Memory Consistency Models

John Mellor-Crummey
Department of Computer Science
Rice University
johnmc@cs.rice.edu
From Coherence to Consistency

• Coherence
  — focus: visible values of an individual variable
  — problems can arise if multiple actors (e.g., multiple cores) have access to multiple copies of a datum (e.g., in multiple caches) and at least one access is a write
    – must appear to be one and only one value per memory location
  — access to stale data (incoherence) is prevented using a coherence protocol
    – set of rules implemented by the distributed actors within a system

• Consistency models
  — focus: visible values for multiple variables
  — define correct shared memory behavior in terms of loads and stores (memory reads and writes)
    – independent of caches or coherence
  — can stores be seen out of order? if so, under what conditions?
    – a spectrum of alternatives
      sequential consistency to weak memory models
Initially all pointers = null, all integers = 0.

P1

while (there are more tasks) {
    Task = GetFromFreeList();
    Task → Data = ...;
    insert Task in task queue
}

Head = head of task queue;

P2, P3, ..., Pn

while (MyTask == null) {
    Begin Critical Section
    if (Head != null) {
        MyTask = Head;
        Head = Head → Next;
    }
    End Critical Section
}

... = MyTask → Data;
Memory Consistency Model

• Memory model
  —formal specification of how shared memory will appear to programmers

• Consistency
  —restricts values that can be returned by a read during execution

• Why memory consistency models? Eliminate gap between
  —expected behavior
  —behavior supported by a system
Impact of Memory Models

• Programmability
  — programmers must reason about allowable behaviors
    – surprisingly subtle!

• Performance
  — determines what reorderings of loads and stores are legal
    – hardware
    – compiler

• Portability
  — different systems implement different memory models
Multiple Levels of Memory Models

• Machine level
  —affects hardware design (processor, memory, interconnect)
  —affects assembly-code programmer

• Language level
  —affects both designers and users of high-level languages
Memory Models for Uniprocessors

- Memory operations
  - occur one at a time
  - in order specified by program (program order)
- Simple, intuitive sequential semantics for memory
- Expectation
  - read of X will return value of last write (in program order) to X

In practice: a uniprocessor can relax strict ordering
- suffices to maintain control and data dependences
- order constrained only when
  - same location
  - one controls execution of other
Why Relax Strict Ordering?

Overlapping and reordering memory accesses enables a range of hardware and software optimizations

- **Compiler optimizations**
  - register allocation
  - code motion
  - loop transformations

- **Hardware optimizations**
  - pipelining
  - multiple issue
  - write buffer bypassing
  - forwarding a value from one cache to another
  - lockup-free caches: don’t delay accesses that follow a miss
A Memory Model for Multiprocessors?

- Intuitively, a read of a memory location should return the value of its “last” write
- Natural for uniprocessors
- Not obvious what this means for multiprocessors with concurrent operations
- Idea: require that all memory operations appear to execute one at a time, and the operations of a single processor appear to execute in the order described by that processor’s program
Sequential Consistency

- Intuitive memory model defined by Lamport [1979]
- Result of an execution appears as if
  - all operations appear as if executed in some sequential order
  - memory operations of each thread appear in program order

simple memory system: no caches, no write buffers
Consider the Following ...

<table>
<thead>
<tr>
<th>Initially, $x == y == 0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 1</td>
</tr>
<tr>
<td>1: $r2 = x$;</td>
</tr>
<tr>
<td>2: $y = 1$;</td>
</tr>
</tbody>
</table>

After all statements execute, could $r2 == 2$ and $r1 == 1$?
Consider the Following …

Initially, \( x == y == 0 \)

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

After all statements execute, could \( r2 == 2 \) and \( r1 == 1 \)?

Possible interleavings
Consider the Following …

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

After all statements execute, could $r2 == 2$ and $r1 == 1$?

Possible interleavings

$1, 2, 3, 4$ ?
Consider the Following …

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

After all statements execute, could r2 == 2 and r1 == 1?

Possible interleavings

1, 2, 3, 4 ? ✓
Consider the Following …

Initially, \( x == y == 0 \)

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

After all statements execute, could \( r2 == 2 \) and \( r1 == 1 \)?

Possible interleavings

- 1, 2, 3, 4 ? ✓
- 1, 3, 2, 4 ?
Consider the Following …

<table>
<thead>
<tr>
<th>Initially, $x == y == 0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 1</td>
</tr>
<tr>
<td>1: $r_2 = x$;</td>
</tr>
<tr>
<td>2: $y = 1$;</td>
</tr>
<tr>
<td>Thread 2</td>
</tr>
<tr>
<td>3: $r_1 = y$;</td>
</tr>
<tr>
<td>4: $x = 2$</td>
</tr>
</tbody>
</table>

After all statements execute, could $r_2 == 2$ and $r_1 == 1$?

Possible interleavings

- $1, 2, 3, 4$? ✓
- $1, 3, 2, 4$? ✓
Consider the Following ...  

Initially, \( x == y == 0 \)

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

After all statements execute, could \( r2 == 2 \) and \( r1 == 1 \)?

Possible interleavings

- 1, 2, 3, 4 ? ✓
- 1, 3, 2, 4 ? ✓
- 3, 4, 1, 2 ?
Consider the Following ...

Initially, \( x == y == 0 \)

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

After all statements execute, could \( r2 == 2 \) and \( r1 == 1 \)?

Possible interleavings

- 1, 2, 3, 4 ? ✓
- 1, 3, 2, 4 ? ✓
- 3, 4, 1, 2 ? ✓
Consider the Following …

Initially, \( x == y == 0 \)

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

After all statements execute, could \( r2 == 2 \) and \( r1 == 1 \)?

Possible interleavings

- 1, 2, 3, 4 ? ✓
- 1, 3, 2, 4 ? ✓
- 3, 4, 1, 2 ? ✓
- 4, 1, 2, 3 ?
Consider the Following …

Initially, \( x == y == 0 \)

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

After all statements execute, could \( r2 == 2 \) and \( r1 == 1 \)?

Possible interleavings

- 1, 2, 3, 4 ? ✓
- 1, 3, 2, 4 ? ✓
- 3, 4, 1, 2 ? ✓
- 4, 1, 2, 3 ? Sequential consistency would not allow this
Does Program Order Really Matter?

Both threads could enter critical section

— if the hardware allows a thread’s read to complete before a prior write completes
— if the compiler reorders the thread’s read and write
Implications of Sequential Consistency

• Assumption: memory atomicity
  — memory operations cannot overlap

• Impact
  — limits aggressive hardware designs
  — limits compiler optimizations

• Result: severely hampers performance
Write Buffers (without Caches)

1. W=>R order using write buffers
   — write buffer with bypassing hides latency of writes
   — reads of different locations can bypass pending writes
Initially: Flag1=Flag2=0.

Can write buffers allow an ordering that violates sequential consistency?

1. W=>R order using **write buffers**
   - write buffer with bypassing hides latency of writes
   - reads of different locations can bypass pending writes
Write Buffers (without Caches)

1. \texttt{W=>R} order using \texttt{write buffers}
   
   — write buffer with bypassing hides latency of writes
   
   — reads of different locations can bypass pending writes

Initially: Flag1=Flag2=0.

Can write buffers allow an ordering that violates sequential consistency?

Yes: 1, 2, 3, 4
Overlapping Writes (without Caches)

2. $W \rightarrow W$ order using overlapping writes
   
   - general interconnect vs. bus (memory parallelism)
   - writes to different memory locations issued by same processor handled by different memory modules
Overlapping Writes (without Caches)

2. $W \Rightarrow W$ order using **overlapping writes**
   
   —general interconnect vs. bus (memory parallelism)
   
   —multiple writes to different locations issued by same processor handled by different memory modules

Initially: Data=Head=0

Can overlapping writes violate sequential consistency?

```
P1
Write Head  Write Data
Memory      Head: 0

P2
Read Data  Read Head

P1
Data = 2000
Head = 1

P2
while (Head == 0) {
    ...
    = Data
```
Overlapping Writes (without Caches)

2. W=>W order using overlapping writes
   —general interconnect vs. bus (memory parallelism)
   —multiple writes to different locations issued by same processor handled by different memory modules

Initially: Data=Head=0

Can overlapping writes violate sequential consistency?

Yes: 1, 2, 3, 4

\[ t: \text{completion order} \]
Non-blocking Reads (without Caches)

3. R=>R|W order using **non-blocking reads**
   — non-blocking reads + general memory interconnect
   — non-blocking caches, dynamic scheduling, speculative execution
Non-blocking Reads (without Caches)

3. \( R \rightarrow R \| W \) order using **non-blocking reads**
   - non-blocking reads + general memory interconnect
   - non-blocking caches, dynamic scheduling, speculative execution

Initially: Data=Head=0

Can non-blocking reads violate sequential consistency?
Non-blocking Reads (without Caches)

3. R=>R|W order using **non-blocking reads**
   — non-blocking reads (+ same memory interconnect)
   — non-blocking caches, dynamic scheduling, speculative execution

Initially: Data=Head=0

Can non-blocking reads violate sequential consistency?

Yes: [spec] 1, 2, 3, 4
Even without caches, hardware optimizations can
   — violate program order
   — violate sequential consistency
Adding Caches

- Multiple caches can result in multiple copies of data values.
- Copies induce three requirements:
  - Coherence protocol: ensure any copies are up to date
    - Typical strategies:
      - Invalidate protocol: invalidate copies
      - Update protocol: update copies
    - Memory consistency bounds interval when values must propagate
  - Detecting when a write is complete
    - Harder with copies present
  - Propagating changes to copies is non-atomic
    - Requires acknowledgments
    - ... and you thought things were hard to reason about before!
• **W=>W order using write-through cache**  
  —general interconnect instead of bus (memory parallelism)  
  —write-through cache for each processor (cache not shared)
Caches

- W→W order using **write-through cache**
  - general interconnect instead of bus (memory parallelism)
  - write-through cache for each processor (cache not shared)

Initially: Data=Head=0
P2 cache has Data
Can write-through caches violate sequential consistency?

Initially: Data=Head=0
P2 cache has Data

Can write-through caches violate sequential consistency?

P1
Data = 2000
Head = 1

P2
while (Head == 0) {
   ...
   = Data
}
Caches

- **W=>W order using write-through cache**
  - general interconnect instead of bus (memory parallelism)
  - write-through cache for each processor (cache not shared)

Initially: Data=Head=0
P2 cache has Data

Can write-through caches violate sequential consistency?

Yes: 1, 2, 3, 4, 5

$t$: completion order
Caching Effects Summary

- Caches can
  - violate memory atomicity
  - violate sequential consistency
Compilers

- Reordering accesses to different locations can be problematic
- Register allocation of what should be a volatile is bad
- Assuming data is not shared can cause a variety of problems
- In the absence of analysis, must preserve order among memory operations
  - conflicts with code motion, register allocation, CSE, tiling, software pipelining ...

Compiler Effects Summary

- Compiler optimizations can
  - violate program order, and thus
  - violate sequential consistency
Summary

- Everybody violates sequential consistency!
  - hardware optimizations
  - caches
  - compilers
Relaxed Memory Models

Relaxed orderings allowed by relaxed memory models

- Relax Write to Read program order
- Relax Write to Write program order
- Relax Read to Read and Read to Write program orders
- Read others’ write early
- Read own write early

only for different locations
Relaxing W→R Order

Allows write buffers

—write buffer with bypassing hides latency of writes
—reads to different locations can bypass pending writes

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flag1 = 1</td>
<td>Flag2 = 1</td>
</tr>
<tr>
<td>if (Flag2 == 0)</td>
<td>if (Flag1 == 0)</td>
</tr>
</tbody>
</table>

critical section critical section

![Diagram of process synchronization with flags and shared bus]

Memory

Flag1: 0
Flag2: 0
Relaxing W→W Order

Allows overlapping writes

—general interconnect vs. bus (memory parallelism)
—multiple writes to different issued by same processor handled by different memory modules

P1
Data = 2000
Head = 1

P2
while (Head == 0) {;}
... = Data

P1
Write Head
Write Data
Head: 0
Memory
Data: 0

P2
Read Data
Read Head
General Interconnect
Read Data
Read Head
Relaxing $R\rightarrow R\mid W$ Order

Allows non-blocking reads

—non-blocking reads + general memory interconnect
—non-blocking caches, dynamic scheduling, speculative execution

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data = 2000</td>
<td>while (Head == 0) {}</td>
</tr>
<tr>
<td>Head = 1</td>
<td>... = Data</td>
</tr>
</tbody>
</table>

Diagram:
- P1 writes Head to 0 and Data to 2000.
- P2 reads Head and Data.
- General Interconnect connects the processes.
- P1 and P2 may execute concurrently with speculative execution.
Relaxing Write Atomicity

• Allow a processor to return value of its own write before all cached copies of data are invalidated or updated
  — allows read to return value before
    – write is serialized with other writes to same location
    – before invalidates or updates reach other processors
  — how?
    – forward value in write buffer to a later read
    – let read following write in write-through-cache return before write completes

• Allow a thread to return value of another thread’s write before all cached copies of data are invalidated or updated
Benefits of Relaxed Orderings

- Permits high performance hardware
- Permits compiler optimizations
  — reorder instructions between synchronization instructions
Darker Side of Relaxed Ordering

- Complicated safety nets
- “Explaining how [to precisely preserve the atomicity of a write] is difficult within the simple framework presented in this article” (p. 16)
Weak Ordering

- Assume two types of operations
  - synchronization
  - data (i.e., ‘everything else’)

- Observation: typically, reordering data accesses between synchronization operations does not affect correctness
Weak Ordering

• Assume two types of operations.
  —synchronization
  —data (i.e., ‘everything else’)

• Observation: typically, reordering data accesses between synchronization operations does not affect correctness

Approach
- allow reordering among normal data accesses
- require stricter ordering constraints for accesses to synchronization variables
Release Consistency

• Types of operations

—ordinary ~ data accesses in weak ordering
—special
  – syncs: two types
    acquire: e.g. lock operation to gain access to a CS
    release: write to grant permission to access CS
  – nsyncs: asynchronous operations that are not synchronization ops
“Safety Net” Mechanisms

• Serialization instructions (IBM 370)
  — e.g. compare-and-swap, branches
  — placing serialization instruction after write guarantees SC

• Atomic read-modify-write operations
  — e.g. SPARC TSO: program order appears to be preserved between W and following R if one of them is part of a RMW operation
    – can replace R with “identity” RMW to force ordering on R
    – can replace W with “oblivious” RMW to force ordering on W
  — preserving R → W ordering with PC: replace R with “identity” RMW

• Fence instructions
  — memory barrier: fence for all memory ops
  — store barrier: fence for writes only
  — SPARC MEMBAR: can enforce orderings between access types selectively
## Relaxed Ordering in Practice

<table>
<thead>
<tr>
<th>Relaxation</th>
<th>W → R Order</th>
<th>W → W Order</th>
<th>R → RW Order</th>
<th>Read Others’ Write Early</th>
<th>Read Own Write Early</th>
<th>Safety net</th>
</tr>
</thead>
<tbody>
<tr>
<td>SC [16]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>IBM 370 [14]</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>serialization instructions</td>
</tr>
<tr>
<td>TSO [20]</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td>√</td>
<td>RMW</td>
</tr>
<tr>
<td>PC [13, 12]</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td></td>
<td>√</td>
<td>RMW</td>
</tr>
<tr>
<td>PSO [20]</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
<td>√</td>
<td>RMW, STBAR</td>
</tr>
<tr>
<td>WO [5]</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td></td>
<td>√</td>
<td>synchronization</td>
</tr>
<tr>
<td>RCsc [13, 12]</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td></td>
<td>√</td>
<td>release, acquire, nsync, RMW</td>
</tr>
<tr>
<td>RCpc [13, 12]</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>release, acquire, nsync, RMW</td>
</tr>
<tr>
<td>Alpha [19]</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td></td>
<td>√</td>
<td>MB, WMB</td>
</tr>
<tr>
<td>RMO [21]</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td></td>
<td>√</td>
<td>various MEMBAR’s</td>
</tr>
<tr>
<td>PowerPC [17, 4]</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>SYNC</td>
</tr>
</tbody>
</table>
Coping with Relaxed Models

• What if programmers had to keep all these details in mind?
  – relaxed program order + relaxed memory atomicity...

• Abstraction: we want a memory model for a language
  — general enough
    – to permit performance
    – to be widely used
  — simple enough
    – to reason about
    – to be portable
Examples of Language Level Models

- **Java**
  - detailed memory model specification for security, portability

- **C++**
  - simple to understand defaults to simplify development
  - full control for top performance
  - no concern for security

- **Unified Parallel C**
  - supports both “strict” and “relaxed” memory models
  - simplicity vs. performance
  - default and per access choices
Take-away Points

• Memory models, which describe the semantics of shared variables, are crucial to both correct multithreaded applications and the entire underlying implementation stack.

• Major programming languages are converging on a model that guarantees simple interleaving-based semantics for “data-race-free” programs and most hardware vendors have committed to support this model.

• This process has exposed fundamental shortcomings in our languages and a hardware-software mismatch:
  — semantics for programs that contain data races seem fundamentally difficult, but are necessary for concurrency safety and debuggability.
  — call upon software and hardware communities to develop languages and systems that enforce data-race-freedom, and co-designed hardware that exploits and supports such semantics.
References


