EECS 570
Lecture 13
Memory Consistency II

Winter 2022
Prof. Yatin Manerkar

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

Store Atomicity

A=0  B=0

P1
A=1;
P2
while (A==0);
P3
B = 1;
while (B==0);
print A;

• Intuition says: P3 prints A=1
  □ But, with caches:
    ○ A=0 initially cached at P3 in shared state
    ○ Update for A arrives at P2; sends out B=1
    ○ Update for B arrives at P3
    ○ P3 prints A=0 before invalidation from P1 arrives

• Some commercial systems allow this behavior
  □ Key issue here: store atomicity
    ○ Do new values reach all nodes at the same time?
Coherence vs. Consistency

Coherence is not sufficient to guarantee a memory model

Coherence concerns only one memory location
Consistency concerns programmer-visible ordering for all locations

Coherence is a subset of consistency

Coherence and consistency implementations are often intertwined for performance reasons (will revisit this)

Notion of a coherence-consistency interface (CCI) [Manerkar et al. MICRO 2015]
Litmus Tests

- Small 4-8 instruction programs used for MCM testing and verification
- All variables initially 0 by convention
- Think of r-variables as registers

<table>
<thead>
<tr>
<th>Message Passing (mp) litmus test</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Core 0</strong></td>
</tr>
<tr>
<td>(i1) x = 1;</td>
</tr>
<tr>
<td>(i2) y = 1;</td>
</tr>
</tbody>
</table>

**SC Forbids:** r1 = 1, r2 = 0
Naïve SC Processor Design

Requirement: Perform memory operations in program order

Assume

- coherence
- store atomicity
- + memory ordering restrictions

Memory ordering restrictions

- Processor core waits for store to complete, before issuing next memory op
- Processor core waits for load to complete, before issuing next op
Dekker's Algorithm

- Mutually exclusive access to a critical region
  - Works as advertised under sequential consistency

```c
/* initial A = B = 0 */

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

P2
B = 1;
if (A != 0) goto retry;
/* enter critical section*/
```
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
A = 1;
if (B != 0) goto retry;
/* enter critical section*/

P2
B = 1;
if (A != 0) goto retry;
/* enter critical section*/
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

Memory accesses issue one-at-a-time
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-write-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
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 \( \Rightarrow \)
    - 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 \( \Rightarrow \)
    - it must be performed after critical section is done
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
**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 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

TSO hides store miss latency
TSO Variants

- Differ in their notions of write atomicity
- IBM 370 was the same as TSO except that loads could not read from the store buffer early

Consider:

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i1) ( x = 1; )</td>
<td>(i4) ( y = 1; )</td>
</tr>
<tr>
<td>(i2) ( r1 = x; )</td>
<td>(i5) ( r3 = y; )</td>
</tr>
<tr>
<td>(i3) ( r2 = y; )</td>
<td>(i6) ( r4 = x; )</td>
</tr>
</tbody>
</table>

**IBM 370 Forbids:** \( r1 = 1, \ r2 = 0, \ r3 = 1, \ r4 = 0 \)
Processor Consistency (PC)

- [Goodman 1989]
- Basically TSO with the relaxation of store atomicity

- Consider the IRIW litmus test:

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
<th>Core 2</th>
<th>Core 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 1;</td>
<td>y = 1;</td>
<td>r1 = x;</td>
<td>r3 = y;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r2 = y;</td>
<td>r4 = x;</td>
</tr>
<tr>
<td>Allowed under PC: r1 = 1, r2 = 0, r3 = 1, r4 = 0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Variants of Store Atomicity

- Notation from [Trippel et al. ASPLOS 2017]

- MCA: a store becomes visible to all cores (including the performing core) at the same time
  - IBM 370 style

- rMCA: a store may become visible to the performing core before other cores, but once it becomes visible to one other core, it must become visible to all other cores at the same time
  - TSO style
  - commonly referred to as “multicopy atomicity” in the literature today, but terminology can vary

- nMCA: a store may become visible to different cores at different times
  - PC style
  - Can **significantly** complicate reasoning
Relaxing Write-to-Write Order

- Motivation: Coalescing store buffers & early drain

- Allows writes to coalesce in SB & drain early

Allows us to drain this
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-write operations 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 RMW and 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]:
  - $X <d Y \land L(X) \Rightarrow X <M Y$
    - RMO will maintain dependence order if preceding insn. is a load
  - $M(X,Y) \Rightarrow X <M Y$
    - MEMBAR instructions order memory operations
  - $Xa <p Ya \land S(Y) \Rightarrow X <M Y$
    - Stores to the same address are performed in program order
  - Assuming $Y$ is a load to memory location $a$,
    \[ Value(La) = Value( \text{Max}_{<m} \{ S \mid Sa <m La \text{ or } Sa <p La \} ) \]
    - where $\text{Max}_{<m}\{..\}$ selects the most recent element with respect to the memory order and where $Value()$ yields the value of a particular memory transaction.
**Execution in RMO w/ fence**

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

<table>
<thead>
<tr>
<th></th>
<th>St A</th>
<th>Miss</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fence</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ld B</td>
<td></td>
<td>Idle</td>
</tr>
<tr>
<td>St C</td>
<td></td>
<td>Idle</td>
</tr>
<tr>
<td>Ld D</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**CPU pipeline**

- St A: Retired
- St C: Idle
- Ld B: Idle
- Ld D: Can’t issue
RMO: Programmer's Perspective

- Programmer must specify MEMBARS wherever they are needed
  - This is hard; Specifying minimum barriers is harder
    - See Vsync [Oberhauser et al. ASPLOS 2021]
  - Below: lock and unlock (from v9 ref. manual; w/o branch delays)
  - RMO also does not preserve same address ld-lld ordering!

```
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)

- [Dubois et al. ISCA 1986]

- Loads and stores can be labeled as “sync”
  - No reordering allowed across sync instructions

```
Read / Write
...  
Read/Write

Synch

Read / Write
...  
Read/Write

Synch

Read / Write
...  
Read/Write

Synch

Read / Write
...  
Read/Write

Synch
```
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 showing release consistency](image)
Release Consistency (RC)

- Two flavours: RCsc and RCpc
  - In RCsc, special accesses are sequentially consistent with respect to each other
  - In RCpc, special accesses are processor consistent with respect to each other
- The difference has significant ramifications!
  - In RCpc, even adding enough synchronization to eliminate data races does not make the program behave like SC!
  - In RCpc, ordering is not maintained between releases and subsequent acquires
    - Why do this?
  - In RCpc, releases do not become visible to all cores at the same time
    - Why do this?
Data Races

A program has a **data race** if it has an execution in which two accesses to the **same address** on **different threads** where **at least one is a write** and **at least one is not a synchronization access** are not ordered by synchronization accesses.

Note: The definition of races can vary depending on the context.

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A: x = 1;</td>
<td>C: r1 = y;</td>
</tr>
<tr>
<td>B: y = 1;</td>
<td>D: z = 1;</td>
</tr>
</tbody>
</table>

Core 0

A: x = 1;
B: y = 1;

Core 1

C: r1 = y;
D: bne r1, 1, C
E: r2 = x;
WO/RC: Programmer Perspective

• A new way of thinking: programmer-centric memory models
  □ If you annotate syncs correctly, your program will behave like SC
    □ WHY?
  □ E.g., Data-Race-Free-0 (DRF0) [Adve and Hill ISCA 1990]
    □ 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
    □ Similar idea in [Gharachorloo et al. ISCA 1990]: “Properly-[Labelled]” Programs
  □ SC-for-DRF forms the basis for programming language memory models
The Ramifications of SC-for-DRF

• Hardware can freely reorder instructions in between acquire and release operations
  ☐ No one else can tell!

• Compilers can freely reorder code in between acquire and release operations
  ☐ Again, no one else can tell!

• So does this solve all memory consistency problems?
  ☐ Not quite!
  ☐ The hardware and compiler still need to maintain ISA-level MCM and PL MCM guarantees
  ☐ What about cases where we want some atomics that are not sequentially consistent? (for performance reasons)
  ☐ Next time: programming language MCMs