Advanced Operating Systems (CS 202)

Memory Consistency, Cache Coherence and Synchronization (Part II)

(some cache coherence slides adapted from Ian Watson; some memory consistency slides from Sarita Adve)
Concurrency and Memory Consistency

References:
- A primer on memory consistency and cache coherence, Sorin, Hill and wood, 2011 (chapters 3 and 4)
- Memory Models: A Case for Rethinking Parallel Languages and Hardware, Adve and Boehm, 2010
Memory Consistency

- Formal specification of memory semantics

- Guarantees as to how shared memory will behave on systems with multiple processors

- Ordering of reads and writes

- Essential for programmer (OS writer!) to understand
Why Bother?

- Memory consistency models affect *everything*
  - Programmability
  - Performance
  - Portability
- Model must be defined at all levels
- Programmers and system designers care
Uniprocessor Systems

- Memory operations occur:
  - One at a time
  - In program order
- Read returns value of last write
  - Only matters if location is the same or dependent
  - Many possible optimizations
- Intuitive!
How does a core reorder? (1)

- **Store-store reordering:**
  - Non-FIFO write buffer

- **Load-load or load-store/store-load reordering:**
  - Out of order execution

- Should the hardware prevent any of this behavior?
# Multiprocessor: Example

## Table 3.1: Should r2 Always be Set to NEW?

<table>
<thead>
<tr>
<th>Core C1</th>
<th>Core C2</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1: Store data = NEW;</td>
<td>L1: Load r1 = flag;</td>
<td>/* Initially, data = 0 &amp; flag ≠ SET */</td>
</tr>
<tr>
<td>S2: Store flag = SET;</td>
<td>B1: if (r1 ≠ SET) goto L1;</td>
<td>/* L1 &amp; B1 may repeat many times */</td>
</tr>
<tr>
<td></td>
<td>L2: Load r2 = data;</td>
<td></td>
</tr>
</tbody>
</table>
**Cont’d**

S2 and S1 reordered

Why? How?

**TABLE 3.2:** One Possible Execution of Program in Table 3.1.

<table>
<thead>
<tr>
<th>cycle</th>
<th>Core C1</th>
<th>Core C2</th>
<th>Coherence state of data</th>
<th>Coherence state of flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>S2: Store flag=SET</td>
<td>L1: Load r1=flag</td>
<td>read-only for C2</td>
<td>read-write for C1</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>L2: Load r2=data</td>
<td>read-only for C2</td>
<td>read-only for C2</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>read-only for C2</td>
<td>read-only for C2</td>
</tr>
<tr>
<td>4</td>
<td>S1: Store data=NEW</td>
<td></td>
<td>read-write for C1</td>
<td>read-only for C2</td>
</tr>
</tbody>
</table>
Example 2

<table>
<thead>
<tr>
<th>TABLE 3.3: Can Both r1 and r2 be Set to 0?</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Core C1</strong></td>
</tr>
<tr>
<td>S1: x = NEW;</td>
</tr>
<tr>
<td>L1: r1 = y;</td>
</tr>
</tbody>
</table>


Sequential Consistency

- The result of any execution is the same as if all operations were executed on a single processor.
- Operations on each processor occur in the sequence specified by the executing program.
One execution sequence

**TABLE 3.1:** Should r2 Always be Set to NEW?

<table>
<thead>
<tr>
<th>Core C1</th>
<th>Core C2</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1: Store data = NEW;</td>
<td>L1: Load r1 = flag;</td>
<td>/* Initially, data = 0 &amp; flag ≠ SET */</td>
</tr>
<tr>
<td>S2: Store flag = SET;</td>
<td>B1: if (r1 ≠ SET) goto L1;</td>
<td>/* L1 &amp; B1 may repeat many times */</td>
</tr>
<tr>
<td></td>
<td>L2: Load r2 = data;</td>
<td></td>
</tr>
</tbody>
</table>

**FIGURE 3.1:** A Sequentially Consistent Execution of Table 3.1’s Program.
program order (<p) of Core C1
S1: x = NEW; /* NEW */
L1: r1 = y; /* 0 */

memory order (<m) S2: y = NEW; /* NEW */
L2: r2 = x; /* NEW */

program order (<p) of Core C2
Outcome: (r1, r2) = (0, NEW)
(a) SC Execution 1

S1: x = NEW; /* NEW */
L1: r1 = y; /* NEW */

S2: y = NEW; /* NEW */
L2: r2 = x; /* 0 */

Outcome: (r1, r2) = (NEW, 0)
(b) SC Execution 2

S1: x = NEW; /* NEW */
L1: r1 = y; /* NEW */

S2: y = NEW; /* NEW */
L2: r2 = x; /* NEW */

Outcome: (r1, r2) = (NEW, NEW)
(c) SC Execution 3

S1: x = NEW; /* NEW */
L1: r1 = y; /* 0 */

S2: y = NEW; /* NEW */
L2: r2 = x; /* 0 */

Outcome: (r1, r2) = (0, 0)
(d) NOT an SC Execution
S.C. Disadvantages

- Difficult to implement!
- Huge lost potential for optimizations
  - Hardware (cache) and software (compiler)
  - Be conservative: err on the safe side
  - Major performance hit
Relaxed Consistency

- **Program Order** relaxations *(different locations)*
  - $W \rightarrow R$;
  - $W \rightarrow W$;
  - $R \rightarrow R/W$

- **Write Atomicity** relaxations
  - Read returns another processor’s Write early

- Combined relaxations
  - Read your own Write *(okay for S.C.)*

- **Safety Net** – available synchronization operations

- **Note:** assume one thread per core
Write → Read

- Can be reordered: same processor, different locations
  - Hides write latency
- Different processors? Same location?
  1. IBM 370
    - Any write must be fully propagated before reading
  2. SPARC V8 – Total Store Ordering (TSO)
    - Can read its own write before that write is fully propagated
    - Cannot read other processors’ writes before full propagation
  3. Processor Consistency (PC)
    - Any write can be read before being fully propagated
Write → Write

- Can be reordered: same processor, different locations
  - Multiple writes can be pipelined/overlapped
  - May reach other processors out of program order
- Partial Store Ordering (PSO)
  - Similar to TSO
    - Can read its own write early
    - Cannot read other processors’ writes early