Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask

Tudor David, Rachid Guerraoui and Vasileios Trigonakis
Ecole Polytechnique Federale de Lausanne (EPFL)

Haksu Lim, Luis, Hwanjin Jeong

2016-04-18
Multi-Core

- Multi-core is used in many systems
- Then number of core ↑, Performance ↑? NO

Synchronization is one of the biggest scalability bottlenecks
Synchronization

• Why does we use?
  ▪ Concurrent access to shared data
  ▪ To ensure the orderly execution

• Why is synchronization bottleneck?
  ▪ Hardware
  ▪ Synchronization algorithm
  ▪ Application context
  ▪ Workload

Focusing this
Cache Coherence

- Multi-core system have a separate cache for each core
  - Write operation break consistency among caches
- Cache coherence
  - To maintain caches of a common memory resource
Cache Coherence protocols

- MSI protocol
Cache Coherence Protocols

• MESI protocol
  ▪ Added exclusive state
    - No other has a copy of this cache line
  ▪ Reduced expensive invalidate operation

• MOESI protocol
  ▪ Added owned state
    - This cache line has been modified but there might be more shared copy on other core
  ▪ Reduced expensive write operation to memory
Cache Coherence Example

• Acquiring lock process

Acq(lock);

Processor

Cache

State

Data

Inval

Held=1

Update

Invalidate

Processor

Cache

State

Data

Mod

Held=1

Read-Exclusive

Shared memory (held = 1)
What to deal with

• Hardware Processors
  ▪ Multi-sockets
    – AMD Opteron
      • 4 x 6172 – 48 cores
    – Intel Xeon
      • 8 x E7-8867L – 80 cores
  ▪ Single-sockets
    – Sun Niagara 2
      • 8 cores
    – Tilera TILE-Gx36
      • 36 cores

• Synchronization layer
  ▪ Concurrent software
    – Hash table, etc.
  ▪ Primitives
    – Lock, etc.
  ▪ Atomic operations
    – Compare & swap, etc.
  ▪ Cache coherence
    – Load & store
Hardware-Level Analysis
Local Accesses

- **Opteron**
  - Within socket: 40 ns

- **Zeon**
  - Within socket: 20 – 40 ns
Remote Accesses

- **Opteron**
  - Within socket: 40 ns
  - Per hop: +40 ns

- **Zeon**
  - Within socket: 20 – 40 ns
  - Per hop: +50 ns
## Operation Latency – Multi Socket

<table>
<thead>
<tr>
<th>System State</th>
<th>Opteron Loads</th>
<th></th>
<th>Xeon Stores</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hops</td>
<td>Same Die</td>
<td>Same MCM</td>
<td>One Hop</td>
</tr>
<tr>
<td></td>
<td>81</td>
<td>161</td>
<td>172</td>
</tr>
<tr>
<td>Modified</td>
<td>109</td>
<td>289</td>
<td>400</td>
</tr>
<tr>
<td>Owned</td>
<td>83</td>
<td>163</td>
<td>175</td>
</tr>
<tr>
<td>Exclusive</td>
<td>92</td>
<td>273</td>
<td>383</td>
</tr>
<tr>
<td>Shared</td>
<td>44</td>
<td>223</td>
<td>334</td>
</tr>
<tr>
<td>Invalid</td>
<td>355</td>
<td>492</td>
<td>601</td>
</tr>
</tbody>
</table>

Crossing sockets is a killer
Up to 7.5x more expensive
Single-Socket Processors

Niagara

• Equidistant from the cache
• Uniform: 23ns

Tilera

• Non uniform
• 1 hop: 40ns
• Per hop: +2 ns
Operation Latency – Single Socket

<table>
<thead>
<tr>
<th>System</th>
<th>Niagara</th>
<th>Tilera</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Hops</td>
<td></td>
</tr>
<tr>
<td>State</td>
<td>same core</td>
<td>other core</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Modified</td>
<td>3</td>
<td>24</td>
</tr>
<tr>
<td>Owned</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Exclusive</td>
<td>3</td>
<td>24</td>
</tr>
<tr>
<td>Shared</td>
<td>3</td>
<td>24</td>
</tr>
<tr>
<td>Invalid</td>
<td>176</td>
<td>176</td>
</tr>
</tbody>
</table>

**loads**

| stores | Modified | 24 | 24 | 57 | 77 |
|        | Owned    | -  | -  | -  | -  |
|        | Exclusive| 24 | 24 | 57 | 77 |
|        | Shared   | 24 | 24 | 86 | 106|

Uniform is expected to scale better, The non-uniform is affected both distance and the number of involved cores
Atomic Operations – Multi Sockets

• Very fast single-thread performance
  ▪ But drops on two or more cores and decreases further when there is cross-socket communication
Atomic Operations – Single Sockets

• Lower single-thread throughput
  ▪ But scale to a maximum value
Software-Level Analysis
Analysis Scope

• 9 Locks
  ▪ Spinlocks
    – Test and test-and-set lock (TTAS), Ticket lock
  ▪ Queue based lock
    – Array based lock, CLH lock, MCS lock
  ▪ Hierarchical lock
    – Hierarchical CLH lock, Hierarchical ticket lock
  ▪ Mutex

• Concurrent software
  ▪ Hash table
Ticket Lock

Acquired Ticket : 0

Acquiring Ticket : 1

Acquiring Ticket : 2

Acquiring Ticket : 3

Acquiring Ticket : 4

Next ticket : 1

Now serving : 0

Spin Spin Spin Spin Spin
Ticket Lock

Release

Spin

Acquiring Ticket: 1

Acquiring Ticket: 2

Acquiring Ticket: 3

Acquiring Ticket: 4

Lock

Next ticket: 2

Now serving: 1
CLH Lock

Acquiring

tail → false
CLH Lock

Acquired

tail → true → false

prev reference
CLH Lock

Acquiring

Spin

false

Acquired

true

false

prev

reference
CLH Lock

Acquiring

Unlock

Spin

tail
false
false
false

prev
reference
CLH Lock

Acquired

prev
reference
Hierarchical Lock
Hierarchical Lock

- NUMA aware lock
  - Using local cache for lock
Locks Microbenchmark

• Initialize N locks & T threads

• Each thread repeatedly
  ▪ Chooses one lock out of N at random
  ▪ Acquires the lock
  ▪ Reads and writes the protected data
  ▪ Releases the lock

• Repeat with 9 different lock algorithms
  ▪ spinlocks, queue-based, hierarchical, mutex

• Report the best total throughput
Locks on Multi Sockets

High contention (4 locks)  Low contention (128 locks)

Multi sockets provide limited scalability due to higher latencies of remote access

X:Y, X: the scalability over the single-thread execution  
Y: the best-performance lock
Locks on Single Sockets

High contention (4 locks)
Low contention (128 locks)

Complex locks are generally the best under extreme contention,
Simple locks perform better under low contention

X:Y, X: the scalability over the single-thread execution
Y: the best-performance lock
Hash Table – best locks

High contention
12 buckets, 12 entries/bucket

Low contention
512 buckets, 12 entries/bucket

Simple locks are powerful
25 / 32
Conclusion

• Crossing sockets is a killer
  ▪ Up to 7.5x more expensive communication

• Intra-socket uniformity matters

• Simple locks are powerful
  ▪ Better in 25 out of 32 data-points on a hash table
Extra Slides
Hardware-Level Analysis

- Multi socket processor
  - Local access latency
  - Remote access latency
- Single socket processor
  - Intra-socket access latency
Key Observations

• Crossing sockets is a killer

• Intra-socket uniformity does matter

• Loads and stores can be as expensive as atomic operations

• Simple locks are powerful
High Contention

- Multi-socket, single lock
High Contention

- Single-socket, single lock
Low Contention

- Multi-socket, 512 locks
Low Contention

• Single-socket, 512 locks
Hash Table on Multi Sockets

- Using 80% get, 10% put, and 10% remove

High contention (12 buckets)  
Low contention (512 buckets)
Hash Table on Single Sockets

• Using 80% get, 10% put, and 10% remove

High contention (12 buckets)  
Low contention (512 buckets)
The Scalable Commutativity Rule:
Designing Scalable Software for Multicore Processors

Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler†
MIT CSAIL and †Harvard University
SOSP 2013

-Presented by-
Luis, Haksu, Hwanjin
• Evaluating scalability of multicore software:

• Focus effort on real issues.
• Different Workloads?
• Higher core #?
• Critical bottlenecks?
• Might not only be implementation.
• Too late for design-lvl solutions?
Approach

• In shared-memory multicore processor with ~MESI coherent cache, a core can scale reads and writes it has cached exclusively and scale reads that are cached in shared mode.

• Operations scale if implementations have conflict-free memory access.

• Consider scalability earlier in the process -> software interface.

  ▪ Before implementation.
  ▪ Before hardware.
The Scalable Commutativity Rule

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

Based on SIM Commutativity

• **State-dependent:** Context of system, op. arguments, and concurrent op. **NOT** all states will commute.

• **Interface-based:** Independent of implementation, just same resulting state.

• **Monotonic:** for any reordering in a prefix sequence of operations the region is commutative.
Formal explanation of the rule

A system executes actions (Invocation or response).

• **Invocation**: System call. **Response**: result.

• A series of actions forms a *history*.

\[ H = A B C A C B D D E E F G H F H G \]

• The rule only considers *well-formed* histories (one outstanding invocation at any point per thread, and each threads history form invocation-response sequence.

\[ H|t = A A D D H H \]
Formal explanation of the rule

• A *specification* (close set of well-formed histories) distinguishes if a history is “correct”, defining the interface.

• I.e. UNIX getpid()

\[
[A = \text{getpid()}, A = 0] \notin \mathcal{S}
\]

• Commutativity = order of operations irrelevant

• A set of actions are commutative when the specification is indifferent to the execution order of that set.

• For \( H = \{A, B, A, C, B, C\} \)
Formal explanation of the rule

• SI-commutation (for Y):

\[ X \parallel Y \parallel Z \in \mathcal{I} \text{ if and only if } X \parallel Y' \parallel Z \in \mathcal{I} \]

• X puts the system into desired state.
• Switching Y for Y'(reorder of Y) requires that the return values of Y are valid regardless of order.
• Z Forces that the results from the reordering of (Y) do not affect future operations.
• However this is non-monotonic, that is for some prefix reordering the region might not be commutative
Formal explanation of the rule $H = X || Y$

- SIM-commutation (for $Y$):
- When for any prefix $P$ of some reordering of $Y$.
- $P$ SIM commutes in $X || P$.

- SIM commutativity is interface based = evaluates consequences of execution order using only specification.
Designing commutative interface

Apply the rule to POSIX results in insights:

• Decompose compound operations

fork()
- New P
- Copy
- mem st
- fd
- signal

exec()
- Replace
- mem st
- fd
- signal

posix_spawn()
- New P
- Load image
Designing commutative interface

- Embrace Specification non-determinism

```
open()
Allocate fd
Return smallest fd
```
Designing commutative interface

• Weak Ordering
Designing commutative interface

- Release resources asynchronously.
  - Operations have global effects visible upon return.
  - Good for usable interface but strict for ops. That return resources.
  - No commute with last close() of a read fd. Must track no. of read fd.
• Understanding commutativity of complex interface is not trivial.

• Develop an implementation that doesn’t share when operations commute increases difficulty

• Automated tool named COMMUTER
Analyzing interface design using COMMUTER

- Input python symbolic model of interface.
- Finds conditions in which the model commutes.
- Outputs commutativity conditions: arguments and states.
- Symbolic model enables focus on external behavior.
Analyzing interface design using COMMUTER

Python model

ANALIZER  →  TESTGEN  →  MTRACE

Commutativity conditions  →  Test cases  →  Shared cache lines

TESTGEN

- Input: Commutativity conditions.
- Convert into test cases.
- Specify concrete values for every symbolic variable in the model.
- Produce actual C test case code.
- Test case code: state setup + functions to run.
- Path coverage – code path.
- Conflict coverage – access pattern.
Analyzing interface design using COMMUTER

MTRACE

- Run the test cases on a real implementation.
- On violation of commutativity rule it reports what variables where shared and the code that accessed them.
- Runs on qemu an starts log for each test case.
Implementation

- Prototype of COMMUTER
  - (ANALYZER and TESTGEN) = 3,050 lines of python.
  - MTRACE 1,594 lines of code changes in qemu.
  - Modify 612 lines of code of linux.
  - 2,865 lines of C++ code to made a program that process the log files.
Finding scalability opportunities

• Modeled 18 POSIX file system and virtual memory system calls in COMMUTER.

• Evaluate Linux kernel 3.8 scalability.

• Develop scalable file and virtual memory system.

• COMMUTER generated 13,664 test cases.

• Running the test cases 8 minutes.
Comparing Scalability

• For Linux kernel Out of 13,664 test cases 4,257 were not conflict free.

• Common cases: shared reference count, coarse grained locks.
Comparing Scalability

Follow commutativity design principles and implement on top of sv6:

- in-memory file system called ScaleFS
- virtual memory system called RadixVM
Comparing Scalability

COMMUTER pointed out

• Layer scalability: Use of data structures that satisfy commutativity rule such as; radix array, hash tabled etc.


• Precede pessimism with optimism: Check first then acquire lock.

• Don’t read unless necessary.
Performance evaluation

• 80 core machine, eight 2.4Ghz 10 core Intel E7-8870 and 25 6 GB RAM.
• Each 30MB socket L3 cache is shared by 10 cores.
• No hardware prefetcher.
• Compare Linux 3.5.7 (Ubuntu Quantal) Vs. Sv6
• Single core baseline
Microbenchmark: statbench

- Scalability of fstat.
- Create single file that \( \frac{n}{2} \) cores `fstat()`.
- Other \( \frac{n}{2} \) core link to new name then unlink
Microbenchmark: openbench

- Scalability of open.
- N threads concurrently open and close per-thread files
Microbenchmark: Mailserver

- More realworld workload.
- Separate comm proc.
- Roughly like qmail.
- Mail client with n threads continuously deliver email by spawning and feeding mail-en queue.
Conclusion

• The new rule enables design for scalability design.

• + scalable implemention == + performance (ALWAYS ???)

• Case specific, what to tune for?

• Tools gives hint about commutative rule implementation feasibility but it wont clearly specify how to achieve this.