Outline

• Announcements
  – Assignment #5
    • Memory hierarchy analysis using a micro-benchmark
    • Run on 2 or 3 different machines that you have access to, discuss behavior
  – Final exam on December 11\textsuperscript{th}: Room 402, WWH
    • Open book/notes or closed?

• This lecture
  – Multithreading and Multiprocessors
    • Cache coherence in small-scale SMPs
    • Cache coherence in large-scale parallel machines

[ Hennessy/Patterson CA:AQA (3rd Edition): Chapter 6]
[ Culler/Singh/Gupta,
  Parallel Computer Architecture: A Hardware/Software Approach: Chapters 3-4]
(Review) The Cache Coherence Problem

- Processors see **stale** values
  - with write-back caches, value written back to memory depends on which cache flushes or writes back value (and when)
  - clearly not a desirable situation!

(Review) So What Should Happen?

- Intuition for a coherent memory system
  
  *reading a location should return latest value written (by any process)*

- What does **latest** mean?
  - several alternatives (even on uniprocessors)
    - source program order, program issue order, order of completion, etc.
  - how to make sense of order among multiple processes?
    - must define a meaningful semantics

- Is cache coherence a problem on uniprocessors?
  - Yes!
    - interaction between caches and I/O devices
      - infrequent software solutions work well
        - uncachable memory, flush pages, route I/O through caches
      - however, the problem is performance-critical in multiprocessors
        - needs to be treated as a basic hardware design issue
Some Basic Definitions

- **Uniprocessors**:
  - memory operation: a single read, write or read-modify-write access
  - assumed to execute atomically with respect to each other
  - issue: a memory operation issues when it leaves the processor’s internal environment and is presented to the memory system (cache, buffer, etc.)
  - perform: operation appears to have taken place, as far as the processor can tell from other memory operations it issues
    - a write performs w.r.t. the processor when a subsequent read by the processor returns the value of that write or a later write
    - a read performs w.r.t the processor when subsequent writes issued by the processor cannot affect the value returned by the read

- **Multiprocessors**
  - all the above stay the same, but replace “the” by “a” processor
  - complete: perform with respect to all processors
  - still need to make sense of order in operations from different processes!

Order Among Multiple Processes: Intuition

- Assume a single shared memory, no caches
  - every read/write to a location accesses the same physical location
    - operation completes when it does so
  - so, memory imposes a serial or total order on operations to the location
    - operations to the location from a given processor are in program order
    - the order of operations to the location from different processors is some interleaving that preserves the individual program orders

- With caches
  - “latest” = *most recent in a serial order that maintains these properties*
    - for the serial order to be consistent, all processors must see writes to the location in the same order (if they bother to look)

- Note that we do not need to construct the total order
  - the program should just behave as if some serial order is enforced
Formal Definition of Coherence

A memory system is coherent if the results of any execution of a program are such that for each location, it is possible to construct a hypothetical serial order of all operations to the location that is consistent with the results of the execution and in which:

- operations issued by any particular process occur in the order issued by that process, and
- the value returned by a read is the value written by the last write to that location in the serial order

• Two necessary features:
  - write propagation: value written must become visible to others
  - write serialization: writes to a location seen in the same order by all

Cache Coherence Using a Bus

Two fundamentals of uniprocessor systems

• Bus transactions
  - three phases: arbitration, command/address, data transfer
  - all devices observe addresses, one is responsible for providing data

• Cache state transitions
  - every block is a finite state machine
  - two states in write-through, write no-allocate caches: valid, invalid
  - write-back caches have one more state: modified ("dirty")

• Multiprocessors extend both these somewhat to implement coherence
  - “snoop” on bus events and take action
  - cache controller receives inputs from two sides: processor and bus
    - actions: update state, respond with data, generate new bus transactions
  - protocol implemented by cooperating state machines
Coherence with Write-through Caches

- Snoop on write transactions and invalidate/update cache
  - memory is always up-to-date (write-through)
  - invalidation causes next read to miss and fetch new value from memory (write propagation)
  - bus transactions impose serial order \( \Rightarrow \) writes are seen in the same order (write serialization)

Write-through State Transition Diagram

- 2 states per block (valid and invalid)
  - state of each memory block is a \( p \)-vector
- Hardware state bits associated with only blocks that are in the cache
  - other blocks can be seen as being in invalid (not-present) state in that cache
- Writes do not change block state locally
  - invalidate other caches
- Protocol allows multiple readers to be simultaneously active
  - until invalidated by writes
Problem with Write-Through

- High bandwidth requirements
  - every write from every processor goes to shared bus and memory
  - consider: 200MHz, 1 CPI processor, and 15% instrs. are 8-byte stores
    - each processor generates 30M stores or 240MB data per second
    - 1GB/s bus can support only about 4 processors without saturating

- Write-back caches absorb most writes as cache hits
  - but need sophisticated protocols to ensure write propagation and serialization

- But, first let us understand other ordering issues ...

How to Order Reads/Writes by Different Processors?

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>/* Assume initial value of A and flag is 0 */</td>
<td></td>
</tr>
<tr>
<td>A = 1;</td>
<td>while (flag == 0); /* spin idly */</td>
</tr>
<tr>
<td>flag = 1;</td>
<td>print A;</td>
</tr>
</tbody>
</table>

Event synchronization

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>/* Assume initial values of A and B are 0*/</td>
<td></td>
</tr>
<tr>
<td>(1a) A = 1;</td>
<td>(2a) print B;</td>
</tr>
<tr>
<td>(1b) B = 2;</td>
<td>(2b) print A;</td>
</tr>
</tbody>
</table>

More generally

Memory consistency model:
- specifies constraints on the order in which memory operations (from any process) can appear to execute with respect to one another
  - what orders are preserved?
  - given a load, constrains the possible values returned by it
- contract between programmer and system
Sequential Consistency (Lamport’79)

[A multiprocessor system is \textit{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.

- Two aspects
  - \textbf{program order}: completion of previous memory operations
    - write completion is more crucial
  - \textbf{write atomicity}: serialization of writes to the same location

- Sufficient conditions
  - every process issues memory operations in program order
  - after a write operation is issued, the issuing process waits for the write to complete before issuing its next operation
  - after a read operation is issued, the issuing process waits for the read to complete, and for the write whose value is being returned by the read to complete, before issuing its next operation (provides write atomicity)

- Above conditions are not necessary
  - hardware needs only to appear to preserve sequential consistency
    - okay to do 1b -> 1a -> 2b -> 2a (indistinguishable from 1a -> 1b -> 2a -> 2b)

Sequential Consistency (contd.)
Memory Consistency and Cache Coherence

- Cache coherence is mechanism for implementing memory consistency
  - detect write completion (read completion is easy)
  - ensure write atomicity

- Centralized bus interconnect makes it easier
  - trivially true for write-through caches (earlier protocol)
    - write and read misses to all locations serialized by bus into bus order
      - if read obtains value of write \( W \), \( W \) is guaranteed to have completed since it caused a bus transaction
      - when write \( W \) is performed w.r.t. any processor, all previous writes in bus order have completed
  - let us see some protocols for write-back caches
    - focus on invalidation-based protocols
    - Also possible to have update-based protocols

Write-back Caches: Invalidation-based Protocols

- States
  - \textit{Shared} (Valid), \textit{Invalid}, \textit{Modified} (Dirty)
  - \textit{Exclusive} (for MESI protocol)
    - processor can modify without notifying anyone else (i.e. no bus transaction)
    - must first get block in exclusive state before writing into it
    - even if already in valid state, need transaction, so called a write miss

- Bus transactions
  - uniprocessors
    - \textit{BusRd}: service a read miss
    - \textit{Flush}: to flush a cache block back to memory
  - multiprocessors
    - \textit{BusRdX}: tell others about impending write
      - makes the write visible, i.e., write is performed
      - only need this on first store to non-dirty data
    - coherence actions driven by BusRd and BusRdX transactions
3-State (MSI) Protocol

Snoop on *BusRd* and *BusRdX* transactions
- **BusRd**
  - if cache block is in *Modified* state, downgrade state to *Shared*, and flush data to memory
- **BusRdX**
  - if cache block is in *Shared* state, downgrade state to *Invalid*
  - if cache block is in *Modified* state, downgrade state to *Invalid*, and flush data to memory
- Lower-level choices
  - can also go to *Invalid* from *Modified* when *BusRd* is detected
    - decision depends on sharing pattern

Correctness of 3-State (MSI) Protocol

- Coherence conditions
  - write propagation because of *BusRdX* transactions
  - write serialization
    - all writes that appear on the bus (*BusRdX*) ordered by the bus
    - reads that appear on the bus ordered with respect to these
    - writes that don’t appear on the bus appear between two bus transactions
      - only issuing processor sees intermediate writes
      - other processors see writes serialized by the last bus transaction

- Sequential consistency conditions
  - write completion
    - can detect when write (the one that matters) appears on the bus
  - write atomicity
    - if a read returns the value of a write, that write has already become visible to all others already (can reason different cases)
4-state (MESI/Illinois) Protocol

- Problem with MSI protocol
  - reading and modifying data is 2 bus transactions, even with no sharing
    - I→S followed by S→M

- Exclusive state
  - free to modify without transaction
  - main-memory is still kept up-to-date
  - I→E if no one else has a shared copy
    - needs “shared” line

- Who returns data when not in M state?
  - originally: cache-to-cache sharing (Illinois protocol)
  - these days: memory

- Extension: MOESI protocol
  - owned state: exclusive and memory is not up-to-date

4-state Protocol: Example (0)

A1 and A2 map to the same cache block
All cache blocks are initially in I (invalid) state

<table>
<thead>
<tr>
<th>Step</th>
<th>P1 state</th>
<th>P1 addr</th>
<th>P1 value</th>
<th>P2 state</th>
<th>P2 addr</th>
<th>P2 value</th>
<th>Bus action</th>
<th>Bus proc</th>
<th>Bus addr</th>
<th>Bus value</th>
<th>Memory addr</th>
<th>Memory value</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1: write 10, A1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P1: read A1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: read A1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: write 20, A1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: write 40, A2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
4-state Protocol: Example (1)

A1 and A2 map to the same cache block
All cache blocks are initially in I (invalid) state

<table>
<thead>
<tr>
<th>Step</th>
<th>P1</th>
<th>P2</th>
<th>Bus</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>state</td>
<td>addr</td>
<td>value</td>
<td>state</td>
</tr>
<tr>
<td>P1: write 10, A1</td>
<td>M</td>
<td>A1</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>P1: read A1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: read A1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: write 20, A1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: write 40, A2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4-state Protocol: Example (2)

A1 and A2 map to the same cache block
All cache blocks are initially in I (invalid) state

<table>
<thead>
<tr>
<th>Step</th>
<th>P1</th>
<th>P2</th>
<th>Bus</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>state</td>
<td>addr</td>
<td>value</td>
<td>state</td>
</tr>
<tr>
<td>P1: write 10, A1</td>
<td>M</td>
<td>A1</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>P1: read A1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: read A1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: write 20, A1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: write 40, A2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
4-state Protocol: Example (3)

A1 and A2 map to the same cache block
All cache blocks are initially in I (invalid) state

<table>
<thead>
<tr>
<th>Step</th>
<th>P1</th>
<th>P2</th>
<th>Bus</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>state</td>
<td>addr</td>
<td>value</td>
<td>state</td>
</tr>
<tr>
<td>P1: write 10, A1</td>
<td>M</td>
<td>A1</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>P1: read A1</td>
<td>M</td>
<td>A1</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>P2: read A1</td>
<td>S</td>
<td>A1</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: write 20, A1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: write 40, A2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4-state Protocol: Example (4)

A1 and A2 map to the same cache block
All cache blocks are initially in I (invalid) state

<table>
<thead>
<tr>
<th>Step</th>
<th>P1</th>
<th>P2</th>
<th>Bus</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>state</td>
<td>addr</td>
<td>value</td>
<td>state</td>
</tr>
<tr>
<td>P1: write 10, A1</td>
<td>M</td>
<td>A1</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>P1: read A1</td>
<td>M</td>
<td>A1</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>P2: read A1</td>
<td>S</td>
<td>A1</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: write 20, A1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: write 40, A2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
4-state Protocol: Example (5)

A1 and A2 map to the same cache block
All cache blocks are initially in I (invalid) state

<table>
<thead>
<tr>
<th>Step</th>
<th>P1</th>
<th>P2</th>
<th>Bus</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>state</td>
<td>addr</td>
<td>value</td>
<td>state</td>
</tr>
<tr>
<td>P1: write 10, A1</td>
<td>M</td>
<td>A1</td>
<td>10</td>
<td>RdX</td>
</tr>
<tr>
<td>P1: read A1</td>
<td>M</td>
<td>A1</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>P2: read A1</td>
<td>S</td>
<td>A1</td>
<td></td>
<td>Rd</td>
</tr>
<tr>
<td></td>
<td>S</td>
<td>A1</td>
<td>10</td>
<td>Flush</td>
</tr>
<tr>
<td>P2: write 20, A1</td>
<td>I</td>
<td>M</td>
<td>A1</td>
<td>20</td>
</tr>
<tr>
<td>P2: write 40, A2</td>
<td></td>
<td>M</td>
<td>A2</td>
<td>40</td>
</tr>
</tbody>
</table>

Implementing Snooping Caches

- Multiple processors must be on bus, access to both addresses and data
- Add a few new commands to perform coherency, in addition to read and write
- Processors continuously snoop on address bus
  - If address matches tag, either invalidate or update
- Since every bus transaction checks cache tags, could interfere with CPU just to check:
  - *solution 1*: duplicate set of tags for L1 caches just to allow checks in parallel with CPU
  - *solution 2*: L2 cache already duplicate (provided L2 obeys inclusion with L1 cache)
    - block size, associativity of L2 affects L1
Implementation Complications

- Write Races:
  - Cannot update cache until bus is obtained
    - Otherwise, another processor may get bus first, and then write the same cache block!
  - Two step process:
    - Arbitrate for bus
    - Place miss on bus and complete operation
  - If miss occurs to block while waiting for bus, handle miss (invalidate may be needed) and then restart
  - Split transaction bus:
    - Bus transaction is not atomic:
      - can have multiple outstanding transactions for a block
    - Multiple misses can interleave, allowing two caches to grab block in the Exclusive state
    - Must track and prevent multiple misses for one block
  - Must support interventions and invalidations

Bus-based Cache Coherence: Performance Factors

- Impact of protocol optimizations
  - 3-state (MSI) versus 4-state (MESI) does not seem to matter much
    - workload-based evaluation (see text)

- Impact of block size
  - affects compulsory and coherence misses
    - Other kinds: capacity, conflict
  - increasing block size has advantages and disadvantages
    - can reduce misses if spatial locality is good
    - can increase misses due to false sharing
    - can increase traffic due to fetching unnecessary data and false sharing
    - can increase miss penalty and hit cost
  - in practice
    - impact of block size on miss rate varies with application
      - how well an application exploits spatial locality
      - bus traffic almost always increases
Case Study: Sun Enterprise

• SUN Enterprise 3000-6000, with Gigaplane interconnect (circa 1998)

30 UltraSparcs (9 GFLOPS)
2.7 GB/s bus (16 slots)
64-byte cache line
split-transaction with
112 outstanding transactions
transactions take 11-18 cycles

• MOESI protocol
  – owned state for cache-to-cache sharing
• 300ns read miss latency
  – 11 cycle min bus protocol at 83.5 Mhz is 130ns of this time
  – rest is path through caches and the DRAM access

Case Study: Sun Enterprise (cont’d)

• Sun Enterprise 10000 (Starfire): circa 2001

High-level architecture remains the same
• Upto 16 system boards, each with
  – Up to 4 400-500 MHz UltraSPARC II processors (total: 64)
  – 4 GB memory (total: 64 GB)
• Gigaplane XB interconnect
  – “bus” for addresses and control
    • 4 global interleaved address buses
      – 2 cycle address transfer rate (highly pipelined)
      – 167 million snoops per second (at 83.3 MHz clock)
  – 16x16 “crossbar” for data
    • Sustained data bandwidth of 12.8 GB/s (versus 2.7 GB/s)
    – Transactions take 26-38 cycles (versus 11-18): 500 ns
      • Latency has increased slightly, traded off versus bandwidth improvements
Extending Bus-based Coherence Schemes

- Motivation
  - leverage SMP building blocks, packaging, optimized technology
  - preserve tight cluster interaction
- Issues
  - can these systems be put together (without changing anything inside)
  - what kind of overheads do these systems incur?
- Several examples
  - Encore Gigamax, Convex Exemplar, Sequent NUMA-Q

Hierarchy of Busses

- Snoopy cache protocol
  - on B1 with L2 acting as memory, on B2 with L2 acting as processor
- Scaling considerations
  - number of processors: limited by packaging
  - memory latency and bandwidth: single traversal to root
  - coherency protocol bandwidth: many B1, one B2
  - locality/placement: sharing locality avoids B2 broadcasts
Cluster-based Hierarchies (Encore Gigamax)

- Main memory distributed among clusters
  - L2 cache can be replaced by a tag-only router-coherence switch

- Scaling considerations
  - number of processors: limited by packaging
  - memory latency and bandwidth: multiple, fast local access
  - coherency protocol bandwidth: many B1, one B2
  - locality/placement: important

Cache Coherence in Encore Gigamax

- Router-coherence switch must know about
  - local memory words in remote caches and their state (clean/dirty)
  - remote memory words in local caches and their state

- Operation
  - write to B1 is passed to B2 if
    - reference to a remote memory word
    - reference to a local memory word, but present in some remote cache
  - read to B1 is passed to B2 if
    - reference to a remote memory word (and not in cluster cache)
    - reference to a local memory word, but dirty in some remote cache
  - write to B2 is passed to B1 if
    - reference to a local memory word
    - data belongs to remote memory, but the block is dirty in a local cache
  - ...
  - many race conditions possible: write-back going out as request coming in
Ring-based Cache Coherence (KSR)

- Buses can be replaced by rings
  - any media capable of a broadcast
- Scaling considerations
  - number of processors: many
  - memory latency and bandwidth: linear in P
  - coherency protocol bandwidth:
    - broadcast medium, but no global serialization
    - many concurrent transactions, out of phase (many, many states)
  - locality/placement: within B1, all same on R2

Lessons from Hierarchical Coherence Schemes

- Why does bus-based coherence work?
  - FSM sequences with effectively atomic transitions ensure consensus on status of memory block and therefore coherence
- Why do extensions to bus-based schemes work?
  - layers extend these FSM transitions, delaying additional accesses until a global decision can be enforced
  - scaling limitations, but the overall scheme works because of global agreement
- General formulation: “directory-based” structure
  - associate an explicit state with each memory block
  - query and update this state using atomic transitions
    - memory consistency is ensured by restricting what this state can be