x86 CPU trends
x86 CPU trends
x86 CPU trends

Clock speed (MHz)

Power (watts)

Sources: Stanford CPUDB, Intel ARK
x86 CPU trends

Sources: Stanford CPUDB, Intel ARK
x86 CPU trends

- Clock speed (MHz)
- Power (watts)
- Cores per socket

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

Exim mail server

Problem lies in the OS kernel
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
  - OSDI '08

- Linux scalability
  - OSDI '10

- Bonsai VM
  - ASPLOS '12

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

- Corey
  - OSDI '08

- Linux scalability
  - OSDI '10

- Bonsai VM
  - ASPLOS '12

- RadixVM
  - EuroSys '13

Workload
Current approach to scalable software development

- 2008: Corey (OSDI '08)
- 2009: Linux scalability (OSDI '10)
- 2010: Bonsai VM (ASPLOS '12)
- 2011: RadixVM (EuroSys '13)
- **Plot scalability**
- **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

Differential profile

Plot scalability

Workload

|x()|
Current approach to scalable software development

- Corey (OSDI '08)
- Linux scalability (OSDI '10)
- Bonsai VM (ASPLOS '12)
- RadixVM (EuroSys '13)

Diagram:
- Plot scalability
- Differential profile
- Workload
- Fix top bottleneck

The image shows a timeline from 2008 to 2014 with key events and concepts related to software scalability and development.
Current approach to scalable software development

- Corey (OSDI '08)
- Linux scalability (OSDI '10)
- Bonsai VM (ASPLOS '12)
- RadixVM (EuroSys '13)

Diagram showing the current approach:
- Workload
- Plot scalability
- Fix top bottleneck
- Differential profile
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
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

crea("x")  crea("y")  crea("z")
Interface scalability example
Interface scalability example

Solution: Change the interface?
Interface scalability example

Solution: Change the interface?
Whenever interface operations commute, they can be implemented in a way that scales.
Whenever interface operations commute, they can be implemented in a way that scales.

The scalable commutativity rule

Scalable implementation exists

Commutes

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.

creat with lowest FD

Scalable implementation exists

Commutes

creat → 3
creat → 4
The scalable commutativity rule

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

Scalable implementation exists

Commutates

creat with lowest FD
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: ?
  - creat → 42
  - creat → 17

Scalable implementation exists
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

Commutes rule
## 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>
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

Cycles to read

1 writer + N readers
Cost of a contended cache line

![Graph showing cycles to read for 1 writer + N readers](image_url)
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><img src="x" alt="Cross" /> <img src="x" alt="Cross" /> <img src="%E2%9C%93" alt="Check" /></td>
</tr>
<tr>
<td>R</td>
<td><img src="x" alt="Cross" /> <img src="%E2%9C%93" alt="Check" /> <img src="%E2%9C%93" alt="Check" /></td>
</tr>
<tr>
<td>-</td>
<td><img src="%E2%9C%93" alt="Check" /> <img src="%E2%9C%93" alt="Check" /> -</td>
</tr>
</tbody>
</table>
What scales on today's multicores?

<table>
<thead>
<tr>
<th></th>
<th>Core X</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>W</td>
</tr>
<tr>
<td>Core Y</td>
<td>W</td>
</tr>
<tr>
<td></td>
<td>R</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>✗ ✗</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*.

Good approximation of current hardware.
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 \( \text{iszero()} \rightarrow F \)

T2 \( \text{iszero()} \rightarrow F \)

T3 \( \text{dec()} \rightarrow 2 \)

T4 \( \text{dec()} \rightarrow 1 \)

T5 \( \text{dec()} \rightarrow 0 \)
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
T3  iszero() → F  dec() → 2
T4  dec() → 1
T5  dec() → 0

✓ 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 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 commutes; conflict-free implementation: shared counter
✗ R2 does not commute because dec() returns counter value
Example: Reference counter

T1 \( \text{iszero}() \rightarrow F \)
T2 \( \text{iszero}() \rightarrow F \)
T3 \( \text{dec}() \rightarrow \text{ok} \)
T4 \( \text{dec}() \rightarrow \text{ok} \)
T5 \( \text{dec}() \rightarrow \text{ok} \)

\( R1 \)
\( R2' \)

✓ R1 commutes; conflict-free implementation: shared counter
✗ R2 does not commute because \( \text{dec}() \) returns counter value
Example: Reference counter

1. \( \text{iszero()} \rightarrow \text{F} \)
2. \( \text{iszero()} \rightarrow \text{F} \)
3. \( \text{dec()} \rightarrow \text{ok} \)
4. \( \text{dec()} \rightarrow \text{ok} \)
5. \( \text{dec()} \rightarrow \text{ok} \)

\[ \text{R1} \]

- ✓ R1 commutes; conflict-free implementation: shared counter
- ✗ R2 does not commute because \( \text{dec()} \) returns counter value

\[ \text{R2'} \]

- ✓ 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 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**: ✓ R1 commutes; conflict-free implementation: shared counter
- **R2**: ✗ R2 does not commute because `dec()` returns counter value
- **R2'**: ✓ R2' does commute; conflict-free implementation: per-core counter

**R3** depends on state
- ✓ Initial value > 3
- ✗ Initial value ≤ 3
Example: Reference counter

R1 commutes; conflict-free implementation: shared counter

R2 does not commute because \texttt{dec()} returns counter value

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

R3 depends on state

- Initial value $> 3$
- Initial value $\leq 3$
Formalizing the rule

Definitions

- History
- Reordering
- Commutativity

Formal scalable commutativity rule
A **history** $H$ is a sequence of invocations and responses on threads.

\[
\begin{align*}
T_1 & \xrightarrow{\text{inc()}} \text{ok} \\
T_2 & \xrightarrow{\text{iszero()}} T \\
\end{align*}
\]
A **history** $H$ is a sequence of invocations and responses on threads.

A **specification** $\mathcal{S}$ defines an interface. $\mathcal{S}$ is the set of legal histories giving the allowed behavior of an interface. [Herlihy & Wing, '90]
A **history** $H$ is a sequence of invocations and responses on threads.

A **specification** $\mathcal{S}$ defines an interface. $\mathcal{S}$ 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.
A **reordering** $H'$ is a permutation of $H$ that maintains operation order for each individual thread ($H|t = H'|t$ for all $t$).
A **reordering** $H'$ is a permutation of $H$ that maintains operation order for each individual thread ($H| t = H'| t$ for all $t$).

\[
\begin{align*}
T1 & \quad \text{inc()} \quad \rightarrow \quad \text{ok} \\
T2 & \quad \text{iszero()} \quad \rightarrow \quad T
\end{align*}
\]

\[
\begin{align*}
T1 & \quad \text{inc()} \quad \rightarrow \quad \text{ok} \\
T2 & \quad \text{iszero()} \quad \rightarrow \quad T
\end{align*}
\]

\[
\begin{align*}
T1 & \quad \text{inc()} \quad \rightarrow \quad \text{ok} \\
T2 & \quad \text{iszero()} \quad \rightarrow \quad T
\end{align*}
\]

\[
\begin{align*}
T1 & \quad \text{inc()} \quad \rightarrow \quad \text{ok} \\
T2 & \quad \text{iszero()} \quad \rightarrow \quad T
\end{align*}
\]
A **reordering** $H'$ is a permutation of $H$ that maintains operation order for each individual thread ($H|t = H'|t$ for all $t$).

\[
\begin{align*}
T1 & \quad \text{inc()} \quad \rightarrow \quad \text{ok} \\
T2 & \quad \text{iszero()} \quad \rightarrow \quad T
\end{align*}
\]

\[
\begin{align*}
T1 & \quad \text{inc()} \quad \rightarrow \quad \text{ok} \\
T2 & \quad \text{iszero()} \quad \rightarrow \quad T
\end{align*}
\]

\[
\begin{align*}
T1 & \quad \text{inc()} \quad \rightarrow \quad \text{ok} \\
T2 & \quad \text{iszero()} \quad \rightarrow \quad T
\end{align*}
\]
A region $Y$ of a legal history $XY$ \textbf{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 $Y$.)
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 $Y$.)
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 $Y$.)
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 $Y$.)
Let $\mathcal{S}$ 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 $\mathcal{S}$ whose execution of $XY$ is conflict-free in the commutative region $Y$. 
Let $\mathcal{S}$ 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 $\mathcal{S}$ whose execution of $XY$ is conflict-free in the commutative region $Y$. 

\[ M' \quad \bullet \quad \bullet \quad \bullet \quad \bullet \quad \bullet \quad X \]
Let $\mathcal{S}$ 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 $\mathcal{S}$ whose execution of $XY$ 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 $XY$ where $Y$ commutes in $XY$ and $M$ can generate $XY$.

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

The formal scalable commutativity rule
Applying the rule to real systems
Applying the rule to real systems

Commuter
Applying the rule to real systems

Interface specification (e.g., POSIX)

Commuter

Implementation (e.g., Linux)

All scalability bottlenecks
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
```
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
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

```python
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
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
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
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

```c
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

```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
Linux
Mtrace/QEMU
Conflicting cache lines

Output: Conflicting cache lines
---
010100010111001110010110011010101010101

d_entry.d_lock
inode_cache
+17 more conflicts
Does the rule help build scalable systems?
Commuter finds non-scalable cases in Linux

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

(Linux 3.8, ramfs)
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
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"

Two lseek operations of same FD to the same offset
Two pwrite operations of same data to same offset
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

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

Total emails/sec vs # cores
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
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.
Whenever interface operations commute, they can be implemented in a way that scales.

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