18-600 Foundations of Computer Systems

Lecture 21: "Multicore Cache Coherence"

John P. Shen & Zhiyi Yu

November 14, 2016

Prevalence of multicore processors:

- 2006: 75% for desktops, 85% for servers
- 2007: 90% for desktops and mobiles, 100% for servers
- Today: 100% multicore processors with core counts ranging from 2 to 8 cores for desktops and mobiles, 8+ for servers

➢ Recommended Reference:

- "Parallel Computer Organization and Design," by Michel Dubois, Murali Annavaram, Per Stenstrom, Chapters 5 and 7, 2012.

Electrical & Computer ENGINEERING

Carnegie Mellon University

18-600 Foundations of Computer Systems

Lecture 21: "Multicore Cache Coherence"

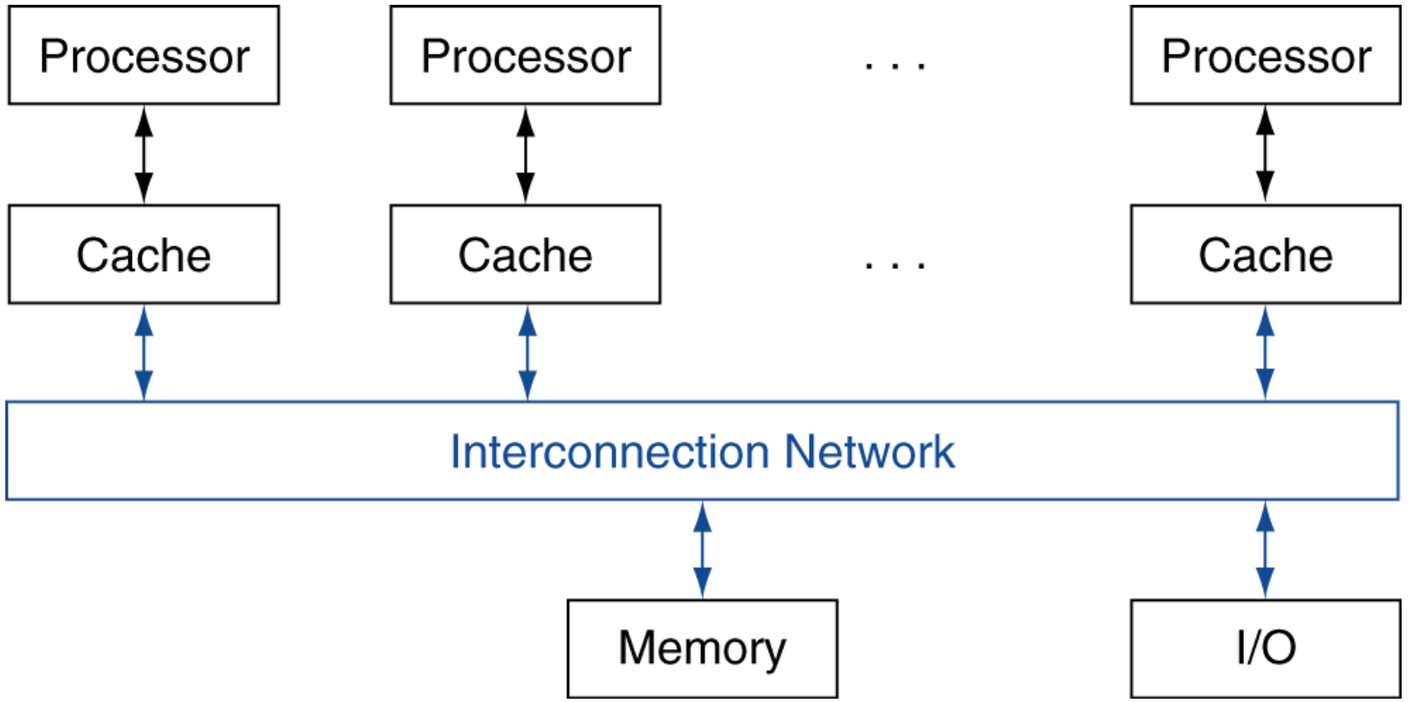
A. Cache Coherence Problem
B. Cache Coherence Protocols
   - Write Update
   - Write Invalidate
C. Bus-Based Snoopy Protocols
   - VI & MI Protocols
   - MSI, MESI, MOESI Protocols
D. Directory-Based Protocols

Electrical & Computer ENGINEERING

Shared-Memory Multiprocessors/Multicores

- All processor cores have access to unified physical memory
  - They can communicate via the shared memory using loads and stores
- Advantages
  - Supports multi-threading (TLP) using multiple cores
  - Requires relatively simple changes to the OS for scheduling
  - Threads within an app can communicate implicitly without using OS
    - Simpler to code for and lower overhead
  - App development: first focus on correctness, then on performance
- Disadvantages
  - Implicit communication is hard to optimize
  - Synchronization can get tricky
  - Higher hardware complexity for cache management

Shared Memory Multiprocessors/Multicores



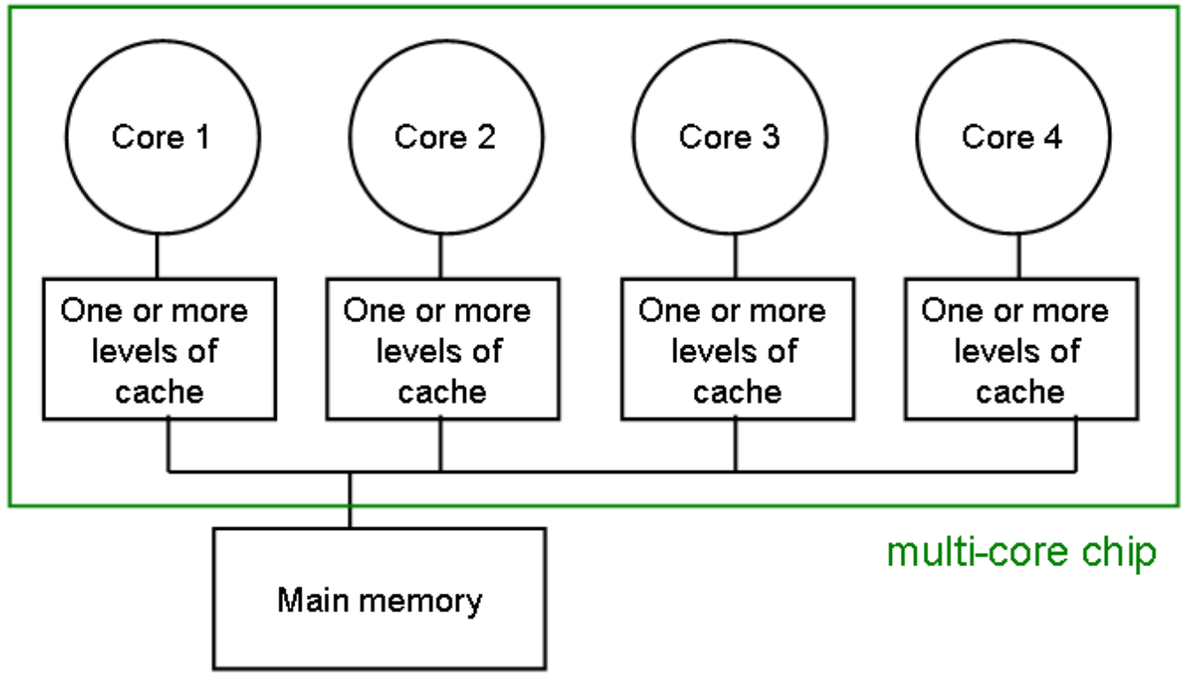
- Caches are (equally) helpful with multicores
  - Reduce access latency, reduce bandwidth requirements
  - For both private and shared data across cores
- But caches introduce the problems of coherence & consistency

Private vs. Shared Caches

- Advantages of private:
  - They are closer to core, so faster access
  - Reduces contention
- Advantages of shared:
  - Threads on different cores can share the same cache data
  - More cache space available if a single (or a few) high-performance thread runs on the system

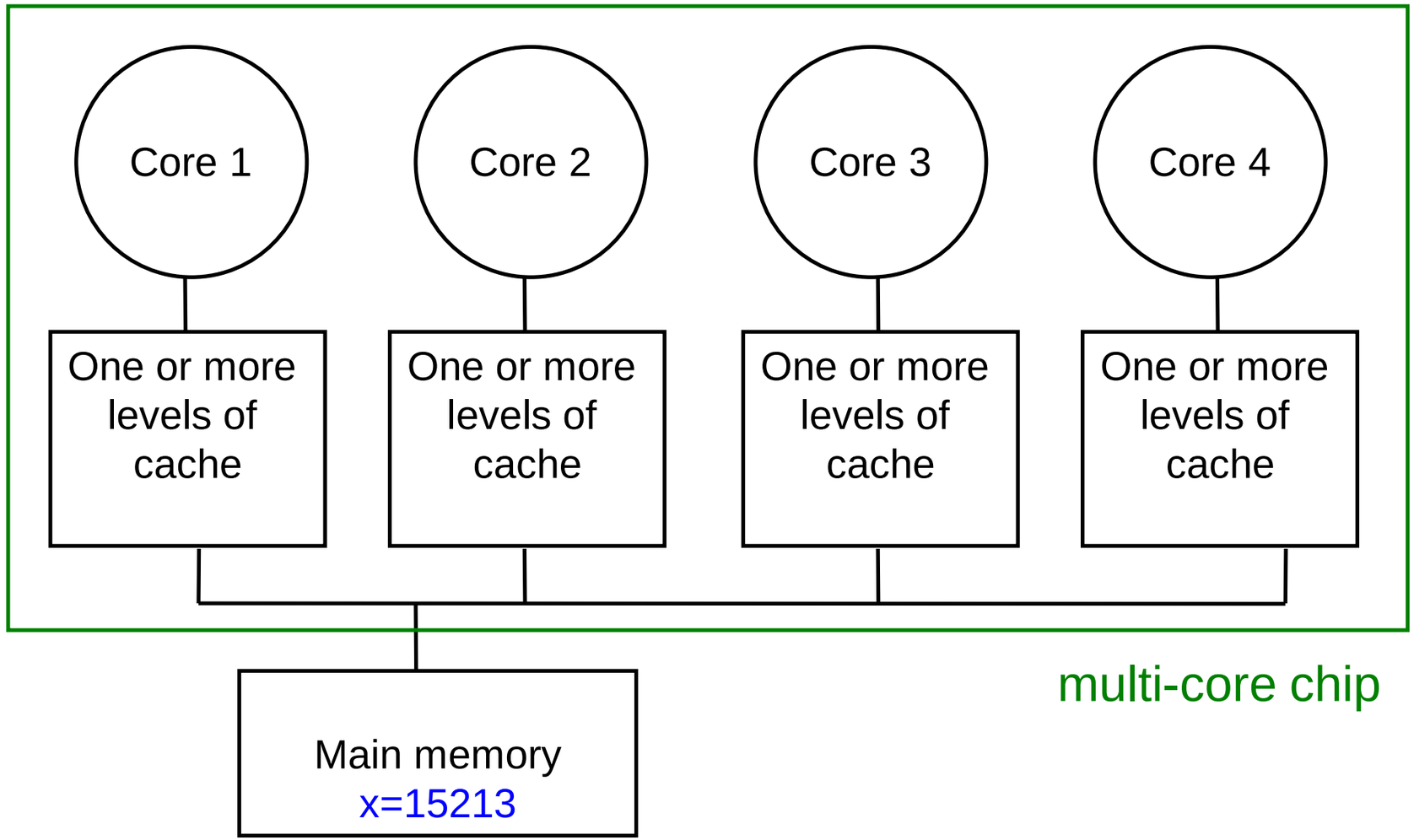
The Cache Coherence Problem

- Since we have private caches:
  How to keep the data consistent across caches?
- Each core should perceive the memory as a monolithic array, shared by all the cores



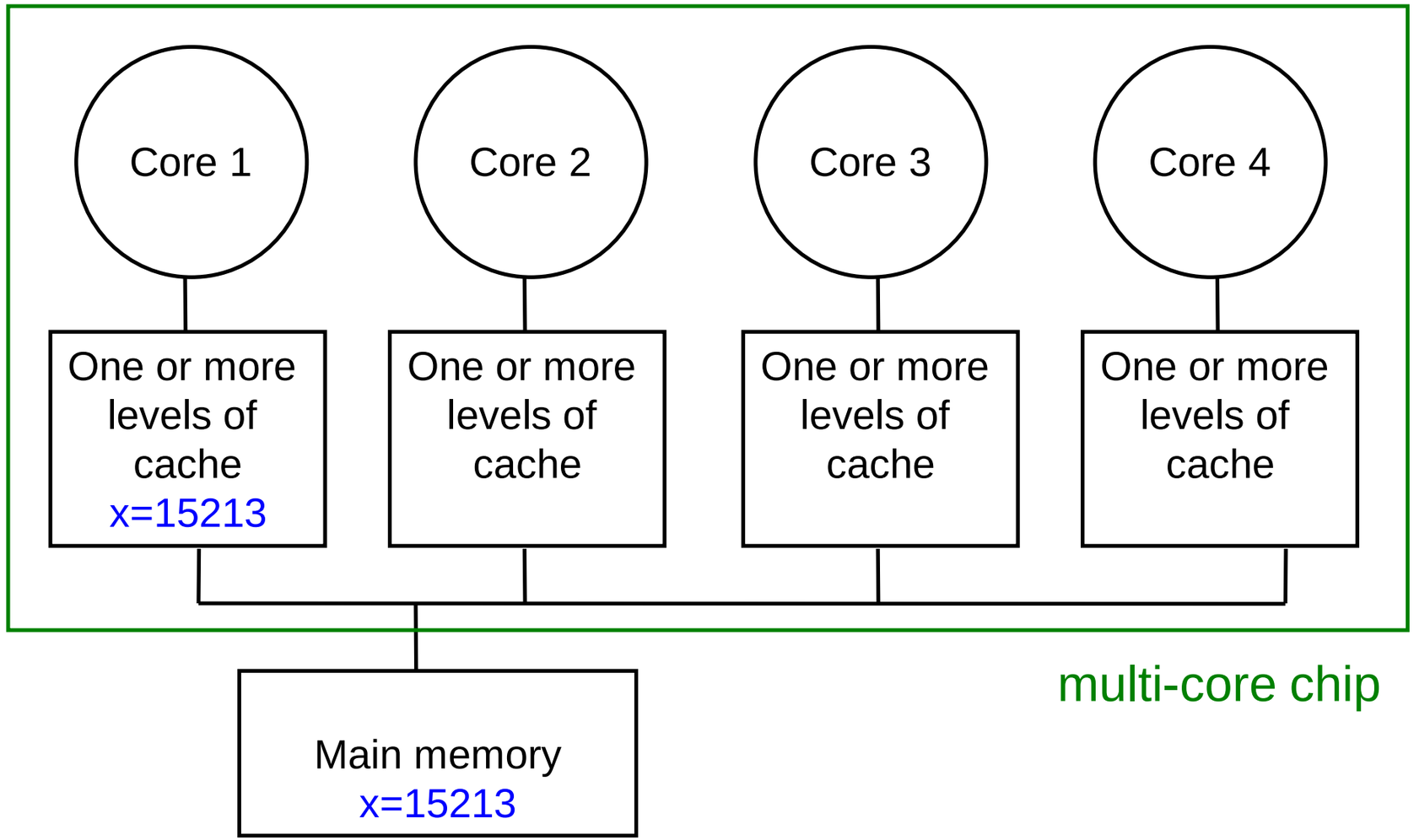
The Cache Coherence Problem

Suppose variable x initially contains 15213



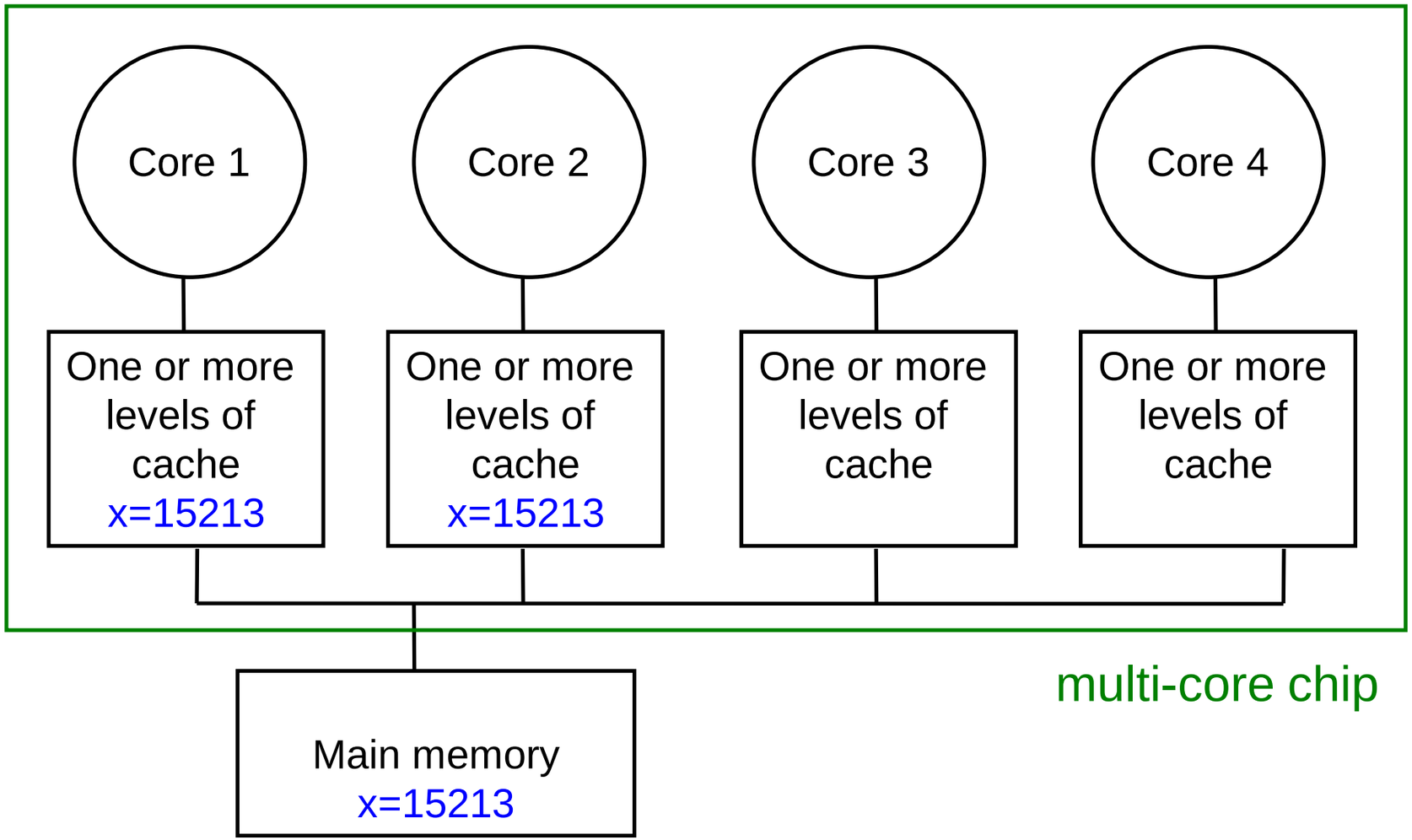
The Cache Coherence Problem

Core 1 reads x



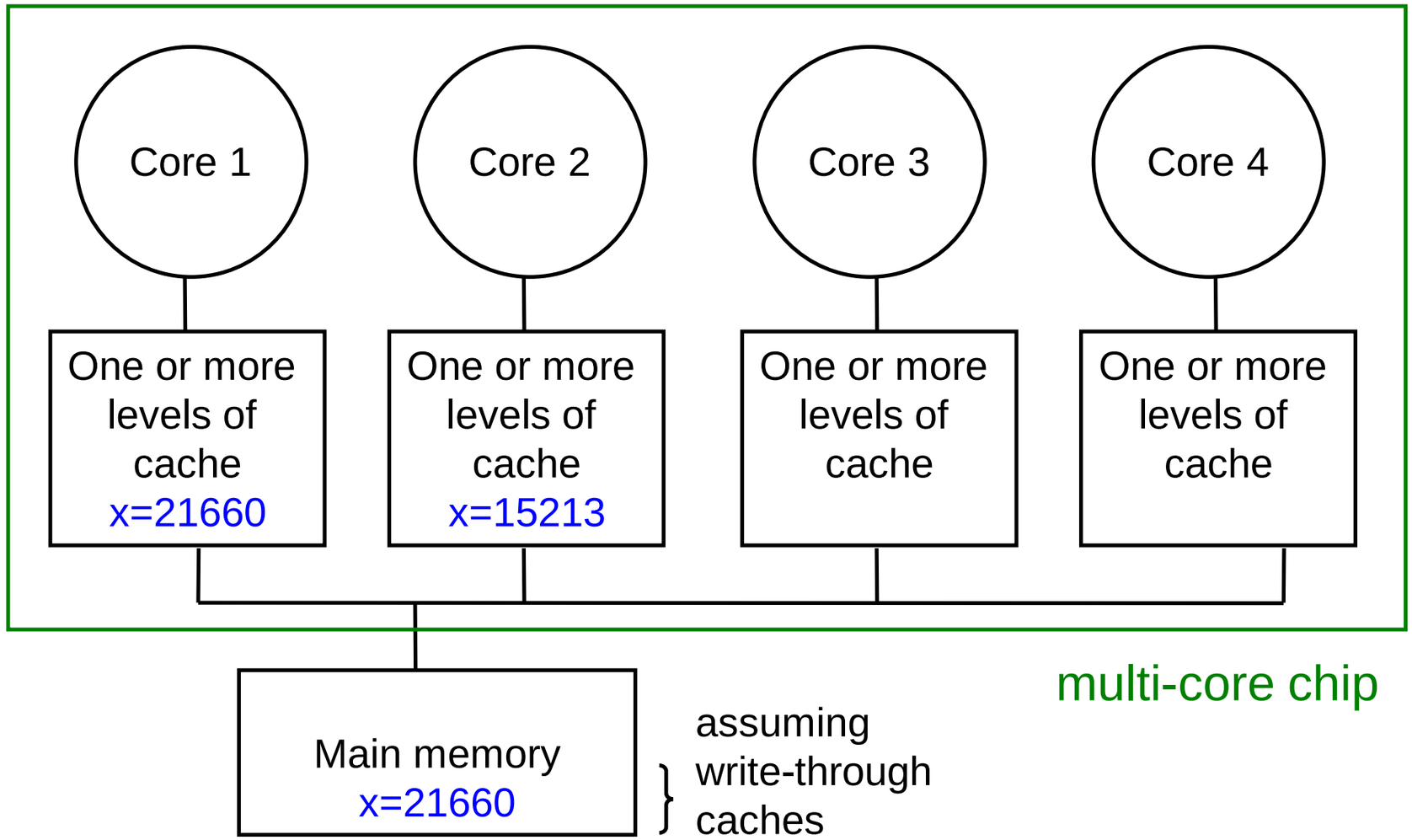
The Cache Coherence Problem

Core 2 reads x



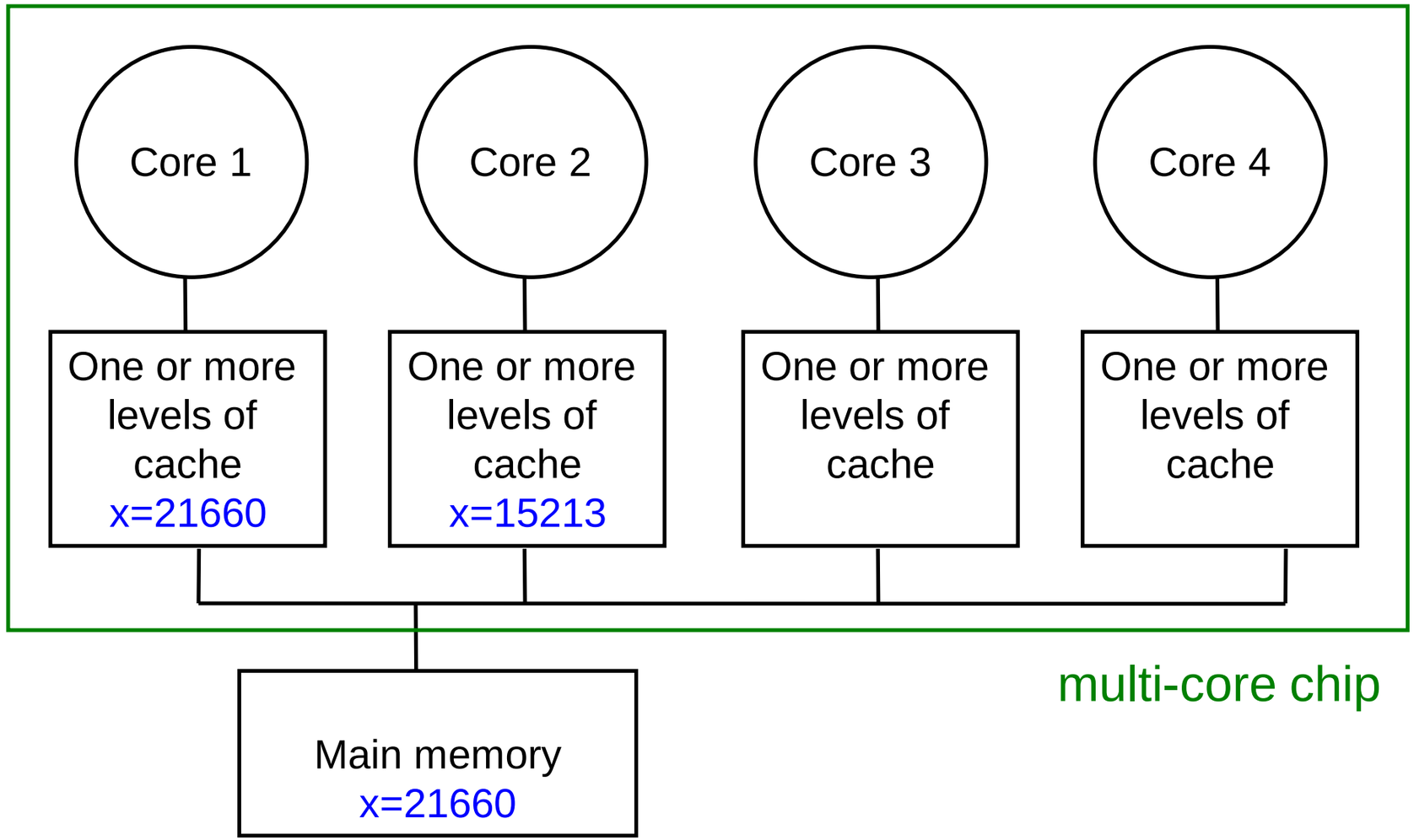
The Cache Coherence Problem

Core 1 writes to x, setting it to 21660



The Cache Coherence Problem

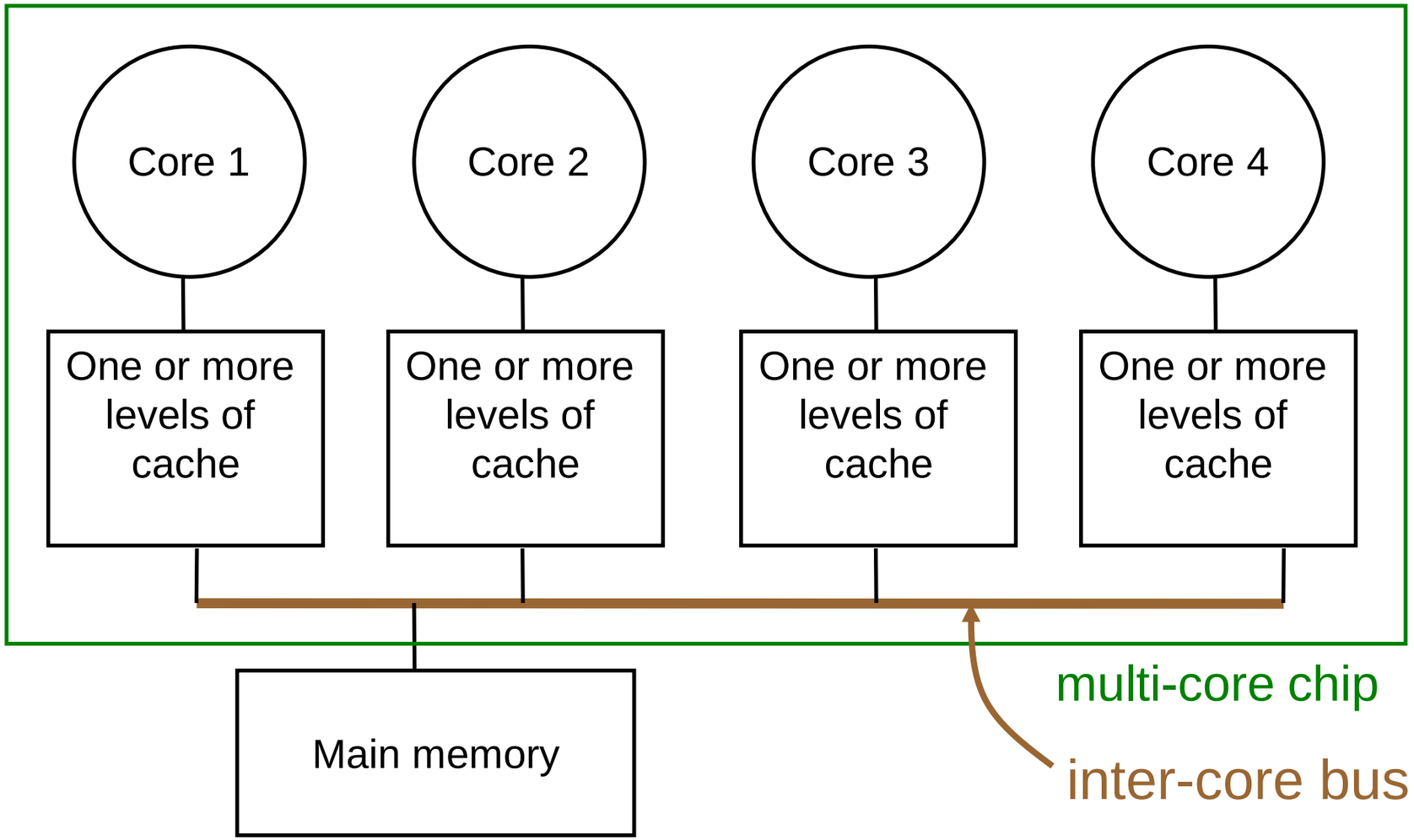
Core 2 attempts to read x… gets a stale copy



Solutions for Cache Coherence Problem

- This is a general problem with shared memory multiprocessors and multicores, with private caches
- Coherence Solution:
  - Use HW to ensure that loads from all cores will return the value of the latest store to that memory location
  - Use metadata to track the state for cached data
  - There exist two major categories with many specific coherence protocols.

Bus Based Multicore Processor

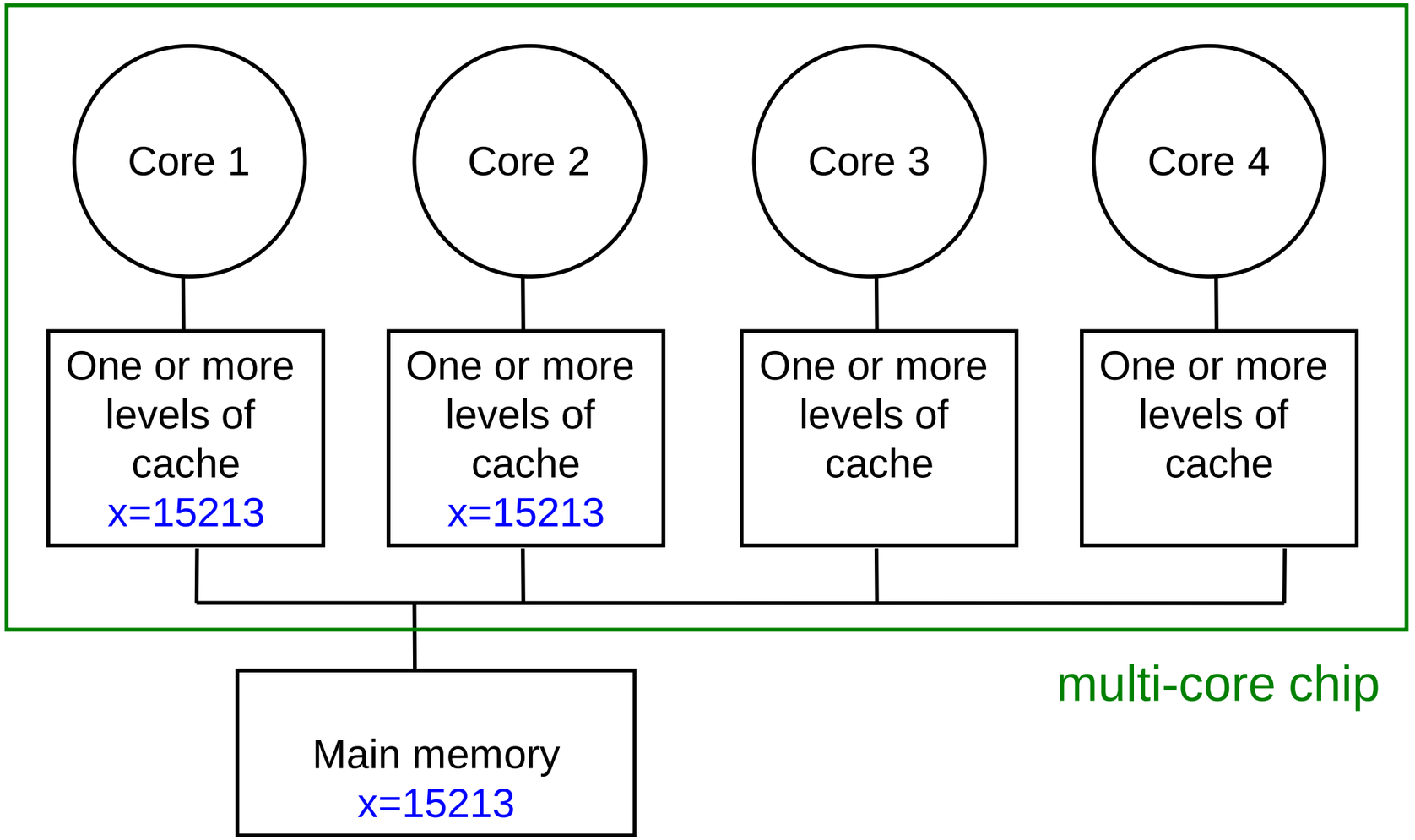


Invalidation Protocol with Snooping

- Invalidation:
  If a core writes to a data item, all other copies of this data item in other caches are *invalidated*
- Snooping:
  All cores continuously "snoop" (monitor) the bus connecting the cores.

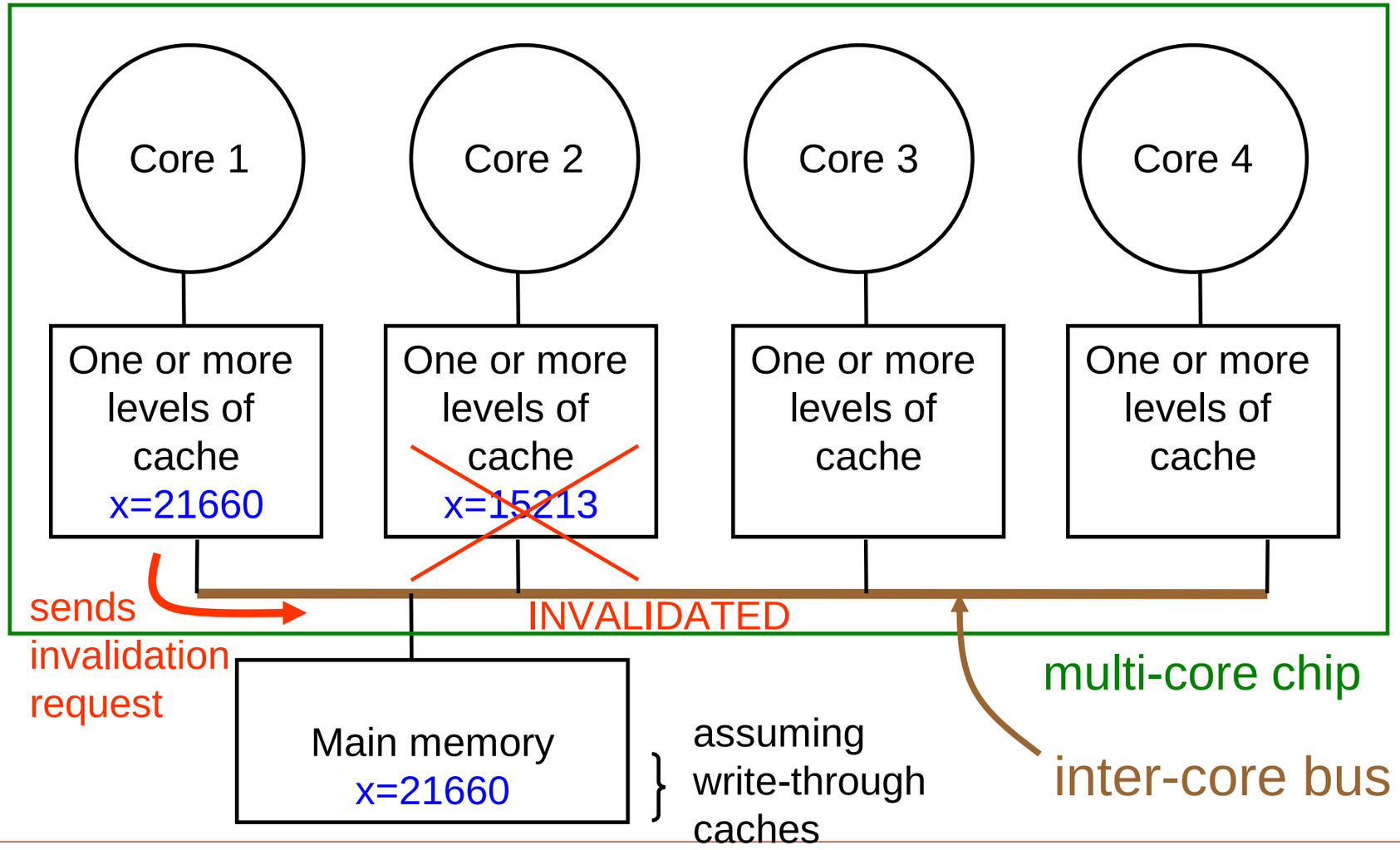
Invalidation Based Cache Coherence Protocol

Revisited: Cores 1 and 2 have both read x



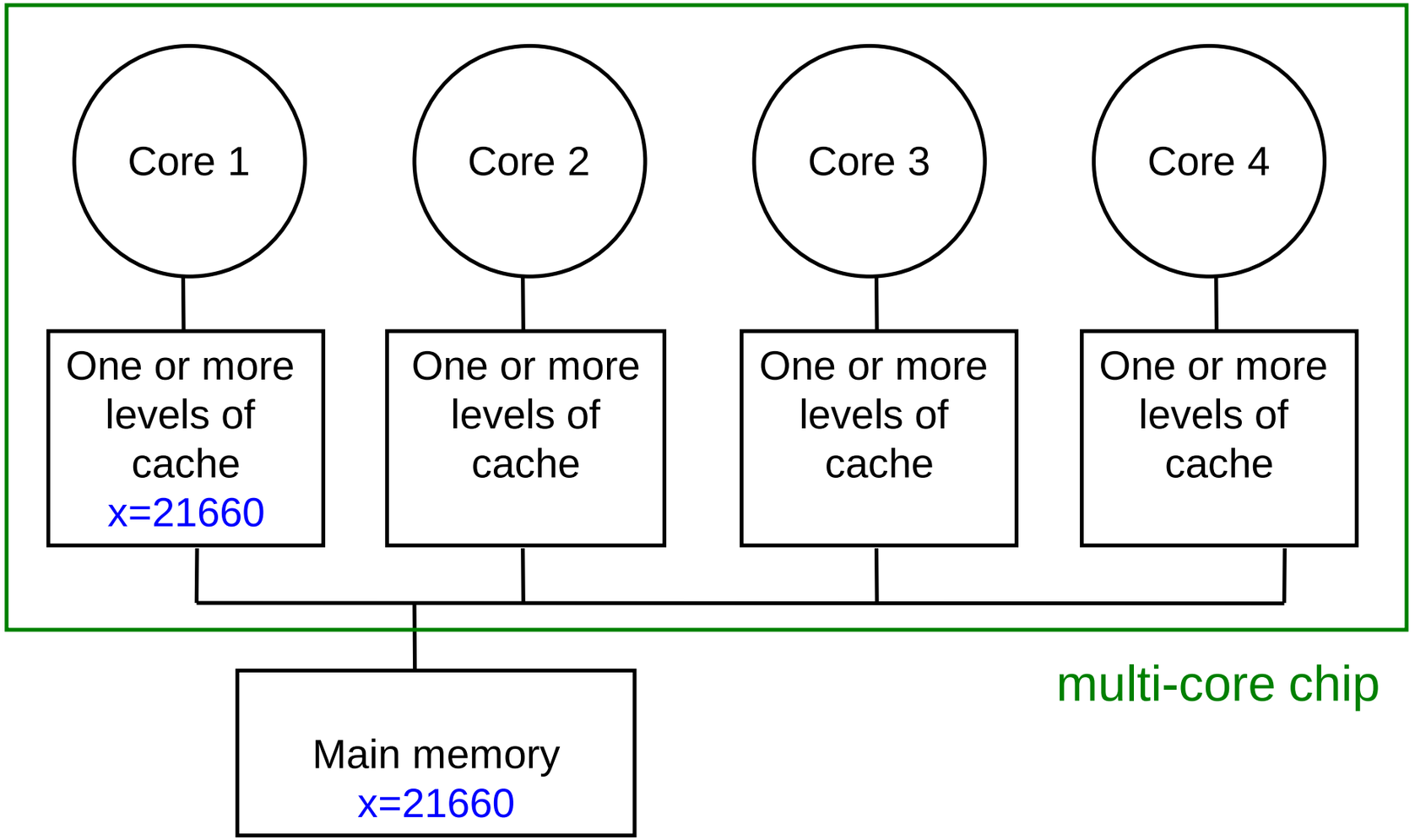
Invalidation Based Cache Coherence Protocol

Core 1 writes to x, setting it to 21660



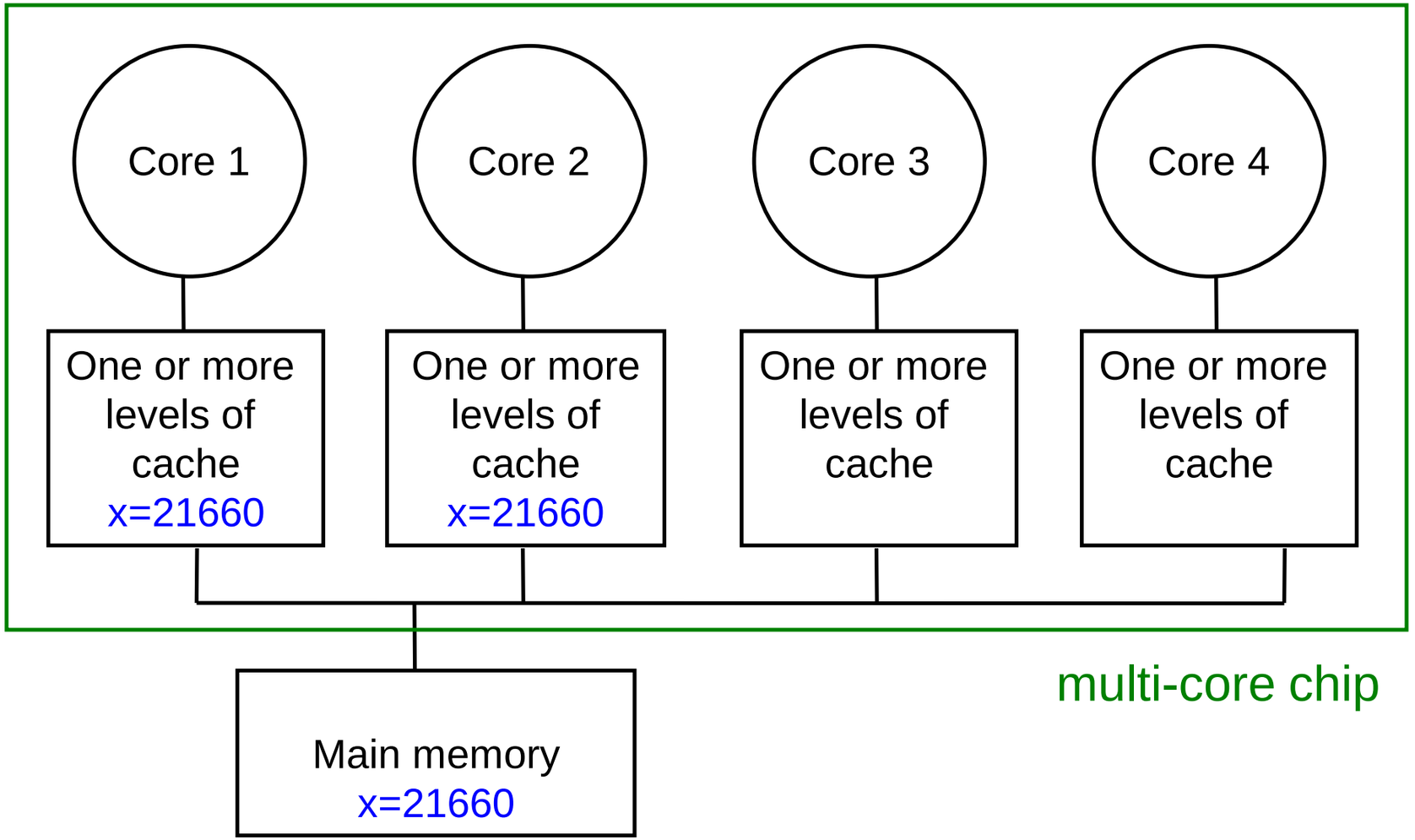
Invalidation Based Cache Coherence Protocol

After invalidation:



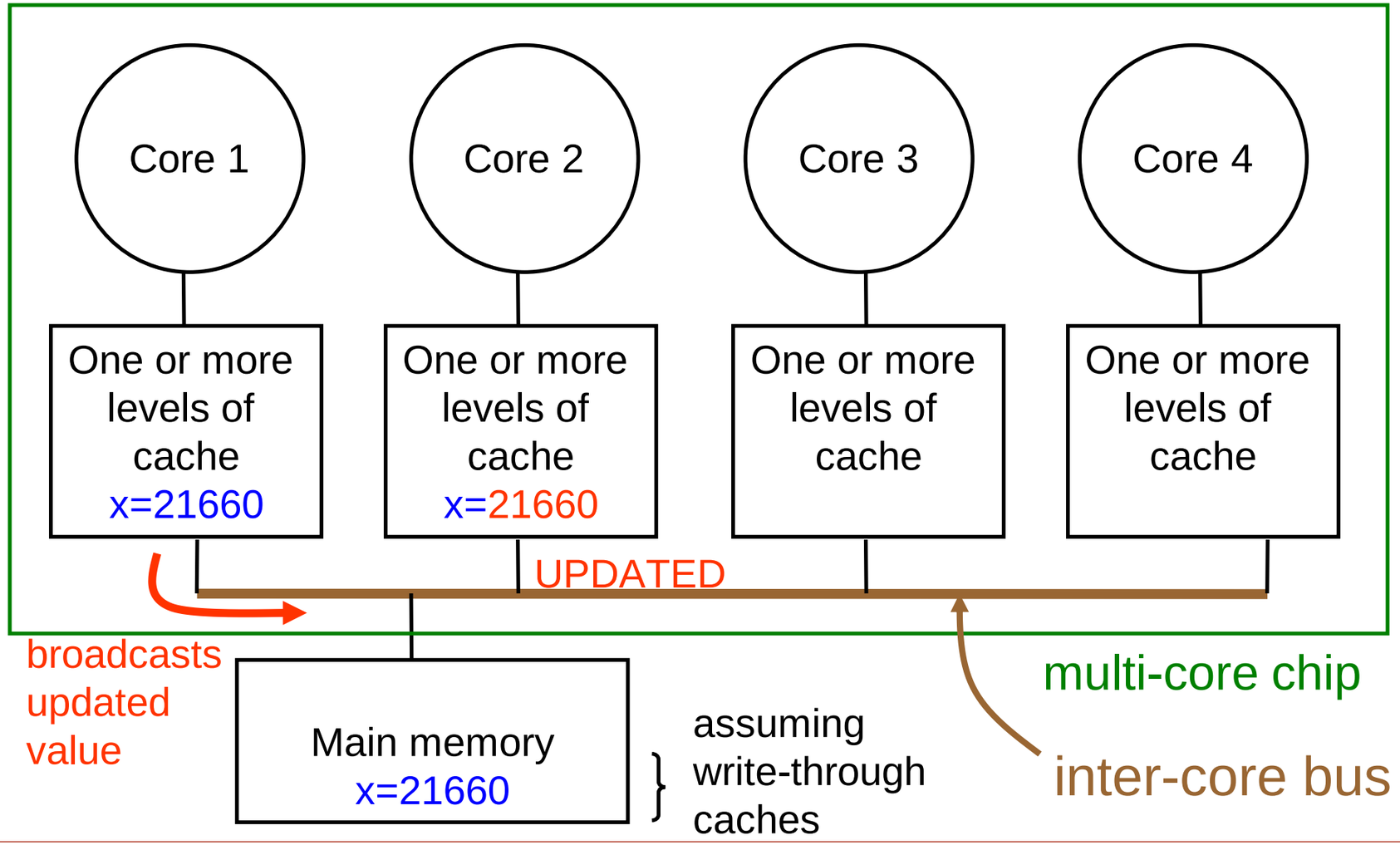
Invalidation Based Cache Coherence Protocol

Core 2 reads x. Cache misses, and loads the new copy.



Update Based Cache Coherence Protocol

Core 1 writes x=21660:



Invalidation vs. Update Protocols

- Multiple writes to the same location
  - invalidation: only the first time
  - update: must broadcast each write
    (which includes new variable value)

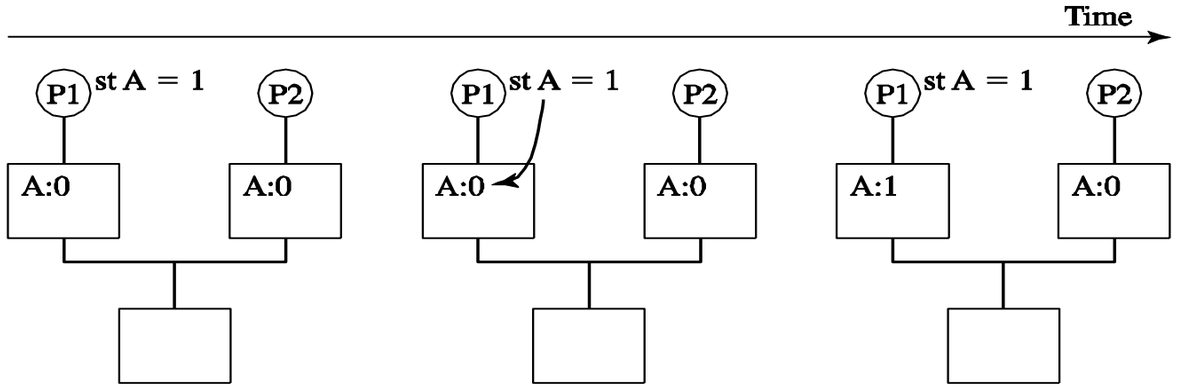
- Invalidation generally performs better:
  it generates less bus traffic

Cache Coherence

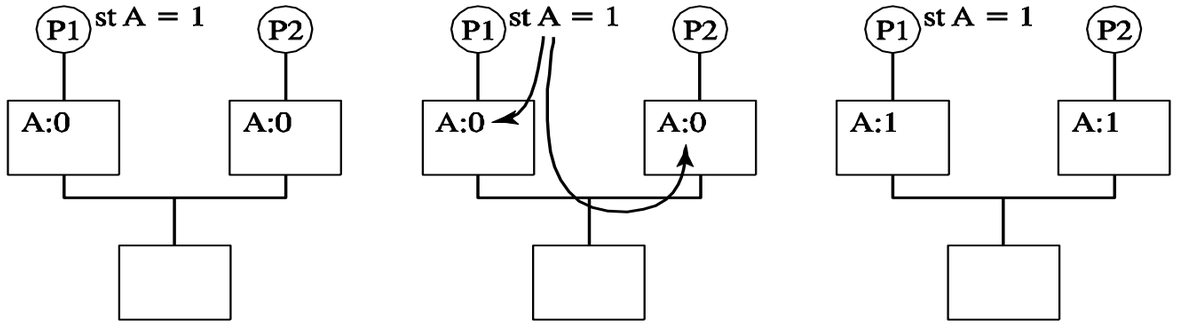
- Informally, with coherent caches: accesses to a memory location *appear* to occur simultaneously in all copies of the memory location
  - "copies" $\Rightarrow$ caches
- Cache coherence suggests an absolute time scale -- this is not necessary
  - What is required is the "appearance" of coherence...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

Write Update vs. Write Invalidate

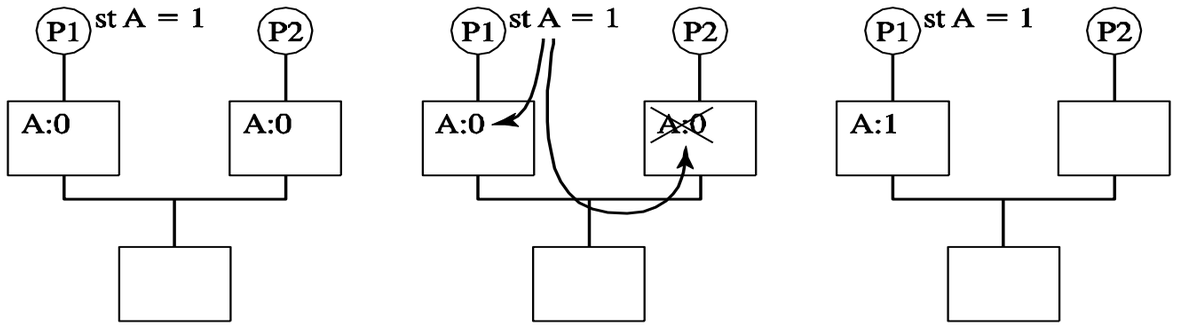
- Coherent caches with Shared Memory
  - All cores see the effects of others' writes
- How/when writes are propagated
  - Determined by coherence protocol



(a) No coherence protocol: stale copy of A at P2



(b) Update protocol writes through to both copies of A



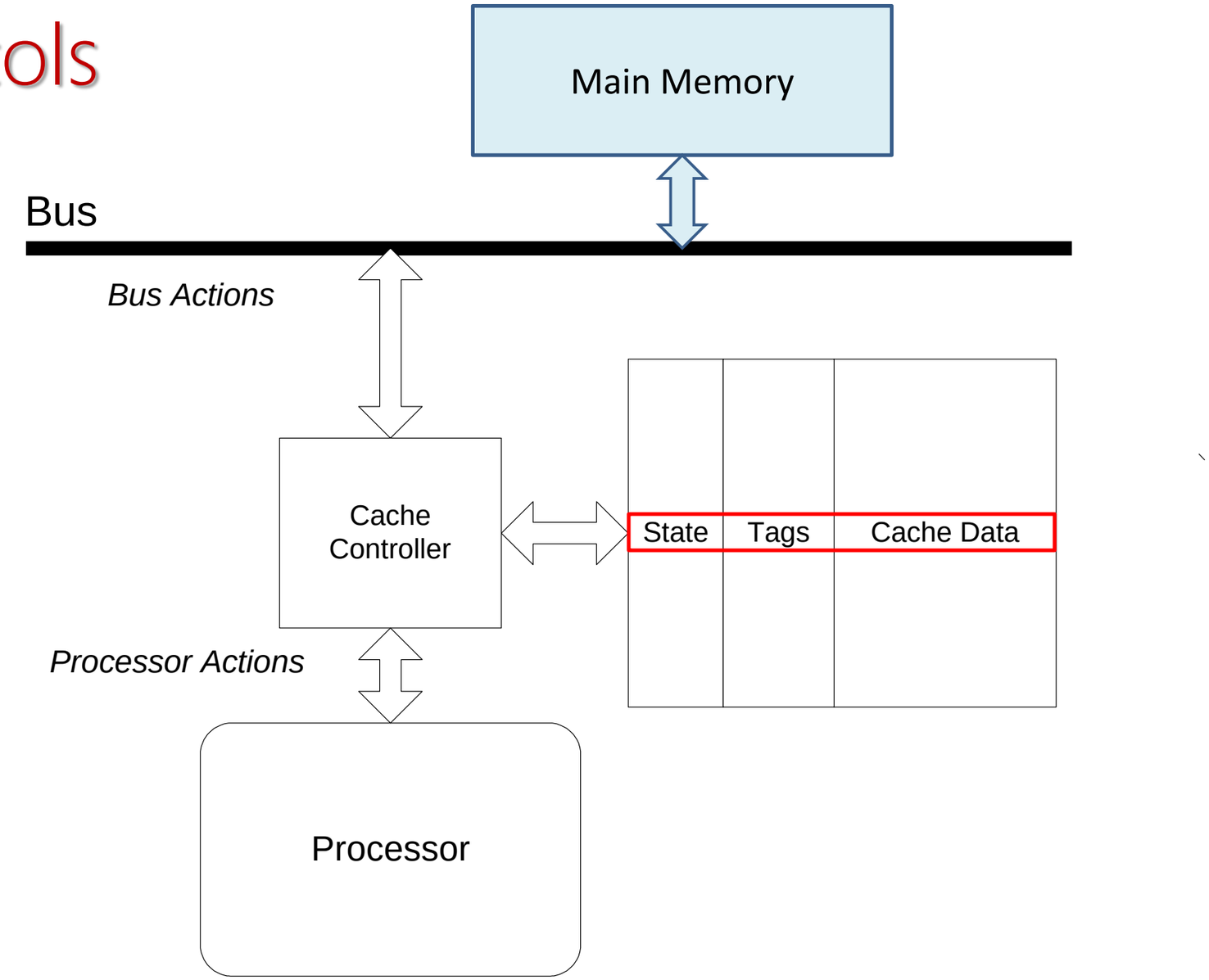
(c) Invalidate protocol eliminates stale remote copy

Bus-Based 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 MC/MP systems
    - Dirty/coherence/sharing misses

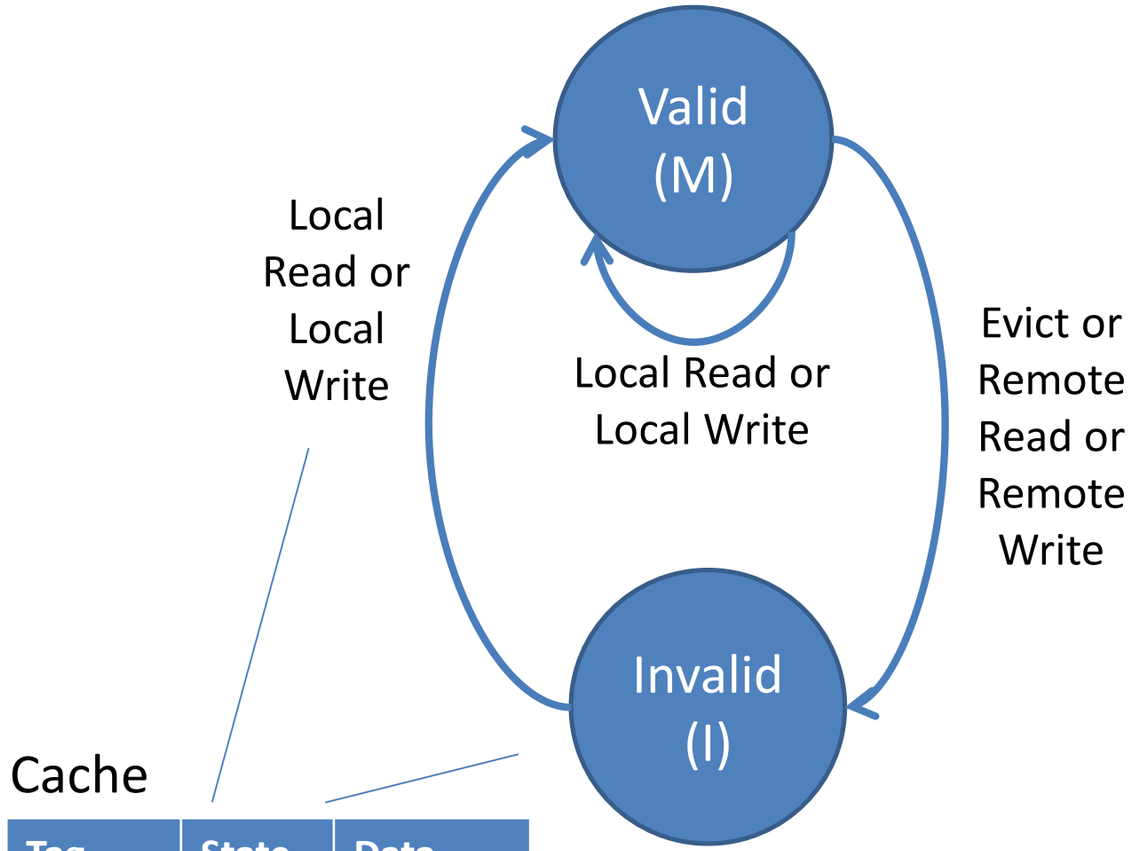
Bus-Based Protocols

- Protocol consists of states and actions (state transitions)
- Actions can be invoked from processor or bus to the cache controller
- Coherence based on per cache line (block)



Minimal Coherence Protocol for Write-Back Caches

- Blocks are always private or exclusive
- State transitions:
  - Local read: I->M, fetch, invalidate other copies
  - Local write: I->M, fetch, invalidate other copies
  - Evict: M->I, write back data
  - Remote read: M->I, write back data
  - Remote write: M->I, write back data



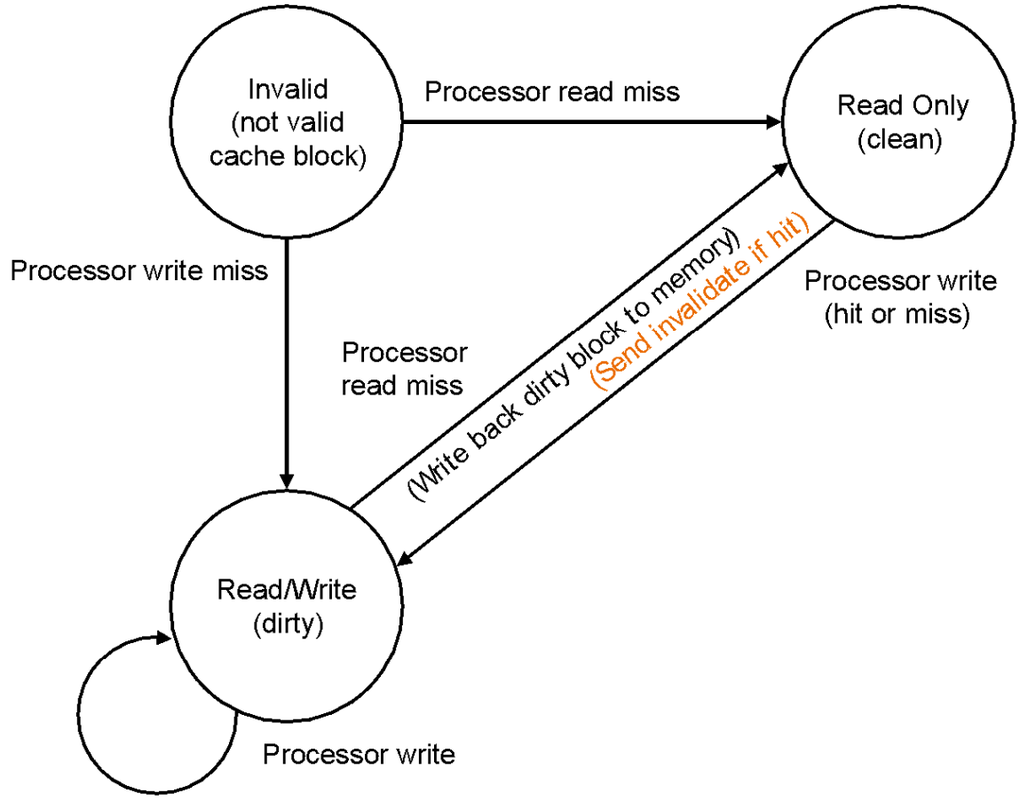
Tag	State	Data
A	M	…
B	I	…

Invalidate Protocol Optimization

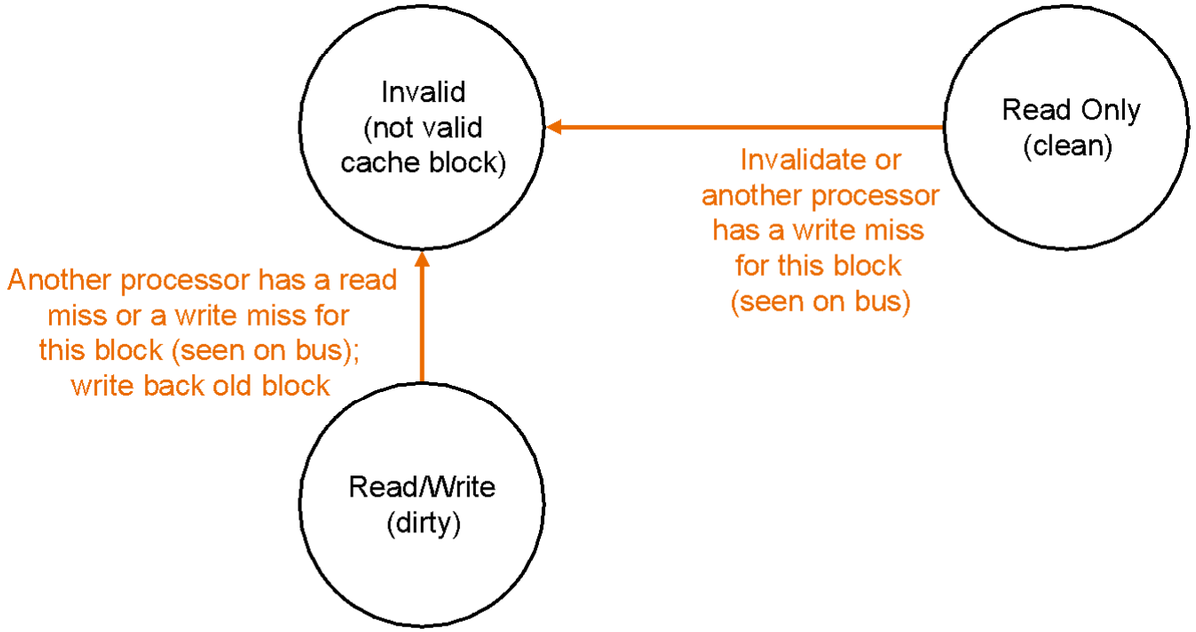
- Observation: data often read shared by multiple CPUs
  - Add S (shared) state to protocol: MSI
- State transitions:
  - Local read: I->S, fetch shared
  - Local write: I->M, fetch modified; S->M, invalidate other copies
  - Remote read: M->I, write back data
  - Remote write: M->I, write back data

Simple Coherence Protocol FSM

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



a. Cache state transitions using signals from the processor



b. Cache state transitions using signals from the bus

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

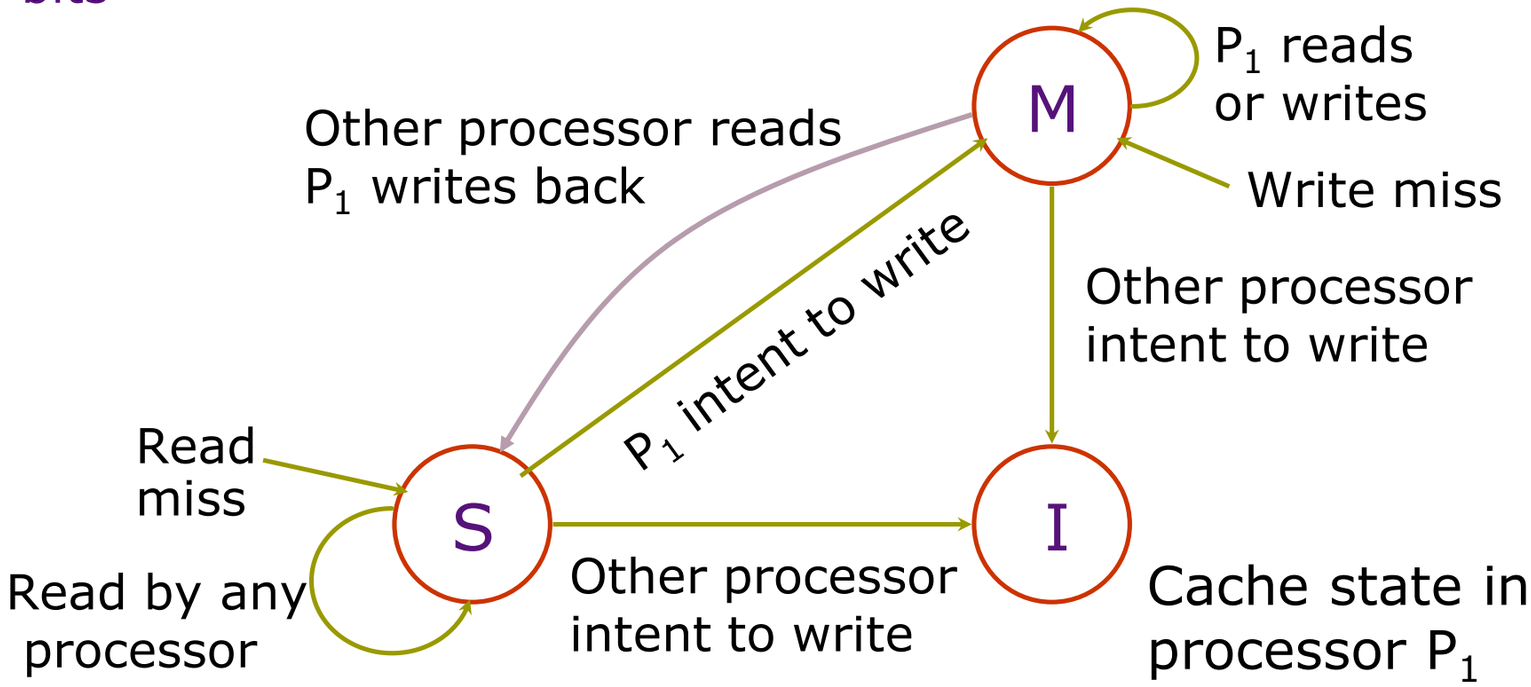
Thread Event	Bus Action	Data From	Global State	Local States: C0	C1	C2
0. Initially:			<0,0,0,1>	I	I	I
1. T0 read→	CR	Memory	<1,0,0,1>	S	I	I
2. T0 write→	CU		<1,0,0,0>	M	I	I
3. T2 read→	CR	C0	<1,0,1,1>	S	I	S
4. T1 write→	CRM	Memory	<0,1,0,0>	I	M	I

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

MSI: A Coherence Protocol for Write Back Caches

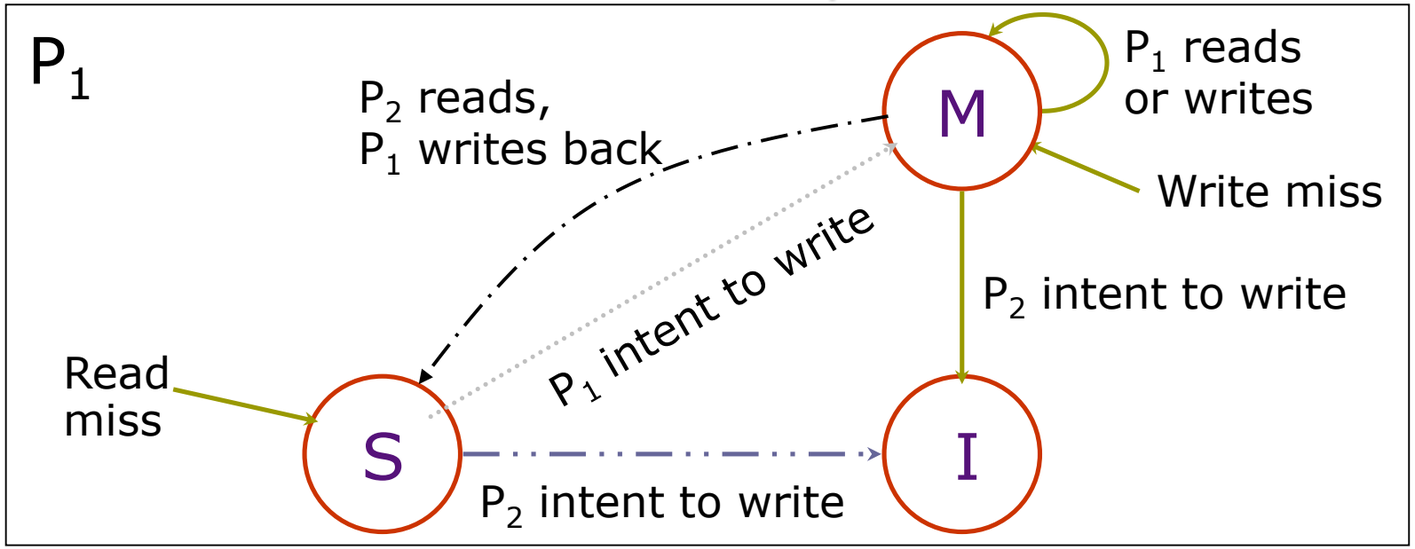
Each cache line has a tag

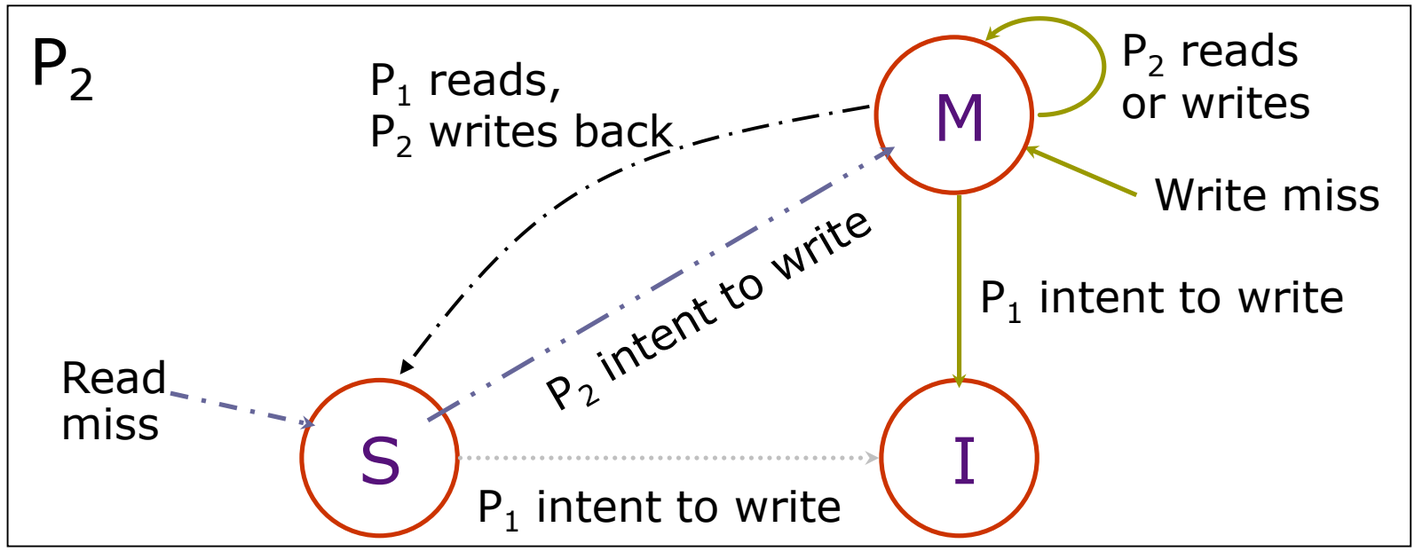
M: Modified
S: Shared
I: Invalid



MSI Coherence Protocol Example with 2 Cores

P_1 reads
P_1 writes
P_2 reads
P_2 writes
P_1 reads
P_1 writes
P_2 writes
P_1 writes





Invalidate Protocol Optimizations

- 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 many Intel processors
- 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

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

Cache-to-Cache Transfers

- Common in many workloads:
  - T0 writes to a block: <1,0,…,0> (block in M state in T0)
  - T1 reads from block: T0 must write back, then T1 reads from memory
- In shared-bus system
  - T1 can *snarf* data from the bus during the writeback
  - Called *cache-to-cache transfer* or *dirty miss* or *intervention*
- Without shared bus
  - Must explicitly send data to requestor and to memory (for writeback)
- Known as the 4th C (cold, capacity, conflict, communication)

MESI Example 2

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. T1 read→	CR	C0	<1,1,0,1>	S	S	I
4. T2 read→	CR	Memory	<1,1,1,1>	S	S	S

MOESI Optimization

- Observation: shared ownership prevents cache-to-cache transfer, causes unnecessary memory read
  - Add O (owner) state to protocol: MOSI/MOESI
  - Last requestor becomes the owner
  - 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,…,1>
  - **I**nvalid: <0,X,X,…X>
  - **O**wned: <1,X,X,X,0> ; only one owner, memory not up to date
- 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

MOESI Coherence Protocol

- A protocol that tracks validity, ownership, and exclusiveness
  - Modified: dirty and private
  - Owned: dirty but shared
    - Avoid writeback to memory on M->S transitions
  - Exclusive: clean but private
    - Avoid upgrade misses on private data
  - Shared
  - Invalid
- There are also some variations (MOSI and MESI)
- What happens when 2 cores read/write different words in a cache line?

Snooping with Multi-level Caches

- Private L2 caches
  - If inclusive, snooping traffic checked at the L2 level first
  - Only accesses that refer to data cached in L1 need to be forwarded
  - Saves bandwidth at the L1 cache

- Shared L2 or L3 caches
  - Can act as serialization points even if there is no bus
  - Track state of cache line and list of sharers (bit mask)
  - Essentially the shared cache acts like a coherence directory

Scaling Coherence Protocols

- The problem
  - Too much broadcast traffic for snooping (probing)
- Solution: probe filters
  - Maintain info of which address ranges that are definitely not shared or definitely shared
  - Allows filtering of snoop traffic
- Solution: directory based coherence
  - A directory stores all coherence info (e.g., sharers)
  - Consult directory before sending coherence messages
  - Caching/filtering schemes to avoid latency of 3-hops

Scaleable Cache Coherence

- No 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->Requestor
      - Directory: Requestor->Directory->Owner->Requestor

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 Based Protocol Implementation

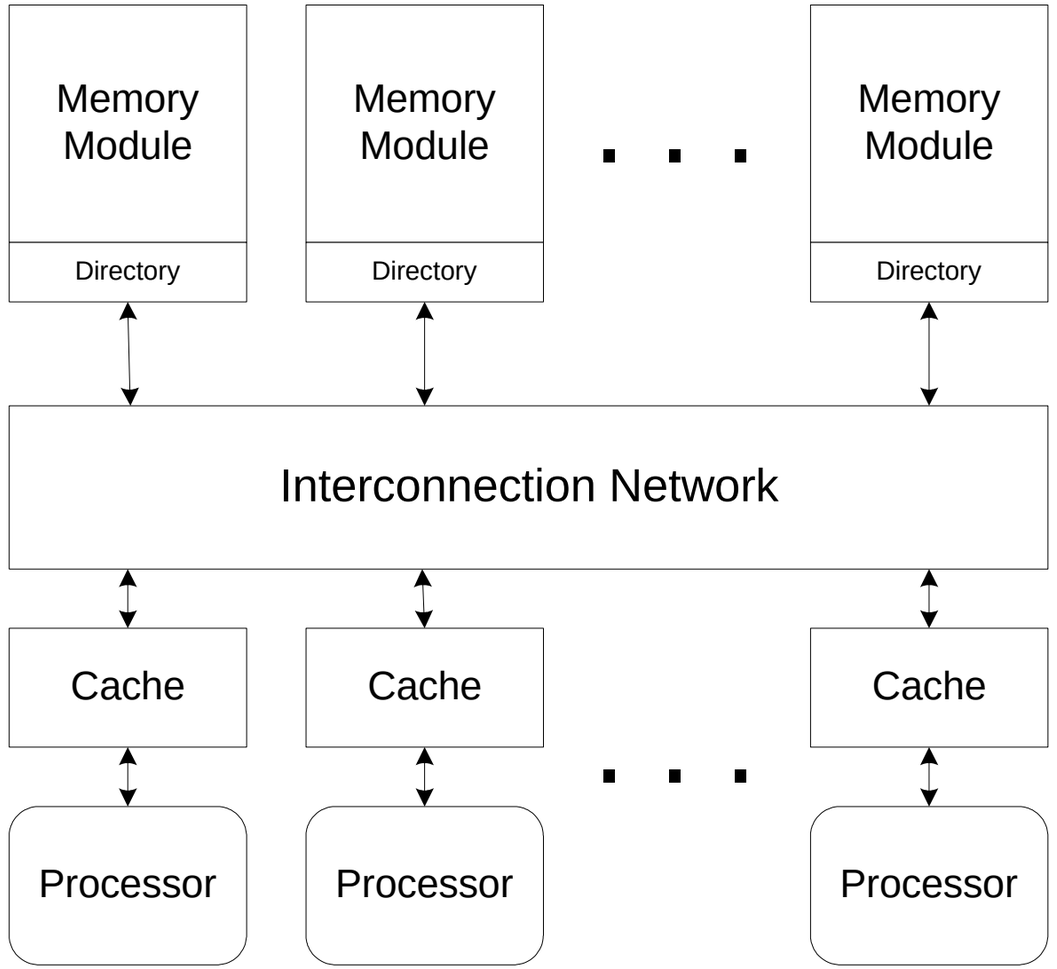
- Basic idea: Centralized directory keeps track of data location(s)
- Scalable
  - Address traffic roughly proportional to number of processors
  - Directory & traffic can be distributed with memory banks (interleaved)
  - Directory cost (SRAM) or latency (DRAM) can be prohibitive
- Presence bits track 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 Based Protocol Latency

- Access to non-shared data
  - Overlap directory read with data read
  - Best possible latency given distributed memory
- 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

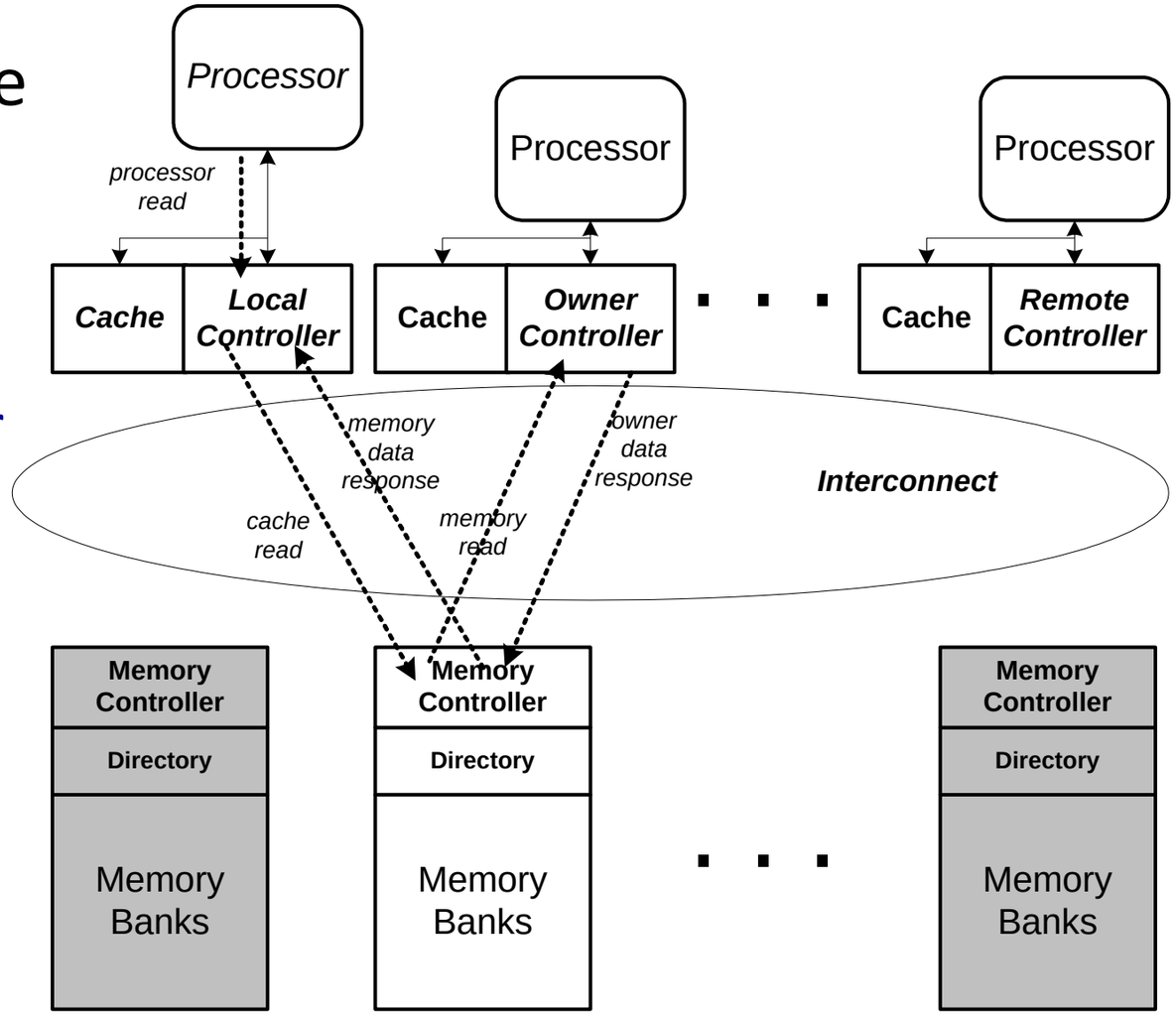


Example: Simple Directory Protocol

- 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 Table

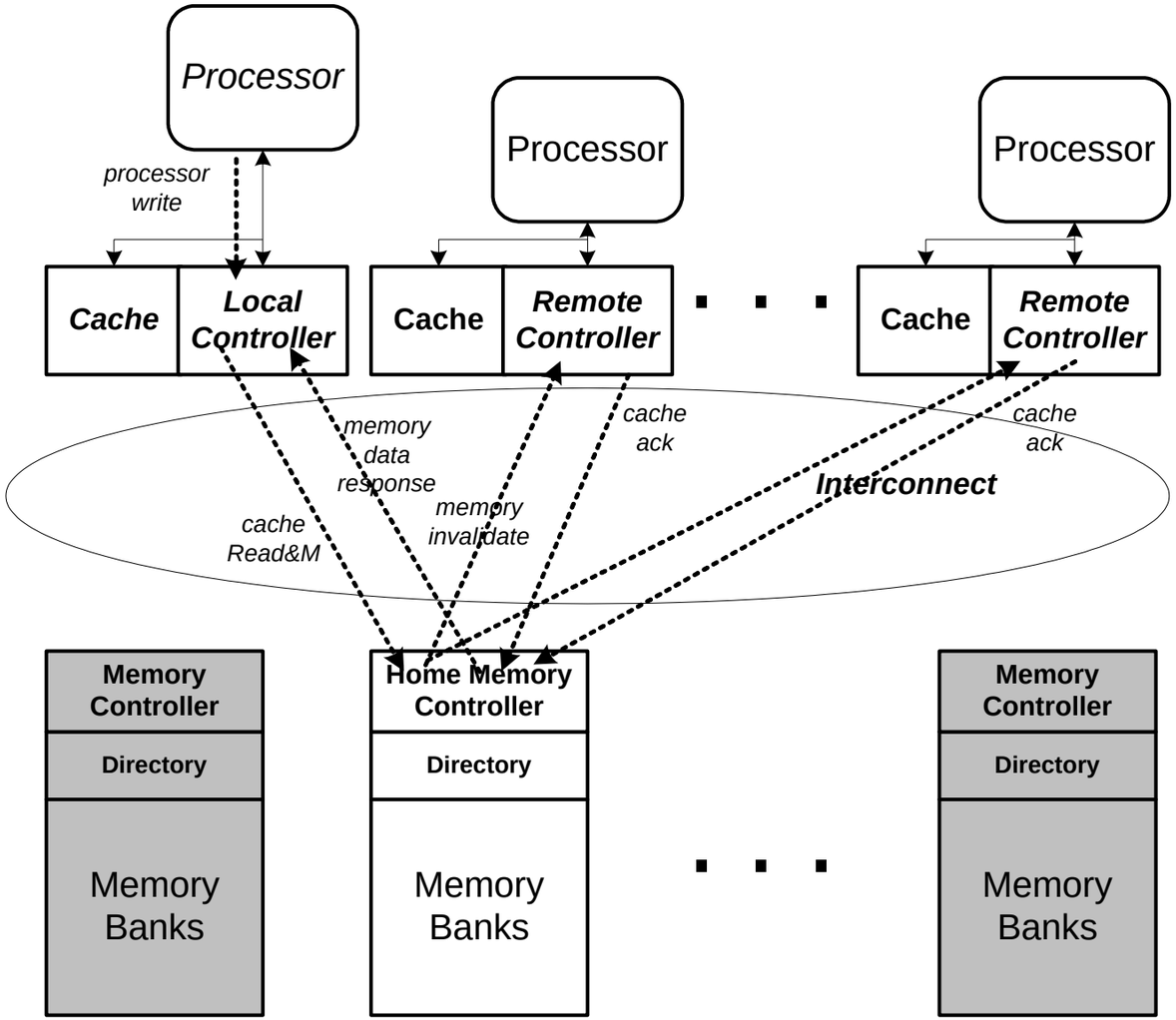
	Cache Controller Actions and Next States								
	from Processor Side				from Memory Side				
Current State	*Processor Read*	*Processor Write*	*Eviction*		*Memory Read*	*Memory Read&M*	*Memory Invalidate*	*Memory Upgrade*	*Memory Data*
I	*Cache Read* → I'	*Cache Read&M* → I''					No Action → I		
S	No 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 Table

	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	

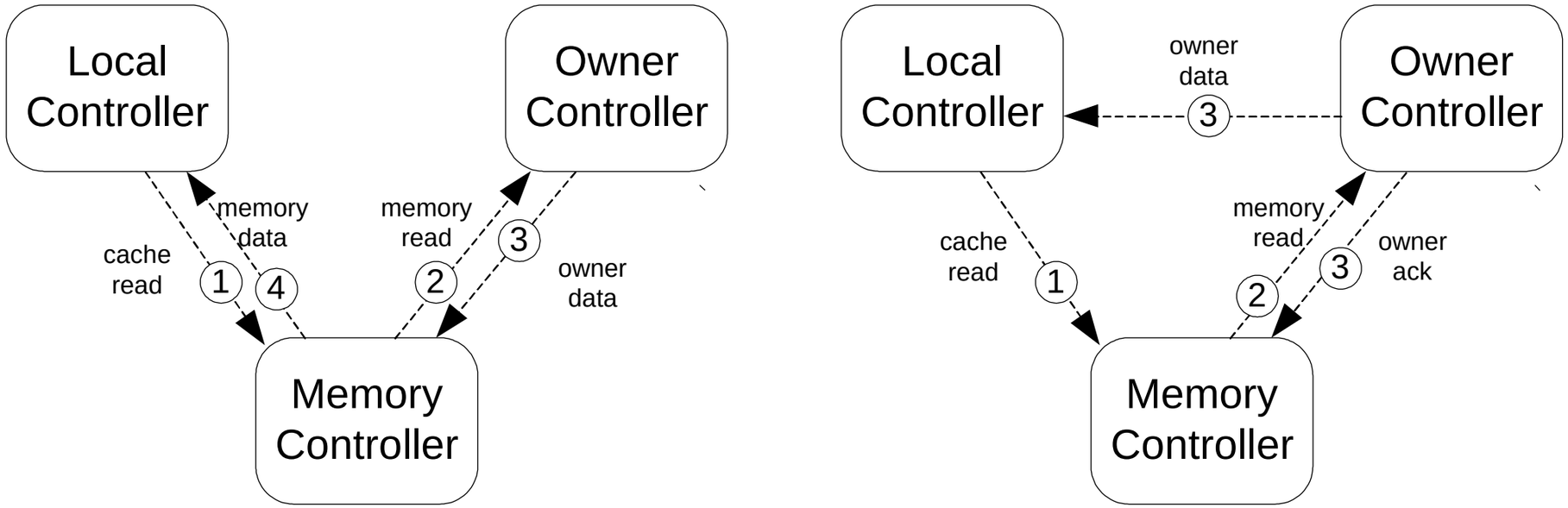
Another Example

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



Variation: Three Hop Protocol

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



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

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]
  - Virtual Tree Coherence [Enright Jerger MICRO 2008]

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 vs. Invalidate

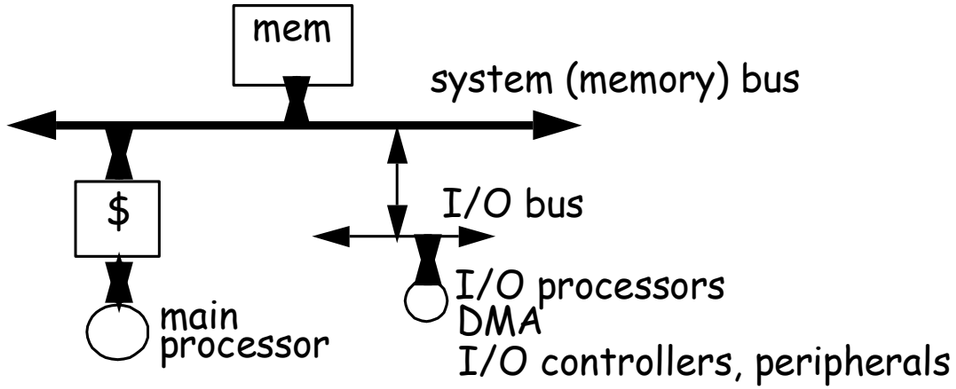
- [Weber & Gupta, ASPLOS3]
  - Consider sharing patterns
- No Sharing
  - Independent threads
  - Coherence due to thread migration
  - Update protocol performs many wasteful updates
- Read-Only
  - No significant coherence issues; most protocols work well
- Migratory Objects
  - Manipulated by one processor 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

Uniprocessor Coherence

- IN UNIPROCESSORS, A LOAD MUST RETURN THE VALUE OF THE LATEST STORE IN THREAD ORDER
  - THIS IS DONE THROUGH MEMORY DISAMBIGUATION AND MANAGMENT OF CACHE HIERARCHY
  - SOME PROBLEMS WITH I/O, AS I/O IS OFTEN CONNECTED TO MEMORY BUS



- COHERENCE BETWEEN I/O TRAFFIC AND CACHE MUST BE ENFORCED
  - HOWEVER, THIS IS INFREQUENT AND SOFTWARE IS INFORMED
  - SO SOFTWARE SOLUTIONS WORK
  - UNCACHEABLE MEMORY, UNCACHEABLE OPS, CACHE FLUSHING
  - ANOTHER SOLUTION IS TO PASS I/O THROUGH CACHE
- IN MULTIPROCESSORS THE COHERENCE PROBLEM IS PERVASIVE, PERFORMANCE CRITICAL AND SOFTWARE IS NOT INFORMED
  - SHARING OF DATA, THREAD MIGRATION AND I/O
  - COMMUNICATION IS IMPLICIT
  - THUS HARDWARE MUST SOLVE THE PROBLEM.

18-600 Foundations of Computer Systems

Lecture 22: "Performance and Power Iron Laws"

John P. Shen & Zhiyi Yu
November 16, 2016

Next Time ...

- Recommended References:
  - "Energy per Instruction Trends in Intel® Microprocessors," by Ed Grochowski, Murali Annavaram, 2006.
  - "Best of Both Latency and Throughput," by E. Grochowski, R. Ronen, J. Shen, H. Wang. In 22nd ICCD 2004.
  - "Mitigating Amdahl's Law through EPI Throttling," by M. Annavaram, E. Grochowski, J. Shen. In 32nd ISCA 2005.

Electrical & Computer ENGINEERING