Introduction

- Thread-Level parallelism
  - Have multiple program counters
  - Uses MIMD model
  - Targeted for tightly-coupled, shared-memory multiprocessors

- For $n$ processors, need $n$ threads

- Amount of computation assigned to each thread = grain size
  - Threads can be used for data-level parallelism, but the overhead may outweigh the benefit.
### Types

- **Symmetric multiprocessors (SMP)**
  - Small number of cores
  - Share single memory with uniform memory latency
- **Distributed shared memory (DSM)**
  - Memory distributed among processors
  - Non-uniform memory access/latency (NUMA)
  - Processors connected via direct (switched) and non-direct (multi-hop) interconnection networks

### Centralized Memory Multiprocessor

- Also called symmetric multiprocessors (SMPs) because single main memory has a symmetric relationship to all processors
- Large caches ⇒ single memory can satisfy memory demands of small number of processors
- Can scale to a few dozen processors by using a switch and by using many memory banks
- Although scaling beyond that is technically conceivable, it becomes less attractive as the number of processors sharing centralized memory increases
Distributed Memory Multiprocessor

- Pro: Cost-effective way to scale memory bandwidth
  - If most accesses are to local memory
- Pro: Reduces latency of local memory accesses
- Con: Communicating data between processors more complex
- Con: Must change software to take advantage of increased memory BW

Two Models for Communication and Memory Architecture

1. Communication occurs by explicitly passing messages among the processors: 
   message-passing multiprocessors
2. Communication occurs through a shared address space (via loads and stores):
   shared memory multiprocessors either
   - UMA (Uniform Memory Access time) for shared address, centralized memory MP
   - NUMA (Non Uniform Memory Access time multiprocessor) for shared address, distributed memory MP

- In past, confusion whether “sharing” means sharing physical memory (Symmetric MP) or sharing address space
Challenges of Parallel Processing

1. Application parallelism ⇒ primarily via new algorithms that have better parallel performance
2. Long remote latency impact ⇒ both by architect and by the programmer
   - For example, reduce frequency of remote accesses either by
     - Caching shared data (HW)
     - Restructuring the data layout to make more accesses local (SW)
   - Today's lecture on HW to help latency via caches

Cache Coherence

Processors may see different values through their caches:

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
<th>Processor A reads X</th>
<th>Processor B reads X</th>
<th>Processor A stores 0 into X</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Processor A reads X</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Processor B reads X</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>Processor A stores 0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Example Cache Coherence Problem

- Processors see different values for \( u \) after event 3
- With write-back caches, value written back to memory depends on happenstance of which cache flushes or writes back value
  - Processes accessing main memory may see very stale value
  - Unacceptable for programming, and it's frequent!

### Cache Coherence

- **Coherence**
  - All reads by any processor must return the most recently written value.
  - Writes to the same location by any two processors are seen in the same order by all processors.

- **Consistency**
  - When a written value will be returned by a read
  - If a processor writes location A followed by location B, any processor that sees the new value of B must also see the new value of A.

- **Coherence:** *same* memory location
- **Consistency:** *different* memory locations
A memory system is coherent if

- A read by $P$ of $X$ that follows a write by $P$ of $X$, with no other write of $X$ by another processor occurring between the write and the read by $P$, always returns the value written by $P$.
- A read by $P$ of $X$ that follows a write to $X$ by another processor returns the written value, if the read and write are sufficiently separated in time and no other writes to $X$ occur between the two accesses.
- Writes to the same location are serialized; i.e. two writes to the same location by any two processors are seen in the same order by all processors.

Try it

Come up with examples:

a. coherent, but not consistent
b. consistent, but not coherent
c. neither
d. both
**Rules**

- A write does not complete (and allow the next write to occur) until all processors have seen the effect of that write.
- The processor does not change the order of any write with respect to any other memory access.

Effectively:

a processor is allowed to reorder reads, but is forced to finish writes in program order.

---

**Enforcing Coherence**

- Coherent caches provide:
  - *Migration*: movement of data
  - *Replication*: multiple copies of data

- Cache coherence protocols
  - Directory based
    - Sharing status of each block kept in one location
  - Snooping
    - Each core tracks sharing status of each block
Basic Schemes for Enforcing Coherence

- Program on multiple processors will normally have copies of the same data in several caches
  - Unlike I/O, where its rare
- Rather than trying to avoid sharing in SW, SMPs use a HW protocol to maintain coherent caches
  - Migration and Replication key to performance of shared data
- Migration - data can be moved to a local cache and used there in a transparent fashion
  - Reduces both latency to access shared data that is allocated remotely and bandwidth demand on the shared memory
- Replication – for shared data being simultaneously read, since caches make a copy of data in local cache
  - Reduces both latency of access and contention for read shared data

Snoopy Coherence Protocols

- Write invalidate
  - On write, invalidate all other copies
  - Use bus to serialize
    - Write cannot complete until bus access is obtained

<table>
<thead>
<tr>
<th>Processor activity</th>
<th>Bus activity</th>
<th>Contents of processor A's cache</th>
<th>Contents of processor B's cache</th>
<th>Contents of memory location X</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor A reads X</td>
<td>Cache miss for X</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Processor B reads X</td>
<td>Cache miss for X</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Processor A writes 1</td>
<td>Invalidation for X to X</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Processor B reads X</td>
<td>Cache miss for X</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- Write update
  - On write, update all copies
## Snoopy Coherence Protocols

- Locating an item when a read miss occurs
  - In write-back cache, the updated value must be sent to the requesting processor

- Cache lines marked as shared or exclusive/modified
  - Only writes to shared lines need an invalidate broadcast
    - After this, the line is marked as exclusive

### Snoopy Coherence Protocols Table

<table>
<thead>
<tr>
<th>Request</th>
<th>Source</th>
<th>State of addressed cache block</th>
<th>Type of cache action</th>
<th>Function and explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read hit</td>
<td>Processor</td>
<td>Shared or modified</td>
<td>Normal hit</td>
<td>Read data in local cache.</td>
</tr>
<tr>
<td>Read miss</td>
<td>Processor</td>
<td>Invalid</td>
<td>Normal miss</td>
<td>Place read miss on bus.</td>
</tr>
<tr>
<td>Read miss</td>
<td>Processor</td>
<td>Shared</td>
<td>Replacement</td>
<td>Address conflict miss; place read miss on bus.</td>
</tr>
<tr>
<td>Read miss</td>
<td>Processor</td>
<td>Modified</td>
<td>Replacement</td>
<td>Address conflict miss: write-back block, then place read miss on bus.</td>
</tr>
<tr>
<td>Write hit</td>
<td>Processor</td>
<td>Modified</td>
<td>Normal hit</td>
<td>Write data in local cache.</td>
</tr>
<tr>
<td>Write miss</td>
<td>Processor</td>
<td>Shared</td>
<td>Coherence</td>
<td>Place invalidate on bus. These operations are often called upgrade or ownership misses, since they do not fetch the data but only change the state.</td>
</tr>
<tr>
<td>Write miss</td>
<td>Processor</td>
<td>Invalid</td>
<td>Normal miss</td>
<td>Place write miss on bus.</td>
</tr>
<tr>
<td>Write miss</td>
<td>Processor</td>
<td>Shared</td>
<td>Replacement</td>
<td>Address conflict miss: place write miss on bus.</td>
</tr>
<tr>
<td>Write miss</td>
<td>Processor</td>
<td>Modified</td>
<td>Replacement</td>
<td>Address conflict miss: write-back block, then place write miss on bus.</td>
</tr>
<tr>
<td>Read miss</td>
<td>Bus</td>
<td>Shared</td>
<td>No action</td>
<td>Allow shared cache or memory to service read miss.</td>
</tr>
<tr>
<td>Read miss</td>
<td>Bus</td>
<td>Modified</td>
<td>Coherence</td>
<td>Attempt to share data: place cache block on bus and change state to shared.</td>
</tr>
<tr>
<td>Invalidate</td>
<td>Bus</td>
<td>Shared</td>
<td>Coherence</td>
<td>Attempt to write shared block; invalidate the block.</td>
</tr>
<tr>
<td>Write miss</td>
<td>Bus</td>
<td>Shared</td>
<td>Coherence</td>
<td>Attempt to write shared block; invalidate the cache block.</td>
</tr>
<tr>
<td>Write miss</td>
<td>Bus</td>
<td>Modified</td>
<td>Coherence</td>
<td>Attempt to write block that is exclusive elsewhere; write-back the cache block and make its state invalid to the local cache.</td>
</tr>
</tbody>
</table>
Two Classes of Cache Coherence Protocols

1. **Directory based** — Sharing status of a block of physical memory is kept in just one location, the directory

2. **Snooping** — Every cache with a copy of data also has a copy of sharing status of block, but no centralized state is kept
   - All caches are accessible via some broadcast medium (a bus or switch)
   - All cache controllers monitor or snoop on the medium to determine whether or not they have a copy of a block that is requested on a bus or switch access
Snoopy Cache-Coherence Protocols

- Cache Controller “snoops” all transactions on the shared medium (bus or switch)
  - relevant transaction if for a block it contains
  - take action to ensure coherence
    - invalidate, update, or supply value
    - depends on state of the block and the protocol
- Either get exclusive access before write via write invalidate or update all copies on write

Example: Write-thru Invalidate

- Must invalidate before step 3
- Write update uses more broadcast medium BW
  ⇒ all recent MPUs use write invalidate
Architectural Building Blocks

- Cache block state transition diagram
  - FSM specifying how disposition of block changes
    - invalid, valid, dirty
- Broadcast Medium Transactions (e.g., bus)
  - Fundamental system design abstraction
  - Logically single set of wires connect several devices
  - Protocol: arbitration, command/addr, data
    - Every device observes every transaction
  - Broadcast medium enforces *serialization* of read or write accesses ⇒ write serialization
    - 1st processor to get medium invalidates others copies
    - Implies cannot complete write until it obtains bus
    - All coherence schemes require serializing accesses to same cache block
- Also, need to find up-to-date copy of cache block

Locate up-to-date copy of data

- Write-through: get up-to-date copy from memory
  - Write through simpler if enough memory BW
- Write-back is harder
  - Most recent copy can be in a cache
- Can use same snooping mechanism
  1. Snoop every address placed on the bus
  2. If a processor has dirty copy of requested cache block, it provides it in response to a read request and aborts the memory access
    - Complexity from retrieving cache block from a processor cache, which can take longer than retrieving it from memory
- Write-back needs lower memory bandwidth
  ⇒ Supports larger numbers of faster processors
  ⇒ Most symmetric multiprocessors use write-back
**Cache Resources for WB Snooping**

- Normal cache tags can be used for snooping
- Valid bit per block makes invalidation easy
- Read misses easy since rely on snooping
- Writes $\Rightarrow$ Need to know if any other copies of the block are cached
  - No other copies $\Rightarrow$ No need to place write on bus for WB
  - Other copies $\Rightarrow$ Need to place invalidate on bus

---

**Cache Resources for WB Snooping**

- To track whether a cache block is shared, add extra state bit associated with each cache block, like valid bit and dirty bit
  - Write to Shared block $\Rightarrow$ Need to place invalidate on bus and mark cache block as private (if an option)
  - No further invalidations will be sent for that block
  - This processor called **owner** of cache block
  - Owner then changes state from shared to unshared (a.k.a. exclusive)
Cache behavior in response to bus

- Every bus transaction must check the cache-address tags
  - could potentially interfere with processor cache accesses
- A way to reduce interference is to duplicate tags
  - One set for cache access, one set for bus accesses
- Another way to reduce interference is to use L2 tags
  - Since L2 less heavily used than L1
  - every entry in L1 cache must be present in the L2 cache, called the **inclusion property**
  - If Snoop gets a hit in L2 cache, then it must arbitrate for the L1 cache to update the state and possibly retrieve the data, which usually requires a stall of the processor

Key concept

Invalidate all other processors’ copies before writing.

Stated another way:
- gain exclusive ownership before writing.
Example Protocol

- Snooping coherence protocol is usually implemented by incorporating a finite-state controller in each node.
- Logically, think of a separate controller associated with each cache block.
  - That is, snooping operations or cache requests for different blocks can proceed independently.
- In implementations, a single controller allows multiple operations to distinct blocks to proceed in interleaved fashion.
  - That is, one operation may be initiated before another is completed, even through only one cache access or one bus access is allowed at time.

Write-through Invalidate Protocol

- 2 states per block in each cache.
  - As in uniprocessor.
  - State of a block is a p-vector of states.
  - Hardware state bits associated with blocks that are in the cache.
  - Other blocks can be seen as being in invalid (not present) state in that cache.
- Writes invalidate all other cache copies.
  - Can have multiple simultaneous readers of block, but write invalidates them.

PrRd: Processor Read
PrWr: Processor Write
BusRd: Bus Read
BusWr: Bus Write
Is 2-state Protocol Coherent?

- Processor only observes state of memory system by issuing memory operations
- Assume bus transactions and memory operations are atomic and a one-level cache
  - all phases of one bus transaction complete before next one starts
  - processor waits for memory operation to complete before issuing next
  - with one-level cache, assume invalidations applied during bus transaction
- All writes go to bus + atomicity
  - Writes serialized by order in which they appear on bus (bus order)
  - => invalidations applied to caches in bus order
- How to insert reads in this order?
  - Important since processors see writes through reads, so determines whether write serialization is satisfied
  - But read hits may happen independently and do not appear on bus or enter directly in bus order
- Let’s understand other ordering issues

---

Ordering

- Writes establish a partial order
- Doesn’t constrain ordering of reads, though shared-medium (bus) will order read misses too
  - any order among reads between writes is fine, as long as in program order

---

Chapter 2 — Instructions: Language of the Computer

26 November 2013
Example Write Back Snoopy Protocol

- Invalidation protocol, write-back cache
  - Snoops every address on bus
  - If it has a dirty copy of requested block, provides that block in response to the read request and aborts the memory access
- Each memory block is in one state:
  - Clean in all caches and up-to-date in memory (Shared)
  - OR Dirty in exactly one cache (Exclusive)
  - OR Not in any caches
- Each cache block is in one state (track these):
  - Shared: block can be read
  - Exclusive: cache has the only copy, its writeable, and dirty
  - Invalid: block contains no data (in uniprocessor cache too)
- Read misses: cause all caches to snoop bus
- Writes to clean blocks are treated as misses

Write-Back State Machine - CPU

- State machine for CPU requests for each cache block
- Non-resident blocks invalid
- Cache Block State
  - Invalid
  - Shared (read/only)
  - Exclusive (read/write)

- CPU Read hit
- Place read miss on bus
- CPU Write
- Place Write Miss on bus
- CPU Read
- Place Write Miss on Bus
- CPU Write Miss ()
- Write back cache block
- Place write miss on bus
Write-Back State Machine - Bus request

- State machine for bus requests for each cache block

Invalid → Write miss for this block → Shared (read/only)

Exclusive (read/write) → Write Back Block; (abort memory access) → Invalid

Shared (read/only) → Read miss for this block → Write Back Block; (abort memory access) → Shared (read/only)

Block-replacement

- State machine for CPU requests for each cache block

Invalid → CPU Read → CPU Read hit → Invalid

Invalid → CPU Write → Place Write Miss on bus → CPU Read

Invalid → CPU Write Miss → Place write miss on bus → CPU Write

Invalid → CPU Read miss → Place read miss on bus → CPU Read

Shared (read/only) → CPU Read miss → Place read miss on bus → Shared (read/only)

Exclusive (read/write) → CPU read hit → CPU write hit → Exclusive (read/write)

CPU Write → Place Write Miss on Bus → Exclusive (read/write)
Chapter 2 — Instructions: Language of the Computer

Write-back State Machine: showing all transitions

- State machine for CPU requests for each cache block and for bus requests for each cache block

Cache Block State

CPU read hit
CPU write hit

Write Back Block; (abort memory access)

Write miss for this block

Write Back State Machine:

Exclusive (read/write)

Invalid

Shared (read/only)

CPU Read hit

Place read miss on bus

CPU Read

CPU Write

Place Write Miss on bus

CPU Write

Place Write Miss on Bus

CPU read miss

Write back block, Place read miss on bus

Write Back Block; (abort memory access)

Write miss for this block

Place Write Miss on bus

CPU Write

Place write miss on bus

CPU Write Miss

Write back cache block

Assumes A1 and A2 map to same cache block, initial cache state is invalid

Example

<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>Proc Addr</th>
<th>Proc Value</th>
<th>Memory Addr</th>
<th>Memory Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1: Write 10 to A1</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>
</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>
</tr>
<tr>
<td>P2: Write 20 to A1</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 to A2</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>

Assumes A1 and A2 map to same cache block, initial cache state is invalid
### Example

<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</td>
<td>Excl.</td>
<td>A1</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>P1: Write 10 to A1</td>
<td></td>
<td></td>
<td></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 to A1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: Write 40 to A2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Assumes A1 and A2 map to same cache block

### Example

<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</td>
<td>Excl.</td>
<td>A1</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>P1: Write 10 to A1</td>
<td></td>
<td></td>
<td></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 to A1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: Write 40 to A2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Assumes A1 and A2 map to same cache block
### Example

<table>
<thead>
<tr>
<th>Step</th>
<th>P1 State</th>
<th>Addr</th>
<th>Value</th>
<th>P2 State</th>
<th>Addr</th>
<th>Value</th>
<th>Bus Action</th>
<th>Proc. Addr</th>
<th>Value</th>
<th>Memory Addr</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1 Write 10 to A1</td>
<td>Excl.</td>
<td>A1</td>
<td>10</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>Excl.</td>
<td>A1</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P3: Write 20 to A1</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 to A2</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>

Assumes A1 and A2 map to same cache block

---

### Example

<table>
<thead>
<tr>
<th>Step</th>
<th>P1 State</th>
<th>Addr</th>
<th>Value</th>
<th>P2 State</th>
<th>Addr</th>
<th>Value</th>
<th>Bus Action</th>
<th>Proc. Addr</th>
<th>Value</th>
<th>Memory Addr</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1 Write 10 to A1</td>
<td>Excl.</td>
<td>A1</td>
<td>10</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>Excl.</td>
<td>A1</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: Write 20 to A1</td>
<td>Inv.</td>
<td>Excl.</td>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: Write 40 to A2</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>

Assumes A1 and A2 map to same cache block
Example

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>P1: Write 10 to A1</td>
<td>Excl.</td>
<td>A1</td>
<td>10</td>
<td>WrMs</td>
<td>P1</td>
<td>A1</td>
<td></td>
</tr>
<tr>
<td>P1: Read A1</td>
<td>Excl.</td>
<td>A1</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: Read A1</td>
<td>Shar.</td>
<td>A1</td>
<td>10</td>
<td>RdMs</td>
<td>P2</td>
<td>A1</td>
<td></td>
</tr>
<tr>
<td>P2: Write 20 to A1</td>
<td>Inv.</td>
<td>A1</td>
<td>20</td>
<td>WrMs</td>
<td>P2</td>
<td>A1</td>
<td>10</td>
</tr>
<tr>
<td>P2: Write 40 to A2</td>
<td>Excl.</td>
<td>A2</td>
<td>40</td>
<td>WrMs</td>
<td>P2</td>
<td>A2</td>
<td>10</td>
</tr>
</tbody>
</table>

Assumes A1 and A2 map to same cache block, but A1 != A2

Snoopy Coherence Protocols

- Complications for the basic MSI protocol:
  - Operations are not atomic
    - E.g. detect miss, acquire bus, receive a response
    - Creates possibility of deadlock and races
    - One solution: processor that sends invalidate can hold bus until other processors receive the invalidate

- Extensions:
  - Add exclusive state to indicate clean block in only one cache (MESI protocol)
    - Prevents needing to write invalidate on a write
  - Owned state
Coherence Protocols: Extensions

- Shared memory bus and snooping bandwidth is bottleneck for scaling symmetric multiprocessors
  - Duplicating tags
  - Place directory in outermost cache
  - Use crossbars or point-to-point networks with banked memory

Coherence Protocols

- AMD Opteron:
  - Memory directly connected to each multicore chip in NUMA-like organization
  - Implement coherence protocol using point-to-point links
  - Use explicit acknowledgements to order operations
Performance

- Review capacity, conflict, and compulsory misses
- Coherence influences cache miss rate
  - Coherence misses
    - True sharing misses
      - Write to shared block (transmission of invalidation)
      - Read an invalidated block
    - False sharing misses
      - Read an unmodified word in an invalidated block

Performance Study: Commercial Workload

![Chart showing normalized execution time for OLTP with varying L3 cache size (MB)]
Performance Study: Commercial Workload

Directory Protocols

- Directory keeps track of every block
  - Which caches have each block
  - Dirty status of each block
- Implement in shared L3 cache
  - Keep bit vector of size = # cores for each block in L3
  - Not scalable beyond shared L3
- Implement in a distributed fashion:
Directory Protocols

- For each block, maintain state:
  - Shared
    - One or more nodes have the block cached, value in memory is up-to-date
    - Set of node IDs
  - Uncached
  - Modified
    - Exactly one node has a copy of the cache block, value in memory is out-of-date
    - Owner node ID

- Directory maintains block states and sends invalidation messages

Messages

<table>
<thead>
<tr>
<th>Message type</th>
<th>Source</th>
<th>Destination</th>
<th>Message contents</th>
<th>Function of this message</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read miss</td>
<td>Local cache</td>
<td>Home directory</td>
<td>P, A</td>
<td>Node P has a read miss at address A; request data and make P a read sharer.</td>
</tr>
<tr>
<td>Write miss</td>
<td>Local cache</td>
<td>Home directory</td>
<td>P, A</td>
<td>Node P has a write miss at address A; request data and make P the exclusive owner.</td>
</tr>
<tr>
<td>Invalidate</td>
<td>Local cache</td>
<td>Home directory</td>
<td>A</td>
<td>Request to send invalidation to all remote caches that are caching the block at address A.</td>
</tr>
<tr>
<td>Invalidate</td>
<td>Home directory</td>
<td>Remote cache</td>
<td>A</td>
<td>Invalidate a shared copy of data at address A.</td>
</tr>
<tr>
<td>Fetch</td>
<td>Home directory</td>
<td>Remote cache</td>
<td>A</td>
<td>Fetch the block at address A and send it to its home directory; change the state of A in the remote cache to shared.</td>
</tr>
<tr>
<td>Fetch/invalidate</td>
<td>Home directory</td>
<td>Remote cache</td>
<td>A</td>
<td>Fetch the block at address A and send it to its home directory; invalidate the block in the cache.</td>
</tr>
<tr>
<td>Data value reply</td>
<td>Home directory</td>
<td>Local cache</td>
<td>D</td>
<td>Return a data value from the home memory.</td>
</tr>
<tr>
<td>Data write-back</td>
<td>Remote cache</td>
<td>Home directory</td>
<td>A, D</td>
<td>Write-back a data value for address A.</td>
</tr>
</tbody>
</table>
Directory Protocols

For uncached block:
- Read miss
  - Requesting node is sent the requested data and is made the only sharing node, block is now shared
- Write miss
  - The requesting node is sent the requested data and becomes the sharing node, block is now exclusive

For shared block:
- Read miss
  - The requesting node is sent the requested data from memory, node is added to sharing set
- Write miss
  - The requesting node is sent the value, all nodes in the sharing set are sent invalidate messages, sharing set only contains requesting node, block is now exclusive
Directory Protocols

- For exclusive block:
  - Read miss
    - The owner is sent a data fetch message, block becomes shared, owner sends data to the directory, data written back to memory, sharers set contains old owner and requestor
  - Data write back
    - Block becomes uncached, sharer set is empty
  - Write miss
    - Message is sent to old owner to invalidate and send the value to the directory, requestor becomes new owner, block remains exclusive

Synchronization

- Basic building blocks:
  - Atomic exchange
    - Swaps register with memory location
  - Test-and-set
    - Sets under condition
  - Fetch-and-increment
    - Reads original value from memory and increments it in memory
    - Requires memory read and write in uninterruptable instruction
  - load linked/store conditional
    - If the contents of the memory location specified by the load linked are changed before the store conditional to the same address, the store conditional fails
Uninterruptable Instruction to Fetch and Update Memory

- **Atomic exchange:** interchange a value in a register for a value in memory
  - 0 ⇒ synchronization variable is free
  - 1 ⇒ synchronization variable is locked and unavailable
- Set register to 1 & swap
- New value in register determines success in getting lock
  - 0 if you succeeded in setting the lock (you were first)
  - 1 if other processor had already claimed access
- Key is that exchange operation is indivisible

- **Test-and-set:** tests a value and sets it, if the value passes the test
- **Fetch-and-increment:** it returns the value of a memory location and atomically increments it
  - 0 ⇒ synchronization variable is free

---

Uninterruptable Instruction to Fetch and Update Memory

- Hard to have read & write in 1 instruction: use 2 instead
- **Load linked** (or load locked) + store conditional
  - Load linked returns the initial value
  - Store conditional returns 1 if it succeeds (no other store to same memory location since preceding load) and 0 otherwise

**Example doing atomic swap with LL & SC:**

```assembly
try: mov R3,R4 ; mov exchange value
ll R2,0(R1) ; load linked
sc R3,0(R1) ; store conditional
beqz R3,try ; branch store fails (R3 = 0)
mov R4,R2 ; put load value in R4
```

**Example doing fetch & increment with LL & SC:**

```assembly
try: ll R2,0(R1) ; load linked
addi R2,R2,#1 ; increment (OK if reg-reg)
sc R2,0(R1) ; store conditional
beqz R2,try ; branch store fails (R2 = 0)
```
User Level Synchronization Using this Primitive

- **Spin locks:** processor continuously tries to acquire, spinning around a loop trying to get the lock

  ```
  li R0,#1
  lockit:  exch R2,0(R1)  ;atomic exchange
           bnez R2,lockit  ;already locked?
  ```

- **What about MP with cache coherency?**
  - Want to spin on cache copy to avoid full memory latency
  - Likely to get cache hits for such variables

- **Problem:** exchange includes a write, which invalidates all other copies; this generates considerable bus traffic

- **Solution:** start by simply repeatedly reading the variable; when it changes, then try exchange ("test and test&set"):?

  ```
  try:    li  R2,#1
  lockit: lw  R2,0(R1) ;load value
           bnez R3,lockit ;≠ 0 ⇒ not free ⇒ spin
           exch R2,0(R1);atomic exchange
           bnez R2,try  ;already locked?
  ```

Implementing Locks

- **Spin lock**
  - **If no coherence:**
    ```
    DADDUI R2,R0,#1
    lockit: EXCH R2,0(R1)  ;atomic exchange
            BNEZ R2,lockit  ;already locked?
    ```
  - **If coherence:**
    ```
    lockit:  LD R2,0(R1)  ;load of lock
              BNEZ R2,lockit  ;not available-spin
    DADDUI R2,R0,#1  ;load locked value
    EXCH R2,0(R1)  ;swap
    BNEZ R2,lockit  ;branch if lock wasn’t 0
    ```
Implementing Locks

- If coherence, improved:

lockit: \texttt{LL R2,0(R1)} ;load linked
\texttt{BNEZ R2,lockit} ;not available-spin
\texttt{DADDUI R2,R0,#1} ;load locked value
\texttt{SC R2,0(R1)} ;store conditional
\texttt{BNEZ R2,lockit} ;branch if store fails

Read & Write are explicitly separated
so load-linked causes no bus traffic.

Advantage of this scheme: reduces memory traffic.
Models of Memory Consistency

Processor 1:  
A=0  
  ...  
  A=1  
  if (B==0) ...

Processor 2:  
B=0  
  ...  
  B=1  
  if (A==0) ...

- It should be impossible for both if-statements to be evaluated as true.
  - Delayed write invalidate?

- Sequential consistency:
  - Result of execution should be the same as long as:
    - Accesses on each processor were kept in order
    - Accesses on different processors were arbitrarily interleaved

Implementing Locks

- To implement, delay completion of all memory accesses until all invalidations caused by the access are completed
  - Reduces performance!

- Alternatives:
  - Program-enforced synchronization to force write on processor to occur before read on the other processor
    - Requires synchronization object for A and another for B
      - "Unlock" after write
      - "Lock" after read
Relaxed Consistency Models

- **Rules:**
  - X → Y
    - Operation X must complete before operation Y is done
  - Sequential consistency requires:
    - R → W, R → R, W → R, W → W
  - Relax W → R
    - "Total store ordering"
  - Relax W → W
    - "Partial store order"
  - Relax R → W and R → R
    - "Weak ordering" and "release consistency"

Relaxed Consistency Models

- Consistency model is multiprocessor specific

- Programmers will often implement explicit synchronization

- Speculation gives much of the performance advantage of relaxed models with sequential consistency
  - Basic idea: if an invalidation arrives for a result that has not been committed, use speculation recovery