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

- 2012: Bonsai VM
  ASPLOS '12

- 2013: RadixVM
  EuroSys '13
Current approach to scalable software development

2008
2009
2010
2011
2012
2013
2014

Corey
OSDI '08

Linux scalability
OSDI '10

Bonsai VM
ASPLOS '12

RadixVM
EuroSys '13

Workload
Current approach to scalable software development

2009
Corey
OSDI '08

2010
Linux scalability
OSDI '10

2011
Bonsai VM
ASPLOS '12

2012
RadixVM
EuroSys '13

2013

Plot scalability
Workload
Current approach to scalable software development

- 2008: Corey, OSDI '08
- 2009: Linux scalability, OSDI '10
- 2010: Bonsai VM, ASPLOS '12
- 2012: RadixVM, EuroSys '13
- 2013: Plot scalability
- Differential profile
- Workload

Current approach to scalable software development

- **2009**: Corey (OSDI '08)
- **2010**: Linux scalability (OSDI '10)
- **2011**: Bonsai VM (ASPLOS '12)
- **2012**: RadixVM (EuroSys '13)

Diagram:

- **Workload**
- **Plot scalability**
- **Differential profile**
- **Fix top bottleneck**

Legend:

- \(\triangle x()\)
- +++
- ---
Current approach to scalable software development

2008
Corey
OSDI '08

2009
Linux scalability
OSDI '10

2010
Bonsai VM
ASPLOS '12

2011
RadixVM
EuroSys '13

2012

2013

2014

Differential profile

Fix top bottleneck

Workload

Plot scalability

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
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

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

stdin  stdout  stderr
Interface scalability example

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

stdin  stdout  stderr
Whenever interface operations commute, they can be implemented in a way that scales.
The scalable commutativity rule

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

creat with lowest FD

Scalable implementation exists

Commutes

?
Approach: Interface-driven scalability

The scalable commutativity rule

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

creat with lowest FD

Scalable implementation exists

Commutes

creat → 3
creat → 4
Whenever interface operations commute, they can be implemented in a way that scales.

The scalable commutativity rule

Creat with lowest FD

Scalable implementation exists

Commutes
The scalable commutativity rule

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

<table>
<thead>
<tr>
<th>creat with lowest FD</th>
<th>creat with any FD</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>?</td>
</tr>
<tr>
<td>creat → 42</td>
<td>creat → 17</td>
</tr>
</tbody>
</table>

Scalable implementation exists

Approach: Interface-driven scalability
Approach: Interface-driven scalability

The scalable commutativity rule

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

- creat with lowest FD: ✗
- creat with any FD: ✓

Scalable implementation exists

Rule

Commutes
# Advantages of interface-driven scalability

The rule enables reasoning about scalability throughout the software design process.

<table>
<thead>
<tr>
<th>Design</th>
<th>Guides design of scalable interfaces</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implement</td>
<td>Sets a clear implementation target</td>
</tr>
<tr>
<td>Test</td>
<td>Systematic, workload-independent scalability testing</td>
</tr>
</tbody>
</table>
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

The graph illustrates the normalized throughput of two different applications, `gmake` and `Exim`, as a function of the number of cores. The x-axis represents the number of cores, ranging from 1 to 48, and the y-axis represents the normalized throughput, ranging from 0 to 40. The graph shows that `gmake` experiences a steady increase in throughput with an upward trend, while `Exim` initially shows a sharp increase but then reaches a peak and starts to decrease. This indicates a scalability bottleneck for `Exim` beyond a certain number of cores.
A scalability bottleneck

A single contended cache line can wreck scalability
Cost of a contended cache line

Cycles to read

1 writer + N readers
Cost of a contended cache line

Cycles to read

1 writer + N readers

open

Cost of a contended cache line
What scales on today's multicores?

<table>
<thead>
<tr>
<th>Core Y</th>
<th></th>
<th>Core X</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>W</td>
<td>X</td>
<td>W</td>
<td>✓</td>
</tr>
<tr>
<td>R</td>
<td>X</td>
<td>R</td>
<td>✓</td>
</tr>
<tr>
<td>-</td>
<td>✓</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Core X</td>
<td>W</td>
<td>R</td>
<td>-</td>
</tr>
<tr>
<td>--------</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>Core Y</td>
<td>W</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>R</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>-</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
</tr>
</tbody>
</table>
What scales on today's multicores?

<table>
<thead>
<tr>
<th>Core Y</th>
<th>Core X</th>
</tr>
</thead>
<tbody>
<tr>
<td>W</td>
<td>X</td>
</tr>
<tr>
<td>R</td>
<td>✓</td>
</tr>
<tr>
<td>-</td>
<td>✓</td>
</tr>
</tbody>
</table>
What scales on today's multicores?

<table>
<thead>
<tr>
<th></th>
<th>Core X</th>
</tr>
</thead>
<tbody>
<tr>
<td>W</td>
<td>R</td>
</tr>
<tr>
<td>W</td>
<td>x</td>
</tr>
<tr>
<td>R</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>✓</td>
</tr>
</tbody>
</table>

We say two or more operations are *scalable* if they are *conflict-free*. 
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 $X \parallel Y :=$
\[ \forall Y' \in \text{reorderings}(Y), Z: X \parallel Y \parallel Z \in \mathcal{I} \iff X \parallel Y' \parallel Z \in \mathcal{I}. \]

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

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

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

Proof by simulation construction.
Formalizing the rule

Commutativity is sensitive to operations, arguments, and state

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

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

Proof by simulation construction.

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

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

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

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

P1: creat
P1: creat

P1: creat("/tmp/x")
P2: creat("/etc/y")

P1: creat("/x")
P2: creat("/y")

Scalable implementation exists

✓ ✓

(Linux)
Example of using the rule

<table>
<thead>
<tr>
<th>Commutes</th>
<th>Scalable implementation exists</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓ ✓</td>
<td>✓ (Linux)</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>P1: creat(&quot;/tmp/x&quot;)</td>
<td>✓</td>
</tr>
<tr>
<td>P2: creat(&quot;/etc/y&quot;)</td>
<td>✓</td>
</tr>
<tr>
<td>P1: creat(&quot;/x&quot;)</td>
<td>✓</td>
</tr>
<tr>
<td>P2: creat(&quot;/y&quot;)</td>
<td>✓</td>
</tr>
</tbody>
</table>
Example of using the rule

<table>
<thead>
<tr>
<th>Commutes</th>
<th>Scalable implementation exists</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1: creat</td>
<td>✗</td>
</tr>
<tr>
<td>P1: creat</td>
<td></td>
</tr>
<tr>
<td>P1: creat(&quot;/tmp/x&quot;)</td>
<td>✓</td>
</tr>
<tr>
<td>P2: creat(&quot;/etc/y&quot;)</td>
<td>✓</td>
</tr>
<tr>
<td>P1: creat(&quot;/x&quot;)</td>
<td>✓</td>
</tr>
<tr>
<td>P2: creat(&quot;/y&quot;)</td>
<td>✓</td>
</tr>
<tr>
<td>P1: creat(&quot;x&quot;, O_EXCL)</td>
<td>✓</td>
</tr>
<tr>
<td>P2: creat(&quot;x&quot;, O_EXCL)</td>
<td>✓</td>
</tr>
</tbody>
</table>
### Example of using the rule

<table>
<thead>
<tr>
<th>Commutes</th>
<th>Scalable implementation exists</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1: creat</td>
<td>✗</td>
</tr>
<tr>
<td>P1: creat</td>
<td></td>
</tr>
<tr>
<td>P1: creat(&quot;/tmp/x&quot;)</td>
<td>✓</td>
</tr>
<tr>
<td>P2: creat(&quot;/etc/y&quot;)</td>
<td>✓</td>
</tr>
<tr>
<td>P1: creat(&quot;/x&quot;)</td>
<td>✓</td>
</tr>
<tr>
<td>P2: creat(&quot;/y&quot;)</td>
<td>✓</td>
</tr>
<tr>
<td>P1: creat(&quot;x&quot;, O_EXCL)</td>
<td></td>
</tr>
<tr>
<td>P2: creat(&quot;x&quot;, O_EXCL)</td>
<td></td>
</tr>
<tr>
<td>Same CWD</td>
<td>✗</td>
</tr>
<tr>
<td>Different CWD</td>
<td>✓</td>
</tr>
</tbody>
</table>
Applying the rule to real systems

- Interface specification (e.g., POSIX)
- Implementation (e.g., Linux)
- Commuter
- All scalability bottlenecks
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
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
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

```c
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

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

Symbolic model
Analyzer
Commutativity conditions
Testgen
Test cases
Mtrace/QEMU

Output: Conflicting cache lines
Evaluation

Does the rule help build scalable systems?
Commuter finds non-scalable cases in Linux

13,664 total test cases
68% are conflict-free
Many are "corner cases," many are not.
Commuter finds non-scalable cases in Linux

Directory-wide locking

File descriptor reference counts

Address space-wide locking

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 scalability

qmail-like multithreaded mail server

Total emails/sec

# cores

Non-commutative APIs:
Lowest FD
Ordered sockets
fork+exec
Commutative operations matter to app scalability

qmail-like multithreaded mail server

Commutative APIs:
- Any FD
- Unordered sockets
- posix_spawn

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

Graph showing the relationship between total emails per second and the number of cores.
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]
Whenever interface operations commute, they can be implemented in a way that scales.

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