The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors

Phil Gibbons

15-712 F15

Lecture 18

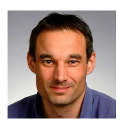
Today's Reminders

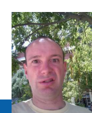
- **Project proposals due Friday midnight**
  - Email Kevin and me
- **No Class Monday**
- **Midterm on Wednesday**
  - Will cover assigned papers for 9/11 through 10/21
  - Understand high-level concepts & compare ideas/approaches across papers

The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors

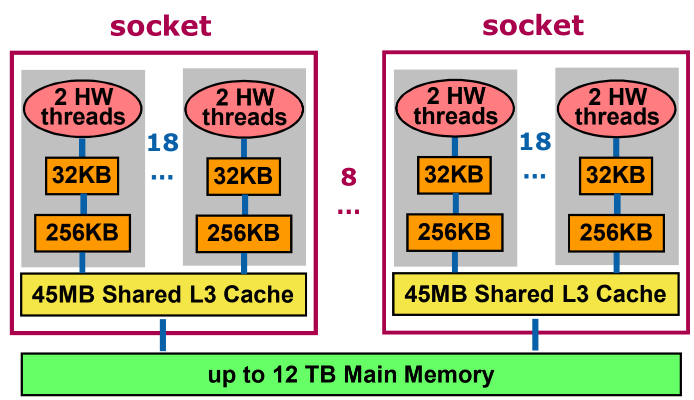
[SOSP'13 best paper]

- **Austin Clements** (Google)
- **Frans Kaashoek** (MIT)
- **Nickolai Zeldovich** (MIT)
- **Robert Morris** (MIT)
- **Eddie Kohler** (Harvard)

Multicore: 144-core Xeon Haswell E7-v3



Attach: Hard Drives & Flash Devices

An Analysis of Linux Scalability to Many Cores [OSDI'10]

- **Analyzed 7 system apps running on Linux on 48-cores**
  - Exim, memcached, Apache, PostgreSQL, gmake, Psearchy, MapReduce
  - Used an in-memory file system to explore non-disk limitations
- **Identified & sought to remove all scalability bottlenecks**
- **Kernel changes are all localized**
  - typically involve avoiding locks & atomic instructions by organizing data structures to avoid unnecessary sharing

Serializing Actions

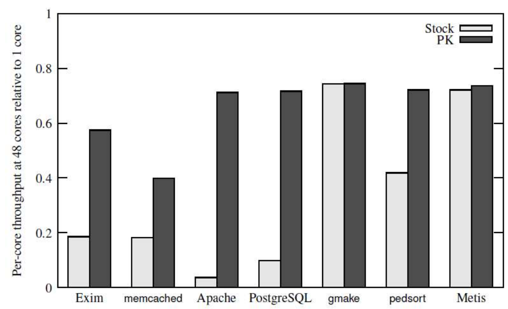
Scalability bottlenecks can arise when tasks:

- **Lock a shared data structure**
- **Write a shared memory location**
- **Compete for on-chip cache space**
- **Compete for on-chip interconnect or DRAM interface**
- **Are already mostly idle**

Scalability Problems & Fixes

Problem	Cause		Fix	Applications
Parallel `accept`	Concurrent `accept` system calls contend on shared `socket` fields.	⇒	User per-core backlog queues for listening sockets.	Apache
`dentry` reference counting	File name resolution contends on directory entry reference counts.	⇒	Use sloppy counters to reference count directory entry objects.	Apache, Exim
Mount point (`vfsmount`) reference counting	Walking file name paths contends on mount point reference counts.	⇒	Use sloppy counters for mount point objects.	Apache, Exim
IP packet destination (`dst_entry`) reference counting	IP packet transmission contends on routing table entries.	⇒	Use sloppy counters for IP routing table entries.	memcached, Apache
Protocol memory usage tracking	Cores contend on counters for tracking protocol memory consumption.	⇒	Use sloppy counters for protocol usage counting.	memcached, Apache
Acquiring directory entry (`dentry`) spin locks	Walking file name paths contends on per-directory entry spin locks.	⇒	Use a lock-free protocol in `dlookup` for checking filename matches.	Apache, Exim
Mount point table spin lock	Resolving path names to mount points contends on a global spin lock.	⇒	Use per-core mount table caches.	Apache, Exim
Adding files to the open list	Cores contend on a per-super block list that tracks open files.	⇒	Use per-core open file lists for each super block that has open files.	Apache, Exim
Allocating DMA buffers	DMA memory allocations contend on the memory node 0 spin lock.	⇒	Allocate Ethernet device DMA buffers from the local memory node.	memcached, Apache
False sharing in `net_device` and `device`	False sharing causes contention for read-only structure fields.	⇒	Place read-only fields on their own cache lines.	memcached, Apache, PostgreSQL
False sharing in `page`	False sharing causes contention for read-mostly structure fields.	⇒	Place read-only fields on their own cache lines.	Exim
`inode` lists	Cores contend on global locks protecting lists used to track `inodes`.	⇒	Avoid acquiring the locks when not necessary.	memcached, Apache
Dcache lists	Cores contend on global locks protecting lists used to track `dentrys`.	⇒	Avoid acquiring the locks when not necessary.	memcached, Apache
Per-`inode` mutex	Cores contend on a per-`inode` mutex in `lseek`.	⇒	Use atomic reads to eliminate the need to acquire the mutex.	PostgreSQL
Super-page fine grained locking	Super-page soft page faults contend on a per-process mutex.	⇒	Protect each super-page memory mapping with its own mutex.	Metis
Zeroing super-pages	Zeroing super-pages flushes the contents of on-chip caches.	⇒	Use non-caching instructions to zero the contents of super-pages.	Metis

Throughput: Before & After



Application	Bottleneck
Exim	App: Contention on spool directories
memcached	HW: Transmit queues on NIC
Apache	HW: Receive queues on NIC
PostgreSQL	App: Application-level spin lock
gmake	App: Serial stages and stragglers
pedsort	HW: Cache capacity
Metis	HW: DRAM throughput

Remaining Bottlenecks are not due to Linux

[Slides from SOSP’13 talk]

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

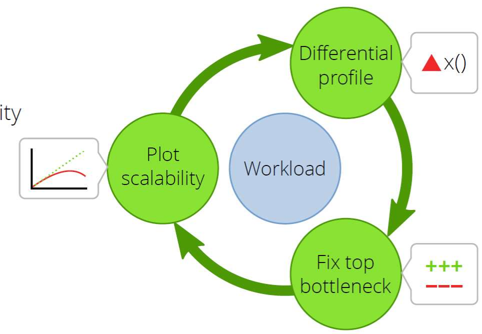
2009 — Corey, OSDI '08

2010 — Linux scalability, OSDI '10

2011

2012 — Bonsai VM, ASPLOS '12

2013 — RadixVM, EuroSys '13

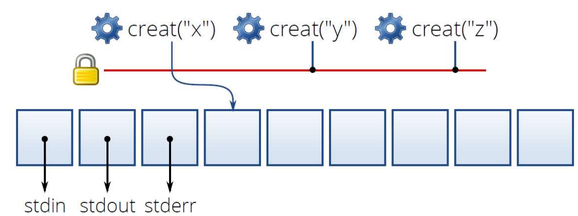


Current approach to scalable software development

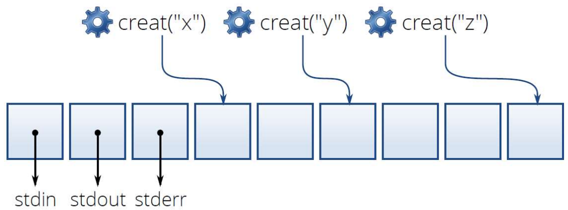
Successful in practice because it focuses developer effort

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

Interface scalability example



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	? 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	✗	
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	✓	$\xrightarrow{\text{rule}}$	✓

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

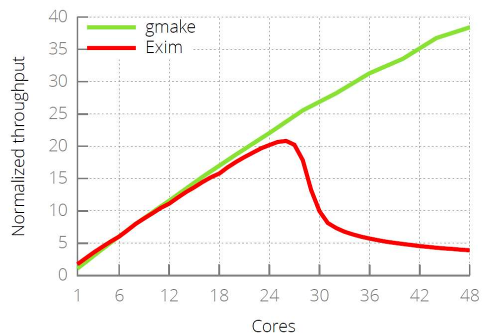
sv6: A scalable POSIX-like kernel

Outline

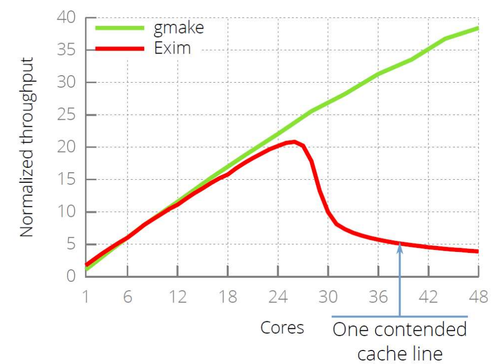
Defining the rule
- Definition of scalability
- Intuition
- Formalization

Applying the rule
- Commuter
- Evaluation

A scalability bottleneck

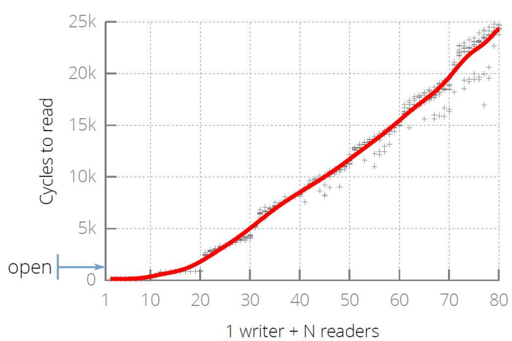


A scalability bottleneck



A single contended cache line can wreck scalability

Cost of a contended cache line



What scales on today's multicores?

		Core *X*		
		W	R	-
Core *Y*	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

Formalizing the rule

Y *SI-commutes* in $X \mathbin{||} Y :=$

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

Y *SIM-commutes* in $X \mathbin{||} Y :=$

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

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

Given a specification $\mathscr{S}$,
a history $X \mathbin{||} Y$ in which Y SIM-commutes,
and a reference implementation M that can generate $X \mathbin{||} Y$,
∃ 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

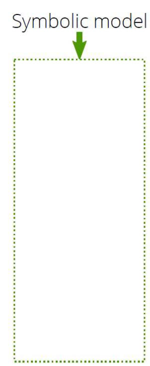
	Commutes	Scalable implementation exists
P1: creat P1: creat	✗	
P1: creat("/tmp/x") P2: creat("/etc/y")	✓	✓ (Linux)
P1: creat("/x") P2: creat("/y")	✓	✓
P1: creat("x", O_EXCL) P2: creat("x", O_EXCL)		
Same CWD	✗	
Different CWD	✓	✓

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:

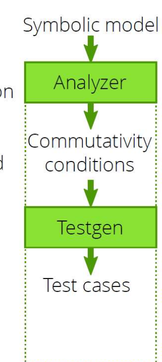
- Both source files exist and all names are different
- Neither source file exists
- a xor c exists, and it is not the other rename's destination
- Both calls are self-renames
- One call is a self-rename of an existing file and a != c
- a & c are hard links to the same inode, a != c, and b == d

Test cases

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

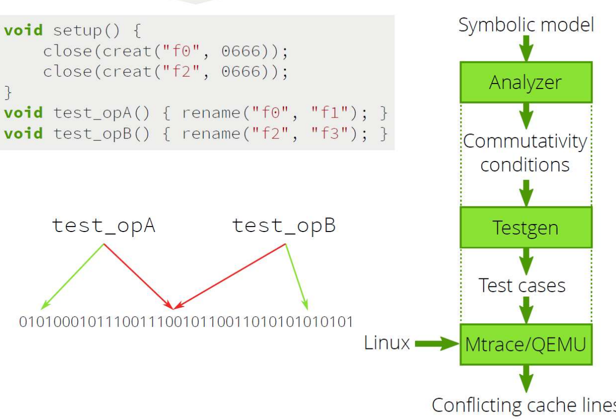
- Both source files exist and all names are different
- Neither source file exists
- a xor c exists, and it is not the other rename's destination
- Both calls are self-renames
- One call is a self-rename of an existing file and a != c
- a & c are hard links to the same inode, a != c, and b == d

```
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"); }
```

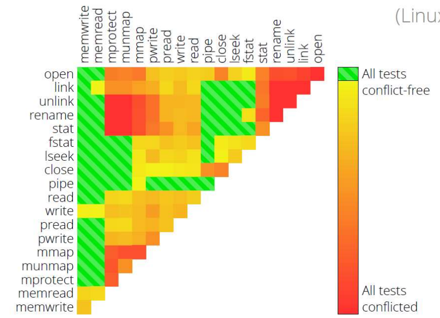


Evaluation

Does the rule help build scalable systems?

Commuter finds non-scalable cases in Linux

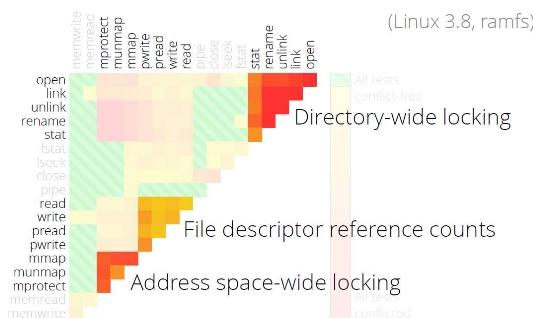
(Linux 3.8, ramfs)



13,664 total test cases
68% are conflict-free
Many are "corner cases," many are not.

Commuter finds non-scalable cases in Linux

(Linux 3.8, ramfs)



13,664 total test cases
68% are conflict-free
Many are "corner cases," many are not.

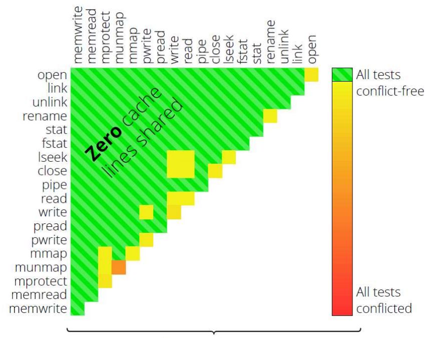
sv6: A scalable OS

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"

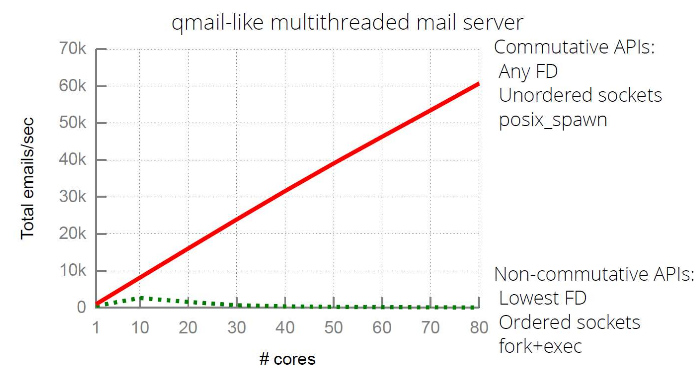
Commutative operations can be made to scale

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



Related work

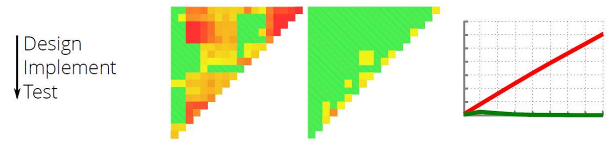
Commutativity and concurrency
- [Bernstein '81]
- [Weihl '88]
- [Steele '90]
- [Rinard '97]
- [Shapiro '11]

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.



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

[End of slides from SOSP'13 talk]

Discussion

- **Scalable Commutativity Rule only implies that there exists an implementation with conflict-free accesses**
  - Implementation constructed in proof is not practical
- **Can scalability suffer even for conflict-free accesses?**
  1. Lock a shared data structure
  2. Write a shared memory location
  3. Compete for on-chip cache space
  4. Compete for on-chip interconnect or DRAM interface
  5. Are already mostly idle

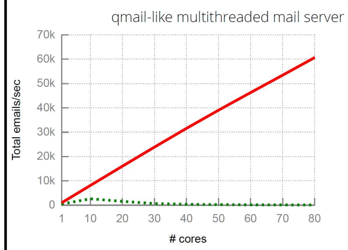
Yes. Only addresses first two sources of lack of scalability

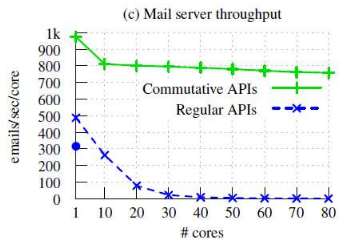
Interface Changes for POSIX

1. **Decompose compound operations**
2. **Embrace specification non-determinism**
3. **Permit weak ordering**
4. **Release resources asynchronously**

- Lowest FD versus any FD **(2)**
- stat versus xstat **(1)**
- Unordered sockets **(3)**
- Delayed munmap **(4)**
- fork+exec versus posix_spawn **(1)**

Application Performance





Next Class

Wednesday's Midterm