ECE 454
Computer Systems Programming
Performance Implications of Parallel Architectures

Ashvin Goel
ECE Dept, University of Toronto
http://www.eecg.toronto.edu/~ashvin
We know that we need parallelization
But will more parallelization always yield better performance?

We will study the reasons for this behavior
Topics

• Cache coherence
• Performance of memory operations
• Implications for software design
Cache Coherence
Modern Shared Memory Parallel Architectures

- Provide several processing elements (cores or processors)
- Provide shared memory
  - Any processor can directly reference any memory location
  - Communication occurs implicitly through loads and stores
- Cores have private caches to improve performance

![Diagram of processors, caches, and memory]

48 core AMD Opteron
Cache Coherence Problem

• With multiple cores, data is cached in multiple locations, so how do you ensure consistency?
Example 1: Coherence Problem

Thread A:
- Store X = 3
- Cache: X = 2

Thread B:
- Load X
- Cache: X = ?

Shared Memory (X = 2)
Example 2: Coherence Problem

Thread A:

Thread B: Load X

Shared Memory (X=2)
Example 2: Coherence Problem

Thread A:

Thread B:

Load X

Read

Shared Memory (X=2)
Example 2: Coherence Problem

Thread A:

Thread B:

Load X

Shared Memory (X=2)
Example 2: Coherence Problem

Thread A: Store X=3

Thread B:

Shared Memory (X=2)
Example 2: Coherence Problem

Thread A: Store X=3
Thread B: Inconsistency!

Inconsistency!

Shared Memory (X=2)
Cache Coherence Definition

• **Consistency (or coherence):** the behavior of the system is equivalent to there being only a single copy of the data except for the performance benefit of the cache. [Gray and Cheriton 83]

• Ensuring consistency requires hardware support
  • Called *coherence protocol*
MSI Coherence Protocol

- Add three (exclusive) states to each cache line:
  - Modified – core has written to the cache line, and the cache line is inconsistent with primary storage (RAM or last-level cache)
    - Cache line is not shared with other cores
  - Shared – core has read from the cache line, and the cache line is consistent with primary storage
    - Cache line can be shared with other cores
  - Invalid
MSI Coherence Protocol

Thread A:

- Processor
- Cache
- State: Invalid
- Tag: -
- Data: -

Thread B:

- Processor
- Cache
- State: Invalid
- Tag: -
- Data: -

Shared Memory (X=2)
MSI Coherence Protocol

Thread A:

Load X

Thread B:

Read

Shared Memory (X=2)

Processor

Cache

State Tag Data
Invalid - -

Processor

Cache

State Tag Data
Invalid - -
MSI Coherence Protocol

Thread A:

Thread B:

Load X

Processor

Cache

State  Tag  Data
Invalid  -  -

Shared Memory (X=2)

Processor

Cache

State  Tag  Data
Shared  X  2

Read  Fill
**MSI Coherence Protocol**

**Thread A:** Store X=3

**Thread B:** Load X

Invalidation

Shared Memory (X=2)

invalidates all other copies
MSI Coherence Protocol

Thread A: Store X=3

Thread B: Load X

Invalidates all other copies
MSI Coherence Protocol

Thread A: Store X=3

- Processor
  - Cache
    - State: Modified
    - Tag: X
    - Data: 3

Invalidation

Shared Memory (X,2 (stale))

Thread B: Load X

- Processor
  - Cache
    - State: Invalid
    - Tag: -
    - Data: -

Invalidation
Problem with MSI

- If a core reads a value that is not cached on any core and then writes to the value, then two cache coherence requests are generated.
  - A request to read the value (required)
  - A request to write the value (unnecessary invalidation request sent because the MSI protocol doesn’t know that no one else has a copy)

MESI (aka Illinois) protocol: Adds Exclusive state to MSI

- **Exclusive**: core has read from the cache line, and the cache line is consistent with primary storage
  - Cache line is not shared by other cores
  - Write to Exclusive state does **not** generate invalidation request
MESI Details: Writing

- An attempt to write to a block that is in Invalid state is called a write miss
  - Must cache the block in Exclusive state before writing to it
  - Generates a read-exclusive (or read for ownership) request
    - A read request with intent to write to the memory address
    - If other caches have copy of data, they send it, invalidate their copy
    - Completes when there are no more valid copies
  - Can then perform the write and enter the modified state
    - This step doesn’t require invalidation request
Example 1: MESI Coherence

Thread A:

Thread B:

Shared Memory (X=2)
Example 1: MESI Coherence

Thread A: Load X

Thread B:

Shared Memory (X=2)
Example 1: Maudi Coherence

Thread A:

Load X

Processor
Cache
State: Invalid
Tag: -
Data: -

Thread B:

Read

Shared Memory (X=2)
Example 1: MESI Coherence

Thread A:
- Load X
- Processor
  - Cache
    - State: Excl
    - Tag: X
    - Data: 2

Thread B:
- Load X
- Processor
  - Cache
    - State: Invalid
    - Tag: -
    - Data: -

Shared Memory (X=2)
Example 1: Mesi Coherence

Thread A: Load X

Thread B: Load X

Shared Memory (X=2)
Example 1: Muli-Processor System Interface (MPSI) Coherence

Thread A:
- Load X
- Processor
  - Cache
    - State: Excl
    - Tag: X
    - Data: 2

Thread B:
- Load X
- Processor
  - Cache
    - State: Invalid
    - Tag: -
    - Data: -

Read

Shared Memory (X=2)
Example 1: MESI Coherence

Thread A:

Thread B:

Load X

Load X

Processor

Processor

Cache

Cache

State
Share
Share

Tag
X
X

Data
2
2

Notify Shared
Fill
Read

Shared Memory (X=2)
Example 2: MESI Coherence

Thread A:
Processor
Cache
State: Invalid
Tag: -
Data: -

Thread B:
Processor
Cache
State: Invalid
Tag: -
Data: -

Shared Memory (X=2)
Example 2: Mesi Coherence

Thread A:

Thread B:

Load X

Read

Shared Memory (X=2)
Example 2: MESI Coherence

Thread A:

Thread B:

Load X

Shared Memory (X=2)
Example 2: MESI Coherence

Thread A: Store X=3

Thread B: Load X

Processor
Cache
State  Tag  Data
Invalid - -

Processor
Cache
State  Tag  Data
Excl.  X  2

Read-Exclusive

Shared Memory (X=2)

read-exclusive invalidates all other copies
Example 2: MESI Coherence

Thread A: Store \( X=3 \)

Thread B: Load \( X \)

### Processor

- **Cache**
  - **State:** Invalid
  - **Tag:** -
  - **Data:** -

### Read-Exclusive

### Shared Memory (\( X=2 \))

- read-exclusive invalidates all other copies
Example 2: MESI Coherence

Thread A: Store X=3

Thread B: Load X

the state ‘dirty’ implies exclusiveness
Example 3: MESI Coherence

Thread A:

- Processor
  - Cache
    - State
    - Tag
    - Data

Thread B:

- Processor
  - Cache
    - State
    - Tag
    - Data

Shared Memory (X=2)
Example 3: MESI Coherence

Thread A: Store X=5

Thread B:

Shared Memory (X=2)
Example 3: MESI Coherence

Thread A: Store X=5

Thread B: Read-Exclusive

Shared Memory (X=2)
Example 3: MESI Coherence

Thread A: Store X=5

Thread B: Read-Exclusive

Shared Memory (X=out-of-date)
Example 3: MESI Coherence

Thread A: Store X=5
Thread B: Load X

Shared Memory (X, 2 (out of date))
Example 3: MESI Coherence

Thread A: Store X=5

Thread B: Load X

Shared Memory (X=out-of-date)
Example 3: MESI Coherence

Thread A:

Store X=5

Thread B:

Load X

- Cache

  - Processor

    - State
      - Dirty
      - X

    - Tag
      - 5

    - Data

  - Shared Memory (X=out-of-date)

  - Read request

  - Read
Example 3: MESI Coherence

Thread A: Store X=5

Thread B: Load X

Store X=5

Load X

Read request

Update

Read

Shared Memory (X=5)
Example 3: MESI Coherence

Thread A: Store X=5

Thread B: Load X

Shared Memory (X=5)
### Mesi Permitted States and Transitions

**Local Event** | **Initial State** | **Local** | **Message** | **Remote**
--- | --- | --- | --- | ---
Read hit | S, E, M |  |  |  
Read miss | I | I → (S, E) | READ | (S, E) → S M → S + WB
Write hit | S E, M | S → M E → M | INVALIDATE | S → I  
Write miss | I | I → M | READEX | (S,E) → I M → I + WB
Performance of Memory Operations
Local Caches and Memory Latencies

- Cost of accessing memory
  - Best case
    - Data is cached locally: L1 < 10 cycles (remember this)
  - Worst case
    - Data is accessed from DRAM: 136 – 355 cycles (remember this)

<table>
<thead>
<tr>
<th></th>
<th>Opteron</th>
<th>Xeon</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>L2</td>
<td>15</td>
<td>11</td>
</tr>
<tr>
<td>LLC</td>
<td>40</td>
<td>44</td>
</tr>
<tr>
<td>RAM</td>
<td>136</td>
<td>355</td>
</tr>
</tbody>
</table>
Interconnect Between Sockets

Cross-sockets communication can be 2-hops
Latency of Remote Access: Read (cycles)

<table>
<thead>
<tr>
<th>System</th>
<th>Opteron</th>
<th>Xeon</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Hops</td>
<td>one hop</td>
</tr>
<tr>
<td>State</td>
<td>same die</td>
<td>81</td>
</tr>
<tr>
<td>Modified</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>die</td>
<td>172</td>
</tr>
<tr>
<td>Exclusive</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>die</td>
<td>175</td>
</tr>
<tr>
<td>Shared</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>die</td>
<td>176</td>
</tr>
</tbody>
</table>

- **State** is the MESI state of a cache line in a remote cache
  - Local cache line state is invalid

- **Cross-socket communication is expensive!**
  - Xeon: cross-socket latency is 4-7.5 larger than within socket
  - Opteron: cross-socket latency even larger than RAM
Latency of Remote Access: Write (cycles)

<table>
<thead>
<tr>
<th>System</th>
<th>Opteron</th>
<th>Xeon</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>same die</td>
<td>one hop</td>
</tr>
<tr>
<td>State</td>
<td>Hops</td>
<td>two hops</td>
</tr>
<tr>
<td>Modified</td>
<td>83</td>
<td>191</td>
</tr>
<tr>
<td>Exclusive</td>
<td>83</td>
<td>191</td>
</tr>
<tr>
<td>Shared</td>
<td>246</td>
<td>286</td>
</tr>
</tbody>
</table>

- **State** is the MESI state of a cache line in a remote cache
  - Local cache line state is invalid

- **Cross-socket communication is expensive!**
  - Comparable or more expensive than DRAM accesses
Implications for Software Design
False Sharing

Thread A: Store X=5

Thread B: Read-Exclusive

X and Y are on the same cache line
False Sharing

Thread A: Store X=5

Thread B:

Shared Memory (X,Y=out-of-date)
False Sharing

Thread A:
Store X=5

Processor
Cache
State: Dirty
Tag: X,Y
Data: 5,0

Thread B:
Store Y=2

Processor
Cache
State: -,-
Tag: -,-
Data: -,-

Invalidation

Read-Exclusive

Shared Memory (X,Y=out-of-date)
False Sharing

Thread A:
- Store X=5

Thread B:
- Store Y=2

Shared Memory (X=5, Y=0)
False Sharing

Thread A:
Store X=5

Thread B:
Store Y=2

Shared Memory (X,Y=out-of-date)
False Sharing

Thread A: Store X=5

Thread B: Store Y=2

Shared Memory (X,Y=out-of-date)
False Sharing

Thread A:
while(1)
Store X=5

Thread B:
while(1)
Store Y=2

X,Y cache line will ping-pong back & forth
False Sharing Summary

- **False sharing**
  - Threads on different cores access unrelated objects
  - Objects are located in same cache block
  - Block will ping-pong between caches on different cores

- **Avoid false sharing by careful data arrangement**
  - Ensure that unrelated elements are mapped to separate blocks
  - E.g., insert padding (unused data) between shared items
  - Whose responsibility is it? compiler? malloc? programmer?
Implications for Programmers

- Cache coherence is expensive (more than you thought)
  - Avoid unnecessary sharing (e.g., false sharing)

- Crossing processors/sockets is a killer
  - Can be slower than running the same program on single core!
  - Pthreads provides CPU affinity mask
    - Pin cooperative threads on cores within the same die

- Next we will see other implications of modern architectures on software design