Lecture 11
Thread Level Parallelism (4)

EEC 171 Parallel Architectures
John Owens
UC Davis
Credits

Programming Model 1: Shared Memory

- Program is a collection of threads of control.
  - Can be created dynamically, mid-execution, in some languages
- Each thread has a set of private variables, e.g., local stack variables
- Also a set of shared variables, e.g., static variables, shared common blocks, or global heap.
  - Threads communicate implicitly by writing and reading shared variables.
  - Threads coordinate by synchronizing on shared variables.
Shared Memory

\[ y = s \]

\[ s = \ldots \]

Shared memory

Private memory

Po

P1

Pn
Programming Model 2: Message Passing

- Program consists of a collection of named processes.
  - Usually fixed at program startup time
  - Thread of control plus local address space—NO shared data.
  - Logically shared data is partitioned over local processes.

\[\begin{align*}
  y &= \ldots s \\
  s &\colon 12 \\
  i &\colon 2 \\
  s &\colon 14 \\
  i &\colon 3 \\
  s &\colon 11 \\
  i &\colon 1
\end{align*}\]
Machine Model 2a: Distributed Memory

- Cray T3E, IBM SP2
- PC Clusters (Berkeley NOW, Beowulf)
- IBM SP-3, Millennium, CITRIS are distributed memory machines, but the nodes are SMPs.
- Each processor has its own memory and cache but cannot directly access another processor’s memory.
- Each “node” has a Network Interface (NI) for all communication and synchronization.
Tflop/s Clusters

- The following are examples of clusters configured out of separate networks and processor components
  - 72% of Top 500 (Nov 2005), 2 of top 10
- Dell cluster at Sandia (Thunderbird) is #4 on Top 500
  - 8000 Intel Xeons @ 3.6GHz
  - 64TFlops peak, 38 TFlops Linpack
  - Infiniband connection network
- Walt Disney Feature Animation (The Hive) is #96
  - 1110 Intel Xeons @ 3 GHz
  - Gigabit Ethernet
- Saudi Oil Company is #107
- Credit Suisse/First Boston is #108
Machine Model 2b: Internet/Grid Computing

- SETI@Home: Running on 500,000 PCs
  - ~1000 CPU Years per Day, 485,821 CPU Years so far
- Sophisticated Data & Signal Processing Analysis
- Distributes Datasets from Arecibo Radio Telescope

Next Step—Allen Telescope Array
Arecibo message
Programming Model 2c: Global Address Space

- Program consists of a collection of named threads.
  - Usually fixed at program startup time
- Local and shared data, as in shared memory model
- But, shared data is partitioned over local processes
- Cost model says remote data is expensive
- Examples: UPC, Titanium, Co-Array Fortran
- Global Address Space programming is an intermediate point between message passing and shared memory
Machine Model 2c: Global Address Space

- Cray T3D, T3E, X1, and HP Alphaserver cluster
- Clusters built with Quadrics, Myrinet, or Infiniband
- The network interface supports RDMA (Remote Direct Memory Access)
  - NI can directly access memory without interrupting the CPU
  - One processor can read/write memory with one-sided operations (put/get)
  - Not just a load/store as on a shared memory machine
    - Continue computing while waiting for memory op to finish
  - Remote data is typically not cached locally

Global address space may be supported in varying degrees
Programming Model 3: Data Parallel

- Single thread of control consisting of parallel operations.
- Parallel operations applied to all (or a defined subset) of a data structure, usually an array
  - Communication is implicit in parallel operators
  - Elegant and easy to understand and reason about
  - Coordination is implicit—statements executed synchronously
  - Similar to Matlab language for array operations
- Drawbacks:
  - Not all problems fit this model
  - Difficult to map onto coarse-grained machines

\[ A = \text{array of all data} \]
\[ fA = f(A) \]
\[ s = \text{sum}(fA) \]
Programming Model 4: Hybrids

- These programming models can be mixed
  - Message passing (MPI) at the top level with shared memory within a node is common
  - New DARPA HPCS languages mix data parallel and threads in a global address space
  - Global address space models can (often) call message passing libraries or vice versa
  - Global address space models can be used in a hybrid mode
    - Shared memory when it exists in hardware
    - Communication (done by the runtime system) otherwise
Machine Model 4: Clusters of SMPs

- SMPs are the fastest commodity machine, so use them as a building block for a larger machine with a network.

- Common names:
  - CLUMP = Cluster of SMPs
  - Hierarchical machines, constellations

- Many modern machines look like this:
  - Millennium, IBM SPs, ASCI machines

- What is an appropriate programming model for #4?
  - Treat machine as “flat”, always use message passing, even within SMP (simple, but ignores an important part of memory hierarchy).
  - Shared memory within one SMP, but message passing outside of an SMP.
Challenges of Parallel Processing

- Application parallelism ⇒ primarily via new algorithms that have better parallel performance
- 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
Fundamental Problem

- Many processors working on a task
- Those processors share data, need to communicate, etc.
- For efficiency, we use caches
- This results in multiple copies of the data
- Are we working with the right copy?
Symmetric Shared-Memory Architectures

- From multiple boards on a shared bus to multiple processors inside a single chip

- Caches:
  - 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
  - introduces a 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!
Intuitive Memory Model

- Reading an address should return the last value written to that address
  - Easy in uniprocessors, except for I/O
- Too vague and simplistic; 2 issues
  - **Coherence** defines values returned by a read
  - **Consistency** determines when a written value will be returned by a read
  - **Coherence** defines behavior for same processor, **Consistency** defines behavior for other processors
Defining Coherent Memory System

- **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.
  - P writes D to X
  - Nobody else writes to X
  - P reads X \( \rightarrow \) always gives D
Defining Coherent Memory System

- **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

  - P1 writes D to X
  - Nobody else writes to X
  - ... wait a while ...
  - P2 reads X, should get D
Defining Coherent Memory System

- **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
  • 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
  • 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 Coherence with Performance

• Program on multiple processors will normally have copies of the same data in several caches
  • Unlike I/O, where it’s rare
• SMPs use a HW protocol to maintain coherent caches
  • Migration and Replication key to performance of shared data
Basic Schemes for Coherence with Performance

- **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 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.
2 Classes of Cache Coherence Protocols

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

- **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)
- Does this transaction concern data that I have?
- If so, take action to ensure coherence
  - invalidate (my val), update (my val), or supply (my value) (when, when, and when?)
- 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, exclusive
- Broadcast Medium Transactions (e.g., bus)
  - Fundamental system design abstraction
  - Logically single set of wires connect several devices
  - Protocol: arbitration, command/address, data
  - Every device observes every transaction
Architectural Building Blocks

- 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 (on read for instance)
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
  - Snoop every address placed on the bus
  - 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 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 ⇒ Need to know if any other copies of the block are cached (“shared”)
  • No other copies ⇒ No need to place write on bus for WB
  • Other copies ⇒ 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
  - 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 though only one cache access or one bus access is allowed at time
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, it’s 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

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

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

**CPU Read hit**
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)

Shared (read/only)

Write miss for this block

Read miss for this block

Write Back Block; (abort memory access)
Write-back State Machine-III

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

Diagram:

- **Invalid** State:
  - Write miss for this block
  - CPU Read
  - Place read miss on bus

- **Shared** (read/only) State:
  - Write miss for this block
  - CPU Read hit

- **Exclusive** (read/write) State:
  - Write miss for this block
  - CPU Read miss on bus
  - Write back block, Place read miss on bus

- **Cache Block State**:
  - CPU read hit
  - CPU write hit
  - Write back cache block
  - Place write miss on bus

- Transitions:
  - CPU Write
  - Place Write
  - Miss on bus
  - CPU read miss
  - Write Back block; (abort memory access)
  - CPU Write
  - Place Write Miss on Bus
  - Read miss for this block
  - Write Back Block; (abort memory access)
**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>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>
<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>

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

Assumes A1 and A2 map to same cache block
Example

Assumes A1 and A2 map to same cache block
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</th>
<th>Addr</th>
<th>Value</th>
<th>Memory Addr</th>
<th>Memory 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>RdMs</td>
<td>P2</td>
<td>A1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Shar.</td>
<td>A1</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td>WrBk</td>
<td>P1</td>
<td>A1</td>
<td>10</td>
<td>A1</td>
<td>10</td>
</tr>
<tr>
<td>P2: Write 20 to A1</td>
<td>Inv.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>RdDa</td>
<td>P2</td>
<td>A1</td>
<td>10</td>
<td>A1</td>
<td>10</td>
</tr>
<tr>
<td>P2: Write 40 to A2</td>
<td>Excl.</td>
<td>A1</td>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td>WrMs</td>
<td>P2</td>
<td>A1</td>
<td></td>
<td>A1</td>
<td>10</td>
</tr>
</tbody>
</table>

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

Assumes A1 and A2 map to same cache block, but A1 != A2
Performance of Symmetric Shared-Memory Multiprocessors

- Cache performance is combination of
  - Uniprocessor cache miss traffic
  - Traffic caused by communication
    - Results in invalidations and subsequent cache misses
- 4th C: coherence miss
  - Joins Compulsory, Capacity, Conflict
Coherency Misses

- True sharing misses arise from the communication of data through the cache coherence mechanism
  - Invalidates due to 1st write to shared block
  - Reads by another CPU of modified block in different cache
  - Miss would still occur if block size were 1 word

- False sharing misses when a block is invalidated because some word in the block, other than the one being read, is written into
  - Invalidation does not cause a new value to be communicated, but only causes an extra cache miss
  - Block is shared, but no word in block is actually shared
    $\Rightarrow$ miss would not occur if block size were 1 word
Example: True v. False Sharing v. Hit?

- Assume $x_1$ and $x_2$ in same cache block. P1 and P2 both read $x_1$ and $x_2$ before.

<table>
<thead>
<tr>
<th>Time</th>
<th>P1</th>
<th>P2</th>
<th>True, False, Hit? Why?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Write $x_1$</td>
<td></td>
<td>True miss; invalidate $x_1$ in P2</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>Read $x_2$</td>
<td>False miss; $x_1$ irrelevant to P2</td>
</tr>
<tr>
<td>3</td>
<td>Write $x_1$</td>
<td></td>
<td>False miss; $x_1$ irrelevant to P2</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>Write $x_2$</td>
<td>False miss; $x_1$ irrelevant to P2</td>
</tr>
<tr>
<td>5</td>
<td>Read $x_2$</td>
<td></td>
<td>True miss; invalidate $x_2$ in P1</td>
</tr>
</tbody>
</table>
Review

- Caches contain all information on state of cached memory blocks
- Snooping cache over shared medium for smaller MP by invalidating other cached copies on write
- Sharing cached data $\Rightarrow$ Coherence (values returned by a read), Consistency (when a written value will be returned by a read)
A Cache Coherent System Must:

- Provide set of states, state transition diagram, and actions
- Manage coherence protocol
  - (o) Determine when to invoke coherence protocol
  - (a) Find info about state of block in other caches to determine action
    - whether need to communicate with other cached copies
  - (b) Locate the other copies
  - (c) Communicate with those copies (invalidate/update)
- (o) is done the same way on all systems
  - state of the line is maintained in the cache
  - protocol is invoked if an “access fault” occurs on the line
- Different approaches distinguished by (a) to (c)
Bus-based Coherence

- All of (a), (b), (c) done through broadcast on bus
  - faulting processor sends out a “search”
  - others respond to the search probe and take necessary action
- Could do it in scalable network too
  - broadcast to all processors, and let them respond
- Conceptually simple, but broadcast doesn’t scale with p
  - on bus, bus bandwidth doesn’t scale
  - on scalable network, every fault leads to at least $p$ network transactions
- Scalable coherence:
  - can have same cache states and state transition diagram
  - different mechanisms to manage protocol
Scalable Approach: Directories

• Every memory block has associated directory information
  • keeps track of copies of cached blocks and their states
  • on a miss, find directory entry, look it up, and communicate only with the nodes that have copies if necessary

• in scalable networks, communication with directory and copies is through network transactions

• Many alternatives for organizing directory information
Basic Operation of Directory

- k processors.
- With each cache-block in memory: k presence-bits, 1 dirty-bit
- With each cache-block in cache: 1 valid bit, and 1 dirty (owner) bit

- Read from main memory by processor i:
  - If dirty-bit OFF then { read from main memory; turn p[i] ON; }
  - If dirty-bit ON then { recall line from dirty proc (cache state to shared); update memory; turn dirty-bit OFF; turn p[i] ON; supply recalled data to i; }

- Write to main memory by processor i:
  - If dirty-bit OFF then { supply data to i; send invalidations to all caches that have the block; turn dirty-bit ON; turn p[i] ON; ... }
  - ...
Directory Protocol

- Similar to Snoopy Protocol: Three states
  - Shared: ≥ 1 processors have data, memory up-to-date
  - Uncached (no processor has it; not valid in any cache)
  - Exclusive: 1 processor (owner) has data; memory out-of-date
- In addition to cache state, must track which processors have data when in the shared state (usually bit vector, 1 if processor has copy)
  - Why isn’t this necessary for snoopy protocol?
- Keep it simple:
  - Writes to non-exclusive data ⇒ write miss
  - Processor blocks until access completes
  - Assume messages received and acted upon in order sent
Directory Protocol

• No bus and don’t want to broadcast:
  • interconnect no longer single arbitration point
  • all messages have explicit responses

• Terms: typically 3 processors involved
  • Local node where a request originates
  • Home node where the memory location of an address resides
  • Remote node has a copy of a cache block, whether exclusive or shared

• Example messages on next slide:
  P = processor number, A = address
<table>
<thead>
<tr>
<th>Message type</th>
<th>Source</th>
<th>Destination</th>
<th>Msg Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read miss</td>
<td>Local cache</td>
<td>Home directory</td>
<td>P, A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>– Processor P reads data at address A; make P a read sharer and request data</td>
</tr>
<tr>
<td>Write miss</td>
<td>Local cache</td>
<td>Home directory</td>
<td>P, A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>– Processor P has a write miss at address A; make P the exclusive owner and request data</td>
</tr>
<tr>
<td>Invalidate</td>
<td>Home directory</td>
<td>Remote caches</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>– Invalidate a shared copy at address A</td>
</tr>
<tr>
<td>Fetch</td>
<td>Home directory</td>
<td>Remote cache</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></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>
</tr>
<tr>
<td></td>
<td></td>
<td></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>Data</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>– Return a data value from the home memory (read miss response)</td>
</tr>
<tr>
<td>Data write back</td>
<td>Remote cache</td>
<td>Home directory</td>
<td>A, Data</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>– Write back a data value for address A (invalidate response)</td>
</tr>
</tbody>
</table>
State Transition Diagram for One Cache Block in Directory Based System

- States identical to snoopy case; transactions very similar
- Transitions caused by read misses, write misses, invalidates, data fetch requests
- Generates read miss & write miss message to home directory
- Write misses that were broadcast on the bus for snooping ⇒ explicit invalidate & data fetch requests
- Note: on a write, a cache block is bigger, so need to read the full cache block
CPU—Cache State Machine

- State machine for CPU requests for each memory block
- Invalid state if in memory

CPU read hit
CPU write hit

CPU read miss:
- Send Read Miss message to home directory

CPU write miss:
- Send Data Write Back message and Write Miss to home directory

Shared (read/only)

Invalid

Fetch/Invalidate
- Send Data Write Back message to home directory

Exclusive (read/write)

CPU Read
- Send Read Miss message

CPU Write:
- Send Write Miss message to home directory

Fetch:
- send Data Write Back message to home directory
State Transition Diagram for Directory

- Same states & structure as the transition diagram for an individual cache
- 2 actions: update of directory state & send messages to satisfy requests
- Tracks all copies of memory block
- Also indicates an action that updates the sharing set, Sharers, as well as sending a message
Directory State Machine

- State machine for **Directory** requests for each **memory block**
- Uncached state if in memory

Uncached

- Data Write Back: Sharers = {} (Write back block)
- Write Miss: Sharers = {P}; send Data Value Reply msg

Exclusive (read/write)

- Write Miss: Sharers = {P}; send Fetch/Invalidate; send Data Value Reply msg to remote cache
- Read miss: Sharers += {P}; send Fetch; send Data Value Reply msg to remote cache

Shared (read only)

- Write Miss: Sharers = {P}; send Invalidate to Sharers; then Sharers = {P}; send Data Value Reply msg
- Read miss: Sharers += {P}; send Data Value Reply msg

Read miss:
Sharers += {P}; send Data Value Reply msg
Example Directory Protocol

- Message sent to directory causes two actions:
  - Update the directory
  - More messages to satisfy request

- Block is in Uncached state: the copy in memory is the current value; only possible requests for that block are:
  - Read miss: requesting processor sent data from memory & requestor made only sharing node; state of block made Shared.
  - Write miss: requesting processor is sent the value & becomes the Sharing node. The block is made Exclusive to indicate that the only valid copy is cached. Sharers indicates the identity of the owner.
Example Directory Protocol

- Message sent to directory causes two actions:
  - Update the directory
  - More messages to satisfy request
- Block is Shared ⇒ the memory value is up-to-date:
  - Read miss: requesting processor is sent back the data from memory & requesting processor is added to the sharing set.
  - Write miss: requesting processor is sent the value. All processors in the set Sharers are sent invalidate messages, & Sharers is set to identity of requesting processor. The state of the block is made Exclusive.
Example Directory Protocol

- Block is Exclusive: current value of the block is held in the cache of the processor identified by the set Sharers (the owner) ⇒ three possible directory requests:
  - Read miss: owner processor sent data fetch message, causing state of block in owner’s cache to transition to Shared and causes owner to send data to directory, where it is written to memory & sent back to requesting processor. Identity of requesting processor is added to set Sharers, which still contains the identity of the processor that was the owner (since it still has a readable copy). State is shared.
Example Directory Protocol

- Block is Exclusive: current value of the block is held in the cache of the processor identified by the set Sharers (the owner) ⇒ three possible directory requests:
  - Data write-back: owner processor is replacing the block and hence must write it back, making memory copy up-to-date (the home directory essentially becomes the owner), the block is now Uncached, and the Sharer set is empty.
Example Directory Protocol

- Block is Exclusive: current value of the block is held in the cache of the processor identified by the set Sharers (the owner) ⇒ three possible directory requests:
  - Write miss: block has a new owner. A message is sent to old owner causing the cache to send the value of the block to the directory from which it is sent to the requesting processor, which becomes the new owner. Sharers is set to identity of new owner, and state of block is made Exclusive.
### Example

<table>
<thead>
<tr>
<th>Processor 1</th>
<th>Processor 2</th>
<th>Interconnect</th>
<th>Directory</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>P2</td>
<td>Bus</td>
<td>Directory</td>
<td>Memory</td>
</tr>
</tbody>
</table>

<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>
<th></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>
<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>
<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>
<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>
<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>
<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>
<td></td>
</tr>
</tbody>
</table>

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

A1 and A2 map to the same cache block

<table>
<thead>
<tr>
<th>Step</th>
<th>Processor 1</th>
<th>Processor 2</th>
<th>Interconnect</th>
<th>Directory</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1: Write 10 to A1</td>
<td>P1: Excl. A1 10</td>
<td>P2: DaRp P1 A1 0</td>
<td></td>
<td>Directory</td>
<td>{P1}</td>
</tr>
<tr>
<td>P1: Read A1</td>
<td>P1: Excl. A1 10</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>
</tr>
<tr>
<td>P2: Write 20 to A1</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>
</tr>
</tbody>
</table>
**Example**

<table>
<thead>
<tr>
<th>Step</th>
<th>Processor 1</th>
<th>Processor 2</th>
<th>Interconnect</th>
<th>Directory</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1: Write 10 to A1</td>
<td>Excl. A1 10</td>
<td>WrMs P1 A1</td>
<td>A1 Ex {P1}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P1: Read A1</td>
<td>Excl. A1 10</td>
<td>DaRp P1 A1 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: Read A1</td>
<td>Shar. A1 10</td>
<td>RdMs P2 A1</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>
</tr>
</tbody>
</table>

**Write Back**

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

A1 and A2 map to the same cache block

<table>
<thead>
<tr>
<th>Step</th>
<th>Processor 1</th>
<th>Processor 2</th>
<th>Interconnect</th>
<th>Directory</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1: Write 10 to A1</td>
<td>Excl. A1 10</td>
<td>DaRp P1 A1 0</td>
<td>WrMs P1 A1</td>
<td>A1 Ex {P1}</td>
<td></td>
</tr>
<tr>
<td>P1: Read A1</td>
<td>Excl. A1 10</td>
<td>DaRp P1 A1 0</td>
<td>WrMs P1 A1</td>
<td>A1 Ex {P1}</td>
<td></td>
</tr>
<tr>
<td>P2: Write 40 to A2</td>
<td>Inv. A1 20</td>
<td>Inv. P1 A1</td>
<td>Inv. P1 A1</td>
<td>A1 Excl. {P2}, 10</td>
<td></td>
</tr>
</tbody>
</table>

P1 = Processor 1, P2 = Processor 2, Dir = Directory, Mem = Memory
### Example

A1 and A2 map to the same cache block  
(but different memory block addresses A1 ≠ A2)

<table>
<thead>
<tr>
<th>Processor 1</th>
<th>Processor 2</th>
<th>Interconnect</th>
<th>Directory</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1: Write 10 to A1</td>
<td>Excl. A1 10</td>
<td>WrMs P1 A1</td>
<td>A1 Ex {P1}</td>
<td>Memor</td>
</tr>
<tr>
<td>P1: Read A1</td>
<td>Excl. A1 10</td>
<td>DaRp P1 A1 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: Read A1</td>
<td>Shar. A1 10</td>
<td>RdMs P2 A1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Shar. A1 10</td>
<td></td>
<td>DaRp P2 A1 10 A1 Shar. P1,P2} 10</td>
<td></td>
</tr>
<tr>
<td>P2: Write 20 to A1</td>
<td>Excl. A1 20</td>
<td>WrMs P2 A1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: Write 40 to A2</td>
<td>Excl. A2 40</td>
<td>DaRp P2 A2 0 A2 Excl. {P2} 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A1 and A2 map to the same cache block  
(but different memory block addresses A1 ≠ A2)
Example Directory Protocol (1st Read)

\[ \text{P1: pA} \]

\[ \text{Id vA } \rightarrow \text{ rd pA} \]
Example Directory Protocol (Read Share)

P1: pA
P2: pA

ld vA -> rd pA
ld vA -> rd pA
Example Directory Protocol (Wr to shared)

R/req

P1: pA

P2: pA

R/req

W/req E

R/req

W/req E

R/req

Inv/_

Inv ACK

Read_to_update pA

Invalidate pA

st vA -> wr pA

E

S

I

S

D

U

Dir ctrl

M

RX/invalidate&reply

reply xD(pA)

Inv ACK

E

S

I

E

S

I

P1

P2
Example Directory Protocol (Wr to Ex)

P1: pA

Inv pA

Write_back pA

Reply xD(pA)

Read_toUpdate pA

W/req E

W/req E

R/req

I

Inv/

S

R/

E

W/_

W/_

R/_.

D

RX/invalidate&reply

Dir ctrl

M

$ P1

$ P2

st vA -> wr pA
A Popular Middle Ground

- Two-level “hierarchy”
- Individual nodes are multiprocessors, connected non-hierarchically
  - e.g. mesh of SMPs
- Coherence across nodes is directory-based
  - directory keeps track of nodes, not individual processors
- Coherence within nodes is snooping or directory
  - orthogonal, but needs a good interface of functionality
- SMP on a chip directory + snoop?
Another MP Issue: Memory Consistency Models

- What is consistency? When must a processor see the new value? e.g., seems that

- P1: \( A = 0; \) P2: \( B = 0; \)
  
  ..... 
  
  A = 1; B = 1;

- L1: if \((B == 0)\) ...
  
  L2: if \((A == 0)\) ...

- Impossible for both if statements L1 & L2 to be true?

- What if write invalidate is delayed & processor continues?
Another MP Issue: Memory Consistency Models

- **Memory consistency** models: what are the rules for such cases?
- **Sequential consistency**: result of any execution is the same as if the accesses of each processor were kept in order and the accesses among different processors were interleaved $\Rightarrow$ assignments before ifs above
- SC: delay all memory accesses until all invalidates done
Memory Consistency Model

- Schemes faster execution to sequential consistency
- Not an issue for most programs; they are synchronized
  
  - A program is synchronized if all access to shared data are ordered by synchronization operations
    
    - `write (x)`
    - `...`
    - `release (s) {unlock}`
    - `...`
    - `acquire (s) {lock}`
    - `...`
    - `read(x)`

- Only those programs willing to be nondeterministic are not synchronized: “data race”: outcome f(proc. speed)

- Several Relaxed Models for Memory Consistency since most programs are synchronized; characterized by their attitude towards: RAR, WAR, RAW, WAW to different addresses
Relaxed Consistency Models: The Basics

- Key idea: allow reads and writes to complete out of order, but to use synchronization operations to enforce ordering, so that a synchronized program behaves as if the processor were sequentially consistent

- By relaxing orderings, may obtain performance advantages

- Also specifies range of legal compiler optimizations on shared data

- Unless synchronization points are clearly defined and programs are synchronized, compiler could not interchange read and write of 2 shared data items because might affect the semantics of the program
Relaxed Consistency Models: The Basics

- 3 major sets of relaxed orderings:
  - $W \rightarrow R$ ordering (all writes completed before next read)
    - Because retains ordering among writes, many programs that operate under sequential consistency operate under this model, without additional synchronization. Called processor consistency
  - $W \rightarrow W$ ordering (all writes completed before next write)
  - $R \rightarrow W$ and $R \rightarrow R$ orderings, a variety of models depending on ordering restrictions and how synchronization operations enforce ordering
- Many complexities in relaxed consistency models; defining precisely what it means for a write to complete; deciding when processors can see values that it has written
Mark Hill observation

• Instead, use speculation to hide latency from strict consistency model

• If processor receives invalidation for memory reference before it is committed, processor uses speculation recovery to back out computation and restart with invalidated memory reference

• 1. Aggressive implementation of sequential consistency or processor consistency gains most of advantage of more relaxed models

• 2. Implementation adds little to implementation cost of speculative processor

• 3. Allows the programmer to reason using the simpler programming models
And in Conclusion ...

- Snooping and Directory Protocols similar; bus makes snooping easier because of broadcast (snooping \(\rightarrow\) uniform memory access)

- Directory has extra data structure to keep track of state of all cache blocks

- Distributing directory:
  - scalable shared address multiprocessor
  - Cache coherent, Non uniform memory access

- MPs are highly effective for multiprogrammed workloads

- MPs proved effective for intensive commercial workloads, such as OLTP (assuming enough I/O to be CPU-limited), DSS applications (where query optimization is critical), and large-scale, web searching applications