Cache Coherence

• No lecture or recitation next week
• Don’t forget Lab 7 checkoff

Happy Thanksgiving!
Modern microprocessors usually have 2 to 8 cores where each core has a *private cache* for performance.

Cores can be used cooperatively to speed up an application.

Cores communicate with each other via memory.
Cache Coherence Avoids Stale Data

- Need to provide the illusion of a single shared memory even though multicores have multiple private caches

- Problem:

  Solution: A cache coherence protocol controls cache contents to avoid stale lines
  - e.g., invalidate core 0’s copy of A before letting core 2 write to it

1. LD 0xA → 2
2. ST 3 → 0xA
3. LD 0xA → 2 (stale!)
Maintaining Coherence

- In a *coherent memory* all loads and stores can be placed in a global order
  - multiple copies of an address in various caches can cause this property to be violated

- This property can be ensured if:
  - Only one cache at a time has the write permission for an address
  - No cache can have a stale copy of the data after a write to the address has been performed
Implementing Cache Coherence

- Coherence protocols must enforce two rules:
  - **Write propagation**: Writes eventually become visible to all processors
  - **Write serialization**: Writes to the same location are serialized (all processors see them in the same order)

- How to ensure write propagation?
  - **Write-invalidate protocols**: Invalidate all other cached copies before performing the write
  - **Write-update protocols**: Update all other cached copies after performing the write

- How to ensure write serialization?
  - **Snooping-based protocols**: All caches observe each other’s actions through a shared bus
  - **Directory-based protocols**: A coherence directory tracks contents of private caches and serializes requests
Snooping-Based Coherence [Goodman 1983]

Caches watch (snoop on) bus to keep all processors’ view of memory coherent
Snooping-Based Coherence

- Bus provides serialization point
  - Broadcast, totally ordered
  - Each cache controller “snoops” all bus transactions
  - Controller updates state of cache in response to processor and snoop events and generates bus transactions

- Snoopy protocol (FSM)
  - State-transition diagram
  - Actions

```
State | Tag | Data
-----|-----|-----
      |     |     
      |     |     
...  |    |    
      |     |     
```

Snoop (observed bus transaction)
A Simple Protocol: Valid/Invalid (VI)

- Assume write-through caches

Diagram:

- Valid
  - PrRd / BusRd
  - PrWr / BusWr
- Invalid
  - PrRd / BusRd
  - PrWr / BusWr

Actions:

- Processor Read (PrRd)
- Processor Write (PrWr)
- Bus Read (BusRd)
- Bus Write (BusWr)
Valid/Invalid Example

1. LD 0xA

BusRd 0xA

Core 0

Core 1

Main Memory

Cache

<table>
<thead>
<tr>
<th>Tag</th>
<th>State</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xA</td>
<td>V</td>
<td>2</td>
</tr>
</tbody>
</table>

Cache

<table>
<thead>
<tr>
<th>Tag</th>
<th>State</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Valid/Invalid Example

Additional loads satisfied locally, without BusRd
Valid/Invalid Example

BusWr 0xA, 3

Core 0
1. LD 0xA
2. ST 0xA

Core 1
2. LD 0xA

<table>
<thead>
<tr>
<th>Tag</th>
<th>State</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xA</td>
<td>V</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tag</th>
<th>State</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xA</td>
<td>I</td>
<td>2</td>
</tr>
</tbody>
</table>
Valid/Invalid Example

1. **LD 0xA**
2. **LD 0xA**
3. **ST 0xA**
4. **LD 0xA**

**VI Problems?**

Every write updates main memory
Every write requires broadcast & snoop
Modified/Shared/Invalid (MSI) Protocol

- Each line in each cache maintains MSI state:
  I - cache doesn’t contain the address
  S - cache has the address but so may other caches; hence it can only be read
  M - only this cache has the address; hence it can be read and written
  - any other cache that had this address got invalidated
VI Drawbacks: Every write updates main memory, and every write requires broadcast & snoop

MSI: Allows writeback caches + satisfies writes locally

**Actions**

- Processor Read (PrRd)
- Processor Write (PrWr)
- Bus Read (BusRd)
- Bus Read Exclusive (BusRdX)
- Bus Writeback (BusWB)
MSI Example

```
1  LD 0xA

BusRd 0xA

Core 0

<table>
<thead>
<tr>
<th>Tag</th>
<th>State</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xA</td>
<td>S</td>
<td>2</td>
</tr>
</tbody>
</table>

Core 1

<table>
<thead>
<tr>
<th>Tag</th>
<th>State</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Cache

Main Memory

Core 1

Cache

Core 0

<table>
<thead>
<tr>
<th>Tag</th>
<th>State</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xA</td>
<td>S</td>
<td>2</td>
</tr>
</tbody>
</table>
```

BusRd 0xA

Novermber 21, 2019
MSI Example

1. LD 0xA

Additional loads satisfied locally, without BusRd (like in VI)
MSI Example

Additional loads *and stores* from core 0 satisfied locally, without bus transactions (unlike in VI)
MSI Example

1. LD 0xA
2. LD 0xA
3. ST 0xA
4. ST 0xA
Cache Interventions

- MSI lets caches serve writes without updating memory, so main memory can have stale data
  - Core 0’s cache needs to supply data
  - But main memory may also respond!
- Cache must override response from main memory

**Diagram:**

- **Main Memory**
  - **BusWB 0xA, 3**
  - **BusRdX 0xA**

**Cache 0**

- **Tag:** 0xA
- **State:** I
- **Data:** 3

**Cache 1**

- **Tag:** 0xA
- **State:** M
- **Data:** 10
MSI Example

1. LD 0xA
2. LD 0xA
3. ST 0xA
4. ST 0xA
5. LD 0xA
Observation: Doing read-modify-write sequences on private data is common
  - What’s the problem with MSI?

  2 bus transactions for every read-modify-write of private data.

Solution: E state (exclusive, clean)
  - If no other sharers, a read acquires line in E instead of S
  - Writes silently cause E→M (exclusive, dirty)
**MESI: An Enhanced MSI protocol**

*increased performance for private read-write data*

*Each* cache line has a tag

- **M**: Modified Exclusive
- **E**: Exclusive, unmodified
- **S**: Shared
- **I**: Invalid

---

Each state bit has a tag:

- **PrWr**: Private Write
- **PrRd**: Private Read
- **BusRd**: Bus Read
- **BusRdX**: Bus Read with Invalidate
- **BusWB**: Bus Write with Invalidate

**State Transitions**

- **M** to **E**:
  - BusRd / BusWB
  - PrWr / BusRdX

- **E** to **M**:
  - PrWr / --

- **S** to **I**:
  - PrRd / --
  - BusRdX / --
  - PrRd / BusRd
  - PrRd / BusRdX

- **I** to **S**:
  - PrRd / BusRd
  - PrRd / BusRdX

- **S** to **M**:
  - PrRd / --
  - BusRd / --

- **M** to **S**:
  - PrWr / --
  - PrRd / --
  - BusRdX / --

- **E** to **S**:
  - BusRd / BusWB

- **S** to **E**:
  - PrWr / BusRd

- **I** to **E**:
  - PrRd / BusRd

- **E** to **I**:
  - PrRd / --

- **M** to **I**:
  - PrWr / --

- **I** to **M**:
  - PrRd / --

---

*PrRd* / *BusRd* if other sharers

*PrRd* / *BusRdX* if no other sharers
Directory-Based Coherence

- Route all coherence transactions through a directory
  - Tracks contents of private caches → No broadcasts
  - Serves as ordering point for conflicting requests → Unordered networks
A cache line contains more than one word, and cache coherence is done at line granularity.

| state | line addr | word0 | word1 | ... | wordN |

Suppose $P_1$ writes $word_i$ and $P_2$ writes $word_k$ and both words have the same line address.

What can happen?

The line may be invalidated (ping-pong) many times unnecessarily because addresses are in the same line.
Thank you!

Next lecture: Branch Prediction