Parallelism: Memory Consistency Models

Todd C. Mowry, Dave Eckhardt & Dave O’Hallaron
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

- (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

```c
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;
...
```

if hardware guesses that this is true then execute “then” part (speculatively) (without waiting for x or z)
How Out-of-Order Pipelining Works in Modern Processors

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

Branch Predictor

Inst. Cache

PC: 0x1c

0x1c: \( b = c / 3 \);
0x18: \( z = a + 2 \);
0x14: \( y = x + 1 \);
0x10: \( x = *p \);

Reorder Buffer

- Issue (out-of-order)
- Issue (out-of-order)
- Can’t issue
- Issue (cache miss)

- 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

```
write A
write B
read A
read B
```

Reads check for matching addresses in write buffer

![Diagram of processor and cache](attachment:diagram.png)
In Parallel Machines (with a Shared Address Space)

- Order between accesses to different locations becomes important

\[(\text{Initially } A \text{ and } \text{Ready} = 0)\]

\begin{align*}
\textbf{P1} & \quad \textbf{P2} \\
A &= 1; \\
\text{Ready} &= 1; \\
\text{while } (\text{Ready} \neq 1); \\
... &= A;
\end{align*}
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

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

```
A = 1;
wait(A == 1);
B = 1;
wait(B == 1);
...
```

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
Simple Synchronization:

\begin{align*}
\text{P0} & \\
A &= 1 \quad (a) \\
\text{Ready} &= 1 \quad (b) \\
\text{P1} & \\
x &= \text{Ready} \quad (c) \\
y &= A \quad (d)
\end{align*}

- 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. \(\text{Ready} = 1, A = 0\)):
  - we know \(a \rightarrow b\) and \(c \rightarrow d\) by program order
  - \(b \rightarrow c\) implies that \(a \rightarrow d\)
  - \(y = 0\) implies \(d \rightarrow a\) 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):

\[
\begin{align*}
\text{P0} & \quad \text{P1} \\
\text{want}[0] &= 1 \quad & \text{want}[1] &= 1 \\
x &= \text{want}[1] \quad & y &= \text{want}[0]
\end{align*}
\]

- 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->b and c->d implied by program order
  - \(x = 0\) implies b->c which implies a->d
  - a->d says \(y = 1\) which leads to a contradiction
  - similarly, \(y = 0\) implies \(x = 1\) which is also a contradiction
  - \textit{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*

```
load r1 ← x
```

\[ X = ??? \]

\[ X = 17 \]

\[ r1 = 17 \]

*(Find \( X \) in memory system)*
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

  \[
  \text{store } 23 \rightarrow x
  \]

  \[
  x = 23
  \]

  (Commit to memory order)

  (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

  ![Diagram showing READ and WRITE operations with arrows indicating order]

  Don’t start until previous access completes

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

• Severely restricts common hardware and compiler optimizations

Carnegie Mellon
Performance of Sequential Consistency

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

![Bar chart showing normalized execution time for MP3D, LU, and PTHOR]

- **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**

```
A = 1;
B = 1;
unlock L;
lock L;
... = A;
... = B;
```

**Sufficient Order**

```
A = 1;
B = 1;
unlock L;
lock L;
... = A;
... = 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

- **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?

*Example:*

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

- **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

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: **xchg** does this implicitly!
Common Misconception about MFENCE

- MFENCE operations **do NOT push values out to other threads**
  - it is not a magic “make every thread up-to-date” operation
- Instead, they simply **stall the thread that performs the MFENCE**

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

Where exactly should we insert MFENCE operations to fix this?

P0

[1: Here?]
A = 1

[2: Here?]
Ready = 1

[3: Here?]

P1

[4: Here?]
x = Ready

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

- **DON’T** use only normal memory operations for synchronization
  - e.g., Peterson’s solution (from Synchronization #1 lecture)
    
    ```java
    boolean want[2] = {false, false};
    int turn = 0;

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

- **DO** use either explicit synchronization operations (e.g., `xchg`) or fences (on x86)
  
  ```java
  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?
Take-Away Messages, Continued

• Beyond x86, *synchronization doesn’t necessarily imply a fence!*
  – on ARM and Power, you may still need to combine LL/SC with a fence
  – the fact that xchg performs a full-fence is x86-specific

• There are weaker versions of fences (e.g., Intel’s LFENCE and SFENCE), but...
  – they probably cause more harm than good
    • very good chance of accidentally introducing bugs
    • performance gains are questionable
Impact of Compiler Optimizations

• Compilers like to move memory references around!
  – e.g., move loads out of a loop

• Modern programming languages provide facilities for constraining this
  – Old school: volatile
  – C: `<stdatomic.h>`
  – Linux: `READ_ONCE(p), WRITE_ONCE(p,v)`
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**