CS 110
Computer Architecture

Advanced Caches

Instructor:

Sören Schwertfeger and Chundong Wang

https://robotics.shanghaitech.edu.cn/courses/ca/22s

School of Information Science and Technology SIST

ShanghaiTech University

Slides based on UC Berkeley's CS61C (2015)

Review

- WSC
  - A giant computer
- Map/Reduce
  - Map
  - Reduce

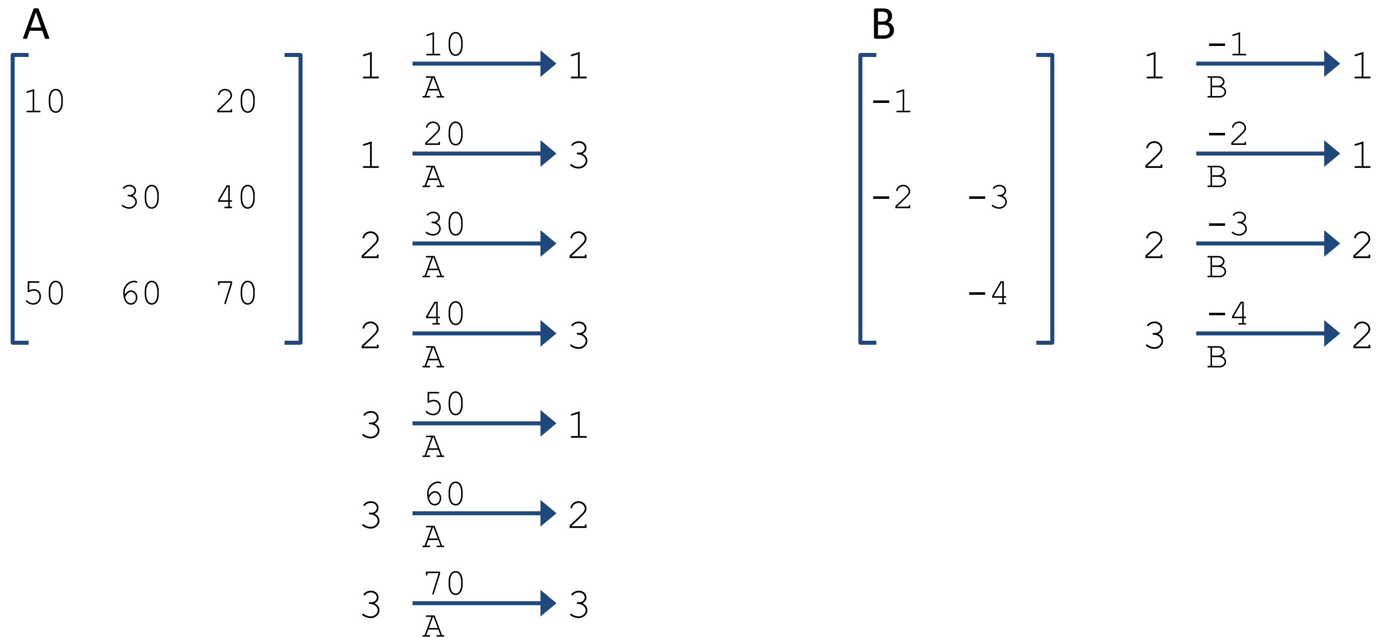
Review: Map/Reduce for Sparse Matrices

$$A = \begin{bmatrix} 10 & & 20 \\ & 30 & 40 \\ 50 & 60 & 70 \end{bmatrix} \times B = \begin{bmatrix} -1 & \\ -2 & -3 \\ & -4 \end{bmatrix} = C = \begin{bmatrix} -10 & -80 \\ -60 & -250 \\ -170 & -460 \end{bmatrix}$$

- Task: Compute product C = A·B
- Assume most matrix entries are 0

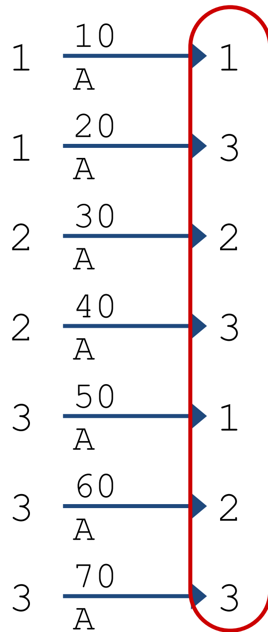
- Motivation
  - Core problem in scientific computing
  - Challenging for parallel execution
  - Demonstrate expressiveness of Map/Reduce

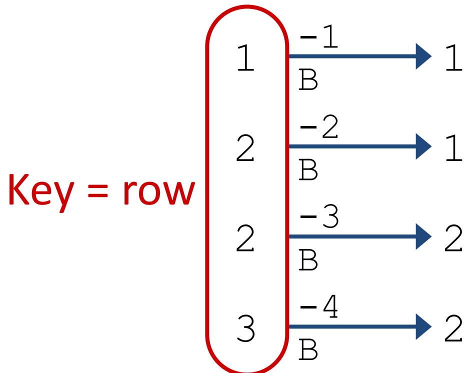
Computing Sparse Matrix Product

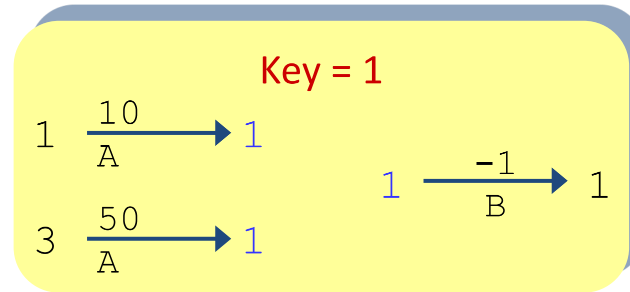


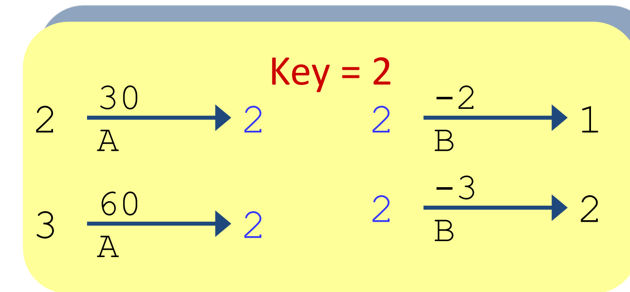
- Represent matrix as list of nonzero entries
  ⟨row, col, value, matrixID⟩
- Strategy
  - **Phase 1: Compute all products $a_{i,k} \cdot b_{k,j}$**
  - **Phase 2: Sum products for each entry i,j**
  - **Each phase involves a Map/Reduce**

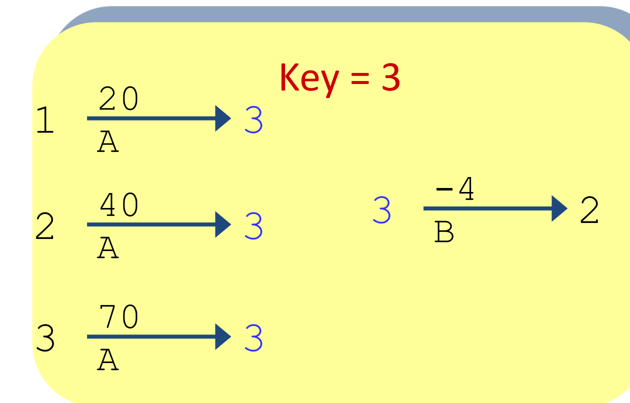
Phase 1 Map of Matrix Multiply











– Group values $a_{i,k}$ and $b_{k,j}$ according to key k

Phase 1 "Reduce" of Matrix Multiply

Key = 1

1 —10/A→ 1
3 —50/A→ 1

X

1 —-1/B→ 1

Key = 2

2 —30/A→ 2
3 —60/A→ 2

X

2 —-2/B→ 1
2 —-3/B→ 2

Key = 3

1 —20/A→ 3
2 —40/A→ 3
3 —70/A→ 3

X

3 —-4/B→ 2

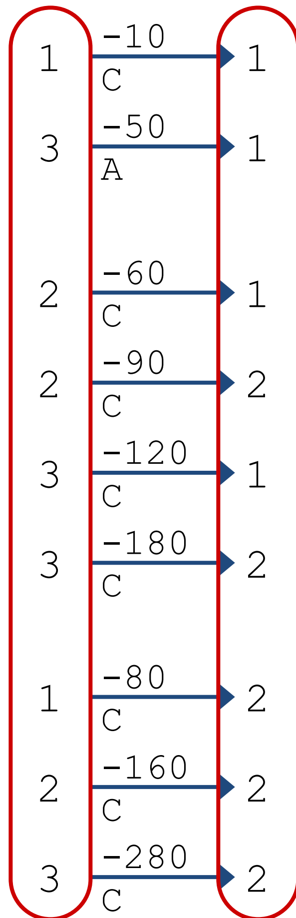
1 —-10/C→ 1
3 —-50/C→ 1

2 —-60/C→ 1
2 —-90/C→ 2
3 —-120/C→ 1
3 —-180/C→ 2

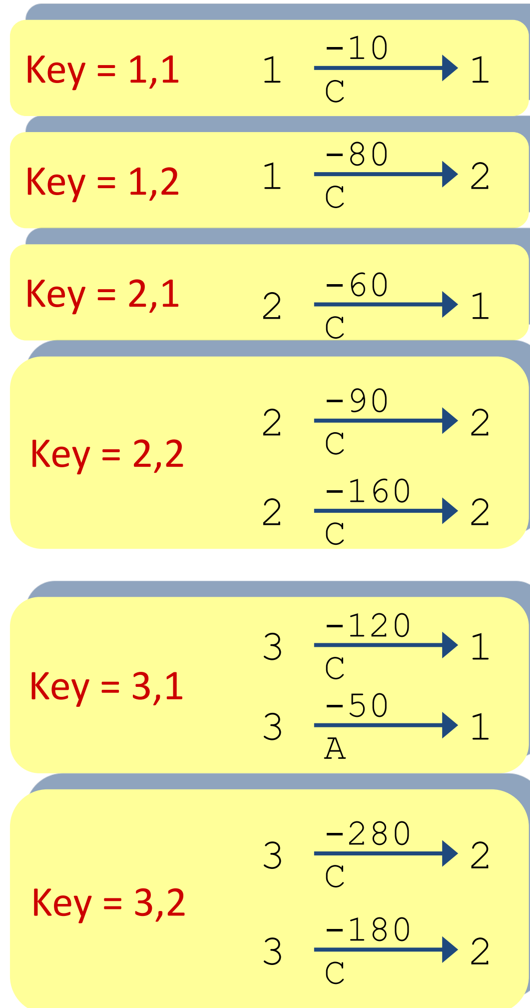
1 —-80/C→ 2
2 —-160/C→ 2
3 —-280/C→ 2

– Generate all products $a_{i,k} \cdot b_{k,j}$

Phase 2 Map of Matrix Multiply



Key = row,col



– Group products $a_{i,k} \cdot b_{k,j}$ with matching values of i and j

Phase 2 Reduce of Matrix Multiply

Key	Entries	Reduced
Key = 1,1	1 —(-10, C)→ 1	1 —(-10, C)→ 1
Key = 1,2	1 —(-80, C)→ 2	1 —(-80, C)→ 2
Key = 2,1	2 —(-60, C)→ 1	2 —(-60, C)→ 1
Key = 2,2	2 —(-90, C)→ 2 2 —(-160, C)→ 2	2 —(-250, C)→ 2
Key = 3,1	3 —(-120, C)→ 1 3 —(-50, A)→ 1	3 —(-170, C)→ 1
Key = 3,2	3 —(-280, C)→ 2 3 —(-180, C)→ 2	3 —(-460, C)→ 2

C

$$\begin{bmatrix} -10 & -80 \\ -60 & -250 \\ -170 & -460 \end{bmatrix}$$

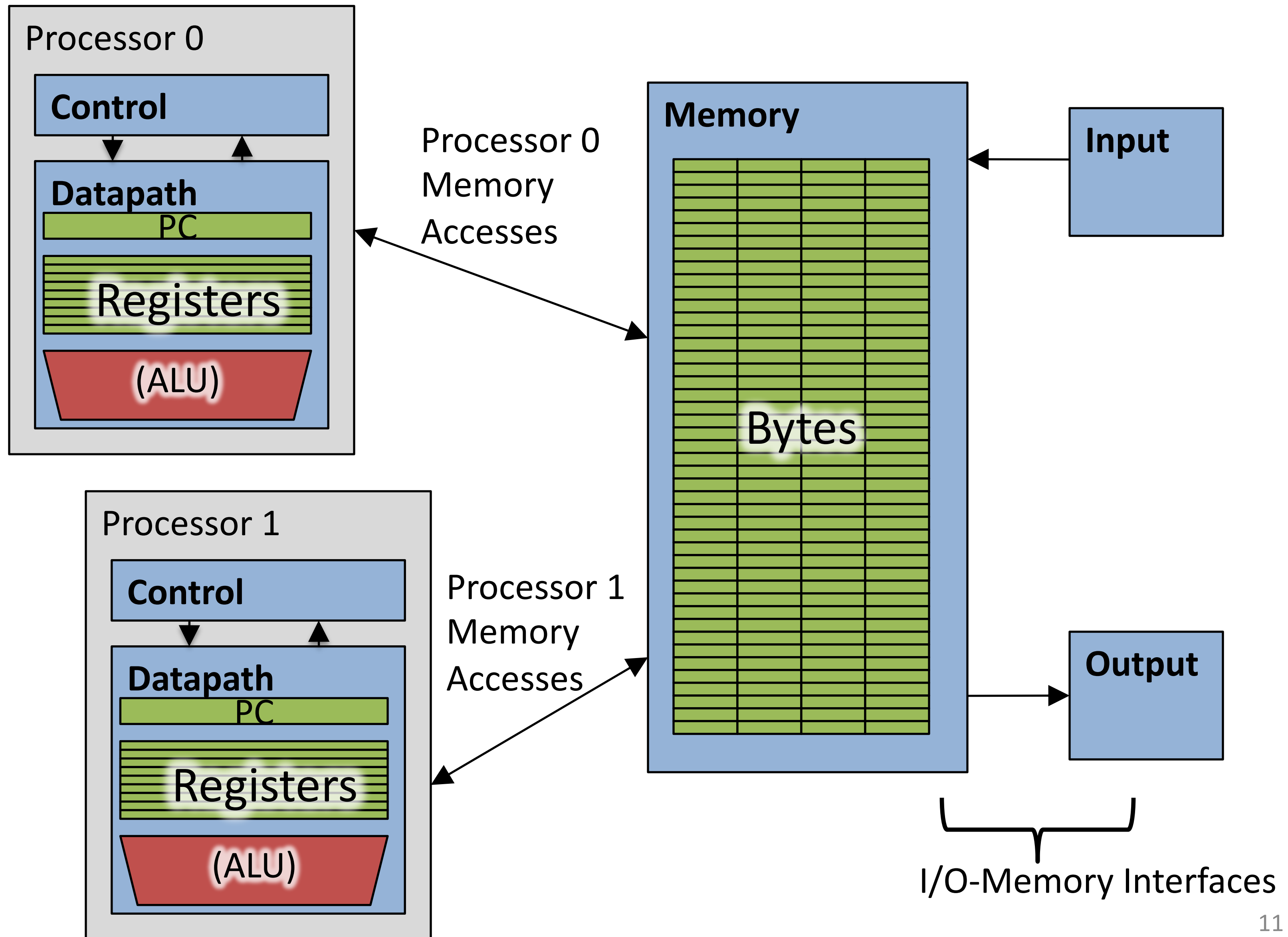
– Sum products to get final entries

Lessons from Sparse Matrix Example

- Associative matching is powerful communication primitive
  - Intermediate step in Map/Reduce
- Similar Strategy Applies to Other Problems
  - Shortest path in graph
  - Database join
- Many Performance Considerations
  - *Pairwise Element Computation with MapReduce* (HPDC '10, by Kiefer, Volk, Lehner from TU Dresden)
  - Should do systematic comparison to other sparse matrix implementations

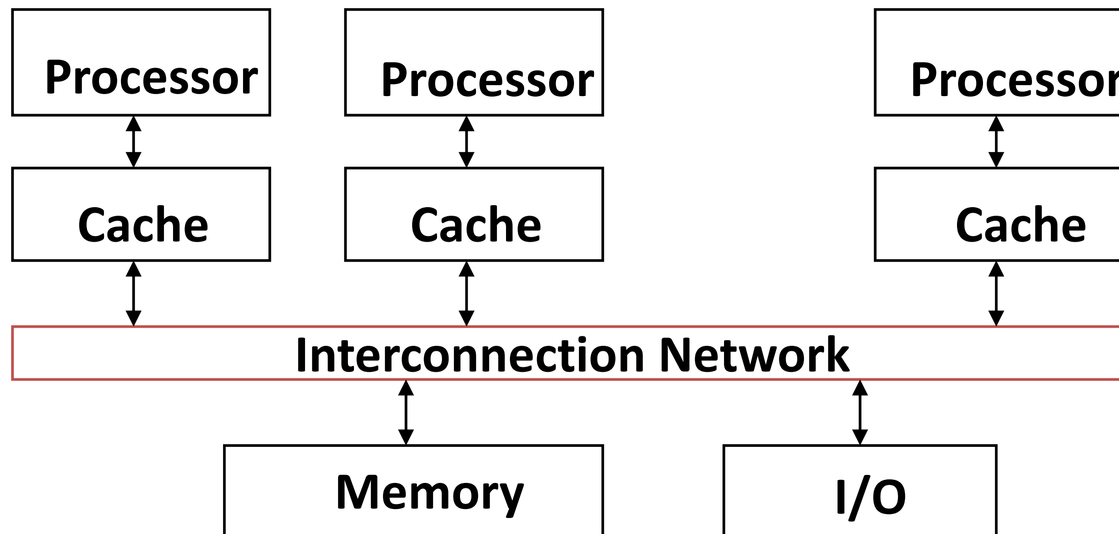
CACHE COHERENCE

Simple Multi-core Processor



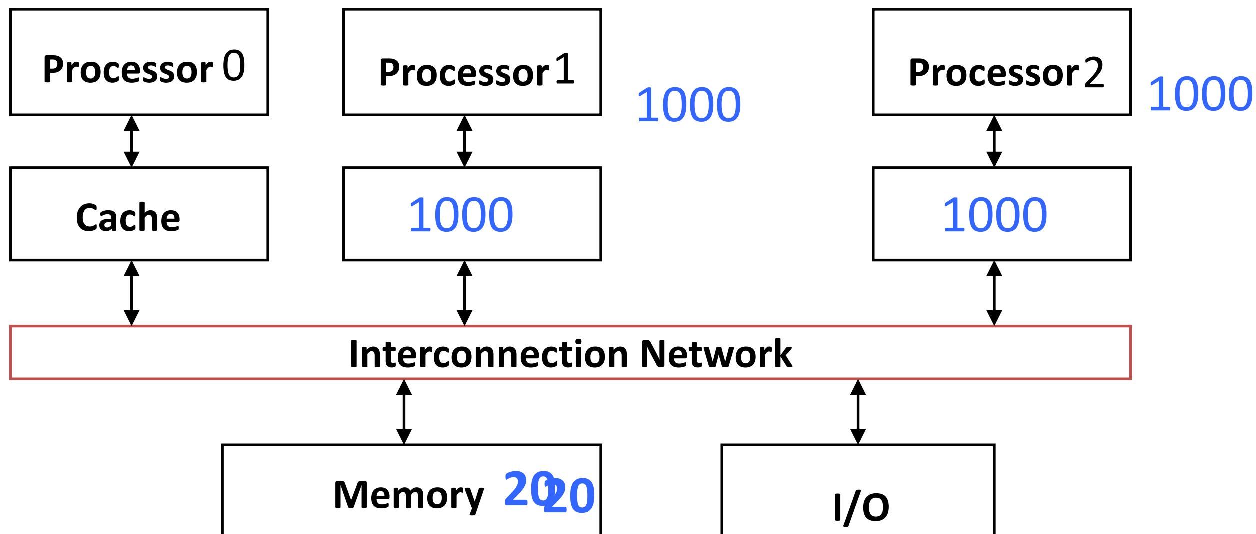
Multiprocessor Caches

- Memory is a performance bottleneck even with one processor
- Use caches to reduce bandwidth demands on main memory
- Each core has a local private cache holding data it has accessed recently
- Only cache misses have to access the **shared** common memory



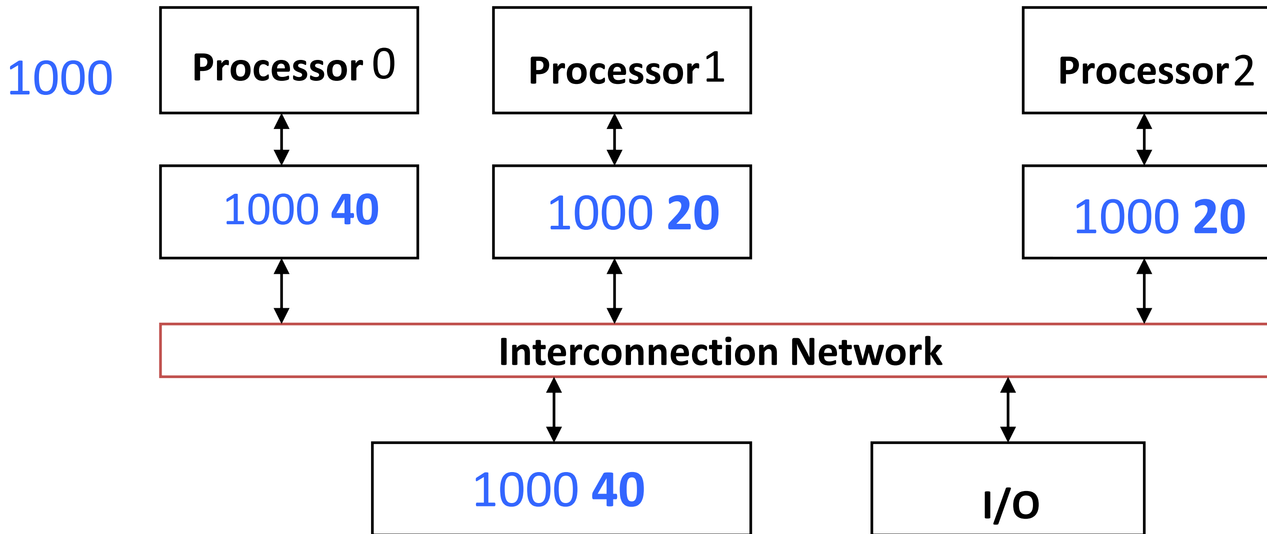
Shared Memory and Caches

- What if?
  - Processors 1 and 2 **read** Memory[1000] (value 20)



Shared Memory and Caches

- Now:
  - Processor 0 **writes** Memory[1000] with 40



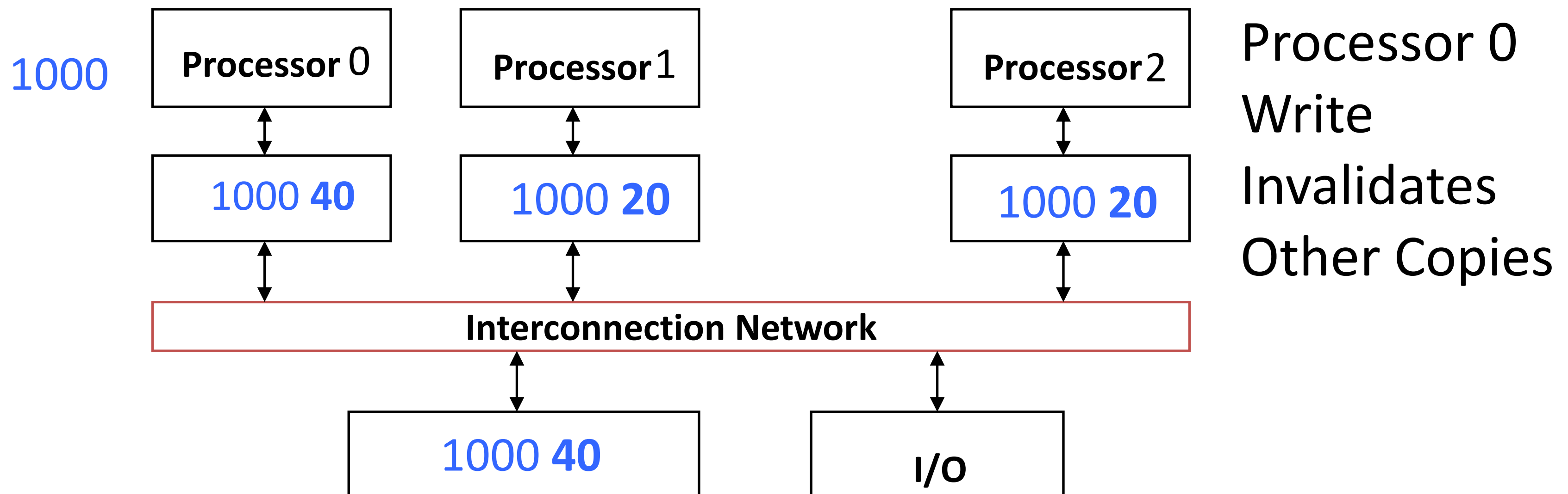
Problem?

Keeping Multiple Caches Coherent

- Architect's job: shared memory
  => keep cache values coherent
- Idea: When any processor has cache miss or writes, notify other processors via interconnection network
  - If only reading, many processors can have copies
  - If a processor writes, invalidate any other copies
- Write transactions from one processor, other caches "snoop" the common interconnect checking for tags they hold
  - Invalidate any copies of same address modified in other cache

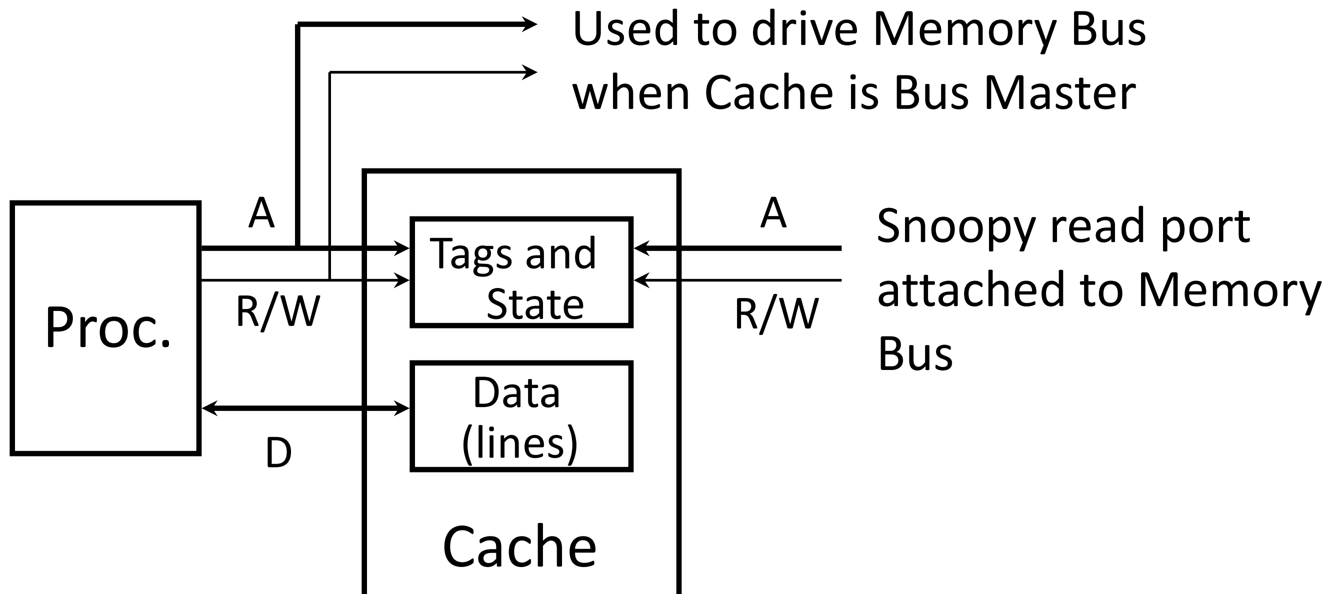
Shared Memory and Caches

- Example, now with cache coherence
  - Processors 1 and 2 read Memory[1000]
  - Processor 0 writes Memory[1000] with 40

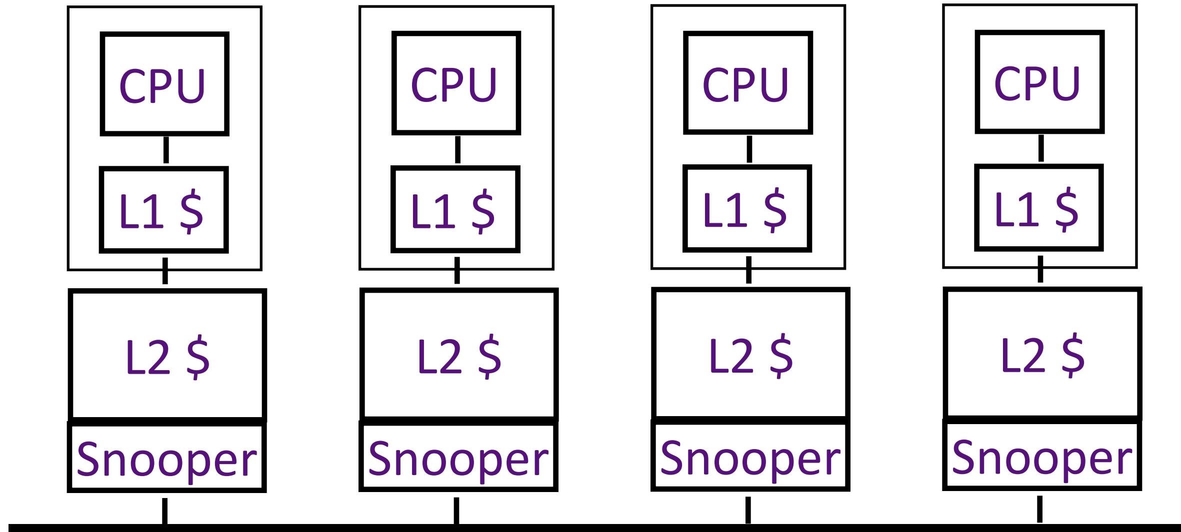


Snoopy Cache, *Goodman 1983*

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

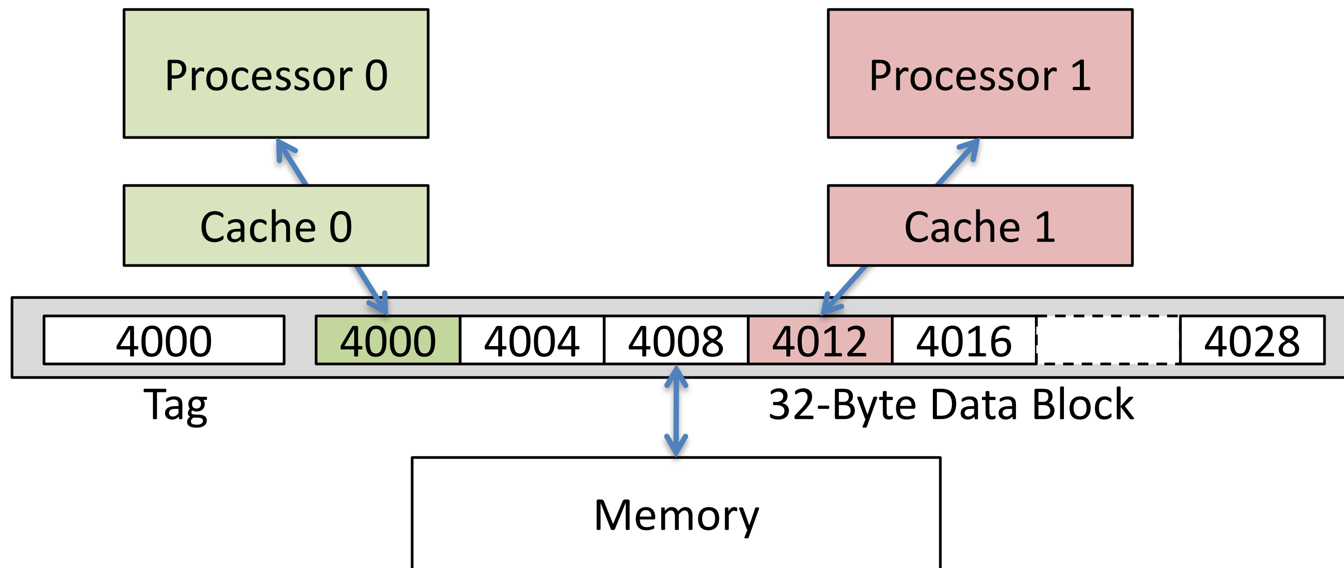


Optimized Snoop with Level-2 Caches



- Processors often have two-level caches
  - small L1, large L2 (usually both on chip now)
- 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 Coherency Tracked by Block



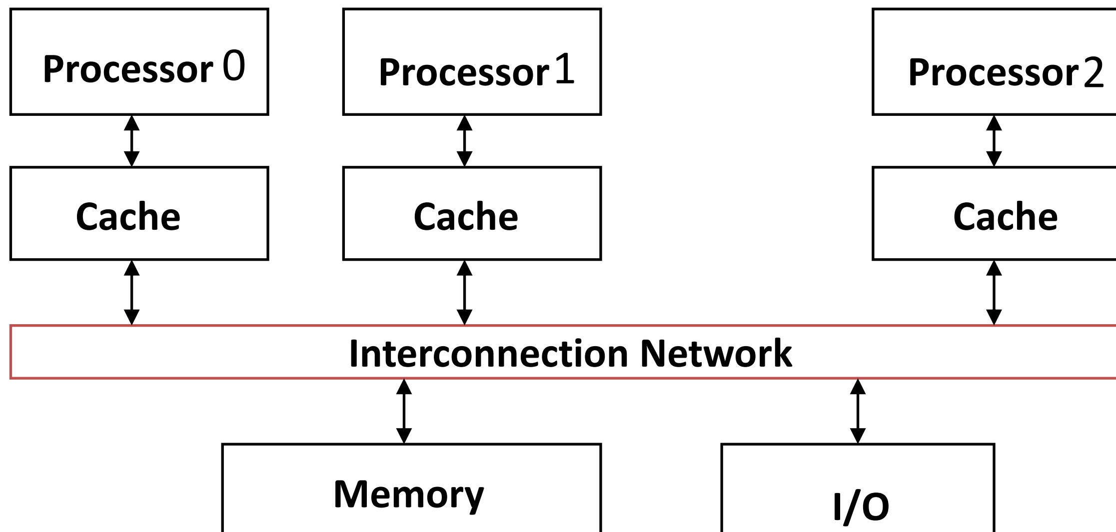
- Suppose block size is 32 bytes
- Suppose Processor 0 reading and writing variable X, Processor 1 reading and writing variable Y
- Suppose in X location 4000, Y in 4012
- What will happen?

Coherency Tracked by Cache Block

- Block ping-pongs between two caches even though processors are accessing disjoint variables
- Effect called *false sharing*
- How can you prevent it?
  - Keep variables far apart (at least block size (64 byte))

Shared Memory and Caches

- Use valid bit to "unload" cache lines (in Processors 1 and 2)
- Dirty bit tells me: "I am the only one using this cache line"! => no need to announce on Network!



Review: Understanding Cache Misses: The 3Cs

- Compulsory (cold start or process migration, 1st reference):
  - First access to block, impossible to avoid; small effect for long-running programs
  - Solution: increase block size (increases miss penalty; very large blocks could increase miss rate)
- Capacity (not compulsory and…)
  - Cache cannot contain all blocks accessed by the program ***even with perfect replacement policy in fully associative cache***
  - Solution: increase cache size (may increase access time)
- Conflict (not compulsory or capacity and…):
  - Multiple memory locations map to the same cache location
  - Solution 1: increase cache size
  - Solution 2: increase associativity (may increase access time)
  - Solution 3: improve replacement policy, e.g.. LRU

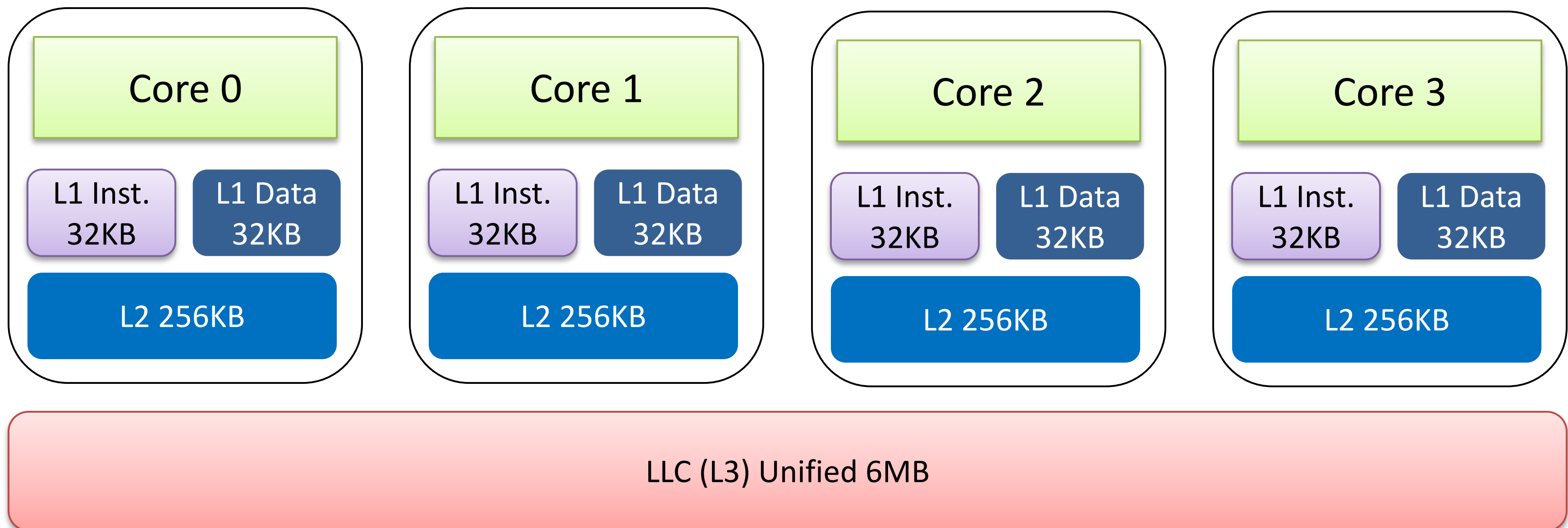
Fourth "C" of Cache Misses: *Coherence* Misses

- Misses caused by coherence traffic with other processor
- Also known as *communication* misses because represents data moving between processors working together on a parallel program
- For some parallel programs, coherence misses can dominate total misses

Advanced Caches:
MRU is LRU

Cache Inclusion

- Multilevel caches

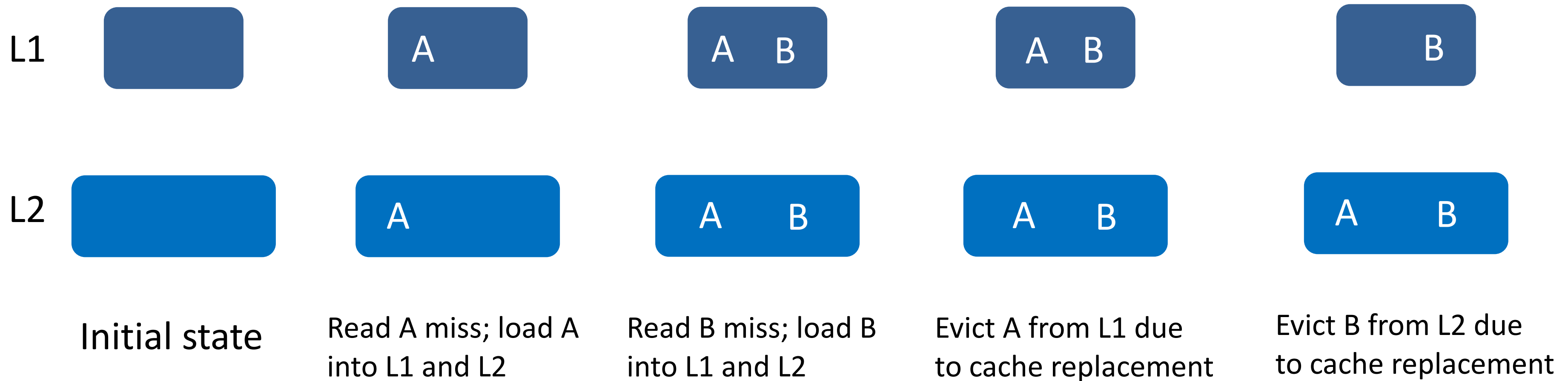


Intel Ivy Bridge Cache Architecture (Core i5-3470)

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.

Inclusive

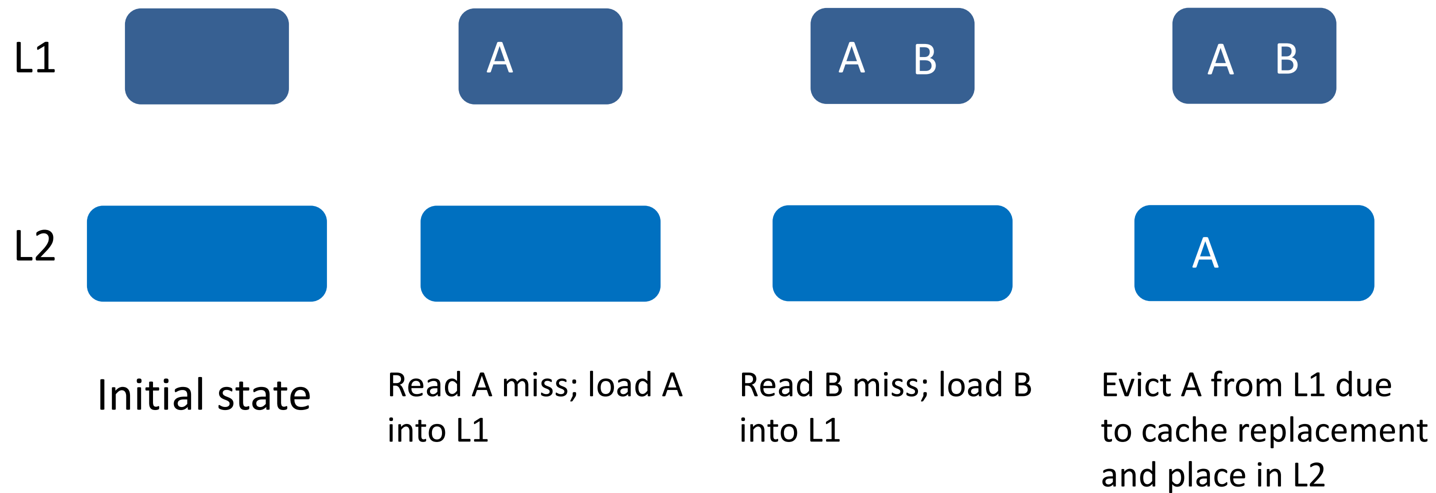
$$L_n \subsetneqq L_{n+1} \; (n \geq 1)$$



Back invalidation

Exclusive

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



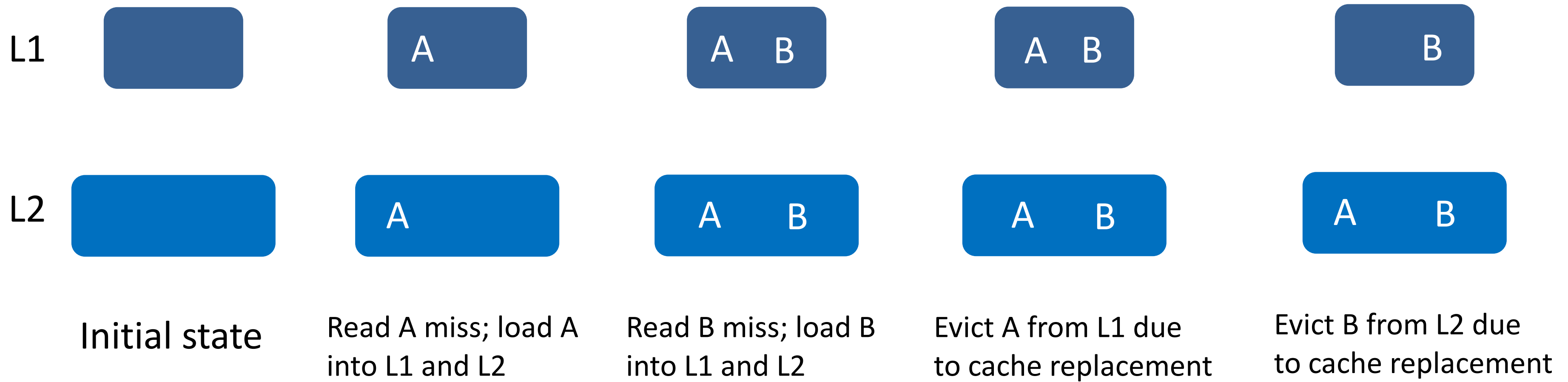
Initial state

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



Real-world CPUs

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

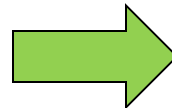
Inclusive, or not?

- Inclusive cache eases coherence
  - A cache block in a higher-level surely existing in lower-level(s)
  - A non-inclusive LLC, say L2 cache, which needs to evict a block, **must** ask L1 cache if it has the block, because such information is not present in LLC.
- Non-inclusive cache yields higher performance though, why?
  - No back invalidation
  - More data can be cached ← larger capacity

'Sneaky' LRU for Inclusive Cache

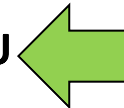
CPU Core

A is frequently used

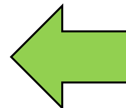


A is frequently hit in L1 cache. It is **MRU** in L1 cache.

In LLC, A is not frequently hit



In LLC, A is **LRU**



A is evicted for replacement, in both L1 and L2

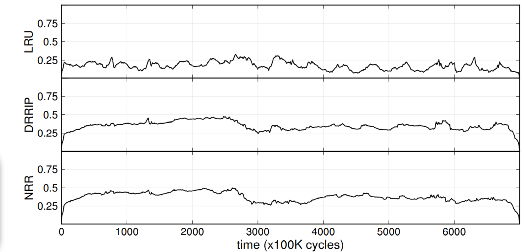
As a result, MRU block that should be retained might be evicted, which causes performance penalty.

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.

Advanced Caches: Reduce the size of LLC

Reduce LLC for high performance



(a) Changes in the fraction of live lines over time

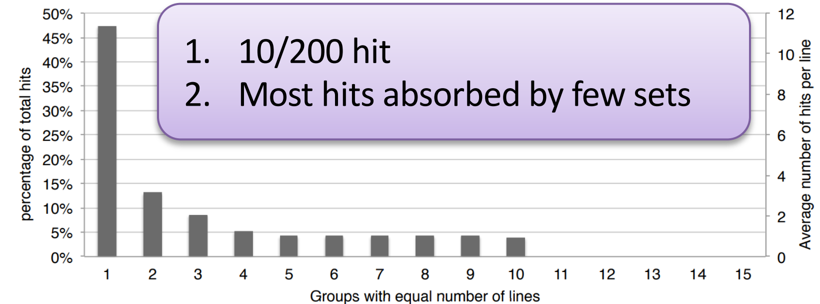
More than 83.8% LLC lines not productive

- Problem
  - A considerable portion of the shared LLC is dead
- Why?
  - LLC accesses, caused by L1 and L2 misses
  - Locality not accurate due to filtering by L1 and L2
  - LLC uniformly handles any access request for line allocation/deallocation
- How to resolve?
  - Leverage the reuse locality to selectively allocate LLC lines

Jorge Albericio, Pablo Ibáñez, Víctor Viñals, and José M. Llabería. 2013. The reuse cache: downsizing the shared last-level cache. In Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46). Association for Computing Machinery, New York, NY, USA, 310–321.

Selective allocation upon reuse locality

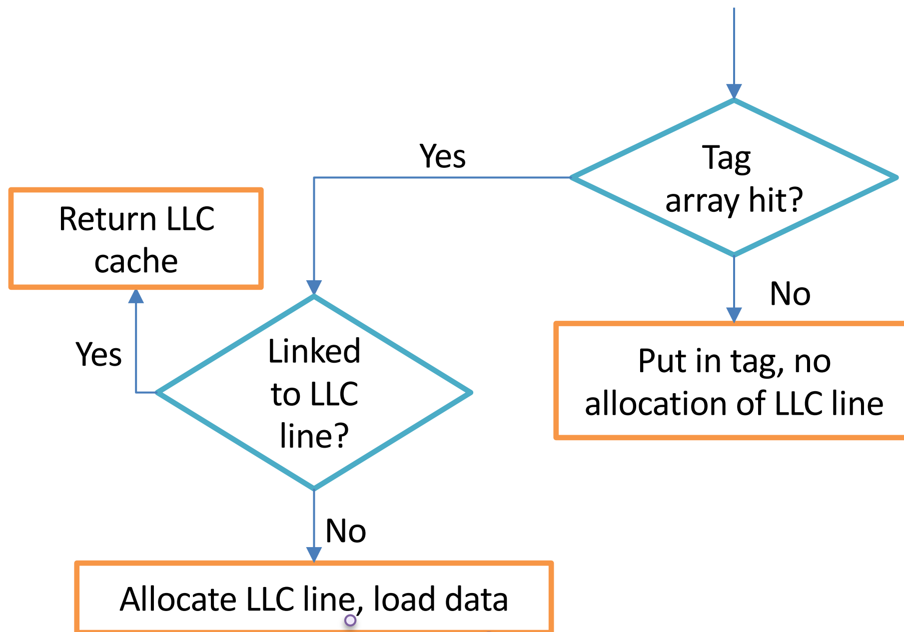
- Reuse locality

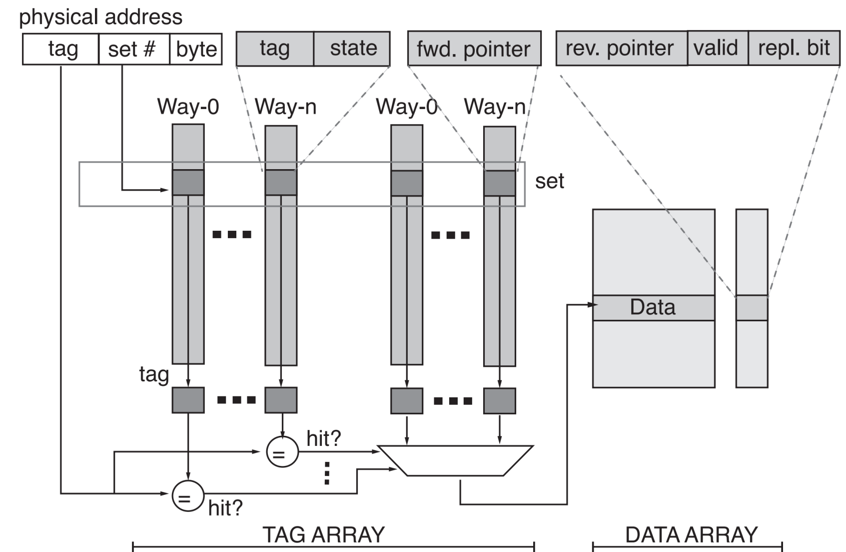


(b) Distribution of hits among all lines loaded (or reloaded) into the LRU SLLC during their stay. Each group represents 0.5% of the loaded lines

- Selective allocation
  - Tag and cache line decoupled
    - Conventionally, one tag for one cache line
    - Now, more tags than cache lines
      - Some place holders
  - Only keeping reused cache line

Allocation policy

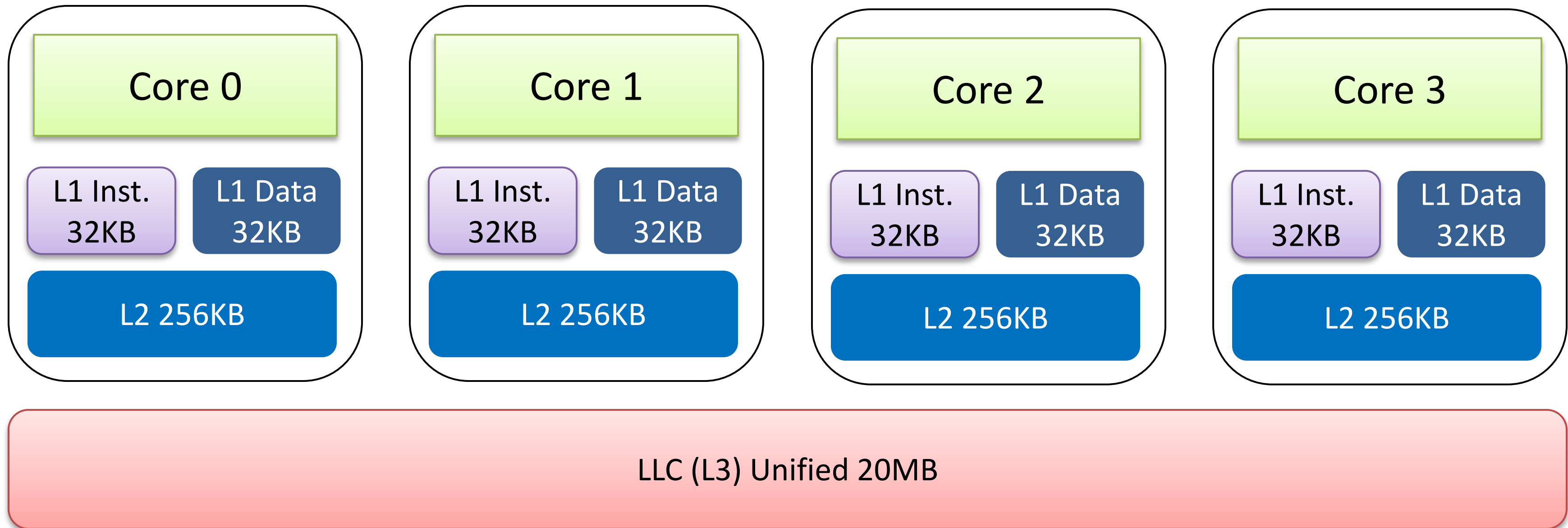




On replacement, the tag of evicted LLC line is kept. Once hit again, i.e., reused, data reloaded again.

Advanced Caches:
LLC is not monolithic

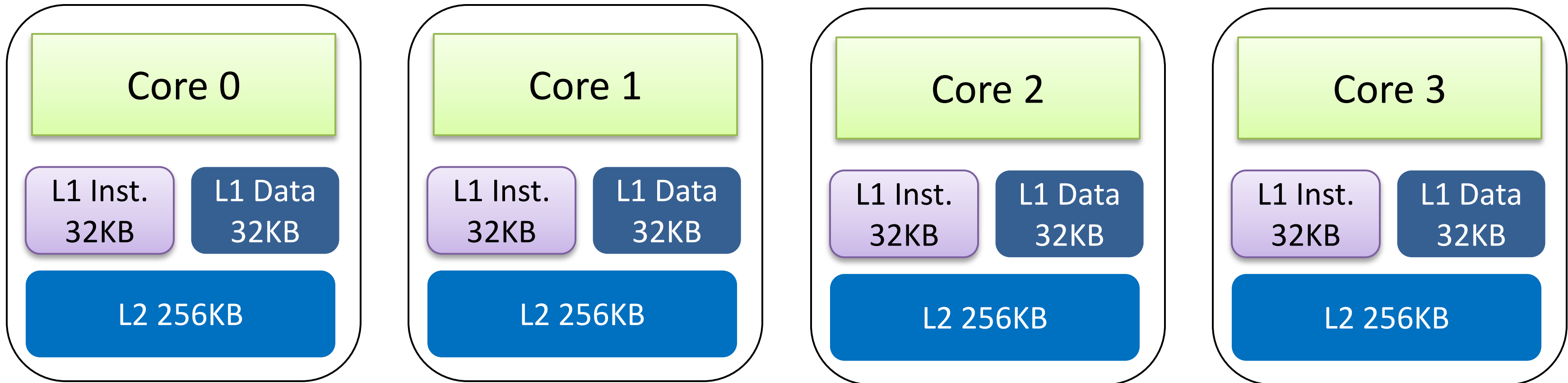
LLC is not monolithic

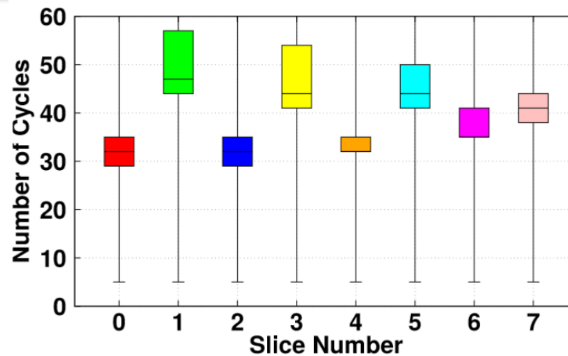


Previously, it's considered that, to CPU cores, LLC is monolithic. No matter where a cache block in the LLC, a core would load it into private L2 and L1 cache with **the same** time cost.

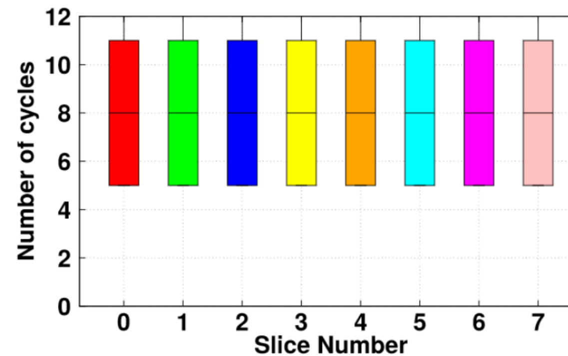
LLC is fine-grained

Intel® Xeon® Processor E5-2667 v3





(a) Read.



(b) Write.

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.

Conclusion

- Map/Reduce can be useful for you
  - e.g., matrix multiplication
- There are many interesting facts of CPU cache
- To make the best of cache can boost your program's performance!