SHARED MEMORY SYSTEMS

Mahdi Nazm Bojnordi
Assistant Professor
School of Computing
University of Utah
Overview

- Announcement
  - Final exam: in-class, 10:30AM-12:30PM, Dec. 13th

- This lecture
  - Shared memory systems
  - Cache coherence with write back policy
  - Memory consistency
Recall: Cache Coherence Problem

- Multiple copies of each cache block
  - In main memory and caches
- Multiple copies can get inconsistent when writes happen
  - Solution: propagate writes from one core to others
Cache Coherence

- The key operation is **update/invalidate** sent to all or a subset of the cores
  - Software based management
    - Flush: write all of the dirty blocks to memory
    - Invalidate: make all of the cache blocks invalid
  - Hardware based management
    - Update or invalidate other copies on every write
    - Send data to everyone, or only the ones who have a copy
- Invalidation based protocol is better. **Why?**
Snoopy Protocol

- Relying on a broadcast infrastructure among caches
  - For example shared bus
- Every cache monitors (snoop) the traffic on the shared media to keep the states of the cache block up to date
Simple Snooping Protocol

- Relies on write-through, write no-allocate cache
- Multiple readers are allowed
  - Writes invalidate replicas
- Employs a simple state machine for each cache unit
Simple Snooping State Machine

- Every node updates its one-bit valid flag using a simple finite state machine (FSM)

- Processor actions
  - Load, Store, Evict

- Bus traffic
  - BusRd, BusWr

Diagram:
- States: Valid, Invalid
- Transitions:
  - Load/--, Store/BusWr
  - Load/BusRd
  - Evict/--
  - BusWr/--
  - Store/BusWr

Transaction types:
- Solid arrows: Transaction by local actions
- Dotted arrows: Transaction by bus traffic
Shared Memory Systems

- Multiple threads employ a shared memory system
  - Easy for programmers

- Complex synchronization mechanisms are required
  - Cache coherence
    - All the processors see the same data for a particular memory address as they should have if there were no caches in the system
      - e.g., snoopy protocol with write-through, write no-allocate
        - Inefficient

- Memory consistency
  - All memory instructions appear to execute in the program order
    - e.g., sequential consistency
Snooping with Writeback Policy

- **Problem**: writes are not propagated to memory until eviction
  - Cache data maybe different from main memory

- **Solution**: identify the **owner** of the most recently updated replica
  - Every data may have only one owner at any time
  - Only the owner can update the replica
  - Multiple readers can share the data
    - No one can write without gaining ownership first
Every cache block transitions among three states
- **Invalid**: no replica in the cache
- **Shared**: a read-only copy in the cache
  - Multiple units may have the same copy
- **Modified**: a writable copy of the data in the cache
  - The replica has been updated
  - The cache has the only valid copy of the data block

**Processor actions**
- Load, store, evict

**Bus messages**
- BusRd, BusRdX, BusInv, BusWB, BusReply
MSI Example
MSI Example

invalid -> Load/BusRd -> shared

BusRd/[BusReply]

Load/--

P1

P2

S

I

BUS

Load

BusRd
MSI Example

Diagram showing the states and transitions between invalid and shared states for Load/BusRd and BusRd/[BusReply] transitions. Additionally, Evict/-- and Evict connections are depicted.
MSI Example

Load, Store/

Load/BusRd

BusRd/[BusReply]

Evict/--

BusRdX/[BusReply]

Load/--

invalid

shared

modified

P1

P2

S

I

BUS

Store
MSI Example

Graph:
- Invalid
- Shared
- Modified

Edges:
- Load/BusRd
- BusRdX/[BusReply]
- Evict/--
- Load/--
- Store/BusRdX
- BusRd/BusReply

Nodes:
- P1
- P2
- I
- M
- BUS

Actions:
- Load
- Store
MSI Example
MSI Example

- **invalid**
  - Load, Store/--
  - BusInv, BusRdX/[BusReply]
  - BusRd/[BusReply]

- **modified**
  - Store/BusRdX
  - BusRdX/BusReply
  - Store/BusInv

- **shared**
  - Load/BusRd
  - Evict/--
  - Load/--

- **P1**
  - M

- **P2**
  - I

- **BUS**

- **Store**
Modified, Exclusive, Shared, Invalid

- Also known as Illinois protocol
  - Employed by real processors
  - A cache may have an exclusive copy of the data
  - The exclusive copy may be copied between caches

- Pros
  - No invalidation traffic on write-hits in the E state
  - Lower overheads in sequential applications

- Cons
  - More complex protocol
  - Longer memory latency due to the protocol
Alternatives to Snoopy Protocols

- **Problem:** snooping based protocols are not scalable
  - Shared bus bandwidth is limited
  - Every node broadcasts messages and monitors the bus

- **Solution:** limit the traffic using directory structures
  - Home directory keeps track of sharers of each block
Memory Consistency Model

- Memory operations are reordered to improve performance.
- A memory consistency model for a shared address space specifies constraints on the order in which memory operations must appear to be performed with respect to one another.

Initially $A = \text{flag} = 0$

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A=1;$</td>
<td>while (flag==0);</td>
</tr>
<tr>
<td>$\text{flag} = 1;$</td>
<td>printf (&quot;%d&quot;, A);</td>
</tr>
</tbody>
</table>

What is the expected output of this application?
Memory Consistency

- Recall: load-store queue architecture
  - Check availability of operands
  - Compute the effective address
  - Send the request to memory if no memory hazards

Initially $A = \text{flag} = 0$

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(1) \text{flag} = 1;$</td>
<td>while (flag==0);</td>
</tr>
<tr>
<td>$(2) A=1;$</td>
<td>printf (“%d”, A);</td>
</tr>
<tr>
<td></td>
<td>$0$</td>
</tr>
<tr>
<td></td>
<td>$1$</td>
</tr>
</tbody>
</table>
Dekker’s Algorithm Example

- Critical region with mutually exclusive access
  - Any time, one process is allowed to be in the region
- Reordering in load-store queue may result in failure

Initially $A = B = 0$

```
P1
(2) LOCK_A: A = 1;
(1) if (B != 0) {
    A = 0;
    goto LOCK_A;
}
// ...
A = 0;
```

```
P2
(2) LOCK_B: B = 1;
(1) if (A != 0) {
    B = 0;
    goto LOCK_B;
}
// ...
B = 0;
```
Sequential Consistency

- 1. within a program, program order is preserved
- 2. each instruction executes atomically
- 3. instructions from different threads can be interleaved arbitrarily

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>A</td>
</tr>
<tr>
<td>b</td>
<td>B</td>
</tr>
<tr>
<td>c</td>
<td>C</td>
</tr>
<tr>
<td>d</td>
<td>D</td>
</tr>
</tbody>
</table>

1. abAcBCDdeE
2. aAbBcCdDeE
3. ABCDEabcde

Bad Performance!
Relaxed Consistency Model

- Real processors do not implement sequential consistency
  - Not all instructions need to be executed in program order
  - e.g., a read can bypass earlier writes
- A fence instruction can be used to enforce ordering among memory instructions
  - e.g., Dekker’s algorithm with fence

```
P1
LOCK_A: A = 1;
fence;
if (B != 0) {
  A = 0;
goto LOCK_A;
}

P2
LOCK_B: B = 1;
fence;
if (A != 0) {
  B = 0;
goto LOCK_B;
}
```
Fence Example

P1

{ Region of code with no races }
Fence
Acquire_lock
Fence

{ Racy code }
Fence
Release_lock
Fence

P2

{ Region of code with no races }
Fence
Acquire_lock
Fence

{ Racy code }
Fence
Release_lock
Fence