CSCE 4610: Computer Architecture

Final Exam: May 9 10:30-12:30
Term projects Due: May 4, 2017.

Help Session Friday May 5 1:30pm  F 223

HW #9 (Due May 4) 5.2, 5.4, 5.6

Two Surveys to Complete:
1. UNT SPOT – evaluate the class
   2. CSE exit survey
      – you need to let us know how wel you learned the course objectives

Review
   Amdahl’s law of speedup
   Implementing locks using Load Linked (LL) and Store Conditional (SC)
      instead of Test and Set
The BEQZ R3, Try indicates that SC was not successful - no value was written to the lock.

BNEQZ R2, Try checks the value read by LL to see if it is zero (indicates a successful lock acquisition) or not (lock not acquired)

*This is covered in section 5.5 pages 386-391*

Implementing barrier synchronization using a counter
CSCE 4610: Computer Architecture

Barrier Synchronization.

Fork and Join semantics

How would we implement the join (barrier) using the atomic instructions we have seen so far. We need two locks: first acquire the join variable -- and increment; and one to make sure all processes wait until the last process arrives at the barrier.

```
.Lock (counter_lock);           /*to make sure updates are atomic
If (counter ==0) release = 0;   /* first reset release
    /* this is to make sure that processes
    /* are forced to wait

    count = count+1;
    unlock (counter_lock);
If (count ==total) {
    count = 0;                  /* after all arrive, reset the counter
    release = 1; }              /* and release waiting processes
else spin (release = 1);        /* otherwise spin or wait until release
```

There are two locks -- the counter_lock and the release

Cache Coherency Problem in Shared multicore systems

In principle, each cache can monitor traffic on bus, see if local copies need to be invalidated or updated local values.

Each cache also need to decide if any local changes need to be placed on the shared bus.

Shared Memory MIMD

Implementation of a Snoopy Protocol (MESI)

Each cache cache line will be associated with one of 4 states.
Invalid (I); Modified (M); Exclusive (E) or Shared (S)
CSCE 4610: Computer Architecture

Read hit: No special actions. Read data from local cache

Read Miss: That is, the data is not available in local cache or the local data is Invalid.

Cases:
- ✓No other cache has the data. Missing data is fetched from main memory. The local copy is set to Exclusive state.

- ✓One or more caches have copies of the data (in Shared or read only). Main memory supplies the data, the local copy is set to Shared (or read only).

- ✓Only one other cache has a copy (either in Exclusive or Modified state).
  - If in M, data is written back to memory, data is supplied to requesting cache; all copies are set to Shared.
  - If in E, data is supplied to requesting cache and all copies are set to Shared
CSCE 4610: Computer Architecture

Write Hit: The address to which you are planning to write is in local cache (and not in Invalid state).

Cases:

✓ The local copy is in Shared State. An invalidation message is broadcast on the bus; all other cache copies are set to Invalid; local copy is set to Modified.

✓ Local copy is in Exclusive state. Only copy, hence write can proceed; local copy is set to Modified

✓ Local copy is in Modified state; only copy and write can proceed; local copy remain in Modified state

Write-Miss
Treat this as Read-Miss followed by Write-hit
CSCE 4610: Computer Architecture

The same implementation is described in more detail on pages 357-361

Note, we can combine E and M states.

On page 360, the diagrams show what happens to the state of a local cache data and remote cache data. This diagram uses only 3 states: Invalid, Exclusive and Shared.

Figure on page 361 combines the two state diagrams on page 360

Local actions in bold lines and remote actions on bus in light lines.
CSCE 4610: Computer Architecture

Cache state transitions based on requests from the bus

CPU read hit

CPU read miss

Write miss for this block

Write-back block; abort memory access

CPU write hit

CPU write miss

Write-back cache block

Place write miss on bus

CPU read hit

CPU read miss

Write miss for this block

Write-back block; abort memory access

Invalid

Shared (read only)

Exclusive (read/write)

CPU read

Place read miss on bus

CPU write

Place write miss on bus

CPU read hit

Cache state transitions based on requests from CPU
Cache coherency: MESI protocol example
Notation for the sequence of memory accessess
\[ P\#: \langle \text{op} \rangle \langle \text{address} \rangle [ \leftarrow \langle \text{value} \rangle ] \]

<table>
<thead>
<tr>
<th>P0: read</th>
<th>P0: 120 E</th>
<th>P1: 120 I</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>110 I</td>
<td>110 I</td>
</tr>
<tr>
<td>P1: read</td>
<td>P0: 120 S</td>
<td>P1: 120 S</td>
</tr>
<tr>
<td></td>
<td>110 I</td>
<td>110 I</td>
</tr>
<tr>
<td>P1: write 120 ← 80</td>
<td>P0: 120 I</td>
<td>P1: 120 M 80</td>
</tr>
<tr>
<td></td>
<td>110 I</td>
<td>110 I</td>
</tr>
<tr>
<td>P0: write 120 ← 60</td>
<td>P0: 120 M 60</td>
<td>P1: 120 I</td>
</tr>
<tr>
<td></td>
<td>110 I</td>
<td>110 I</td>
</tr>
<tr>
<td>P1: read 120</td>
<td>P0: 120 S 60</td>
<td>P1: 120 S 60</td>
</tr>
<tr>
<td></td>
<td>110 I</td>
<td>110 I</td>
</tr>
<tr>
<td>P1: read 110</td>
<td>P0: 120 S 60</td>
<td>P1: 120 S 60</td>
</tr>
<tr>
<td></td>
<td>110 I</td>
<td>110 E</td>
</tr>
<tr>
<td>P0: read 110</td>
<td>P0: 120 S 60</td>
<td>P1: 120 S 60</td>
</tr>
<tr>
<td></td>
<td>110 S</td>
<td>110 S</td>
</tr>
<tr>
<td>P0: write 110 ← 90</td>
<td>P0: 120 M 60</td>
<td>P1: 120 S 60</td>
</tr>
<tr>
<td></td>
<td>110 M 90</td>
<td>110 I</td>
</tr>
</tbody>
</table>
Variations to MESI protocol

Invalidation vs Update Protocols
instead of invalidating other copies, broadcast the new data performance issues?

MOSI protocol: Instead of Exclusive, we will have a owner
Owner will have the most recently updated copy and is responsible for write-back

Ownership changes, but only one owner at any given time
Owner is responsible for supplying data on request from other processors
Note the difference between O and M
P0: read
P0: 120 O 110 I
P1: 120 I

P1: read
P0: 120 O 110 I
P1: 120 S 110 I

P1: write 120 ← 80
P0: 120 I 110 I
P1: 120 M 80 110 I

P0: write 120 ←60
P0: 120 M 60 110 I
P1: 120 I 110 I

P1: read 120
P0: 120 O 60 110 I
P1: 120 S 60 110 I

P1: read 110
P0: 120 O 60 110 I
P1: 120 S 60 110 O

P0: read 110
P0: 120 O 60 110 I
P1: 120 S 60

P0: write 110 ← 90
P0: 120 O 60 110 M 90
P1: 120 S 60 110 I
CSCE 4610: Computer Architecture

Comparing MOSI and MESI for performance

We are given that it takes 100ns to get data from memory, 70ns to get data from another cache, 15ns to invalidate data.
Calculate the total time needed to complete the above sequence of memory accesses.

Consider using MOSI

| P0: read       | 100ns | (get data from memory) |
| P1: read       | 70ns  | (get data from P0 cache) |
| P1: write 120 ← 80 | 15ns  | (invalidate P0 data) |
| P0: write 120 ← 60 | 70ns+15ns | (get data from P1, and invalidate P1 cache) |
| P1: read 120   | 70ns  | (get data from P0 cache) |
| P1: read 110   | 100ns | (get data from memory) |
| P0: read 110   | 70ns  | (get data from P1) |
| P0: write 110 ← 90 | 15ns  | (invalidate P1 cache) |

Total: 525ns
Consider using MESI (more accesses to memory)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0: read</td>
<td>100</td>
</tr>
<tr>
<td>P1: read</td>
<td>100</td>
</tr>
<tr>
<td>P1: write 120 ← 80</td>
<td>15</td>
</tr>
<tr>
<td>P0: write 120 ← 60</td>
<td>200 + 15</td>
</tr>
<tr>
<td>P1: read 120</td>
<td>100</td>
</tr>
<tr>
<td>P1: read 110</td>
<td>100</td>
</tr>
<tr>
<td>P0: read 110</td>
<td>100</td>
</tr>
<tr>
<td>P0: write 110 ← 90</td>
<td>15</td>
</tr>
<tr>
<td>Total</td>
<td>745</td>
</tr>
</tbody>
</table>
CSCE 4610: Computer Architecture

Relying on Cache coherency to implement LOCKs

Where do we keep the lock variables?
   In local cache?
   Global memory?

Spin locks? Keep a local copy
   Local copy is automatically invalidated when there a change in the global memory

Let us review how we implemented locks. If you have a test-and-set like instruction

    test-&set:  DADDUI  R2, R0, #1    /* Set R2=1
    LOCKIT:    EXCH     R2 0(R1)     /* atomic exchange R2 with lock location
                                 /* can be implemented using LL and SC
                        BNEZ   R2, LOCKIT   /* if lock is not available, try again

Note, the EXCH repeatedly stores 1 in lock location (even if it is already 1)

Consider what happens if we use MESI or MOSI protocol
You keep invalidating data at other caches – causing a significant performance loss

Let us consider how Spin locks may work.

We do not store a value until you check the value first
This is sometimes called test and test&set

```
ttas:    LD    R2, 0(R2)        /* read the current value of lock
          BNEZ  R2, ttas        /* if the value is 1, try again
          DADDUI R2, R0, #1     /* set R2 =1
          EXCH R2, 0(R1)        /* atomic exchange R2 with lock location
          BNEZ  R2, ttas        /* lock is still not available
```

Here we keep testing the local value stored in cache (or spin on local copy). When the lock is released (write zero into lock), the write will invalidate all other local or spinning copies
Then you try to store 1 into lock using exchange, if lucky you get the lock
If not you get 1 into your local cache, and spin again.

See problem 5.7. Here we have 3 processors. The problem compares spin locks with using EXCH instruction
We assume lock is at memory location 100. On page 391 we can see the traffic on bus caused by 3 processors P0, P1, P2.

If these processors are trying to acquire lock, how long will it take for all of them to acquire the lock?

<table>
<thead>
<tr>
<th>Step</th>
<th>P0</th>
<th>P1</th>
<th>P2</th>
<th>Coherence state of lock at end of step</th>
<th>Bus/directory activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Has lock</td>
<td>Begins spin, testing if lock = 0</td>
<td>Begins spin, testing if lock = 0</td>
<td>Shared</td>
<td>Cache misses for P1 and P2 satisfied in either order. Lock state becomes shared.</td>
</tr>
<tr>
<td>2</td>
<td>Set lock to 0</td>
<td>(Invalidate received)</td>
<td>(Invalidate received)</td>
<td>Exclusive (P0)</td>
<td>Write invalidate of lock variable from P0.</td>
</tr>
<tr>
<td>3</td>
<td>Cache miss</td>
<td>Cache miss</td>
<td></td>
<td>Shared</td>
<td>Bus/directory services P2 cache miss; write-back from P0; state shared.</td>
</tr>
<tr>
<td>4</td>
<td>(Waits while bus/directory busy)</td>
<td></td>
<td></td>
<td>Shared</td>
<td>Cache miss for P2 satisfied</td>
</tr>
<tr>
<td>5</td>
<td>Lock = 0</td>
<td>Executes swap, gets cache miss</td>
<td></td>
<td>Shared</td>
<td>Cache miss for P1 satisfied</td>
</tr>
<tr>
<td>6</td>
<td>Executes swap, gets cache miss</td>
<td></td>
<td>Completes swap; returns 0 and sets lock = 1</td>
<td>Exclusive (P2)</td>
<td>Bus/directory services P2 cache miss; generates invalidate; lock is exclusive.</td>
</tr>
<tr>
<td>7</td>
<td>Swap completes and returns 1, and sets lock = 1</td>
<td>Enter critical section</td>
<td></td>
<td>Exclusive (P1)</td>
<td>Bus/directory services P1 cache miss; sends invalidate and generates write-back from P2.</td>
</tr>
<tr>
<td>8</td>
<td>Spins, testing if lock = 0</td>
<td></td>
<td></td>
<td>None</td>
<td></td>
</tr>
</tbody>
</table>
Initially all caches are invalid

Assume P0 is the first to create read miss and gets a copy with zero from 100
it takes 100 cycles for this operation
And P0 successfully stores a 1 locally (with Modified state)
P0 acquired the lock

Assume P1 is the next processor to Exchange (read and write)
P0 detects this request, provides the data to P1
But P1 is also writing so P0 invalidates its copy and P1 will have Modified
Note P1 simply stored a 1 into 100 which is already 1

This takes 70 + 15 cycles
Assume P2 is the next processor to cause a exchange miss on 100
P1 has the data in M; supplies the data to P2 and invalidate local copy
P2 receives the data in M state
Again, P2 only writes 1 does not really modify.

This takes 70+15 cycles

At this point, P0 does not access the lock variable any longer. Completes critical section
(we are told critical section is 1000 cycles long)

But, P1 and P2 repeatedly try to invalidate each other’s copy
(each taking 85 cycles)

After 1000 cycles, P0 wants to release the lock

Depending on which of P1 and P2 has the data in M, supplies to P0 and invalidate local
P0 has the data in M state (and writes a zero)
Takes 70+15 cycles
Then, P1 acquires the lock
   P1 cause EXCH miss
   P0 provides the data (now zero) and invalidates local copy
   P1 writes a 1, and has the data in M state
   This takes 70+15 cycles

P2 tries to acquire and causes EXCH miss
   P1 provides the data (now 1) and invalidates local copy
   P2 gets the data in M (but only writes 1)
   This takes 70+15 cycles

P1 proceeds to complete the critical section (1000 cycles)
P2 spins (or repeatedly writes 1 to the lock)

After 1000 cycles, P1 wants to release the lock
   P2 supplies data and invalidates local copy
   P1 has the data in M state (and writes a zero)
   Takes 70+15 cycles
CSCE 4610: Computer Architecture

P2 tries again to acquire and causes EXCH miss
  P1 supplies the data (now 0) and invalidates local
  P2 receives the data in M, and writes a 1.

P2 can now proceed to the critical section
### CSCE 4610: Computer Architecture

<table>
<thead>
<tr>
<th>Operation</th>
<th>Source/State</th>
<th>Destination/States</th>
<th>Time 1</th>
<th>Time 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0 EXCH</td>
<td>from Mem</td>
<td>P0: M; P1: I; P2: I</td>
<td>100</td>
<td>1000</td>
</tr>
<tr>
<td>P0 critical section</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P1 EXCH</td>
<td>from P0</td>
<td>P0: I; P1: M; P2: I</td>
<td>70+15</td>
<td>85</td>
</tr>
<tr>
<td>P2 EXCH</td>
<td>from P1</td>
<td>P0: I; P1: I; P2: M</td>
<td>70+15</td>
<td>85</td>
</tr>
<tr>
<td>P1 and P2 spin</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P0 Release</td>
<td>from P1 or P2</td>
<td>P0: M; P1: I; P2: I</td>
<td>70+15</td>
<td>85</td>
</tr>
<tr>
<td>P1 EXCH</td>
<td>from P0</td>
<td>P0: I; P1: M; P2: I</td>
<td>70+15</td>
<td>85</td>
</tr>
<tr>
<td>P1 Critical section</td>
<td></td>
<td></td>
<td></td>
<td>1000</td>
</tr>
<tr>
<td>P2 EXCH</td>
<td>from P1</td>
<td>P0: I; P1: I; P2: M</td>
<td>70+15</td>
<td>85</td>
</tr>
<tr>
<td>P2 spins</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P1 release</td>
<td>from P2</td>
<td>P0: I; P1: M; P2: I</td>
<td>70+15</td>
<td>85</td>
</tr>
<tr>
<td>P2 EXCH</td>
<td>from P1</td>
<td>P0: I; P1: I; P2: M</td>
<td>70+15</td>
<td>85</td>
</tr>
<tr>
<td>P2 critical section</td>
<td></td>
<td></td>
<td></td>
<td>1000</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
<td></td>
<td>3695</td>
</tr>
</tbody>
</table>

P1 and P2 spin
Let us now compare the cycles needed when we use test-test&set

Assume P0 is the first to create read miss and gets a copy with zero from 100
it takes 100 cycles for this operation
And P0 successfully stores a 1 locally (with Modified state)

Assume P1 reads the data
P0 provides the data, changes local state to O; P1 receives in S
Takes 70+15 cycles
P1 does not try to change the data because it received 1

Then P2 reads the data
P0 provides the data (still O); P2 receives in S
Takes 70+15 cycles
P2 does not change the data

P0 completes critical section (1000 cycles)

P0 releases the lock
Invalidates P1 and P2 (15 cycles)
P0 has the lock in M
CSCE 4610: Computer Architecture

P1 reads the data
   P0 provides the data, changes to O; P1 in S
   Takes 70 cycles
   P1 tries EXCH
   P0 data is invalidated, P1 changes state to M
   Takes 15 cycles
   P1 acquired the lock

P2 reads the data
   P1 supplies the data, change to O, P2 in S
   Takes 70 Cycles
   P2 spins on local copy

P1 proceeds to complete critical section (1000 cycles)
P1 releases
   P1 provides the data and invalidates local copy
   P2 writes 0 (and changes state to M)
   Takes 70+15 cycle
## CSCE 4610: Computer Architecture

<table>
<thead>
<tr>
<th>Process</th>
<th>Operation</th>
<th>Instructions</th>
<th>Source</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>Read 100</td>
<td>P0:E</td>
<td>From Mem</td>
<td>100</td>
</tr>
<tr>
<td>P0</td>
<td>Store 1</td>
<td>P0:M</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P1</td>
<td>read</td>
<td>P0:O; P1:S</td>
<td>from P0</td>
<td>70</td>
</tr>
<tr>
<td>P2</td>
<td>Read 100</td>
<td>P0:O; P1:S; P2: S</td>
<td>from P0</td>
<td>70</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Critical Section</td>
<td></td>
<td></td>
<td>1000</td>
</tr>
<tr>
<td>P0</td>
<td>Store 0</td>
<td>P0:M; P1:I; P2:I</td>
<td>Invalidate</td>
<td>15</td>
</tr>
<tr>
<td>P1</td>
<td>Read 100</td>
<td>P0:O; P1:S; P2:I</td>
<td>From P0</td>
<td>70</td>
</tr>
<tr>
<td>P1</td>
<td>Store 1</td>
<td>P0:I; P1:M; P2:I</td>
<td>Invalidate</td>
<td>15</td>
</tr>
<tr>
<td>P2</td>
<td>Read</td>
<td>P0:I; P1:O; P2:S</td>
<td>From P1</td>
<td>70</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Critical Section</td>
<td></td>
<td></td>
<td>1000</td>
</tr>
<tr>
<td>P1</td>
<td>Store 0</td>
<td>P0:I; P1:M; P2:I</td>
<td>Invalidate</td>
<td>15</td>
</tr>
<tr>
<td>P2</td>
<td>Read</td>
<td>P0:I; P1:O; P2:S</td>
<td>from P1</td>
<td>70</td>
</tr>
<tr>
<td>P2</td>
<td>Store</td>
<td>P0:I; P1:I; P2:M</td>
<td>Invalidate</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Critical Section</td>
<td></td>
<td></td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td></td>
<td></td>
<td>3510</td>
</tr>
</tbody>
</table>

With TATAS, we reduced traffic on the bus
False Sharing

Variables X1 and X2 are in the same cache block

Thread 1 on P1
......Write X1
............
......Write X1
............
......Read X2

Thread 2 on P2
.....
............
............
............
............

Consider using MESI like protocols “unnecessary invalidation of the cache line”

Using smaller cache blocks reduces false sharing – but performance penalties
CSCE 4610: Computer Architecture

Thread 1

……Write X1

……Write X1

……Read X1

Larger cache block – more false sharing
CSCE 4610: Computer Architecture

Read the performance comparisons presented in the text book section 5.3 (page 366-378) -- to see how the sharing and cache coherency impacts performance

For example, if you have more tasks, more processors, there are more coherency related cache misses.

Here we are focusing on L3 cache with each of 4 nodes.
Here we show memory access times

Figure 5.13 on page 371

With larger caches, instruction misses, Capacity/conflict become smaller (percentage) Sharing misses are now larger contributors
Figure on page 373 shows the impact of block size. Note larger block does reduce all miss rates -- even sharing.

False sharing does increase (relatively)
CSCE 4610: Computer Architecture

MESI or MOSI work only if we have a common bus
What if we do not have a common bus
  Distinguish between distributed memory systems that use message passing to access remote memories
  with systems that still view the distributed memory as “shared memory”
  distributed shared memory systems (DSM)

DSM use additional hardware to translate requests to remote memory into messages instead of requiring programmer to generate messages

Directory based protocols
  Each memory at each node/processor includes a “directory”
  The directory keeps track of the copies of the data at that node
  Each directory entry keeps track of data at “cache line” granularity

  Each processor cache still need state information
    Shared, Invalid or Modified