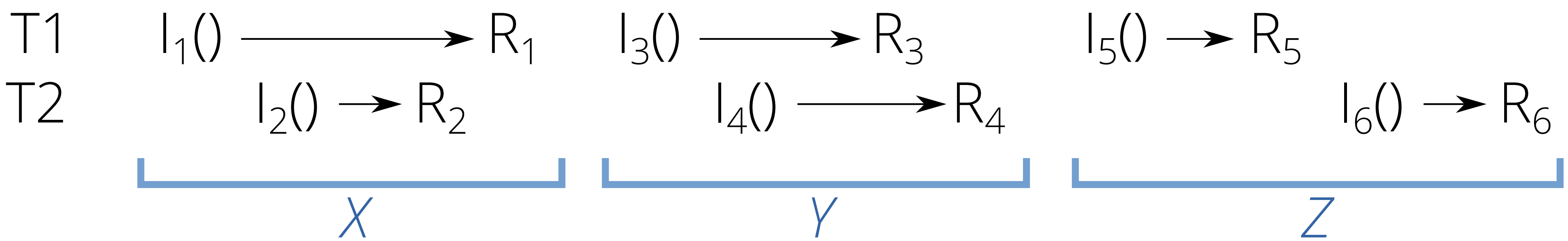
Commutativity



A region *Y* of a legal history *XY* **SIM-commutes** if every reordering *Y'* of *Y* also yields a legal history and every legal extension *Z* of *XY* is also a legal extension of *XY'*.

(And this must be true for every prefix of every reordering of *Y*.)

Commutativity



A region *Y* of a legal history *XY* **SIM-commutes** if every reordering *Y'* of *Y* also yields a legal history and every legal extension *Z* of *XY* is also a legal extension of *XY'*.

(And this must be true for every prefix of every reordering of *Y*.)

The formal scalable commutativity rule

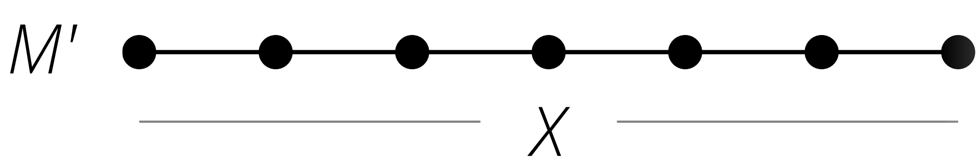
Let $\mathscr{S}$ be a specification with a reference implementation M. Consider a history XY where Y commutes in XY and M can generate XY.

There exists a correct implementation of $\mathscr{S}$ whose execution of XY is conflict-free in the commutative region Y.

The formal scalable commutativity rule

Let $\mathscr{S}$ be a specification with a reference implementation M.
Consider a history XY where Y commutes in XY and M can generate XY.

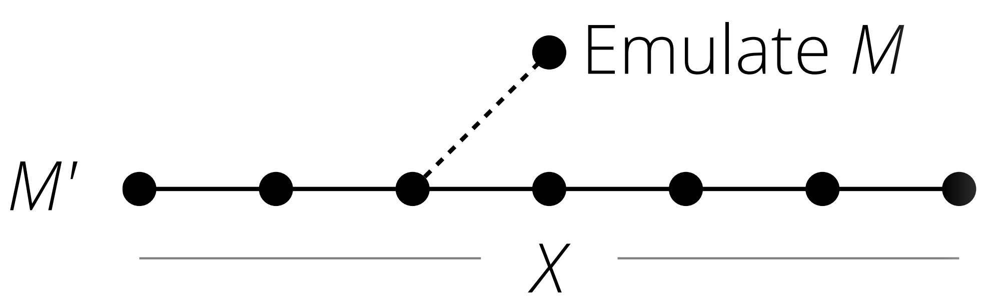
There exists a correct implementation of $\mathscr{S}$ whose execution of XY is conflict-free in the commutative region Y.



The formal scalable commutativity rule

Let $\mathscr{S}$ be a specification with a reference implementation *M*. Consider a history *XY* where *Y* commutes in *XY* and *M* can generate *XY*.

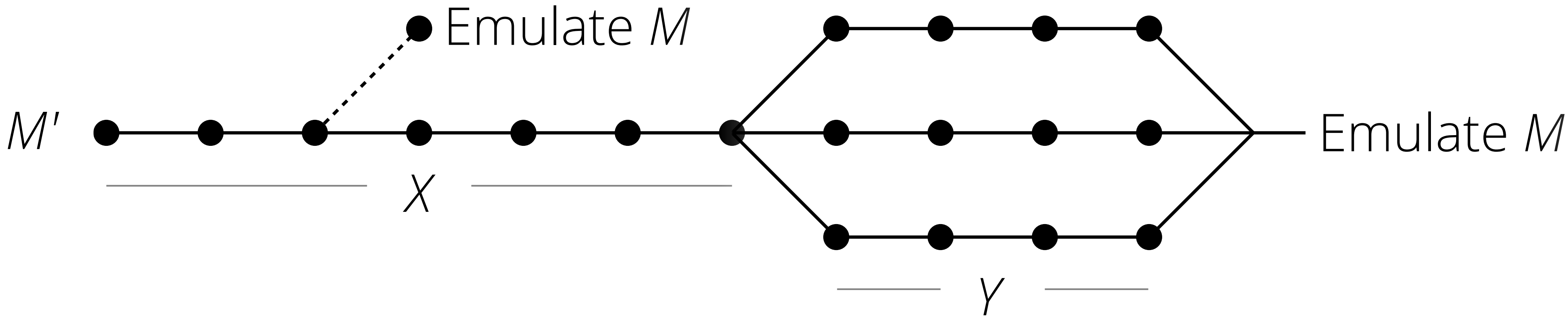
There exists a correct implementation of $\mathscr{S}$ whose execution of *XY* is conflict-free in the commutative region *Y*.



The formal scalable commutativity rule

Let $\mathscr{S}$ be a specification with a reference implementation *M*.
Consider a history *XY* where *Y* commutes in *XY* and *M* can generate *XY*.

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

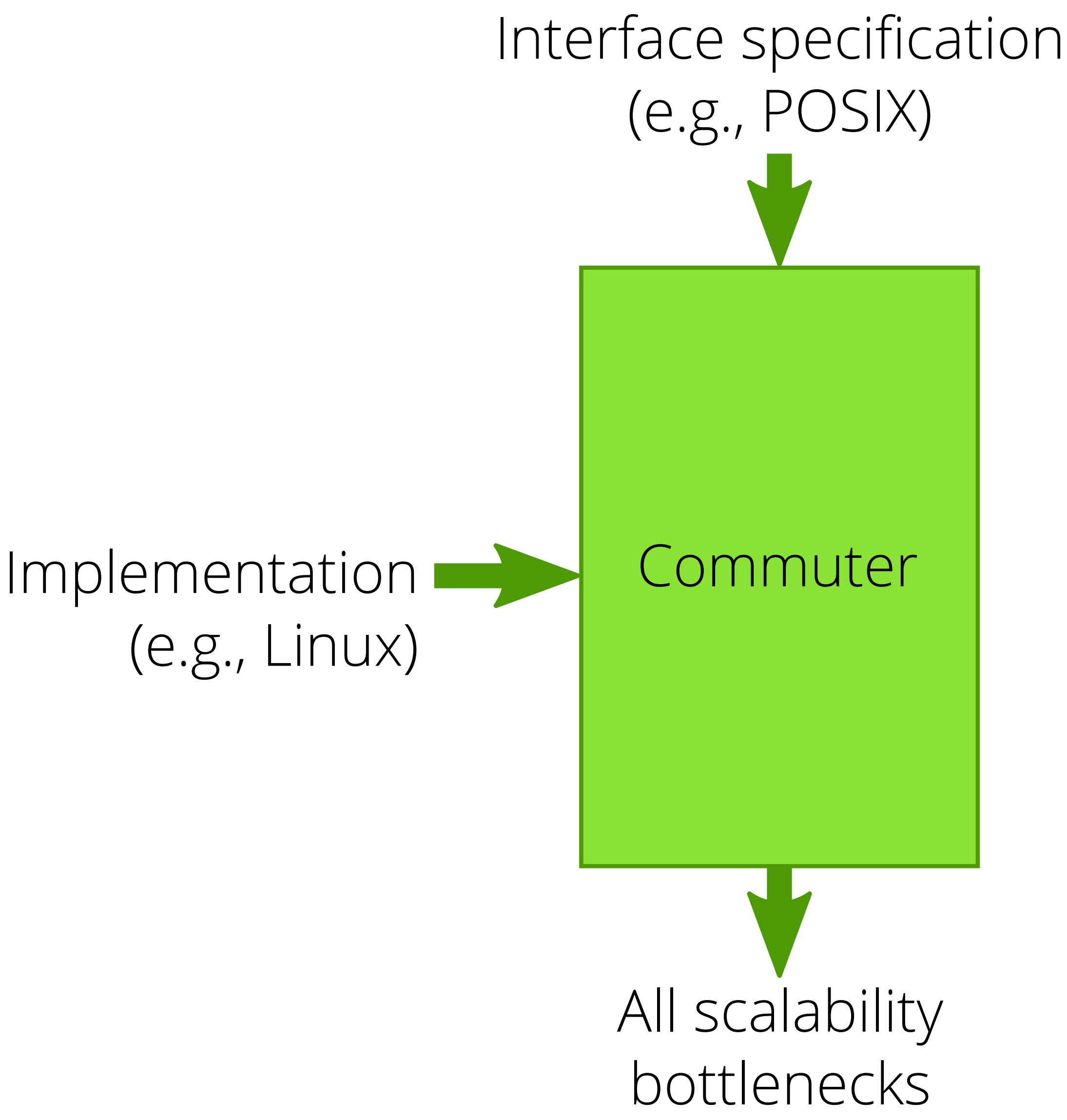


Applying the rule to real systems

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

Commutativity conditions

```
        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

Important to have discriminating commutativity conditions

- ∀states, rename almost never commutes
- More commutative cases ⇒ more opportunities to scale
- Captures more operations applications actually do

Commutativity conditions

```
        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

Important to have discriminating commutativity conditions

- ∀states, rename almost never commutes
- More commutative cases ⇒ more opportunities to scale
- Captures more operations applications actually do

Commutativity conditions

```
        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 $a \neq c$
- a and c are hard links to the same inode, $a \neq c$, and $b = d$

Important to have discriminating commutativity conditions

- ∀states, rename almost never commutes
- More commutative cases ⇒ more opportunities to scale
- Captures more operations applications actually do

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 $a \neq c$
- a and c are hard links to the same inode, $a \neq c$, and $b = d$

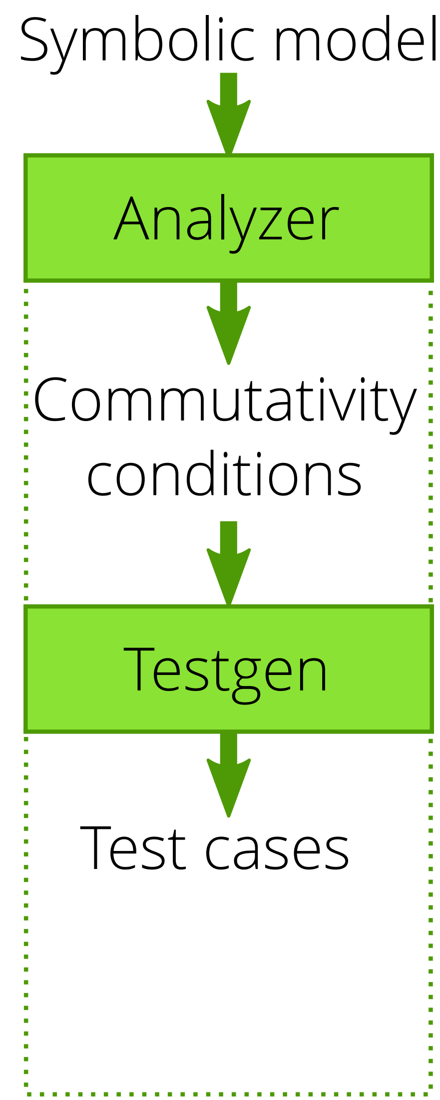
Test cases

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

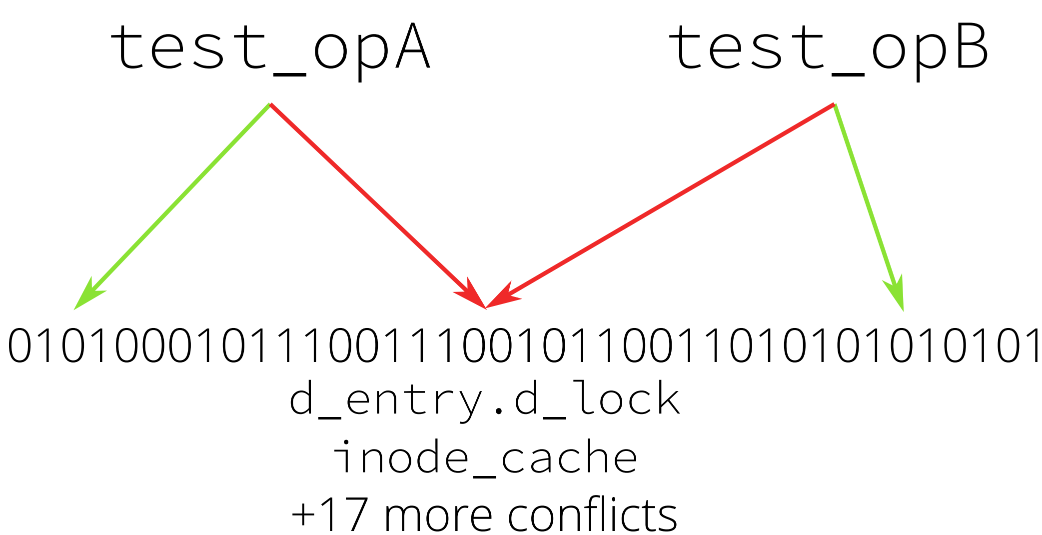
```
void setup() {
    close(creat("f0", 0666));
    close(creat("f2", 0666));
}
void test_opA() { rename("f0", "f1"); }
void test_opB() { rename("f2", "f3"); }
```

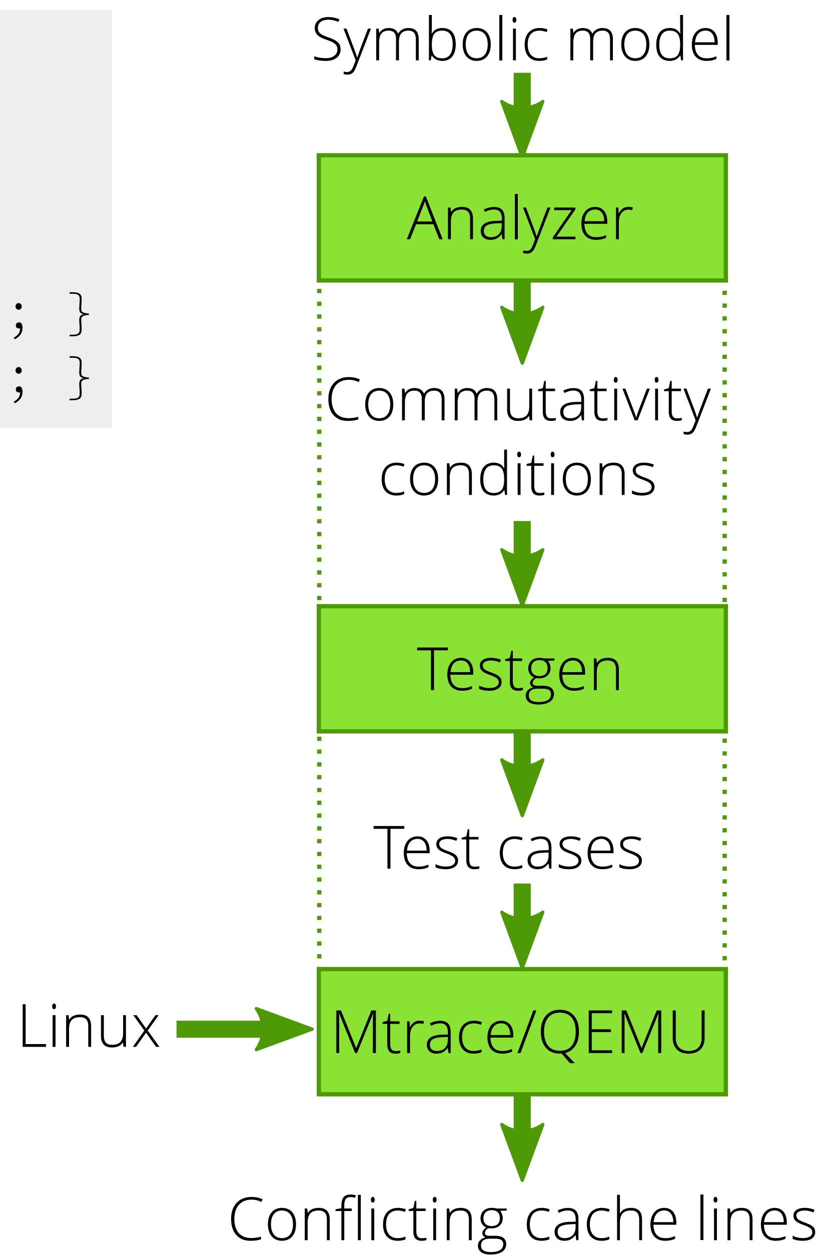
+ 26 more



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"); }
```



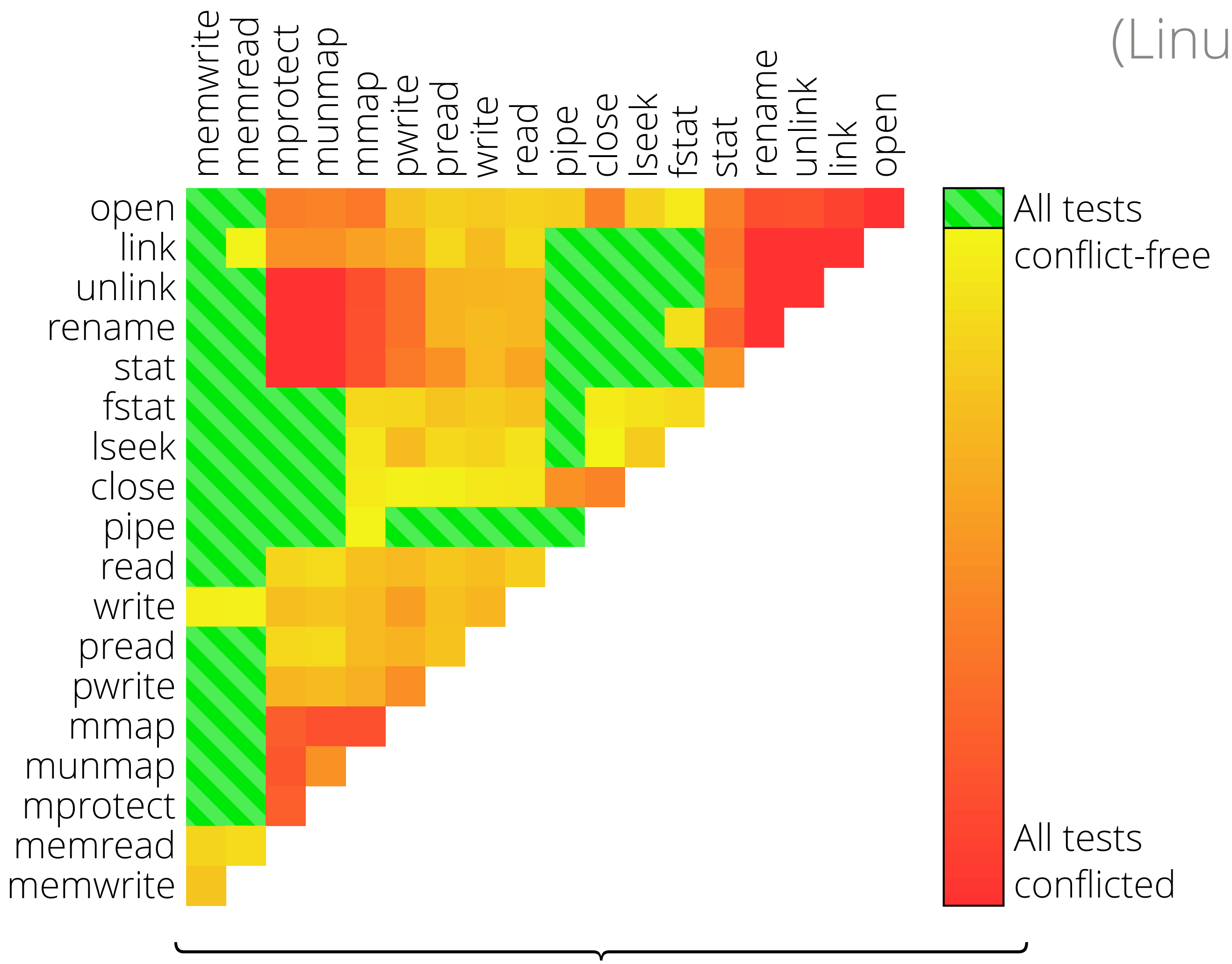


Evaluation

Does the rule help build scalable systems?

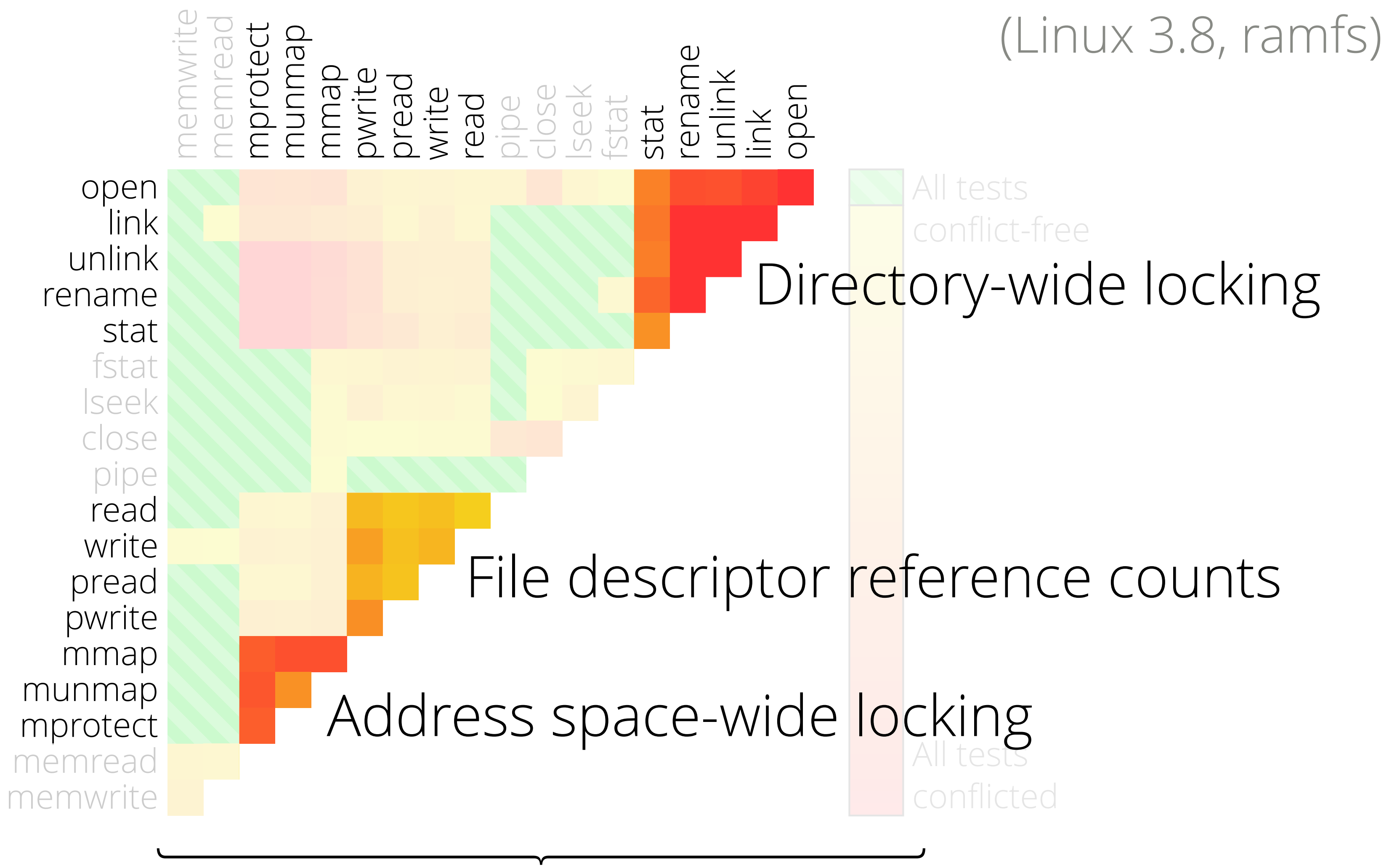
Commuter finds non-scalable cases in Linux

(Linux 3.8, ramfs)



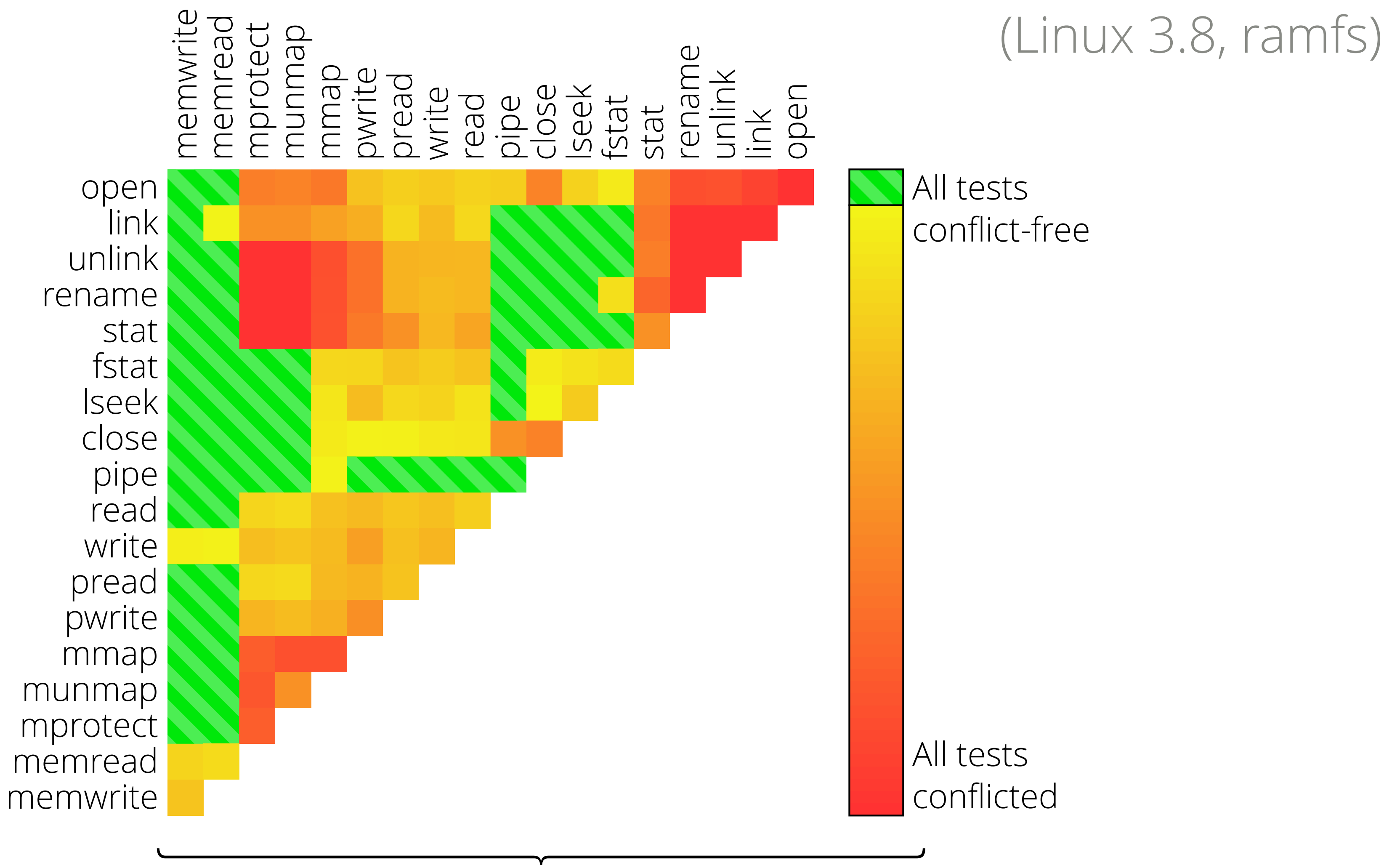
13,664 total test cases
68% are conflict-free

Commuter finds non-scalable cases in Linux



13,664 total test cases
68% are conflict-free

Commuter finds non-scalable cases in Linux



13,664 total test cases

68% are conflict-free

Many potential future bottlenecks

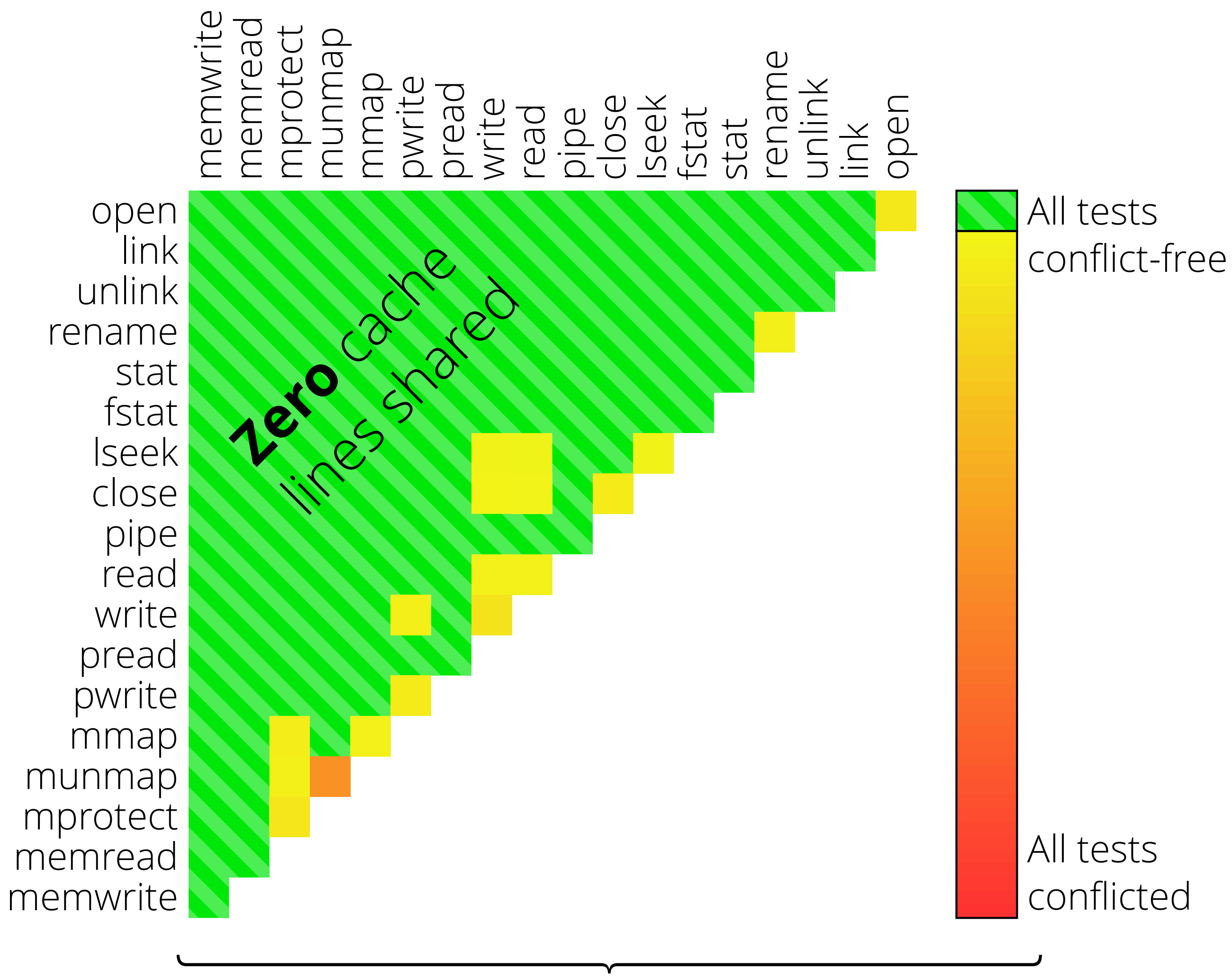
sv6: A scalable OS

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

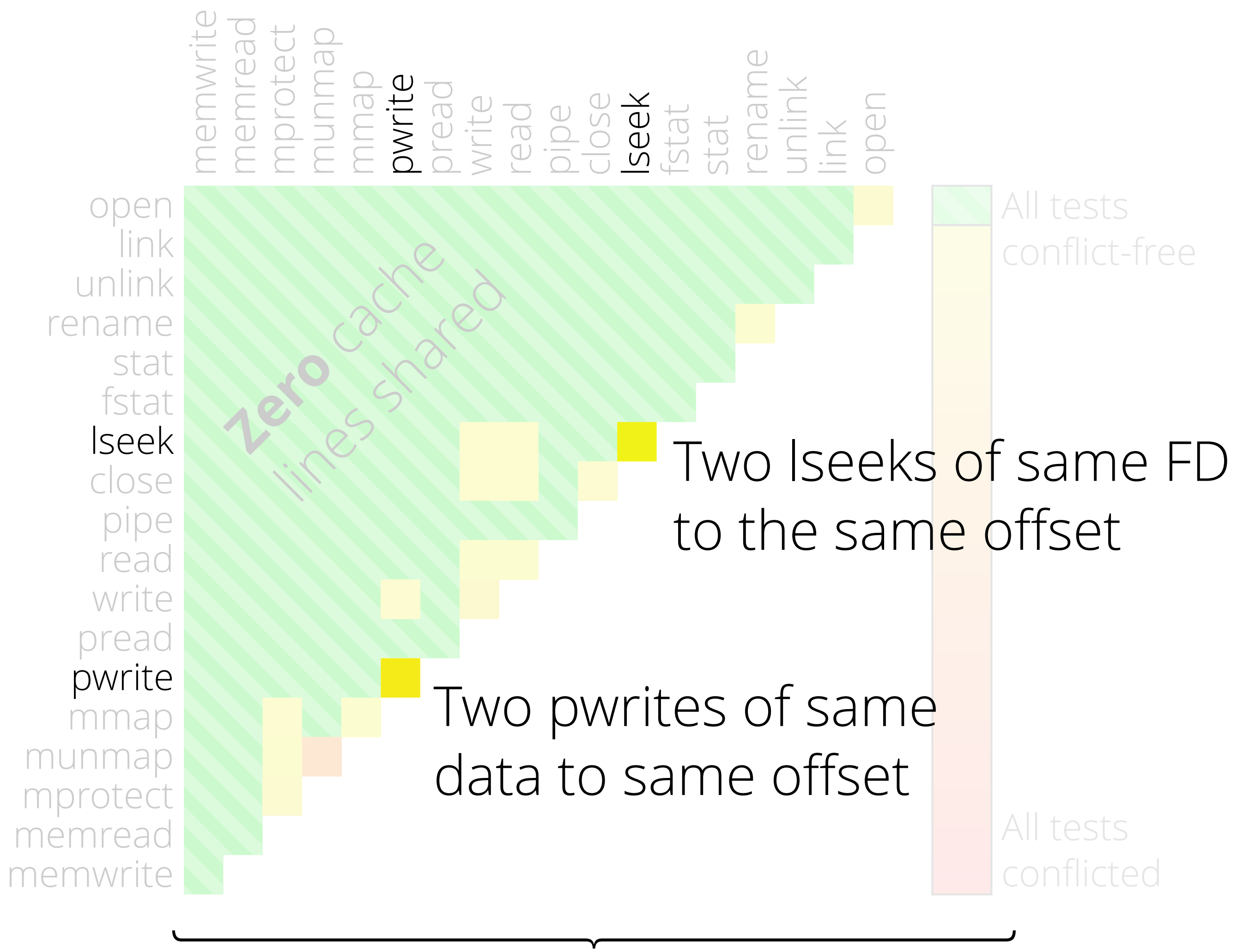


13,664 total test cases

99% are conflict-free

Remaining 1% are mostly "idempotent updates"

Commutative operations can be made to scale



13,664 total test cases

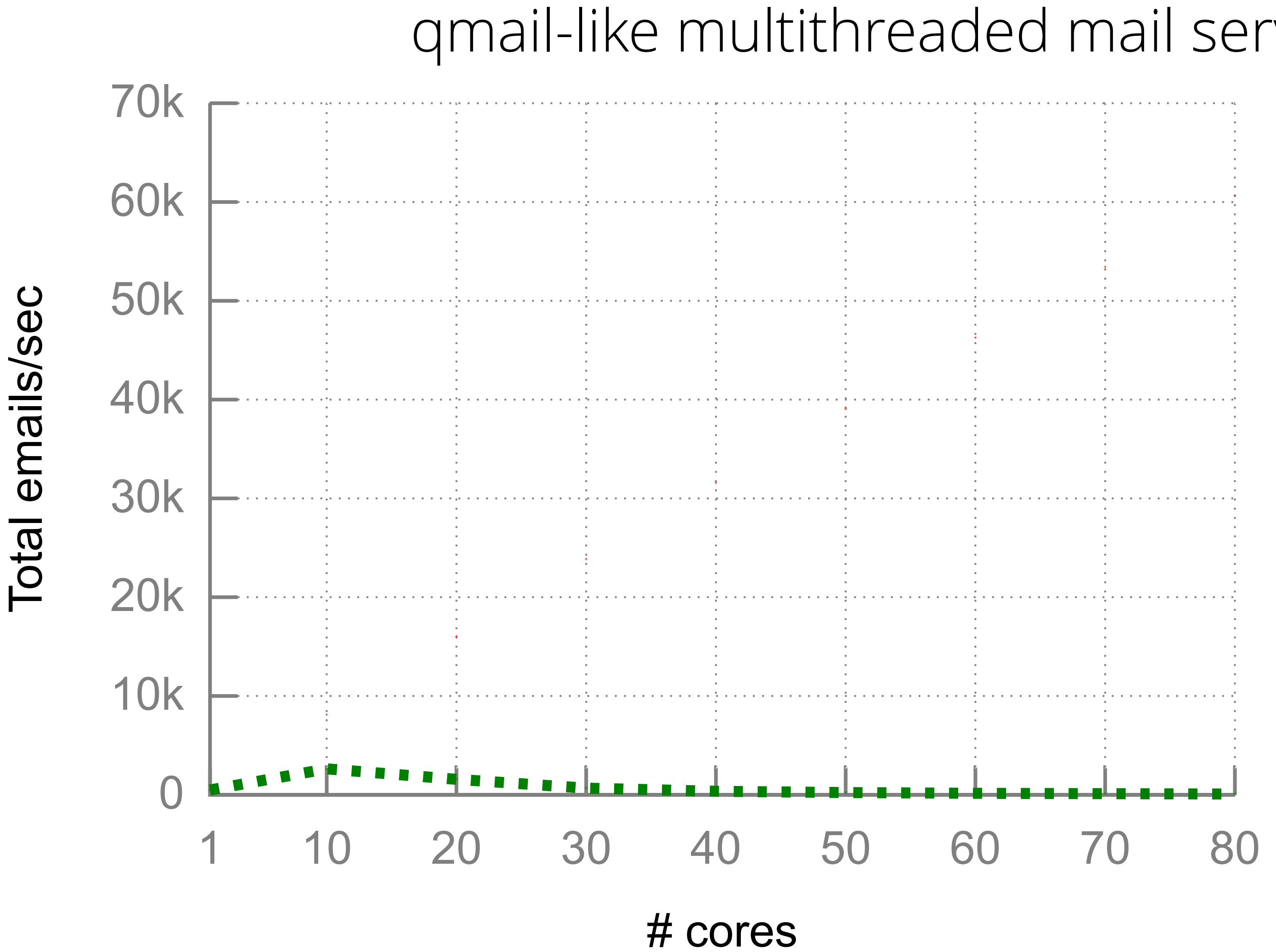
99% are conflict-free

Remaining 1% are mostly "idempotent updates"

Refining POSIX with the rule

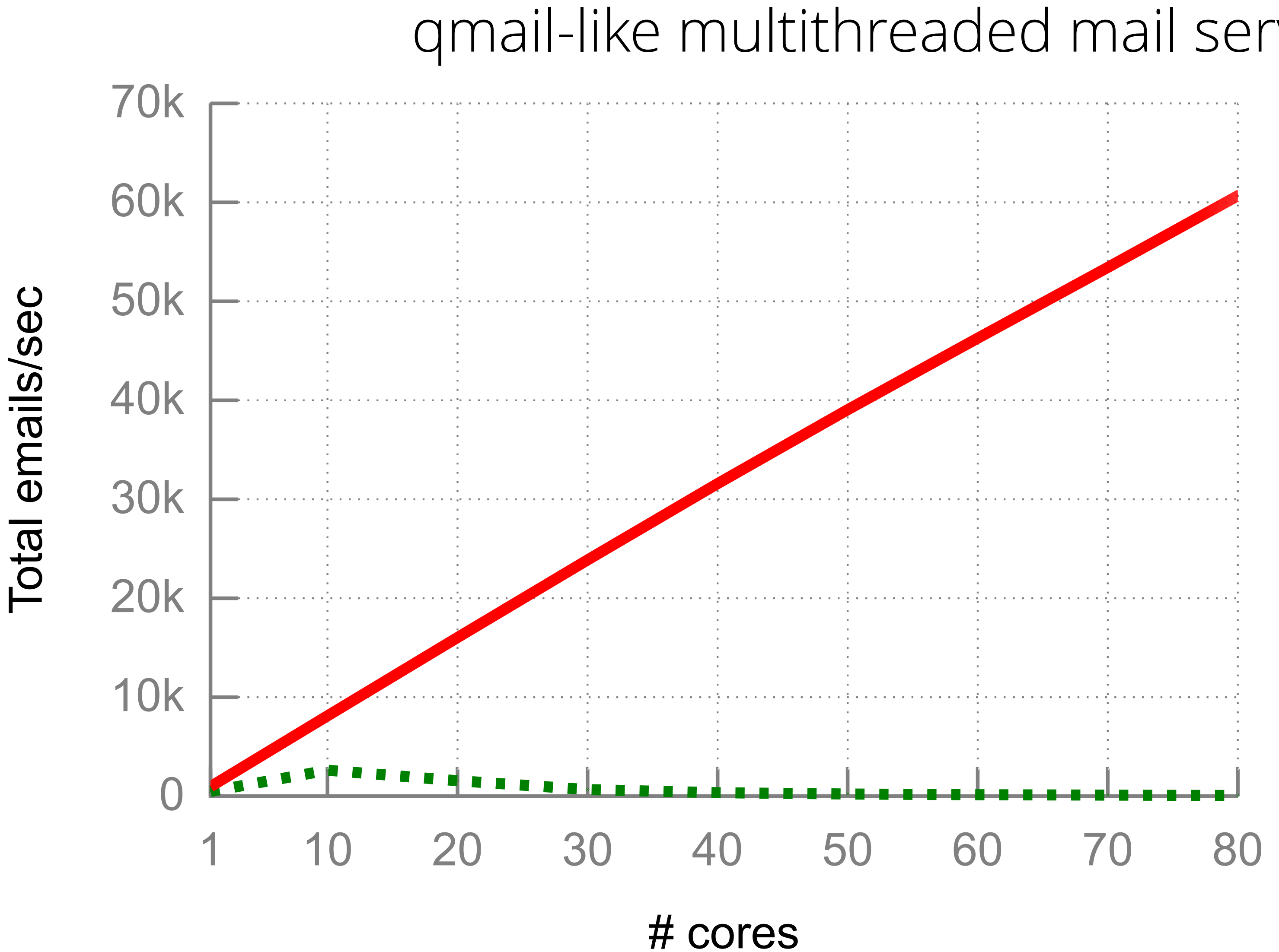
- Lowest FD versus any FD
- stat versus xstat
- Unordered sockets
- Delayed munmap
- fork+exec versus posix_spawn

Commutative operations matter to app scalabiliy



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

Commutative operations matter to app scalabiliy



Commutative APIs:
Any FD
Unordered sockets
posix_spawn

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

Related work

Commutativity and concurrency

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

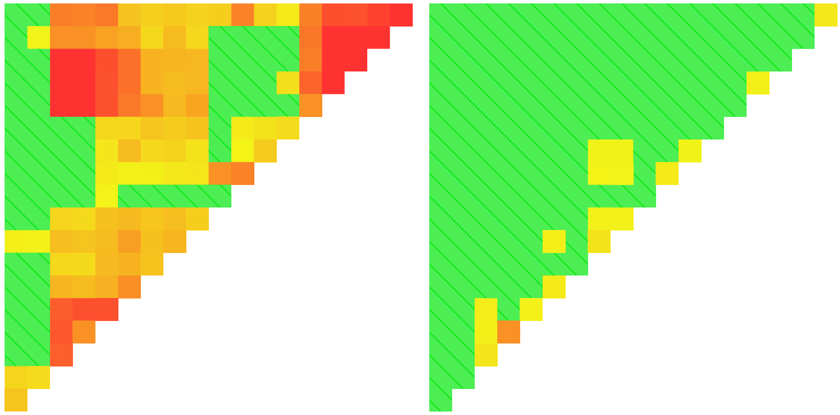
Laws of Order [Attiya '11]

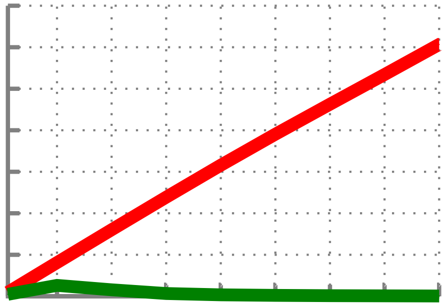
Disjoint-access parallelism [Israeli '94]
Scalable locks [MCS '91]
Scalable reference counting [Ellen '07, Corbet '10]

Conclusion

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

- Design
- Implement
- Test

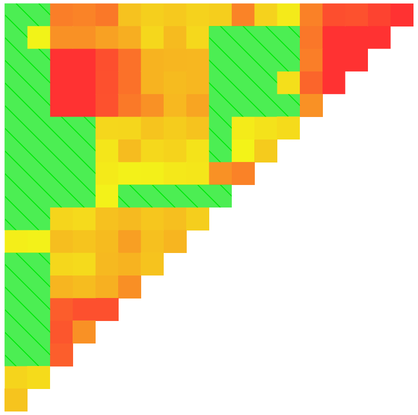


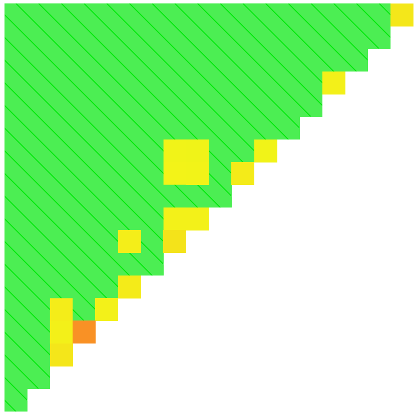


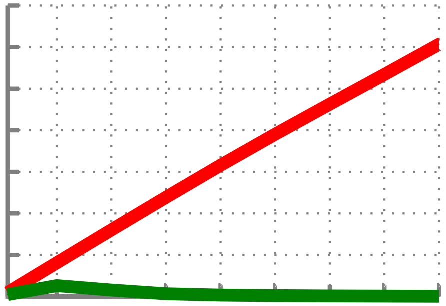
Conclusion

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

Design
Implement
Test







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