EECS 570
Lecture 12
Memory Consistency I

Winter 2022
Prof. Yatin Manerkar

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

Announcements

• Everyone should have received feedback on their final projects now
• We are working on being able to let you see the correct answers to your own quiz questions (after the quiz is due)
Memory Consistency
Memory Consistency Model (MCM)

A memory (consistency) model specifies the order in which memory accesses may be performed by one thread, and the order in which they become visible to other threads in the program.

It is a contract between layers of the hardware/software stack (e.g. at the ISA layer)

Loosely, the memory model specifies:

- the set of legal values a load operation can return
- the set of legal final memory states for a program
Who cares about memory models?

• Programmers want:
  - A framework for writing correct parallel programs
  - Simple reasoning - “principle of least astonishment”
  - The ability to express as much concurrency as possible

• Compiler/Language designers want:
  - To allow as many compiler optimizations as possible
  - To allow as much implementation flexibility as possible
  - To leave the behavior of “bad” programs undefined

• Hardware/System designers want:
  - To allow as many HW optimizations as possible
  - To minimize hardware requirements / overhead
  - Implementation simplicity (for verification)

We will consider all three perspectives
Uniprocessor memory model

- Loads return value of nearest preceding matched store in <p
  - Need to make partial overlaps work
  - Probably need some special cases for I/O
  - Otherwise, any sort of reordering goes!

- Programmer’s perspective:
  - Generally, no way to tell what order things actually happened

- Compiler’s perspective
  - “as-if” rule – any optimization is legal as long as it produces the same output “as-if” executed in program order
  - No “random” changes to memory values (except volatile)

- HW perspective
  - Out-of-order, store buffers, speculation all ok
  - Order only needed per-address, enforced via LSQ
Message Passing (mp) litmus test

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i1) x = 1;</td>
<td>(i3) r1 = y;</td>
</tr>
<tr>
<td>(i2) y = 1;</td>
<td>(i4) r2 = x;</td>
</tr>
<tr>
<td>SC Forbids: r1 = 1, r2 = 0</td>
<td></td>
</tr>
</tbody>
</table>

Core 0

\[
x = 1; \\
y = 1;
\]

Core 1

\[
r1 = y = 1; \\
r2 = x = 0;
\]

Memory

\[
x: 0 \\
y: y: 0
\]
MCMs as Programmer-Visible Optimizations

- Compiler optimizations can also result in weak memory behaviours
  - Example below: assume CPU performs instrs in order and 1 at a time

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4 r1 = y;</td>
</tr>
<tr>
<td>2 y = 1;</td>
<td>5 r2 = x;</td>
</tr>
<tr>
<td>3 x = 2;</td>
<td></td>
</tr>
</tbody>
</table>

Can r1 = 1 and r2 = 0?

Compiler may coalesce these 2 stores (since no same-thread reads of x in between)

Now 2 4 5 3 gives r1 = 1 and r2 = 0!
Sequential Consistency (SC)

• MCMs specify required orderings among memory operations

  • Dictate what value will be returned when your parallel program does a load!

• All parallel software and hardware need to care about MCMs!

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>① x = 1;</td>
<td>③ if (y == 1)</td>
</tr>
<tr>
<td></td>
<td>print(&quot;Answer is:&quot;);</td>
</tr>
<tr>
<td>② y = 1;</td>
<td>④ if (x == 1)</td>
</tr>
<tr>
<td></td>
<td>print(&quot;42&quot;);</td>
</tr>
</tbody>
</table>

Can it print “Answer is: 42”? Yes, eg: ① ② ③ ④

How about just “42”? Yes, eg: ① ③ ④ ②

Could it print nothing? Yes, eg: ③ ④ ① ②

These executions obey **Sequential Consistency (SC)** [Lamport79], which requires that the results of the overall program correspond to some inorder interleaving of the statements from each individual thread.
Memory Consistency Models (MCMs)

• All commercial processors weaken SC for performance

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>① x = 1;</td>
<td>③ if (y == 1)</td>
</tr>
<tr>
<td></td>
<td>print(&quot;Answer is:&quot;));</td>
</tr>
<tr>
<td>② y = 1;</td>
<td>④ if (x == 1)</td>
</tr>
<tr>
<td></td>
<td>print(&quot;42&quot;);</td>
</tr>
</tbody>
</table>

• E.g. Printing “Answer is:” (i.e. ② ③ ④ ①) would be allowed on ARM!

• Introduces additional complexity (excerpt from ARM manual below):

A dependency creates externally-visible order between a read and another Memory effect generated by the same Observer. A read R₁ is Dependency-ordered-before a read or write RW₂ from the same Observer if and only if R₁ appears in program order before RW₂ and any of the following cases apply:

<list of 6 cases>
MCMs in the Hardware/Software Stack

• MCMs specified at interfaces between layers of hardware/software stack, including:
  - Instruction Set (ISA)
  - High-level languages (e.g. C/C++, Java)

• Similar ordering verification necessary in other domains:
  - Distributed systems
  - Security
Programming Language MCMs

• What compiler optimizations are allowed?

• What orderings between program statements (and within a statement) must be maintained?
  - Do synchronization instructions need to be added when compiling?

• What sort of programs are given semantics by the language, and which are not?
  - We will discuss notions of data-race-freedom and “out-of-thin-air” values

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 ( x = 1; )</td>
<td>4 ( r1 = y; )</td>
</tr>
<tr>
<td>2 ( y = 1; )</td>
<td>5 ( r2 = x; )</td>
</tr>
<tr>
<td>3 ( x = 2; )</td>
<td></td>
</tr>
</tbody>
</table>

Can \( r1 = 1 \) and \( r2 = 0 \)?
Sequential Consistency (SC)

processors appear to perform memory ops in program order

P1 P2 P3

switch randomly set after each memory op provides total order among all operations

Memory
SC: Hardware Perspective

• 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
Non-Speculative Hardware Reorderings break SC

Message Passing (mp) litmus test

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i1) x = 1;</td>
<td>(i3) r1 = y;</td>
</tr>
<tr>
<td>(i2) y = 1;</td>
<td>(i4) r2 = x;</td>
</tr>
<tr>
<td><strong>SC Forbids:</strong> r1 = 1, r2 = 0</td>
<td></td>
</tr>
</tbody>
</table>

Core 0

x = 1;
y = 1;
x = 1;
y = 1;

Core 1

r1 = y = 1;
r2 = x = 0;
r1 = y;
r2 = x;

Memory

x: 0
y: 1: 0
SC: Programmer’s Perspective

- Generally the least astonishing alternative
  - Looks a lot like a multitasking uniprocessor
  - Memory behaves as intuition would suggest
  - Causality is maintained (SC implies store atomicity)

- But, still plenty of rope to hang yourself
  - Any memory access is potentially a synchronization
  - Arbitrary wild memory races are legal
  - There is still weirdness with C/C++ bit fields

- And it’s overkill
  - Most programmers use libraries for sync...
  - ...hence, they don’t actually need SC guarantees
  - SC-for-DRF guarantee generally allows you to have your cake and eat it too
SC: Compiler’s Perspective

• Disaster! Nearly all optimizations initially appear illegal!
  □ Anything could be a sync ⇒ no mem ops may be reordered
  □ Effectively disallows:
    ◆ Loop invariant code motion
    ◆ Common sub-expression elimination
    ◆ Register allocation
    ◆ ...

• Not quite that bad...
  □ C/C++ specify order only across sequence points (statements)
    ◆ Operations within an expression may be reordered

• Conflict analysis can improve things [Shasa & Snir’88]
  □ Static analysis identifies conflicting (racing) accesses
  □ Can determine the minimal set of delays to enforce SC
  □ But, needs perfect whole-program analysis
Fixing SC Performance

- **Option 1: Change the memory model**
  - Weak/Relaxed Consistency
  - Programmer specifies when order matters
    - Other access happen concurrently/out-of-order
  - Simple hardware can yield high performance
    - Programmer must reason under counter-intuitive rules

- **Option 2: Speculatively ignore ordering rules**
  - In-window Speculation & InvisiFence
    - Order matters only if re-orderings are observed
      - Ignore the rules and hope no-one notices
      - Works because data races are rare
  - Performance of relaxed consistency with simple programming model
    - More sophisticated HW; speculation can be tricky to always get right
Review: Coherence

A Memory System is Coherent if
- there is a total order on all stores to any given address
- every core sees the stores to a given address in the same total order

There is broad consensus that coherence is a good idea.

But, that is not enough for consistency...
Program Order Relaxations

A=0  flag=0

Processor 0  Processor 1
A=1;  while (!flag); // spin
flag=1;  print A;

- **Intuition says**: P1 prints A=1

- **Coherence says**: absolutely nothing
  - P1 can see P0’s write of `flag` before write of `A`!! How?
    - P0 has a coalescing store buffer that reorders writes
    - Or out-of-order execution
    - Or compiler re-orders instructions

- Imagine trying to figure out why this code sometimes “works” and sometimes doesn’t
  - Need for formal verification of memory consistency (will come back to this)

- **Real systems** act in this strange manner
  - What is allowed is defined as part of the ISA of the processor
Store Atomicity

\[ A=0 \quad B=0 \]

\begin{align*}
P1 & \quad P2 & \quad P3 \\
A=1; & \quad \text{while } (A==0); & \quad \text{while } (B==0); \\
B = 1; & \quad \text{print } A;
\end{align*}

- 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

Message Passing (mp) litmus test

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

/* 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

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

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 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
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
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) (\text{x} = 1;)</td>
<td>(i4) (\text{y} = 1;)</td>
</tr>
<tr>
<td>(i2) (\text{r1} = \text{x};)</td>
<td>(i5) (\text{r3} = \text{y};)</td>
</tr>
<tr>
<td>(i3) (\text{r2} = \text{y};)</td>
<td>(i6) (\text{r4} = \text{x};)</td>
</tr>
</tbody>
</table>

*IBM 370 Forbids: \(\text{r1} = 1, \text{r2} = 0, \text{r3} = 1, \text{r4} = 0\)*
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>
</tbody>
</table>

Allowed under PC: \(r1 = 1, r2 = 0, r3 = 1, r4 = 0\)
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