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>
<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>
S2 and S1 reordered

Why? How?
Example 2

**TABLE 3.3**: Can Both r1 and r2 be Set to 0?

<table>
<thead>
<tr>
<th>Core C1</th>
<th>Core C2</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1: x = NEW; L1: r1 = y;</td>
<td>S2: y = NEW; L2: r2 = x;</td>
<td>/* Initially, x = 0 &amp; y = 0*/</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.
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></td>
<td>/* Initially, data = 0 &amp; flag ≠ SET */</td>
</tr>
<tr>
<td>S2: Store flag = SET;</td>
<td>L1: Load r1 = flag;</td>
<td>/* L1 &amp; B1 may repeat many times */</td>
</tr>
<tr>
<td></td>
<td>B1: if (r1 ≠ SET) goto L1;</td>
<td></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.
(a) SC Execution 1

(b) SC Execution 2

(c) SC Execution 3

(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 → R; W → W; R → 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