ECE 259 / CPS 221
Advanced Computer Architecture II
(Parallel Computer Architecture)

Memory Consistency Models

Copyright 2010 Daniel J. Sorin
Duke University

Slides are derived from work by
Sarita Adve (Illinois), Babak Falsafi (CMU),
Mark Hill (Wisconsin), Alvy Lebeck (Duke), Steve
Reinhardt (Michigan), and J. P. Singh (Princeton).
Thanks!
Outline

• Difference Between Coherence and Consistency

• Sequential Consistency

• Relaxed Memory Consistency Models

• Consistency Optimizations

• Synchronization Optimizations
Coherence vs. Consistency

• Intuition says load should return latest value
  – What is latest?

• Coherence concerns only one memory location
• Consistency concerns apparent ordering for ALL locations

• A memory system is coherent if, for all locations,
  – Can serialize all operations to that location such that,
  – Operations performed by any processor appear in program order
    » Program order = order defined by program text or assembly code
  – A read gets the value written by last store to that location
Why Consistency is Important

• Consistency model defines correct behavior
  – It is a contract between the system and the programmer
  – Analogous to the ISA specification
  – Part of architecture → software-visible

• Coherence protocol is only a means to an end
  – Coherence is not visible to software (i.e., not architectural)
  – Enables new system to present same consistency model despite using newer, fancier coherence protocol
  – Systems maintain backward compatibility for consistency (like ISA)

• Consistency model restricts ordering of loads/stores
  – Does NOT care at all about ordering of coherence messages
Why Coherence != Consistency

/* initially, A = B = flag = 0 */
P1
A = 1; while (flag == 0); /* spin */
B = 1; print A;
flag = 1; print B;

• Intuition says we should print $A = B = 1$
• Yet, in some consistency models, this isn’t required!
• Coherence doesn’t say anything … why?
Sequential Consistency

• Leslie Lamport 1979:
  “A multiprocessor is **sequentially consistent** if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program”

Abstraction: a multitasking uniprocessor
The Memory Model

sequential processors issue memory ops in program order

P1  P2  Pn

switch randomly set after each memory op

Memory
SC: Definitions

• Sequentially consistent execution
  – Result is same as one of the possible interleavings on uniprocessor

• Sequentially consistent system
  – Any possible execution corresponds to some possible total order

• Alternate equivalent definition of SC
  – There exists a total order of all loads and stores (across all processors), such that the value returned by each load equals the value of the most recent store to that location
SC: More Definitions

- **Memory operation**
  - Load, store, atomic read-modify-write to mem location

- **Issue**
  - An operation is *issued* when it leaves processor and is presented to memory system (cache, write-buffer, local and remote memories)

- **Perform**
  - A store is *performed* wrt to a processor P when a load by P returns value produced by that store or a later store
  - A load is *performed* wrt to a processor when subsequent stores cannot affect value returned by that load

- **Complete**
  - A memory operation is *complete* when performed wrt all processors.

- **Program execution**
  - Memory operations for specific run only (ignore non-memory-referencing instructions)
Sufficient Conditions for Sequential Consistency

- Processors issue memory ops in program order
- Processor must wait for store to complete before issuing next memory operation
- After load, issuing proc waits for load to complete, and store that produced value to complete before issuing next op
- Easily implemented with shared (physical) bus

This is sufficient, but more than necessary
• **MIPS R10000 is dynamically scheduled**
  – Allows memory operations to issue and execute out of program order
  – But ensures that they become visible and complete in order
  – Doesn’t satisfy sufficient conditions, but provides SC

• **An interesting issue w.r.t. preserving SC**
  – On a write to a shared block, requestor gets two types of replies:
    » Exclusive reply from the home, indicates write is serialized at memory
    » Invalidation acks, indicate that write has completed wrt processors
  – But microprocessor expects only one reply (as in a uniprocessor)
    » So replies have to be dealt with by requestor’s HUB
  – To ensure SC, Hub must wait until inval acks are received before replying to proc
    » Can’t reply as soon as exclusive reply is received
      • Would allow later accesses from proc to complete (writes become visible) before this write
Outline

• Difference Between Coherence and Consistency
• Sequential Consistency
• Relaxed Memory Consistency Models
  – Motivation
  – Processor Consistency
  – Weak Ordering & Release Consistency
• Consistency Optimizations
• Synchronization Optimizations
**Why Relaxed Memory Models?**

- **Motivation with directory protocols**
  - Misses have longer latency
  - Collecting acknowledgments can take even longer

- **Recall SC requires strict ordering of reads/writes**
  - Each processor generates a local total order of its reads and writes (R→R, R→W, W→W, & R→W)
  - All local total orders are interleaved into a global total order

- **Relaxed models relax some of these constraints**
  - **PC**: Relax ordering from writes to reads (to different addresses)
  - **RC**: Relax all read/write orderings (but add “fences”)

(C) 2010 Daniel J. Sorin from Adve, Falsafi, Hill, Lebeck, Reinhardt, Singh

ECE 259 / CPS 221
Processor Consistency (PC): Relax Write to Read Order

/* initially, A = B = 0 */

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A = 1;</td>
<td>B = 1</td>
</tr>
<tr>
<td>r1 = B;</td>
<td>r2 = A;</td>
</tr>
</tbody>
</table>

• Processor Consistency (PC)
  - Allows r1==r2==0 (not allowed by SC)
  - Examples: Sun Total Store Order (TSO), Intel IA-32

• Why do this?
  » Allows FIFO write buffers → performance!
  » Does not confuse programmers (too much)
Write Buffers w/ Read Bypass

- P1
  - Read Flag 2 -> Write Flag 1
  - Write Flag 1 -> t3
  - t3

- P2
  - Read Flag 1
  - Write Flag 2 -> t4
  - t4

- Shared Bus
  - Flag 1: 0
  - Flag 2: 0

P1
- Flag 1 = 1
- if (Flag 2 == 0)
- critical section

P2
- Flag 2 = 1
- if (Flag 1 == 0)
- critical section

(C) 2010 Daniel J. Sorin from Adve, Falsafi, Hill, Lebeck, Reinhardt, Singh

ECE 259 / CPS 221
Also Want “Causality” (Transitivity)

/* initially all 0 */

P1
A = 1;
flag1 = 1;

P2
while (flag1==0) {};
flag2 = 1;

P3
while (flag2==0) {};
r3 = A;

• We expect P3’s r3=A to get value 1
• All commercial versions of PC guarantee causality
So Why Not Relax All Order?

/* initially all 0 */

P1
A = 1;
B = 1;
flag = 1;

P2
while (flag == 0); /* spin */

r1 = A;
r2 = B;

• We’d like to be able to reorder “A = 1”/“B = 1” and/or “r1 = A”/“r2 = B”
  – Useful because it could allow for OOO processors, non-FIFO write buffers, delayed directory acknowledgments, etc.

• But, for sanity, we still would like to order
  – “A = 1” / “B = 1” before “flag =1”
  – “flag != 0” before “r1 = A” / “r2 = B”
Order with “Synch” Operations

/* initially all 0 */
P1
A = 1; while (SYNCH flag == 0);
B = 1; r1 = A;
SYNCH flag = 1; r2 = B;

P2
while (SYNCH flag == 0);

• Called weak ordering (WO) or “weak consistency”
• SYNCH orders all prior and subsequent operations
• Alternatively, release consistency (RC) specializes
  – Acquire: forces subsequent reads/writes after
  – Release: forces previous reads/writes before
Review: Directory Example with Sequential Consistency
Commercial Models Use “Fences”

/* initially all 0 */

```
P1
A = 1;
B = 1;
FENCE;
flag = 1;

P2
while (flag == 0);
FENCE;
r1 = A;
r2 = B;
```

- **Examples:** Compaq Alpha, IBM PowerPC, & Sun RMO
  - Can specialize fences (e.g., RMO)

- **Intel IA-64 is RCpc** (acquires & releases obey PC)
The Programming Interface

• WO and RC require *synchronized programs*

• All synchronization operations must be labeled and visible to the hardware
  – Easy (easier!) if synchronization library used
  – Must provide language support for arbitrary Ld/St synchronization (event notification, e.g., flag)

• Program written for weaker model OK on stricter
  – E.g., SC is a valid implementation of TSO, WO, or RC