EECS 570
Lecture 16
Relaxed Consistency

Winter 2017
Prof. Thomas Wenisch

http://www.eecs.umich.edu/courses/eecs570/

Announcements

• Programming Assignment 2
  □ Waypoint due today
  □ Submit via Canvas

• Project Milestone 2
  □ Delayed 1 week to 3/22
  □ Meetings 3/23 & 3/24
  □ Prepare a brief slide deck in lieu of a written report
  □ Submit via Canvas
Readings

For today:

- Boehm & Adve - Foundations of the C++ Concurrency Model

For Wednesday 3/15:

- Gharachorloo et al - Two Techniques to Enhance the performance of Memory Consistency Models - ICPP 1991
Relaxed Consistency
Review: Problems with SC

• Difficult to implement efficiently in hardware
  □ Straight-forward implementations:
    ○ No concurrency among memory access
    ○ Strict ordering of memory accesses at each node
    ○ Essentially precludes out-of-order CPUs

• Unnecessarily restrictive
  □ Most parallel programs won’t notice out-of-order accesses

• Conflicts with latency hiding techniques
Execution in strict SC

- Miss on Wr A stalls all unrelated accesses

<table>
<thead>
<tr>
<th>Wr A</th>
<th>Miss</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rd B</td>
<td>Idle</td>
</tr>
<tr>
<td>Rd C</td>
<td>Idle</td>
</tr>
<tr>
<td>Wr D</td>
<td>Idle</td>
</tr>
</tbody>
</table>

CPU pipeline

Rd E

Memory accesses issue one-at-a-time
Relaxing Write-to-Read Order

• Motivation: Post-retirement store buffers

• Allow reads to bypass incomplete writes
  □ Reads search store buffer for matching values
  □ Hides all latency of store misses in uniprocessors

• Writes are still ordered w.r.t. other writes

• Reads are still ordered w.r.t. other reads
Dekker’s Algorithm w/ Store Buffer

P1
Read B
Write A
Read
A = 1;
if (B != 0) goto retry;
/* enter critical section*/

P2
Write B
Read A
Write
B=1;
if (A != 0) goto retry;
/* enter critical section*/
Processor Consistency (PC)

• Formal requirements [Goodman 1989]:
  □ Before LOAD is performed w.r.t. any other processor, all prior LOADs must be performed
  □ Before STORE is performed w.r.t. any other processor, all prior mem ops must be performed

• Does not require store atomicity
• This is basically what x86 and VAX do
• Also roughly what IBM’s System-370 did (1960’s)
  □ Oddly, on 370, loads must stall if they hit in the SB
Sun’s “Total Store Order” (TSO)

- Formal requirements [v8 architecture manual]:
  - **Order** - There exists a partial memory order (<M) and it is *total* for all operations with store semantics
  - **Atomicity** - Atomic read-modify-writes do not allow a store between the read and write parts
  - **Termination** - All stores are performed in finite time
  - **Value** – Loads return the most recent value w.r.t. memory order (<M) and w.r.t. local program order (<p)
  - **LoadOP** – Loads are blocking
  - **StoreStore** – Stores are ordered

- This is the same as PC, except that it requires store atomicity
PC/TSO: Programmer’s Perspective

• Can occasionally lead to astonishing behavior changes
  □ E.g., Dekker’s algorithm doesn’t work
  □ ISAs provide an STBAR (store barrier) to manually force order
    ○ Semantics – store buffer must be empty before memory operations after STBAR may be executed
  □ Can also enforce order by replacing stores with RMWs

• But, the key case, where sync is done with locks, simply works
  □ Lock acquires are RMW operations ⇒
    they force order for preceding/succeeding loads/stores
    ○ Load semantics of RMW imply load-load orderings
    ○ Ditto for store semantics
  □ Lock release is a store operation ⇒
    it must be performed after critical section is done
PC/TSO: Compiler’s Perspective

• Compiler may now hoist loads across stores
  ❑ Still can’t reorder stores or move loads across loads
  ❑ Not clear how helpful this is in practice…
  ❑ …Recent results from Prof. Satish’s group:
    ○ 5-30% perf. gap vs. compiler that preserves SC  [PLDI’11]

• No new crazy memory barriers to emit
  ❑ Library/OS writers need to use a little bit of caution;
    use RMWs instead of loads/stores for synchronization
  ❑ TSO-unsafe code is rare enough that it can be the
    programmer’s problem
• No need to invoke “undefined behavior” to avoid onerous
  implementation requirements
PC/TSO: HW Perspective

- Allows a FIFO-ordered, non-coalescing store buffer
  - Typically maintains stores at word-granularity
  - Loads search buffer for matching store(s)
    - Some ISAs must deal with merging partial load matches
  - Coalescing only allowed among adjacent stores to same block
  - Must force buffer to drain on RMW and STBAR
  - Often, this is implemented in same HW structure as (speculative) store queue

- Can hide store latency!
  - But, store buffer may need to be quite big
    - Stores that will be cache hits remain buffered behind misses
  - Associative search limits scalability
    - E.g., certainly no more than 64 entries
Execution in PC / TSO

- Stores misses do not stall retirement
  - St A, St C, Ld D misses overlapped
  - Ld B retired without waiting for St A to fill
  - St C consumes space in store buffer even though it will hit

PC/TSO hides store miss latency
Relaxing Write-to-Write Order

- Motivation: Coalescing store buffers & early drain

- Allows writes to coalesce in SB & drain early
Sun’s “Partial Store Order” (PSO)

- Formal requirements [v8 architecture manual]:
  - **Order** - There exists a partial memory order (<M) and it is total for all operations with store semantics
  - **Atomicity** - Atomic read-modify-writes do not allow a store between the read and write parts
  - **Termination** - All stores are performed in finite time
  - **Value** – Loads return the most recent value w.r.t. memory order (<M) and w.r.t. local program order (<p)
  - **LoadOP** – Loads are blocking
  - **StoreStore** – Stores are ordered only if they are separated by a *membar* (memory barrier) instruction
  - **StoreStoreEq** – Stores to the same address are ordered
PSO: Compiler/HW Perspective

- Allows an unordered, coalescing post-retirement store buffer
  - Can now use a cache-block-grain set-associative structure
  - Store misses leave store buffer upon cache fill
  - Loads search buffer for matching store(s)
  - Must force buffer to drain on STBAR

- Much more efficient store buffer

- But, still doesn’t allow out-of-order loads
  - No OoO execution (without speculation)
  - Compiler’s hands are still tied
Relaxing all Order

• Motivation: Out-of-order execution & multiple load misses

• Now Ld E can complete even though earlier insn aren’t done
Two ISA Approaches to enforce order

- Approach 1: Using explicit “fence” (aka memory barrier)
  - Sun’s Relaxed Memory Order (RMO), Alpha, PowerPC, ARM
    - Ld, St, ...
    - L L S S
    - Fence ↓↓↓↓ Enforces order if bit is set
    - L S L S
    - Ld, St, ...

- Approach 2: Annotate loads/stores that do synchronization
  - Weak Ordering, Release Consistency (RC)
  - Data-Race-Free-0 (DRF0) – prog. language-level model
    - Load.acquire       Lock1
    - ...               ...
    - Store.release      Lock1
More definitions... Dependence Order

• A refinement of program order (<p)

• Dependence order (<d) captures the minimal subset of (<p) that guarantees self-consistent execution traces. X <p Y implies X <d Y if at least one of the following is true:
  □ The execution of Y is conditional on X and S(Y) (Y is a store)
  □ Y reads a register that is written by X
  □ X and Y access the same memory location and S(X) and L(Y)

• Dependence order captures what an out-of-order core is allowed to do (ignoring exceptions)
Sun’s “Relaxed Memory Order” (RMO)

• Formal requirements [v9 architecture manual]:
  1. $X <d Y \land L(X) \Rightarrow X <M Y$
     - RMO will maintain dependence order if preceding insn. is a load
  2. $M(X,Y) \Rightarrow X <M Y$
     - MEMBAR instructions order memory operations
  3. $Xa <p Ya \land S(Y) \Rightarrow X <M Y$
     - Stores to the same address are performed in program order
Execution in RMO w/ fence

- “Fence” indicates ordering affects correctness
  - Retirement stalls at Fence
  - Typically, accesses after fence don’t issue

CPU pipeline

- St A
- Fence
- Ld B
- St C

Can’t issue

- Ld D

St A | Miss
--- | ---
Ld B | Idle
St C | Idle
RMO: Programmer’s Perspective

- Programmer must specify MEMBARS wherever they are needed
  - This is hard; Specifying *minimum* barriers is harder
  - E.g., lock and unlock (from v9 ref. manual; w/o branch delays)

```
LockWithLDSTUB(lock)
  retry:  ldstub [lock],%10  
          tst %10  
          be out  
loop:   ldub [lock],%10  
          tst %10  
          bne loop  
          ba,a retry  
out:    membar #LoadLoad | #LoadStore  

UnLockWithLDSTUB(lock)
  membar #StoreStore !RMO and PSO only  
  membar #LoadStore    !RMO only  
  stub %g0,[lock]  
```
**RMO: Compiler’s Perspective**

- **Sweet, sweet freedom!**
  - Compiler may freely re-order between fences
  - Also across fences if the fence allows it (partial fences)
  - Note: still can’t reorder stores to same address
  - Programmer’s problem if the fences are wrong...
RMO: HW Perspective

• Unordered, coalescing post-retirement store buffer
  □ Just like PSO

• Out-of-order execution!
  □ Need a standard uniprocessor store queue
  □ Fence instruction implementation
    ○ Easy – stall at issue
    ○ Hard – retire to store buffer and track precise constraints
Weak Ordering (WO)

- Loads and stores can be labeled as “sync”
  - No reordering allowed across sync instructions
Release Consistency (RC)

- Specialize loads and stores as “acquires” and “releases”
  - Load at top of critical section is an acquire
  - Store at bottom of critical section is a release
  - Allows reordering **into** critical sections, but not out

Diagram:

```
weak consistency (WCsc)

release consistency (RCpc)
```

u cannot perform with respect to any other processor until u is performed.
WO/RC: Programmer Perspective

• A new way of thinking: **programmer-centric memory models**
  - If you annotate syncs correctly, your program will behave like SC
  - E.g., Data-Race-Free-0 (DRF0)
    - Accesses are either normal or sync
    - Sync accesses are sequentially consistent, and order normal accesses
    - Data races among normal accesses are prohibited
    - DRF0 programs appear as if they ran under SC

• Lot’s of ongoing work on race detection
WO/RC: Compiler’s Perspective

- DRF0 is foundation of C++ and Java memory models
  - Compiler may freely re-order between syncs
  - But, it may not introduce any new data races
    - Can’t do speculative loads (can disallow optimizations)

- Why are races so evil?
  - They break reasonable optimization:
    ```c
    unsigned int i = x;  // x is shared variable
    if (i < 2) {
        foo: ...  // suppose x changes here...
        switch (i) {
            case 0: ... break;
            case 1: ... break;
            default: ... break;  // opt: range inference tells
            }  // compiler this case is
            }  // impossible, drop the check
    ```