SHARED MEMORY SYSTEMS

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

- Shared memory systems
  - Inconsistent vs. consistent data
- Cache coherence with write back policy
  - MSI protocol
  - MESI protocol
- Memory consistency
  - Sequential consistency
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
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
Modified-Shared-Invalid Protocol

- 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

---

P1

I

Load

BusRd

BusReply

---

P2

I

Load

BusRd

BusReply

---

invalid

Load/BusRd

shared
MSI Example

invalid ➔ Load/BusRd ➔ shared

BusRd/[BusReply] ➔ Load/--

P1

P2

S

I

BUS

Load

BusRd
MSI Example
MSI Example

- **invalid** to **shared** with **Load/BusRd**
- **BusRdX/[BusReply]** from **invalid** to **shared**
- **Evict/** from **shared** to **modified**
- **Load/** from **shared** to **modified**
- **Store/** from **modified** to **invalid**
- **Load, Store/** from **modified** to **invalid**

Diagram:
- **P1** and **P2**
- **S** and **I**
- **BUS**
MSI Example

Load, Store/

Load

Load/BusRd

BusRd/[BusReply]

BusRdX/[BusReply]

Evict/--

Load/--

invalid

shared

modified

Load, Store/--

P1

P2

I

M

BUS
MSI Example

invalid

shared

modified

Load/BusRd

BusInv, BusRdX/[BusReply]

Evict/--

Load/--

BusRd/[BusReply]

Store/BusRdX

Store/BusInv

Load, Store/--

S

P1

P2

BUS

Store
MSI Example

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

- shared
  - Load/--
  - BusRd/[BusReply]
  - BusRdX/BusReply
  - Store/BusRdX

- modified
  - Store/BusInv

- Load, Store/--
MSI Example

- **Invalid**: Load/BusRd, BusInv, BusRdX/[BusReply], Evict/--
- **Shared**: BusRd/[BusReply], Load/--
- **Modified**: Store/BusRdX, BusRdX/BusReply, BusRd/BusReply, Store/BusInv, Evict/BusWB, BusWB

Diagram:
- Processes: P1, P2
- Memory: M
- Bus: BUS
- States: Invalid, Shared, Modified
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>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) flag = 1;</td>
<td>while (flag==0);</td>
</tr>
<tr>
<td>(2) A=1;</td>
<td>printf (“%d”, A);</td>
</tr>
</tbody>
</table>

0

1
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

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

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

Fence
Acquire_lock
Fence

P2
{                      
Region of code with no races

Fence
Acquire_lock
Fence

{                      
Racy code

Fence
Release_lock
Fence

Fence
Acquire_lock
Fence

Fence
Release_lock
Fence