MultiProcessors
Multiprocessors

• 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
Frequency and Power

- \( P = CfV^2 \)
- \( f = \) processor frequency
- \( V = \) supply voltage
- \( C = \) circuit capacitance (basically xtr count)
- To increase \( f \) you need to increase \( V \) as well
  - Approximately: \( P = Cf^3 \)

- This means that even for in-order processors, frequency scaling is not power efficient
  - doubling the frequency doubles performance
  - increased power by 8x
- It is, however, very area-efficient/xtr-efficient
Multi-processors

• An alternative approach to increased performance: Build more processors
• N processors will do N times as much work per time

• Area efficiency:
  • Pretty good -- twice the area -> twice the performance
    (Maybe. Sometimes. More on this in moment)

• Power efficiency:
  • $P = Cf^3$
  • Two processors means doubling C, so 2x the power
    (compared to 8x for doubling the clock)

• This seems like a no-brainer, but it only works if you actually get 2x performance from two cores.
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
  - $Stot = \frac{1}{(x/S + (1-x))}$

- 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 1-4 actively running threads.
  • Servers can have more if needed (the sysadmins will hopefully configure it that way)
Parallel Programming is Hard

- **Difficulties**
  - Correctly identifying independent portions of complex programs
  - Sharing data between threads safely.
  - Using locks correctly
  - Avoiding deadlock

- **There do not appear to be good solutions**
  - We have been working on this for 30 years (remember, multi-processors have been around for a long time.)
  - It remains stubbornly hard.
Critical Sections and Locks

- A critical section is a piece of code that only one thread should be executing at a time.

```c
int shared_value = 0;
void IncrementSharedVariable()
{
    int t = shared_value + 1; // Line 1
    shared_value = t;         // line 2
}
```

- If two threads execute this code, we would expect the shared_value to go up by 2
- However, they could both execute line 1, and then both execute line 2 -- both would write back the same new value.
Critical Sections and Locks

- A critical section is a piece of code that only one thread should be executing at a time.

```c
int shared_value = 0;
void IncrementSharedVariable()
{
    int t = shared_value + 1;  // Line 1
    shared_value = t;         // line 2
}
```

- If two threads execute this code, we would expect the shared_value to go up by 2
- However, they could both execute line 1, and then both execute line 2 -- both would write back the same new value.

Instructions in the two threads can be interleaved in any way.
Critical Sections and Locks

- By adding a lock, we can ensure that only one thread executes the critical section at a time.

```c
int shared_value = 0;
lock shared_value_lock;
void IncrementSharedVariable()
{
    acquire(shared_value_lock);
    int t = shared_value + 1; // Line 1
    shared_value = t;         // line 2
    release(shared_value_lock);
}
```

- In this case we say shared_value_lock “protects” shared_value.
Locks are Hard

- The relationship between locks and the data they protect is not explicit in the source code and not enforced by the compiler.
- In large systems, the programmers typically cannot tell you what the mapping is.
- As a result, there are many bugs.
void Swap(int * a, lock * a_lock, 
    int * b, lock * b_lock) {
    lock(a_lock);
    lock(b_lock);
    int t = a;
    a = b;
    b = t;
    unlock(a_lock);
    unlock(b_lock);
}

Thread 1
Swap(foo, foo_lock, 
    bar, bar_lock);

Thread 2
Swap(bar, bar_lock, 
    foo, foo_lock);
Locking Bug Example

```c
void Swap(int * a, lock * a_lock, int * b, lock * b_lock) {
    lock(a_lock);
    lock(b_lock);
    int t = a;
    a = b;
    b = t;
    unlock(a_lock);
    unlock(b_lock);
}
```

Thread 1

Swap(foo, foo_lock, bar, bar_lock);

Thread 2

Swap(bar, bar_lock, foo, foo_lock);

Thread 1 locks foo_lock, thread 2 locks bar_lock, both wait indefinitely for the other lock.
Finding, preventing, and fixing this kind of bug are all hard
The Future of Threads

• Optimists believe that we will solve the parallel program problem this time!
  • New languages
  • New libraries
  • New paradigms
  • Revamped undergraduate programming courses

• Pessimists believe that we won’t
  • There is probably not a good, general solution
  • We will make piecemeal progress
  • Most programs will stop getting faster
  • CMPs just make your spyware run faster.

• Intel and Microsoft believe typical users can utilize up to about 8 cores effectively.
  • Your laptop will be there in 2-3 years. Is already there
Architectural Support for Multiprocessors

- The Big-A architecture gives meaning to programs.
- So far, we’ve been concerned with a single processor.
- What do multi-threaded programs mean?
  - It depends on what they share.
  - In most systems, they share memory
- So, 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.
• With multiple caches, there can be many copies.
• No one processor can see them all.
• Which one has the “right” value?
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.
• There is one copy of the state machine for each line in each coherence cache.
Caches, Caches, Everywhere

Store A to 0x1000

Local caches

Exclusive
0x1000: A

Main Memory

0x1000: Z

Bus/arbiter
Caches, Caches, Everywhere

Store 0x1000
Shared 0x1000:A

Read 0x1000
Shared 0x1000:A

Local caches

Bus/arbiter

Main Memory

0x1000: A
Caches, Caches, Everywhere
Coherence in Action

while(1) {
    a++;
}

while(1) {
    print(a);
}

a = 0

Thread 1

while (1) {
    a++;
}

Thread 2

while (1) {
    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?
Coherence in Action

```plaintext
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>
</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
Coherence in Action

while(1) {
    a++;
}

while(1) {
    print(a);
}

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>
</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

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 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
  A = 10;
  A_is_valid = true;

  Thread 1
  while(!A_is_valid);
  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**

\[
\begin{align*}
A &= 10; \\
A\_is\_valid &= true;
\end{align*}
\]

**Thread 1**

\[
\begin{align*}
\text{while}(!A\_is\_valid); \\
B &= A;
\end{align*}
\]

- If thread 1 sees the write to A\_is\_valid, it will also see the write to A.
Sequential Consistency

- There is a total order over all operations in a program
- That order is consistent with the “program orders” of each of the processors.

- Program order is the order in which instructions are executed on a single processor
  - It matches your intuition for single-threaded execution.
What about this?

a = 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 1000
6 1000
7 1000
8 1000
What about this?

```c
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);
}
```

possible under sequential consistency? yes

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
```
What about this?

\[
a = b = 0
\]

Thread 1
\[
\text{while}(1) \{ \\
a++; \\
b++; \\
\}
\]

Thread 2
\[
\text{while}(1) \{ \\
p\text{rint}(a, b); \\
\}
\]

possible under sequential consistency? yes no

Sample outputs

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1</td>
<td>1 1</td>
</tr>
<tr>
<td>2 2</td>
<td>2 2</td>
</tr>
<tr>
<td>3 3</td>
<td>3 1000</td>
</tr>
<tr>
<td>4 4</td>
<td>2 1000</td>
</tr>
<tr>
<td>5 5</td>
<td>3 1000</td>
</tr>
<tr>
<td>6 6</td>
<td>4 1000</td>
</tr>
<tr>
<td>7 7</td>
<td></td>
</tr>
<tr>
<td>8 8</td>
<td></td>
</tr>
</tbody>
</table>
Simple Consistency

Thread 0
a A = 10;
b A_is_valid = true;

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

• We want to show that B = A will “see” A = 10 under sequential consistency
  • By program order, a happens before b, and c happens before d
  • Further, d only happens if !A_is_valid is false
  • !A_is_valid can only be false, if b has occurred.
  • So the instance b happens before of c
  • Therefore, a happens before b happens before c happens before d
“Relaxed Consistency”

• Not many machines are sequentially consistent
  • It requires global communication on every store.
  • It interferes with single-processor performance:

    array[4] = 10;
    x = array[4];

• The load from array[4] has to wait until all other processors are aware of the store to array[4]. That means, for instance, that the store cannot be speculative or out-of-order.
Where is SC?

Thread 0
a  A = 10;
b  A_is_valid = true;

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

• We want to show that \( B = A \) will “see” \( A = 10 \) under sequential consistency
  • By program order, \( a \) happens before \( b \), and \( c \) happens before \( d \) (Sequential Consistency)
  • Further, \( d \) only happens if \( !A\_is\_valid \) is false
  • \( !A\_is\_valid \) can only be false, if \( b \) has occurred.
  • So the instance \( b \) happens before \( c \)
  • Therefore, \( a \) happens before \( b \) happens before \( c \) happens before \( d \)
Live demo.
Relaxing Orderings for OOO

• For out of order execution, machines often maintain a store buffer to hide out-of-order writes. This violates SC.

• In this example, the mem ops at P1 may appear to happen in a different order from the perspective of P2 (and vice versa).
Dealing with Consistency

- Many programmers never consciously worry about consistency
  - Pthreads, java, or some other library/runtime makes the program look sequentially consistent as long as you use locks to protect shared variables.
  - If you don’t use locks properly, you are on your own.
  - In fact many systems declare “the behavior of incorrectly synchronized programs is undefined”
In modern systems “undefined” behavior is not acceptable
• In practice, large programs are almost never “correctly synchronized”
• It violates type safety (e.g., in Java)
• It may create security problems
• It makes it impossible to reason rigorously about the meaning of programs.

Much work has gone into trying to provide rigorous semantics for incorrectly synchronized code while still allowing for the optimizations that relaxed models allow.
• The models have become incomprehensible (e.g., the second Java memory model)
• and none of them have been successful.

When we start talking about language-level guarantees, the compiler gets involved too.

Most compilers do not preserve sequentially consistent semantics (this means you need to write incorrectly synchronized code in assembly)
The Midterm

- 70% of the class got > B-
- The two “F”s didn’t take the test

The Curve

<table>
<thead>
<tr>
<th>Grade</th>
<th># of Students</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>0</td>
</tr>
<tr>
<td>F+</td>
<td>6.2</td>
</tr>
<tr>
<td>D-</td>
<td>6</td>
</tr>
<tr>
<td>D</td>
<td>18</td>
</tr>
<tr>
<td>D+</td>
<td>15</td>
</tr>
<tr>
<td>C-</td>
<td>18</td>
</tr>
<tr>
<td>C</td>
<td>21</td>
</tr>
<tr>
<td>C+</td>
<td>24</td>
</tr>
<tr>
<td>B-</td>
<td>27</td>
</tr>
<tr>
<td>B</td>
<td>30</td>
</tr>
<tr>
<td>B+</td>
<td>33</td>
</tr>
<tr>
<td>A-</td>
<td>36</td>
</tr>
<tr>
<td>A</td>
<td>39</td>
</tr>
<tr>
<td>A+</td>
<td>42</td>
</tr>
</tbody>
</table>

- Number of students

Graph showing the distribution of grades.