Cache Coherence

Mengjia Yan
Computer Science & Artificial Intelligence Lab
M.I.T.

Based on slides from Daniel Sanchez
The Shift to Multicore

The Shift to Multicore

The Shift to Multicore

End of Dennard Scaling


[https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/]
The Shift to Multicore

- Since 2005, improvements in system performance mainly due to increasing cores per chip

See [https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/](https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/) for more detailed data and analysis.
The Shift to Multicore

Since 2005, improvements in system performance mainly due to increasing cores per chip

Why?

[https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/]
The Shift to Multicore

- Since 2005, improvements in system performance mainly due to increasing cores per chip
- Why? Technology scaling

---

End of Dennard Scaling

Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp
[https://www.karlripp.net/2018/02/42-years-of-microprocessor-trend-data/]
The Shift to Multicore

Since 2005, improvements in system performance mainly due to increasing cores per chip

Why? Technology scaling

Limited instruction-level parallelism

End of Dennard Scaling

Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp
[https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/]
Multicore Performance

Cost / perf curve of possible core designs

Performance

Cost (area, energy...)

High-perf, expensive core

Cost / perf curve of possible core designs
Multicore Performance

Cost/perf curve of possible core designs

- High-perf, expensive core
- Moderate perf, efficient core

Cost (area, energy...) vs. Performance
Multicore Performance

Cost/perf curve of possible core designs

- High-perf, expensive core
- Moderate perf, efficient core
- 2 cores

Cost (area, energy...)

Performance
Multicore Performance

Cost/perf curve of possible core designs

- High-perf, expensive core
- Moderate perf, efficient core

Cost (area, energy...) vs. Performance

2 cores

4 cores
Multicore Performance

What factors may limit multicore performance?
Multicore Performance

Cost/perf curve of possible core designs

Cost (area, energy...) vs. Performance

- High-perf, expensive core
- Moderate perf, efficient core

4 cores
2 cores

What factors may limit multicore performance?

- Limited application parallelism
Multicore Performance

Cost/perf curve of possible core designs

High-perf, expensive core

Moderate perf, efficient core

2 cores

4 cores

Performance

Cost (area, energy...)

What factors may limit multicore performance?

- Limited application parallelism
- Memory accesses and inter-core communication
Multicore Performance

What factors may limit multicore performance?

- Limited application parallelism
- Memory accesses and inter-core communication
- Programming complexity
Amdahl’s Law

- Speedup = \frac{\text{time}_{\text{without enhancement}}}{\text{time}_{\text{with enhancement}}}
- Suppose an enhancement speeds up a fraction \( f \) of a task by a factor of \( S \)
Amdahl’s Law

- Speedup = $\frac{\text{time}_{\text{without enhancement}}}{\text{time}_{\text{with enhancement}}}$
- Suppose an enhancement speeds up a fraction $f$ of a task by a factor of $S$

$$\text{time}_{\text{old}}$$

$$(1 - f) \quad f$$

$$\text{time}_{\text{new}}$$

$$(1 - f) \quad \frac{f}{S}$$
Amdahl’s Law

- Speedup = \frac{\text{time}_{\text{without enhancement}}}{\text{time}_{\text{with enhancement}}}
- Suppose an enhancement speeds up a fraction \( f \) of a task by a factor of \( S \)
  \[ \text{time}_{\text{new}} = \text{time}_{\text{old}} \cdot \left( (1-f) + \frac{f}{S} \right) \]
Amdahl’s Law

- Speedup = \frac{\text{time}_{\text{without enhancement}}}{\text{time}_{\text{with enhancement}}}
- Suppose an enhancement speeds up a fraction \( f \) of a task by a factor of \( S \)

\[
\text{time}_{\text{new}} = \text{time}_{\text{old}} \cdot \left( (1-f) + \frac{f}{S} \right)
\]

\[
S_{\text{overall}} = \frac{1}{(1-f) + \frac{f}{S}}
\]
Amdahl’s Law

- Speedup = \( \frac{\text{time}_{\text{without enhancement}}}{\text{time}_{\text{with enhancement}}} \)
- Suppose an enhancement speeds up a fraction \( f \) of a task by a factor of \( S \)

\[
\text{time}_{\text{new}} = \text{time}_{\text{old}} \cdot (1 - f) + \frac{f}{S}
\]

\[
S_{\text{overall}} = \frac{1}{(1 - f) + \frac{f}{S}}
\]

**Corollary:** Make the common case fast
Amdahl’s Law and Parallelism

- Say you write a program that can do 90% of the work in parallel, but the other 10% is sequential.
- What is the maximum speedup you can get by running on a multicore machine?
Amdahl’s Law and Parallelism

- Say you write a program that can do 90% of the work in parallel, but the other 10% is sequential.
- What is the maximum speedup you can get by running on a multicore machine?

\[ S_{\text{overall}} = \frac{1}{(1-f) + \frac{f}{S}} \]
Amdahl’s Law and Parallelism

- Say you write a program that can do 90% of the work in parallel, but the other 10% is sequential.
- What is the maximum speedup you can get by running on a multicore machine?

\[ S_{\text{overall}} = \frac{1}{(1-f) + \frac{f}{S}} \]

\[ f = 0.9, \ S=\infty \Rightarrow S_{\text{overall}} = 10 \]
Amdahl’s Law and Parallelism

• Say you write a program that can do 90% of the work in parallel, but the other 10% is sequential
• What is the maximum speedup you can get by running on a multicore machine?

\[ S_{overall} = \frac{1}{(1-f) + \frac{f}{S}} \]

\[ f = 0.9, \ S=\infty \rightarrow S_{overall} = 10 \]

What \( f \) do you need to use a 1000-core machine well?
Communication Models

• Shared memory:
  – Single address space
  – Implicit communication by reading/writing memory
    • Data
    • Control (semaphores, locks, barriers, …)
  – Low-level programming model: threads
Communication Models

• Shared memory:
  - Single address space
  - Implicit communication by reading/writing memory
    • Data
    • Control (semaphores, locks, barriers, ...)
  - Low-level programming model: threads

• Message passing:
  - Separate address spaces
  - Explicit communication by send/recv messages
    • Data
    • Control (blocking msgs, barriers, ...)
  - Low-level programming model: processes + inter-process communication (e.g., MPI)
Communication Models

• Shared memory:
  – Single address space
  – Implicit communication by reading/writing memory
    • Data
    • Control (semaphores, locks, barriers, ...)
  – Low-level programming model: threads

• Message passing:
  – Separate address spaces
  – Explicit communication by send/rcv messages
    • Data
    • Control (blocking msgs, barriers, ...)
  – Low-level programming model: processes + inter-process communication (e.g., MPI)

• Pros/cons of each model?
Coherence & Consistency

• Shared memory systems:
  – Have multiple private caches for performance reasons
Coherence & Consistency

• Shared memory systems:
  – Have multiple private caches for performance reasons
  – Need to provide the illusion of a single shared memory
Coherence & Consistency

- Shared memory systems:
  - Have multiple private caches for performance reasons
  - Need to provide the illusion of a single shared memory

- Intuition: A read should return the most recently written value
  - What is “most recent”? 
Coherence & Consistency

**Shared memory systems:**
- Have *multiple private caches* for performance reasons
- Need to provide the illusion of a single shared memory

**Intuition:** A read should return the most recently written value
- What is “most recent”? 

**Formally:**
- **Coherence:** What values can a read return?
  - Concerns reads/writes to a single memory location
- **Consistency:** When do writes become visible to reads?
  - Concerns reads/writes to multiple memory locations
Coherence & Consistency

- **Shared memory systems:**
  - Have multiple private caches for performance reasons
  - Need to provide the illusion of a single shared memory

- **Intuition:** A read should return the most recently written value
  - What is “most recent”?

- **Formally:**
  - **Coherence:** What values can a read return?
    - Concerns reads/writes to a single memory location
  - **Consistency:** When do writes become visible to reads?
    - Concerns reads/writes to multiple memory locations
Cache Coherence Avoids Stale Data

Diagram showing a system with cores and caches connected to main memory.
Cache Coherence Avoids Stale Data

1. LD 0xA → 2
Cache Coherence Avoids Stale Data

1. LD 0xA \rightarrow 2
Cache Coherence Avoids Stale Data

1. LD 0xA → 2
2. ST 3 → 0xA
Cache Coherence Avoids Stale Data

1. LD 0xA → 2
2. ST 3 → 0xA
Cache Coherence Avoids Stale Data

1. LD 0xA → 2
2. ST 3 → 0xA
3. LD 0xA → 2 (stale!)
Cache Coherence Avoids Stale Data

- A cache coherence protocol controls cache contents to avoid stale cache lines.

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

Diagram:
- Main Memory
- Cache: 
  - Core 0: $[0xA] = 2$
  - Core 1: $[0xA] = 3$
  - Core 2: $[0xA] = 3$
  - Core 3: $[0xA] = 3$
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)
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
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 track sharing state of cached data and serialize requests to the same address?
  - *Snooping-based protocols*: All caches observe each other’s actions through a shared bus (bus is the serialization point)
  - *Directory-based protocols*: A coherence directory tracks contents of private caches and serializes requests (directory is the serialization point)
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
Snooping-Based Coherence

- Bus provides serialization point
  - Broadcast, totally ordered

- Controller
  - One cache controller for each core “snoops” all bus transactions
  - Controller
    - Responds to requests from core and the bus
    - Changes state of the selected cache block
    - Generates bus transactions to access data or invalidate
Snooping-Based Coherence

- Bus provides serialization point
  - Broadcast, totally ordered

- Controller
  - One cache controller for each core “snoops” all bus transactions
  - Controller
    - Responds to requests from core and the bus
    - Changes state of the selected cache block
    - Generates bus transactions to access data or invalidate

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

![State transition diagram](image.png)
Snooping-Based Coherence

- **Bus provides serialization point**
  - Broadcast, totally **ordered**

- **Controller**
  - One cache controller for each core “snoops” all bus transactions
  - Controller
    - Responds to requests from core and the bus
    - Changes state of the selected cache block
    - Generates bus transactions to access data or invalidate

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

- **Handling writes:**
  - Write-invalidates
  - Write-updates

![Snoopy protocol FSM diagram](image-url)
A Simple Protocol: Valid/Invalid (VI)

- Assume write-through caches
- Transition:
  
  **Actions**
  
<table>
<thead>
<tr>
<th>Actions</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor Read (PrRd)</td>
<td></td>
</tr>
<tr>
<td>Processor Write (PrWr)</td>
<td></td>
</tr>
<tr>
<td>Bus Read (BusRd)</td>
<td></td>
</tr>
<tr>
<td>Bus Write (BusWr)</td>
<td></td>
</tr>
</tbody>
</table>

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

![Diagram of cache and main memory]

### Core 0

<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>

### 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>

### Main Memory
Valid/Invalid Example

1. LD 0xA
Valid/Invalid Example

1. LD 0xA
Valid/Invalid Example

LD 0xA

Main Memory

BusRd 0xA

<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>

Core 0

Core 1
Valid/Invalid Example

Main Memory

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

Core 0

1. LD 0xA

Core 1
Valid/Invalid Example

1. LD 0xA
2. LD 0xA
Valid/Invalid Example

1. LD 0xA

2. LD 0xA
Valid/Invalid Example

1. LD 0xA

2. LD 0xA
Valid/Invalid Example

Additional loads satisfied locally, without BusRd
Valid/Invalid Example

1. LD 0xA
2. LD 0xA
Valid/Invalid Example

1. LD 0xA
2. LD 0xA
3. ST 0xA
Valid/Invalid Example

Main Memory

BusWr 0xA, 3

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>

Core 0

1. LD 0xA

2. LD 0xA

3. ST 0xA
Valid/Invalid Example

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

BusWr 0xA, 3
Valid/Invalid Example

1. LD 0xA
2. LD 0xA
3. ST 0xA
Valid/Invalid Example

BusWr 0xA, 3

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD 0xA</td>
<td>LD 0xA</td>
</tr>
<tr>
<td>ST 0xA</td>
<td></td>
</tr>
</tbody>
</table>
Valid/Invalid Example

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>3</td>
</tr>
</tbody>
</table>

Core 0

1. LD 0xA
2. ST 0xA

Core 1

2. LD 0xA
Valid/Invalid Example

1. LD 0xA
2. LD 0xA
3. ST 0xA
4. LD 0xA
Valid/Invalid Example

1. LD 0xA
2. LD 0xA
3. ST 0xA
4. LD 0xA
Valid/Invalid Example

1. LD 0xA
2. LD 0xA
3. ST 0xA
4. LD 0xA
Valid/Invalid Example

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

VI Problems?
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

- Allows writeback caches + satisfying writes locally

**Actions**

<table>
<thead>
<tr>
<th>Processor Read (PrRd)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor Write (PrWr)</td>
</tr>
<tr>
<td>Bus Read (BusRd)</td>
</tr>
<tr>
<td>Bus Read Exclusive (BusRdX)</td>
</tr>
<tr>
<td>Bus Writeback (BusWB)</td>
</tr>
</tbody>
</table>
MSI Example

Main Memory

Cache

Tag | State | Data
---|---|---

Core 0

Cache

Tag | State | Data
---|---|---

Core 1
MSI Example

1. LD 0xA
MSI Example

1. LD 0xA
MSI Example

1. LD 0xA
MSI Example

1. LD 0xA
MSI Example

1. LD 0x10

2. LD 0x10
MSI Example

1. LD 0xA

2. LD 0xA
MSI Example

1. **LD 0xA**

2. **LD 0xA**
MSI Example

1. LD 0xA
2. LD 0xA

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

Main Memory

<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 0

1. LD 0xA

Core 1

2. LD 0xA
MSI Example

1. LD 0xA
2. LD 0xA
3. ST 0xA
MSI Example

Main Memory

BusRdX 0xA

<table>
<thead>
<tr>
<th>Cache</th>
<th>Core 0</th>
<th>Cache</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tag</td>
<td>State</td>
<td>Data</td>
<td>Tag</td>
</tr>
<tr>
<td>0xA</td>
<td>S</td>
<td>2</td>
<td>0xA</td>
</tr>
</tbody>
</table>

1. LD 0xA
2. LD 0xA
3. ST 0xA
MSI Example

Main Memory

BusRdX 0xA

<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 0

1. LD 0xA
2. ST 0xA

Core 1

2. LD 0xA
MSI Example

Main Memory

BusRdX 0xA

Cache

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

Core 0

1. LD 0xA

Core 1

2. LD 0xA

3. ST 0xA
MSI Example

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

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
MSI Example

1. LD 0xA
2. LD 0xA
3. ST 0xA
4. ST 0xA
MSI Example

1. LD 0xA
2. LD 0xA
3. ST 0xA
4. ST 0xA
MSI Example

Main Memory

BusWB 0xA, 3

Cache

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

Core 0

1. LD 0xA
2. ST 0xA

Cache

<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>

Core 1

3. LD 0xA
4. ST 0xA

BusRdX 0xA
MSI Example

1. LD 0xA
2. LD 0xA
3. ST 0xA
4. ST 0xA
### MSI Example

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LD 0xA</strong></td>
<td><strong>LD 0xA</strong></td>
</tr>
<tr>
<td><strong>ST 0xA</strong></td>
<td><strong>ST 0xA</strong></td>
</tr>
</tbody>
</table>

#### Main Memory

![Memory Diagram]

<table>
<thead>
<tr>
<th>Cache</th>
<th>Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Tag</strong></td>
<td><strong>State</strong></td>
</tr>
<tr>
<td>0xA</td>
<td>I</td>
</tr>
<tr>
<td>0xA</td>
<td>M</td>
</tr>
</tbody>
</table>

#### Bus Transfers

- **BusWB 0xA, 3**
- **BusRdX 0xA**
Cache interventions

- MSI allows caches to serve writes without updating memory, so main memory can have stale data.
Cache interventions

- MSI allows caches to 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 interventions

- MSI allows caches to 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
MSI Example

1. LD 0xA
2. LD 0xA
3. ST 0xA
4. ST 0xA
MSI Example

1. LD 0xA
2. LD 0xA
3. ST 0xA
4. ST 0xA
5. LD 0xA
MSI Example

1. LD 0xA
2. LD 0xA
3. ST 0xA
4. ST 0xA
5. LD 0xA
MSI Example

1. LD 0xA
2. LD 0xA
3. ST 0xA
4. ST 0xA
5. LD 0xA
MSI Example

1. LD 0xA
2. LD 0xA
3. ST 0xA
4. ST 0xA
5. LD 0xA
MSI Example

1. LD 0xA
2. LD 0xA
3. ST 0xA
4. ST 0xA
5. LD 0xA
MSI Optimizations: Exclusive State

- Observation: Doing read-modify-write sequences on private data is common
  - What’s the problem with MSI?
MSI Optimizations: Exclusive State

• Observation: Doing read-modify-write sequences on private data is common
  – What’s the problem with MSI?

• Solution: E state (exclusive, clean)
  – If no other sharers, a read acquires line in E instead of S
  – Writes silently cause $E \rightarrow M$ (exclusive, dirty)
MESI: An Enhanced MSI protocol
increased performance for private read-write data

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

Diagram:

- M: Modified Exclusive
  - PrWr / --
  - PrRd / --
  - BusRd / BusWB
  - PrWr / BusRdX

- S: Shared
  - PrRd / --
  - BusRd / --
  - BusRdX / BusWB

- I: Invalid
  - PrWr / BusRdX
  - BusRdX / --
  - -- / --
MESI: An Enhanced MSI protocol
increased performance for private read-write data

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

PrWr / --
PrRd /--
BusRd / BusWB
PrWr/ BusRdX

PrRd /--
PrWr /--
BusRdX /--
BusRdX/ BusWB
M: Modified Exclusive
E: Exclusive, unmodified
S: Shared
I: Invalid
M: Modified Exclusive
E: Exclusive, unmodified
S: Shared
I: Invalid

PrWr / --
PrRd / --

BusRd / BusWB

PrWr / BusRdX

PrWr / --
PrRd / --

PrRd / BusRd
if no other sharers

PrRd / --
PrWr / --

PrWr / BusRdX

PrRd / BusRd
if other sharers

BusRdX / --
BusRdX / BusWB
MESI: An Enhanced MSI protocol
increased performance for private read-write data

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

PrWr / --
PrRd /--

BusRd / BusWB

PrWr / BusRdX

PrRd / BusRd
if no other sharers

BusRdX / --

PrRd / BusRd
if other sharers

BusRdX / --

BusRdX / BusWB

PrRd / --

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

PrWr / --
PrRd /--

BusRd / BusWB
PrWr/ BusRdX

M

PrWr / --
PrRd /--

BusRd / BusWB
PrWr/ BusRdX

E

BusRdX / --
PrRd / BusRd
if no other sharers

I

BusRdX / --
PrRd / BusRd
if other sharers
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

<table>
<thead>
<tr>
<th>Address tag</th>
<th>State bits</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

PrWr / --
PrRd / --

M

PrWr / --
PrRd / --

E

BusRd / BusWB

S

PrWr / --
PrRd / --

I

BusRdX / --
PrRd / BusRd if other sharers

if no other sharers

BusRdX / --
BusRdX / --

BusRd / --
BusRdX / --
MSI Optimizations: Owner State

• Observation: On M→S transitions, must write back line!
  – What happens with frequent read-write sharing?
  – Can we defer the write after S?
MSI Optimizations: Owner State

• Observation: On \( M \rightarrow S \) transitions, must write back line!
  – What happens with frequent read-write sharing?
  – Can we defer the write after \( S \)?

• Solution: \( O \) state (Owner)
  – \( O = S + \) responsibility to write back
  – On \( M \rightarrow S \) transition, one sharer (typically the one who had the line in \( M \)) retains the line in \( O \) instead of \( S \)
  – On eviction, \( O \) writes back line (or another sharer does \( S \rightarrow O \))
MSI Optimizations: Owner State

- Observation: On M→S transitions, must write back line!
  - What happens with frequent read-write sharing?
  - Can we defer the write after S?

- Solution: O state (Owner)
  - O = S + responsibility to write back
  - On M→S transition, one sharer (typically the one who had the line in M) retains the line in O instead of S
  - On eviction, O writes back line (or another sharer does S→O)

- MSI, MESI, MOSI, MOESI...
MSI Optimizations: Owner State

• Observation: On M→S transitions, must write back line!
  – What happens with frequent read-write sharing?
  – Can we defer the write after S?

• Solution: O state (Owner)
  – O = S + responsibility to write back
  – On M→S transition, one sharer (typically the one who had the line in M) retains the line in O instead of S
  – On eviction, O writes back line (or another sharer does S→O)

• MSI, MESI, MOSI, MOESI...
  – Typically E if private read-write >> shared read-only (common)
  – Typically O only if writebacks are expensive (main mem vs L3)
Split-Transaction and Pipelined Buses

Atomic Transaction Bus

- **Req**
- **Delay**
- **Response**

Simple, but low throughput!
Split-Transaction and Pipelined Buses

Atomic Transaction Bus

Simple, but low throughput!

Split-Transaction Bus

Req1, Req2, Req3

Resp1, Resp3
Split-Transaction and Pipelined Buses

Atomic Transaction Bus

Req

Delay

Response

Simple, but low throughput!

Split-Transaction Bus

Req1  Req2  Req3

Resp1

Resp3

• Supports multiple simultaneous transactions
Split-Transaction and Pipelined Buses

Atomic Transaction Bus

Simple, but low throughput!

Split-Transaction Bus

- Supports multiple simultaneous transactions
  - Higher throughput
  - Responses may arrive out of order
Split-Transaction and Pipelined Buses

Atomic Transaction Bus

- Supports multiple simultaneous transactions
  - Higher throughput
  - Responses may arrive out of order

- Often implemented as multiple buses (req+resp)

Split-Transaction Bus

- Simple, but low throughput!

- Supports multiple simultaneous transactions
  - Higher throughput
  - Responses may arrive out of order

- Often implemented as multiple buses (req+resp)
Non-Atomicity $\rightarrow$ Transient States

- Protocol must handle lack of atomicity
- Two types of states
  - Stable (e.g. MSI)
  - Transient
- Split + race transitions
- More complex

<table>
<thead>
<tr>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bus Request (BusReq)</td>
</tr>
<tr>
<td>Bus Grant (BusGnt)</td>
</tr>
</tbody>
</table>

PrRd / --  PrWr / --

M

BusRd / BusWB

S

BusRdX / BusWB

I

PrRd / --  BusRd / --

BusRdX / --
Non-Atomicity $\Rightarrow$ Transient States

- Protocol must handle lack of atomicity
- Two types of states
  - Stable (e.g. MSI)
  - Transient
- Split + race transitions
- More complex

**Actions**

<table>
<thead>
<tr>
<th>Bus Request</th>
<th>(BusReq)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bus Grant</td>
<td>(BusGnt)</td>
</tr>
</tbody>
</table>
Non-Atomicity → Transient States

- Protocol must handle lack of atomicity
- Two types of states
  - Stable (e.g. MSI)
  - Transient
- Split + race transitions
- More complex

**Actions**

<table>
<thead>
<tr>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bus Request</td>
</tr>
<tr>
<td>(BusReq)</td>
</tr>
<tr>
<td>Bus Grant</td>
</tr>
<tr>
<td>(BusGnt)</td>
</tr>
</tbody>
</table>
Non-Atomicity → Transient States

- Protocol must handle lack of atomicity
- Two types of states
  - Stable (e.g. MSI)
  - Transient
- Split + race transitions
- More complex

### Actions

<table>
<thead>
<tr>
<th>Actions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bus Request (BusReq)</td>
<td></td>
</tr>
<tr>
<td>Bus Grant (BusGnt)</td>
<td></td>
</tr>
</tbody>
</table>
Non-Atomicity → Transient States

- Protocol must handle lack of atomicity
- Two types of states
  - Stable (e.g. MSI)
  - Transient
- Split + race transitions
- More complex

### Actions

<table>
<thead>
<tr>
<th>Actions</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Bus Request (BusReq)</td>
<td>PrWr / BusReq</td>
</tr>
<tr>
<td>Bus Grant (BusGnt)</td>
<td>PrRd / BusReq</td>
</tr>
</tbody>
</table>
Scaling Cache Coherence

- Can implement ordered interconnects that scale better than buses...

Starfire E10000 (drawn with only eight processors for clarity). A coherence request is *unicast* up to the root, where it is serialized, before being *broadcast* down to all processors.
Scaling Cache Coherence

- Can implement ordered interconnects that scale better than buses...

Starfire E10000 (drawn with only eight processors for clarity). A coherence request is *unicast* up to the root, where it is serialized, before being *broadcast* down to all processors

- ... but broadcast is fundamentally unscalable
  - Bandwidth, energy of transactions with 100s of cache snoops?
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

(more on next lecture)
A cache block contains more than one word and cache coherence is done at the block-level and not word-level
A cache block contains more than one word and cache coherence is done at the block-level and not word-level.

Suppose $P_1$ writes $\text{word}_i$ and $P_2$ writes $\text{word}_k$ and both words have the same block address.

What can happen?
Coherence and False Sharing

Performance Issue #1

A cache block contains more than one word and cache coherence is done at the block-level and not word-level.

Suppose $P_1$ writes $\text{word}_i$ and $P_2$ writes $\text{word}_k$ and both words have the same block address.

What can happen? The block may be invalidated (ping-pong) many times unnecessarily because addresses are in the same block.
A cache block contains more than one word and cache coherence is done at the block-level and not word-level.

Suppose $P_1$ writes $\text{word}_i$ and $P_2$ writes $\text{word}_k$ and both words have the same block address.

**What can happen?** The block may be invalidated (ping-pong) many times unnecessarily because addresses are in the same block.

**How to address this problem?**
Coherence and Synchronization
Performance Issue #2

Processor 1
R ← 1
L: swap (mutex), R;
if <R> then goto L;
critical section>
M[mutex] ← 0;

cache

Processor 2
R ← 1
L: swap (mutex), R;
if <R> then goto L;
critical section>
M[mutex] ← 0;

cache

Processor 3
R ← 1
L: swap (mutex), R;
if <R> then goto L;
critical section>
M[mutex] ← 0;

mutex=1

CPU-Memory Bus
Cache coherence protocols will cause `mutex` to ping-pong between P1’s and P2’s caches.
Cache coherence protocols will cause \textit{mutex} to \textit{ping-pong} between P1’s and P2’s caches.

Ping-ponging can be reduced by first reading the \textit{mutex} location (\textit{non-atomically}) and executing a swap only if it is found to be zero (test\&test\&set).
Coherence and Bus Occupancy
Performance Issue #3

• In general, an *atomic read-modify-write* instruction requires two memory (bus) operations without intervening memory operations by other processors.
Coherence and Bus Occupancy
Performance Issue #3

• In general, an *atomic read-modify-write* instruction requires two memory (bus) operations without intervening memory operations by other processors

• In a multiprocessor setting, bus needs to be locked for the entire duration of the atomic read and write operation
  ⇒ expensive for simple buses
  ⇒ *very expensive* for split-transaction buses
Coherence and Bus Occupancy
Performance Issue #3

• In general, an **atomic read-modify-write** instruction requires two memory (bus) operations without intervening memory operations by other processors.

• In a multiprocessor setting, bus needs to be locked for the entire duration of the atomic read and write operation:
  - \( \Rightarrow \) expensive for simple buses
  - \( \Rightarrow \) very expensive for split-transaction buses

• modern processors use
  - *load-reserve*
  - *store-conditional*
Load-reserve & Store-conditional

Special register(s) to hold reservation flag and address, and the outcome of store-conditional

Load-reserve R, (a):
<flag, adr> ← <1, a>; R ← M[a];

Store-conditional (a), R:
if <flag, adr> == <1, a>
then cancel other procs’ reservation on a;
M[a] ← <R>;
status ← succeed;
else status ← fail;

If the snooper sees a store transaction to the address in the reserve register, the reserve bit is set to 0
- Several processors may reserve ‘a’ simultaneously
- These instructions are like ordinary loads and stores with respect to the bus traffic
The total number of memory (bus) transactions is not necessarily reduced, but splitting an atomic instruction into load-reserve & store-conditional:

- *increases bus utilization* (and reduces processor stall time), especially in split-transaction buses

- *reduces cache ping-pong effect* because processors trying to acquire a mutex do not have to perform stores each time
Thank you!

Next lecture: Directory-based Cache Coherence