ESE 545 Computer Architecture

Symmetric Multiprocessors and Snoopy Cache Coherence Protocols
Use snoopy mechanism to keep all processors’ view of memory coherent
Symmetric Shared-Memory Architectures

• From multiple boards on a shared bus to multiple processors inside a single chip
• Caches both
  – *Private data* are used by a single processor
  – *Shared data* are used by multiple processors
• Caching shared data
  ⇒ reduces latency to shared data, memory bandwidth for shared data, and interconnect bandwidth
  ⇒ cache coherence problem
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 when
  » Processes accessing main memory may see very stale value
- Unacceptable for programming, and it's frequent!
Example

/*Assume initial value of A and flag is 0*/
A = 1; while (flag == 0); /*spin idly*/
flag = 1; print A;

- Intuition not guaranteed by coherence
- expect memory to respect order between accesses to different locations issued by a given process
  - to preserve orders among accesses to same location by different processes
- Coherence is not enough!
  - pertains only to single location
Intuitive Memory Model

- Too vague and simplistic; 2 issues
  1. **Coherence** defines values returned by a read
  2. **Consistency** determines when a written value will be returned by a read

- Reading an address should **return the last value written to that address**
  - Easy in uniprocessors, except for I/O
Defining Coherent Memory System

1. **Preserve Program Order**: A read by processor P to location X that follows a write by P to X, with no writes of X by another processor occurring between the write and the read by P, always returns the value written by P.

2. **Coherent view of memory**: Read by a processor to location X that follows a write by another processor to X 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.

3. **Write serialization**: 2 writes to same location by any 2 processors are seen in the same order by all processors.
   - If not, a processor could keep value 1 since saw as last write.
   - For example, if the values 1 and then 2 are written to a location, processors can never read the value of the location as 2 and then later read it as 1.
Write Consistency

• For now assume

1. A write does not complete (and allow the next write to occur) until all processors have seen the effect of that write

2. The processor does not change the order of any write with respect to any other memory access

⇒ 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

• These restrictions allow the processor to reorder reads, but forces the processor to finish writes in program order
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
2 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 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**
  => Support larger numbers of faster processors
  => Most 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 know whether 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 ⇒ 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 (or 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 caches access, one set for bus accesses

• Another way to reduce interference is to use L2 tags
  – Since L2 less heavily used than L1
  \[\Rightarrow\] 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
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

PrRd/ --
PrWr / BusWr

BusWr / -

PrRd / BusRd

PrWr / BusWr

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
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
  – OR Exclusive : cache has only copy, its writeable, and dirty
  – OR 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

CPU Read
Place read miss on bus

CPU Read hit

Shared (read/only)

CPU Write
Place Write Miss on bus

Exclusive (read/write)

CPU read hit
CPU write hit

CPU Write
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
  - Write Back Block; (abort memory access)

- Exclusive (read/write)
  - Read miss for this block
  - Write Back Block; (abort memory access)

- Shared (read/only)
  - Write miss for this block
Block-replacement

- State machine for CPU requests for each cache block

Cache Block State

- Invalid
  - CPU Read
    - Place read miss on bus

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

- Exclusive (read/write)
  - CPU read miss
    - Write back block, Place read miss on bus
  - CPU Read
    - Place read miss on bus
  - CPU Write
    - Place Write Miss on Bus
  - CPU Write Miss
    - Write back cache block Place write miss on bus
  - CPU read hit
    - CPU write hit

CPU Read hit

CPU Read miss

CPU Write

CPU Write Miss
Write-back State Machine-III

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

**Cache Block State**

- **Invalid**
  - Write missing for this block
  - Write Back Block; (abort memory access)
- **Shared (read/only)**
  - Place read miss on bus
- **Exclusive (read/write)**
  - Write miss for this block
  - Write Back Block; (abort memory access)
  - CPU Read
  - Place read miss on bus
  - CPU Write
  - Place Write Miss on bus
  - CPU Read miss
  - Place read miss on bus
  - CPU Write
  - Place Write Miss on Bus
- **CPU Read hit**
- **CPU Read miss**
  - Place read miss on bus
  - CPU Write
  - Place Write Miss on Bus
- **CPU Write Miss**
  - Write Back Cache Block
  - Place write miss on bus
  - CPU Read hit
  - CPU Write
  - Place Write Miss on Bus
  - Read miss for this block
  - Write Back Block; (abort memory access)
## Example

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

<table>
<thead>
<tr>
<th>step</th>
<th>P1</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>P1 Write 10 to A1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P1: Read A1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: Read A1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: Write 20 to A1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: Write 40 to A2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Example

Assumes A1 and A2 map to same cache block

<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.</th>
<th>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>WrMs</td>
<td>P1</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 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>
<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>
<td></td>
</tr>
</tbody>
</table>
### Example

<table>
<thead>
<tr>
<th>step</th>
<th>P1</th>
<th>P2</th>
<th>Bus</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<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: Cache Coherence Operations

<table>
<thead>
<tr>
<th>Step</th>
<th>P1: Write 10 to A1</th>
<th>P1: Read A1</th>
<th>P2: Read A1</th>
<th>P2: Write 20 to A1</th>
<th>P2: Write 40 to A2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Value: 10</td>
<td>Value: 10</td>
<td>Value: 10</td>
<td>Value: 10</td>
<td>Value: 10</td>
</tr>
<tr>
<td>Bus Action</td>
<td>WrMs</td>
<td>RdMs</td>
<td>WrBk</td>
<td>RdDa</td>
<td></td>
</tr>
<tr>
<td>Proc. Addr</td>
<td>P1</td>
<td>P2</td>
<td>P1</td>
<td>P2</td>
<td></td>
</tr>
<tr>
<td>Value</td>
<td>A1</td>
<td>A1</td>
<td>A1</td>
<td>A1</td>
<td></td>
</tr>
<tr>
<td>Memory Addr</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory Value</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Notes:
- Assumes A1 and A2 map to the same cache block.

---

CA SMP and cache coherence

30
Example

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

Assumes A1 and A2 map to same cache block, but A1 != A2.
MESI: An Enhanced MSI protocol

increased performance for private data

*Each* cache line has a tag

M: Modified Exclusive
E: Exclusive but unmodified
S: Shared
I: Invalid

<table>
<thead>
<tr>
<th>state bits</th>
<th>Address tag</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Write miss
- \(P_1\) write or read
- \(P_1\) intent to write
- \(P_1\) write
- \(P_1\) read
- Read miss, not shared
- Read by any processor
- Read miss, shared
- \(P_1\) writes back
- \(P_1\) writes back
- Other processor reads
- Other processor reads
- Other processor reads
- Other processor reads

Cache state in processor \(P_1\)
And in Conclusion

- “End” of uniprocessors speedup => Multiprocessors
- Parallelism challenges: % parallelizable, long latency to remote memory
- Centralized vs. distributed memory
  - Small MP vs. lower latency, larger BW for Larger MP
- Message Passing vs. Shared Address
  - Uniform access time vs. Non-uniform access time
- Snooping cache over shared medium for smaller MP by invalidating other cached copies on write
- Sharing cached data ⇒ Coherence (values returned by a read), Consistency (when a written value will be returned by a read)
- Shared medium serializes writes ⇒ Write consistency
Acknowledgements

- These slides contain material developed and copyright by:
  - Morgan Kauffmann (Elsevier, Inc.)
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)
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 in shared memory symmetric multiprocessors