Summary and Comparison

	Multi-issue	SIMD	SMT	Multi-core
Parallelism	ILP	DLP	TLP	TLP
Datapath	Shared IF/PC/register file (RF), multiple datapaths for different types of instructions	Multiple processing elements/ ALUs, indepedent vector RF	Shared ALU, multiple PC/register files	Indepedent multiple full datapaths
Core	Within single core	Within single core	Within single core, but looks like multi-core (multiple logical cores)	Multiple (phsical) cores or full datapaths (IF, PC, RF, ALU, etc.)
Main issues	Combined with pipeline, may lead to data hazards	Data should be indepedent	Requires Synchronization	Requires Synchronization

信息科学与技术学院
School of Information Science and Technology

CS 110 Computer Architecture Advanced Cache

Instructors:

Chundong Wang, Siting Liu & Yuan Xiao

Course website: https://toast-lab.sist.shanghaitech.edu.cn/courses/CS110@ShanghaiTech/Spring-2025/index.html

School of Information Science and Technology (SIST)

ShanghaiTech University

2025/5/20

Administratives

- Final exam, June 12th 8am-10am; you can bring **3**-page A4-sized double-sided cheat sheet, **handwritten** only! (Teaching center 201/202/203); the whole course will be covered.
- Project 3 released. Speed Competition! ddl May 29th.
- Project 4 released, ddl June 3rd.
- HW 7 released, ddl extended since it is not covered yet, May 30th.
- To check Lab 13 this week, May 20th, 22nd & 26th
- Lab 14 released, to check May 27th, 29th & June 4th (Lab Session 1 only, 1D104); Prepare in advance!
- Discussion May 23rd & 26th on *OS & Virtual memory*.

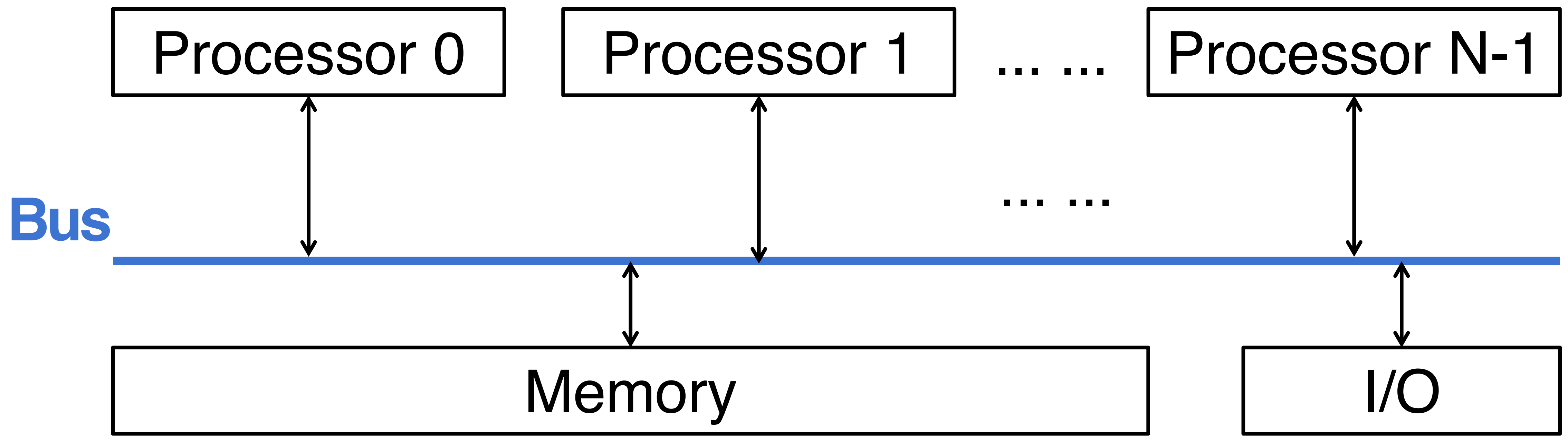
Key Design Questions

- How many processors (or cores) should be supported in this multiprocessor?
  - Depends on the target workload!
  - Most systems: Multiple “best available single core within constraints”
  - Power-critical systems (e.g., phones): “some of the best available single cores” and “some of the most power efficient single cores”
- How do different processors coordinate/communicate?
  - Shared variables in memory and load/store instructions
  - Coordinated access to shared data through synchronization primitives (e.g., locks) that restrict access to one processor at a time
- How do different processors (cores) share data?
  - Via shared-memory

Effectively all multicore computers today use shared memory.

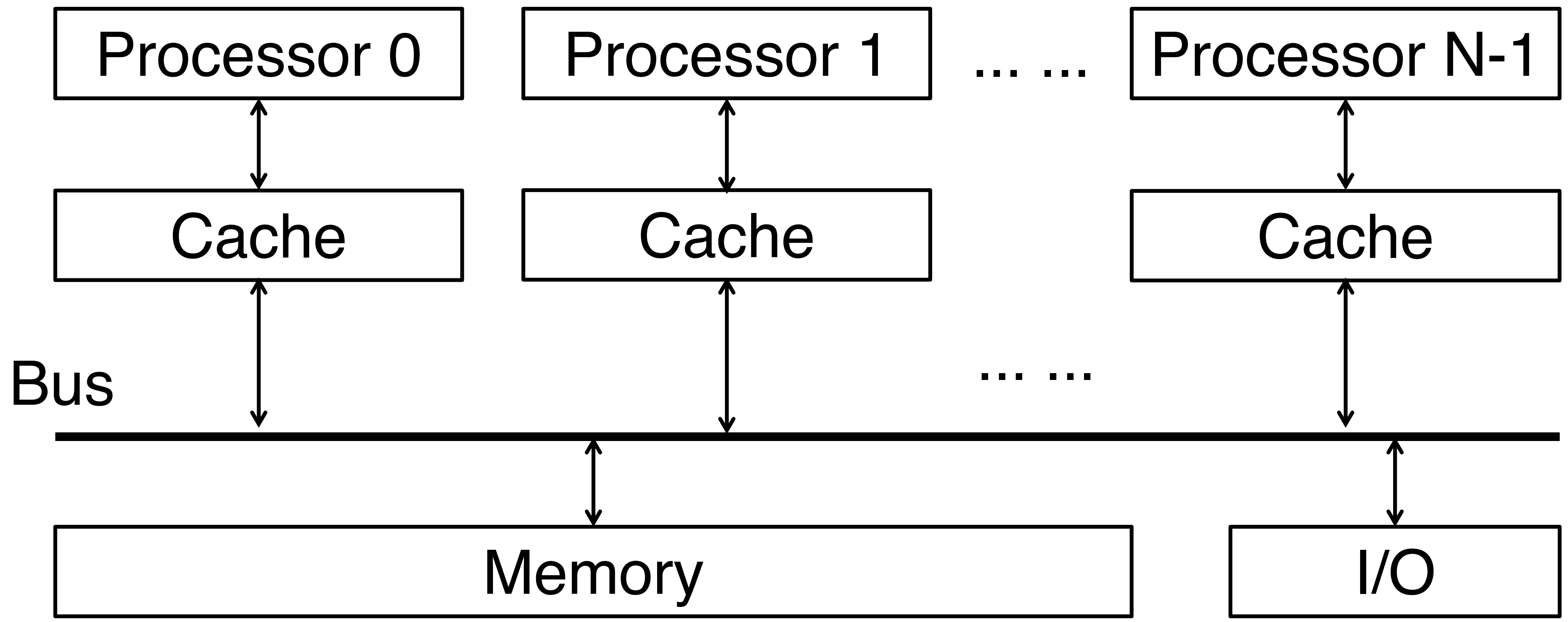
Multiprocessor with Shared-memory

- A **multiprocessor with shared-memory** offers multiple cores/ processors a single, shared, coherent memory.
  - Should be called shared-address multiprocessor, because all processors share single physical address space (more later, VM)

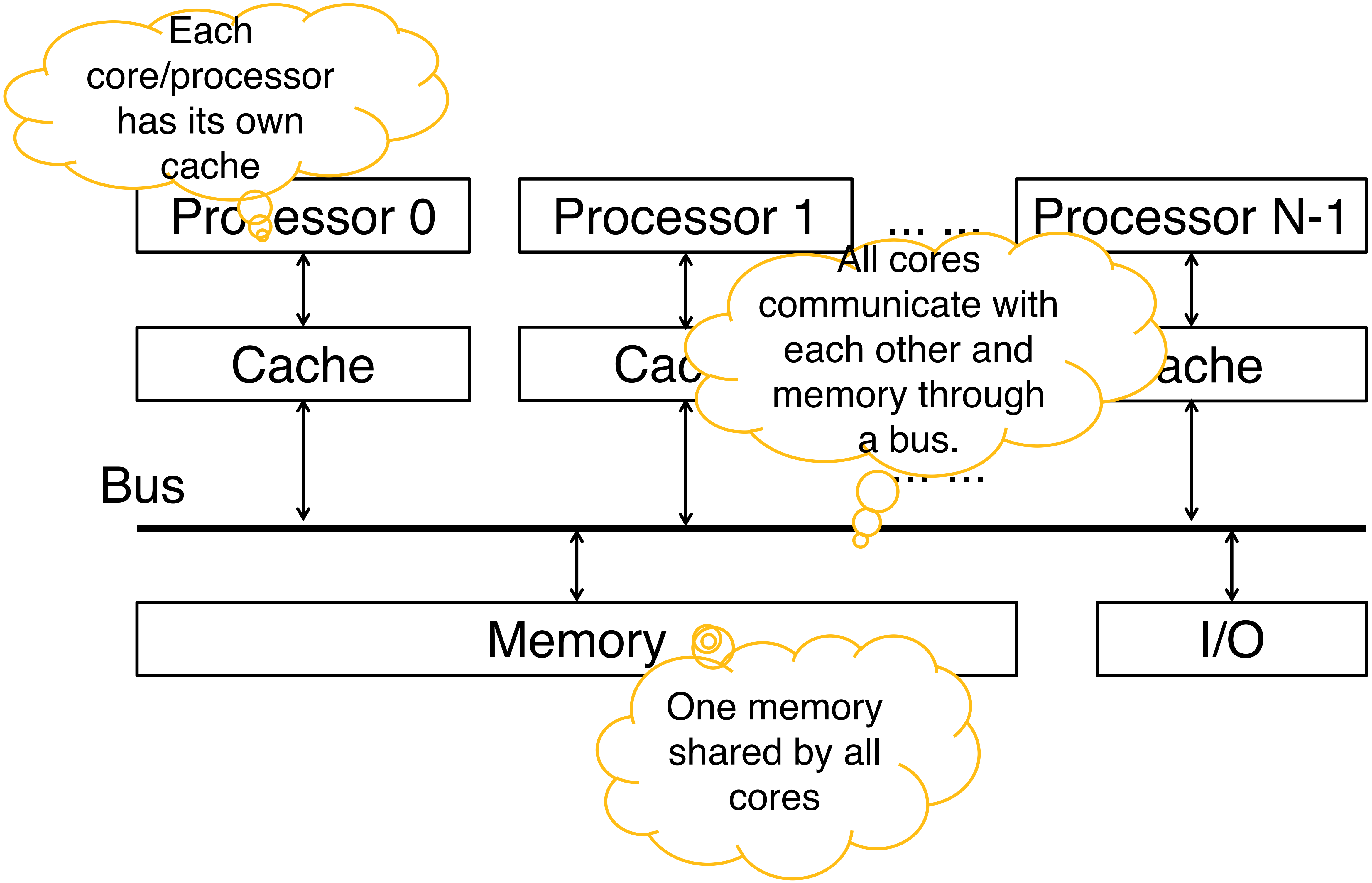


Multiprocessor (Multicore) Cache

- Memory is a performance bottleneck even with one processor.
- Use private **caches** to reduce bandwidth demands on main memory!
- Only cache misses have to access the **shared** common memory

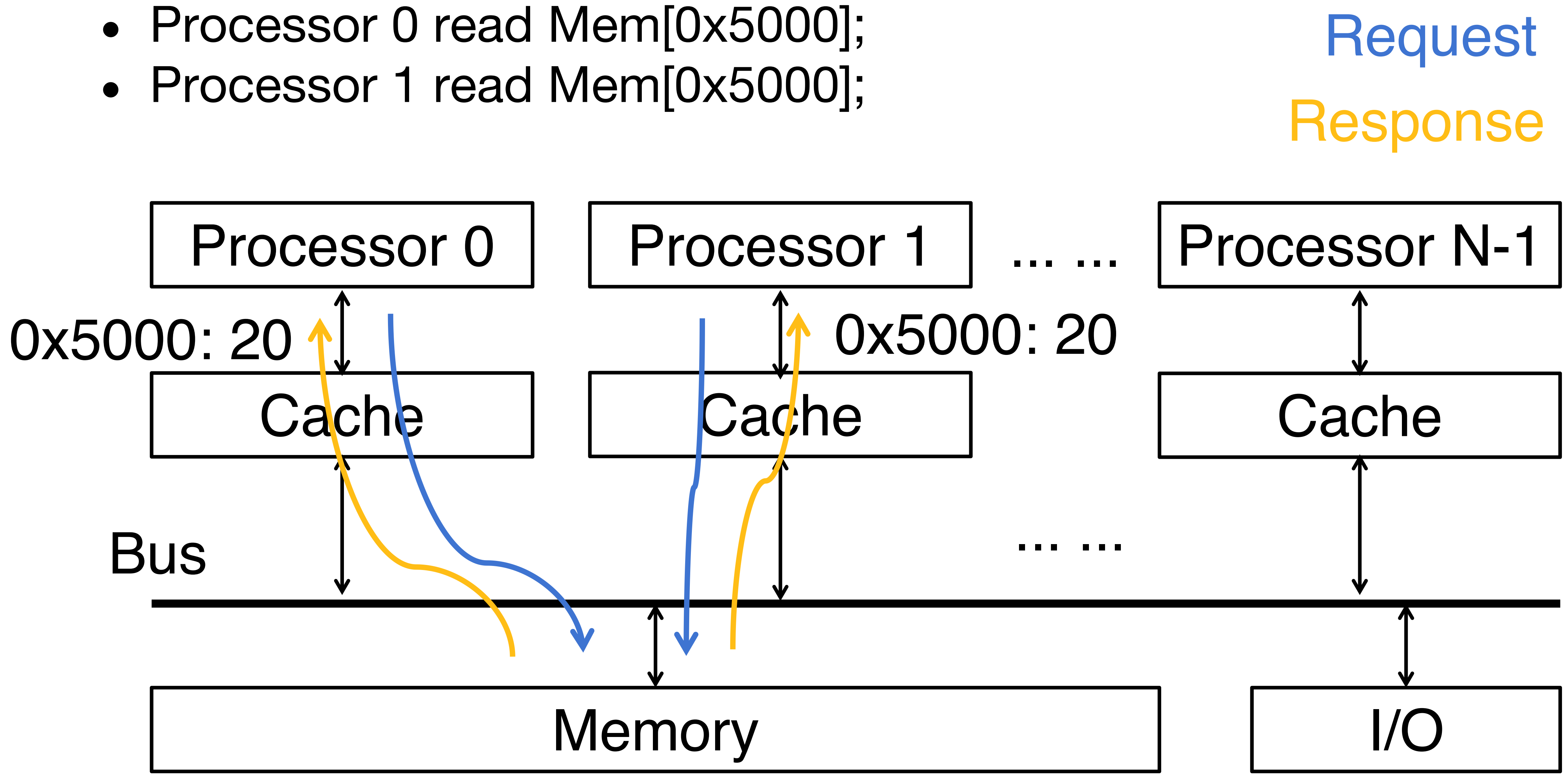


Multiprocessor Cache


Each core/processor has its own cache
Processor 0
Processor 1
... ...
Processor N-1
Cache
Cache
Cache
All cores communicate with each other and memory through a bus.
Bus
... ...
Memory
I/O
One memory shared by all cores

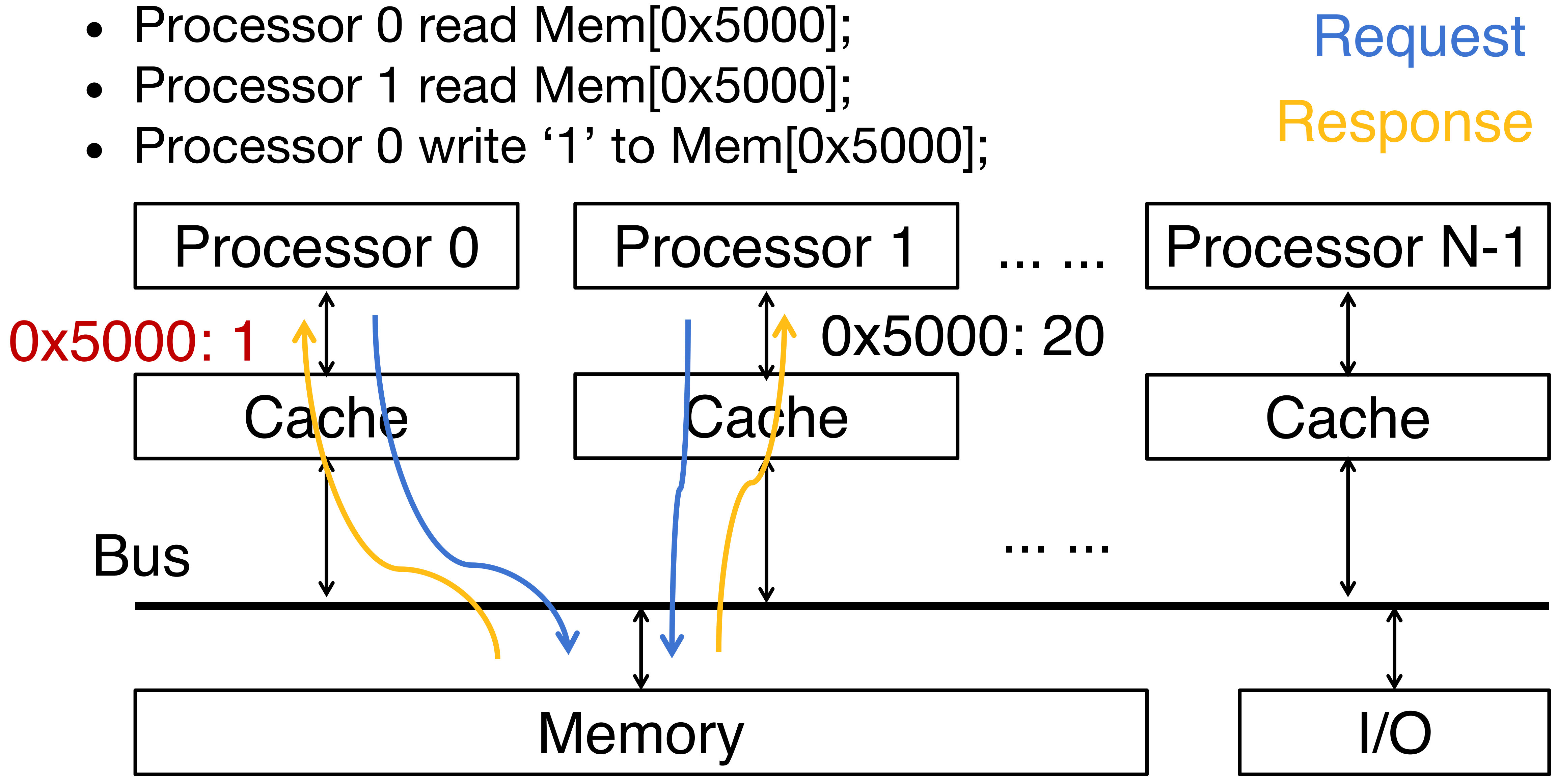
Multiprocessor Cache

- Consider the following scenario
  - Assume value “20” initially @ Mem[0x5000]):
  - Processor 0 read Mem[0x5000];
  - Processor 1 read Mem[0x5000];



Multiprocessor Cache

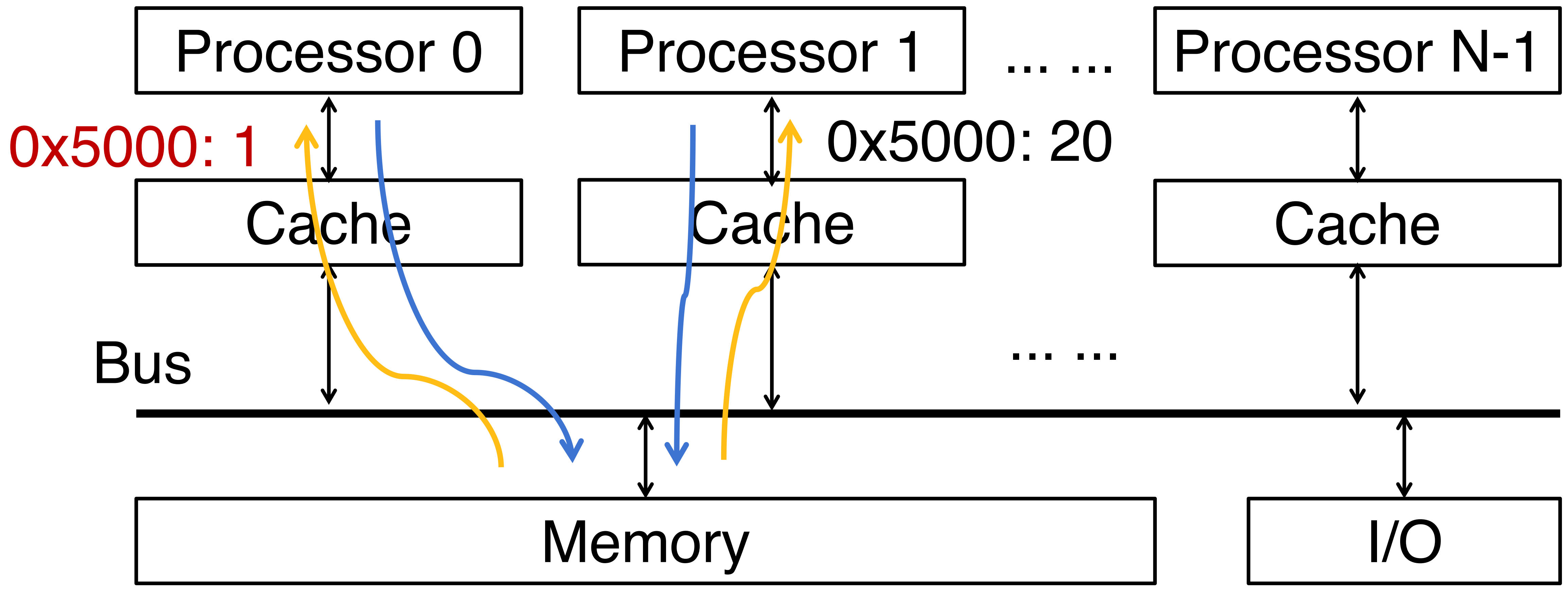
- Consider the following scenario
  - Assume value “20” initially @ Mem[0x5000]):
  - Processor 0 read Mem[0x5000];
  - Processor 1 read Mem[0x5000];
  - Processor 0 write ‘1’ to Mem[0x5000];



Cache (In)coherence

- Consider the following scenario
  - Assume value "20" initially @ Mem[0x
  - Processor 0 read Mem[0x5000];
  - Processor 1 read Mem[0x5000];
  - Processor 0 write '1' to Mem[0x5000];

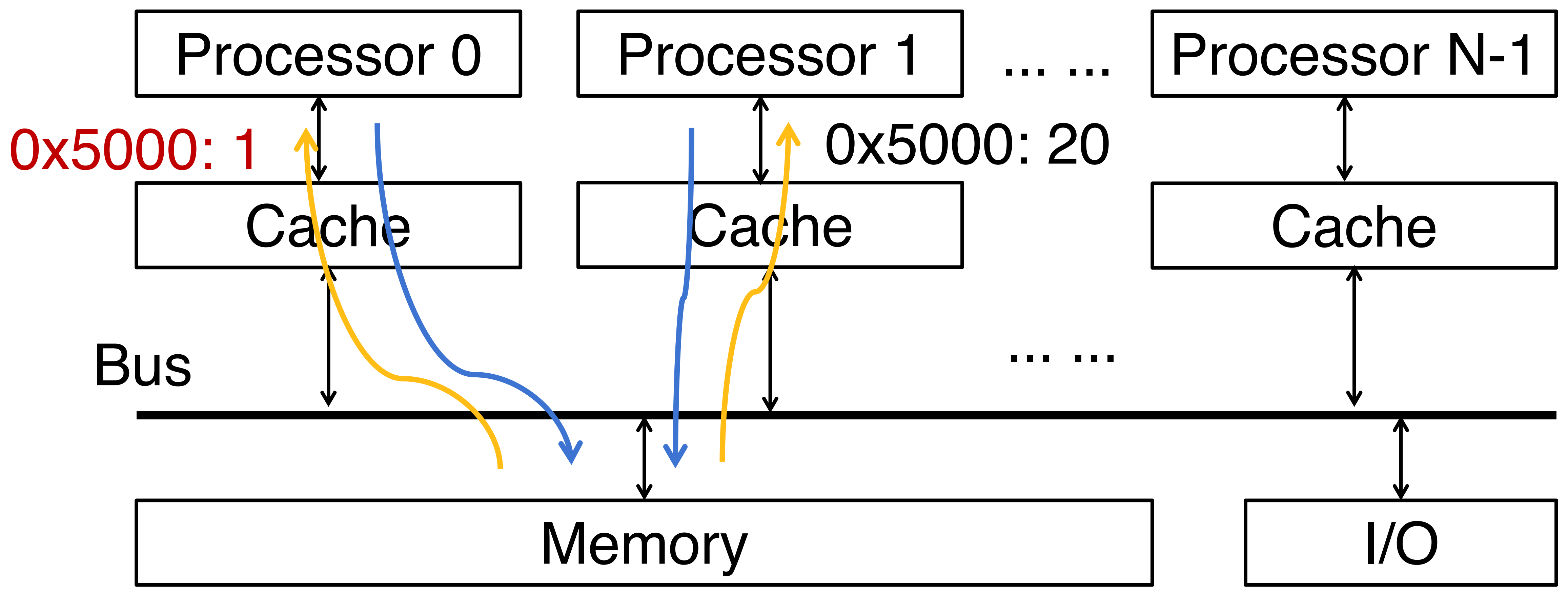
New cache miss type: **coherence miss** (a.k.a. communication miss), caused by writes to shared data made by other processors.



- For some parallel programs, coherence misses can dominate total misses;
- The 4th "C" of cache misses

Cache (In)coherence

- The processor 0 write invalidates other copies in other processors' caches.



Cache Coherence and Snooping

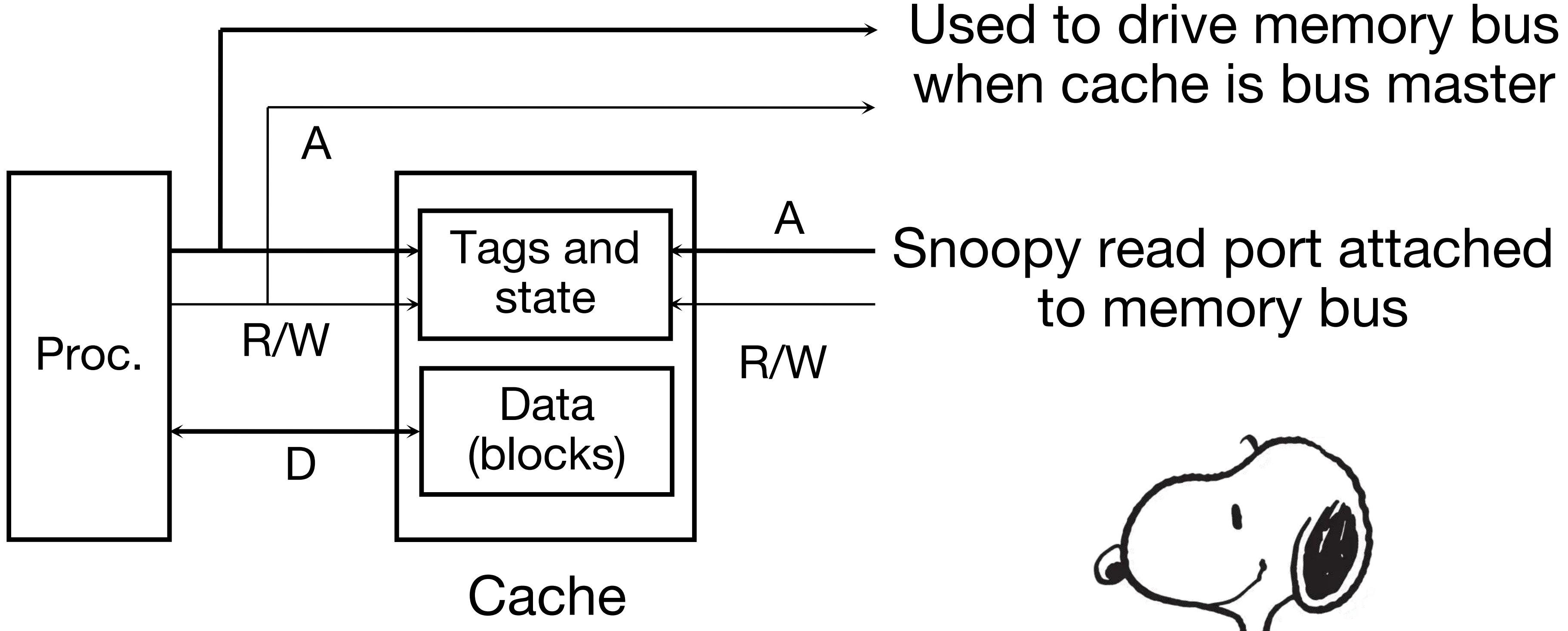
- **Coherent**: any read of a data item returns the most recently written value of that data item
- Because there is shared memory, a computer architect must design the system to keep cache values **coherent**.
- Idea: When any processor has cache miss or writes, use the bus to **notify other processors**.
  - If only reading, many processors can have copies
  - If a processor writes, invalidate any other copies.
- One **cache coherence protocol**: Each cache controller “**snoops**” for write transactions on the common bus
  - Bus is a broadcast medium
  - On any block request to the bus, check if own cache has a copy
    - If exists, then invalidate own cache’s copy

How to Keep Cache Coherent?

- *Cache coherent protocol*, things to think about:
    - How do we communicate when one processor changes the state of shared data?
    - Does every processor action cause data to change state?
    - Who should be responsible for providing the updated data?
    - What happens to memory while all of this is happening?

Snooping/Snoopy Protocols

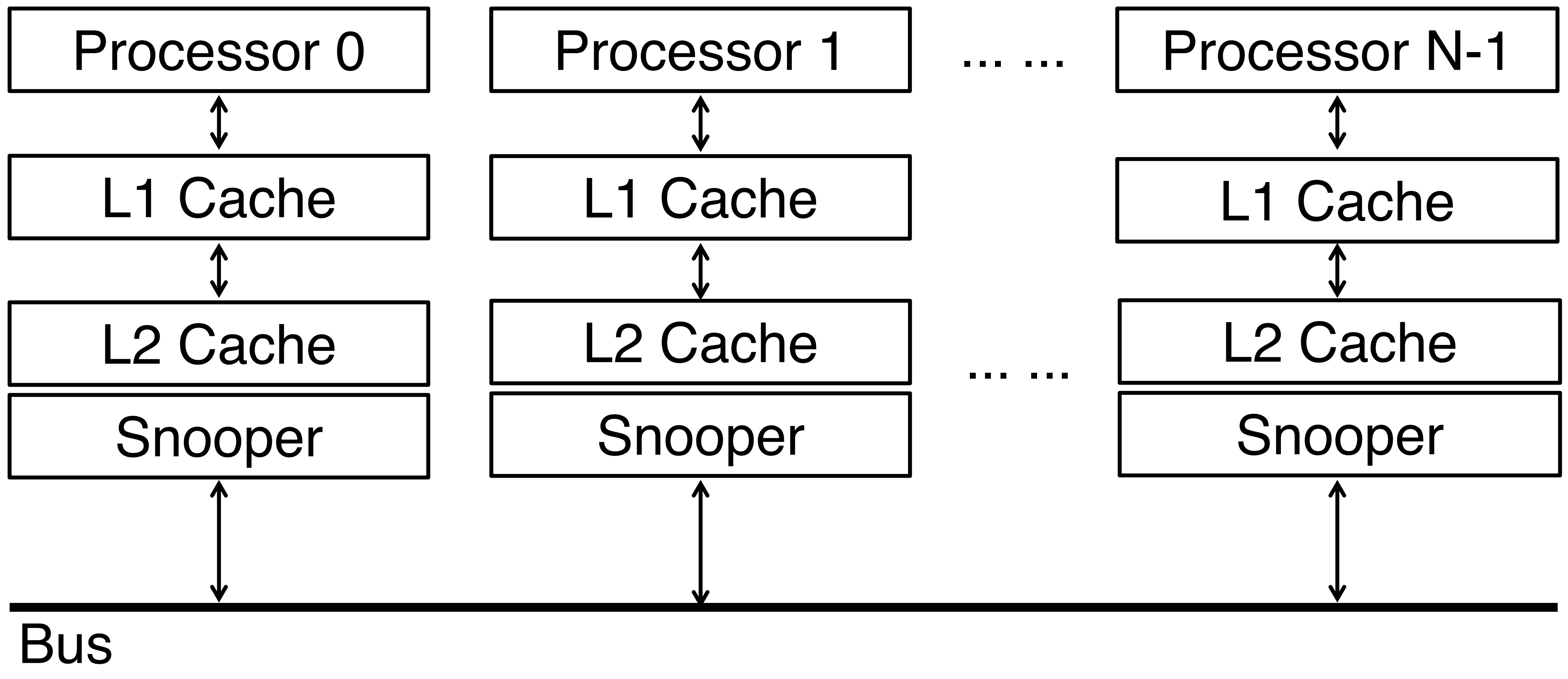
- Snoopy Cache, *[Goodman 1983]*
  - Idea: Have cache watch (or snoop upon) other memory transactions, and then "do the right thing"
  - Snoopy cache tags are dual-ported


Used to drive memory bus when cache is bus master
A
Proc.
R/W
D
Tags and state
Data (blocks)
A
R/W
Snoopy read port attached to memory bus
Cache

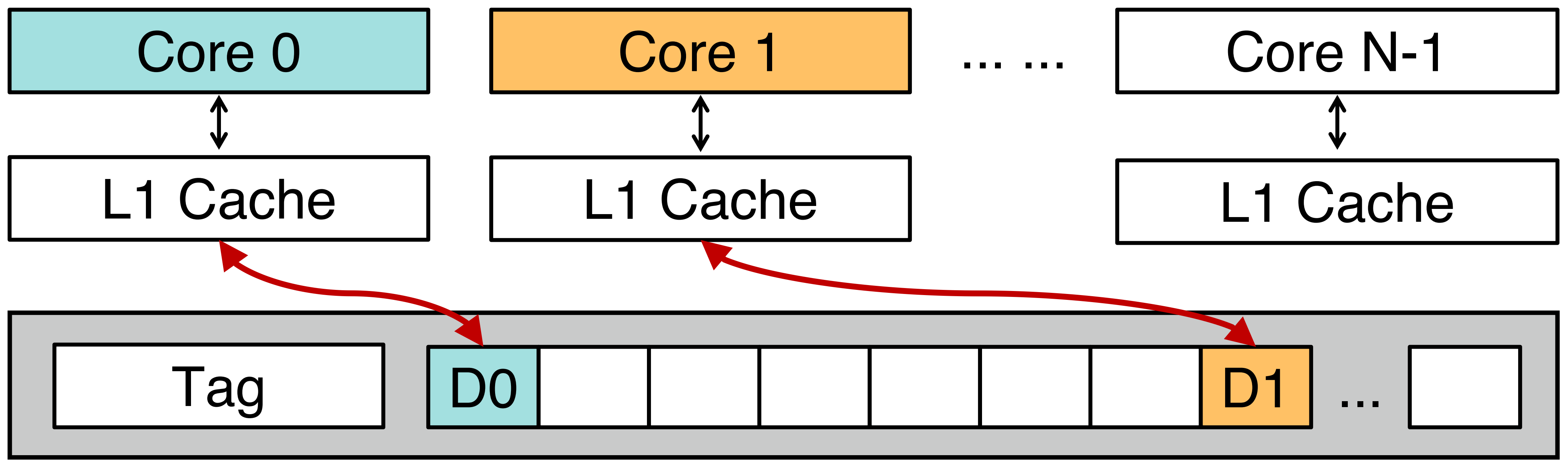
Courtesy: Baidu Baike

Optimized Snoop with L2 Cache

- Processors often have two-level caches
  - Small L1, large L2 (usually both on chip)
- Inclusion property: entries in L1 must be in L2
  - invalidation in L2 => invalidation in L1
- Snooping on L2 does not affect CPU-L1 bandwidth



Cache Coherence Tracked by Block

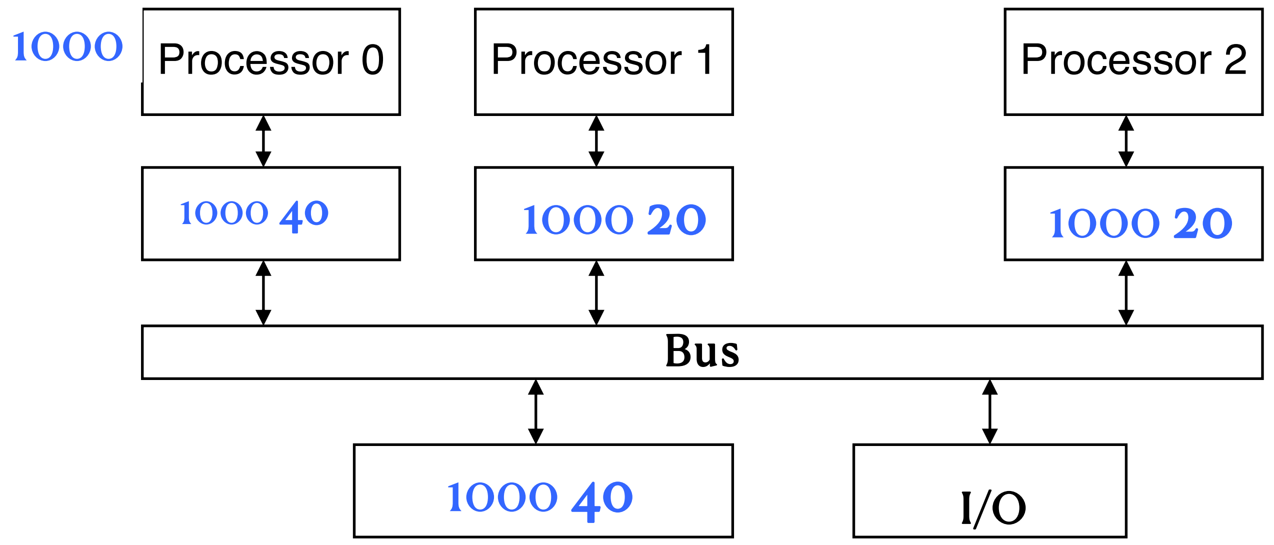


A big cache block in L1 caches of Cores 0 and 1

- Suppose core 0 reads and writes D0, core 1 reads and writes D1
- What will happed?
- ***False sharing*** effect
  - From hardware perspective, use relatively small cache block;
  - Once the hardware is given, keep variables far apart (at least block size away)

Snooping Protocols

- *Write invalidate*
  - Processor *k* wanting to write to an address, grabs a bus cycle and sends a 'write invalidate' message
  - All the other snooping caches invalidate their copy of appropriate cache line
  - Processor *k* writes to its cached copy (assume for now that it also writes through to memory)
  - Any shared read in the other processors will now miss in cache and re-fetch new data.

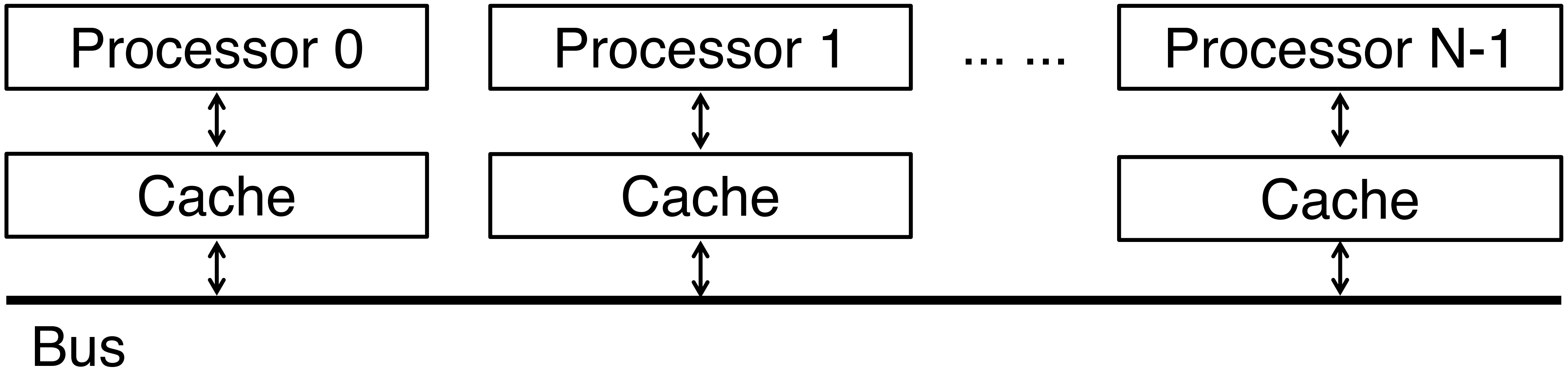


Processor 0 write invalidates other copies

Save the precious bandwidth! Preferred!

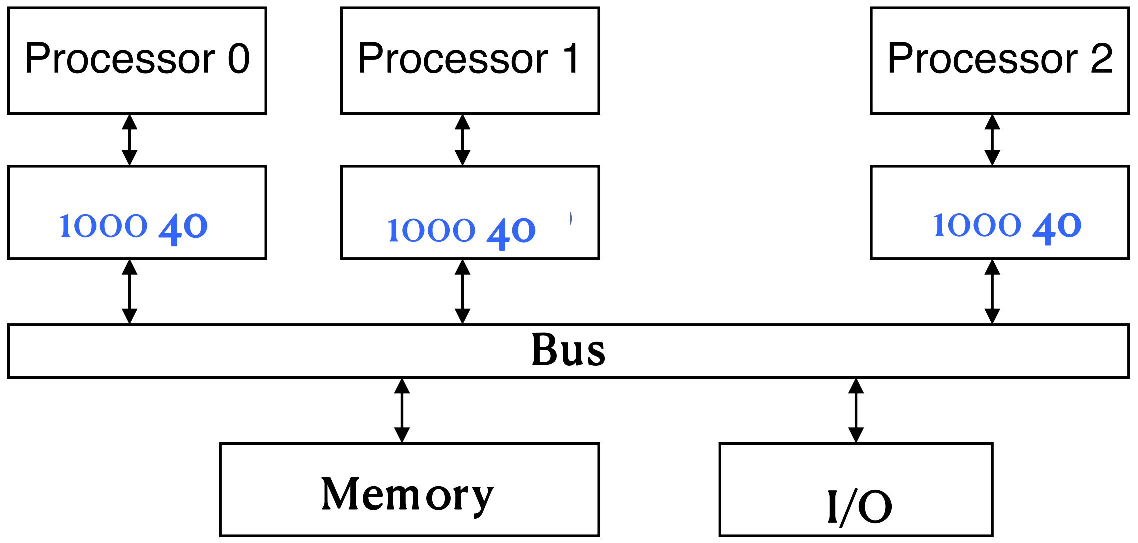
Optimized Snoop with WAW

- Use valid bit to “unload” cache lines (in processors 1~N-1);
- If write-back cache, processor 0 holds a dirty bit;
  - Dirty bit tells me: “I am the only one using this cache line”! => no need to announce on bus again for a second write by processor 0!



Snooping Protocols

- *Write update*
  - CPU wanting to write grabs bus cycle and broadcasts new data as it updates its own copy
  - All snooping caches update their copy



Snooping Protocols

- Write invalidate:
  - Processor *k* wanting to write to an address, grabs a bus cycle and sends a 'write invalidate' message
  - All the other snooping caches invalidate their copy of appropriate cache line
  - Processor *k* writes to its cached copy (assume for now that it also writes through to memory)
  - Any shared read in the other processors will now miss in cache and re-fetch new data.
- Write update:
  - CPU wanting to write grabs bus cycle and broadcasts new data as it updates its own copy
  - All snooping caches update their copy
- In either case, problem of simultaneous writes is taken care of by bus arbitration, i.e., only one processor can use the bus at any one time.

Implementation Issues

- Knowing if a cached value is not shared (copy in another cache) can avoid sending messages
    - But when combined with “write-back” policy, the other processors may re-fetch the old value;
- Requires protocol to handle this;
- The cache coherence protocols ensure that there is a coherent view of data, with migration and replication.
    - A cache line has a state

Example: MOESI Protocols

- For each block in a cache, track its state:
  - **Shared**: up-to-date data, other caches may have a copy; can evict the data without writing it to backing store;
  - **Modified**: up-to-date data, changed (dirty), no other cache has a copy, OK to write, memory out-of-date (i.e., write-back); can be further modified freely;
  - **Invalid**: not in cache (from before: valid flag), must be fetched.
  - and optional performance optimizations:
  - **Exclusive**: up-to-date data, no other cache has a copy, OK to write, memory up-to-date;
  - **Owner**: up-to-date data, other caches may have a copy (they must be in Shared state), the only copy that can update the memory;
  - There are different combinations of them (and the other newly invented states)
    - MSI/MESI/MOESI (AMD processor family)/MESI+F (Intel processor)

More in CAII

True or False?

- Using write-through caches removes the need for cache coherence.
- Every processor store instruction must check contents of other caches.
- Only one processor can cache any memory location at one time.

True or False?

- Using write-through caches removes the need for cache coherence.

FALSE. You have a copy. I do a write (through, to memory). How do you get updated when you do a read?

- Every processor store instruction must check contents of other caches.

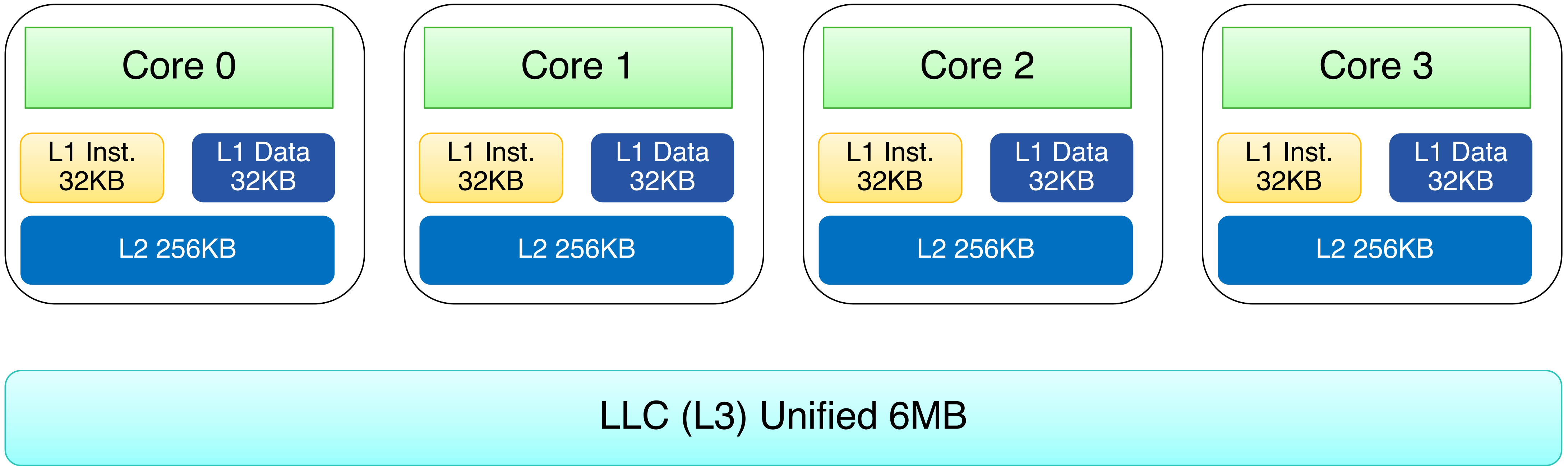
FALSE. That's the point of these protocols, to know if others have copies and whether I need to just do a store or do other work.

- Only one processor can cache any memory location at one time.

FALSE. What if they're all doing reads? That would be inefficient.

Advanced Cache

- Inclusiveness of multi-level caches

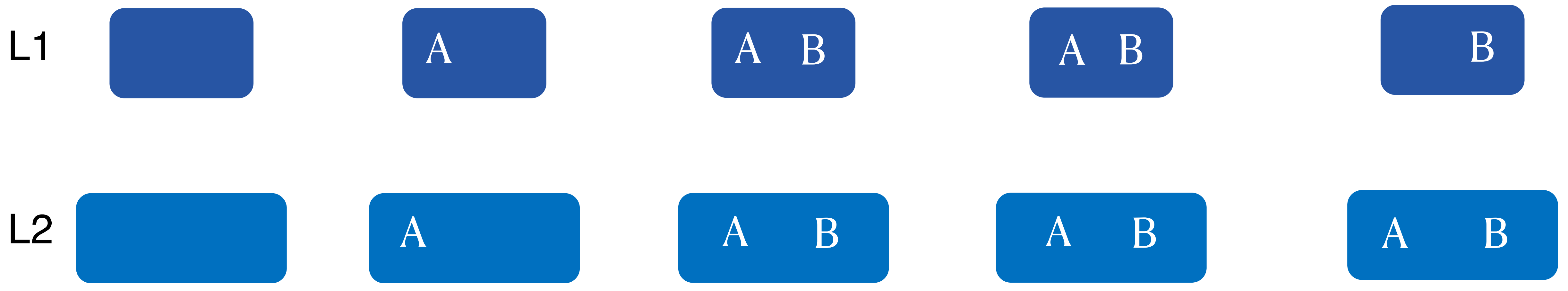


Intel Ivy Bridge Cache Architecture (Core i5-3470)

If all blocks in the higher level cache are also present in the lower level cache, then the lower level cache is said to be **inclusive** of the higher level cache.

Inclusiveness

$$L_n \subsetneqq L_{n+1} \ (n \geq 1)$$



Initial state

Read A miss; load A into L1 and L2

Read B miss; load B into L1 and L2

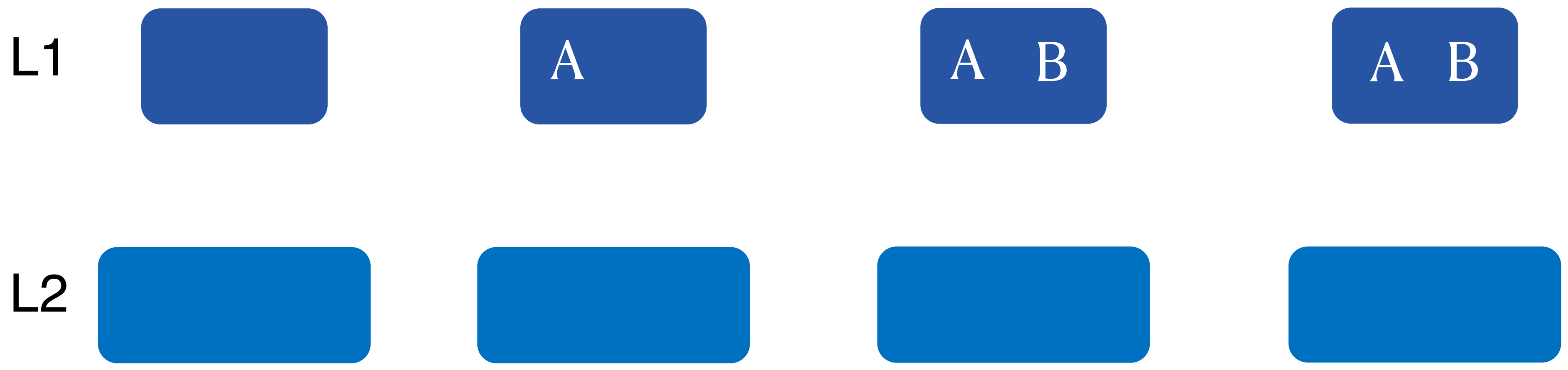
Evict A from L1 due to cache replacement

Evict B from L2 due to cache replacement

Back invalidation

Exclusive

$$L_n \cap L_{n+1} = \emptyset \ (n \geq 1)$$



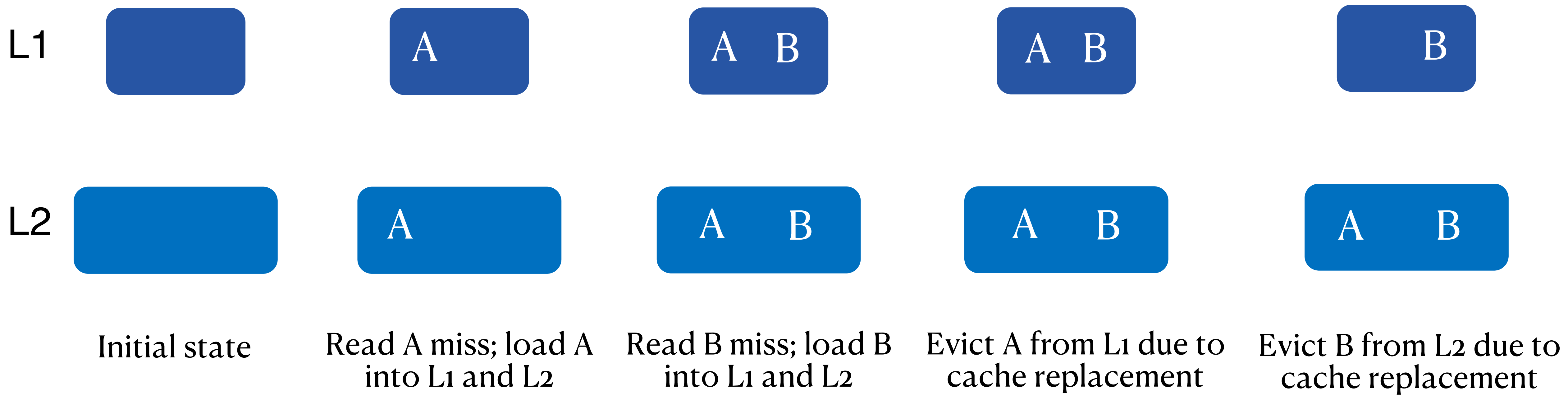
Initial state

Read A miss; load A into L1

Read B miss; load B into L1

Evict A from L1 due to cache replacement and place in L2

Non-inclusive


L1
L2
A
A B
A B
B
A
A B
A B
A B
Initial state
Read A miss; load A into L1 and L2
Read B miss; load B into L1 and L2
Evict A from L1 due to cache replacement
Evict B from L2 due to cache replacement

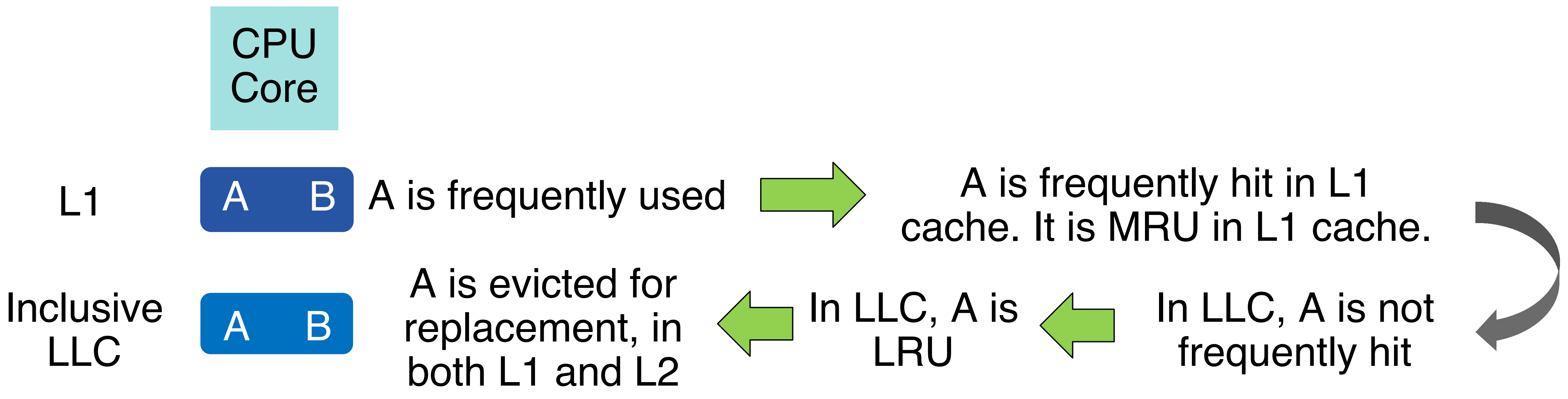
Real Staff

- Intel processors
  - Sandy bridge, inclusive
  - Haswell, inclusive
  - Skylake-S, inclusive
  - Skylake-X, non-inclusive
- ARM processors
  - ARMv7, non-inclusive
  - ARMv8, non-inclusive
- AMD processors
  - K6, exclusive
  - Zen, inclusive
  - Shanghai, LLC non-inclusive

Inclusive or Not?

- Inclusive cache eases coherence
  - A cache block in a higher-level surely exists in lower-level(s)
- Non-inclusive cache yields higher performance though, why?
  - No back invalidation
  - More data can be cached ← larger capacity

“Sneaky” LRU for Inclusive Cache

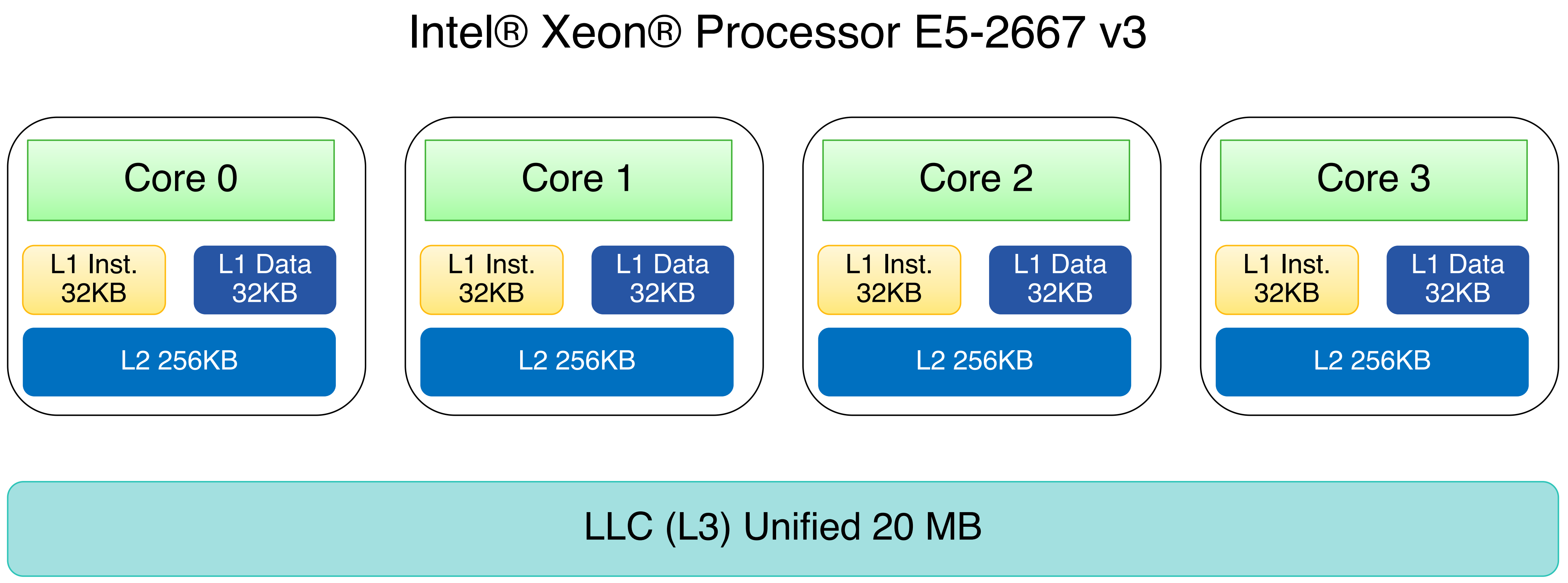

CPU Core
L1
A B
A is frequently used
A is frequently hit in L1 cache. It is MRU in L1 cache.
Inclusive LLC
A B
In LLC, A is not frequently hit
In LLC, A is LRU
A is evicted for replacement, in both L1 and L2

As a result, MRU block that should be retained might be evicted, which causes performance penalty.

What if LLC is non-inclusive?

Should you be interested, you can click https://doi.org/10.1109/MICRO.2010.52 to read the related research paper for details.

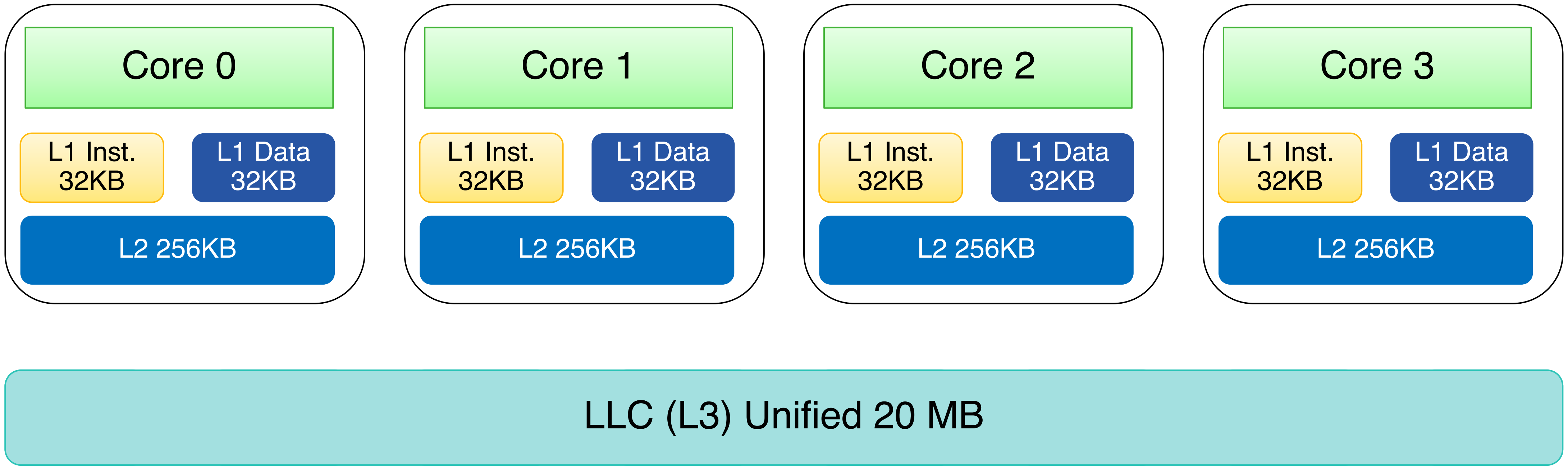
Last-Level Cache (LLC) is not Monolithic

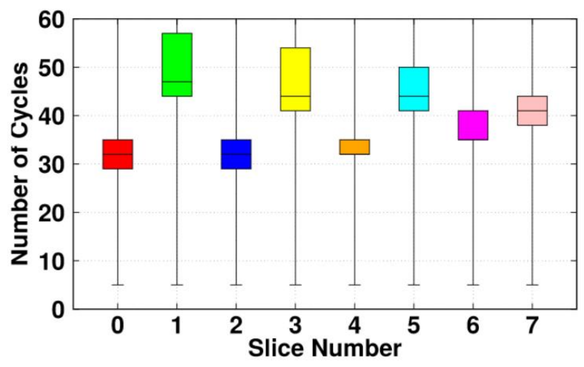


- Previously, it's considered that, to CPU cores, LLC is monolithic. No matter where a cache block in the LLC, a core would load it into private L2 and L1 cache with the same time cost.

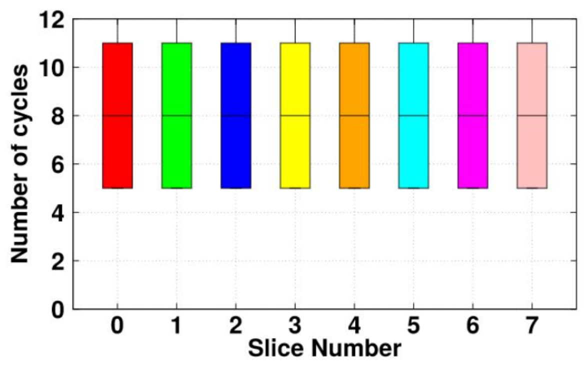
Last-Level Cache (LLC) is not Monolithic

Intel® Xeon® Processor E5-2667 v3





(a) Read.



(b) Write.

LLC is fine-grained
LLC in 8 slices

From the paper https://doi.org/10.1145/3302424.3303977

Slice-aware Memory Management

- The idea seems simple
  - Put your data closer to your program (core)
- But it not *EASY* to do so
  - Cache management is undocumented, not to mention fine-grained slices
  - Researchers did a lot of efforts
    - Click https://doi.org/10.1145/3302424.3303977 for details
    - They managed to improve the average performance by 12.2% for GET operations of a key-value store.
    - 12.2% is a lot, if you consider the huge transactions every day for Google, Taobao, Tencent, JD, etc.

Summary

- There is a huge design space for CPU cache
- To make the best of cache can boost your program’s performance!