Multiprocessor Synchronization

- Multiprocessor Systems
- Memory Consistency
- In addition, read Doeppner, 5.1 and 5.2

(Much material in this section has been freely borrowed from Gernot Heiser at UNSW and from Kevin Elphinstone)

MP Memory Architectures

- Uniform Memory-Access (UMA)
  - Access to all memory locations incurs same latency.

- Non-Uniform Memory-Access (NUMA)
  - Memory access latency differs across memory locations for each processor.
    - e.g. Connection Machine, AMD HyperTransport, Intel Itanium MPs
UMA Multiprocessors: Types

- "Classical" Multiprocessor
  - CPUs with local caches
  - typically connected by bus
  - fully separated cache hierarchy -> cache coherency problems
- Chip Multiprocessor (multicore)
  - per-core L1 caches
  - shared lower on-chip caches
  - cache coherency addressed in HW
- Simultaneous Multithreading
  - interleaved execution of several threads
  - fully shared cache hierarchy
  - no cache coherency problems

Cache Coherency

- What happens if one CPU writes to (cached) address and another CPU reads from the same address?

- Ideally, a read produces the result of the last write to the same memory location. ("Strict Memory Consistency")

- Typically, a hardware solution is used
  - snooping - for bus-based architectures
  - directory-based - for non bus-interconnects
**Snooping**

- Each cache **broadcasts** transactions on the bus.

- Each cache monitors the bus for transactions that affect its state.

- **Conflicts** are typically resolved using some cache coherency protocol.

- Snooping can be easily extended to multi-level caches.

---

**Memory Consistency: Example**

- **Example: End of Critical Section**

```plaintext
/* lock(mutex) */
<whatever it takes...>
/* counter++ */
load r1, counter
add r1, r1, 1
store r1, counter
/* unlock(mutex) */
store zero, mutex
```

- Relies on all CPUs seeing update of **counter** before update of **mutex**

- Depends on assumptions about **ordering of stores** to memory
Example of Strong Ordering: Sequential Ordering

- Observation: **Strict Consistency** is **impossible** to implement.

- **Sequential Consistency:**
  - Loads and stores execute **in program order**
  - Memory accesses of different CPUs are "**sequentialised**"; i.e., any valid interleaving is acceptable, but all processes must see the same sequence of memory references.

- Traditionally used by many simple architectures

<table>
<thead>
<tr>
<th>CPU 0</th>
<th>CPU 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>store r1, adr1</td>
<td>store r1, adr2</td>
</tr>
<tr>
<td>load r2, adr2</td>
<td>load r2, adr1</td>
</tr>
</tbody>
</table>

- In this example, at least one CPU must load the other’s new value.

Sequential Consistency (cont)

- **Sequential consistency** is **programmer-friendly**, but **expensive**.

  - Side note: Lipton & Sandbert (1988) show that improving the read performance makes write performance worse, and vice versa.

- Modern HW features **interfere** with sequential consistency; e.g.:
  - **write buffers** to memory (aka store buffer, write-behind buffer, store pipeline)
  - **instruction reordering** by optimizing compilers
  - **superscalar** execution
  - **pipelining**
Weaker Consistency Models: Total Store Order

- **Total Store Ordering (TSO)** guarantees that the sequence in which store, FLUSH, and atomic load-store instructions appear in memory for a given processor is identical to the sequence in which they were issued by the processor.

- Both x86 and SPARC processors support TSO.

- A later load can bypass an earlier store operation. (!)

  - i.e., local load operations are permitted to obtain values from the write buffer before they have been committed to memory.

Total Store Order (cont)

- Example:
  
<table>
<thead>
<tr>
<th>CPU 0</th>
<th>CPU 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>store r1, adr1</td>
<td>store r1, adr2</td>
</tr>
<tr>
<td>load r2, adr2</td>
<td>load r2, adr1</td>
</tr>
</tbody>
</table>

  - Both CPUs may read old value!

- Need hardware support to force global ordering of privileged instructions, such as:
  - atomic swap
  - test & set
  - load-linked + store-conditional
  - memory barriers

  - For such instructions, stall pipeline and flush write buffer.
It gets weirder: Partial Store Ordering

- Partial Store Ordering (PSO) does not guarantee that the sequence in which store, FLUSH, and atomic load-store instructions appear in memory for a given processor is identical to the sequence in which they were issued by the processor.
- The processor can reorder the stores so that the sequence of stores to memory is not the same as the sequence of stores issued by the CPU.
- SPARC processors support PSO; x86 processors do not.
- Ordering of stores is enforced by memory barrier (instruction STBAR for Sparc): If two stores are separated by memory barrier in the issuing order of a processor, or if the instructions reference the same location, the memory order of the two instructions is the same as the issuing order.

Partial Store Order (cont)

- Example:
  ```
  load r1, counter
  add r1, r1, 1
  store r1, counter
  barrier
  store zero, mutex
  ```
  
  - Store to mutex can “overtake” store to counter.
  - Need to use memory barrier to separate issuing order.
  - Otherwise, we have a race condition.