Lecture 14:
Memory Consistency

Parallel Computer Architecture and Programming
CMU 15-418/15-618, Spring 2019
What is Correct Behavior for a Parallel Memory Hierarchy?

- Note: side-effects of writes are only observable when reads occur
  - so we will focus on the values returned by reads

- Intuitive answer:
  - reading a location should return the latest value written (by any thread)

- Hmm... what does “latest” mean exactly?
  - within a thread, it can be defined by program order
  - but what about across threads?
    - the most recent write in physical time?
      - hopefully not, because there is no way that the hardware can pull that off
        » e.g., if it takes >10 cycles to communicate between processors, there is no way that processor 0 can know what processor 1 did 2 clock ticks ago
    - most recent based upon something else?
      - Hmm...
Refining Our Intuition

### Thread 0

```java
// write evens to X
for (i=0; i<N; i+=2) {
    X = i;
    ...
}
```

(Assume: X=0 initially, and these are the only writes to X.)

### Thread 1

```java
// write odds to X
for (j=1; j<N; j+=2) {
    X = j;
    ...
}
```

### Thread 2

```java
...  
A = X;
...  
B = X;
...  
C = X;
...  
```

- What would be some clearly **illegal combinations** of \( (A,B,C) \)?
- How about:
  
  \[
  (4,8,1)\quad (9,12,3)\quad (7,19,31)\]

- What can we generalize from this?
  - writes from any **particular thread** must be **consistent with program order**
    - in this example, observed even numbers must be increasing (ditto for odds)
  - **across threads**: writes must be consistent with a **valid interleaving of threads**
    - not physical time! (programmer cannot rely upon that)
Visualizing Our Intuition

- Each thread proceeds in program order
- Memory accesses interleaved (one at a time) to a single-ported memory
  - rate of progress of each thread is unpredictable

```
// write evens to X
for (i=0; i<N; i+=2) {
    X = i;
    ...
}

// write odds to X
for (j=1; j<N; j+=2) {
    X = j;
    ...
}
```

```java
Thread 0
```

```java
Thread 1
```

```java
Thread 2
```

CPU 0

CPU 1

CPU 2

Single port to memory

Memory
Correctness Revisited

Thread 0

// write evens to X
for (i=0; i<N; i+=2) {
    X = i;
    ...
}

Thread 1

// write odds to X
for (j=1; j<N; j+=2) {
    X = j;
    ...
}

Thread 2

... A = X;
... B = X;
... C = X;
...

CPU 0

CPU 1

CPU 2

Memory

Single port to memory

Recall: “reading a location should return the latest value written (by any thread)”
→ “latest” means consistent with some interleaving that matches this model
  – this is a hypothetical interleaving; the machine didn’t necessarily do this!
Part 2 of Memory Correctness: Memory Consistency Model

1. "Cache Coherence"
   – do all loads and stores to a given cache block behave correctly?

2. "Memory Consistency Model" (sometimes called "Memory Ordering")
   – do all loads and stores, even to separate cache blocks, behave correctly?

Recall: our intuition
Why is this so complicated?

• **Fundamental issue:**
  – loads and stores are very expensive, even on a uniprocessor
    • can easily take 10’s to 100’s of cycles

• **What programmers intuitively expect:**
  – processor atomically performs *one instruction at a time, in program order*

• **In reality:**
  – if the processor actually operated this way, it would be painfully slow
  – instead, the processor *aggressively reorders instructions* to hide memory latency

• **Upshot:**
  – *within a given thread*, the processor preserves the *program order illusion*
  – but this illusion has *nothing to do with what happens in physical time!*
  – from the perspective of *other threads*, all bets are off!
Hiding Memory Latency is Important for Performance

• **Idea**: overlap memory accesses with other accesses and computation

• Hiding *write* latency is simple in uniprocessors:
  – add a *write buffer*
  – (more on this later)

• (But this affects *correctness* in multiprocessors)
How Can We Hide the Latency of Memory Reads?

“Out of order” pipelining:
– when an instruction is stuck, perhaps there are subsequent instructions that can be executed

```
x = *p;
y = x + 1;
z = a + 2;
b = c / 3;
```

• **Implication:** memory accesses may be performed out-of-order!!!
What About Conditional Branches?

• Do we need to wait for a conditional branch to be resolved before proceeding?
  – No! Just predict the branch outcome and continue executing speculatively.
  • if prediction is wrong, squash any side-effects and restart down correct path

```c
x = *p;
y = x + 1;
z = a + 2;
b = c / 3;
if (x != z)
  d = e - 7;
else d = e + 5;
...  
```
How Out-of-Order Pipelining Works in Modern Processors

- Fetch and graduate instructions in-order, but issue out-of-order

- Intra-thread dependences are preserved, but memory accesses get reordered!
Imagine that each instruction within a thread is a gas particle inside a twisty balloon.

- They were numbered originally, but then they start to move and bounce around.
- **When a given thread observes memory accesses from a *different* thread:**
  - those memory accesses can be (almost) arbitrarily jumbled around
    - like trying to locate the position of a particular gas particle in a balloon.
- As we’ll see later, the only thing that we can do is to put *twists* in the balloon.
Uniprocessor Memory Model

- **Memory model** specifies **ordering constraints among accesses**
- **Uniprocessor model**: memory accesses **atomic and in program order**

- Not necessary to maintain sequential order for correctness
  - **hardware**: buffering, pipelining
  - **compiler**: register allocation, code motion

- **Simple for programmers**

- **Allows for high performance**
In Parallel Machines (with a Shared Address Space)

• Order between accesses to different locations becomes important

\[(Initially \ A \ and \ Ready = 0)\]

\[
P1
\]

\[
A = 1;
\]

\[
\text{Ready} = 1;
\]

\[
\text{while } (\text{Ready} \neq 1);
\]

\[
\ldots = A;
\]

\[
P2
\]
How Unsafe Reordering Can Happen

- Distribution of memory resources
  - accesses issued in order may be observed out of order
Caches Complicate Things More

- Multiple copies of the same location

\[
A = 1; \quad \text{wait} (A == 1); \\
B = 1; \quad \text{wait} (B == 1); \\
\ldots = A; \\
\]

Oops!
Our Intuitive Model: “Sequential Consistency” (SC)

- Formalized by Lamport (1979)
  - accesses of each processor in program order
  - all accesses appear in sequential order

- Any order implicitly assumed by programmer is maintained
Example with Sequential Consistency

Simple Synchronization:

P0
A = 1
Ready = 1

P1
x = Ready
y = A

• all locations are initialized to 0
• possible outcomes for (x,y):
  – (0,0), (0,1), (1,1)
• (x,y) = (1,0) is not a possible outcome (i.e. Ready = 1, A = 0):
  – we know a->b and c->d by program order
  – b->c implies that a->d
  – y==0 implies d->a which leads to a contradiction
  – but real hardware will do this!
Another Example with Sequential Consistency

Stripped-down version of a 2-process mutex (minus the turn-taking):

\[
\begin{align*}
\text{P0} & \\
\text{want}[0] &= 1 \quad (a) \\
x &= \text{want}[1] \quad (b)
\end{align*}
\]

\[
\begin{align*}
\text{P1} & \\
\text{want}[1] &= 1 \quad (c) \\
y &= \text{want}[0] \quad (d)
\end{align*}
\]

- all locations are initialized to 0
- possible outcomes for \((x,y)\):
  - \((0,1), (1,0), (1,1)\)
- \((x,y) = (0,0)\) is not a possible outcome (i.e. \(\text{want}[0] = 0, \text{want}[1] = 0\)):
  - \(a\rightarrow b\) and \(c\rightarrow d\) implied by program order
  - \(x = 0\) implies \(b\rightarrow c\) which implies \(a\rightarrow d\)
  - \(a\rightarrow d\) says \(y = 1\) which leads to a contradiction
  - similarly, \(y = 0\) implies \(x = 1\) which is also a contradiction
  - *but real hardware will do this!*
One Approach to Implementing Sequential Consistency

1. Implement cache coherence
   → writes to the same location are observed in same order by all processors

2. For each processor, delay start of memory access until previous one completes
   → each processor has only one outstanding memory access at a time

• What does it mean for a memory access to complete?
When Do Memory Accesses Complete?

- **Memory Reads**: a read completes when its return value is bound

\[
\text{load } r1 \leftarrow x \\
\text{r1} = 17
\]

(Find \( x \) in memory system)

\[
x = 17
\]
When Do Memory Accesses Complete?

- **Memory Reads:**
  - a read completes when its return value is bound

- **Memory Writes:**
  - a write completes when the new value is “visible” to other processors

  store 23 \(\to\) \(x\)

  \(x = 23\)

  \((\text{Commit to memory order})\)

  \((\text{aka “serialize”})\)

- What does “visible” mean?
  - it does NOT mean that other processors have necessarily seen the value yet
  - it means the new value is committed to the hypothetical serializable order (HSO)
    - a later read of \(x\) in the HSO will see either this value or a later one
  - (for simplicity, assume that writes occur atomically)
Summary for Sequential Consistency

- Maintain order between shared accesses in each processor

- Balloon analogy:
  - like putting a twist between each individual (ordered) gas particle

- Severely restricts common hardware and compiler optimizations
Performance of Sequential Consistency

- Processor issues accesses **one-at-a-time** and **stalls for completion**

- **Low processor utilization (17% - 42%)** even with caching

Alternatives to Sequential Consistency

- Relax constraints on memory order

**Total Store Ordering (TSO)** (Similar to Intel)


**Partial Store Ordering (PSO)**
• Can use a write buffer
• Write latency is effectively hidden
But Can Programs Live with Weaker Memory Orders?

- “Correctness”: same results as sequential consistency
- Most programs don’t require strict ordering (all of the time) for “correctness”

**Program Order**

\[ \text{A} = 1; \quad \text{B} = 1; \quad \text{unlock L}; \quad \text{lock L}; \quad \ldots = \text{A}; \quad \ldots = \text{B}; \]

**Sufficient Order**

\[ \text{A} = 1; \quad \text{B} = 1; \quad \text{unlock L}; \quad \text{lock L}; \quad \ldots = \text{A}; \quad \ldots = \text{B}; \]

- But how do we know when a program will behave correctly?
Identifying Data Races and Synchronization

• Two accesses *conflict* if:
  – (i) access same location, and (ii) at least one is a *write*

• Order accesses by:
  – program order (*po*)
  – dependence order (*do*): op1 --> op2 if op2 reads op1

```
P1
| Write A |
|   po   |
| Write Flag |
```

```
P2
| Read Flag |
|   po     |
| Read A   |
```

• **Data Race:**
  – two conflicting accesses on different processors
  – not ordered by intervening accesses

• **Properly Synchronized Programs:**
  – all synchronizations are explicitly identified
  – all data accesses are ordered through synchronization
Optimizations for Synchronized Programs

• **Intuition**: many parallel programs have mixtures of “private” and “public” parts*
  – the “private” parts must be protected by synchronization (e.g., locks)
  – can we take advantage of synchronization to improve performance?

![Diagram showing synchronization example]

**Example:**

- **Grab a lock**
- **Insert node into data structure**
  - Essentially a “private” activity; reordering is ok
- **Release the lock**
  - Now we make it “public” to the other nodes

*Caveat: shared data is in fact always visible to other threads.*
Optimizations for Synchronized Programs

• Exploit information about synchronization

```
READ/WRITE
READ/WRITE
SYNCH
READ/WRITE
READ/WRITE
SYNCH
READ/WRITE
READ/WRITE
```

**Between** synchronization operations:
- we can allow reordering of memory operations
- *(as long as intra-thread dependences are preserved)*

**Just before and just after** synchronization operations:
- thread must wait for all prior operations to complete

“Weak Ordering” (WO)

• properly synchronized programs should yield the same result as on an SC machine
Intel’s MFENCE (Memory Fence) Operation

- An **MFENCE** operation enforces the ordering seen on the previous slide:
  - does not begin until all prior reads & writes from that thread have completed
  - no subsequent read or write from that thread can start until after it finishes

**Balloon analogy:** it is a twist in the balloon
- no gas particles can pass through it

**Good news:** \texttt{xchg} does this implicitly!
Implementing Lock with Xchg

\[
x\text{chg}(\text{mem, reg}) = \begin{cases} 
\text{temp} &= \ast\text{mem}; \\
\ast\text{mem} &= \text{reg}; \\
\text{reg} &= \text{temp}; 
\end{cases} \quad \text{Done atomically}
\]

**acquire()**:

while (1) {
    reg = 1;
    xchg(&lock, reg);
    if (reg == 0)
        break;
}

**release()**:

reg = 0;
            xchg(&lock, reg);
ARM Processors

• ARM processors have a very relaxed consistency model

• ARM has some great examples in their programmer’s reference:

• A great list regarding relaxed memory consistency in general:
  – http://www.cl.cam.ac.uk/~pes20/weakmemory/
Common Misconception about MFENCE

- MFENCE operation does NOT push values out to other threads
  - it is not a magic “make every thread up-to-date” operation
- It simply stalls the thread that performs the MFENCE until write buffer empty

MFENCE operations create partial orderings
- that are observable across threads
Earlier (Broken) Example Revisited

Where exactly should we insert MFENCE operations to fix this?

\[
\begin{align*}
\text{P0} & \quad \text{P1} \\
[1: \text{Here?}] & \quad [4: \text{Here?}] \\
A = 1 & \quad x = \text{Ready} \\
[2: \text{Here?}] & \quad [5: \text{Here?}] \\
\text{Ready} = 1 & \quad y = A \\
[3: \text{Here?}] & \quad [6: \text{Here?}]
\end{align*}
\]
Earlier (Broken) Example Revisited

Where exactly should we insert MFENCE operations to fix this?

\[ P0 \]
\[ A = 1 \]
\[ MFENCE \]
\[ Ready = 1 \]
\[ MFENCE \]

\[ P1 \]
\[ [1: Here?] \]
\[ x = Ready \]
\[ y = A \]
\[ [6: Here?] \]
Exploiting Asymmetry in Synchronization: “Release Consistency”

- **Lock operation**: only gains ("acquires") permission to access data
- **Unlock operation**: only gives away ("releases") permission to access data

**Weak Ordering (WO)**

1. READ/WRITE
   - READ/WRITE
   - READ/WRITE
2. LOCK
   - READ/WRITE
   - READ/WRITE
3. UNLOCK
   - READ/WRITE
   - READ/WRITE

**Overly Conservative**

1. READ/WRITE
   - READ/WRITE
   - READ/WRITE
2. ACQUIRE
   - READ/WRITE
   - READ/WRITE
3. RELEASE
   - READ/WRITE
   - READ/WRITE

**Release Consistency (RC)**

Make sure writes completed before exit critical section
Make sure don’t read/write shared state until lock acquired
Intel’s Full Set of Fence Operations

• In addition to MFENCE, Intel also supports two other fence operations:
  – LFENCE: serializes only with respect to load operations (not stores!)
  – SFENCE: serializes only with respect to store operations (not loads!)
    • Note: It does slightly more than this; see the spec for details:

• In practice, you are most likely to use:
  – MFENCE
  – xchg
Earlier (Broken) Example Revisited

Where exactly should we insert FENCE operations to fix this?

P0

[1: Here?]

A = 1

SFENCE

Ready = 1

[3: Here?]

P1

[4: Here?]

x = Ready

LFENCE

y = A

[6: Here?]
Take-Away Messages on Memory Consistency Models

- **DON’T** use only normal memory operations for synchronization
  - e.g., Peterson’s solution for mutual exclusion
    
    ```
    boolean want[2] = {false, false};
    int turn = 0;

    want[i] = true;
    turn = 1-i;
    while (want[1-i] && turn == 1-i)
        continue;
    ...
    ... critical section ...
    want[i] = false;
    ```

- **DO** use either explicit synchronization operations (e.g., `xchg`) or fences
  
  ```
  while (!xchg(&lock_available, 0)
      continue;
  ...
  ... critical section ...
  xchg(&lock_available, 1);
  ```

Exercise for the reader: Where should we add fences (and which type) to fix this?
Summary: Relaxed Consistency

• Motivation:
  – obtain higher performance by allowing reordering of memory operations
    • (reordering is not allowed by sequential consistency)

• One cost is software complexity:
  – the programmer or compiler must insert synchronization
    • to ensure certain specific orderings when needed

• In practice:
  – complexities often encapsulated in libraries that provide intuitive primitives
    • e.g., lock/unlock, barriers (or lower-level primitives like fence)

• Relaxed models differ in which memory ordering constraints they ignore