Parallelism II:
MultiProcessors
Key Points

• What is a CMP?
• Why have we started building them?
• Why are they hard to use?
• What is deadlock?
• What is cache coherence?
• What is cache consistency?
Multiprocessors

- Specifically, shared-memory multiprocessors have been around for a long time.
- Originally, put several processors in a box and provide them access to a single, shared memory.
- Expensive and mildly exotic.
  - Big servers
  - Sophisticated users/data-center applications
Chip Multiprocessors (CMPS)

- Multiple processors on one die
- An easy way to spend xtrs
- Now common place
  - Laptops/desktops/game consoles/etc.
  - Less sophisticated users, all kinds of applications.
Why didn’t we get here sooner

• Doubling performance with frequency increases power by 8x
• Doubling performance with multiple cores increases power by 2x
• No brainer?!? -- Only a good deal if
  • Power matters -- for a long time it didn’t
  • and you actually get twice the performance
The Trouble With CMPs

- Amdahl’s law
  - $\text{Stot} = \frac{1}{(x/S + (1-x))}$
- In order to double performance with a 2-way CMP
  - $S = 2$
  - $x = 1$
  - Usually, neither is achievable
Threads are Hard to Find

- To exploit CMP parallelism you need multiple processes or multiple “threads”
- Processes
  - Separate programs actually running (not sitting idle) on your computer at the same time.
  - Common in servers
  - Much less common in desktop/laptops
- Threads
  - Independent portions of your program that can run in parallel
  - Most programs are not multi-threaded.
- We will refer to these collectively as “threads”
  - A typical user system might have 1-8 actively running threads.
  - Servers can have more if needed (the sysadmins will hopefully configure it that way)
Architectural Support for Multiprocessors

- Allowing multiple processors in the same system has a large impact on the memory system.
  - How should processors see changes to memory that other processors make?
  - How do we implement locks?
Shared Memory

- Multiple processors connected to a single, shared pool of DRAM
- If you don’t care about performance, this is relatively easy... but what about caches?
Uni-processor Caches

- Caches mean multiple copies of the same value
- In uniprocessors this is not a big problem
  - From the (single) processor’s perspective, the “freshest” version is always visible.
  - There is no way for the processor to circumvent the cache to see DRAM’s copy.
Caches, Caches, Everywhere

- With multiple caches, there can be many copies.
- No one processor can see them all.
- Which one has the “right” value?
Caches, Caches, Everywhere

• With multiple caches, there can be many copies
• No one processor can see them all.
• Which one has the “right” value?
Caches, Caches, Everywhere

- With multiple caches, there can be many copies
- No one processor can see them all.
- Which one has the “right” value?
With multiple caches, there can be many copies.
No one processor can see them all.
Which one has the “right” value?

Diagram:
- Store 0x1000
- Local caches: 0x1000: A
- Bus/arbiter
- Main Memory: 0x1000: B
Caches, Caches, Everywhere

- With multiple caches, there can be many copies
- No one processor can see them all.
- Which one has the “right” value?
Caches, Caches, Everywhere

- With multiple caches, there can be many copies.
- No one processor can see them all.
- Which one has the “right” value?
Caches, Caches, Everywhere

- With multiple caches, there can be many copies
- No one processor can see them all.
- Which one has the “right” value?
Caches, Caches, Everywhere

- With multiple caches, there can be many copies
- No one processor can see them all.
- Which one has the “right” value?

![Diagram showing cache management and bus/arbiter interactions.](image)
Keeping Caches Synchronized

• We must make sure that all copies of a value in the system are up to date
  • We can update them
  • Or we can “invalidate” (i.e., destroy) them

• There should always be exactly one current value for an address
  • All processors should agree on what it is.

• We will enforce this by enforcing a total order on all load and store operations to an address and making sure that all processors observe the same ordering.

• This is called “Cache Coherence”
The Basics of Cache Coherence

- Every cache line (in each cache) is in one of 3 states
  - Shared -- There are multiple copies but they are all the same. Only reading is allowed
  - Owned -- This is the only cached copy of this data. Reading and write are allowed
  - Invalid -- This cache line does not contain valid data.
- There can be multiple sharers, but only one owner.
Simple Cache Coherence

- There is one copy of the state machine for each line in each coherent cache.
Caches, Caches, Everywhere

Store 0x1000

Local caches

Exclusive 0x1000: A

Bus/arbiter

Main Memory

0x1000: Z
Caches, Caches, Everywhere

Store 0x1000

Read 0x1000

Local caches

Shared 0x1000:A

Shared 0x1000:A

Bus/arbiter

Main Memory

0x1000: A
Caches, Caches, Everywhere

Store 0x1000

Read 0x1000

Store 0x1000

Local caches

invalid 0x1000: A

invalid 0x1000: A

Owned 0x1000: C

Bus/arbiter

Main Memory

0x1000: A
Coherence in Action

```
a = 0
Thread 1
while(1) {
    a++;
}
```

```
Thread 2
while(1) {
    print(a);
}
```

possible?

Sample outputs

```
1 1 1
2 1 2
3 1 5
4 1 8
5 100 3
6 100 5
7 100 2
8 100 4
```
Coherence in Action

\[
a = 0
\]

Thread 1
while(1) {
    a++;
}

Thread 2
while(1) {
    a++;
    print(a);
}

Sample outputs
1  1  1
2  1  2
3  1  5
4  1  8
5  100 3
6  100 5
7  100 2
8  100 4

possible? yes
Coherence in Action

\[ a = 0 \]

Thread 1
while(1) {
    a++;
}

Thread 2
while(1) {
    print(a);
    \[ a = 0 \]
}

Sample outputs

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>100</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>100</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>100</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>100</td>
<td>4</td>
</tr>
</tbody>
</table>

possible? yes yes
Coherence in Action

```
a = 0

Thread 1
while(1) {
    a++;
}
```

```
Thread 2
while(1) {
    print(a);
}
```

Sample outputs

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>100</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>100</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>100</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>100</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

possible? yes yes no
Live demo.
Coherence In The Real World

- Real coherence have more states
  - e.g. “Exclusive” -- I have the only copy, but it’s not modified
- Often don’t bother updating DRAM, just forward data from the current owner.
- If you want to learn more, take 240b
Cache Consistency

• If two operations occur in an order in one thread, we would like other threads to see the changes occur in the same order.
• Example:

  Thread 0  Thread 1
  A = 10;   while(!A_is_valid);
  A_is_valid = true;  B = A;

• We want B to end up with the value 10
• Coherence does *not* give us this assurance, since the state machine only applies to a single cache line
• This is called “cache consistency” or “the consistency model”
Simple Consistency

- The simplest consistency model is called “sequential consistency”
- In which all stores are immediately visible everywhere.

Thread 0
A = 10;
A_is_valid = true;

Thread 1
while(!A_is_valid);
B = A;

- If thread 1 sees the write to A_is_valid, it will also see the write to A.
What about this?

```
while(1) {
    a++;
    b++;
}
```

```
while(1) {
    print(a, b);
}
```

\[ a = b = 0 \]

**Thread 1**

**Thread 2**

Sample outputs

1 1
2 2
3 3
4 4
5 5
6 6
7 7
8 8

1 1
2 2
3 3
4 4
5 5
6 6
7 7
8 8

possible under sequential consistency?
What about this?

```
while(1) {
    a++;
    b++;
}
```

da = b = 0

Thread 1
while(1) {
    a++;
    b++;
}

Thread 2
while(1) {
    print(a, b);
}

possible under sequential consistency?

Sample outputs

```
1 1
2 2
3 3
4 4
5 5
6 6
7 7
8 8
1 1
2 2
3 1000
4 1000
5 5
6 6
7 7
8 8
```
What about this?

```
while(1) { a++; b++;}
```

```
while(1) {
    print(a, b);
}
```

```
a = b = 0
Thread 1
while(1) {
    a++;
    b++;
}
```

```
Thread 2
while(1) {
    print(a, b);
}
```

Is it possible under sequential consistency?

<table>
<thead>
<tr>
<th>Sample outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1</td>
</tr>
<tr>
<td>2 2</td>
</tr>
<tr>
<td>3 3</td>
</tr>
<tr>
<td>4 4</td>
</tr>
<tr>
<td>5 5</td>
</tr>
<tr>
<td>6 6</td>
</tr>
<tr>
<td>7 7</td>
</tr>
<tr>
<td>8 8</td>
</tr>
<tr>
<td>1 1</td>
</tr>
<tr>
<td>2 2</td>
</tr>
<tr>
<td>2 1000</td>
</tr>
<tr>
<td>3 1000</td>
</tr>
<tr>
<td>4 1000</td>
</tr>
</tbody>
</table>

Yes | No
The End
Live demo.
Consistency in the Real World

- Consistency is probably the most subtle aspect of computer architecture
- No one implements sequential consistency because it is too slow
  - Make all accesses visible everywhere, right away takes a long time
- Real machines (like mine) use “relaxed” models.
  - All manner of non-intuitive things can happen
  - Special instructions to enforce sequential consistency when it’s needed
- Threading libraries (like pthreads) provide locking routines that use those special instructions to make locks work properly.
- For more, take 240b