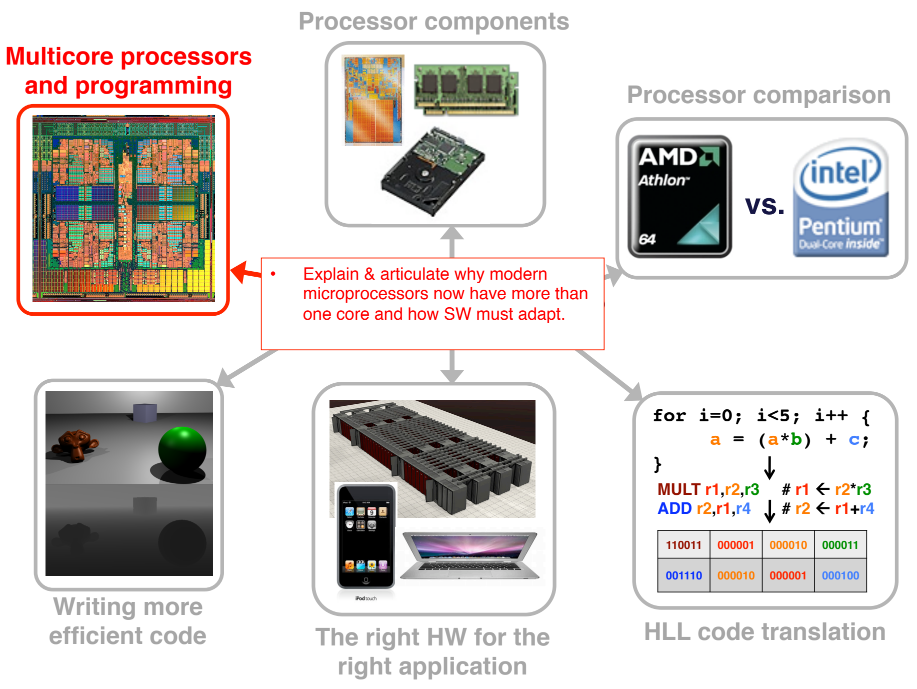
Lecture 24
Cache Coherency

Suggested reading:

(HP Chapter 5.8)

(Could also look at material on CD referenced on p. 538 of your text)


Multicore processors and programming
Processor components
Processor comparison
vs.
Writing more efficient code
The right HW for the right application
HLL code translation

- Explain & articulate why modern microprocessors now have more than one core and how SW must adapt.

```
for i=0; i<5; i++ {
     a = (a*b) + c;
}
MULT r1,r2,r3   # r1 ← r2*r3
ADD r2,r1,r4    # r2 ← r1+r4
```

110011	000001	000010	000011
001110	000010	000001	000100

Fundamental lesson(s)

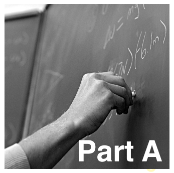
- Additional hardware mechanisms are essential for managing the memory hierarchy discussed in prior lectures such that program operation is (i) efficient and (ii) correct!

Why it's important…

1. The "additional hardware mechanisms" represent overheads that will degrade performance
2. Data management / movement within the memory hierarchy is important such that performance is not significantly degraded

What makes a memory system coherent?

- Program order
- All writes must be seen by all processors
- Causality



Coherency and Caches

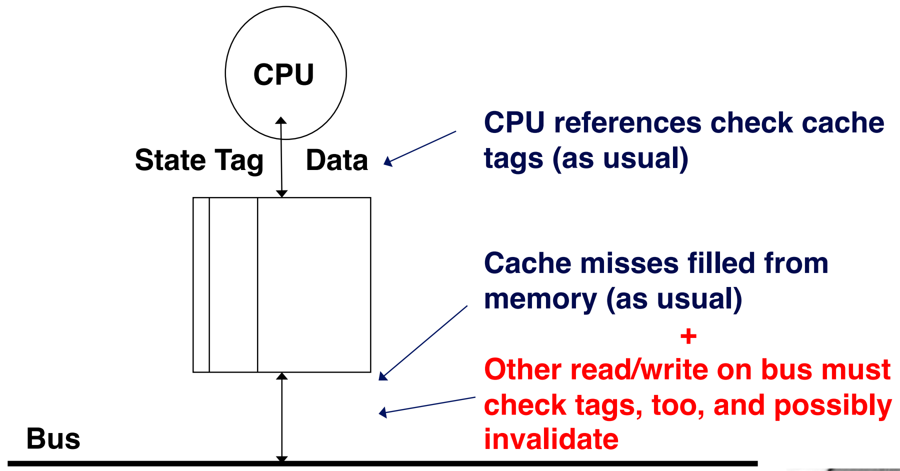
- One centralized shared cache / memory is not practical
  - Data must be cached locally
- Consider the following…
  - 1 node works with data no other node uses
    - Why not cache it?
  - If data is frequently modified, do we always tell everyone else?
  - What if data is cached and read by 2 nodes and now one of them wants to do a write?
    - How is this handled?

  …

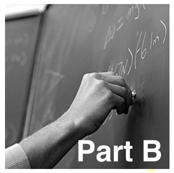
Maintaining Cache Coherence

- Hardware schemes
  - Shared Caches
    - Trivially enforces coherence
    - Not scalable (L1 cache quickly becomes a bottleneck)
  - Snooping
    - Needs a broadcast network (like a bus) to enforce coherence
    - Each cache that has a block tracks its sharing state on its own
  - Directory
    - Can enforce coherence even with a point-to-point network
    - A block has just one place where its full sharing state is kept

How Snooping Works

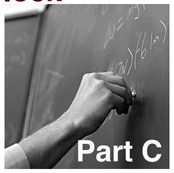

CPU
State Tag
Data
Bus
CPU references check cache tags (as usual)
Cache misses filled from memory (as usual)
+
Other read/write on bus must check tags, too, and possibly invalidate

Often 2 sets of tags…why?



Update vs. Invalidate

- A burst of writes by a processor to one address
  - Update: each sends an update
  - Invalidate: only the first invalidation is sent
- Writes to different words of a block
  - Update: update sent for each word
  - Invalidate: only the first invalidation is sent
- Producer-consumer communication latency
  - Update: producer sends an update, consumer reads new value from its cache
  - Invalidate: producer invalidates consumer's copy, consumer's read misses and has to request the block
- Which is better depends on application
  - But write-invalidate usually wins



Write invalidate example

Processor Activity	Bus Activity	Contents of CPU A's cache	Contents of CPU B's cache	Contents of memory location X
				0
CPU A reads X	Cache miss for X	0		0
CPU B reads X	Cache miss for X	0	0	0
CPU A writes a 1 to X	Invalidation for X	1		0
CPU B reads X	Cache miss for X	1	1	1

- Assumes neither cache had value/location X in it 1st
- When 2nd miss by B occurs, CPU A responds with value canceling response from memory.
- Update B's cache & memory contents of X updated
- Typical and simple…

Write update example

Processor Activity	Bus Activity	Contents of CPU A's cache	Contents of CPU B's cache	Contents of memory location X
				0
CPU A reads X	Cache miss for X	0		0
CPU B reads X	Cache miss for X	0	0	0
CPU A writes a 1 to X	Write broadcast of X	1	1	1
CPU B reads X		1	1	1

(Shaded parts are different than before)

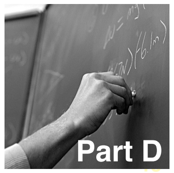
- Assumes neither cache had value/location X in it 1st
- CPU and memory contents show value after processor and bus activity both completed
- When CPU A broadcasts the write, cache in CPU B and memory location X are updated

M(E)SI Snoopy Protocols for $ coherency

- State of block B in cache C can be
  - Invalid: B is not cached in C
    - To read or write, must make a request on the bus
  - Modified: B is dirty in C
    - C has the block, no other cache has the block, and C must update memory when it displaces B
    - Can read or write B without going to the bus
  - Shared: B is clean in C
    - C has the block, other caches have the block, and C need not update memory when it displaces B
    - Can read B without going to bus
    - To write, must send an upgrade request to the bus
  - Exclusive: B is exclusive to cache C
    - Can help to eliminate bus traffic
    - E state not absolutely necessary

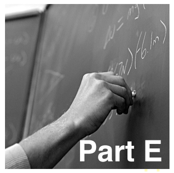
MSI protocol

- See notes and board



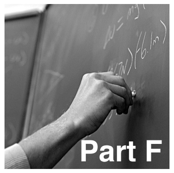
MESI protocol

- See notes and board



Cache to Cache transfers

- Problem
  - P1 has block B in M state
  - P2 wants to read B, puts a read request on bus
  - If P1 does nothing, memory will supply the data to P2
  - What does P1 do?
- Solution 1: abort/retry
  - P1 cancels P2's request, issues a write back
  - P2 later retries RdReq and gets data from memory
  - Too slow (two memory latencies to move data from P1 to P2)
- Solution 2: intervention
  - P1 indicates it will supply the data ("intervention" bus signal)
  - Memory sees that, does not supply the data, and waits for P1's data
  - P1 starts sending the data on the bus, memory is updated
  - P2 snoops the transfer during the write-back and gets the block



Directory-Based Coherence for DSM

- Typically in distributed shared memory
- For every local memory block, local directory has an entry
- Directory entry indicates
  - Who has cached copies of the block
  - In what state do they have the block

Basic Directory Scheme

- Each entry has
  - One dirty bit (1 if there is a dirty cached copy)
  - A presence vector (1 bit for each node)
    Tells which nodes may have cached copies
- All misses sent to block's home
- Directory does needed coherence actions
- Eventually, directory responds with data

Shared Memory Performance

- Another "C" for cache misses
  - Still have Compulsory, Capacity, Conflict
  - Now have Coherence, too
    - We had it in our cache and it was invalidated
- Two sources for coherence misses
  - True sharing
    - Different processors access the same data
  - False sharing
    - Different processors access different data,
      but they happen to be in the same block