CSCE 5610: Computer Architecture

Homework #8: Due on Dec. 4, 2018

Final Exam: Dec. 13, 10:30-12:30
Project Reports Due: Dec 6, 2018 (last day of classes)

Help session: Friday Dec. 7, 2018 at 1pm in F 280

SPOT evaluations

Review: Locks and Barriers
Snooping cache coherence protocols
MSI and MOSI

Using cache coherence to implement locks
test and test and set

Relying on Cache coherency to implement LOCKs

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

ttas:  
id  x2, 0(x1) /* read the current value of lock
BNEZ  x2, ttas /* if the value is 1, try again
add  x2, x0, #1 /* set R2 =1
EXCH  x2, 0(x1) /* atomic exchange R2 with lock location
BNEZ  x2, 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.

False Sharing
Performance lost due to coherency misses. Pages 395-404
The data is somewhat dated.
I will show some charts but not all.
For OLTP (database transactions)

Note coherency misses (false and true sharing) remains the same no matter the size of the cache (L3 caches)
Capacity misses reduce with larger caches

The contribution to memory access cycles increases as processor count increases primarily because of increased sharing.
The compulsory misses slightly increase because each processor must now handle more compulsory misses.
Note that (relative) coherency misses increase as cache size increases and block size increases.

False sharing increases with block size.

(this data is for multithreaded programs – not OLTP and for shared L2 caches)
CSCE 5610: Computer Architecture

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

Centralized Directory. p+1 Directory method

Note each entry in the table represents
information about a “cache block” sized data for the addresses in that memory unit.

Each entry contains one bit per processor and one extra bit to indicate if there is a writer/owner

If there is a writer, we can have the address of the writer
or simply use one of the p bits to indicate which processor is the writer

CSCE 5610: Computer Architecture

Note each local cache also must maintain a state with its cache block
    Shared (read only copy)
    Exclusive/Modified -- allowed to modify.

Consider what happens on Read-Miss, Read-Hit, Write-Miss and Write-Hit

Read-Miss. Request is sent to "home" memory (based on the address)

Memory unit checks its directory for the requested block
If no other processor has the block, grant read access
to the requesting node and mark the directory
If another processor has a copy
    (in read only) -- grant a copy to requesting node and mark directory

If another processor has a copy in write or modified
Send a "write-back" message to the processor with modified copy

After completion of the write back, grant a copy to requesting node
Mark the directory with read copies at both nodes
Local state at both nodes will be "Shared"
Write miss. Send a request to the home memory
If no copies, grant the copy to the requesting node mark the directory with write copy
requesting node sets its local state to "Modified"

If another copy exists?

Read copies --- send invalidation to all nodes containing copies
wait for acknowledgements

Grant write copy to requesting node, modify directory with a single write copy
Receiving node marks the state to "modified"

If a write copy exists, send write back with invalidation
After write back completes, grant write request and modify directory with new writer
receiving node marks the state to "modified"

Write Hit
This implies that a copy already exists locally
if the state of local copy is Modified (that is, multiple writes)
no need to notify anyone else.

if the local state is shared, send a write request to home memory
home memory sends invalidation messages to all other processors
(if any), wait for acknowledgements and then grant
write request -- modify directory entry to reflect a writer
local node sets the state to "modified"
Let us consider the following actions. 110 is in memory at node 0 (M0) and 120 is in memory at node 1 (M1). We will denote D0 and D1 as the directories and C0 and C1 as caches at the nodes.

- P0: read 120
- P1: read 120
- P1: write 120 ← 80
- P0: write 120 ← 60
- P1: read 120
- P1: read 110
- P0: read 110
- P0: write 110 ← 90
- P1: read 120
- P1: writeback 120
- P1: writeback 120
- P0: writeback 120
- P1: read 110
- P1: read 110
- P0: write 110 ← 90
- P0: write 110 ← 90
- P1: writeback 110
- P0: writeback 110
- P1: writeback 110
- P0: writeback 110
- P1: invalidate 120 at P0 and invalidate P0
- P1: invalidate 120 at P0 and invalidate P0
- P0: invalidate 110 at P1 and invalidate P1
- P1: invalidate 110 at P1 and invalidate P1

Performance issues -- memory overhead for directories

(#directory entry) * (p+1)

We also need 2 bits per cache line at each cache.
Note p+1 requires a large amount of memory for the directory

If at any time only a small number of processors share the same data
number of copies is small, say n

If there are p processors, we can identify each processor with log (p) bits
If we allow n copies (n processors with local cache copies),
we need n* log(p) bits to track n copies
We still need one more bit to track if there a modified copy
We still need 2 bits for each cache line

What if we need more than n copies?
We may use one bit to indicate that there are more than n copies
We use broadcast to invalidate copies
a node may receive an invalidation message even if it does not
have a copy

Directory bits = (number of entries)  * [n*(log[p]) + 1 + 1]

Each line requires 2 bits to distinguish if the line is exclusive and if broadcast is needed.
Each cache = number of lines *2

Memory Consistency

How do define the correctness of a parallel program?

Consider the following example.

```
a = 1
b = 1
c = 1
print(b,c)
print(a,c)
print(a,b)
```

What should be the “correct” output to expect?

- is it possible to print 11 11 11?
- what about 00 00 00?

Sequential Consistency (Lamport)

1. Program Order must be maintained
2. The programs should behave as if all instructions are interleaved
   and executed on a single processor
One possible interleaving

\begin{align*}
a &= 1 & P1 \\
b &= 1 & P2 \\
c &= 1 & P3 \\
print(a,c) &= P2 & 11 \\
print(b,c) &= P1 & 11 \\
print(a,b) &= P3 & 11 \\
\end{align*}

Another possible interleaving

\begin{align*}
a &= 1 & P1 \\
print(b,c) &= P1 & 00 \\
b &= 1 & P2 \\
print(a,c) &= P2 & 10 \\
c &= 1 & P3 \\
print(a,b) &= P3 & 11 \\
\end{align*}

Since there are 6 statements, we could have $6!$ possible interleavings but some of them are not valid. Can’t change program order.

The following is not a legal interleaving

\begin{align*}
\text{print}(b,c) &= P1 & 00 \\
b &= 1 & P2 \\
\text{print}(a,c) &= P2 & 00 \\
c &= 1 & P3 \\
a &= 1 & P1 \\
\text{print}(a,b) &= P3 & 11 \\
\end{align*}

In general, guaranteeing sequential consistency (program order and proper interleaving) lead to performance losses in modern processors.

Another example:

\begin{align*}
P1: & \quad A = 0; & P2: & \quad B = 0; \\
& \quad \ldots \ldots & & \quad \ldots \ldots \\
& \quad A = 1; & & \quad B = 1; \\
& \quad \text{if } (B = 0) \ldots & & \quad \text{if } (A = 0) \ldots \\
\end{align*}

Consider possible interleaving of the statements

Is it possible for both conditions to be true?
If assume Sequential consistency, it is not possible for both If conditions to be true

However, if we are using cache coherency, and with say P1 modifies A=1, this did not immediately invalidate the copy in P2

Then it is possible for both If conditions to evaluate to true

This is because of implementation difficulties in guaranteeing SC

How do we achieve Sequential Consistency (SC)?

1. Maintain program order – can be a problem if we are using out-of-order processors
2. All memory accesses from all processors should be "ordered"
   All memory accesses from all processors must be visible to all processors at the same time
   *This either requires modifications to make registers visible or no registers
   Most processors use "write barriers" and "memory barriers" to make visible local changes

Sequential Consistency implementation defeats all architectural optimizations like
out of order violates program order
If we use write-back cache, writes may not be seen by other processors
Same is true if we use write buffers with write-through caches

Even if we use MSI/MOSI protocols, the invalidate may not take place at all caches before data is modified → ideally we need to wait until an acknowledgement that all caches have invalidated data before modifying cached values

Consider the example on page 418:
50 cycles to establish ownership
10 cycles to issue invalidations
80 cycles to received acknowledgement of invalidations

If we have 4 additional processors (in addition to the processor trying to modify)
before a shared item can be modified, the processor must wait
50 + 10+10+10+10 +80 = 170 cycles
Relaxing the strict order. Note that we need to worry only about Read and Write to shared memory.

Sequential consistency requires that we maintain order on all Reads and Writes and the order should be the same on all nodes. That is we have make sure of proper ordering of:

\[ W \rightarrow W; \ W \rightarrow R; \ R \rightarrow R; \ R \rightarrow W \]

What do these dependencies mean?

What if we only maintain order on Writes (W→R)? That means all writes should be ordered and any read will provide the latest value. Again the writes must be visible to all nodes. Cache coherency supports this.

What if we maintain order between W→W (Total Store Order)? That means we order Writes.

But we do not guarantee that a read returns the most recent value.

For example if P0 writes first. This write may not be visible to other processors. However, if both P0 and P1 writes, we define an order on the writes. This allows one to buffer writes.

What if we maintain only R→W? This means that a read and writes on a processor are ordered (but not across processors).

What if we maintain only R→R? No ordering of writes.

The last two allow architecture to explore out of order execution, buffering writes etc.
CSCE 5610: Computer Architecture

How to achieve Sequential Consistency on realistic architectures
If hardware does not provide SC, then the programmer must define ordering

Some architectures included barrier instructions
Write Barrier: Before a write can take place, complete all previous reads and writes at all nodes
Memory barrier: before completing the memory operation (either R or W) complete all previous memory operations at all nodes

Or if we assume that programmers will use locks or critical sections then we can achieve SC like consistency by enforcing order on locks

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock</td>
<td>Lock</td>
<td>Lock</td>
</tr>
<tr>
<td>a = 1</td>
<td>b = 1</td>
<td>c = 1</td>
</tr>
<tr>
<td>if b=0 then a=0</td>
<td>if c = 0 then b =0</td>
<td>if a=0 then c = 0</td>
</tr>
<tr>
<td>Unlock</td>
<td>Unlock</td>
<td>Unlock</td>
</tr>
<tr>
<td>print(b,c)</td>
<td>print(c)</td>
<td>print(a,b)</td>
</tr>
</tbody>
</table>

S stands for synchronization instruction like lock and unlock or barriers
So, S → W means that writes are not allowed until all pending synchronizations are complete
CSCE 5610: Computer Architecture

We will force an order on all lock accesses of all processors, and this order is the same for all processors, we now have a global order on shared memory inside a critical section.

No order on private data

Note that we do not guarantee any order on memory accesses outside of the locks

P1        P2        P3
print(b,c) print(a,c) print(a,b)
Lock      Lock      Lock
a = 1     b = 1     c = 1
if b=0 then a=0 if c = 0 then b =0 if a= 0 then c = 0
Unlock    Unlock    Unlock
print(b,c) print(a,c) print(a,b)

Assume P1 acquires lock first. However until lock is released (release consistency), a is not modified globally

So P2 and P3 may still see old value of a=0

CSCE 5610: Computer Architecture

Note we are not guaranteeing (partial) sequential consistency.
We only guarantee that all modifications inside a critical sections (collectively) are ordered (the order depends on how locks are acquired)

Consider the following example

P1

LOCK (L1)
A=1
B=2
UNLOCK(L1)

P2

LOCK(L1)
x=A
y=B
X1=A
UNLOCK (L1)
X2=B

What are the possible values for x and y if using SC and if using Relaxed Consistency?
Consider the possible orders using SC:

- P1: LOCK (L1)
  - A=1
  - B=2
  - UNLOCK(L1)
- P2: LOCK (L1)
  - x=A
  - y=B
  - X1=A
  - X2=B

In the first case: X1=0, X2=0. In the second case: X1=0, X2=2.

In Sequential, changes to memory, even inside a critical section, are visible to all processors but in Release consistency, changes are made visible only when the lock is released.

Now consider possible orders using RC:

Using RC:

- P1: LOCK (L1)
  - A=1
  - B=2
  - UNLOCK(L1)
- P2: LOCK (L1)
  - x=A
  - y=B
  - X1=A
  - X2=B

In both cases X2=0.

B=2 inside P1 is not visible until P1 UNLOCKS.

In other words, the accesses outside a lock can only appear before or after lock acquired by other processors.

Note that any RC orders are among correct SC orders.

But not all SC orders are possible in RC.