CS 152, Spring 2012
Section 12

Christopher Celio
Agenda

- Snoopy Protocol Handout
- Directory Protocol Handout
- Lab #5
Mystery Chip?
Mystery Chip?

http://www.extremetech.com/compi
125871-apple-sneakily-debuts-32n
new-ipad-2-apple-tv

Friday, April 20, 2012
• Apple iPad 2 (iPhone 4s?)
• 9 metal layers

http://www.adafruit.com/blog/2011/03/15/apple-a5-floorplan/
http://www.electronista.com/articles/11/03/15/larger_die_faster_ram_built_by_samsung/

http://www.adafruit.com/blog/2011/03/15/apple-a5-floorplan/

Friday, April 20, 2012
A5X

- A4 - 53.3mm²
- A5X - 165mm² (x3.1!)
- 45 nm


Friday, April 20, 2012
Quiz #5

• Final Quiz!
  – Thursday (April 26th)
  – Last day of class

• Final Section
  – Tuesday (April 24th)
  – bring questions
Quiz #5: What will it cover?

• Memory consistency
  – sequential consistency
    • is a given memory system SC?
    • how to write correct code using a given memory system
      – using given ISA memory atomics, memory fences, etc.

• Writing code
  – how to write correct, parallel code
  – how to write good performance code
  – where do you need to insert fences?

• Cache Coherency
  – Snoopy protocol
  – Directory protocol
  – given a new memory system, does it break the protocol?
  – how to implement optimizations on the existing protocols discussed in the handouts?
Quiz #5: How to study?

• Past Quizzes (‘09, ‘10, ‘11)
• Handout #7 (Snoopy Protocol)
• Handout #6 (Directory Protocol)
• PSET #5
Handout #7: Snoopy Protocol

- Invalidation Protocol
  - delete other caches’ copies when a write occurs
- write-back caches
  - update memory on cache eviction
Snoopy Protocol: MBus Transactions

• CR: Coherent reads
  – issued by cache on read miss
• CRI: Coherent Read & Invalidate
  – write-allocate after a write-miss
  – (I want to write but need to read value, also invalidate everyone else’s copy)
• CI: Coherent Invalidate
  – issued by cache on a write hit to a block in the shared state
  – (I want to write my copy, invalidate everyone else’s copy)
• WR: Block Write
  – issued by cache on write-back
  – (e.g., eviction)
• CWI: Coherent Write & Invalidate
  – issued by I/O processor (DMA) on block write
Snoopy Protocol: Cache to Cache

- CCI: Cache to Cache Intervention
  - get data from another cache (the “owner”)
  - much faster to get data from cache versus memory
Snoopy Protocol: Cache States

- **Invalid (I)**
  - block not present

- **Clean exclusive (CE)**
  - cached data consistent with memory
  - exclusively held by a single cache
  - cache has write permissions (but hasn’t yet written it)

- **Owned exclusive (OE)**
  - dirty copy (inconsistent with memory)
  - exclusively held by a single cache
  - cache has write permissions (and has written it)

- **Clean shared (CS)**
  - cache has read permission
  - can be inconsistent with memory... (but is clean, relative to the owner’s copy, who holds the data in the OS state)

- **Owned shared (OS)**
  - dirty copy (inconsistent with memory)
  - this cache is responsible for supplying future sharers the correct value
  - OS is entered only from OE
    - this core wrote the data (OE), but somebody else requests a copy, so it downgraded itself to OS
  - other caches who hold this block are in the CS state
Handout #6: Directory Protocols

• Assumptions:
  – Reliable network, FIFO message delivery between any given source-destination pair
Directory Protocol: Cache States

- C-Invalid (I)
  - block not present

- C-Shared (Sh)
  - read permissions
  - clean copy (consistent with memory)
  - multiple copies may exist

- C-modified (Ex)
  - cache has write permissions
  - dirty copy (inconsistent with memory)
  - exclusively held by a single cache

- C-transient (Pending)
  - waiting on messages, etc.
Directory Protocol: Home Directory States

- **R(dir)**
  - The memory block is shared by the sites specified in dir (dir is a set of sites). The data in memory is valid in this state. If dir is empty (i.e., dir = $\epsilon$), the memory block is not cached by any site.

- **W(id)**
  - The memory block is exclusively cached at site id, and has been modified at that site. Memory does not have the most up-to-date data.

- **TR(dir)**
  - The memory block is in a transient state waiting for the acknowledgements to the invalidation requests that the home site has issued.

- **TW(id)**
  - The memory block is in a transient state waiting for a block exclusively cached at site id (i.e., in C-modified state) to make the memory block at the home site up-to-date.
Directory Protocol: Messages

- **Cache to Memory Requests**
  - ShReq
    - read permission request
  - ExReq
    - write permission request

- **Memory to Cache Requests**
  - WbReq
    - write-back request (someone else wants to read your dirty copy)
  - InvReq
    - invalidate copy (someone else wants to write your copy)
  - FlushReq
    - flush copy back to memory (someone else wants to write your dirty copy)

- **Cache to Memory Responses**
  - WbReq
  - InvRep
  - FlusheRep

- **Memory to Cache Responses**
  - ShRep
    - response with data/permission to read data
  - ExRep
    - response with data/permission to write data
P5

- Why would we get a FlushReq while in the C-pending state?
  - we’re waiting on response
  - but we only get a FlushReq if the directory thinks we own the data (Ex)
  - therefore....
    - we must have voluntarily evicted the data, then decided to read or write it again, but the directory doesn’t know that yet

- what does the receiving cache do then?
  - drops FlushReq on the floor
  - Why does this work?
    - directory will eventually receive the voluntary WBRep, and everything gets handled correctly!
• P5
  – What goes wrong if the network is non-FIFO?

  – example:
    • $A$ sends read request (ShReq)
    • $B$ sends write request (ExReq)

  – What if....
    • invalidating a cache ($A$) who hasn’t received his data...
    • $A$ (C-pending) will drop InvReq on the floor, doesn’t respond to directory
    • directory never receives Ack from $A$ (InvRep)
• P5
  – What if the processor “silently drops” a cache line?
  – directory’s sharer list doesn’t get updated
  – a future invalidation will never Ack’ed
  – deadlock!
Lab 5
Lab 5

• Dual-core Rocket Processor
  – single issue, in-order cores
  – non-blocking caches
• Write multi-threaded vvadd, matrix-multiply (in C this time)
• snoopy cache coherence protocol
  – can switch between
    • MI (modified-invalid)
    • MSI (modified/shared/invalid)
• optimize matmul for BOTH protocols
Lab 5: Programming Environment

- no main()
  - called thread_entry(cid, nc)
    - core id, number of cores
    - both threads start executing thread_entry()
- now have printf!
- still using floating point (I lied in the last section)
  - however, printf doesn’t handle floating point (cast to int first)
Lab 5: Synchronization

- barrier()
  - cores wait at barrier until all reach it
  - prevents a core from running ahead
  - written using lower-level primitives
  - is *probably* all you need to write your code
    - (if you need to use a barrier at all...)
Lab 5: Synchronization

• tons of lower level stuff though!
  – RISC-V ISA has a FENCE instruction
    • __sync_synchronize()

• RISC-V gcc compiler supports the following build-ins which call the low-level RISC-V instructions
  – e.g., __sync_fetch_and_add()
  – no need for inline assembly

• probably don’t need these though!
Lab 5: Synchronization

- FENCE vs fence.v.l
- COMPLETELY DIFFERENT!!!!!!

- fence.v.l (lab 4)
  - CP stalls until vector-unit (VU) is empty
  - CP and VU share L1 cache... nothing enforces coherence!
  - requires fence.v.l to make sure store data is visible from one to the other

- FENCE (lab 5)
  - __sync_synchronize()
  - Rocket sends FENCE to cache
    - completes if no outstanding memory requests (i.e., misses)
    - stalls Rocket if cache does have outstanding requests
Lab 5: Warnings (Stacks, TLS, etc.)

- **No ISA Simulator** this time! (it doesn’t support multiple cores yet...)
- Each thread has its own stack
  - its very small!!! (~8kB)
  - no warning if you run off stack (no VM in this lab)
  - don’t allocate big arrays on the stack (use “static” keyword)
  - Also no OS (and thus no malloc)
  - allocate big arrays statically, in the binary
- Each thread has its own “thread local storage space”
  - __thread modifier
  - you will NOT need to use this (I’m just throwing this out there because you’ll see it in the support code)
  - very, very small...
  - no warning if you run out of space
  - initialized to zero (no matter what your code says!)
void thread_entry(int cid, int nc) {
  
  coreid = cid;
  ncores = nc;

  #define N 512
  static data_t x[N], y[N];
  size_t i;
  if (coreid == 0)
    for (i = 0; i < N; i++)
      y[i] = 2*(x[i] = i);
  barrier();

  stats(vvadd0(N, x, y); barrier());

  if (coreid == 0)
    verify(N, x, 3);
  barrier();

  stats(vvadd1(N, x, y); barrier());

  if (coreid == 0)
    verify(N, x, 5);
  barrier();

  exit(0);
}

(again, no more main... it's called “thread_entry” for Lab 5)

both threads start executing thread_entry at start-up

stats() is a cute-way to time the code inside the ()
Lab 5: The Coherence Protocols

- MI (Modify-invalid)
  - no sharing!
  - read & write requests get line in “Modify” state
  - have write permission if you possess the line (“M”)
- MSI (Modify-share-invalid)
  - both caches may hold line if in “Share” state (read-only)
  - read request -> Share state
  - write request -> Modify state
- **NO** cache-to-cache transfers
  - requests go to “cache coherence hub” (logically sits on the “bus”)
    - hub broadcasts request to all caches (sends out “probes”)
    - cache gives up its line if probe is a hit (performs writeback as necessary)
  - hub then lets request continue to main memory