ECE/CS 757: Advanced Computer Architecture II

Instructor:Mikko H Lipasti

Spring 2009
University of Wisconsin-Madison

Lecture notes based on slides created by John Shen, Mark Hill, David Wood, Guri Sohi, and Jim Smith, Natalie Enright Jerger, and probably others

Cache Coherence

- Coherence States
- Snoopy bus-based Invalidate Protocols
- Invalidate protocol optimizations
- Update Protocols (Dragon/Firefly)
- Directory protocols
- Implementation issues

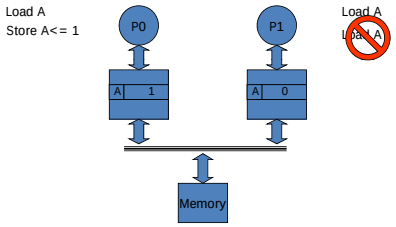
2

Readings

- Readings:
  - Firefly
  - Archibald
  - Sweazey/Smith
  - Laudon/Lenoski: Origin 2000
  - Opteron
  - Gigaplane
  - Power5
  - Intel 870

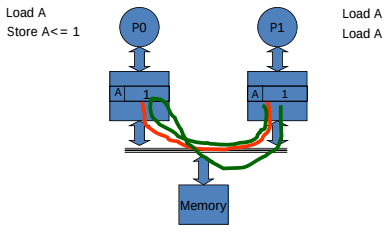
3

Cache Coherence Problem



4

Cache Coherence Problem



5

Possible Causes of Incoherence

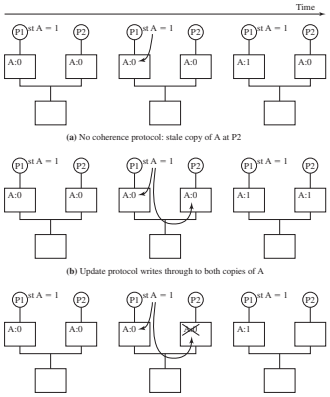
- Sharing of writeable data
  - Cause most commonly considered
- Process migration
  - Can occur even if independent jobs are executing
- I/O
  - Often fixed via O/S cache flushes

Cache Coherence

- Informally, with coherent caches: accesses to a memory location *appear* to occur simultaneously in all copies of the memory location
  - "copies" ⇒ caches
- Cache coherence suggests an absolute time scale -- this is not necessary
  - What is required is the "appearance" of coherence... not absolute coherence
  - E.g. temporary incoherence between memory and a write-back cache may be OK.

Update vs. Invalidation Protocols

- Coherent Shared Memory
  - All processors see the effects of others' writes
- How/when writes are propagated
  - Determine by coherence protocol



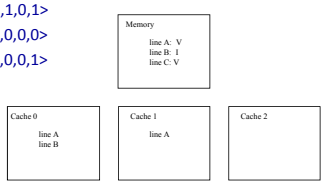
Global Coherence States

- A memory line can be present (valid) in any of the caches and/or memory
- Represent global state with an N+1 element vector
  - First N components => cache states (valid/invalid)
  - N+1st component => memory state (valid/invalid)
- Example:

  Line A: <1,1,0,1>

  Line B: <1,0,0,0>

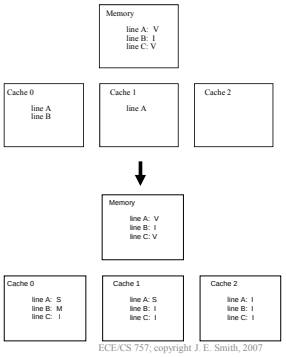
  Line C: <0,0,0,1>



Local Coherence States

- Individual caches can maintain a summary of the state of memory lines, from a "local" perspective
  - Reduces storage for maintaining state
  - May have only partial information
- Invalid (I): <0,X,X,X....X> -- local cache does not have a valid copy; (cache miss)
  - Don't confuse invalid state with empty frame
- Shared (S): <1,X,X,X,...,1> -- local cache has a valid copy, main memory has a valid copy, other caches ??
- Modified(M): <1,0,0,..0,...0> -- local cache has only valid copy.
- Exclusive(E): <1,0,0,..0,...1> -- local cache has a valid copy, no other caches do, main memory has a valid copy.
- Owned(O): <1,X,X,X,....X> -- local cache has a valid copy, all other caches and memory may have a valid copy.
  - Only one cache can be in O state
  - <1,X,1,X,... 0> is included in O, but not included in any of the others.

Example

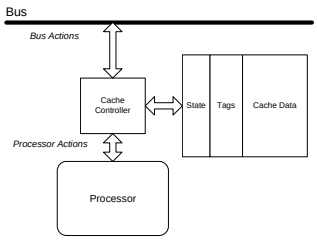


Snoopy Cache Coherence

- All requests broadcast on bus
- All processors and memory snoop and respond
- Cache blocks writeable at one processor or read-only at several
  - Single-writer protocol
- Snoops that hit dirty lines?
  - Flush modified data out of cache
  - Either write back to memory, then satisfy remote miss from memory, or
  - Provide dirty data directly to requestor
  - Big problem in MP systems
    - Dirty/coherence/sharing misses

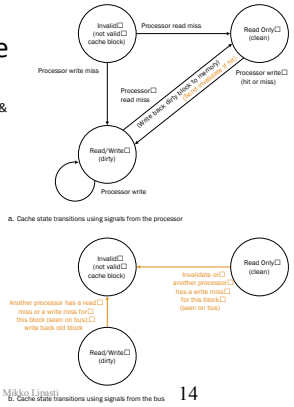
Bus-Based Protocols

- Protocol consists of states and actions (state transitions)
- Actions can be invoked from processor or bus



Minimal Coherence Protocol FSM

[Source: Patterson/Hennessy, Comp. Org. & Design]



MSI Protocol

	Action and Next State					
Current State	*Processor Read*	*Processor Write*	*Eviction*	*Cache Read*	*Cache Read&M*	*Cache Upgrade*
I	*Cache Read* Acquire Copy → S	*Cache Read&M* Acquire Copy → M		No Action → I	No Action → I	No Action → I
S	No Action → S	*Cache Upgrade* → M	No Action → I	No Action → S	*Invalidate Frame* → I	*Invalidate Frame* → I
M	No Action → M	No Action → M	*Cache Write back* → I	Memory inhibit; Supply data; → S	*Invalidate Frame;* Memory inhibit; Supply data; → I	

MSI Example

Thread Event	Bus Action	Data From	Global State	Local States: C0	C1	C2
0. Initially:			<0,0,0,1>	I	I	I
1. T0 read→	CR	Memory	<1,0,0,1>	S	I	I
2. T0 write→	CU		<1,0,0,0>	M	I	I
3. T2 read→	CR	C0	<1,0,1,1>	S	I	S
4. T1 write→	CRM	Memory	<0,1,0,0>	I	M	I

- If line is in no cache
  - Read, modify, Write requires 2 bus transactions
  - Optimization: add **E**xclusive state

Invalidate Protocol Optimizations

- Observation: data can be read shared
  - Add S (shared) state to protocol: MSI
- State transitions:
  - Local read: I->S, fetch shared
  - Local write: I->M, fetch modified; S->M, invalidate other copies
  - Remote read: M->S, supply data
  - Remote write: M->I, supply data; S->I, invalidate local copy
- Observation: data can be write-private (e.g. stack frame)
  - Avoid invalidate messages in that case
  - Add E (exclusive) state to protocol: MESI
- State transitions:
  - Local read: I->E if only copy, I->S if other copies exist
  - Local write: E->M silently, S->M, invalidate other copies

MESI Protocol

- Variation used in Pentium Pro/2/3
  - (cache to cache transfer)
- 4-State Protocol
  - **M**odified: <1,0,0…0>
  - **E**xclusive: <1,0,0,…,1>
  - **S**hared: <1,X,X,…,1>
  - **I**nvalid: <0,X,X,…X>
- Bus/Processor Actions
  - Same as MSI
- Adds *shared* signal to indicate if other caches have a copy

MESI Protocol

	Action and Next State					
Current State	***Processor Read***	***Processor Write***	***Eviction***	***Cache Read***	***Cache Read&M***	***Cache Upgrade***
I	*Cache Read* If no sharers: → E If sharers: → S	*Cache Read&M* → M		No Action → I	No Action → I	No Action → I
S	No Action → S	*Cache Upgrade* → M	No Action → I	Respond Shared: → S	No Action → I	No Action → I
E	No Action → E	No Action → M	No Action → I	Respond Shared; → S	No Action → I	
M	No Action → M	No Action → M	*Cache Write-back* → I	Respond dirty; Write back data; → S	Respond dirty; Write back data; → I	

MESI Example

- Optimization – cache-to-cache transfer for shared data (but only one sharer can respond)
- If modified in some cache and another reads then writeback to memory (w/ snarf)
  - Optimization: add **O**wned state (and perform cache-to-cache transfer)

Thread Event	Bus Action	Data From	Global State	Local States: C0	C1	C2
0. Initially:			<0,0,0,1>	I	I	I
1. T0 read→	CR	Memory	<1,0,0,1>	E	I	I
2. T0 write→	none		<1,0,0,0>	M	I	I

MOESI Optimization

- Observation: shared ownership complicates/delays sourcing data
  - Owner is responsible for sourcing data to any requestor
  - Add O (owner) state to protocol: MOSI/MOESI
  - Last requestor becomes the owner
  - Ownership can be on per-node basis in hierarchically structured system
  - Avoid writeback (to memory) of dirty data
  - Also called *shared-dirty* state, since memory is stale

MOESI Protocol

- Used in AMD Opteron
- 5-State Protocol
  - **M**odified: <1,0,0…0>
  - **E**xclusive: <1,0,0,…,1>
  - **S**hared: <1,X,X,…,X>
  - **I**nvalid: <0,X,X,…X>
  - **O**wned: <1,X,X,X,X> ; only one owner
- Owner can supply data, so memory does not have to
  - Avoids lengthy memory access

MOESI Protocol

	Action and Next State					
Current State	***Processor Read***	***Processor Write***	***Eviction***	***Cache Read***	***Cache Read&M***	***Cache Upgrade***
I	***Cache Read*** If no sharers: → E If sharers: → S	***Cache Read&M*** → M		No Action → I	No Action → I	No Action → I
S	No Action → S	***Cache Upgrade*** → M	No Action → I	Respond shared; → S	No Action → I	No Action → I
E	No Action → E	No Action → M	No Action → I	Respond shared; Supply data; → S	Respond shared; Supply data; → I	
O	No Action → O	***Cache Upgrade*** → M	***Cache Write-back*** → I	Respond shared; Supply data; → O	Respond shared; Supply data; → I	
M	No Action → M	No Action → M	***Cache Write-back*** → I	Respond shared; Supply data; → O	Respond shared; Supply data; → I	

MOESI Example

Thread Event	Bus Action	Data From	Global State	local states C0	C1	C2
0. Initially:			<0,0,0,1>	I	I	I
1. T0 read→	CR	Memory	<1,0,0,1>	E	I	I
2. T0 write→	none		<1,0,0,0>	M	I	I
3. T2 read→	CR	C0	<1,0,1,0>	O	I	S
4. T1 write→	CRM	C0	<0,1,0,0>	I	M	I

Update Protocols

- Basic idea:
  - All writes (updates) are made visible to all caches:
    - (address,value) tuples sent "everywhere"
    - Similar to write-through protocol for uniprocessor caches
  - Obviously not scalable beyond a few processors
  - No one actually builds machines this way
- Simple optimization
  - Send updates to memory/directory
  - Directory propagates updates to all known copies: less bandwidth
- Further optimizations: combine & delay
  - Write-combining of adjacent updates (if consistency model allows)
  - Send write-combined data
  - Delay sending write-combined data until requested
- Logical end result
  - Writes are combined into larger units, updates are delayed until needed
  - Effectively the same as invalidate protocol

Update Protocol: Dragon

- Dragon (developed at Xerox PARC)
- 5-State Protocol
  - **I**nvalid:<0,X,X,…X>
    - Some say no invalid state – due to confusion regarding empty frame versus invalid line state
  - **E**xclusive: <1,0,0,…,1>
  - **S**hared-**C**lean (Sc): <1,X,X,…X> memory may not be up-to-date
  - **S**hared-**M**odified (Sm): <1,X,X,X…0> memory not up-to-date; only one copy in Sm
  - **M**odified: <1,0,0,…0>
- Includes Cache Update action
- Includes Cache Writeback action
- Bus includes Shared flag
  - Appears to also require memory inhibit signal
  - Distinguish shared case where cache (not memory) supplies data

Dragon State Diagram

	Action and Next State					
Current State	***Processor Read***	***Processor Write***	***Eviction***		***Cache Read***	***Cache Update***
I	*Cache Read* If no sharers: → E If sharers: → Sc	*Cache Read* If no sharers: → M If sharers: *Cache Update* → Sm			→ I	→ I
Sc	No Action → Sc	*Cache Update* If no sharers: →M If sharers: → Sm	No Action → I		Respond Shared; → Sc	Respond shared; Update copy; → Sc
E	No Action → E	No Action → M	No Action → I		Respond shared; Supply data → Sc	
Sm	No Action → Sm	*Cache Update* If no sharers: →M If sharers: → Sm	*Cache Write-back* → I		Respond shared; Supply data; → Sm	Respond shared; Update copy; → Sc
M	No Action → M	No Action → M	*Cache Write-back* → I		Respond shared; Supply data; → Sm	

Example

Thread Event	Bus Action	Data From	Global State	local states C0	C1	C2
0. Initially:			<0,0,0,1>	I	I	I
1. T0 read→	CR	Memory	<1,0,0,1>	E	I	I
2. T0 write→	none		<1,0,0,0>	M	I	I
3. T2 read→	CR	C0	<1,0,1,0>	Sm	I	Sc
4. T1 write→	CR,CU	C0	<1,1,1,0>	Sc	Sm	Sc
5. T0 read→	none (hit)	C0	<1,1,1,0>	Sc	Sm	Sc

- Appears to require atomic bus cycles CR,CU on write to invalid line

Update Protocol: Firefly

- Develped at DEC by ex-Xerox people
- 5-State Protocol
  - Similar to Dragon – different state naming based on shared/exclusive and clean/dirty
  - **I**nvalid:<0,X,X,…X>
  - **EC**: <1,0,0,…,1>
  - **SC**: <1,X,X,…X> memory may not be up-to-date
  - **EM**: <1,0,0,…0>
  - **SM**: <1,X,X,X…0> memory not up-to-date; only one copy in Sm
- Performs write-through updates (different from Dragon)

Firefly State Diagram

	Action and Next State					
Current State	***Processor Read***	***Processor Write***	***Eviction***		***Cache Read***	***Cache Read&M***
I	*Cache Read* If no sharers: → Ec If sharers: → Sc	*Cache Read* If no sharers: → Em If sharers: *Cache Update* → Sm			No Action → I	No Action → I
Sc	No Action → Sc	*Cache Read&M* If no sharers: →*Ec* If sharers: → Sc	No Action → I		Respond Shared; → Sc	Respond Shared → Sc
Ec	No Action → Ec	No Action → Em	No Action → I		Respond shared; → Sc	Respond Shared → Sc
Sm	No Action → Sm	*Cache Read&M* If no sharers: →Ec If sharers: → Sc	*Cache Write-back* → I		Respond shared; Supply data; → Sm	Respond shared; Supply data; → Sc
Em	No Action → Em	No Action → Em	*Cache Write-back* → I		Respond shared; Supply data; → Sm	Respond shared; Supply data; → Sc

Update vs Invalidate

- [Weber & Gupta, ASPLOS3]
  - Consider sharing patterns
- No Sharing
  - Independent threads
  - Coherence due to thread migration
  - Update protocol performs many wasteful updates
- Read-Only
  - No significant coherence issues; most protocols work well
- Migratory Objects
  - Manipulated by one processor at a time
  - Often protected by a lock
  - Usually a write causes only a single invalidation
  - E state useful for Read-modify-Write patterns
  - Update protocol could proliferate copies

Update vs Invalidate, contd.

- Synchronization Objects
  - Locks
  - Update could reduce spin traffic invalidations
  - Test&Test&Set w/ invalidate protocol would work well
- Many Readers, One Writer
  - Update protocol may work well, but writes are relatively rare
- Many Writers/Readers
  - Invalidate probably works better
  - Update will proliferate copies
- What is used today?
  - Invalidate is dominant
  - CMP *may* change this assessment
    - more on-chip bandwidth

Nasty Realities

- State diagram is for (ideal) protocol assuming instantaneous and actions
- In reality controller implements more complex diagrams
  - A protocol state transition may be started by controller when bus activity changes local state
  - Example: an upgrade pending (for bus) when an invalidate for same line arrives

Partial Implementation State Table

	Action and Next State						
Current State	*Processor Read*	*Processor Write*	*Bus Grant*	*Bus Response*	*Cache Read*	*Cache Read&M*	*Cache Upgrade*
I	Request Bus → IR	Request Bus → IW			No Action → I	No Action → I	No Action → I
S	No Action → S	Request Bus → SW			Respond Shared: → S	No Action → I	No Action → I
E	No Action → E	No Action → M			Respond Shared; → S	No Action → I	
M	No Action → M	No Action → M			Respond dirty; Write back data; → S	Respond dirty; Write back data; → I	
IR			*Cache Read* → IRR				
IW			*Cache Read&M* → IWR				
IRR				If no sharers: → E If sharers: → S Load line			
IWR				→ M Load line			
SW			*Cache Upgrade* → M		Respond Shared: → SW	No Action → IW	No Action → IW

Further Optimizations

- Observation: Shared blocks should only be fetched from memory once
  - If I find a shared block on chip, forward the block
  - Problem: multiple shared blocks possible, who forwards?
    - Everyone? Power/bandwidth wasted
  - Single forwarder, but who?
    - Last one to receive block: F state
    - I->F for requestor, F->S for forwarder
  - What if F block is evicted?
    - Favor F blocks in replacement?
    - Don't allow silent eviction (force some other node to be F)
    - Fall back on memory copy if can't find F copy
- Very old idea (IBM machines have done this for a long time), but recent Intel patent issued anyway [Hum/Goodman]

Further Optimizations

- Observation: migratory data often "flies by"
  - Add T (transition) state to protocol
  - Tag is still valid, data isn't
  - Data can be snarfed as it flies by
  - Only works with certain kinds of interconnect networks
  - Replacement policy issues
- Many other optimizations are possible
  - Literature extends 25 years back
  - Many unpublished (but implemented) techniques as well

Implementing Cache Coherence

- Snooping implementation
  - Origins in shared-memory-bus systems
  - All CPUs could observe all other CPUs requests on the bus; hence "snooping"
    - Bus Read, Bus Write, Bus Upgrade
  - React appropriately to snooped commands
    - Invalidate shared copies
    - Provide up-to-date copies of dirty lines
      - Flush (writeback) to memory, or
      - Direct intervention (*modified intervention* or *dirty miss*)
- Snooping suffers from:
  - Scalability: shared busses not practical
  - Ordering of requests without a shared bus
  - Lots of recent and on-going work on scaling snoop-based systems

Snooping Cache Coherence

- Basic idea: broadcast snoop to all caches to find owner
- Not scalable?
  - Address traffic roughly proportional to square of number of processors
  - Current implementations scale to 64/128-way (Sun/IBM) with multiple address-interleaved broadcast networks
- Inbound snoop bandwidth: big problem

$$OutboundSnoopRate = s_o = \langle CacheMissRate \rangle + \langle BusUpgradeRate \rangle$$

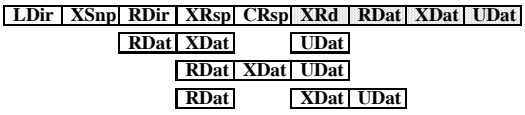
$$InboundSnoopRate = s_i = n \times s_o$$

Snoop Bandwidth

- Snoop filtering of various kinds is possible
- Filter snoops at sink: Jetty filter [Moshovos et al., HPCA 2001]
  - Check small "filter cache" that summarizes contents of local cache
  - Avoid power-hungry lookups in each tag array
- Filter snoops at source: Multicast snooping [Bilir et al., ISCA 1999]
  - Predict likely sharing set, snoop only predicted sharers
  - Double-check at directory to make sure
- Filter snoops at source: Region coherence
  - Concurrent work: [Cantin/Smith/Lipasti, ISCA 2005; Moshovos, ISCA 2005]
  - Check larger region of memory on every snoop; remember when no sharers
  - Snoop only on first reference to region, or when region is shared
  - Eliminate 60%+ of all snoops

Snoop Latency

- Snoop latency:
  - Must reach all nodes, return and combine responses
  - Probably the bigger scalability issue in the future
  - Topology matters: ring, mesh, torus, hypercube
  - No obvious solutions
- Parallelism: fundamental advantage of snooping
  - Broadcast exposes parallelism, enables speculative latency reduction



Scaleable Cache Coherence

- Eschew physical bus but still snoop
  - Point-to-point tree structure (indirect) or ring
  - Root of tree or ring provide ordering point
  - Use some scalable network for data (ordering less important)
- Or, use level of indirection through directory
  - Directory at memory remembers:
    - Which processor is "single writer"
    - Which processors are "shared readers"
  - Level of indirection has a price
    - Dirty misses require 3 hops instead of two
      - Snoop: Requestor->Owner
      - Directory: Requestor->Directory->Owner

Implementing Cache Coherence

- Directory implementation
  - Extra bits stored in memory (directory) record state of line
  - Memory controller maintains coherence based on the current state
  - Other CPUs' commands are not snooped, instead:
    - Directory forwards relevant commands
  - Powerful filtering effect: only observe commands that you need to observe
  - Meanwhile, bandwidth at directory scales by adding memory controllers as you increase size of the system
    - Leads to very scalable designs (100s to 1000s of CPUs)
- Directory shortcomings
  - Indirection through directory has latency penalty
  - If shared line is dirty in other CPU's cache, directory must forward request, adding latency
  - This can severely impact performance of applications with heavy sharing (e.g. relational databases)

Directory Protocol Implementation

- Basic idea: Centralized directory keeps track of data location(s)
- Scalable
  - Address traffic roughly proportional to number of processors
  - Directory & traffic can be distributed with memory banks (interleaved)
  - Directory cost (SRAM) or latency (DRAM) can be prohibitive
- Presence bits track sharers
  - Full map (N processors, N bits): cost/scalability
  - Limited map (limits number of sharers)
  - Coarse map (identifies board/node/cluster; must use broadcast)
- Vectors track sharers
  - Point to shared copies
  - Fixed number, linked lists (SCI), caches chained together
  - Latency vs. cost vs. scalability

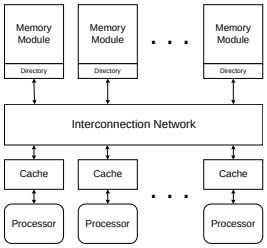
Directory Protocol Latency

LDir	XSnp	RDir	XRd	RDat	XDat	UDat

- Access to non-shared data
  - Overlap directory read with data read
  - Best possible latency given NUMA arrangement
- Access to shared data
  - Dirty miss, modified intervention
  - Shared intervention?
    - If DRAM directory, no gain
    - If directory cache, possible gain (use F state)
  - No inherent parallelism
  - Indirection adds latency
  - Minimum 3 hops, often 4 hops

Directory-based Cache Coherence

- An alternative for large, scalable MPs
- Can be based on any of the protocols discussed thus far
  - We will use MSI
- Memory Controller becomes an active participant
- Sharing info held in memory directory
  - Directory may be distributed
- Use point-to-point messages
- Network is not totally ordered

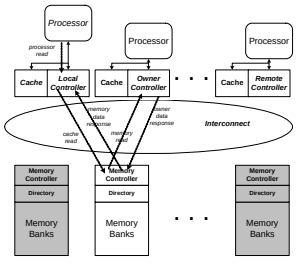


Example: Simple Directory Protocol

- Local cache controller states
  - M, S, I as before
- Local directory states
  - **S**hared: <1,X,X,…1>; one or more proc. has copy; memory is up-to-date
  - **M**odified: <0,1,0,….,0> one processor has copy; memory does not have a valid copy
  - **U**ncached: <0,0,…0,1> none of the processors has a valid copy
- Directory also keeps track of sharers
  - Can keep global state vector in full
  - e.g. via a bit vector

Example

- *Local cache* suffers load miss
- Line in *remote cache* in M state
  - It is the *owner*
- Four messages send over network
  - Cache read from local controller to home memory controller
  - Memory read to remote cache controller
  - Owner data back to memory controller; change state to S
  - Memory data back to local cache; change state to S



Cache Controller State Diagram

- Intermediate States for clarity

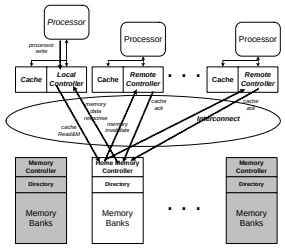
	Cache Controller Actions and Next States							
	from Processor Side			from Memory Side				
Current State	*Processor Read*	*Processor Write*	*Eviction*	*Memory Read*	*Memory Read&M*	*Memory Invalidate*	*Memory Upgrade*	*Memory Data*
I	*Cache Read* → I'	*Cache Read&M* → I''				No Action → I		
S	No Action → S	*Cache Upgrade* → S'	No Action* → I			Invalidate Frame; *Cache ACK*; → I		
M	No Action → M	No Action → M	*Cache Write-back* → I	*Owner Data;* → S	*Owner Data;* → I	Invalidate Frame; *Cache ACK*; → I		
I'								Fill Cache → S
I''								Fill Cache → M
S'							No Action → M	

Memory Controller State Diagram

	Memory Controller Actions and Next States					
	command from Local Cache Controller			response from Remote Cache Controller		
Current Directory State	*Cache Read*	*Cache Read&M*	*Cache Upgrade*	*Data Write-back*	*Cache ACK*	*Owner Data*
U	*Memory Data*; Add Requestor to Sharers; → S	*Memory Data*; Add Requestor to Sharers; → M				
S	*Memory Data*; Add Requestor to Sharers; → S	*Memory Invalidate* All Sharers; → M'	*Memory Upgrade* All Sharers; → M''	No Action → I		
M	*Memory Read* from Owner; → S'	*Memory Read&M*; to Owner → M'		Make Sharers Empty; → U		
S'						*Memory Data* to Requestor; Write memory; Add Requestor to Sharers; → S
M'					When all ACKS *Memory Data*; → M	*Memory Data* to Requestor; → M
M''					When all ACKS then → M	

Another Example

- Local write (miss) to shared line
- Requires invalidations and acks



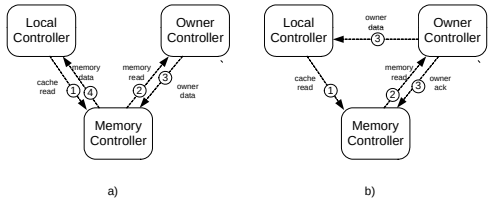
Example Sequence

- Similar to earlier sequences

Thread Event	Controller Actions	Data From	global state	local states: C0	C1	C2
0. Initially:			<0,0,0,1>	I	I	I
1. T0 read→	CR,MD	Memory	<1,0,0,1>	S	I	I
2. T0 write→	CU, MU*,MD		<1,0,0,0>	M	I	I
3. T2 read→	CR,MR,MD	C0	<1,0,1,1>	S	I	S
4. T1 write→	CRM,MI,CA,MD	Memory	<0,1,0,0>	I	M	I

Variation: Three Hop Protocol

- Have owner send data directly to local controller
- Owner Acks to Memory Controller in parallel



Directory Protocol Optimizations

- Remove dead blocks from cache:
  - Eliminate 3- or 4-hop latency
  - Dynamic Self-Invalidation [Lebeck/Wood, ISCA 1995]
  - Last touch prediction [Lai/Falsafi, ISCA 2000]
  - Dead block prediction [Lai/Fide/Falsafi, ISCA 2001]
- Predict sharers
  - Prediction in coherence protocols [Mukherjee/Hill, ISCA 1998]
  - Instruction-based prediction [Kaxiras/Goodman, ISCA 1999]
  - Sharing prediction [Lai/Falsafi, ISCA 1999]
- Hybrid snooping/directory protocols
  - Improve latency by snooping, conserve bandwidth with directory
  - Multicast snooping [Bilir et al., ISCA 1999; Martin et al., ISCA 2003]
  - Bandwidth-adaptive hybrid [Martin et al., HPCA 2002]
  - Token Coherence [Martin et al., ISCA 2003]
  - VCT Multicasting [Enright Jerger thesis 2008]

Protocol Races

- Atomic bus
  - Only stable states in protocol (e.g. M, S, I)
  - All state transitions are atomic (I->M)
  - No conflicting requests can interfere since bus is held till transaction completes
    - Distinguish coherence transaction from data transfer
    - Data transfer can still occur much later; easier to handle this case
- Atomic buses don't scale
  - At minimum, separate bus request/response
- Large systems have broadly variable delays
  - Req/resp separated by dozens of cycles
  - Conflicting requests can and do get issued
  - Messages may get reordered in the interconnect
- How do we resolve them?

Resolving Protocol Races

- Req/resp decoupling introduces transient states
  - E.g. I->S is now I->ItoX->ItoS_nodata->S
- Conflicting requests to blocks in transient states
  - NAK – ugly; livelock, starvation potential
  - Keep adding more transient states …
- Directory protocol makes this a bit easier
  - Can order at directory, which has full state info
  - Even so, messages may get reordered

55

Common Protocol Races

- Read strings: P0 read, P1 read, P2 read
  - Easy, since read is nondestructive
  - Can rely on F state to reduce DRAM accesses
  - Forward reads to previous requestor (F)
- Write strings: P0 write, P1 write, P2 write
  - Forward P1 write req to P0 (M)
  - P0 completes write then forwards M block to P1
  - Build string of writes (write string forwarding)
- Read after write (similar to prev. WAW)
- Writeback race: P0 evicts dirty block, P1 reads
  - Dirty block is in the network (no copy at P0 or at dir)
  - NAK P1, or force P0 to keep copy till dir ACKs WB
- Many others crop up, esp. with optimizations

56

Summary

- Coherence States
- Snoopy bus-based Invalidate Protocols
- Invalidate protocol optimizations
- Update Protocols (Dragon/Firefly)
- Directory protocols
- Implementation issues

57