18-600 Foundations of Computer Systems

Lecture 17: "Multicore Cache Coherence"

John P. Shen

October 25, 2017

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+ cores 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 17: "Multicore Cache Coherence"

A. Multicore Processors
   - The Case for Multicores
   - Programming for Multicores
   - The Cache Coherence Problem

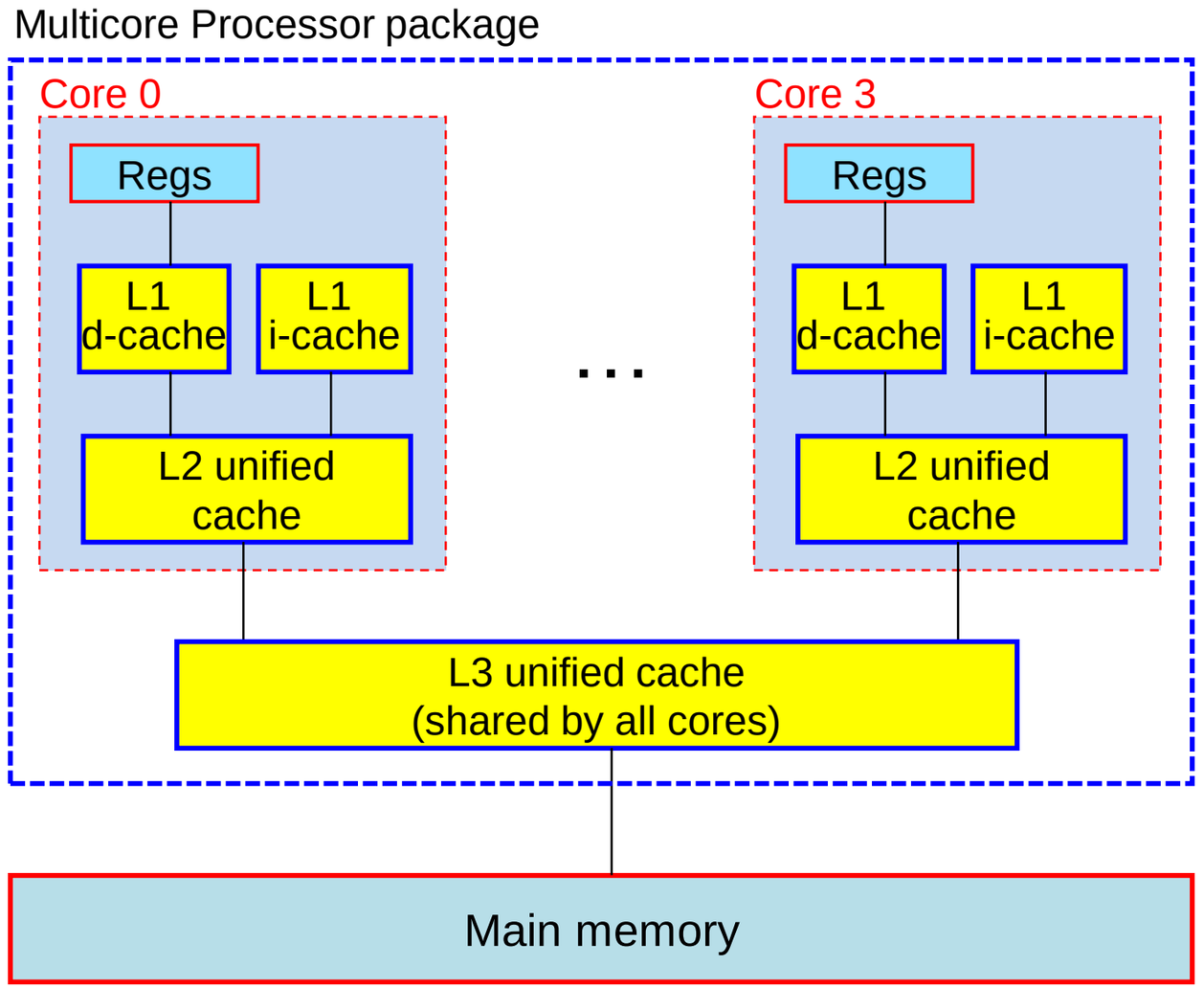
B. Cache Coherence Protocol Categories
   - Write Update
   - Write Invalidate

C. Specific Bus-Based Snoopy Protocols
   - VI & MI Protocols
   - MSI, MESI, MOESI Protocols

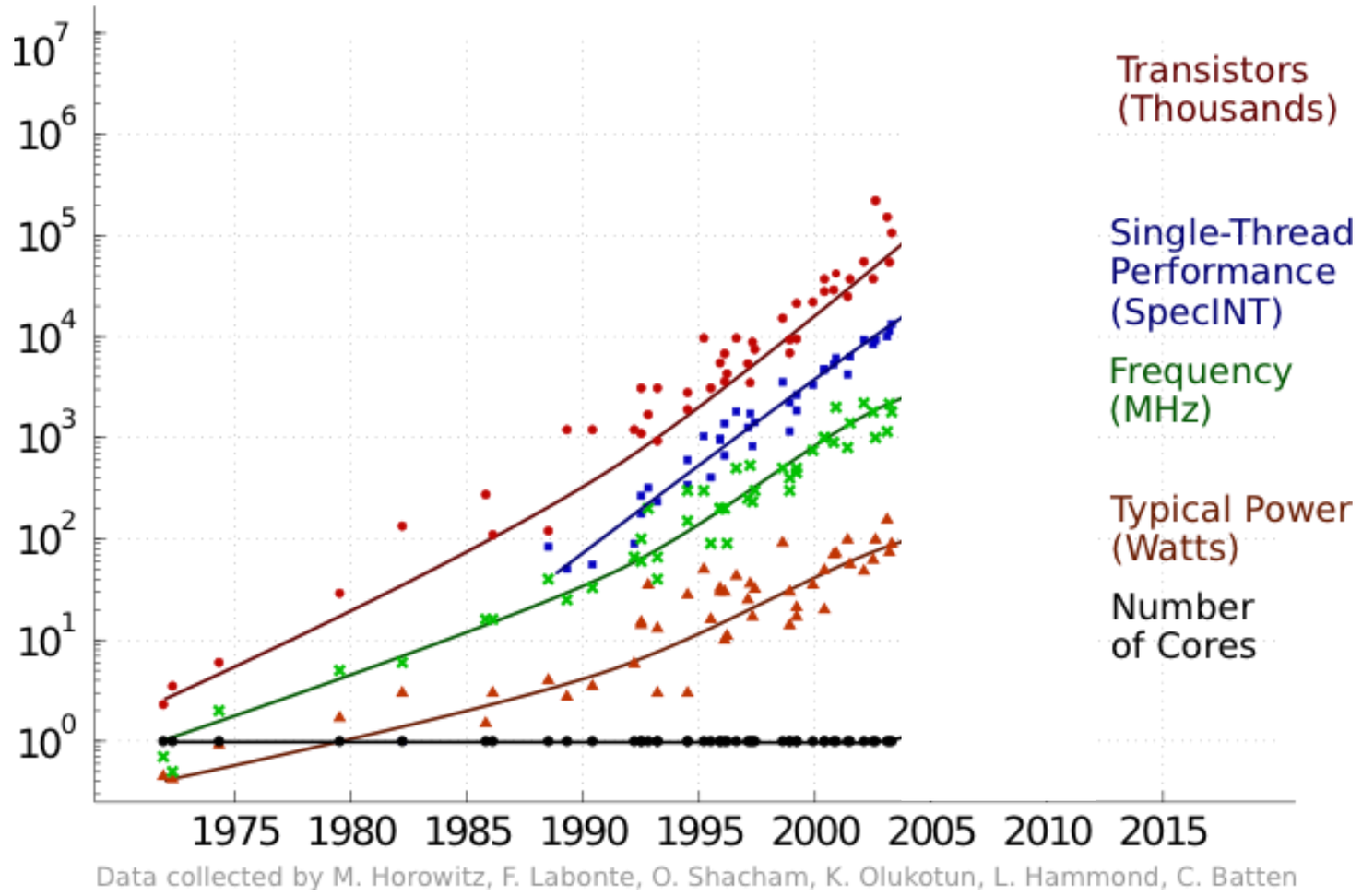
Electrical & Computer ENGINEERING

The Case for Multicore Processors (MCP)

- Stalled Scaling of Single-Core Performance
- Expected continuation of Moore's Law
- Throughput Performance for Server Workloads



Processor Scaling Until ~2004



Processor Development Until ~2004

- Moore's Law: transistor count doubles every 18 months
  - Used to improve processor performance by 2x every 18 months
  - Single core, binary compatible to previous generations
- Contributors to performance improvements
  - More ILP through OOO superscalar techniques
    - Wider issue, better branch prediction, better instruction scheduling, …
    - Better memory hierarchies, faster and larger
  - Clock frequency improvements with deeper pipelines

Problems with Single Core Performance

- Moore's Law is still doing well (for the foreseeable future…)
- The Power Wall
  - Power ≈ C_L * V_{dd}^2 * Freq

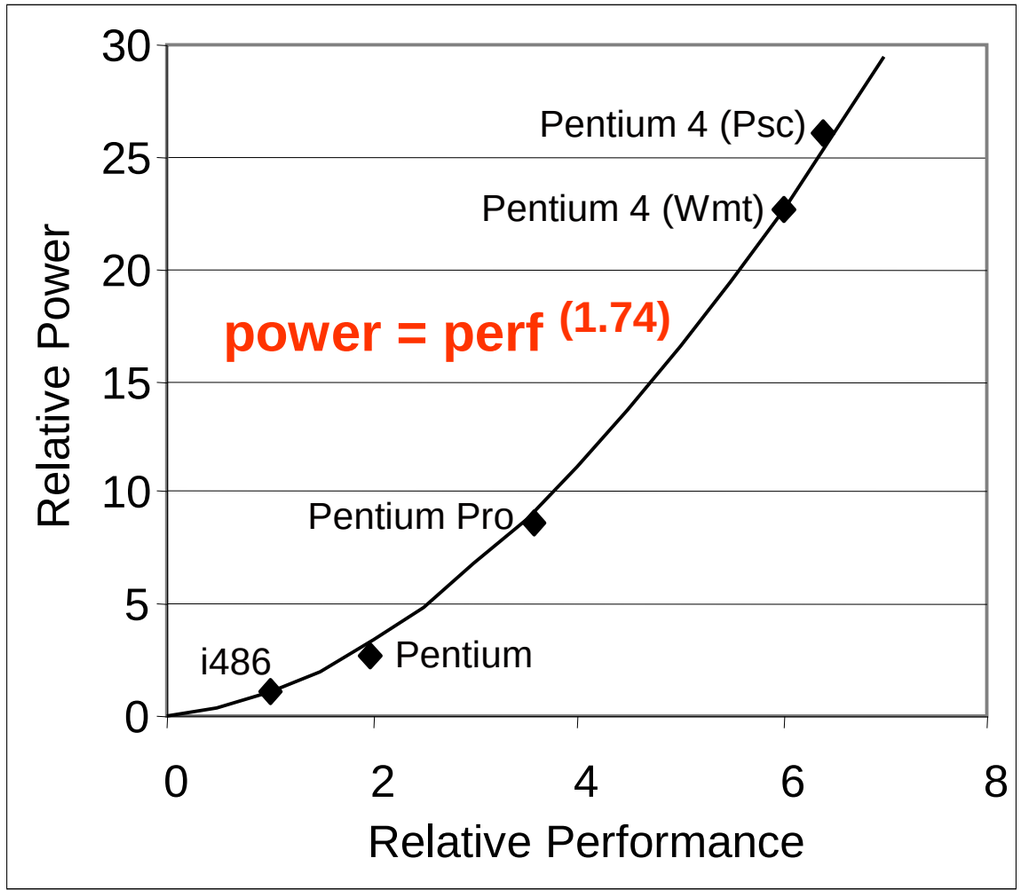
$$P_{dyn} = C_L V_{DD}^2 f$$

  - Cannot scale transistor count and frequency without reducing V_{dd}
  - Unfortunately, voltage scaling has essentially stalled
- The Complexity Wall
  - Designing and verifying increasingly large OOO cores is very expensive
    - 100s of engineers for 3-5 years
  - Caches are easier to design but can only help so much…

[Ed Grochowski, 2004]

Power & Latency (Single-Thread) Performance

- For comparison
  - Factor out contributions due to process technology
  - Keep contributions due to microarchitecture design
  - Normalize to i486™ processor
- Relative to i486™ Pentium® 4 (Wmt) processor is
  - 6x faster (2X IPC at 3X frequency)
  - 23x higher power
  - Spending 4 units of power for every 1 unit of scalar performance



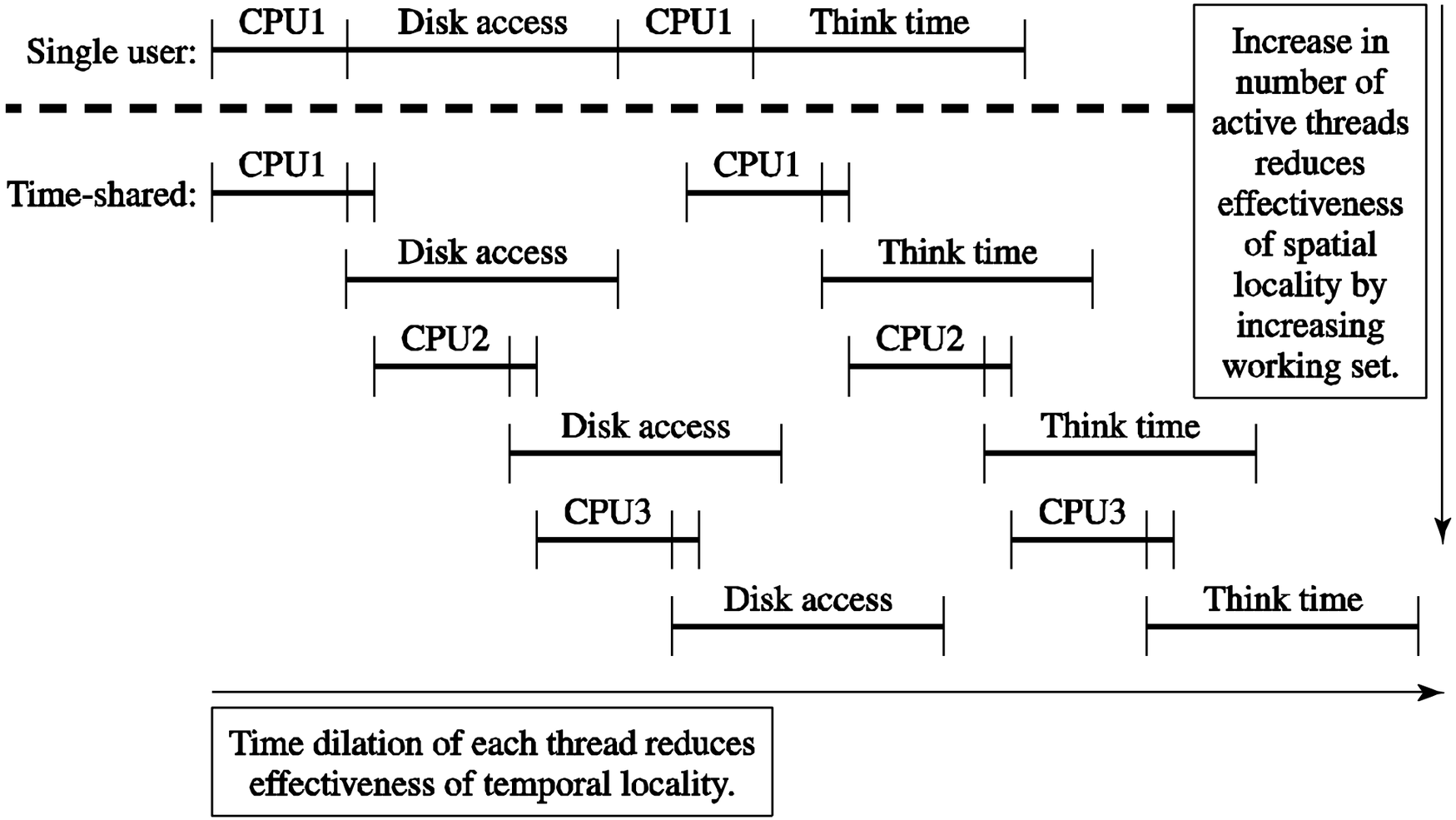
From ILP to TLP

- So far, we run single process, single thread
  - Extracting ILP from sequential instruction stream
- Single-thread performance can't scale indefinitely!
  - Limited ILP within each thread
  - Power consumption & complexity of superscalar cores
- We will now pursue Thread-Level Parallelism (TLP)
  - To increase utilization and tolerate latency in single core
  - To exploit multiple cores

Thread-Level Parallelism

- Instruction-level parallelism (ILP)
  - Reaps performance by finding independent work in a single thread
- Thread-level parallelism (TLP)
  - Reaps performance by finding independent work across multiple threads
- Historically, requires explicitly parallel workloads
  - Originate from mainframe time-sharing workloads
  - Even then, CPU speed >> I/O speed
  - Had to overlap I/O latency with “something else” for the CPU to do
  - Hence, operating system would schedule other tasks/processes/threads that were “time-sharing” the CPU

Thread-Level Parallelism



- Reduces effectiveness of temporal and spatial locality

Thread-Level Parallelism

- Initially motivated by time-sharing of single CPU
  - OS, applications written to be multithreaded
- Quickly led to adoption of multiple CPUs in a single system
  - Enabled scalable product line from entry-level single-CPU systems to high-end multiple-CPU systems
  - Same applications, OS, run seamlessly
  - Adding CPUs increases throughput (performance)
- More recently:
  - Multiple threads per processor core
    - Coarse-grained multithreading (aka “switch-on-event”)
    - Simultaneous multithreading (aka “hyper-threading”)
  - Multiple processor cores per die
    - Chip multiprocessors (CMP) or “Muticore processors” (MCP)
    - Chip multithreading (CMT)

Recall: Processes and Software Threads

- **Process**: an instance of a program executing in a system
  - OS supports concurrent execution of multiple processes
  - Each process has its own address space, set of registers, and PC
  - Two different processes can partially share their address spaces to communicate
- **Thread**: an independent control stream within a process
  - A process can have one or more threads
  - Private state: PC, registers (int, FP), stack, thread-local storage
  - Shared state: heap, address space (VM structures)
- A "**parallel program**" is one process but multiple threads

Reminder: Classic OS Context Switch

- OS context-switch
  - Timer interrupt stops a program mid-execution (precise)
  - OS saves the context of the stopped thread
    - PC, GPRs, and more
    - Shared state such as physical pages are not saved
  - OS restores the context of a previously stopped thread (all except PC)
  - OS uses a “return from exception” to jump to the restarting PC
    - The restored thread has no idea it was interrupted, removed, later restarted
  - Take a few hundred cycles per switch (why?)
    - Amortized over the execution “quantum”
- What latencies can you hide using OS context switching?
- How much faster would a user-level thread switch be?

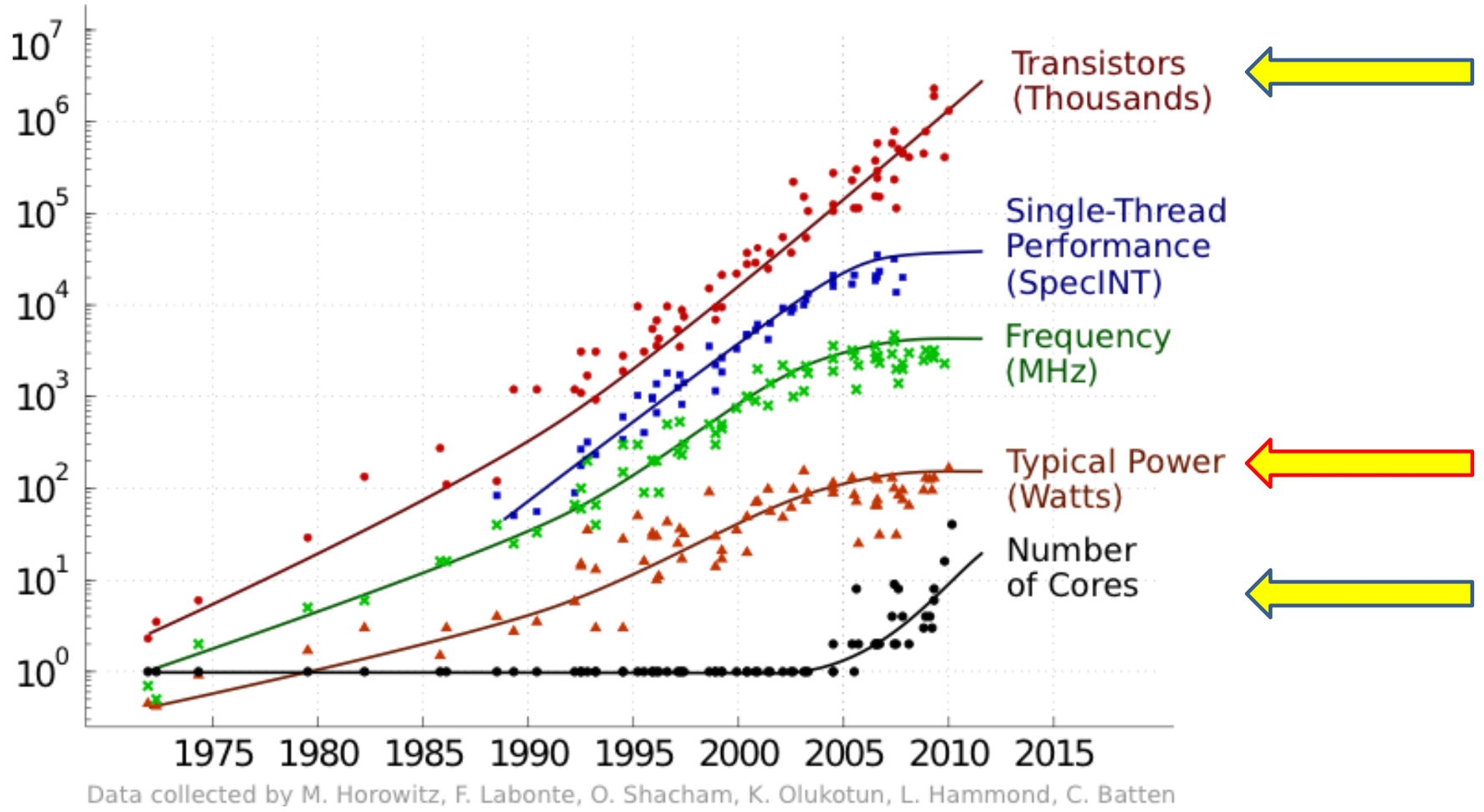
Multithreaded Cores (old "Multiprogramming")

- Basic idea:
  - CPU resources are expensive and should not be idle
- 1960's: Virtual memory and multiprogramming
  - Virtual memory/multiprogramming invented to tolerate latency to secondary storage (disk/tape/etc.)
  - Processor-disk speed mismatch:
    - microseconds to tens of milliseconds (1:10,000 or more)
  - OS context switch used to bring in other useful work while waiting for page fault or explicit file read/write accesses
  - Cost of context switch must be much less than I/O latency (easy)

Multithreaded Cores (new "Multithreading")

- 1990's: Memory wall and multithreading
  - Processor-DRAM speed mismatch:
    - nanosecond to fractions of a microsecond (1:500)
  - H/W task switch used to bring in other useful work while waiting for cache miss
  - Cost of context switch must be much less than cache miss latency
- Very attractive for applications with abundant thread-level parallelism
  - Commercial multi-user (transaction processing) workloads

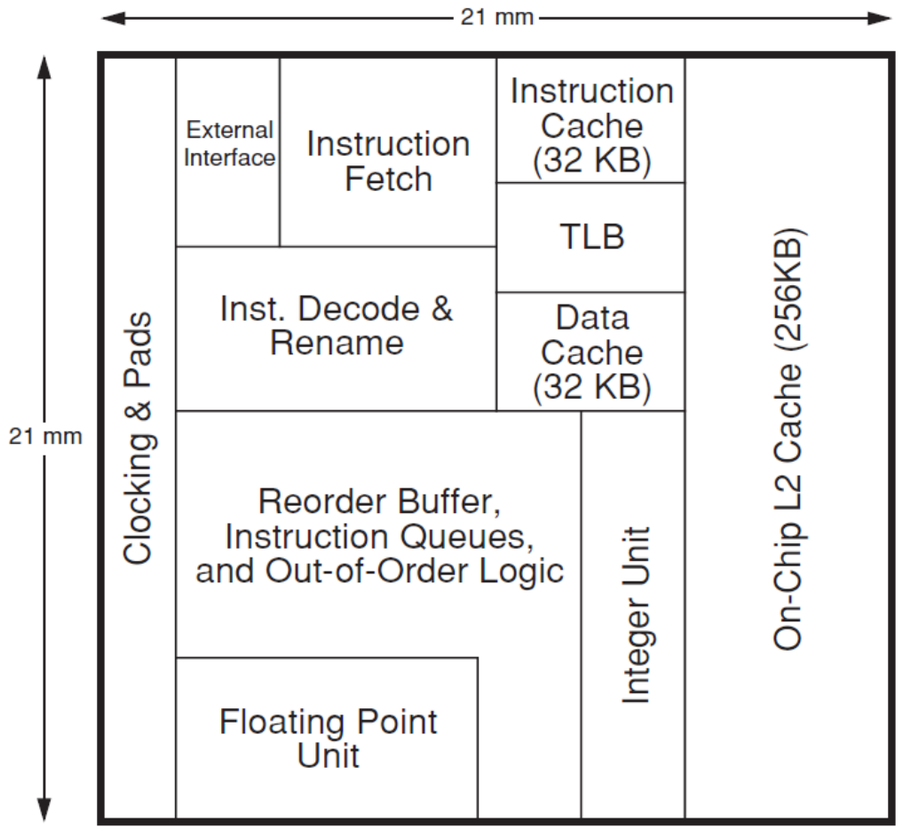
Processor Scaling Since ~2005

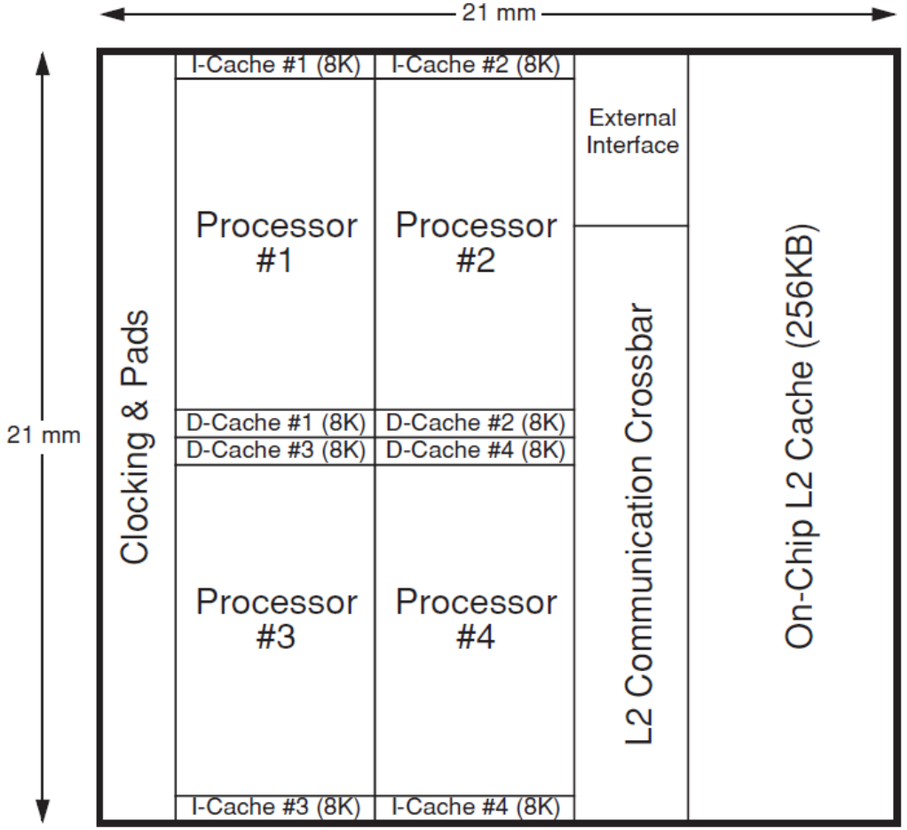


The Multicore Alternative

- Use Moore's law to place more cores per chip
  - Potentially 2x cores/chip with each CMOS generation
  - Without significantly compromising clock frequency
  - Known as Multi-Core Processors (MCP) or Chip Multiprocessors (CMP)
- The good news
  - Continued scaling of chip-level peak (throughput) performance
  - Mitigate the undesirable superscalar power scaling ("wrong side of the square law")
  - Facilitate design and verification, and product differentiation
- The bad news
  - Require multithreaded workloads: multiple programs or parallel programs
  - Require parallelizing single applications into parallel programs
  - Power is still an issue as transistors shrink due to leakage current

Big OOO Superscalar vs. Multicore Processor

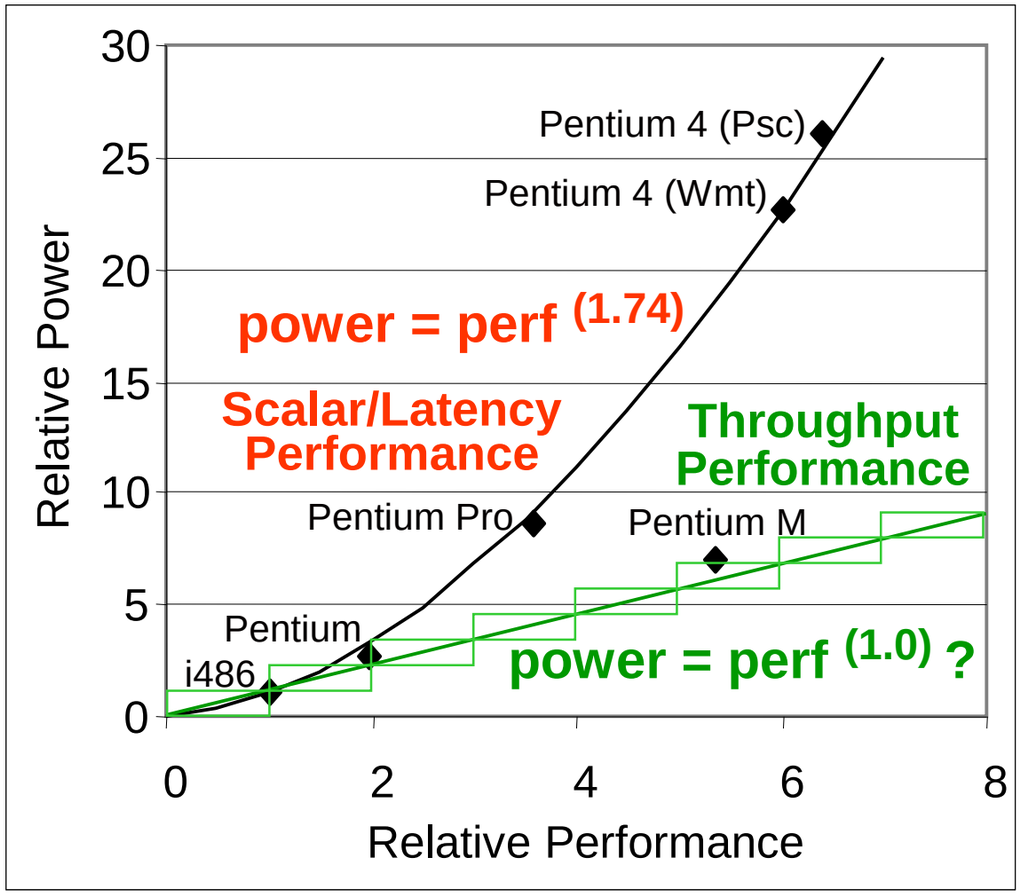




[Ed Grochowski, 2004]

Power & Throughput (Multi-Thread) Performance

- Assume a large-scale multicore processor (MCP) with potentially many cores
- Replication of cores results in nearly proportional increases to both throughput performance and power (hopefully).



Programming for Multicore Processors (MCP)

- Programmers must write parallel programs using threads/processes.
- Spread the workload across multiple cores at run time.
- OS will map threads/processes to cores at run time.

Assigning Threads to Cores:

- Each thread/process has an *affinity* mask
- Affinity mask specifies what cores the thread is allowed to run on.
- Different threads can have different masks
- Affinities are inherited across fork()

Shared-Memory Multiprocessors or Multicores

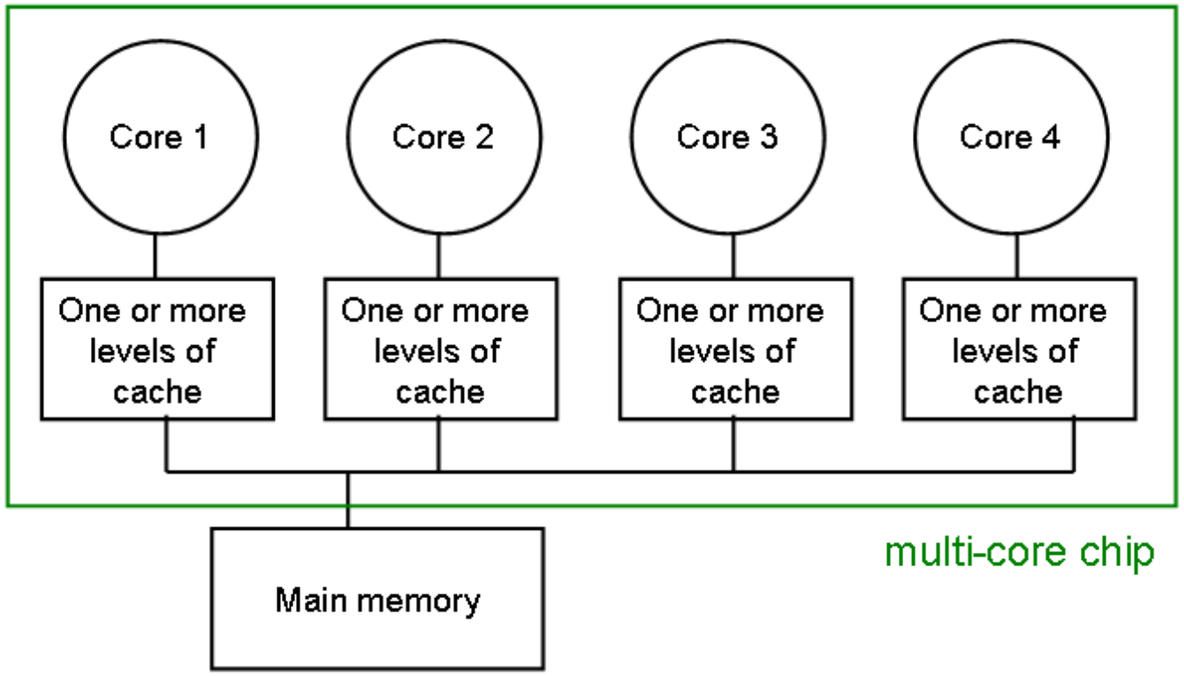
- All processor cores have access to unified physical memory
  - They can communicate via the shared memory using loads and stores
- Advantages
  - Supports multi-threading (TLP) using multiple cores
  - Requires relatively simple changes to the OS for scheduling
  - Threads within an app can communicate 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

Caches for Multicores (or Multicore Processors)

- Caches are (equally) helpful with multicores
  - Reduce access latency, reduce bandwidth requirements
  - For both private and shared data across cores
- Advantages of private caches:
  - They are closer to core, so faster access
  - Reduces contention to cache by cores
- Advantages of shared cache:
  - 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
- But multiple private caches introduce the two problems of
  - **Cache Coherence** (cover in this lecture)
  - **Memory Consistency** (beyond this course)

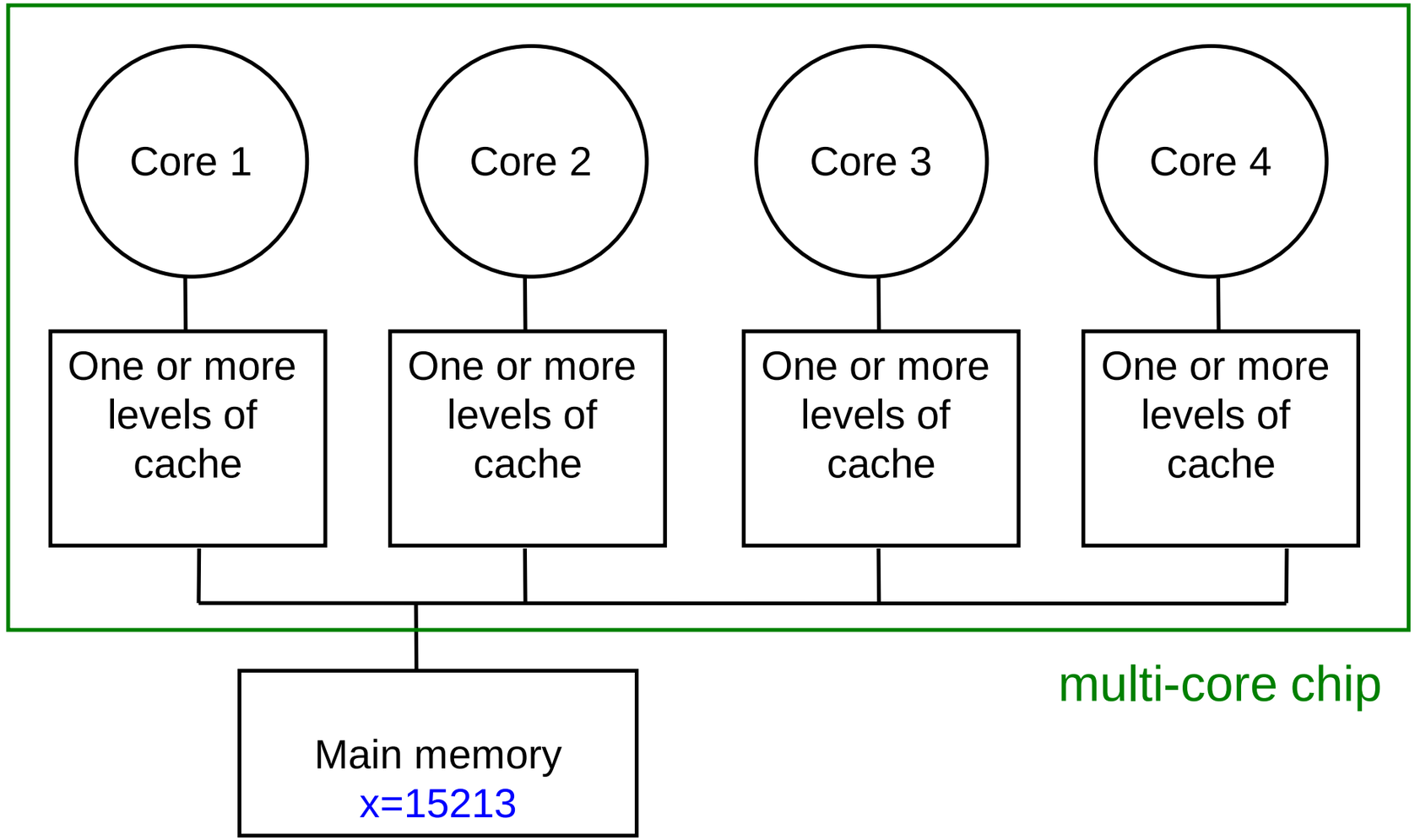
The Cache Coherence Problem

- 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



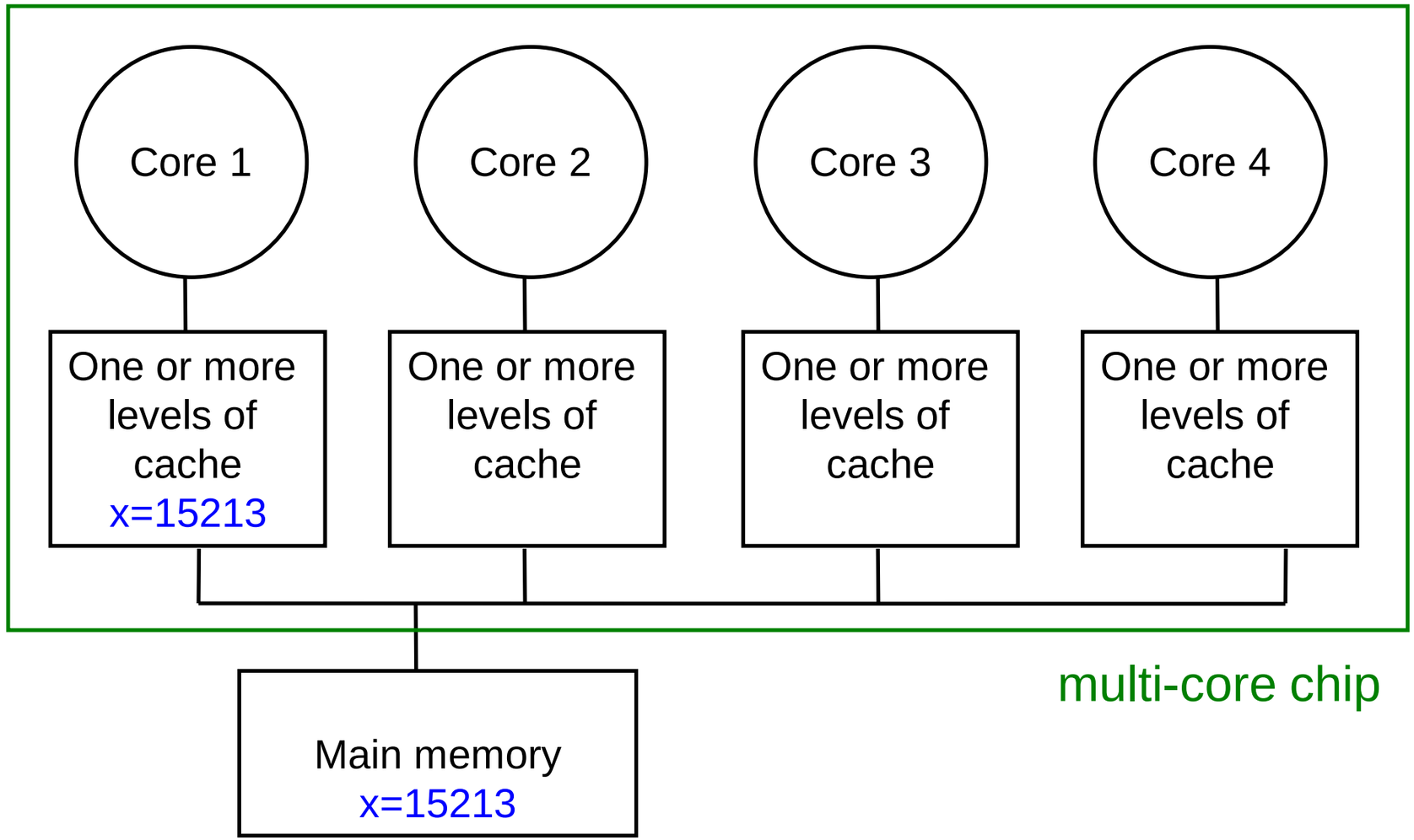
The Cache Coherence Problem

Suppose variable x initially contains 15213



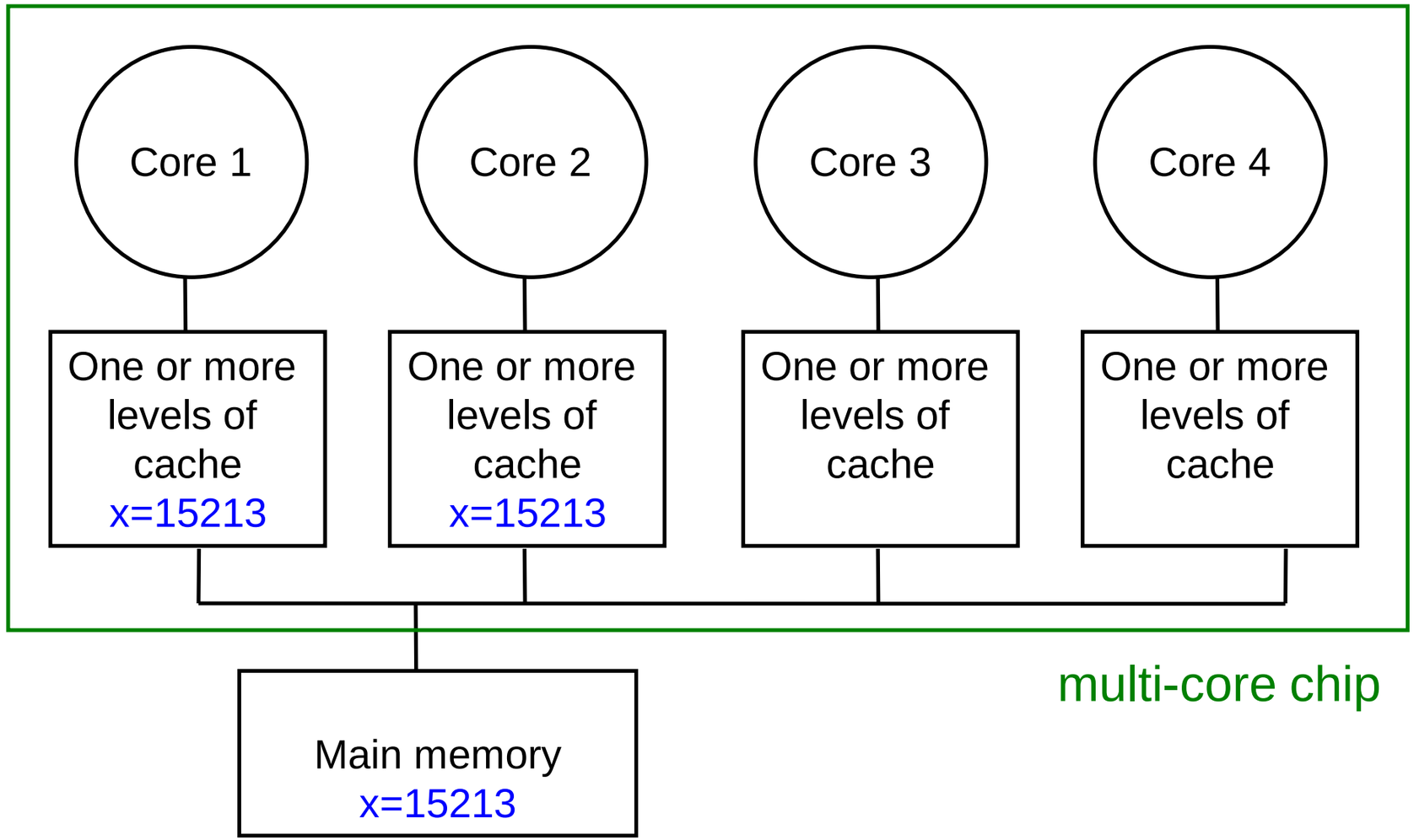
The Cache Coherence Problem

Core 1 reads x



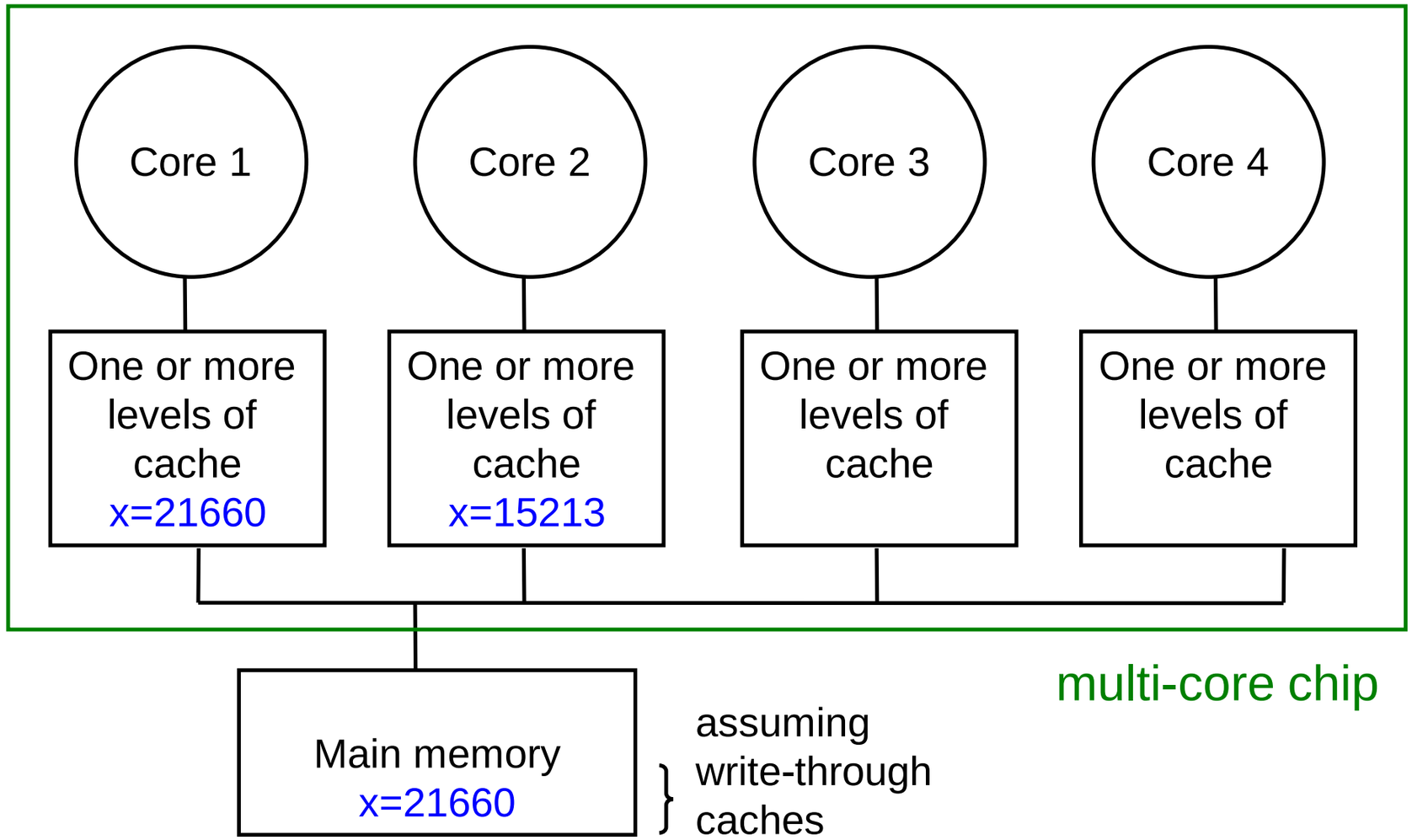
The Cache Coherence Problem

Core 2 reads x



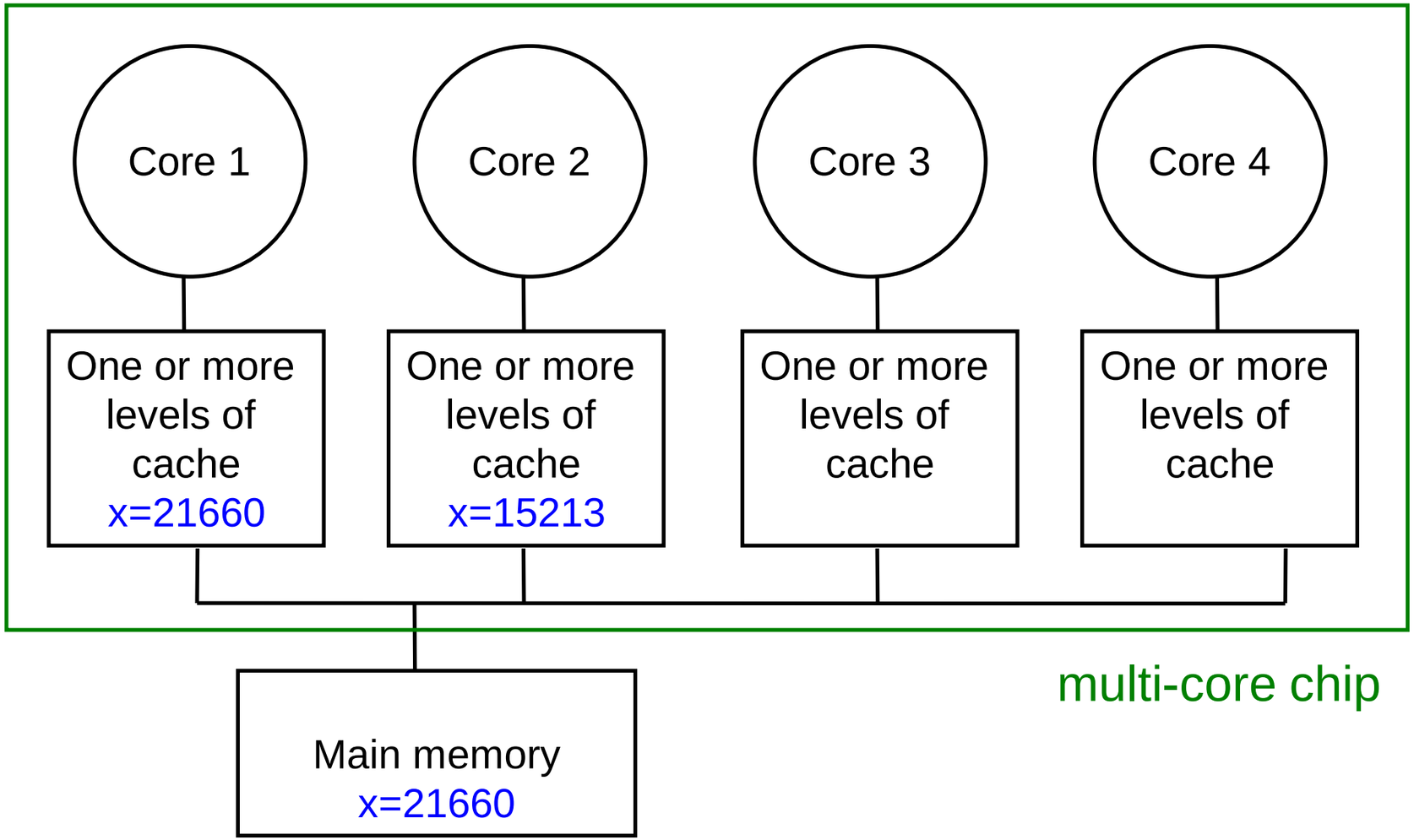
The Cache Coherence Problem

Core 1 writes to x, setting it to 21660



The Cache Coherence Problem

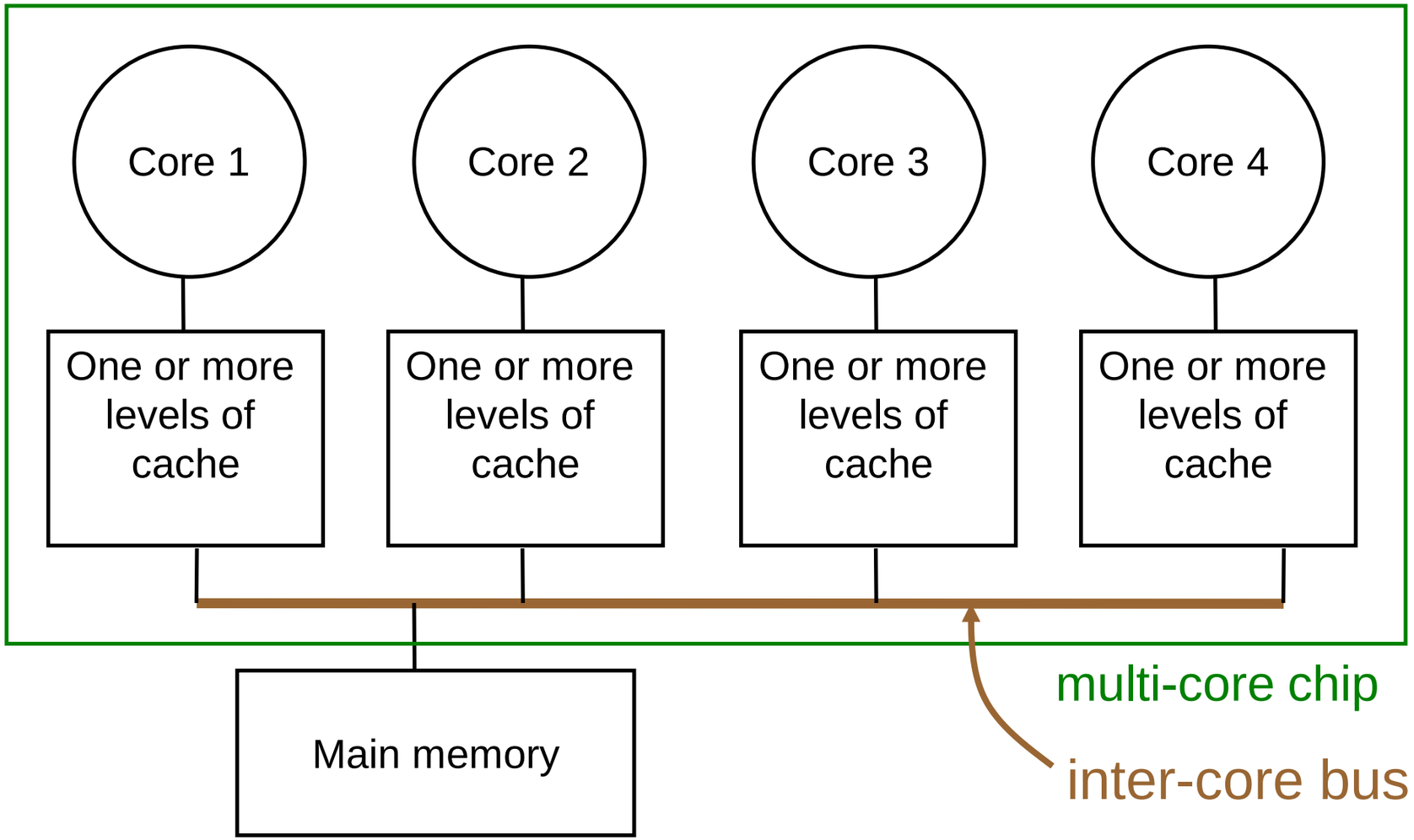
Core 2 attempts to read x… gets a stale copy



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 ("Snooping") Multicore Processor

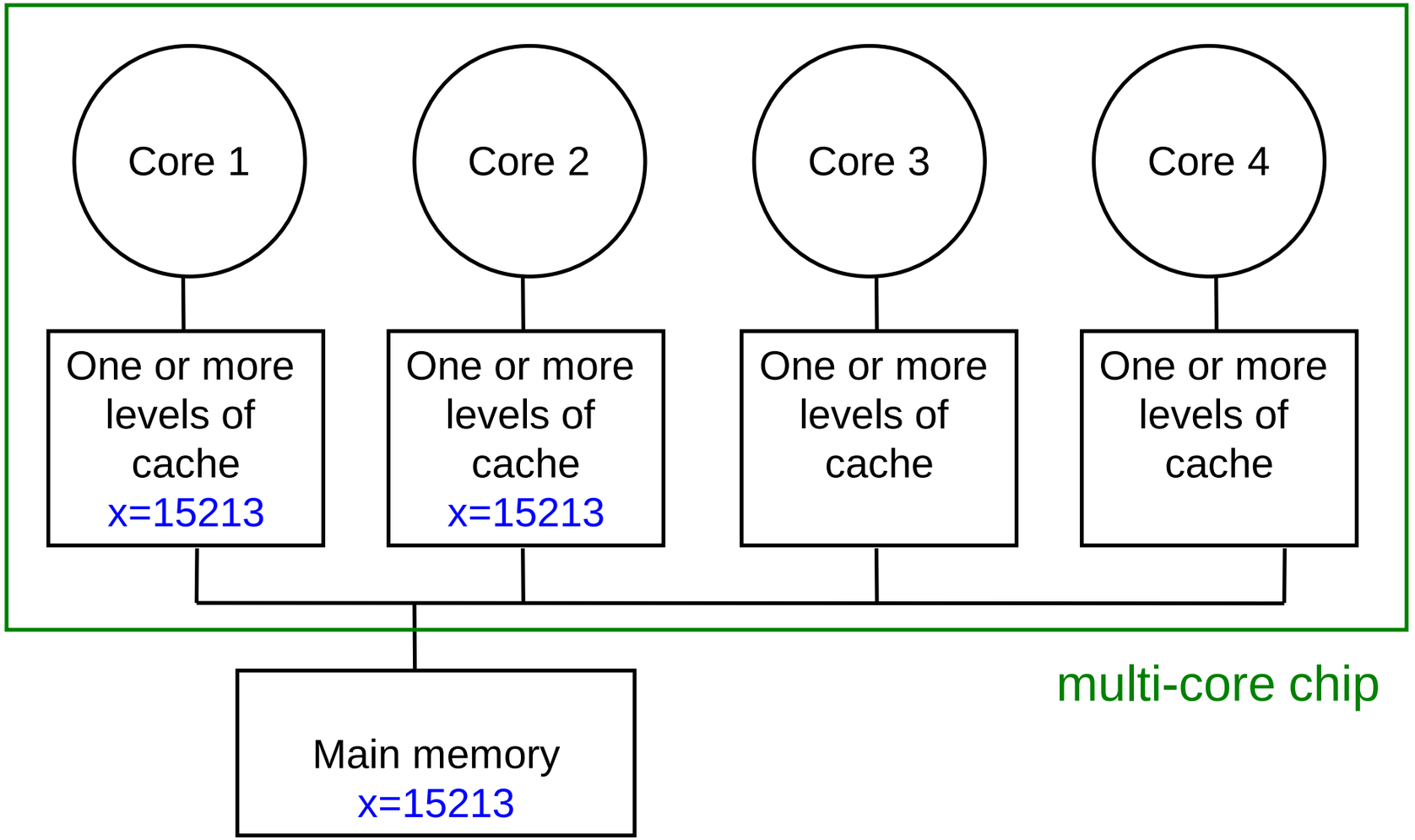


Invalidation Protocol with Snooping

- 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.

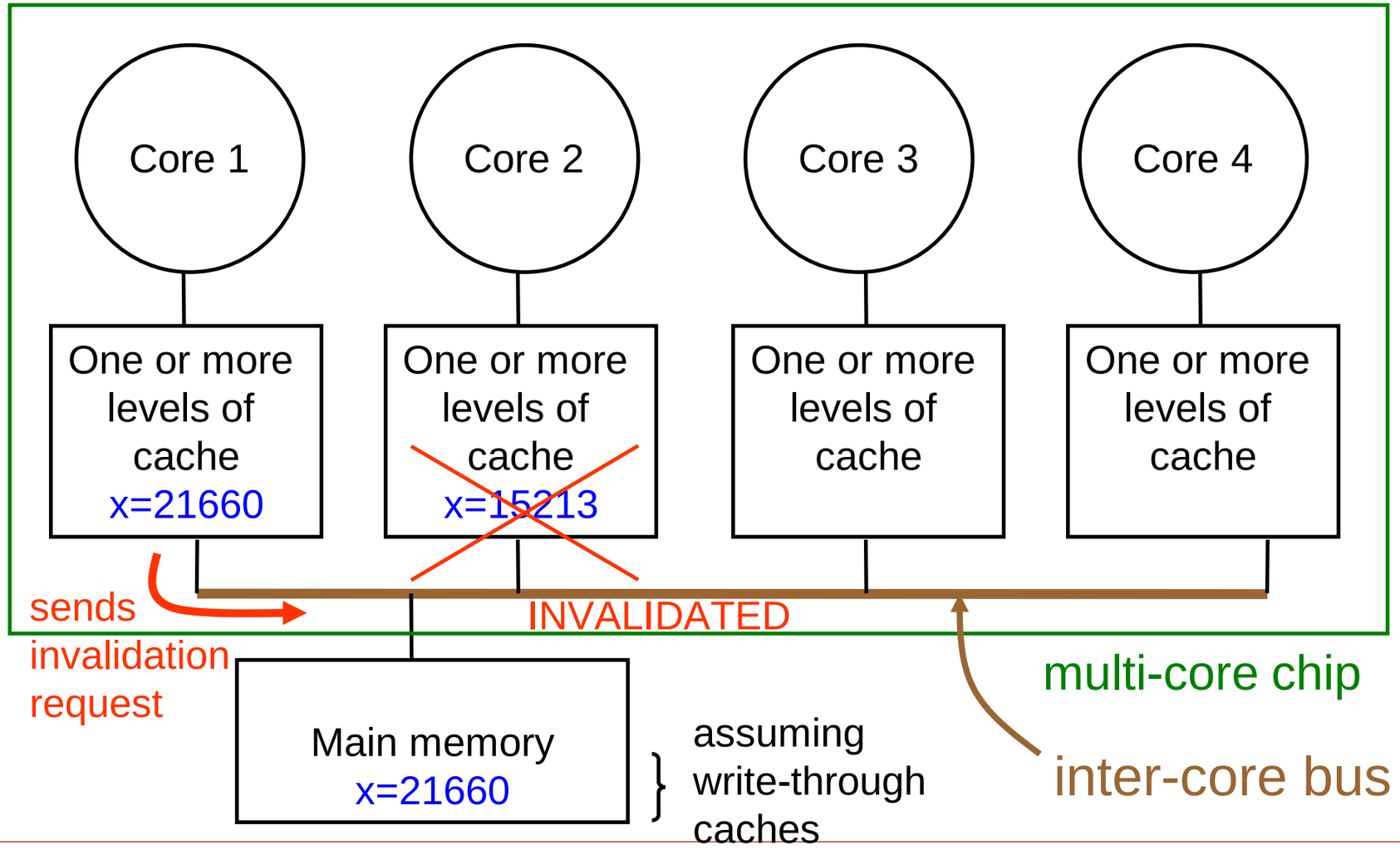
Invalidation Based Cache Coherence Protocol

Revisited: Cores 1 and 2 have both read x



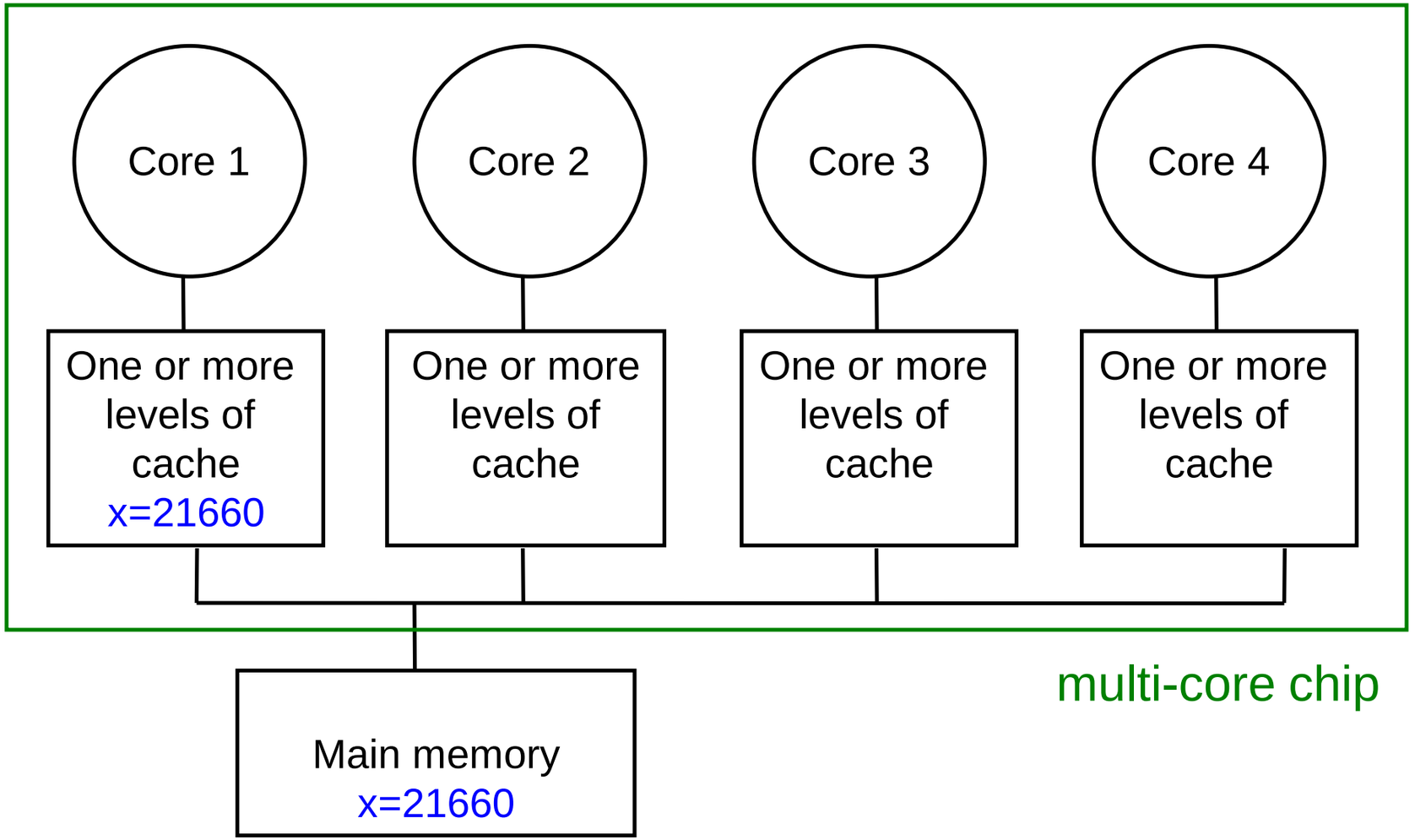
Invalidation Based Cache Coherence Protocol

Core 1 writes to x, setting it to 21660



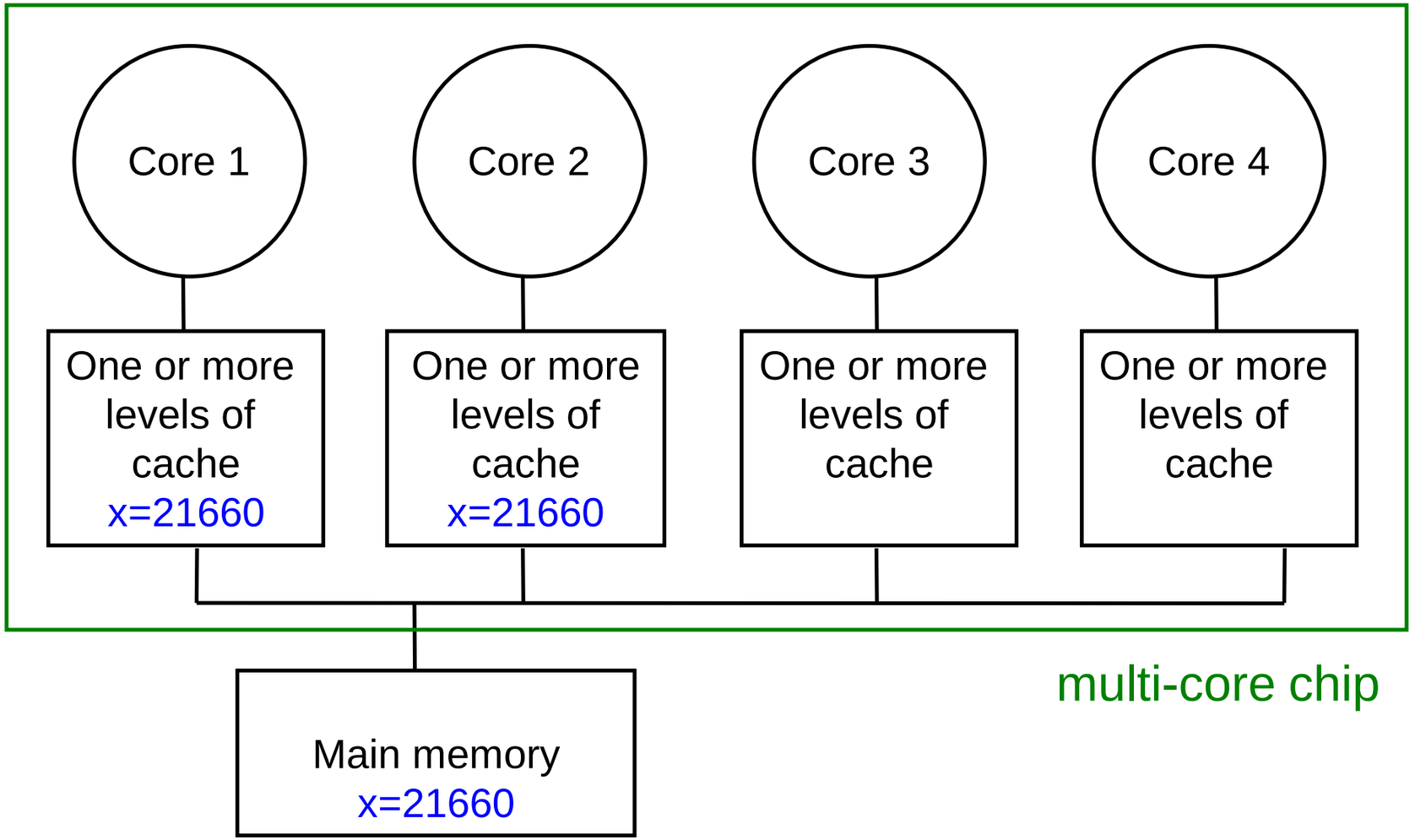
Invalidation Based Cache Coherence Protocol

After invalidation:



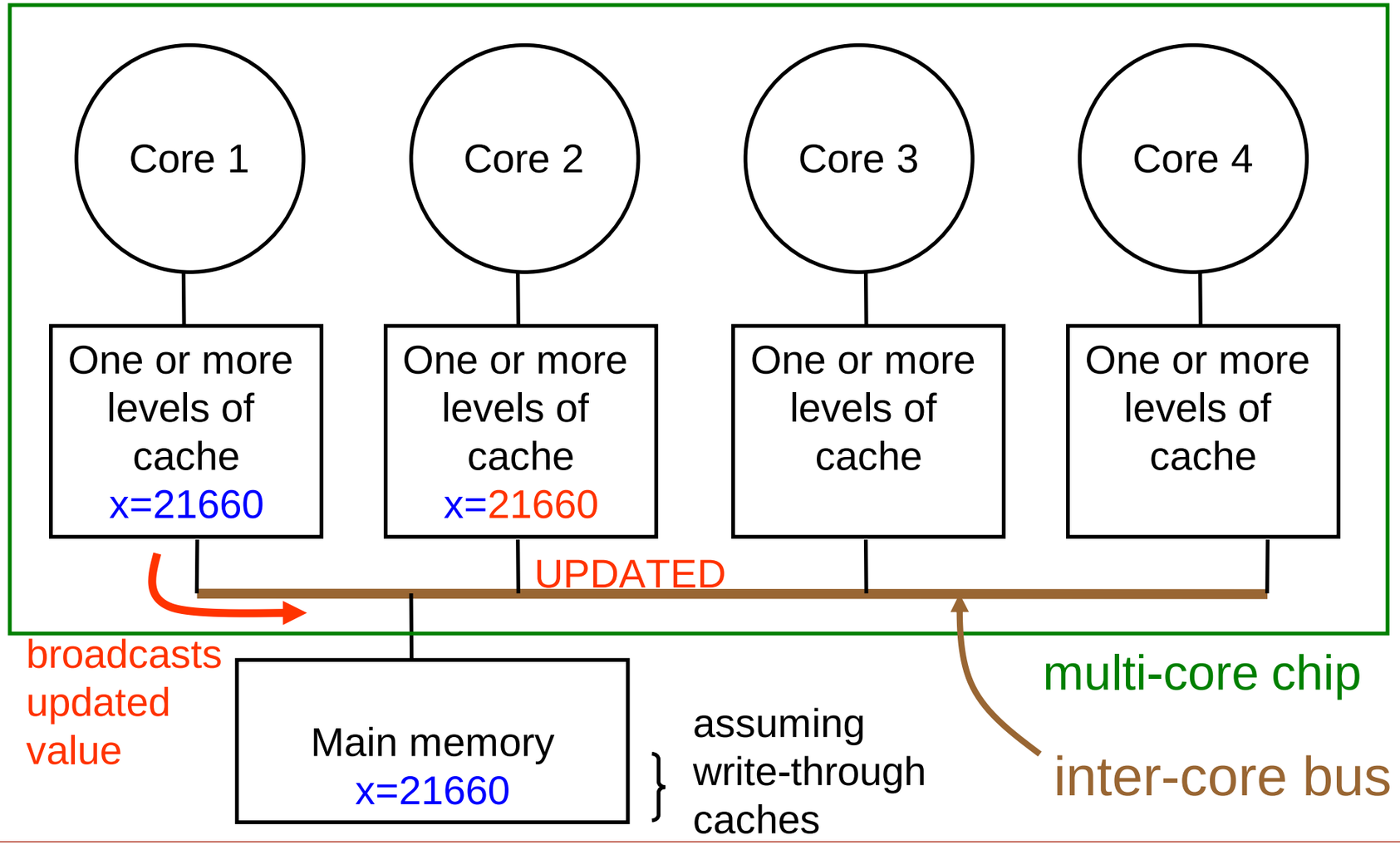
Invalidation Based Cache Coherence Protocol

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

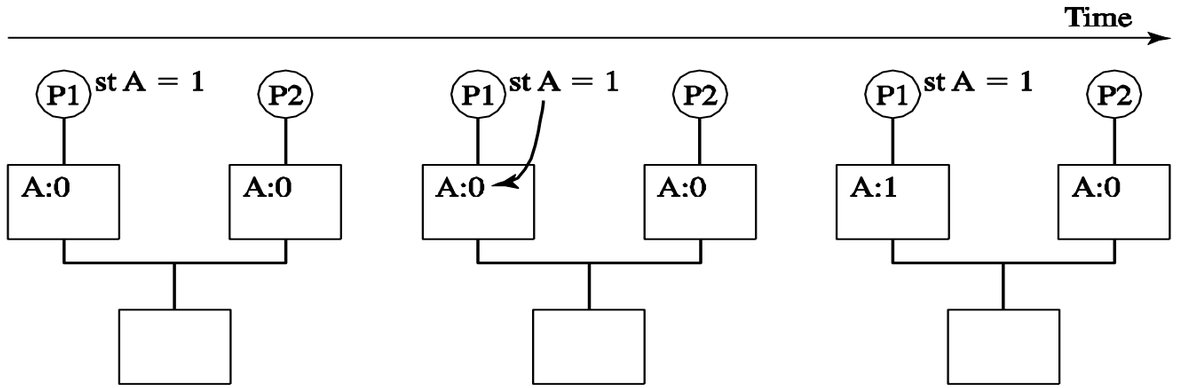
Cache Coherence

- Informally, with coherent caches: accesses to a memory location *appear* to occur simultaneously in all copies of that memory location

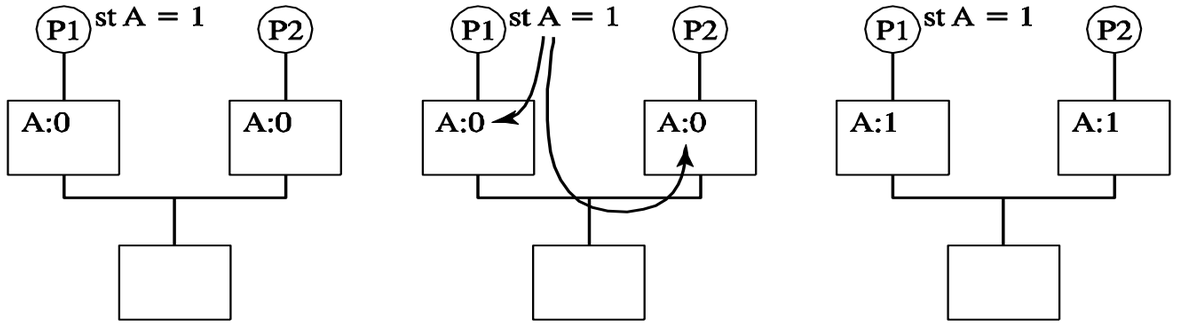
  "copies" $\Rightarrow$ caches + memory
- Cache coherence suggests an absolute time scale -- this is not necessary
  - What is required is the "appearance" of coherence... not absolute coherence
  - E.g. temporary incoherence between memory and a write-back cache may be OK.

Write Update vs. Write Invalidate

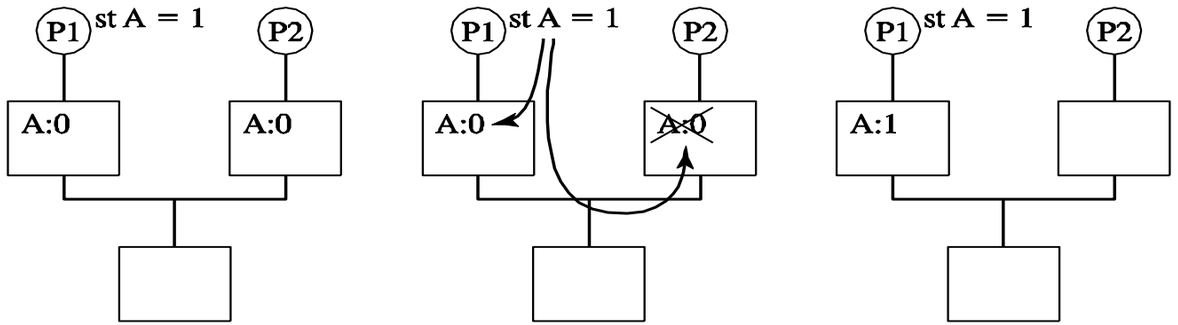
- Coherent caches with Shared Memory
  - All cores see the effects of others' writes
- How/when writes are propagated
  - Determined by coherence protocol



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



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



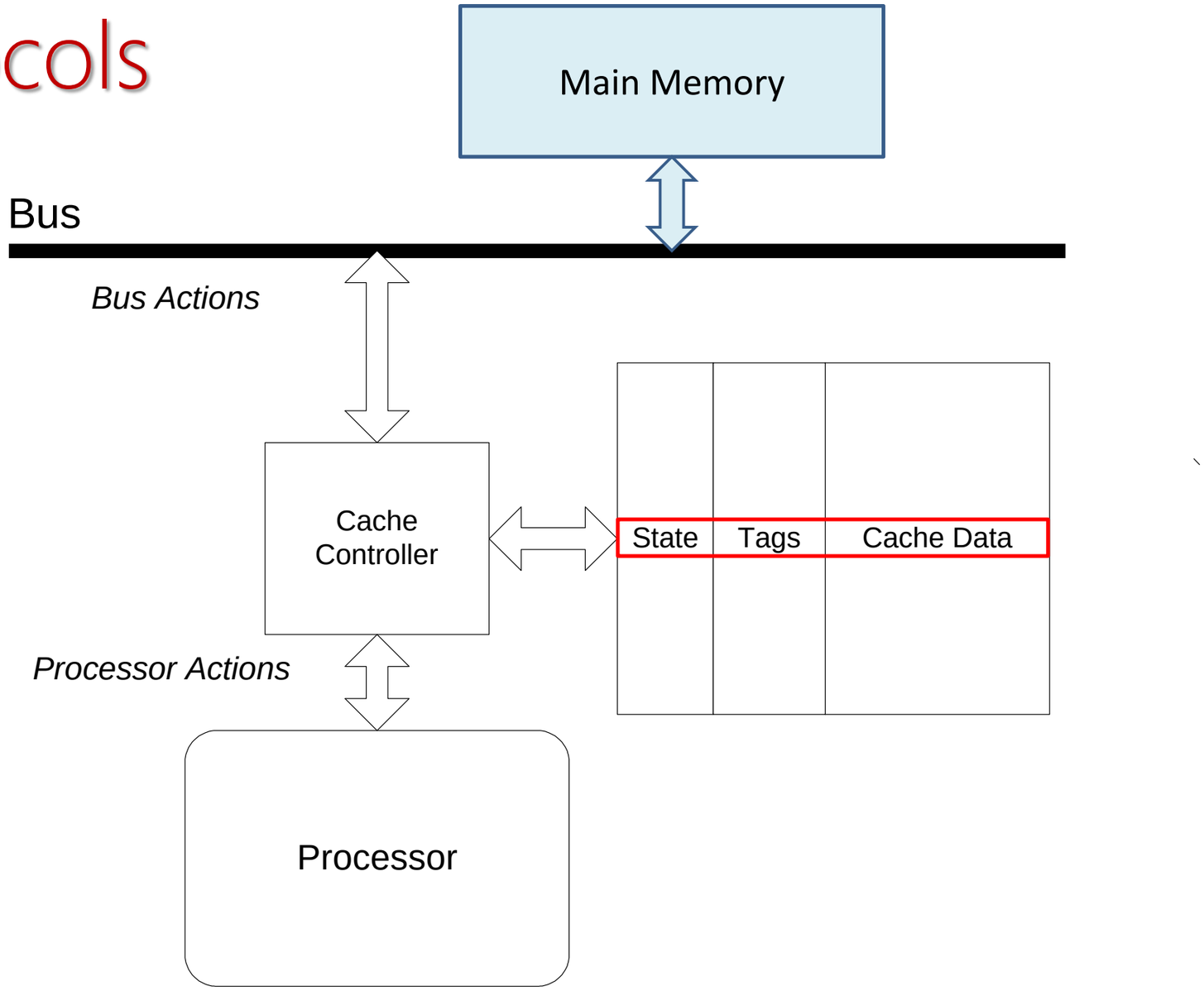
(c) Invalidate protocol eliminates stale remote copy

Bus-Based Snoopy Cache Coherence

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

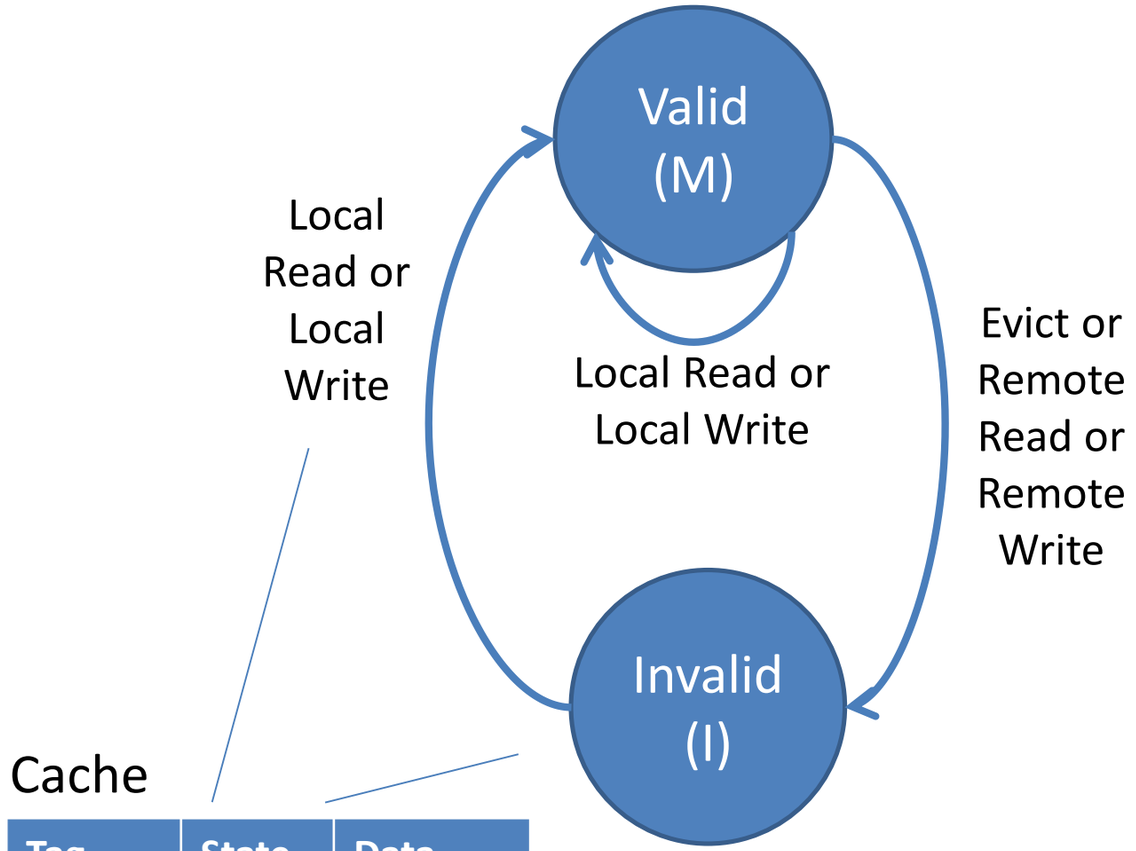
Bus-Based Protocols

- 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 (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



Cache

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

MSI Protocol (with Write Back Cache)

	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

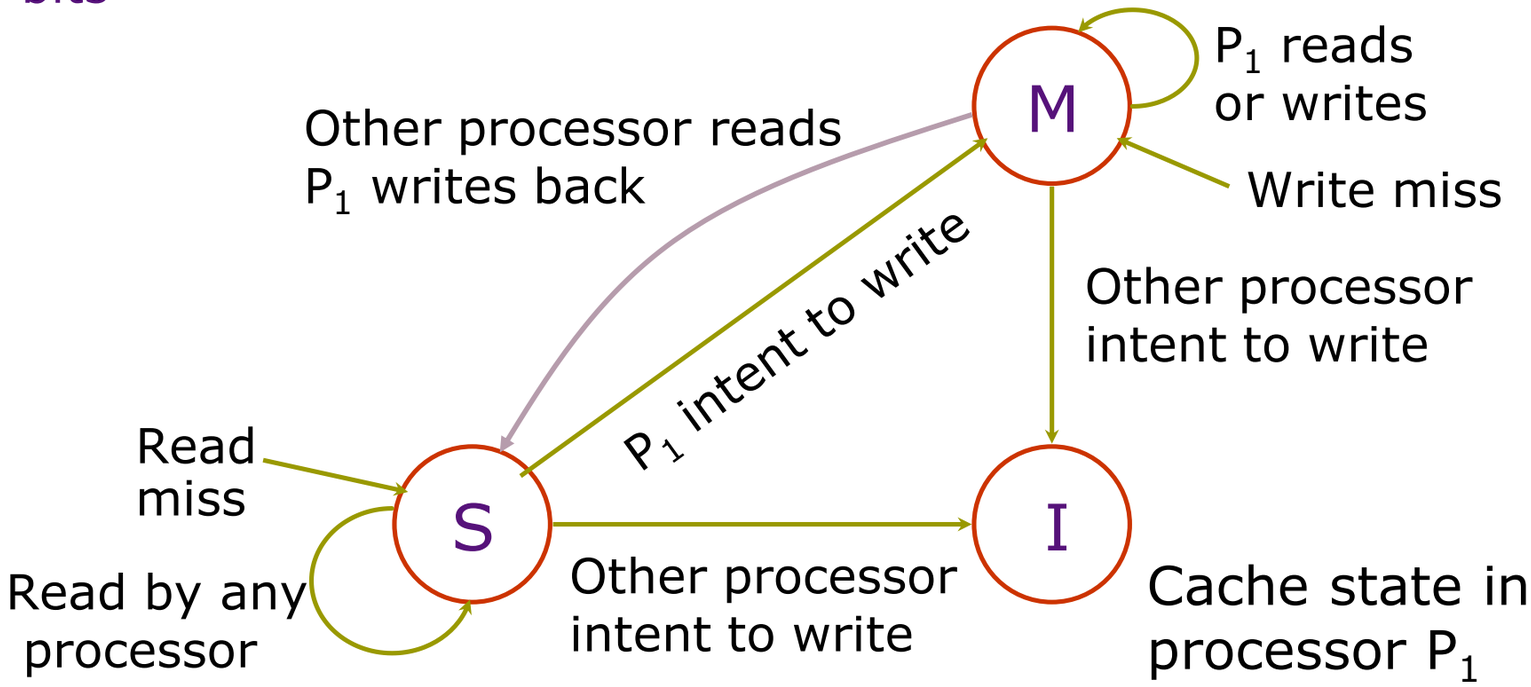
Thread Event	Bus Action	Data From	Global State	Local States: C0	C1	C2
0. Initially:			<0,0,0,1>	I	I	I
1. T0 read→	CR	Memory	<1,0,0,1>	S	I	I
2. T0 write→	CU		<1,0,0,0>	M	I	I
3. T2 read→	CR	C0	<1,0,1,1>	S	I	S
4. T1 write→	CRM	Memory	<0,1,0,0>	I	M	I

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

MSI: A Coherence Protocol (Write Back Caches)

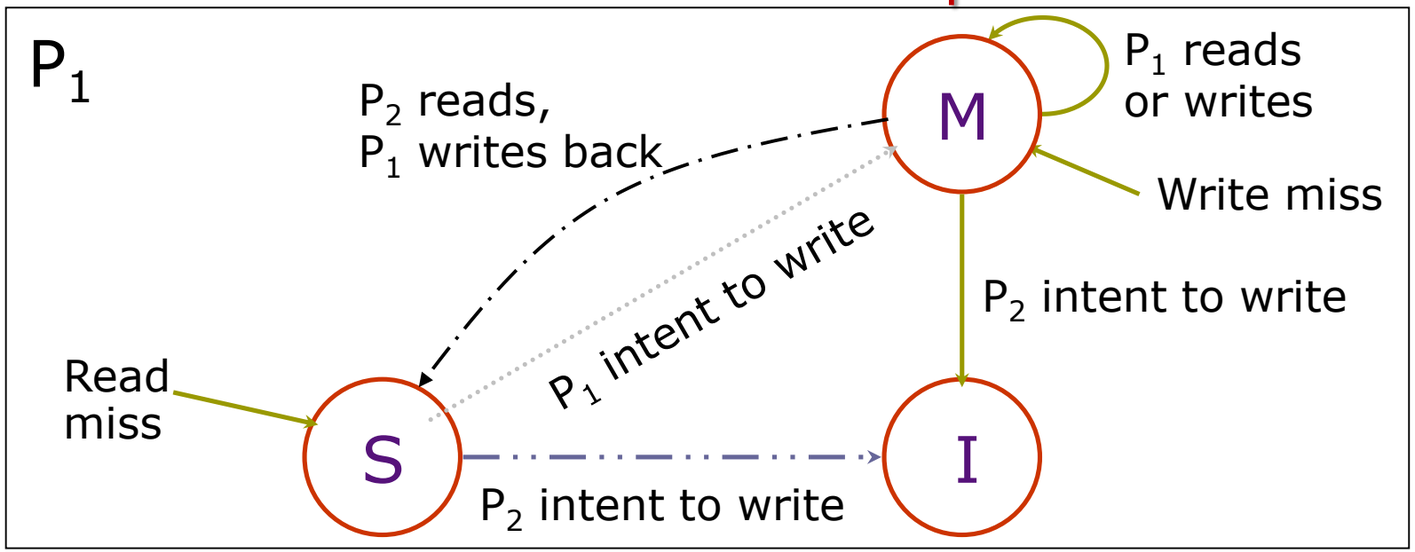
Each cache line has a tag

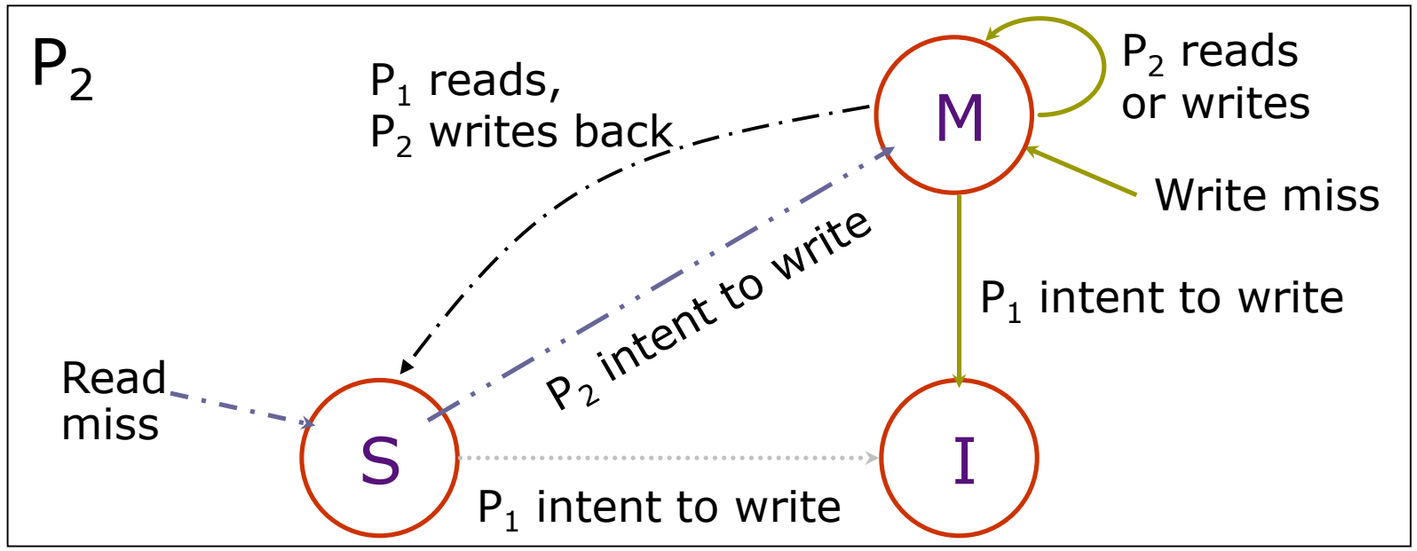
M: Modified
S: Shared
I: Invalid



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 (most common in industry)

- 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 (IEEE Standard)

- 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

The Memory Consistency Problem: Example

P_1	P_2
/*Assume initial value of A and flag is 0*/	
`A = 1;`	`while (flag == 0);` /*spin idly*/
`flag = 1;`	`print A;`

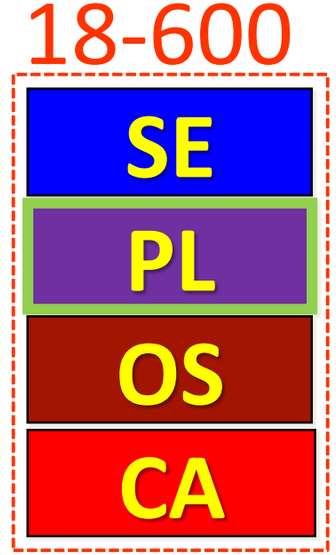
- Intuitively, you expect to print A=1
  - But can you think of a case where you will print A=0?
  - Even if cache coherence is available
- **Coherence** talks about accesses to a single location
- **Consistency** is about ordering for accesses to difference locations
- Alternatively
  - **Coherence** determines what value is returned by a read
  - **Consistency** determines when a write value becomes visible

18-600 Foundations of Computer Systems

Lecture 18: "Program Performance Optimizations"

John P. Shen & Gregory Kesden

November 1, 2017



Next Time ...

➢ Required Reading Assignment:
- **Chapter 5 of CS:APP (3rd edition) by Randy Bryant & Dave O'Hallaron.**

Electrical & Computer ENGINEERING