Homework 7

- Optional, no due date
- Topics: Prefetching, multiprocessors, cache coherence

For your benefit:
- To reinforce your understanding of recent material
- To help you prepare for the final exam (April 6)
Homework 5 Grade Distribution

Average: 83%
Lab 7: Multi-Core Cache Coherence

- Due May 3
- Cycle-level modeling of the MESI cache coherence protocol
Final Exam: May 6

- May 6

- Comprehensive (over all topics in course)

- Three cheat sheets allowed

- We will have a review session (stay tuned)

- Remember this is 30% of your grade
  - I will take into account your improvement over the course
  - Know the previous midterm concepts by heart
Last Lecture

- Bottlenecks in parallel processing

- Multiprocessor correctness
  - Sequential consistency
  - Weaker consistency

- Cache coherence
  - Software vs. hardware
  - Update vs. invalidate
  - Snoopy cache vs. directory based
  - VI → MSI → MESI → MOESI
Today

- Wrap up cache coherence
- Interconnection networks
Readings: Multiprocessing

**Required**

**Recommended**
Readings: Cache Coherence

- **Required**
  - Culler and Singh, *Parallel Computer Architecture*
    - Chapter 5.1 (pp 269 – 283), Chapter 5.3 (pp 291 – 305)
  - P&H, *Computer Organization and Design*
    - Chapter 5.8 (pp 534 – 538 in 4th and 4th revised eds.)

- **Recommended:**
Cache Coherence
Review: MESI State Machine from Lab 7

[Diagram showing the MESI state machine with transitions between states: Invalid, Shared, Modified, Exclusive. The transitions include:
- Cache miss ( > 1 requester)
- Other cache has write-miss (invalidate)
- Write (mark dirty)
- Other cache has read-miss (downgrade)
- Write (upgrade and inval. others)
- Other cache has write-miss (invalidate)]
Review: Intel Pentium Pro

- Write Allocate
- L1 can have data not in L2
- Hit: Someone has it clean
- Hitm: Someone has it dirty
Review: The Problem with MESI

- Shared state requires the data to be clean
  - i.e., all caches that have the block have the up-to-date copy and so does the memory
- Problem: Need to write the block to memory when BusRd happens when the block is in Modified state

- Why is this a problem?
  - Memory can be updated unnecessarily → some other processor may write to the block while it is cached
Review: Improving on MESI

- Idea 1: Do not transition from M→S on a BusRd. Invalidate the copy and supply the modified block to the requesting processor directly without updating memory.

- Idea 2: Transition from M→S, but designate one cache as the owner (O), who will write the block back when it is evicted.
  - Now “Shared” means “Shared and potentially dirty”
  - This is a version of the MOESI protocol.
Tradeoffs in Sophisticated Cache Coherence Protocols

- The protocol can be optimized with more states and prediction mechanisms to
  + Reduce unnecessary invalidates and transfers of blocks

- However, more states and optimizations
  -- Are more difficult to design and verify (lead to more cases to take care of, race conditions)
  -- Provide diminishing returns
Revisiting Two Cache Coherence Methods

- How do we ensure that the proper caches are updated?

- **Snoopy Bus** [Goodman ISCA 1983, Papamarcos+ ISCA 1984]
  - Bus-based, single point of serialization for all requests
  - Processors observe other processors’ actions
    - E.g.: P1 makes “read-exclusive” request for A on bus, P0 sees this and invalidates its own copy of A

- **Directory** [Censier and Feautrier, IEEE ToC 1978]
  - Single point of serialization per block, distributed among nodes
  - Processors make explicit requests for blocks
  - Directory tracks ownership (sharer set) for each block
  - Directory coordinates invalidation appropriately
    - E.g.: P1 asks directory for exclusive copy, directory asks P0 to invalidate, waits for ACK, then responds to P1
Snoopy Cache vs. Directory Coherence

- **Snoopy Cache**
  + Critical path is short: miss $\rightarrow$ bus transaction to memory
  + Global serialization is easy: bus provides this already (arbitration)
  + Simple: adapt bus-based uniprocessors easily
    - Relies on broadcast messages to be seen by all caches:
      $\rightarrow$ single point of serialization (bus): *not scalable*

- **Directory**
  - Adds indirection to critical path: request $\rightarrow$ directory $\rightarrow$ mem
  - Requires extra storage space to track sharer sets
    - Can be approximate (false positives are OK)
  - Protocols and race conditions are more complex
  + Exactly as scalable as interconnect and directory storage
    *(much more scalable than bus)*
Revisiting Directory-Based Cache Coherence
Remember: Directory Based Coherence

- Idea: A logically-central directory keeps track of where the copies of each cache block reside. Caches consult this directory to ensure coherence.

- An example mechanism:
  - For each cache block in memory, store $P+1$ bits in directory
    - One bit for each cache, indicating whether the block is in cache
    - Exclusive bit: indicates that the cache that has the only copy of the block and can update it without notifying others
  - On a read: set the cache’s bit and arrange the supply of data
  - On a write: invalidate all caches that have the block and reset their bits
  - Have an “exclusive bit” associated with each block in each cache
Remember: Directory Based Coherence

Example directory based scheme

$p+1$ bits: for block $A$

$P_1$ takes a read miss to block $A$

$P_3$ takes a read miss
Directory-Based Protocols

- Required when scaling past the capacity of a single bus
- Distributed, *but*:
  - Coherence still requires single point of serialization (for write serialization)
  - Serialization location can be different for every block (striped across nodes)

- We can reason about the protocol for a single block: one *server* (directory node), many *clients* (private caches)

- Directory receives *Read* and *ReadEx* requests, and sends *Invl* requests: invalidation is explicit (as opposed to snoopy buses)
Key operation to support is *set inclusion test*

- False positives are OK: want to know which caches *may* contain a copy of a block, and spurious invalidations are ignored
- False positive rate determines *performance*

Most accurate (and expensive): full bit-vector

Compressed representation, linked list, Bloom filters are all possible
Directory: Basic Operations

- Follow *semantics* of snoop-based system
  - but with explicit request, reply messages

- Directory:
  - Receives *Read, ReadEx, Upgrade* requests from nodes
  - Sends *Inval/Downgrade* messages to sharers if needed
  - Forwards request to memory if needed
  - Replies to requestor and updates sharing state

- Protocol design is flexible
  - Exact forwarding paths depend on implementation
  - For example, do cache-to-cache transfer?
P0 acquires an address for reading:

1. Read

2. DatEx (DatShr)
RdEx with Former Owner

1. RdEx

P0  Home  Owner

2. Invl

3a. Rev

3b. DatEx
Contestation Resolution (for Write)

1a. RdEx
4. InvI
3. RdEx
2a. DatEx
5a. Rev

P0

Home

1b. RdEx
3. RdEx
2b. NACK
5b. DatEx

P1

5a. Rev
Issues with Contention Resolution

- Need to escape race conditions by:
  - NACKing requests to busy (pending invalidate) entries
    - Original requestor retries
  - OR, queuing requests and granting in sequence
  - (Or some combination thereof)

- Fairness
  - Which requestor should be preferred in a conflict?
  - Interconnect delivery order, and distance, both matter

- Ping-ponging is a higher-level issue
  - With solutions like combining trees (for locks/barriers) and better shared-data-structure design
Scaling the Directory: Some Questions

- How large is the directory?

- How can we reduce the access latency to the directory?

- How can we scale the system to thousands of nodes?
Interconnect Basics
Where Is Interconnect Used?

- To connect components

- Many examples
  - Processors and processors
  - Processors and memories (banks)
  - Processors and caches (banks)
  - Caches and caches
  - I/O devices

![Interconnection network diagram](image-url)
Why Is It Important?

- Affects the scalability of the system
  - How large of a system can you build?
  - How easily can you add more processors?

- Affects performance and energy efficiency
  - How fast can processors, caches, and memory communicate?
  - How long are the latencies to memory?
  - How much energy is spent on communication?
Interconnection Network Basics

- **Topology**
  - Specifies the way switches are wired
  - Affects routing, reliability, throughput, latency, building ease

- **Routing (algorithm)**
  - How does a message get from source to destination
  - Static or adaptive

- **Buffering and Flow Control**
  - What do we store within the network?
    - Entire packets, parts of packets, etc?
  - How do we throttle during oversubscription?
  - Tightly coupled with routing strategy
Topology

- Bus (simplest)
- Point-to-point connections (ideal and most costly)
- Crossbar (less costly)
- Ring
- Tree
- Omega
- Hypercube
- Mesh
- Torus
- Butterfly
- ...

Metrics to Evaluate Interconnect Topology

- Cost
- Latency (in hops, in nanoseconds)
- Contention

- Many others exist you should think about
  - Energy
  - Bandwidth
  - Overall system performance
Bus

+ Simple
+ Cost effective for a small number of nodes
+ Easy to implement coherence (snooping and serialization)
- Not scalable to large number of nodes (limited bandwidth, electrical loading → reduced frequency)
- High contention → fast saturation
Point-to-Point

Every node connected to every other

+ Lowest contention
+ Potentially lowest latency
+ Ideal, if cost is not an issue

-- Highest cost
  O(N) connections/ports per node
  O(N²) links

-- Not scalable
-- How to lay out on chip?
Crossbar

- Every node connected to every other (non-blocking) except one can be using the connection at any given time
- Enables concurrent sends to non-conflicting destinations
- Good for small number of nodes

+ Low latency and high throughput
- Expensive
- Not scalable $\rightarrow O(N^2)$ cost
- Difficult to arbitrate as $N$ increases

Used in core-to-cache-bank networks in
- IBM POWER5
- Sun Niagara I/II
Another Crossbar Design
Sun UltraSPARC T2 Core-to-Cache Crossbar

- High bandwidth interface between 8 cores and 8 L2 banks & NCU
- 4-stage pipeline: req, arbitration, selection, transmission
- 2-deep queue for each src/dest pair to hold data transfer request
Buffered Crossbar

- Simpler arbitration/scheduling
- Efficient support for variable-size packets
- Requires $N^2$ buffers
Can We Get Lower Cost than A Crossbar?

- Yet still have low contention?
- Idea: Multistage networks
Multistage Logarithmic Networks

- Idea: Indirect networks with multiple layers of switches between terminals/nodes
- Cost: $O(N\log N)$, Latency: $O(\log N)$
- Many variations (Omega, Butterfly, Benes, Banyan, ...)
- **Omega Network:**

![Omega Network Diagram](image-url)
Multistage Circuit Switched

- More restrictions on feasible concurrent Tx-Rx pairs
- But more scalable than crossbar in cost, e.g., $O(N \log N)$ for Butterfly
Packets “hop” from router to router, pending availability of the next-required switch and buffer.
Aside: Circuit vs. Packet Switching

- **Circuit switching** sets up full path
  - Establish route then send data
  - (no one else can use those links)
  + faster arbitration
  -- setting up and bringing down links takes time

- **Packet switching** routes per packet
  - Route each packet individually (possibly via different paths)
  - if link is free, any packet can use it
  -- potentially slower --- must dynamically switch
  + no setup, bring down time
  + more flexible, does not underutilize links
Switching vs. Topology

- Circuit/packet switching choice independent of topology
- It is a higher-level protocol on how a message gets sent to a destination

- However, some topologies are more amenable to circuit vs. packet switching
Another Example: Delta Network

- Single path from source to destination
- Does not support all possible permutations
- Proposed to replace costly crossbars as processor-memory interconnect
Another Example: Omega Network

- Single path from source to destination
- All stages are the same
- Used in NYU Ultracomputer
Ring

+ Cheap: $O(N)$ cost
- High latency: $O(N)$
- Not easy to scale
  - Bisection bandwidth remains constant

Used in Intel Haswell, Intel Larrabee, IBM Cell, many commercial systems today
Unidirectional Ring

- Simple topology and implementation
  - Reasonable performance if N and performance needs (bandwidth & latency) still moderately low
  - O(N) cost
  - N/2 average hops; latency depends on utilization
Mesh

- O(N) cost
- Average latency: O(sqrt(N))
- Easy to layout on-chip: regular and equal-length links
- Path diversity: many ways to get from one node to another

- Used in Tilera 100-core
- And many on-chip network prototypes
Torus

- Mesh is not symmetric on edges: performance very sensitive to placement of task on edge vs. middle
- Torus avoids this problem
  + Higher path diversity (and bisection bandwidth) than mesh
  - Higher cost
  - Harder to lay out on-chip
    - Unequal link lengths
Torus, continued

- Weave nodes to make inter-node latencies ~constant
Trees

Planar, hierarchical topology
Latency: $O(\log N)$
Good for local traffic
+ Cheap: $O(N)$ cost
+ Easy to Layout
- Root can become a bottleneck
  Fat trees avoid this problem (CM-5)
CM-5 Fat Tree

- Fat tree based on 4x2 switches
- Randomized routing on the way up
- Combining, multicast, reduction operators supported in hardware

![CM-5 Thinned Fat Tree](image_url)
Hypercube

- Latency: \(O(\log N)\)
- Radix: \(O(\log N)\)
- #links: \(O(N\log N)\)
+ Low latency
- Hard to lay out in 2D/3D
Caltech Cosmic Cube

- 64-node message passing machine
Handling Contention

- Two packets trying to use the same link at the same time
- What do you do?
  - Buffer one
  - Drop one
  - Misroute one (deflection)
- Tradeoffs?
Key idea: Packets are never buffered in the network. When two packets contend for the same link, one is deflected.

New traffic can be injected whenever there is a free output link.

Bufferless Deflection Routing

- Input buffers are eliminated: flits are buffered in **pipeline latches** and on **network links**

```
North      North
South      South
East       East
West       West
Local      Local
```

Input Buffers

Deflection Routing Logic
Routing Algorithm

- **Types**
  - **Deterministic**: always chooses the same path for a communicating source-destination pair
  - **Oblivious**: chooses different paths, without considering network state
  - **Adaptive**: can choose different paths, adapting to the state of the network

- **How to adapt**
  - Local/global feedback
  - Minimal or non-minimal paths
Deterministic Routing

- All packets between the same (source, dest) pair take the same path

- **Dimension-order routing**
  - E.g., XY routing (used in Cray T3D, and many on-chip networks)
  - First traverse dimension X, then traverse dimension Y

  + Simple
  + Deadlock freedom (no cycles in resource allocation)
  - Could lead to high contention
  - Does not exploit path diversity
Deadlock

- No forward progress
- Caused by circular dependencies on resources
- Each packet waits for a buffer occupied by another packet downstream
Handling Deadlock

- Avoid cycles in routing
  - Dimension order routing
    - Cannot build a circular dependency
  - Restrict the “turns” each packet can take

- Avoid deadlock by adding more buffering (escape paths)

- Detect and break deadlock
  - Preemption of buffers
Turn Model to Avoid Deadlock

- **Idea**
  - Analyze directions in which packets can turn in the network
  - Determine the cycles that such turns can form
  - Prohibit just enough turns to break possible cycles


![Fig. 2. The possible turns and simple cycles in a two-dimensional mesh.](image1)

![Fig. 3. The four turns allowed by the xy routing algorithm.](image2)

![Fig. 4. Six turns that complete the cycles and allow deadlock.](image3)
Oblivious Routing: Valiant’s Algorithm

- An example of oblivious algorithm
- Goal: Balance network load
- Idea: Randomly choose an intermediate destination, route to it first, then route from there to destination
  - Between source-intermediate and intermediate-dest, can use dimension order routing

+ Randomizes/balances network load
- Non minimal (packet latency can increase)

- Optimizations:
  - Do this on high load
  - Restrict the intermediate node to be close (in the same quadrant)
Adaptive Routing

- **Minimal adaptive**
  - Router uses network state (e.g., downstream buffer occupancy) to pick which “productive” output port to send a packet to
  - Productive output port: port that gets the packet closer to its destination
    - Aware of local congestion
    - Minimality restricts achievable link utilization (load balance)

- **Non-minimal (fully) adaptive**
  - “Misroute” packets to non-productive output ports based on network state
    - Can achieve better network utilization and load balance
    - Need to guarantee livelock freedom
On-Chip Networks

- Connect **cores, caches, memory controllers, etc**
  - Buses and crossbars are not scalable
- Packet switched
- **2D mesh**: Most commonly used topology
- Primarily serve **cache misses** and memory requests

**Diagram**

- **PE**: Processing Element (Cores, L2 Banks, Memory Controllers, etc)
- **R**: Router
Motivation for Efficient Interconnect

- In many-core chips, on-chip interconnect (NoC) consumes **significant power**
  - **Intel Terascale:** ~28% of chip power
  - **Intel SCC:** ~10%
  - **MIT RAW:** ~36%

- Recent work\(^1\) uses **bufferless deflection routing** to reduce power and die area