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

Phil Gibbons
15-712 F15
Lecture 18

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

up to 12 TB Main Memory

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

Scalability Problems & Fixes

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

Throughput: Before & After

Remaining Bottlenecks are not due to Linux
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

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.

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.
### Approach: Interface-driven scalability

**The scalable commutativity rule**

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

<table>
<thead>
<tr>
<th>Scalable</th>
<th>Commutes</th>
<th>Implement</th>
<th>exists</th>
</tr>
</thead>
<tbody>
<tr>
<td>creat with lowest FD</td>
<td>✗ rule</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>creat with any FD</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

### 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

![Graph showing normalized throughput vs cores for gmake and Exim](Image)

A single contended cache line can wreck scalability

Cost of a contended cache line

![Graph showing cycles to read vs 1 writer + N readers](Image)

What scales on today's multicores?

<table>
<thead>
<tr>
<th>Core X</th>
<th>Core Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>W</td>
<td>R</td>
</tr>
<tr>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>✔</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.
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

\[
\begin{align*}
Y \text{ SI-commutes in } X \parallel Y & \Rightarrow \\
\forall Y' \in \text{ reorderings}(Y), Z : X \parallel Y' \parallel Z \in \mathcal{S} \iff X \parallel Y' \parallel Z \in \mathcal{S}.
\end{align*}
\]

\[
\begin{align*}
Y \text{ SIM-commutes in } X \parallel Y & \Rightarrow \\
\forall P \in \text{ prefixes(reorderings}(Y}): P \text{ SI-commutes in } X \parallel P.
\end{align*}
\]

An implementation \( m \) is a step function: state \( \times \) inv \( \Rightarrow \) state \( \times \) resp.

Given a specification \( \mathcal{S} \), a history \( X \parallel Y \) in which \( Y \text{ SIM-commutes} \), 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.

Formalizing the rule

Commutativity is sensitive to operations, arguments, and state

An implementation \( m \) of \( \mathcal{S} \) needs to be state = resp.

Given a specification \( \mathcal{S} \), a history \( X \parallel Y \) in which \( Y \text{ SIM-commutes} \), 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.

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>X</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;, 0_EXCL)</td>
<td>X</td>
</tr>
<tr>
<td>P2: creat(&quot;x&quot;, 0_EXCL)</td>
<td>✔</td>
</tr>
</tbody>
</table>

Same CWD | X |

Different CWD | ✔ | ✔
Input: Symbolic model

```
Symbolic model

class POSIX:
def __init__(self):
    self.frame_to_inum = SymDir

def rename(src, dst):
    if src not in self.frame_to_inum:
        return (-1, errno.EINOTENT)
    if dst in self.frame_to_inum:
        self.frame_to_inum[dst].nlink -= 1
        self.frame_to_inum[src] = self.frame_to_inum[dst]
        del self.frame_to_inum[dst]
    return 0
```

Commutativity conditions

```
@symsrc(src=SymFilename, dst=SymFilename)
def rename(self, src, dst):
    if src not in self.frame_to_inum:
        return (-1, errno.EINOTENT)
    if src == dst:
        return 0
    if dst in self.frame_to_inum:
        self.frame_to_inum[dst].nlink -= 1
        self.frame_to_inum[src] = self.frame_to_inum[dst]
        del self.frame_to_inum[dst]
    return 0
```

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**

(Linear 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**

(File descriptor reference counts
Address space-wide locking
Directory-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
Cummative 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

Cummative operations matter to app scalability

Plot showing the performance of qmail-like multithreaded mail server with commutative APIs vs non-commutative APIs.
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