Defining Coherent Memory System

- **Preserve Program Order**: A write by processor P to location X followed by a read by P from X, if no write to X by another processor occurs between the write and the read by P, always returns the value written by P.
- **Coherent view of memory**: A write by one processor to location X followed by a read of X by another processor returns the newly written value if the read and write are sufficiently separated in time and no other writes to X occur between the two accesses.
- **Write serialization**: Two writes to same location by any two processors are seen in the same order by all processors.
  - If not, a processor could keep value 1 if saw it 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
  - "Writes 1 => 2: same order everywhere"

Write Consistency (for writes to 2+ variables)

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

- If one 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 processors to reorder reads, but forces all processors to finish writes in program order {Reads not change data => any order OK}

Basic Schemes for Enforcing Coherence

- A program on multiple processors will normally have copies of the same data in several caches.
  - Unlike I/O, where multiple copies of cached data are very rare.
- Rather than trying to avoid sharing in SW, SMPs use a HW protocol to maintain coherent caches.
  - Migration and replication are keys to performance for shared data.
  - Migration – (private) 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 and interconnection.
  - Replication – for reading shared data simultaneously, since caches make a copy of data in local cache.
    - Reduces both latency of access and contention for read-shared data.
    - Like “mirror” sites for web downloads.
Two Classes of Cache Coherence Protocols

- **Directory-based** — Sharing status of a physical memory block is kept in just one location, the directory entry for that block

- **Snooping ("Snoopy")** — Every cache with a copy of a data block also has a copy of the sharing status of the block, but no centralized state is kept
  - All caches have access to addresses of writes and cache-misses via some broadcast medium (a bus or cross-bar switch)
  - All cache controllers monitor or snoop on the shared medium to determine whether or not they have a local cache copy of each block that is requested by a bus or switch access

Example: Write-Thru Invalidate

- Must invalidate at least P1’s cache copy \( u=5 \) before step 3
- Write update uses more broadcast medium bandwidth (must share both full address and new value)
  \[ \Rightarrow \text{all recent MPUs use write_invalidate (send just cache block #)} \]

Snoopy Cache-Coherence Protocols

- Cache Controller "snoops" on all transactions on the shared medium (bus or switch)
  - Relevant transaction if it is for a block the \( P1 \) cache contains
  - If relevant, a cache controller takes action to ensure coherence
    - invalidate, update, or supply the latest value
    - depends on state of the block and the protocol
- A cache either gets exclusive access before a write via write_invalidate, or updates all copies when it writes

Architectural Building Blocks

- Cache block state transition diagram
  - Finite State Machine (FSM) specifying conditions of block state changes
    - Minimum number of states is 3: invalid, valid/shared, dirty
- Broadcast Medium Transactions (e.g., bus)
  - Fundamental system design abstraction
  - Logically, a single set of wires connect several devices
  - Protocol: arbitration, command/address, data
  \[ \Rightarrow \text{Every device observes every transaction} \]
- Broadcast medium enforces serialization of read or write accesses
  \[ \Rightarrow \text{Write serialization} \]
  - 1st processor to get medium (e.g., bus) invalidates others’ copies
  - Implies that a write to shared item cannot complete until PU obtains bus
  - Coherence schemes require serializing all accesses to the same cache block
  - Also need to find up-to-date copy of cache block
Locate up-to-date copy of data

- Write-through (old): memory has up-to-date copy
  - Write-through simpler logic if enough memory BW to support it
- Write-back harder, but uses must less memory BW
  - The most recent version of a cache block may not be in memory
- Can use same snooping mechanism for write-back
  - Snoop every address placed on the bus, as for write-through
    - If a processor has "dirty" copy of a requested cache block, it provides it in response to a read request from another processor’s cache system and aborts the access to memory
  - Complexity of retrieving cache block from a processor cache, which can take longer than retrieving it from memory
- Write-back caches need lower memory bandwidths
  - Support larger numbers of faster processors
  - Most multiprocessor caches: write-back, maybe not L1 ➔ L2
  - All modern processors: write-back caches ➔ memory

Cache Resources for Write-Back Snooping

- Normal cache indices+tags can be used for snooping
  - But often have 2nd copy of tags (without data) for speed
- Valid bit per cache block makes invalidation easy
- Read misses are easy since they can rely on snooping
- Writes (hit) ⇒ Need to know whether any other valid copies of the block are cached
  - If no other copies ⇒ No need to (wait to) place write on bus for WB
  - If other copies ⇒ Must wait for bus access to place invalidate on bus

Cache Resources for WB Snooping (cont.)

- To track whether a cache block is shared, add one more state bit to each cache block, like the valid and dirty bits (Dirty says block copy of block as modified (valid but not in memory)
  - Write miss ⇒ Put read miss block# on bus and mark own cache copy of block as modified (valid but not in memory)
- Read miss ⇒ Put invalid block# on bus and mark own cache copy of block as modified (valid but not in memory)
  - If another cache has a valid copy or if there is no exclusive option in the protocol, mark new copy as shared
    - Otherwise, mark new copy as exclusive (the only valid copy)
- No more invalidations are sent if CPU writes to its own modified or exclusive blocks; writes switch exclusive blocks to modified
  - The last processor that modified a cache block is its owner and, if protocol allows, may send copies of the block to other caches
- Cache states: Modified/dirty, Shared, Invalid, Exclusive, Owner
- Common snoopy protocols: MSI, MESI, MOESI, MOSSI

Cache Behavior in Response to Bus

- Every bus transaction must check the cache-address tags
  - Could slow processor memory loads from L1 caches
- A way to reduce interference is to duplicate tags
  - One set for CPU cache accesses, one set for bus accesses
- Another way to reduce interference is to use L2 tags
  - Level 2 (L2) caches are less frequently used than L1 caches so snooping bursts of bus transactions rarely stalls the processor
    - Checking L2 tags requires every block in the L1 cache always to be in the L2 cache also; this restriction is the "inclusion property"
    - If a Snoop hits an L2 cache tag, the L2 must arbitrate with its L1 cache to update the L1 block state and maybe to retrieve a new cache block; retrieving new L1 data usually stalls the processor
Example Protocol

- Snooping coherence protocol is usually implemented by incorporating a finite-state machine controller (FSM) 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 a time.

Is Two-State Protocol Coherent?

- Processor only observes state of memory system by issuing memory operations.
- Assume bus transactions and memory operations are atomic and each processor has 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, which determine 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.

Write-through Snoopy Invalidate Protocol

- 2 states per block in any of p caches.
  - As in uniprocessor (Valid, Invalid)
  - State of a memory block is a p-vector of states.
  - Hardware state bits are associated with blocks that are in a cache.
  - Other blocks are seen as having invalid (not-present) state in that cache.
- Writes invalidate all other cache copies.
  - Can have multiple simultaneous readers of a block, but each write invalidates all other copies held by multiple readers.

Ordering

- Writes establish a partial order.
- Does not constrain ordering of reads, though shared-medium (bus) will order read misses too.
  - Any order of reads by different CPUs between writes is fine, so long as reads are in program order for each CPU.
Example Write-Back Snoopy Protocol

- Invalidation protocol, write-back cache
  - Each cache controller snoops every address on shared bus
  - If cache 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 (non-dirty) in all caches and up-to-date in memory (Shared), or
  - Dirty in exactly one cache (Modified), or
  - Not in any caches
- Each cache block is in one of the three states (track these):
  - Shared: block can be read, or
  - Modified: cache has only copy, it is writeable or readable, and it is dirty, or
  - Invalid: block contains no data (used in uniprocessor cache too)
- Read misses: cause all caches to snoop bus
- Writes to clean blocks are treated as misses

Write-Back State Machine – Request from Bus

- State machine for bus requests for each cache block
- Non-resident blocks are invalid

Cache Block States

- Invalid
- Shared (read-only)
- Modified (read/write)

Example

Assumes A1 maps to the same cache block on both CPUs and each initial cache block state for A1 is invalid (last slide in this example also assumes that addresses A1 and A2 map to the same block index but have different address tags, so they are in different cache blocks that complete for the same location in the cache).
Example

Assumes A1 maps to the same cache block on both CPUs.

Example

Assumes A1 maps to the same cache block on both CPUs.

Example

Assumes A1 maps to the same cache block on both CPUs.

Note: in this protocol the only states for a valid cache block are “modified” and “shared”, so each new reader of a block assumes it is “shared”, even if it is the first CPU reading the block. The state changes to “modified” when a CPU first writes to the block and makes any other copies become “invalid”. If a dirty cache block is forced from “modified” to “shared” by a RdMiss from another CPU, the cache with the latest value writes its block back to memory for the new CPU to read the data.

Example

Assumes A1 maps to the same cache block on both CPUs.
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>
</tr>
</thead>
<tbody>
<tr>
<td>P1 Write A1 to A1</td>
<td>Mod</td>
<td>A1</td>
<td>10</td>
<td>Mod</td>
<td>A1</td>
<td>10</td>
<td>WMA</td>
<td>P1</td>
<td>A1</td>
<td></td>
</tr>
<tr>
<td>P2 Write A2 to A1</td>
<td>Inc</td>
<td>Mod</td>
<td>A1</td>
<td>20</td>
<td>WMA</td>
<td>P2</td>
<td>A1</td>
<td>A1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2 Write A2 to A2</td>
<td>Mod.</td>
<td>A2</td>
<td>20</td>
<td>WMA</td>
<td>P2</td>
<td>A1</td>
<td>20</td>
<td>A1</td>
<td>20</td>
<td></td>
</tr>
</tbody>
</table>

Assumes that, like A1, A2 maps to the same cache block on both CPUs and addresses A1 and A2 map to the same block index but have different address tags, so A1 and A2 are in different memory blocks that complete for the same location in the caches on both CPUs. Writing A2 forces P2’s dirty cache block for A1 to be written back before it is replaced by A2’s soon-dirty memory block.

In Conclusion [SMP]

- “Decline” of uniprocessor's speedup rate/year => Multiprocessors are good choices for MPU chips
- Parallelism challenges: % parallelizable, long latency to remote memory
- Centralized vs. distributed memory
  - Small MP limit but lower latency; need larger BW for larger MP
- Message Passing vs. Shared Address MPs
  - Shared: Uniform access time or Non-uniform access time (NUMA)
- Snooping cache over shared medium for smaller MP by invalidating other cached copies on write
- Sharing cached data ⇒ Coherence (values returned by reads to one address), Consistency (when a written value will be returned by a read for multiple addresses)
- Shared medium serializes writes, ⇒ Write consistency