Chapter 5

Multiprocessors and Thread-Level Parallelism
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 overheads 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
Cache Coherence

- Processors may see different values through their caches:

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
<th>Cache contents for processor A</th>
<th>Cache contents for processor B</th>
<th>Memory contents for location X</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>Processor A reads X</td>
<td>1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Processor B reads X</td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>Processor A stores 0 into X</td>
<td>0</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>
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
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
Snoopy Coherence Protocols

- **Write invalidate**
  - On write, invalidate all other copies
  - Use bus itself 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 a 1 to X</td>
<td>Invalidation for X</td>
<td>1</td>
<td></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>
<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 hit</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 in the local cache.</td>
</tr>
</tbody>
</table>
Snoopy Coherence Protocols

Centralized Shared-Memory Architectures
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

- 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

Performance of Symmetric Shared-Memory Multiprocessors

Normalized execution time

L3 cache size (MB)

- Idle
- PAL code
- Memory access
- L2/L3 cache access
- Instruction execution
Performance Study: Commercial Workload

Graph showing memory cycles per instruction for different cache sizes (1 MB, 2 MB, 4 MB, 8 MB) with categories: Instruction, Capacity/conflict, Compulsory, False sharing, True sharing.
Performance Study: Commercial Workload

![Bar chart showing memory cycles per instruction across different processor counts.](image)

- Instruction
- Capacity/conflict
- Compulsory
- False sharing
- True sharing
Performance Study: Commercial Workload

Performance of Symmetric Shared-Memory Multiprocessors

- Instruction
- Capacity/conflict
- Compulsory
- False sharing
- True sharing

Misses per 1000 instructions

Block size (bytes):
- 32
- 64
- 128
- 256
DirectoryProtocols

- 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 invalidates 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

Distributed Shared Memory and Directory-Based Coherence
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
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

- Advantage of this scheme: reduces memory traffic

<table>
<thead>
<tr>
<th>Step</th>
<th>P0 Description</th>
<th>P1 Description</th>
<th>P2 Coherence state</th>
<th>Lock at end of step</th>
<th>Bus/directory activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Has lock</td>
<td>Begins spin, testing if lock = 0</td>
<td>Begins spin, testing if lock = 0</td>
<td>Shared</td>
<td>Cache misses for P1 and P2 satisfied in either order. Lock state becomes shared.</td>
</tr>
<tr>
<td>2</td>
<td>Set lock to 0</td>
<td>(Invalidate received)</td>
<td>(Invalidate received)</td>
<td>Exclusive (P0)</td>
<td>Write invalidate of lock variable from P0.</td>
</tr>
<tr>
<td>3</td>
<td>Cache miss</td>
<td>Cache miss</td>
<td>Shared</td>
<td></td>
<td>Bus/directory services P2 cache miss; write-back from P0; state shared.</td>
</tr>
<tr>
<td>4</td>
<td>(Waits while bus/directory busy)</td>
<td>Lock = 0 test succeeds</td>
<td></td>
<td>Shared</td>
<td>Cache miss for P2 satisfied</td>
</tr>
<tr>
<td>5</td>
<td>Lock = 0</td>
<td>Executes swap, gets cache miss</td>
<td></td>
<td>Shared</td>
<td>Cache miss for P1 satisfied</td>
</tr>
<tr>
<td>6</td>
<td>Executes swap, gets cache miss</td>
<td>Completes swap: returns 0 and sets lock = 1</td>
<td></td>
<td>Exclusive (P2)</td>
<td>Bus/directory services P2 cache miss; generates invalidate; lock is exclusive.</td>
</tr>
<tr>
<td>7</td>
<td>Swap completes and returns 1, and sets lock = 1</td>
<td>Enter critical section</td>
<td></td>
<td>Exclusive (P1)</td>
<td>Bus/directory services P1 cache miss; sends invalidate and generates write-back from P2.</td>
</tr>
<tr>
<td>8</td>
<td>Spins, testing if lock = 0</td>
<td></td>
<td></td>
<td>None</td>
<td></td>
</tr>
</tbody>
</table>
Models of Memory Consistency

- 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