CS315A/EE386A: Lecture 9

Symmetric Multiprocessors II
Implementation Details

Kunle Olukotun
Stanford University

http://eeclass.stanford.edu/cs315a

Announcements

- PS2 due Monday May 8
  - no late day
- Midterm exam Wed May 10
  - 7-9pm Gates B03
  - Lectures 1-9
  - Open book, open notes, calculator, no computer
- Midterm review Friday May 5
  - Gates B01
  - 4:15-5:05pm
  - Broadcast live on E4
- PA2 due Mon May 15

Today's Outline

- SMP performance
- SMP Implementation detail

SMP Performance

- Cache coherence protocol
  - Update vs. invalidate
  - Bus bandwidth
- Memory hierarchy performance
  - Miss rate
  - Number of processors
  - Cache size
  - Block size
- Highly application dependent
  - Commercial
  - Scientific

Update versus Invalidate

- Much debate over the years: tradeoff depends on sharing patterns
- Intuition:
  - If reads and writes are interleaved, update should do better
    - e.g. producer-consumer pattern
  - If those that use unlikely to use again, or many writes between reads, updates not good
    - particularly bad under process migration
    - useless updates where only last one will be used
- Can construct scenarios where one or other is much better
- Can combine them in hybrid schemes
  - E.g. competitive: observe patterns at runtime and change protocol

Bus Traffic for Invalidate vs. Update

- Pattern 1:

```
for i = 1 to k
  P1(write, x);     // one write before reads
  P2-PN(read, x);
end for i
```

- Pattern 2:

```
for i = 1 to k
  for j = 1 to m
    P1(write, x);   // many writes before reads
  end for j
  P2(read, x);
end for i
```

Assume:

1. **Invalidation/upgrade = 6 bytes (5 addr, 1 cmd)**
2. **Update = 14 bytes (6 addr/cmd + 8 data)**
3. **Cache miss = 70 bytes (6 addr/cmd + 64 data)**

Bus Traffic for Invalidate vs. Update, cont.

- Pattern 1:

```
for i = 1 to k
   P1(write, x);
   P2-PN(read, x);
end for i
```

- Pattern 2:

```
for i = 1 to k
   for j = 1 to m
     P1(write, x);
   end for j
   P2(read, x);
end for i
```

- Pattern 1 (one write before reads)
  - N = 16, m = 10, k = 10
  - Update
    - Iteration 1: N regular cache misses (70 bytes)
    - Remaining iterations: update per iteration (14 bytes)
    - Total Update Traffic = 16*70 + 9*14 = 1246 bytes
  - Invalidate
    - Iteration 1: N regular cache misses (70 bytes)
    - Remaining: P1 generates upgrade (6), 15 others Read miss (70)
    - Total Invalidate Traffic = 16*70 + 9*6 + 15*9*70 = 10,624 bytes
- Pattern 2 (many writes before reads)
  - Update = 2*70 + 10*9*14 = 1400 bytes
  - Invalidate = 11*70 + 9*6 = 824 bytes

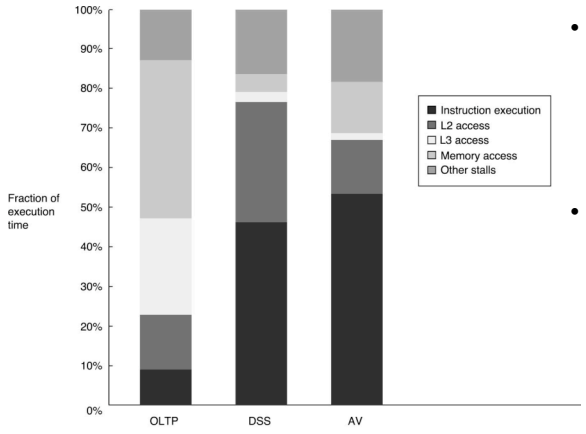
Invalidate vs. Update Reality

- What about real workloads?
  - Update can generate too much traffic
  - Must limit (e.g., competitive snooping)
- Current Assessment
  - Update very hard to implement correctly (consistency discussion coming next week)
  - Rarely done
- Future Assessment
  - May be same as current or
  - Chip multiprocessors may revive update protocols
    - More intra-chip bandwidth
    - Easier to have predictable timing paths?

Memory Hierarchy Performance

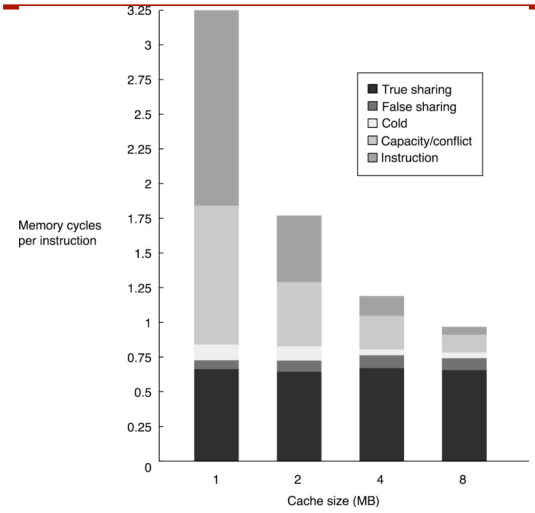
- Uniprocessor 3C's
  - (Compulsory, Capacity, Conflict)
- SM adds Coherence Miss Type (communication)
  - True Sharing miss fetches data written by another processor
  - False Sharing miss results from independent data in same coherence block
- Increasing cache size
  - Usually fewer capacity/conflict misses
  - No effect on true/false "coherence" misses (so may dominate)
- Block size is unit of transfer and of coherence
  - Doesn't have to be, could make coherence smaller
- Increasing block size
  - Usually fewer 3C misses but more bandwidth
  - Usually more false sharing misses

Commercial Application Performance on a 4-Proc AlphaServer



- Alphaserver 4100
  - L1: 8KB D.M.
  - L2: 96KB 3-way S.A.
  - L3: 2MB D.M.
- Performance
  - OLTP: 7.0 CPI
  - DSS: 1.6 CPI
  - AltaVista: 1.3

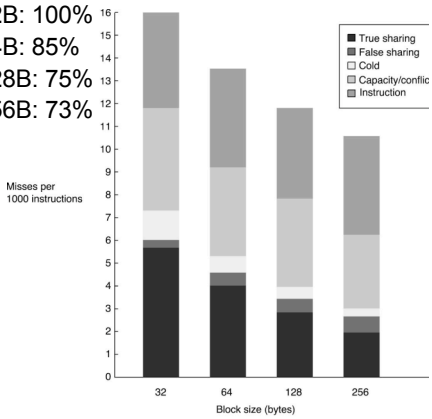
OLTP Memory Performance

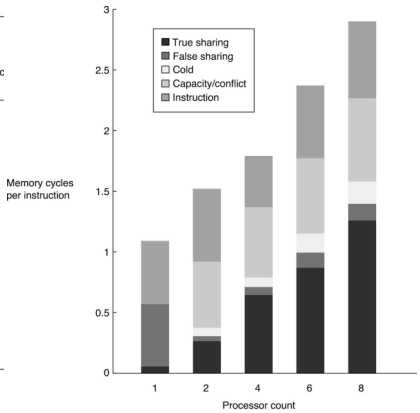


- 2-way S.A.
- Dominant sources
  - Instruction
  - Capacity/conflict
- Increasing cache size
  - Reduces 1-proc misses
  - Sharing misses remain

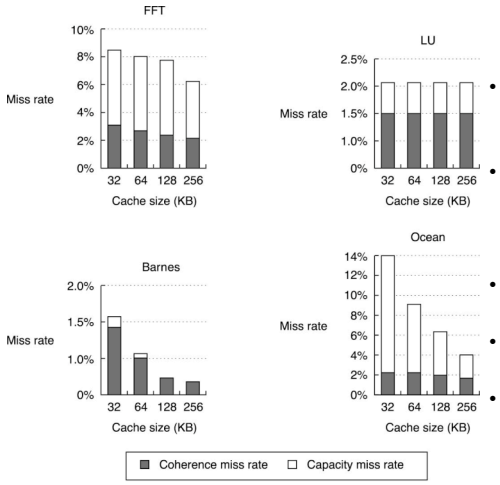
Block Size and Processor Count Effect on OLTP Memory Performance

- Miss rate reduction in 2 MB, 2-way S.A.
  - 32B: 100%
  - 64B: 85%
  - 128B: 75%
  - 256B: 73%



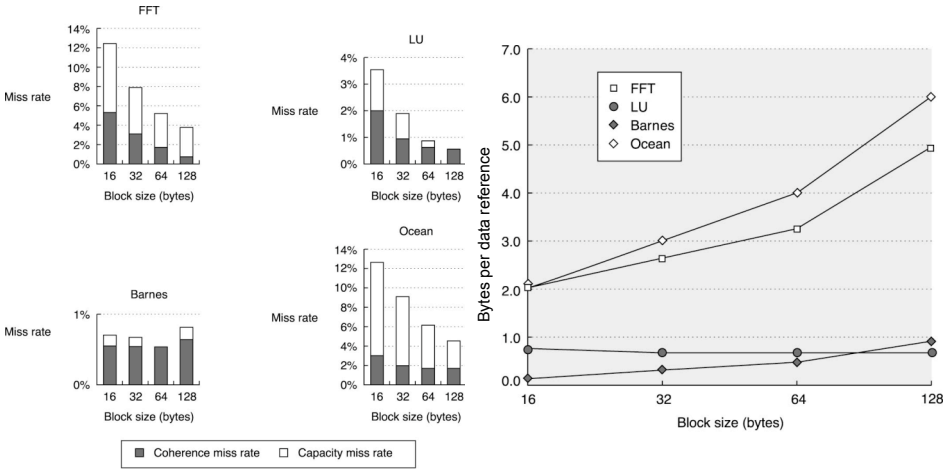


Scientific App. Cache Size vs. Miss rate



- Arch parameters
  - 16 processors
  - 32B block size
- FFT
  - Benefit limited due to . . .
  - Communication for transpose
  - Working set effects
- LU
  - Next WS is both matrices
- Barnes
  - Comm *and* capacity reduced
- Ocean
  - Large impact on capacity

Scientific App. Block Size vs. Miss rate and Buss Traffic (16 proc, 64 KB cache)

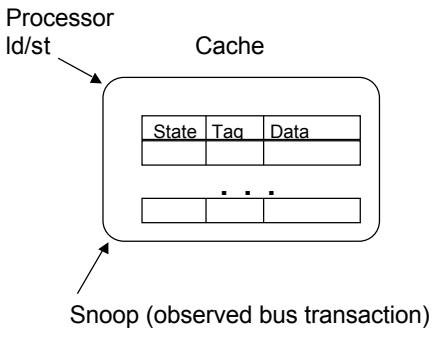


Review: Symmetric Multiprocessors (SMP)

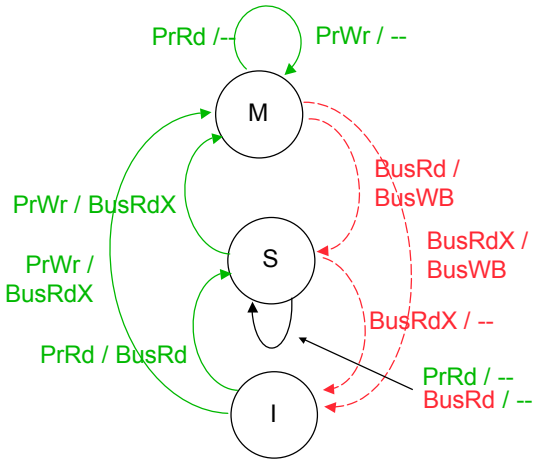
- Multiple microprocessors
- Each has cache hierarchy
- Connect with logical bus (totally-ordered broadcast)
- Implement Snooping Cache Coherence Protocol
  - Broadcast all cache "misses" on bus
  - All caches "snoop" bus and may act
  - Memory responds otherwise
- Performance
  - OLTP requires large caches (≥ 4 MB)
  - OLTP performance limited by sharing misses
  - Scientific apps show working set effects, smaller caches (≤ 0.5 MB)
  - Optimized scientific apps don't share much

Snooping Cache-Coherence Protocols

- Bus provides serialization point
- Each cache controller "snoops" all bus transactions
  - Controller updates state of blocks in response to processor and snoop events and generates bus transactions
- Snoopy protocol
  - set of states
  - state-transition diagram
  - actions
- Basic Choices
  - write-through vs. write-back
  - invalidate vs. update



MSI State Diagram



Abbreviation	Action
PrRd	Processor Read
PrWr	Processor Write
BusRd	Bus Read
BusRdX	Bus Read Exclusive
BusWB	Bus Writeback

Unanswered Questions

- How does memory know another cache will respond so it need not?
- What do we do if a cache miss is not an atomic event (check tags, queue for bus, get bus, etc.)?
- What about L1/L2 caches & split transactions buses?
- Is deadlock a problem?
- What happens on a PTE update with multiple TLBs?

Outline

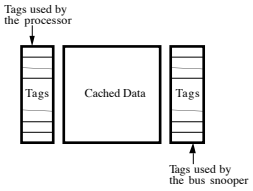
- Coherence control implementation
- Writebacks & Non-Atomicity
- Cache hierarchies
- Split buses
- Deadlock, livelock, & starvation
- A case study
- TLB coherence

Base Cache Coherence Design

- Single-level write-back cache
- Invalidation protocol
- One outstanding memory request per processor
- Atomic memory bus transactions
  - no interleaving of transactions
- Atomic operations within process
  - one finishes before next in program order
- Examine snooping,write serialization, and atomicity
- Then add more concurrency and re-examine

Cache Tags

- Cache controller must monitor bus and processor
  - Can view as two controllers: bus-side, and processor-side
  - With single-level cache: dual tags (not data) or dual-ported tag RAM
    - must reconcile when updated, but usually only looked up
  - Respond to bus transactions



Reporting Snoop Results: How?

- Collective response from caches must appear on bus
- Example: in MESI protocol, need to know
  - Is block dirty; i.e. should memory respond or not?
  - Is block shared; i.e. transition to E or S state on read miss?
- Three wired-OR signals
  - Shared: asserted if any cache has a copy
  - Dirty: asserted if some cache has a dirty copy
    - needn't know which, since it will do what's necessary
  - Snoop-valid: asserted when OK to check other two signals
    - actually inhibit until OK to check
- Illinois MESI requires priority scheme for cache-to-cache transfers
  - Which cache should supply data when in shared state?
  - Commercial implementations allow memory to provide data

Reporting Snoop Results: When?

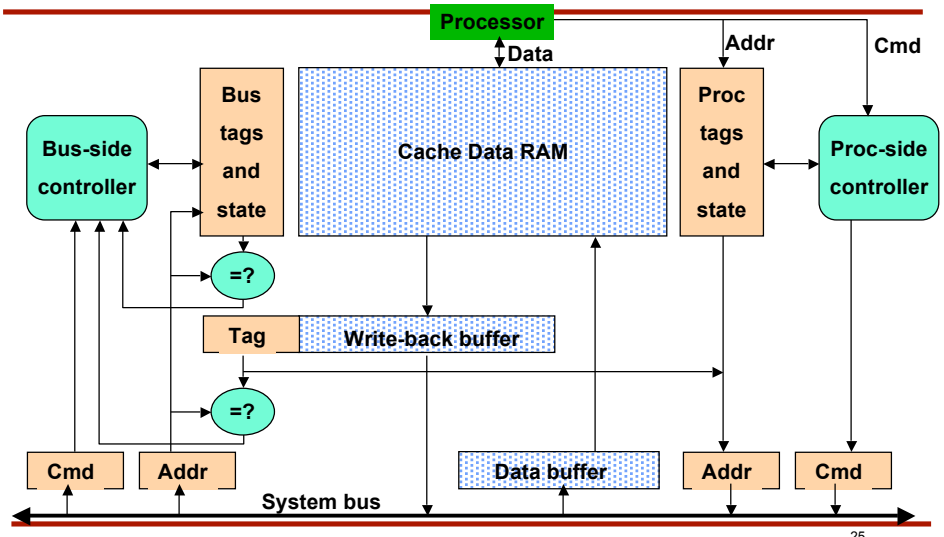
- Memory needs to know what, if anything, to do
- Fixed number of clocks from address appearing on bus
  - Dual tags required to reduce contention with processor
  - Still must be conservative (tags inaccessible on write: S → M)
  - Pentium, HP servers, Sun Enterprise
- Variable delay
  - Memory assumes cache will supply data till all say “sorry”
  - Less conservative, more flexible, more complex
  - Memory can fetch data and hold just in case (SGI Challenge)
- Immediately
  - Bit-per-block in memory
  - Extra hardware complexity in commodity main memory system

Writebacks

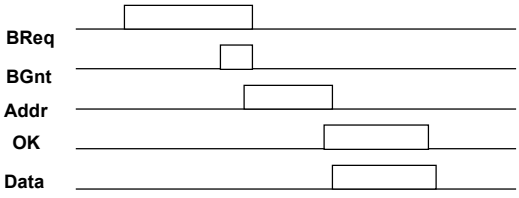
- Write back block in M state
- Must allow processor to proceed on a miss
  - fetch the block
  - perform writeback later

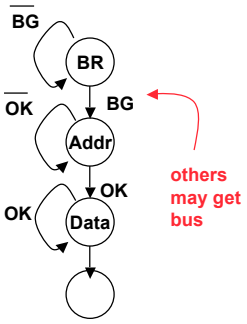
- Need a writebuffer
  - Must handle bus transactions in the write buffer
  - Check writebuffer on snoop, if hit supply data and cancel writeback

Snooping Cache



Typical Bus Protocol





- On a miss processor must:
  - Assert request for bus
  - Wait for bus grant
  - Drive address and command lines
  - Wait for command to be accepted by relevant device
  - Transfer data

Non-Atomic State Transitions

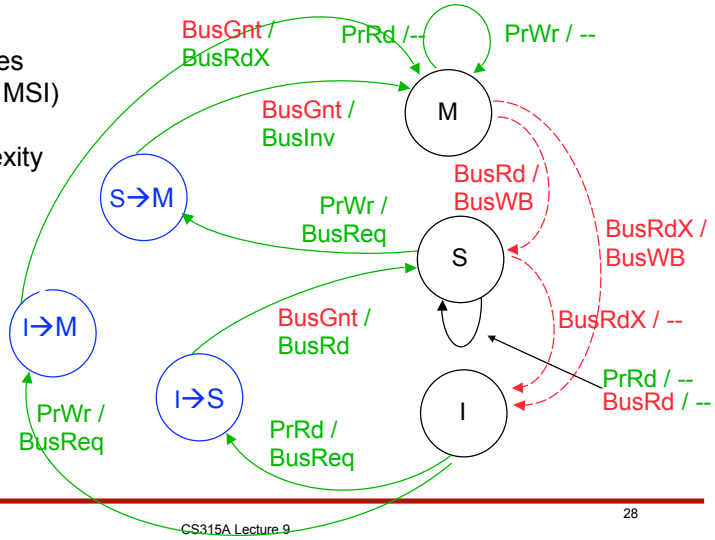
- Memory operations involve multiple actions
  - Look up cache tags
  - Arbitrate for bus
  - Check for writeback
  - Even if bus is atomic, overall set of actions is not
  - Race conditions among multiple operations
- Suppose P1 and P2 attempt to write cached block A
  - Each decides to issue BusInv to allow S → M
- The cache controller must
  - Handle requests for other blocks while waiting to acquire bus
  - Handle requests for this block A

Non-Atomicity → Transient States

Extend protocol
Two types of states
- Stable (e.g. MSI)
- Transient

Increases complexity

Action	Abbr.
Bus Request	BusReq
Bus Grant	BusGnt



Multi-level Cache Hierarchies

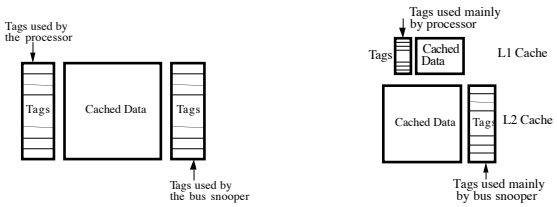
- How to snoop with multi-level caches?
  - independent bus snooping at every level?
  - maintain cache inclusion
- Requirements for Inclusion
  - data in higher-level is superset of data in lower-level
  - modified in lower-level → marked modified in higher-level
- Now only need to snoop highest-level cache
  - If L2 says not present, then not so in L1
- Is inclusion automatically preserved?
  - Natural if higher-level is larger, low-level is DM but same block size
  - Maintaining inclusion can be tricky (Baer and Wang 1988)

Inclusion to be or not to be

- Most common inclusion solution
  - Ensure L2 holds superset of L1I and L1D
  - On L2 replacement or coherence request that must source data or invalidate, forward actions to L1 caches
  - L2 cache with inclusion often removes the need for dual tags (next slide)
- But
  - Restricted associativity in unified L2 can limit blocks in split L1's
  - CMPs make inclusion expensive
    - Total size of L1s maybe comparable to L2
  - Not that hard to always snoop L1's
- Thus, many new designs don't maintain inclusion

Contention of Cache Tags

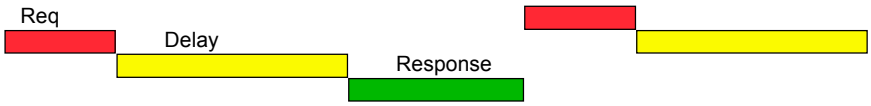
- L2 filter reduces contention on L1 tags



Split-transaction (Pipelined) Bus

- Supports multiple simultaneous transactions (many designs)

Atomic Transaction Bus



Split-transcation Bus

- Typically two separate buses with tagged transactions
  - Request : address and command
  - Response: data

Potential Problems

- New request can appear on bus before previous one serviced
  - Even before snoop result obtained
  - P1 and P2 both try to write block A which is invalid in both caches
  - PI issues BusRdX, P2 in invalid state so no response
  - P2 issues BusRdX, P1 in invalid state so no response
  - P1 gets memory response and places block in modified state
  - P2 gets memory response and places block in modified state
  - Disaster! Memory is incoherent
- Buffer requests and responses
  - Need flow control to prevent deadlock from limited buffering
- Ordering of Snoop responses
  - when does snoop response appear wrt data response

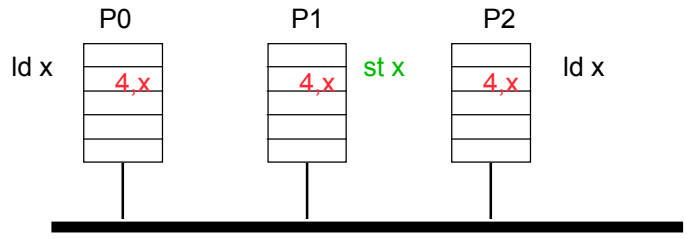
One Solution

- Disallow conflicting transactions
  - All processors can see outstanding transactions
  - P2 won't issue BusRdX for block A if it sees P1's request
- NACK for flow control
- Out-of-order responses
  - snoop results presented with data response

A Split-transaction Bus Design

- 4 Buses + Flow Control and Snoop Results
  - Command (type of transaction)
  - Address
  - Tag (unique identifier for response)
  - Data (doesn't require address)
- Form of transactions
  - BusRd, BusRdX (request + response)
  - Writeback (request + data)
  - Invalidate (request only)
- Per processor request table tracks all transactions

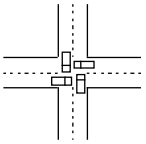
A Simple Example

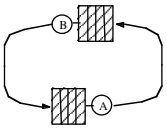


P2 can snarf data from first ld
P1 must hold st operation until entry is clear

Protocol Correctness

- Protocol must maintain coherence and consistency
- Protocol implementation should prevent:
- Deadlock:
  - all system activity ceases
  - Cycle of resource dependences





- Livelock:
  - no processor makes forward progress
  - constant on-going transactions at hardware level
  - e.g. simultaneous writes in invalidation-based protocol
- Starvation:
  - some processors make no forward progress
  - e.g. a processor always loses bus arbitration

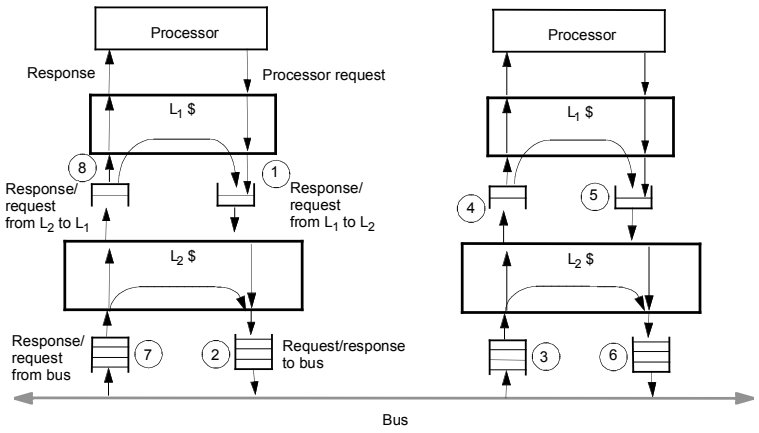
Deadlock, Livelock, Starvation

- Request-reply protocols can lead to fetch *deadlock*
  - When issuing requests, must service incoming transactions
  - e.g. cache awaiting bus grant must snoop & writeback blocks
  - else may not respond to request that will release bus: deadlock
- Livelock:
  - Many processors want to write same line
  - Invalidation happens between obtaining ownership & write
  - Ownership changes but no processor actually writes data
  - Solution: don't let ownership be stolen before write
- Starvation:
  - solve by using fair arbitration on bus and FIFO buffers

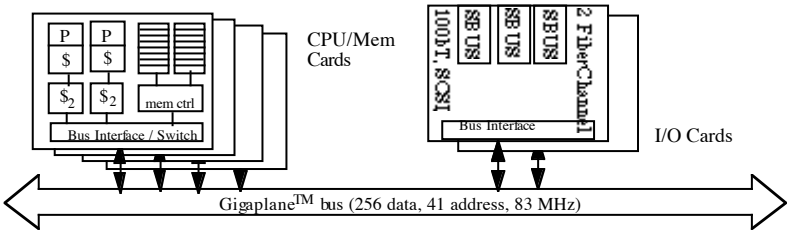
Multi-level Caches with Split-Transaction Bus

- General structure uses queues between
  - Bus and L2 cache
  - L2 cache and L1 cache
- How do you avoid Deadlock?
- Classify all transactions
  - Request, only generates responses
  - Response, doesn't generate any other transactions
- Requestor guarantees space for all responses
- Use Separate Request and Response queues
- Responses are never delayed by requests waiting for a response
- Responses are guaranteed to be serviced
- Requests will eventually be serviced since the number of responses is bounded by outstanding requests

Multi-Level Caches with Split Bus



SUN Enterprise 6000 Overview

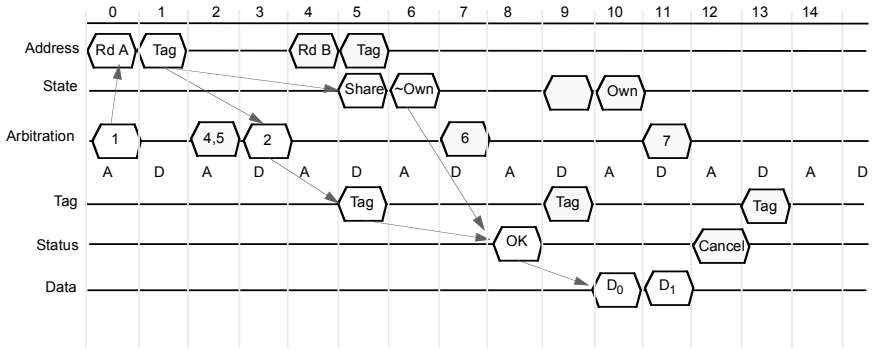


- Up to 30 UltraSPARC processors, MOESI protocol
- Gigaplane™ bus has peak bw 2.67 GB/s, 300 ns latency
- Up to 112 outstanding transactions (max 7 per board)
- 16 bus slots, for processing or I/O boards
  - 2 CPUs and 1GB memory per board
    - memory distributed, but protocol treats as centralized (UMA)

Sun Gigaplane Bus

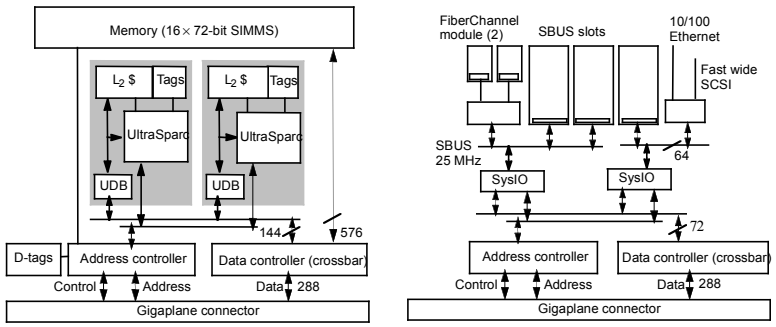
- Non-multiplexed, split-transaction, 256-data/41-address, 83.5 MHz (Plus 32 ECC lines, 7 tag, 18 arbitration, etc. Total 388)
- Cards plug in on both sides: 8 per side
- 112 outstanding transactions, up to 7 from each board
  - Designed for multiple outstanding transactions per processor
- Emphasis on reducing latency
  - Speculative arbitration if address bus not scheduled from prev. cycle
  - Else regular 1-cycle arbitration, and 7-bit tag assigned in next cycle
- Snoop result associated with request (5 cycles later)
- Main memory can stake claim to data bus 3 cycles into this, and start memory access speculatively
  - Two cycles later, asserts tag bus to inform others of coming transfer
- MOESI protocol
  - Owned state says this processor instead of memory will provide data

Gigaplane Bus Timing



Enterprise Processor and Memory System

- 2 procs / board, ext. L2 caches, 2 mem banks w/ x-bar
- Data lines buffered through UDB to drive internal 1.3 GB/s UPA bus
- Wide path to memory so full 64-byte line in 2 bus cycles

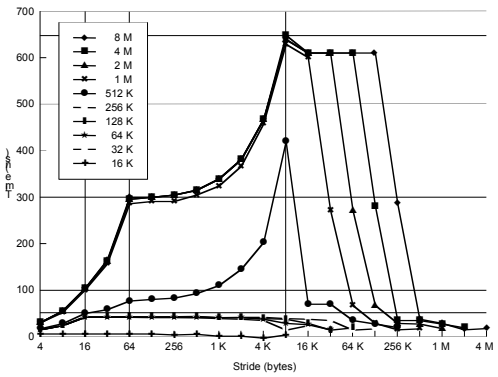


Enterprise I/O System

- I/O board has same bus interface ASICs as processor boards
- But internal bus half as wide, and no memory path
- Only cache block sized transactions, like processing boards
  - Uniformity simplifies design
  - ASICs implement single-block cache, follows coherence protocol
- Two independent 64-bit, 25 MHz Sbuses
  - One for two dedicated FiberChannel modules connected to disk
  - One for Ethernet and fast wide SCSI
  - Can also support three SBUS interface cards for arbitrary peripherals
- Performance and cost of I/O scale with no. of I/O boards

Memory Access Latency

- 300ns read miss latency (130 ns on bus)
- Rest is path through caches & the DRAM access
- TLB misses add 340 ns



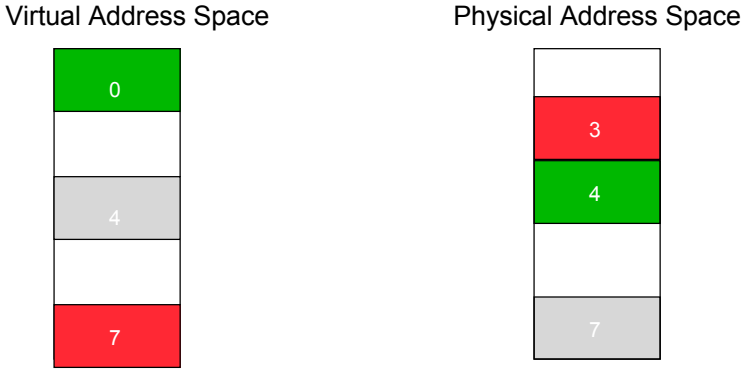
Ping-pong microbenchmark is 1.7 µs round-trip (5 mem accesses)

Sun Enterprise 10000

- How far can you go with snooping coherence?
- Quadruple request/snoop bandwidth using four address busses
  - each handles 1/4 of physical address space
  - impose *logical* ordering for consistency: for writes on same cycle, those on bus 0 occur "before" bus 1, etc.
- Get rid of data bandwidth problem: use a network
  - E10000 uses 16x16 crossbar betw. CPU boards & memory boards
  - Each CPU board has up to 4 CPUs: max 64 CPUs total
- 10.7 GB/s max BW, 468 ns unloaded miss latency
- See "Starfire: Extending the SMP Envelope", IEEE Micro, Jan/Feb 1998

Translation Lookaside Buffer

- Cache of Page Table Entries
- Page Table Maps Virtual Page to Physical Frame



The TLB Coherence Problem

- Since TLB is a cache, must be kept coherent
- Change of PTE on one processor must be seen by all processors
- Process migration
- Changes are infrequent
  - get OS to do it
  - Always flush TLB is often adequate

TLB Shootdown

- To modify TLB entry, modifying processor must
  - LOCK page table,
  - flush TLB entries,
  - queue TLB operations,
  - send interprocessor interrupt,
  - spin until other processors are done
  - UNLOCK page table

- SLOW...
  - But most common solution today
- Some ISAs have “flush TLB entry” instructions