Chapter 5

Multiprocessors and Thread-Level Parallelism
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
SMP and DSM

- Shared memory:
  - Communication among threads through a shared address space
  - A memory reference can be made by any processor to any memory location.
Types

INTERPROCESSOR COMMUNICATION IS A CRITICAL PERFORMANCE ISSUE IN DSM MULTIPROCESSORS!

- 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
You want to achieve a speedup of 80 with 100 identical processors. What fraction of the original computation can be sequential?

\[ S(n) = \frac{T(1)}{T(n)} = \frac{T}{f \times T + (1 - f) \times T/n} = \frac{n}{n \cdot f + (1 - f)} \]

\[ \Rightarrow f = \frac{(n/S(n) - 1)}{n - 1} \Rightarrow f = \frac{(100/80) - 1}{100 - 1} = 0.002525 \]

\[ \Rightarrow (1 - f) = 0.997475 \]

The program must be almost 100% parallelizable!
Challenges of Parallel Processing

- How does the speedup vary with n and f?

![Graph showing speedup vs number of processors for different values of f]
Challenges of Parallel Processing: Large Latency of Remote Access

- An application runs on a 32-processor MP that has a 200 ns time to handle a reference to remote memory. Assume that all references (except those involving communication) hit the cache. The processor clock rate is 3.3 GHz and CPI (cycles per instruction) is 0.5 if all instructions hit the local cache. How much faster is the MP if there is no communication vs. the case in which 0.2% of the instructions involve a remote reference?

\[ CPI = 0.5 + 0.2\% \times \text{Remote Request Cost} \]

\[
\text{Remote Request Cost} = \frac{\text{Remote Access Cost}}{\text{Cycle time}} = \frac{200\text{ns}}{1/3.3 \text{ ns}} = 660 \text{ cycles}
\]

**Effective CPI** = \( 0.5 + 0.2\% \times 660 = 1.82 \)

- MP with no remote access is \( 1.82/0.5 = 3.64 \) times faster.
Centralized Shared Memory Architectures

- SMPs: both shared and private data can be cached.
- Shared data provides a mechanism for processors to communicate through reads and writes to shared memory.
- The effect of caching private data on program behavior is the same as that of a uniprocessor because no other processor access these data.
- The value of shared data may be replicated in the multiple caches:
  - + reduction in cache contention
  - - cache coherence!

Centralized Shared-Memory Architectures
Cache Coherence

■ Processors may see different values through their caches:

- Processor A reads X (cache miss)

- Processor A reads X (cache miss)
Cache Coherence

- Processors may see different values through their caches:

- Processor A reads X

![Diagram showing cache coherence](image)
Cache Coherence

- Processors may see different values through their caches:

  - Processor B reads X (cache miss)

  

  ![Diagram showing cache coherence]

  "X=1"

- Processor B reads X (cache miss)
Cache Coherence

- Processors may see different values through their caches:

  - Processor B reads X

  ![Diagram showing cache coherence](image)

  - Proc. A: X=1
  - Proc. B: X=1
  - Centralized Shared-Memory Architectures
Cache Coherence

- Processors may see different values through their caches:

  - Processor A writes 0 into X (assume write-through cache)

- Processor A writes 0 into X (assume write-through cache)
Cache Coherence

- Processors may see different values through their caches:

  Processor B reads X (cache hit) and reads the old value.
Cache Coherence

- If there are no intervening writes by another processor, a read must return the last value written by the same processor.
  \[
  \text{Write}(P,X,X'), \ldots, \text{Write}(P',X,X''), \ldots, X' = \text{Read}(P,X)
  \]

- A read by another processor must return the value of the latest write if the read and write are sufficiently separated in time.
  \[
  \text{Write}(P,X,X'), \ldots, \text{Write}(P',X,X''), \ldots, X' = \text{Read}(P',X)
  \]

- Writes to the same location are serialized.
  \[
  \text{Write}(P,X,X'), \text{Write}(P',X,X''), X'' = \text{Read}(P'',X), X' = \text{Read}(P'',X)
  \]
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
      - Natural for SMP: shared memory
      - Challenging for DSM
  - Snooping
    - Snooping: cache controllers sniff on the broadcast medium to determine if they have a copy of a block being requested.
    - Each core tracks sharing status of each block
Snooping Coherence Protocols

- **Write invalidate**
  - On write, invalidate all other copies (most common)
  - Use bus to serialize
    - Write cannot complete until bus access is obtained
  - The same item can appear in multiple caches
  - Two caches will never have different values for the same block.
  - Most commonly used in MPs

- **Write update (a.k.a. write broadcast)**
  - On write, update all copies
  - Consumes considerable bandwidth
Write Invalidate Cache Coherence

- Invalidate other caches on write:

  - Processor A reads X (cache miss)
Write Invalidate Cache Coherence

- Invalidate other caches on write:

- Processor A reads X

![Diagram of cache coherence](Image)
Write Invalidate Cache Coherence

- Invalidate other copies on write

- Processor B reads X (cache miss)
Write Invalidate Cache Coherence

- Invalidate other copies on write:
  - Processor B reads X (cache is updated)
Write Invalidate Cache Coherence

- Invalidate other copies on write:

- Processor A writes 0 into X (assume write-through cache)
Write Invalidate Cache Coherence

- Invalidate other copies on write:

  - Proc. A
    - $X=0$

  - Proc. B

- Copy of $X$ in B’s cache was invalidated.
Write Invalidate Cache Coherence

- Invalidate other copies on write

Processor A: X = 0

Processor B: X = 0

- Processor B reads X (cache miss)
Write Invalidate Cache Coherence

- Invalidate other copies on write:
  - Processor B reads X

![Diagram showing cache coherence]

Proc. A
X=0

Proc. B
X=0

X=0
Basic Snooping Cache Implementation

- To invalidate: processor acquires the bus and broadcasts access to be invalidated.
- All processors continuously snoop on the bus watching addresses. They invalidate their caches if they have the address.
Snooping Cache Protocol

- States of a block:
  - Invalid
  - Shared: potentially shared with other caches
  - Modified: updated in the private cache. Implies that the block is exclusive.

- Requests to the cache can come from a core or from the bus.
# Snooping 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>
## Snooping 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>
Write-Invalidate Cache Coherence Protocol For a Write-Back Cache

Circles: state of a block in the cache; Bold on transitions: bus actions generated as part of the state transition. Actions in parentheses allowed by local processor without state change.
Snooping Coherence Protocols

- Complications for the basic MSI (Modified, Shared, Invalid) 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 (E) to indicate clean block in only one cache (MESI protocol)
    - Prevents needing to write invalidate on a write
  - Owned (O) state and out-of-date => MOESI protocol
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
Performance

- Coherence influences cache miss rate
  - Coherence misses
    - True sharing misses
      - Write to shared block (transmission of invalidation)
      - Read an invalidated block by another block results in the block being transferred.
    - False sharing misses
      - Single valid bit per block
      - Read an unmodified word in an invalidated block
Performance

- Words $x_1$ and $x_2$ are in the same cache block, which is in the shared (S) state in the caches of P1 and P2. Assume the following sequence of events, identify each miss as a true/false sharing miss, or a hit.

<table>
<thead>
<tr>
<th>Time</th>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Write $x_1$</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>Read $x_2$</td>
</tr>
<tr>
<td>3</td>
<td>Write $x_1$</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>Write $x_2$</td>
</tr>
<tr>
<td>5</td>
<td>Read $x_2$</td>
<td></td>
</tr>
</tbody>
</table>
Performance

- Words x1 and x2 are in the same cache block, which is in the shared (S) state in the caches of P1 and P2. Assume the following sequence of events, identify each miss as a true/false sharing miss, or a hit.

<table>
<thead>
<tr>
<th>Time</th>
<th>P1</th>
<th>P2</th>
<th>Explanation (Block state after action)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Write x1</td>
<td></td>
<td>True sharing miss: x1 needs to be invalidated in P2. =&gt; (P1: E; P2: I)</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>Read x2</td>
<td>False sharing miss: x2 was invalidated by the write of x1 but x1 was not used by P2. =&gt; (P1: S; P2:S)</td>
</tr>
<tr>
<td>3</td>
<td>Write x1</td>
<td></td>
<td>False sharing miss: block containing x1 is marked as S due to P2’s read. =&gt; (P1: E; P2: I)</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>Write x2</td>
<td>False sharing miss for the same reason as above. =&gt; (P1: I; P2: E)</td>
</tr>
<tr>
<td>5</td>
<td>Read x2</td>
<td></td>
<td>True sharing miss: value read by P1 was written by P2. =&gt; (P1: S; P2: S)</td>
</tr>
</tbody>
</table>
Performance Study: Commercial Workload

- 4-processor machine
- OLTP workload
- PAL code: specialized OS-level instructions executed in privileged mode

Diagram:
-Normalized execution time
-L3 cache size (MB)
-Graph showing performance with different L3 cache sizes.
Performance Study: Commercial Workload

- 4-processor machine
- OLTP workload
- PAL code: specialized OS-level instructions executed in privileged mode

Most improvement occurs when cache size goes from 1MB to 2MB even though cache misses are significant for 4MB and 8M caches. Why?
The contributing causes of memory access cycle shift as the cache size increases.

The major sources of L3 memory access cycles are instruction and capacity/conflict misses.

Compulsory, true sharing, false sharing are unaffected by larger L3 caches.
Assume a 2M 2-way associative L3 cache.

Increase in true sharing with more processors increases memory cycles per instruction.
Assume a 2M 2-way associative L3 cache.

Larger block sizes (32 to 256) affects:
- true sharing miss rate (more than 2x)
- compulsory miss rate decreases
- capacity/conflict decreases by a factor of 1.26
- False sharing miss rate nearly double
Directory vs. Snooping Protocols

- Snooping protocols require communication with all caches on every cache miss.
- Coherence traffic on the bus is a scalability problem as the number of processors increases.
- In recent years, the development of multicore processors forced designers to shift to distributed memory to support the bandwidth demands of individual processors.
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 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>

Possible messages sent among nodes.
P = requesting node number; A = requested address; D = data contents.
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. Shares indicate owner identity.

- 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
Directory Protocols

Diagram for an individual cache block

Requests by local processor shown in black
Requests from home directory shown in gray.
Synchronization in MPs

- Basic building blocks:
  - Atomic exchange
    - Swaps register with memory location
  - Test-and-set
    - Sets under condition
    - Test if memory position if 0 and set to 1 if 0.
  - Fetch-and-increment
    - Reads original value from memory and increments it in memory
  - Requires memory read and write in uninterruptable instruction

- Pair of Load linked/Store conditional instructions
  - 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
Synchronization in MPs: Atomic Exchange

- LOCK = 0 => FREE
- LOCK = 1 => UNAVAILABLE
- Lock value in memory
- EXCH R,M

**Processor A**

Register R

1

**Processor B**

Register R

- 

Memory position M

**Processor A issues a EXCH R,M**
Synchronization in MPs: Atomic Exchange

- LOCK = 0 => FREE
- LOCK = 1 => UNAVAILABLE
- Lock value in memory
- EXCH R,M

**Processor A**

- Register R: 1
- Memory position M: 0

**Processor B**

- Register R: -

Processor A issues a EXCH R,M

**Processor A**

- Register R: 0

**Processor B**

- Register R: -

A value of 0 in R indicates successful lock
Synchronization in MPs: Atomic Exchange

- LOCK = 0 => FREE
- LOCK = 1 => UNAVAILABLE
- Lock value in memory
- EXCH R,M

**Processor A**
- Register R: 0
- Memory position M: 1

**Processor B**
- Register R: 1

Processor B executes EXCH R,M

**Processor A**
- Register R: 0
- Memory position M: 1

**Processor B**
- Register R: 1

A value of 1 in R indicates unsuccessful lock
Synchronization in MPs: Load Linked/Store Conditional

- Goal: atomically exchange R4 and content of memory location specified by R1

```assembly
try: MOV   R3,R4        ; move exchange value
    LL       R2,0(R1)   ; load linked
    SC      R3,0(R1)   ; store conditional
    BEQZ R3,try        ; branch store fails
    MOV   R4,R2       ; put load value in R4
```

Mem pointed by R1: A

R1:
R2:
R3:
R4: B
Synchronization in MPs: Load Linked/Store Conditional

- Goal: atomically exchange R4 and content of memory location specified by R1

```
try: MOV R3,R4 ; move exchange value
     LL R2,0(R1) ; load linked
     SC R3,0(R1) ; store conditional
     BEQZ R3,try ; branch store fails
     MOV R4,R2 ; put load value in R4
```

Mem pointed by R1: A
R1:  
R2:  
R3: B  
R4: B  

Synchronization in MPs: Load Linked/Store Conditional

- Goal: atomically exchange R4 and content of memory location specified by R1

try: MOV R3,R4 ; move exchange value
    LL R2,0(R1) ; load linked
    SC R3,0(R1) ; store conditional
    BEQZ R3,try ; branch store fails
    MOV R4,R2 ; put load value in R4

Mem pointed by R1: A
R1: A
R2: A
R3: B
R4: B
Synchronization in MPs: Load Linked/Store Conditional

- Goal: atomically exchange R4 and content of memory location specified by R1

try: MOV R3,R4 ; move exchange value
    LL R2,0(R1) ; load linked
    SC R3,0(R1) ; store conditional
    BEQZ R3,try ; branch store fails
    MOV R4,R2 ; put load value in R4

Mem pointed by R1: B

R1:
R2: A
R3: B
R4: B
Synchronization in MPs: Load Linked/Store Conditional

- Goal: atomically exchange R4 and content of memory location specified by R1

try: MOV R3,R4 ; move exchange value
LL R2,0(R1) ; load linked
SC R3,0(R1) ; store conditional
BEQZ R3,try ; branch store fails
MOV R4,R2 ; put load value in R4

Mem pointed by R1: B
R1:
R2: A
R3: B
R4: A
Implementing Locks

- Spin lock: a processor tries to continuously acquire it in a loop until it succeeds
  - If no coherence:
    ```
    DADDUI R2,R0,#1 ; set R2 to 1
    lockit:
    EXCH R2,0(R1) ; atomic exchange
    BNEZ R2,lockit ; already locked?
    ```
  - If coherence (lock can be cached):
    ```
    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</th>
<th>P1</th>
<th>P2</th>
<th>Coherence state of 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>Shared</td>
<td></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>Shared</td>
<td></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>Exclusive (P2)</td>
<td></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>Exclusive (P1)</td>
<td></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

<table>
<thead>
<tr>
<th>Processor 1:</th>
<th>Processor 2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>A=0</td>
<td>B=0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>A=1</td>
<td>B=1</td>
</tr>
<tr>
<td>if (B==0)</td>
<td>if (A==0)</td>
</tr>
</tbody>
</table>

- 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 (X must complete before Y)
    - 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