Parallelism: Memory Consistency Models

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

1. Memory Consistency Models
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

![Diagram showing three CPUs connected to a single port to memory]
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
  
  
  \[
  \text{write A} \quad \text{read B} \quad \text{write A} \quad \text{read B}
  \]

- 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

\[
\begin{align*}
x &= *p; \\
y &= x + 1; \\
z &= a + 2; \\
b &= c / 3;
\end{align*}
\]

- suffers expensive cache miss
- stuck waiting on true dependence
- these do not need to wait

• 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 decode instructions in-order, but issue out-of-order

- Intra-thread dependences are preserved, but memory accesses get reordered!

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

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

  \textit{(Initially A and Ready = 0)}

  \[
  P_1 \quad \text{Ready} = 1; \quad P_2
  \]

  \[
  A = 1; \quad \text{while (Ready } \neq 1); \quad \text{... } = A;
  \]
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

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

Interconnection Network
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:**

<table>
<thead>
<tr>
<th>P0</th>
<th>P1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A  = 1 (a)</td>
<td>x = Ready (c)</td>
</tr>
<tr>
<td>Ready = 1 (b)</td>
<td>y = A (d)</td>
</tr>
</tbody>
</table>

- 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):

\[
\begin{align*}
P_0 & \\
\text{want}[0] &= 1 \quad (a) \\
x &= \text{want}[1] \quad (b)
\end{align*}
\]

\[
\begin{align*}
P_1 & \\
\text{want}[1] &= 1 \quad (c) \\
y &= \text{want}[0] \quad (d)
\end{align*}
\]

- 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
  - \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

- 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)

\[
\text{store } 23 \rightarrow x, \quad x = 23
\]

(Commit to memory order)

(aka “serialize”)
Summary for Sequential Consistency

• Maintain order between shared accesses in each processor

  READ
  ↓
  READ

  READ
  ↓
  WRITE

  WRITE
  ↓
  READ

  WRITE
  ↓
  WRITE

  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
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

**Performance Impact of TSO vs. SC**

- "Base" = SC
- "WR" = TSO

<table>
<thead>
<tr>
<th>Processor</th>
<th>Cache</th>
<th>READS</th>
<th>WRITES</th>
<th>write buffer</th>
</tr>
</thead>
</table>

The diagram shows normalized execution time for different scenarios, indicating the performance impact of TSO compared to SC. The write buffer helps in reducing write latency.
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

\[
\begin{align*}
A &= 1; \\
\downarrow \\
B &= 1; \\
\downarrow \\
\text{unlock } L; & \quad \text{lock } L; \\
\downarrow \\
\ldots &= A; \\
\downarrow \\
\ldots &= B;
\end{align*}
\]

Sufficient Order

\[
\begin{align*}
A &= 1; \\
\downarrow \\
B &= 1; \\
\downarrow \\
\text{unlock } L; & \quad \text{lock } L; \\
\downarrow \\
\ldots &= A; \\
\downarrow \\
\ldots &= B;
\end{align*}
\]

• 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:*

- **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
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

![Diagram showing the process of locking and unlocking with read/write operations and the resulting states in Weak Ordering (WO) and Release Consistency (RC)].
More Relaxed Consistency

Thread 0

\[ a: W[x] = 1 - \]
More Relaxed Consistency

Thread 0

a: \(W[x] = 1\)

Thread 1

b: \(R[x] = 1\)

c: \(W[y] = 1\)

\(rf\)

\(po\)
More Relaxed Consistency

Thread 0
a: \( W[x] = 1 \)

Thread 1
b: \( R[x] = 1 \)
c: \( W[y] = 1 \)
po

Thread 2
d: \( R[y] = 1 \)
e: \( R[x] = 0 \)
rf
rf
rf
po
More Relaxed Consistency

Test \( WRC + \text{sync} + \text{addr} : \text{Forbidden} \)
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;
```

• **DON’T** use synchronization operations except when necessary
  – **Recall:** you have likely never seen this issue before today
Take-Away Messages on Memory Consistency Models

• **DO** use either explicit synchronization operations (e.g., `xchg`) and/or* fences

  ```c
  while (!xchg(&lock_available, 0)
    continue;
  ... critical section ...
  xchg(&lock_available, 1);
  ```

• **DO** utilize the capabilities provided by your language
  – C has (optionally) `stdatomic.h`
  – Can also use volatile and hardware fences

*Not all ISAs treat synchronization operations as fences*
Summary

• Memory Consistency Models
  – Be sure to use fences or explicit synchronization operations when ordering matters
    • don’t synchronize through normal memory operations!