Lecture 15:
A Basic Snooping-Based Multi-processor

Parallel Computer Architecture and Programming
CMU 15-418, Spring 2013
Class policy changes

- In response to feedback in the early evaluation, I am dropping participation to 6%, and shifting those points to exam 2 and the project.
  - Assignments: $5 + 10 + 13 + 10 = 38\%$
  - Exams: $12.5 + 15.5 = 28\%$
  - Project: 28\%
  - Participation: 6\% (questions in class + comments + lecture explanations... which have be good!)

- You are encouraged to keep making comments!
  - One well thought out contribution per week seems reasonable
    (or one *really good* comment every two weeks, like those of who have contributed code snippets to test ideas from lecture)
  - Many of you are far below that
  - Perfectly fine to comment on another group’s writeup
Today’s topic:
A basic implementation of cache-coherence

- Wait... haven’t we talked about this before?

- Before spring break we talked about cache coherence protocols
  - But our discussion was very abstract (a protocol is an abstraction)
    - We described what messages/transactions needed to be sent
    - We assumed messages/transactions were atomic
  - Today we will talk about efficiently implementing the desired protocol (in a real machine... behavior is more complex)
Our implementation goals

1. Correct
   - Implements cache coherence
   - Adheres to specified consistency model

2. High performance

3. Minimize cost (e.g., minimize extra hardware)

As you will see: tricks to gain high performance tend to make ensuring correctness tricky.
What you should know

- The concept of pipelining (you should already know this!)
- Deadlock, livelock, starvation
- Basic understanding of how a bus works
  - But keep in mind most modern interconnects are NOT buses! (a future lecture)
- Understand why maintaining coherence is challenging to implement, even when operating under simple machine design parameters
  - Mental model of hardware: many components operating in parallel
  - How do performance optimizations make correctness challenging?
Deadlock
Livelock
Starvation

(deadlock and livelock are clearly about correctness. Starvation is really an issue of fairness)
Deadlock

Deadlock is a state where a system has outstanding operations to complete, but no operation can make progress.

Can arise when each operation has acquired a shared resource that another operation needs.

There is no way for any thread (or, in this illustration, a car) to make progress unless some thread relinquishes a resource (“backs up”)
Yinzer deadlock

Non-technical side note:
Deadlock happens in Pittsburgh all the time

(Deadlock can be amusing when a bus driver decides to let another driver know he has caused deadlock... “go take 418 you fool”)

CMU 15-418, Spring 2013
Deadlock in computer systems

Example 1:

A produces work for B’s work queue

B produces work for A’s work queue

Queues are finite and workers wait if no output space is available

Example 2:

const int numEl = 1024;
float msgBuf1[numEl];
float msgBuf2[numEl];

int processId;
MPI_Comm_rank(MPI_COMM_WORLD, &processId);

... do work ...

MPI_Send(msgBuf1, numEl, MPI_INT, processId+1, ...
MPI_Recv(msgBuf2, numEl, MPI_INT, processId-1, ...

Every process sends a message (blocking send) to its neighbor to the right
Then receives message from neighbor to the left.
Livelock
Livelock
Livelock
Livelock is a state where a system is executing many transactions/operations, but no thread is making meaningful progress.

Can you think of a good everyday example?

Computer system examples:

Operations continually abort and retry
Starvation

State where a system is making overall progress, but some processes make no progress.
(green cars make progress, but yellow cars are stopped)

Starvation is usually not a permanent state
(as soon as green cars pass, yellow cars can go)

Example: assume left-right traffic must yield to top-bottom traffic.
A basic implementation of snooping
Basic system design

- One outstanding memory request per processor
- Single level, write-back cache per processor
- Interconnect is an atomic shared bus
- Cache can stall processor as it is carrying out coherence operations
Cache miss logic on a uniprocessor

1. Determine cache set (using appropriate bits of address)
2. Check cache tags (to determine if line is in cache)
   [Assume no matching tags, must read data from memory]
3. Assert request for bus
4. Wait for bus grant (as determined by bus arbitrator)
5. Send address + command on bus
6. Wait for command to be accepted
7. Receive data on bus

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
</table>

Multi-processor atomic bus:
BusRd, BusRdX: no other bus transactions allowed between issuing address and receiving data
BusWr: address and data sent simultaneously, received by memory before any other transaction allowed
Multi-processor cache controller behavior

Challenge: both requests from processor and bus require tag lookup

If bus receives priority:
During bus transaction, processor is locked out from cache.

If processor receives priority:
During processor cache accesses, cache can’t respond with snoop (delaying other processors even if no sharing of any form is present)

** snoop controller has its mind on the bus and the bus on its mind
Allow simultaneous access by processor-side and snoop controllers

Option 1: Duplicate tags

Option 2: multi-ported tag memory

Note: tags must stay in sync for correctness, so tag update by one controller will still need to block the other controller (but modifying tags is infrequent compared to checking them)
Reporting snoop results in the MESI protocol

- Assume a cache read miss
- Collective response of caches must appear on bus
  - Is line dirty? If so, memory should not respond (MESI)
  - Is line shared? If so, cache should load into S state, not E

HOW?
WHEN?
Reporting snoop results: how

```
Bus
- Address
- Data
- Shared
- Dirty
- Snoop-valid

'OR' of result from all processors
(0 value indicates all processors have responded)
```
Reporting snoop results: when

Mainly an issue of when memory should react to the request

1. **Fixed number of clocks (worst case) after address appearing on bus**
   - All caches guaranteed to respond in a fixed number of clocks
   - Note importance of duplicated tags (to meet guarantee)

2. **Variable delay**
   - Memory assumes one of the caches will service request until it hears otherwise
   - More complex, but lower latency if snoops are completed quickly
Handling write backs

- Write backs involve two bus transactions
  1. Incoming line (line requested by processor)
  2. Outgoing line (evicted dirty line in cache that must be flushed)

- Ideally would like the processor to continue as soon as possible (shouldn’t have to wait for the flush to complete)

- Solution: write-back buffer
  - Stick line to flush in a write-back buffer
  - Immediately load requested immediate (allows processor to continue)
  - Flush contents of write-back buffer at a later time
Cache with write-back buffer

What if a request for the address of the data in the write-back buffer appears on the bus?

Snoop controller must check write-back buffer address in addition to cache tags.

If match:
1. Respond with data from write-back buffer rather than cache
2. Cancel outstanding bus access request (for the write-back)
Non-atomic state transitions

- State transition diagrams during protocol discussion assumed that transitions were atomic
- Today we assume the bus transaction is atomic, but all the operations the system performs as a result of a memory operation are not
  - Look up tags, arbitrate for bus, wait for actions by other controllers, etc.
- Must be careful to handle race conditions appropriately
Example race condition

Processors P1 and P2 write to cache line A simultaneously (both need to issue BusUpg to move line from S state to M state)

P1 “wins” bus access, sends BusUpg

P2 is waiting for bus access (to send its own BusUpg), can’t proceed because P1 has bus

P2 receives BusUpg, must invalidate line A (as per MESI protocol)

*P2 must also change its pending BusUpg request to a BusRdX*

Cache must be able to handle requests while waiting to acquire bus AND be able to modify its own outstanding requests
Write serialization

- Tempting optimization: on processor write, update cache line, allow processor to proceed prior to sending transaction out to bus (to obtain exclusive access)

- Violates coherence. Why?
  - Why does a write-back buffer not cause this problem?

- To ensure write serialization, cache cannot allow processor to proceed until read-exclusive transaction appears on bus
  - At this point, the write is “committed”
  - Key idea: order of transactions on the bus defines the global order
Fetch deadlock

P1 has a modified copy of cache line B
P1 is waiting for the bus to issue BusRdX on cache line A
BusRd for B appears on bus while P1 is waiting

To avoid deadlock, P1 must be able to service incoming transactions while waiting to issue requests
Livelock

Two processors writing to cache line B
P1 acquires bus, issues BusRdX
P2 invalidates
Before P1 performs write, P2 acquires bus, issues BusRdX
P1 invalidates
and so on...

To avoid livelock, a write that obtains exclusive ownership must be allowed to complete before exclusive ownership is relinquished.
Starvation

- Multiple processors competing for bus access
  - must be careful to avoid (or minimize likelihood of) starvation

- FIFO arbitration

- Priority-based heuristics
Design issues

- Design of cache controller and tags (to support access from processor and bus)
- How and when to present snoop results on bus
- Dealing with write backs
- Dealing with non-atomic state transitions
- Avoiding deadlock, livelock, starvation

These issues arose even though we only implemented a few optimizations on a basic invalidation-based, write-back system!

(atomic bus, one outstanding memory request per processor, single-level caches)

Next time: will discuss more advanced (a.k.a. more complex) implementations that strive for higher performance.
Source of the complexity: parallelism

- Processor, cache, and bus all are resources operating in parallel
  - Often contending for shared resources:
    - Processor and bus contending for cache
    - Caches contending for bus access

- “Memory operations” are abstracted by the architecture as atomic are implemented via multiple transactions involving all of these components

- Performance optimization often entails splitting operations into several, smaller transactions
  - Splitting work into smaller transactions reveals more parallelism
    (recall pipelining example)
  - Cost: more hardware needed to exploit additional parallelism
  - Cost: more care needed to ensure abstractions still hold (the machine is correct)