The Scalable Commutativity Rule: Background and Introduction

Brad Karp
UCL Computer Science

M030/GZ03
4th November 2016
The Multi-Core Software Design Problem

• Old days:
  – CPU frequencies steadily increase
  – Take existing binary, runs faster on new CPU

• The multi-core era:
  – CPU frequencies cease increasing; heat dissipation no longer feasible
  – Instead, multiple CPUs (cores) on one die
  – Legacy, single-threaded binary doesn’t increase in speed as number of cores increases!
Challenges of Writing Multi-Core Code

- Must divide computation into multiple threads
- Coordination (locking), communication (data sharing) between cores costly
  - Motivates data structures that eliminate or minimize locks and their use
- Operating system shared by all applications and threads
  - Data structures in kernel bound to be shared
  - Scalable as core count, thread count increase?
“Scalable” in Multi-Core Context

- Typically, choose workload (e.g., multi-threaded application); run on increasing number of cores
- Plot throughput ("work completed per time") vs. number of cores
- Desired outcome: linear speedup in number of cores
- Less preferred: linear up to some K cores, then flat
- Unscalable: linear up to some K, then collapse to very low or zero
Background: Data Sharing on Multi-Core Machines

• What’s a “MESI-like protocol”?
  – Modified, Exclusive, Shared, Invalid states
  – Basically, much like Ivy DSM, but with cache lines (64 bytes) rather than VM pages (4 KB)

• Many cores can concurrently hold the same cache line and read it

• To write a cache line, writing core must have exclusive access to it (i.e., no other cores may have copy of it)
Multi-Core Sharing (cont’d)

- Communication between cores occurs when:
  - One core writes after another has read
  - One core reads after another has written

- Communication between cores may be slow
  - Interconnect among cores shared; fetch of cache line may queue behind other fetches

- False sharing

- **Conflict-free** memory accesses:
  - Set of accesses in which no core writes a cache line previously read or written by another core
  - Linear scaling as number of cores increases
Context: Prior Work on Scalable Many-Core Oses: Barrelfish

- Roscoe et al., SOSP 2009
- Modern many-core machines are distributed systems.
- Hypothesis: shared memory cannot scale to many cores, and encourages programmers to write code that cannot scale.
- Let’s design an OS as a distributed system with only explicit messaging, not shared memory, between cores.
Context: Barreelfish (cont’d)

• Principled, courageous attempt at clean-slate design
• If turns out to be necessary and sufficient, significant paradigm shift in OS design
• Design principle is leap of faith, with no evidence that it is correct (i.e., that prior OS designs and shared memory cannot scale)
• Clean-slate design means many years of hard work to determine whether viable or superior to status quo
• Forcing programmer to do message passing inconvenient; turns back on workloads with many readers, where shared memory scales fine
Context: Prior Work on Extending Linux to Many Cores

- Boyd-Wickizer et al., OSDI 2010
- Run applications on a 48-core Linux box
- What are scaling bottlenecks in kernel as we crank up from 1 to 48 cores?
- Hypothesis: we can fix them by developing more multi-core-friendly data structures for Linux kernel.
- Result: eliminated several bottlenecks in kernel, good speedup to 48 cores
Context: Many-Core Linux (cont’d)

- Pragmatic: doesn’t start by throwing out today’s OS; if successful, easy to adopt improvements
- Empirical: will reveal scaling bottlenecks in Linux if they exist, and real workarounds, if designers can come up with them
- Not final answer: if you remove bottlenecks to scale to 48 cores, how about 64? (OSDI Q: “Can you speculate about more cores?” A: “No.”)
- Might be too late: starts from Linux, but original design didn’t consider scalability to many cores
- Never know if bottleneck fundamental: if you can’t seem to speed up some kernel functionality, is it can’t scale, or because you haven’t found right design yet?
Enter Scalable Commutativity

• Do interfaces (e.g., system call APIs) limit scalability to many cores?
  – Here, “scalable” means conflict-free at cache-line granularity

• How can we determine if an interface (API) is fundamentally amenable to a scalable implementation?

• Proposition:
  If operations in an interface commute, those operations are amenable to an implementation that scales in increasing core count.
Scalable Commutativity: Intuition

• What does “commute” mean?
  – Operations are system calls
  – Regardless of their order of execution, one cannot deduce their execution order using the system call interface
  – i.e., results of system calls are indistinguishable, regardless of their execution order

• Rough idea: if ops commute, their memory accesses should be conflict-free. Their results do not depend on one another, so they should not share state.

• Conflict-free memory accesses scale on MESI-cache-coherence-like multi-core architectures

• If ops do not commute, seems their implementations should involve RAW or WAR data “dependencies”; communication overhead on MESI architectures
Why Might Scalable Commutativity Rule Be Useful?

• Consider file creation in UNIX
  – Two processes creating files in same directory
  – Can creat() be made to scale?
• Seems hard: same directory modified
• But in fact:
  – If two filenames different, creat() calls commute
  – Scalable implementation for this case:
    • Directory is hash table indexed by filename
    • One lock per hash bucket
• Rule lets you know where to concentrate effort in designing for scalability
Contribution: SIM Commutativity Definition

- **State-dependent**: whether two ops commute is with respect to state in implementation (e.g., open file table, inode contents, name-to-inode cache contents, &c.)

- **Interface-based**: ops in question are those in a specific API (in this case, OS syscalls); define “indistinguishable” only with respect to results visible in return values from API (ignoring state hidden in implementation)

- **Monotonic**: in a sequence of calls said to commute, all prefixes of sequence must commute
Why Monotonic?

• Suppose we have action sequence \( X \parallel Y_1 \parallel Y_2 \)
• It may be that \( Y_1 \parallel Y_2 \) commutes, but \( Y_1 \) alone doesn’t:
  
  \[
  Y = [A = \text{set}(1), A, B = \text{set}(2), B, C = \text{set}(2), C]
  \]

  – \( Y \) commutes in any history (every order sets value to 2)
  – But prefix of first four ops/results does not
• Can’t tell if prefix commutes until knowing future operations
• SIM Commutativity excludes such cases
…continue with Austin Clements’s SOSP 2013 slides…