Computer Architecture

Lecture 32: Cache Design and Management

Prof. Onur Mutlu
ETH Zürich

Fall 2023

14 February 2024

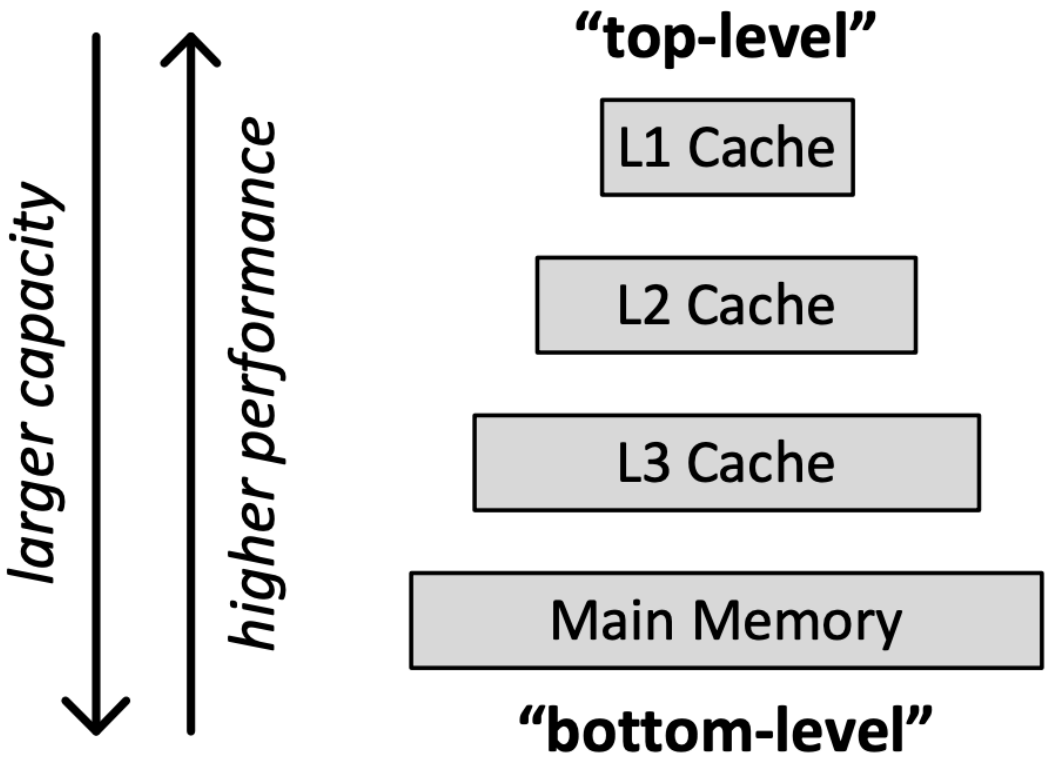
Readings for This Week and Last Week

- Memory Hierarchy and Caches

- Required
  - H&H Chapters 8.1-8.3
  - Refresh: P&P Chapter 3.5
  - Kim & Mutlu, "Memory Systems," Computing Handbook, 2014.
    - https://people.inf.ethz.ch/omutlu/pub/memory-systems-introduction_computing-handbook14.pdf

- Recommended
  - An early cache paper by Maurice Wilkes
    - Wilkes, "Slave Memories and Dynamic Storage Allocation," IEEE Trans. On Electronic Computers, 1965.

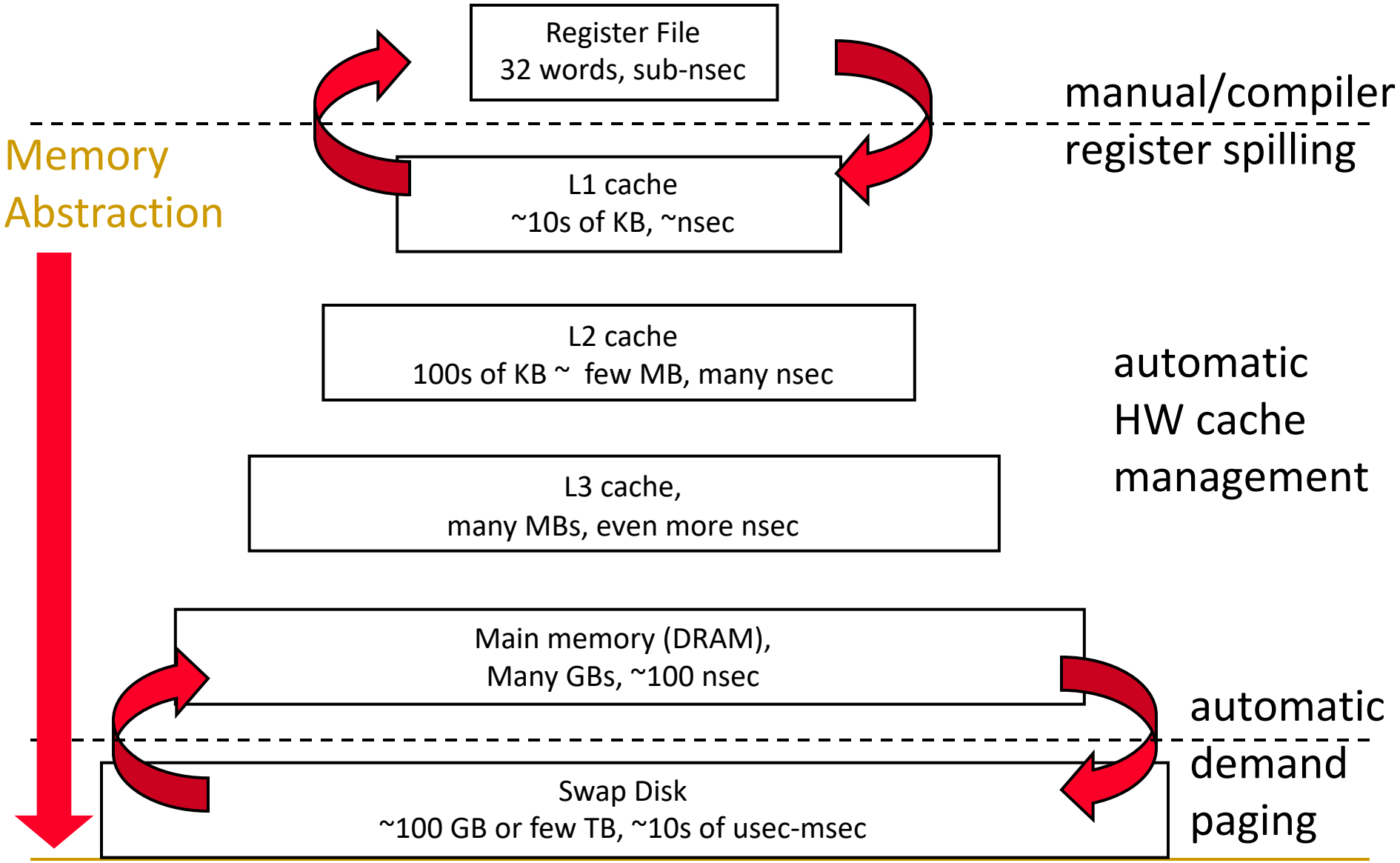
Recall: Memory Hierarchy Example



capacity	latency
10's of KB	≈ 1ns
100's of KB	< 5ns
several MB	≈ 10ns
several GB	≈ 100ns

Kim & Mutlu, "Memory Systems," Computing Handbook, 2014
https://people.inf.ethz.ch/omutlu/pub/memory-systems-introduction_computing-handbook14.pdf

Recall: A Modern Memory Hierarchy



Aside: Remote Memory in Large Servers

- Memory hierarchy extends beyond a single machine
- This enables even higher memory capacity
  - Needed to support modern data-intensive workloads

Compute node (Local)

Low-latency network

Memory node (Remote)

Local memory (and hierarchy)

Remote memory

Calciu+, "Rethinking Software Runtimes for Disaggregated Memory", ASPLOS 2021.

Aside: Remote Memory in Large Servers

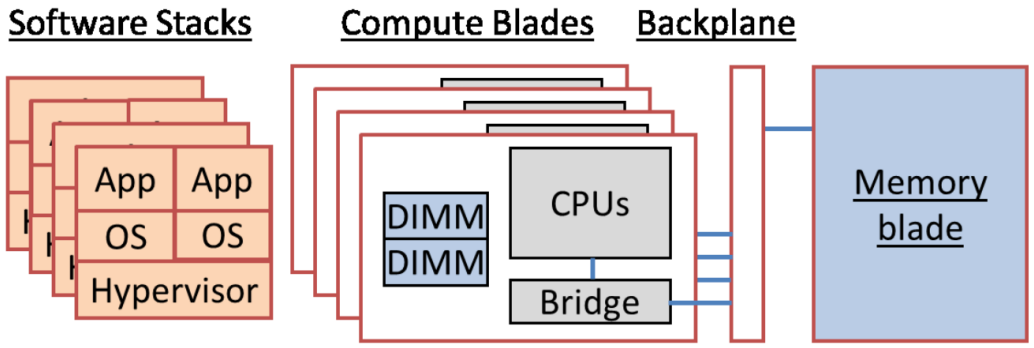


Figure 1. System with memory blades

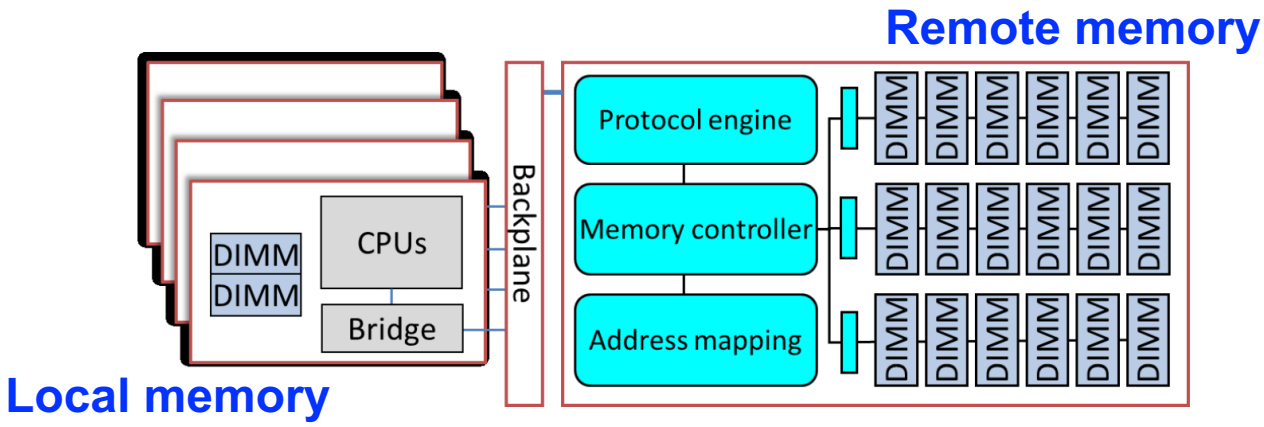


Figure 2. Memory blade architecture

Lim+, "System-level Implications of Disaggregated Memory", HPCA 2012.

Aside: Remote Memory in Large Servers

- Irina Calciu, M. Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli,
  "Rethinking Software Runtimes for Disaggregated Memory"
  Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems (***ASPLOS***), Virtual, March-April 2021.
  [2-page Extended Abstract]
  [Source Code (Officially Artifact Evaluated with All Badges)]
  Officially artifact evaluated as available, reusable and reproducible.

Rethinking Software Runtimes for Disaggregated Memory

Irina Calciu
VMware Research
USA

M. Talha Imran
Penn State University
USA

Ivan Puddu
ETH Zürich
Switzerland

Sanidhya Kashyap
EPFL
Switzerland

Hasan Al Maruf
University of Michigan
USA

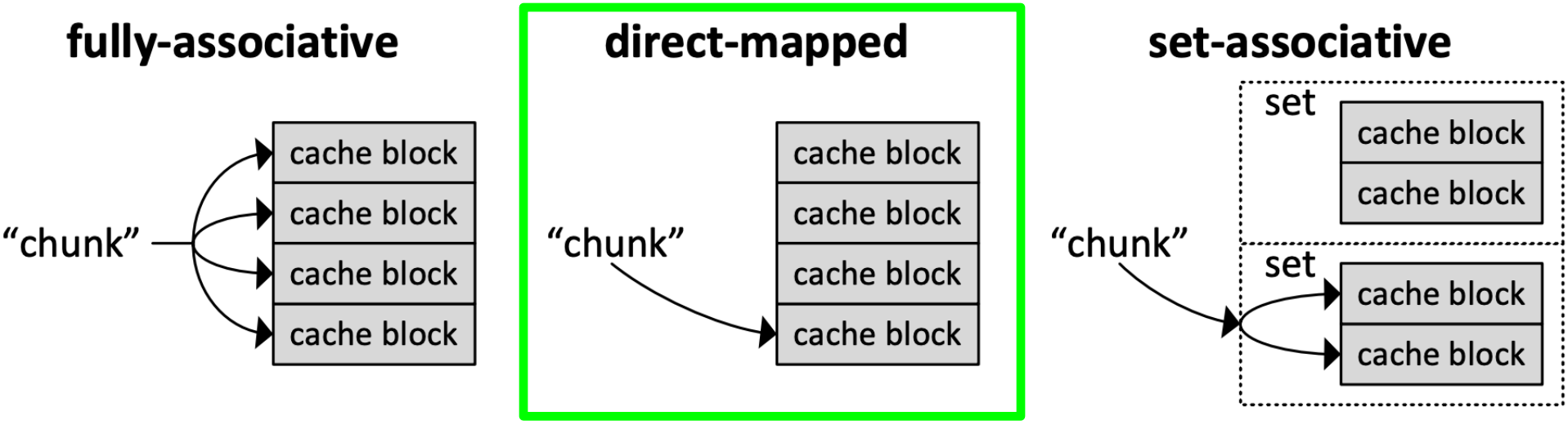
Onur Mutlu
ETH Zürich
Switzerland

Aasheesh Kolli
Penn State University/Google
USA

Recall: Our Toy Cache Example

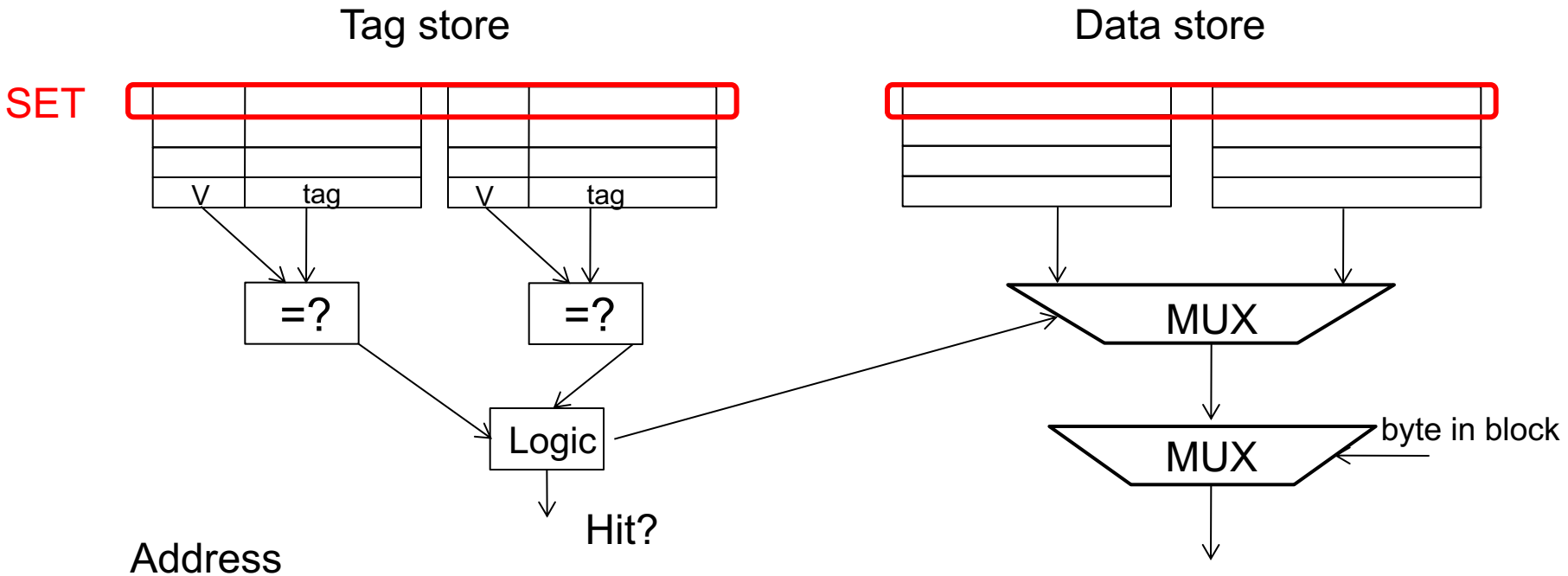
- We will examine a direct-mapped cache first
- Direct-mapped: A given main memory block can be placed in only one possible location in the cache

- Toy example: 256-byte memory, 64-byte cache, 8-byte blocks



Recall: Set Associativity

- Problem: Addresses N and N+8 always conflict in direct mapped cache
- Idea: enable blocks with the same index to map to > 1 cache location
- Example: Instead of having one column of 8, have 2 columns of 4 blocks

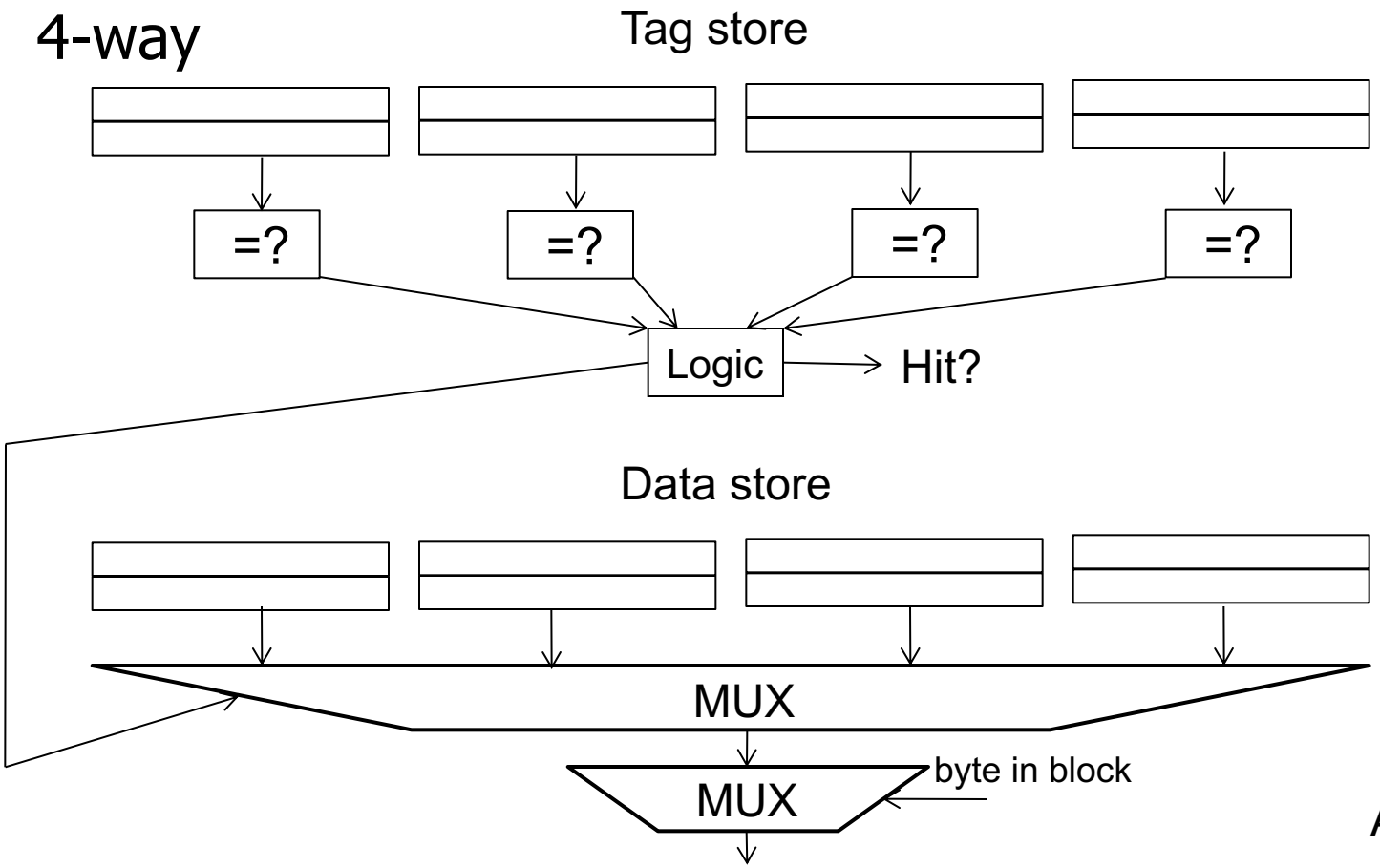


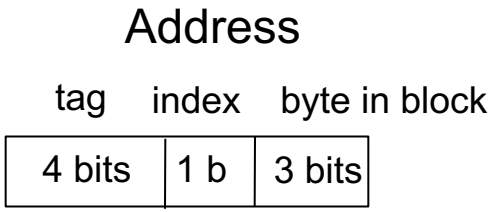
Key idea: Associative memory within the set
+ Accommodates conflicts better (fewer conflict misses)
-- More complex, slower access, larger tag store

2-way set associative cache: Blocks with the same index can map to 2 locations

Recall: Higher Associativity

- 4-way





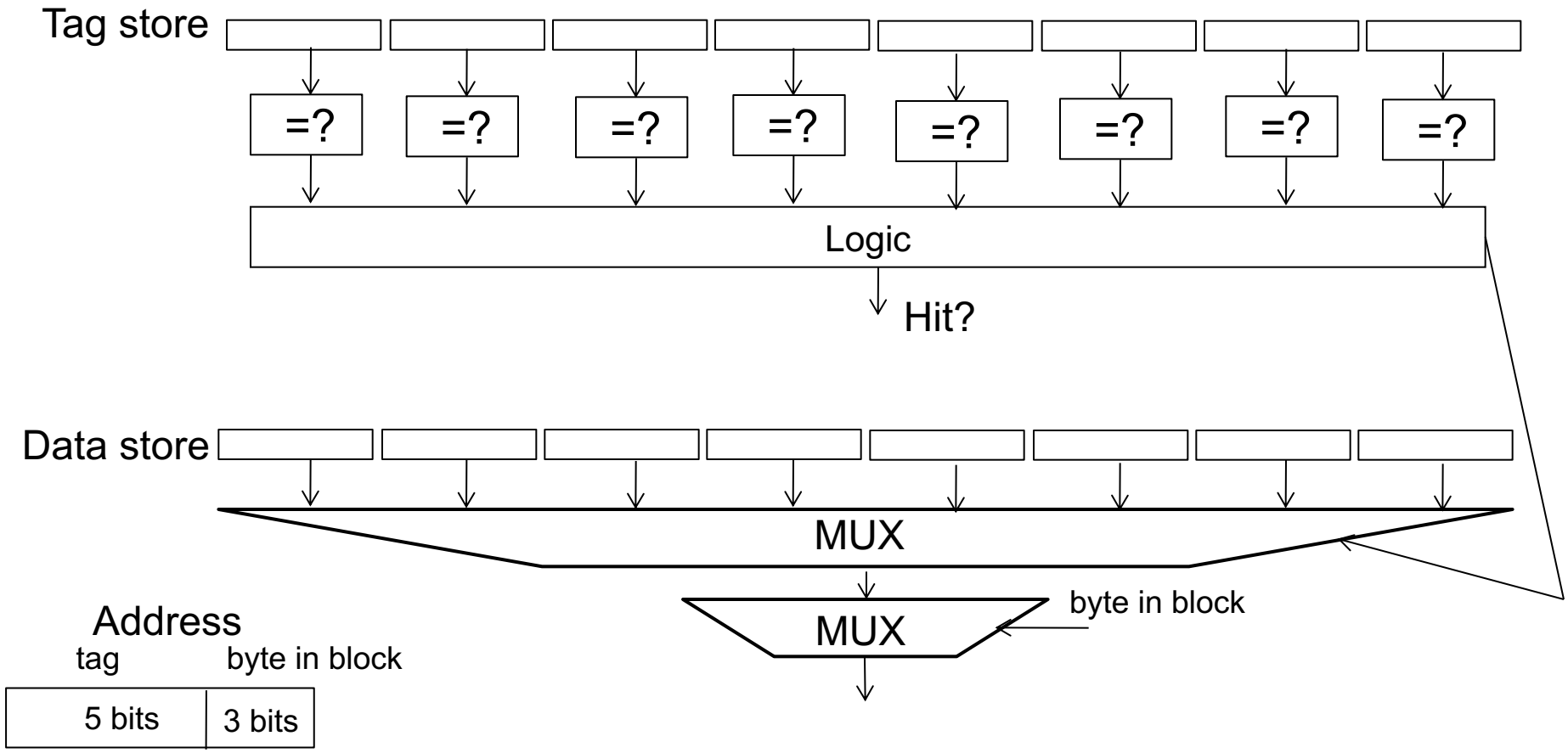
+ Likelihood of conflict misses even lower

-- More tag comparators and wider data mux; larger tag store

4-way set associative cache: Blocks with the same index can map to 4 locations

Recall: Full Associativity

- Fully associative cache
  - A block can be placed in any cache location



Fully associative cache: Any block can map to any location in the cache

Recall: Associativity (and Tradeoffs)

- Degree of associativity: How many blocks can map to the same index (or set)?

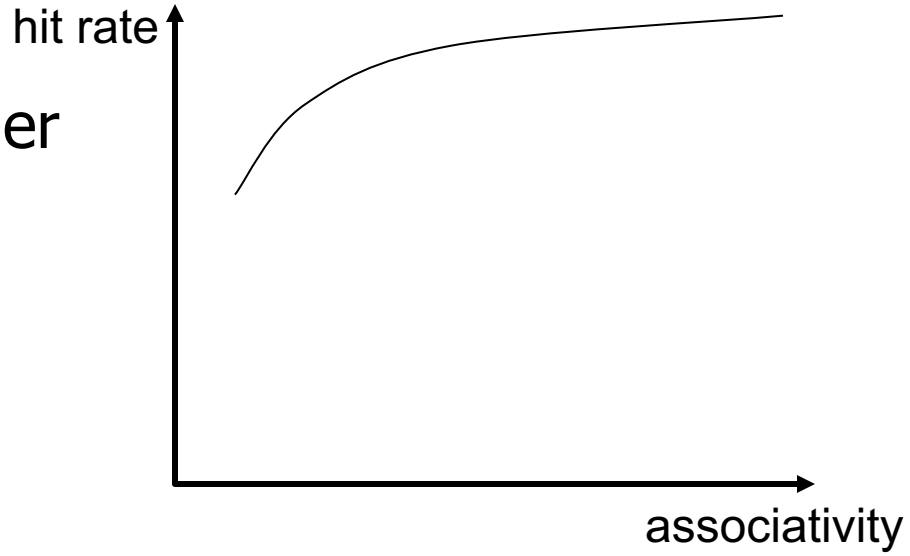
- Higher associativity

  ++ Higher hit rate

  -- Slower cache access time (hit latency and data access latency)

  -- More expensive hardware (more comparators, larger tags/muxes)

- Diminishing returns from higher associativity



Issues in Set-Associative Caches

- Think of each block in a set having a "priority"
  - Indicating how important it is to keep the block in the cache
- Key issue: How do you determine/adjust block priorities?
- There are three key decisions in a set:
  - Insertion, promotion, eviction (replacement)

- Insertion: What happens to priorities on a cache fill?
  - Where to insert the incoming block; whether or not to insert the block
- Promotion: What happens to priorities on a cache hit?
  - Whether and how to change block priority
- Eviction/replacement: What happens to priorities on a cache miss?
  - Which block to evict and how to adjust priorities

Eviction/Replacement Policy

- Which block in the set to replace on a cache miss?
  - Any invalid block first
  - If all are valid, consult the replacement policy
    - Random
    - FIFO
    - Least recently used (how to implement?)
    - Not most recently used
    - Least frequently used?
    - Least costly to re-fetch?
      - Why would memory accesses have different cost?
    - Hybrid replacement policies
    - Optimal replacement policy?

Implementing LRU

- Idea: Evict the least recently accessed block
- Problem: Need to keep track of access order of blocks

- Question: 2-way set associative cache:
  - What do you minimally need to implement LRU perfectly?
- Question: 4-way set associative cache:
  - What do you minimally need to implement LRU perfectly?
  - How many different access orders are possible for the 4 blocks in the set?
  - How many bits needed to encode the LRU order of a block?
  - What is the logic needed to determine the LRU victim?
- Repeat for N-way set associative cache

Approximations of LRU

- Most modern processors do **not** implement "true LRU" (also called "perfect LRU") in highly-associative caches

- Why?
  - True LRU is complex
  - LRU is an approximation to predict locality anyway (i.e., not the best possible cache management policy)

- Examples:
  - Not MRU (not most recently used)
  - Hierarchical LRU: divide the N-way set into M "groups", track the MRU group and the MRU way in each group
  - Victim-NextVictim Replacement: Only keep track of the victim and the next victim

Cache Replacement Policy: LRU or Random

- LRU vs. Random: Which one is better?
  - Example: 4-way cache, cyclic references to A, B, C, D, E
    - 0% hit rate with LRU policy
- Set thrashing: When the "program working set" in a set is larger than set associativity
  - Random replacement policy is better when thrashing occurs
- In practice:
  - Performance of replacement policy depends on workload
  - Average hit rate of LRU and Random are similar
- Best of both Worlds: Hybrid of LRU and Random
  - How to choose between the two? Set sampling
    - See Qureshi et al., "A Case for MLP-Aware Cache Replacement," ISCA 2006.

What Is the Optimal Replacement Policy?

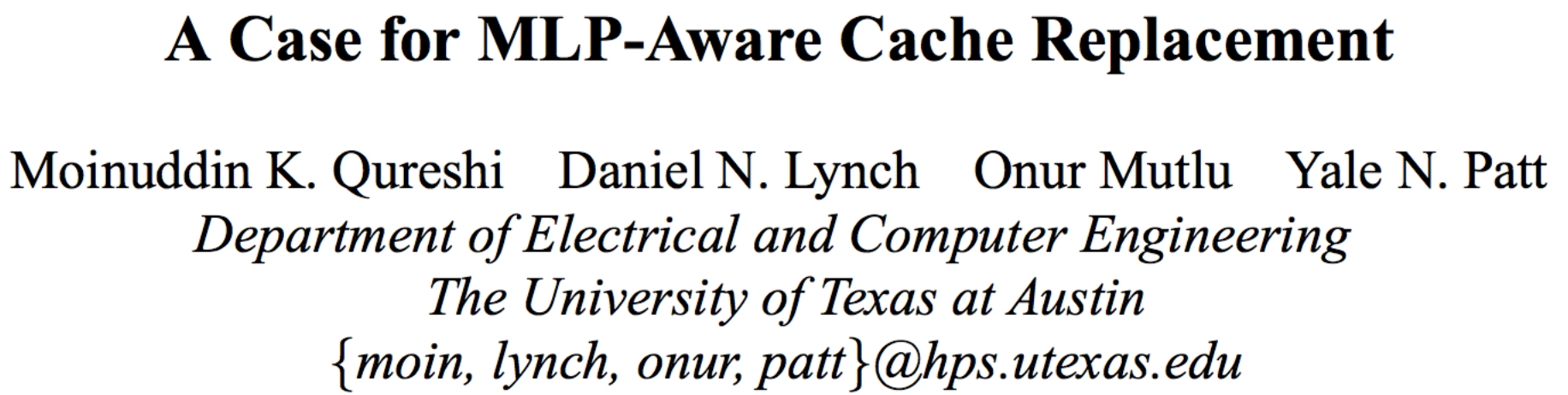
- Belady's OPT
  - Replace the block that is going to be referenced furthest in the future by the program
  - Belady, "A study of replacement algorithms for a virtual-storage computer," IBM Systems Journal, 1966.
  - How do we implement this? Simulate?

- Is this optimal for minimizing miss rate?
- Is this optimal for minimizing execution time?
  - No. Cache miss latency/cost varies from block to block!
  - Two reasons: Where miss is serviced from and miss overlapping
  - Qureshi et al. "A Case for MLP-Aware Cache Replacement," ISCA 2006.

Recommended Reading

- Key observation: Some misses more costly than others as their latency is exposed as stall time. Reducing miss rate is not always good for performance. Cache replacement should take into account cost of misses.

- Moinuddin K. Qureshi, Daniel N. Lynch, Onur Mutlu, and Yale N. Patt, **"A Case for MLP-Aware Cache Replacement"**
  *Proceedings of the 33rd International Symposium on Computer Architecture (**ISCA**)*, pages 167-177, Boston, MA, June 2006. Slides (ppt)



A Case for MLP-Aware Cache Replacement

Moinuddin K. Qureshi Daniel N. Lynch Onur Mutlu Yale N. Patt
Department of Electrical and Computer Engineering
The University of Texas at Austin
{moin, lynch, onur, patt}@hps.utexas.edu

What's In A Tag Store Entry?

- Valid bit
- Tag
- Replacement policy bits

- Dirty bit?
  - Write back vs. write through caches

Handling Writes (I)

- When do we write the modified data in a cache to the next level?
  - Write through: At the time the write happens
  - Write back: When the block is evicted

- Write-back cache

  + Can combine multiple writes to the same block before eviction
    - Potentially saves bandwidth between cache levels + saves energy

  -- Need a bit in the tag store indicating the block is "dirty/modified"

- Write-through cache

  + Simpler design

  + All levels are up to date & consistent → Simpler cache coherence: no need to check close-to-processor caches' tag stores for presence

  -- More bandwidth intensive; no combining of writes

Handling Writes (II)

- Do we allocate a cache block on a write miss?
  - Allocate on write miss: Yes
  - No-allocate on write miss: No

- Allocate on write miss

  + Can combine writes instead of writing each individually to next level

  + Simpler because write misses can be treated the same way as read misses

  -- Requires transfer of the whole cache block

- No-allocate

  + Conserves cache space if locality of written blocks is low (potentially better cache hit rate)

Handling Writes (III)

- What if the processor writes to an entire block over a small amount of time?

- Is there any need to bring the block into the cache from memory in the first place?

- Why do we not simply write to only a *portion* of the block, i.e., subblock?
  - E.g., 4 bytes out of 64 bytes
  - Problem: Valid and dirty bits are associated with the entire 64 bytes, not with each individual 4 bytes

Subblocked (Sectored) Caches

- Idea: Divide a block into subblocks (or sectors)
  - Have separate valid and dirty bits for each subblock (sector)
  - Allocate only a subblock (or a subset of subblocks) on a request

++ No need to transfer the entire cache block into the cache
(A write simply validates and updates a subblock)

++ More freedom in transferring subblocks into the cache (a cache block does not need to be in the cache fully)
(How many subblocks do you transfer on a read?)

-- More complex design; more valid and dirty bits

-- May not exploit spatial locality fully

Instruction vs. Data Caches

- Separate or Unified?

- Pros and Cons of Unified:

  + Dynamic sharing of cache space → **better overall cache utilization**: no overprovisioning that might happen with static partitioning of cache space (i.e., separate I and D caches)

  -- Instructions and data can evict/thrash each other (i.e., no guaranteed space for either)

  -- I and D are accessed in different places in the pipeline. Where do we place the unified cache for fast access?

- First level caches are almost always split
  - Mainly for the last reason above – pipeline constraints
- Outer level caches are almost always unified

Multi-level Cache Design & Management

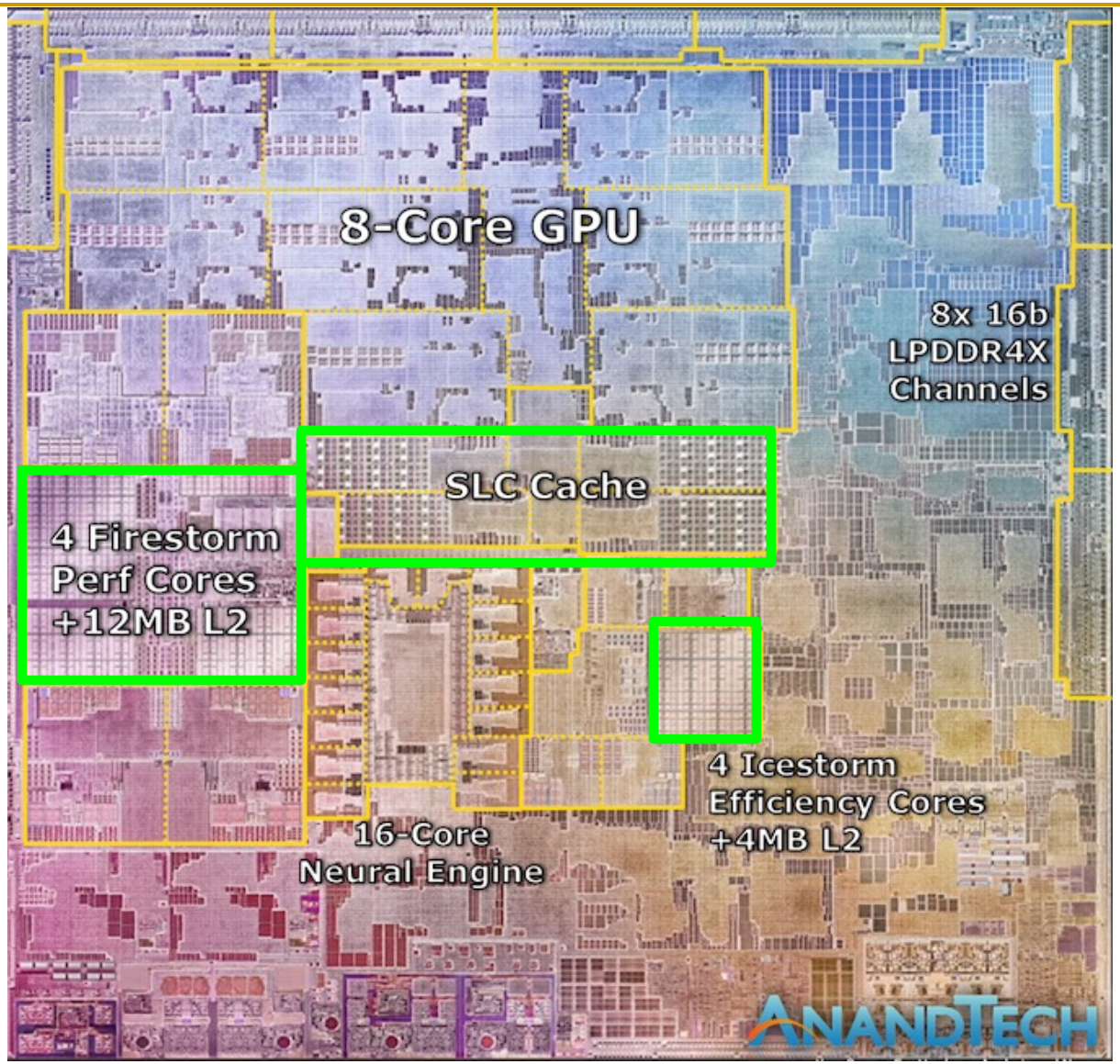
- Cache level greatly affects cache design & management
- First-level caches (instruction and data)
  - Decisions very much affected by cycle time & pipeline structure
  - Small, lower associativity; latency is critical
  - Tag store and data store are usually accessed in parallel
- Second-level caches
  - Decisions need to balance hit rate and access latency
  - Usually large and highly associative; latency not as important
  - Tag store and data store can be accessed serially
- Further-level (larger) caches
  - Access energy is a larger problem due to cache sizes
  - Tag store and data store are usually accessed serially

Serial vs. Parallel Access of Cache Levels

- Parallel: Next level cache accessed in parallel with the previous level → a form of speculative access
  - + Faster access to data if previous level misses
  - -- Unnecessary accesses to next level if previous level hits

- Serial: Next level cache accessed only if previous-level misses
  - -- Slower access to data if previous level misses
  - + No wasted accesses to next level if previous level hits
  - Next level does not see the same accesses as the previous
    - Previous level acts as a filter (filters some temporal & spatial locality)
    - **Management policies are different across cache levels**

Deeper and Larger Cache Hierarchies



Apple M1, 2021

Source: https://www.anandtech.com/show/16252/mac-mini-apple-m1-tested

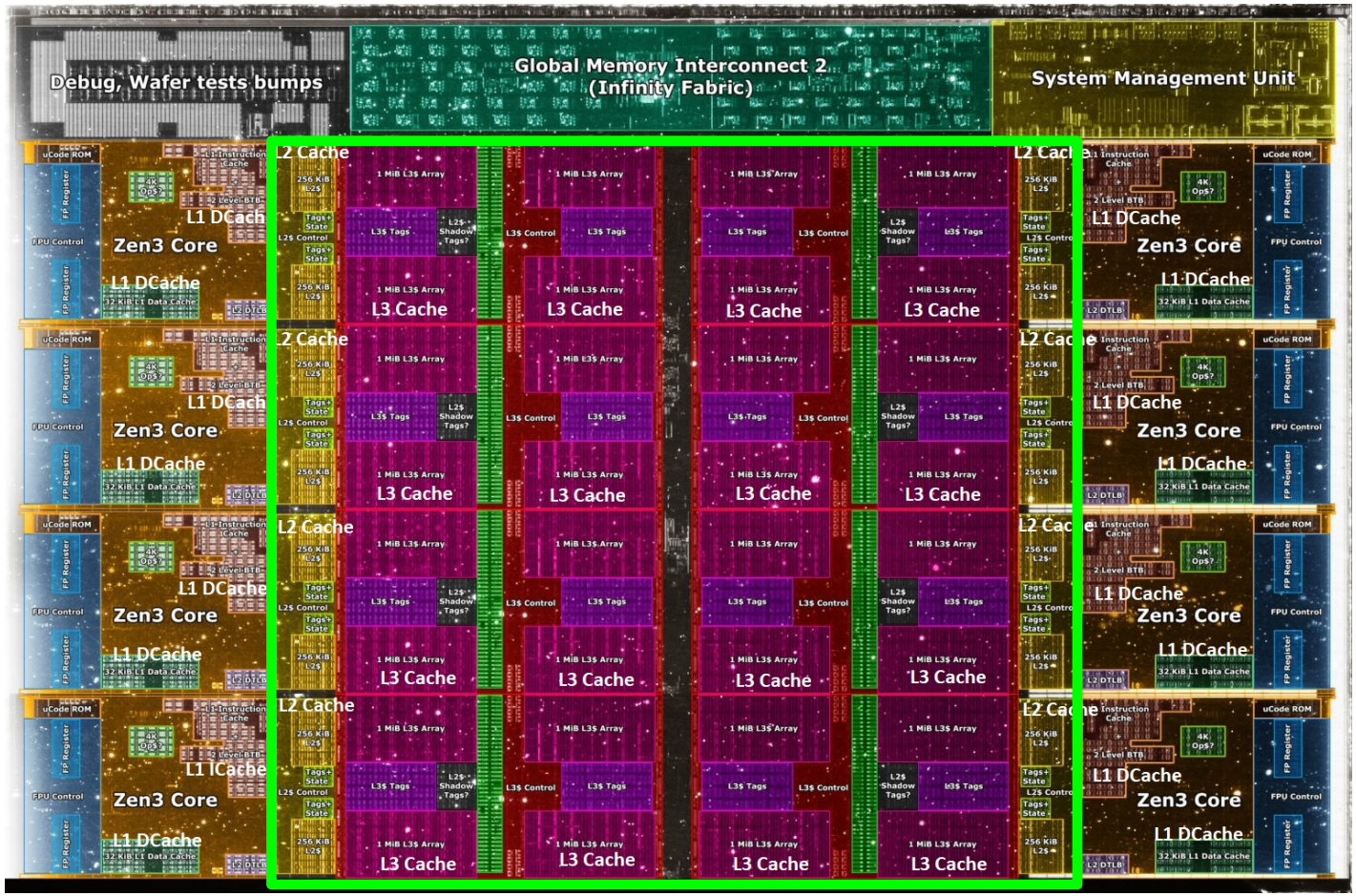
Deeper and Larger Cache Hierarchies

10nm ESF=Intel 7 Alder Lake die shot (~209mm²) from Intel: https://www.intel.com/content/www/us/en/newsroom/news/12th-gen-core-processors.html
Die shot interpretation by Locuza, October 2021

Intel Alder Lake, 2021

Source: https://twitter.com/Locuza_/status/1454152714930331652

Deeper and Larger Cache Hierarchies



AMD Ryzen 5000, 2020

Core Count:

8 cores/16 threads

L1 Caches:

32 KB per core

L2 Caches:

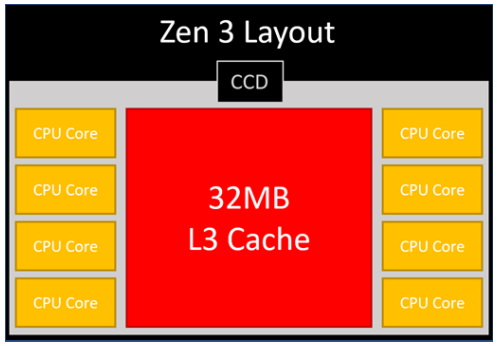
512 KB per core

L3 Cache:

32 MB shared

https://wccftech.com/amd-ryzen-5000-zen-3-vermeer-undressed-high-res-die-shots-close-ups-pictured-detailed/

AMD's 3D Last Level Cache (2021)

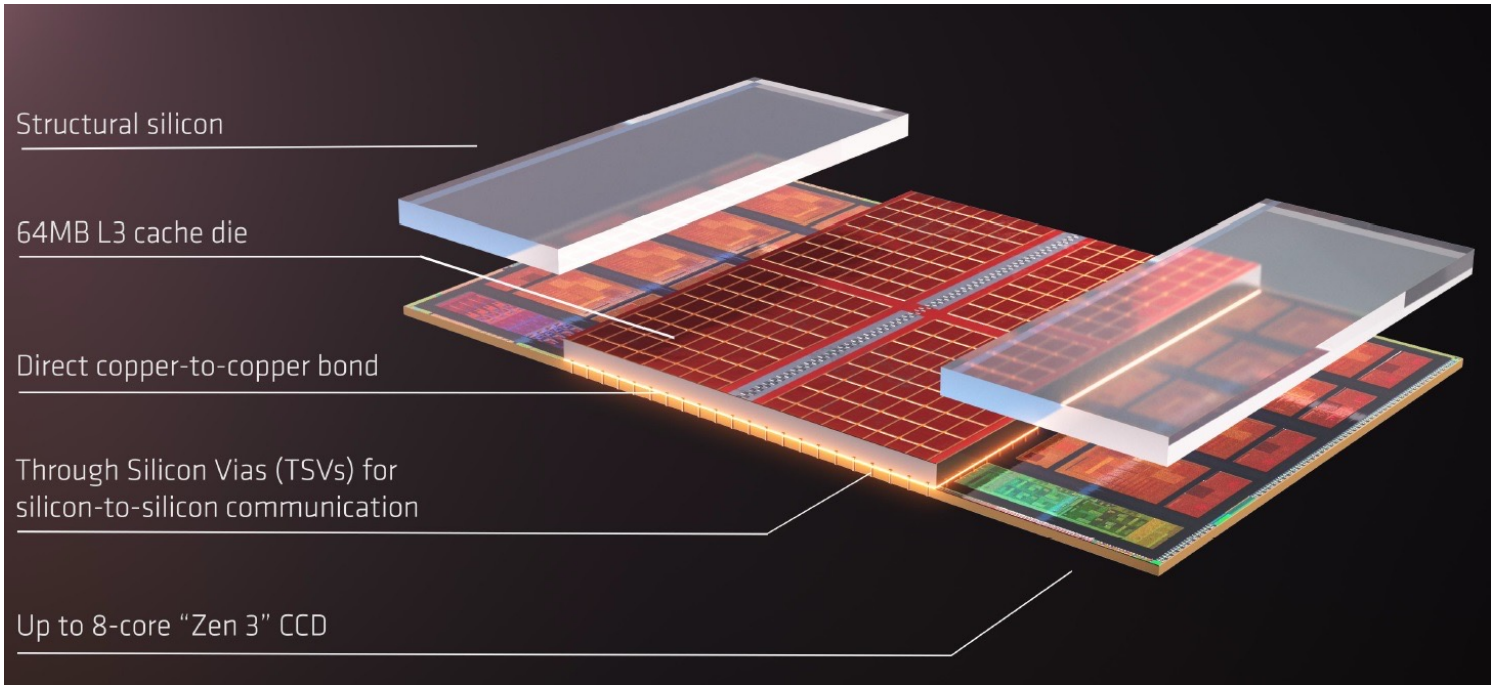


https://community.microcenter.com/discussion/5134/comparing-zen-3-to-zen-2

AMD increases the L3 size of their 8-core Zen 3 processors from 32 MB to 96 MB

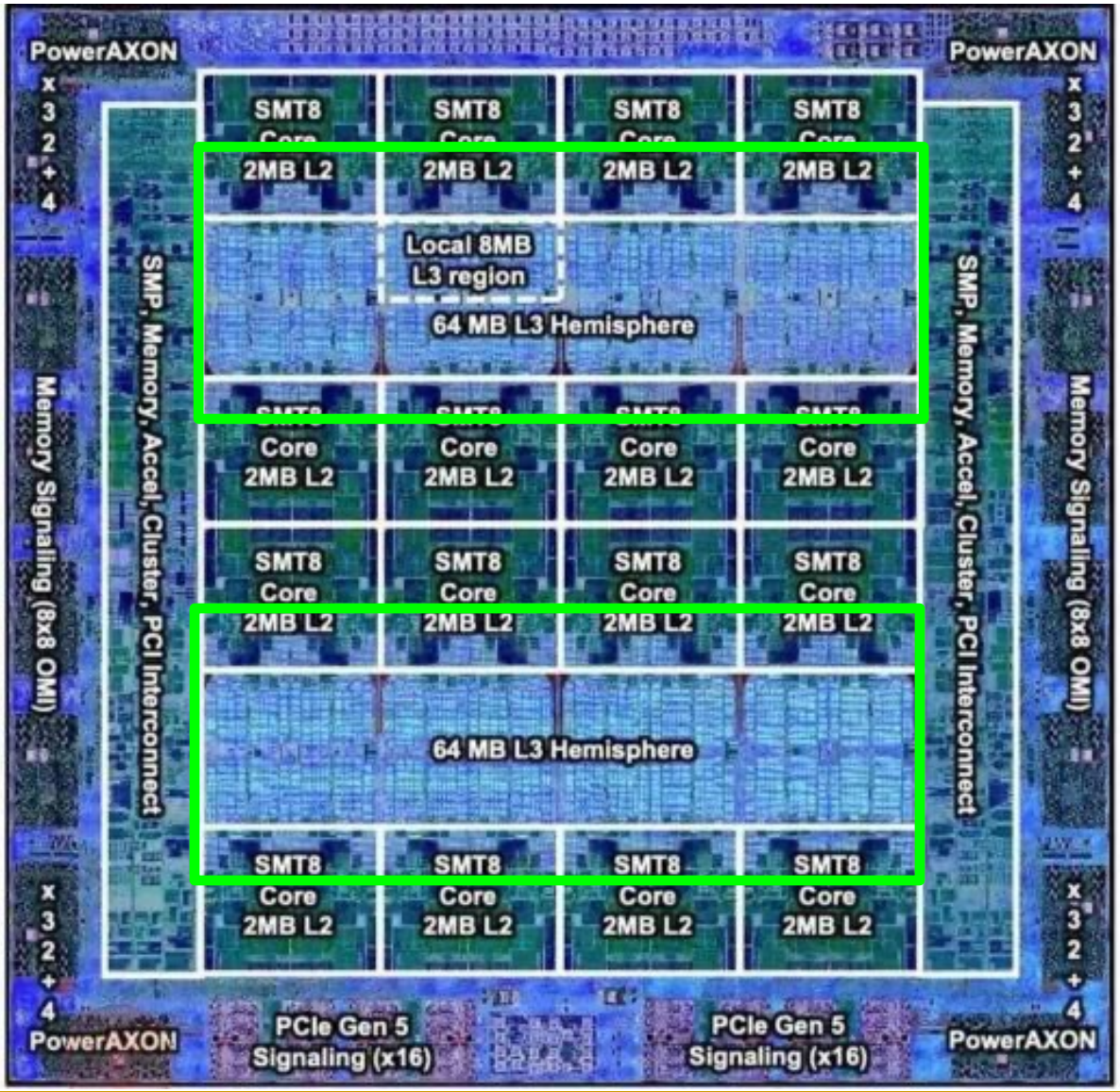
Additional 64 MB L3 cache die stacked on top of the processor die

- Connected using Through Silicon Vias (TSVs)
- Total of 96 MB L3 cache



https://youtu.be/gqAYMx34euU
https://www.tech-critter.com/amd-keynote-computex-2021/

Deeper and Larger Cache Hierarchies



IBM POWER10, 2020

Cores:

15-16 cores,
8 threads/core

L2 Caches:

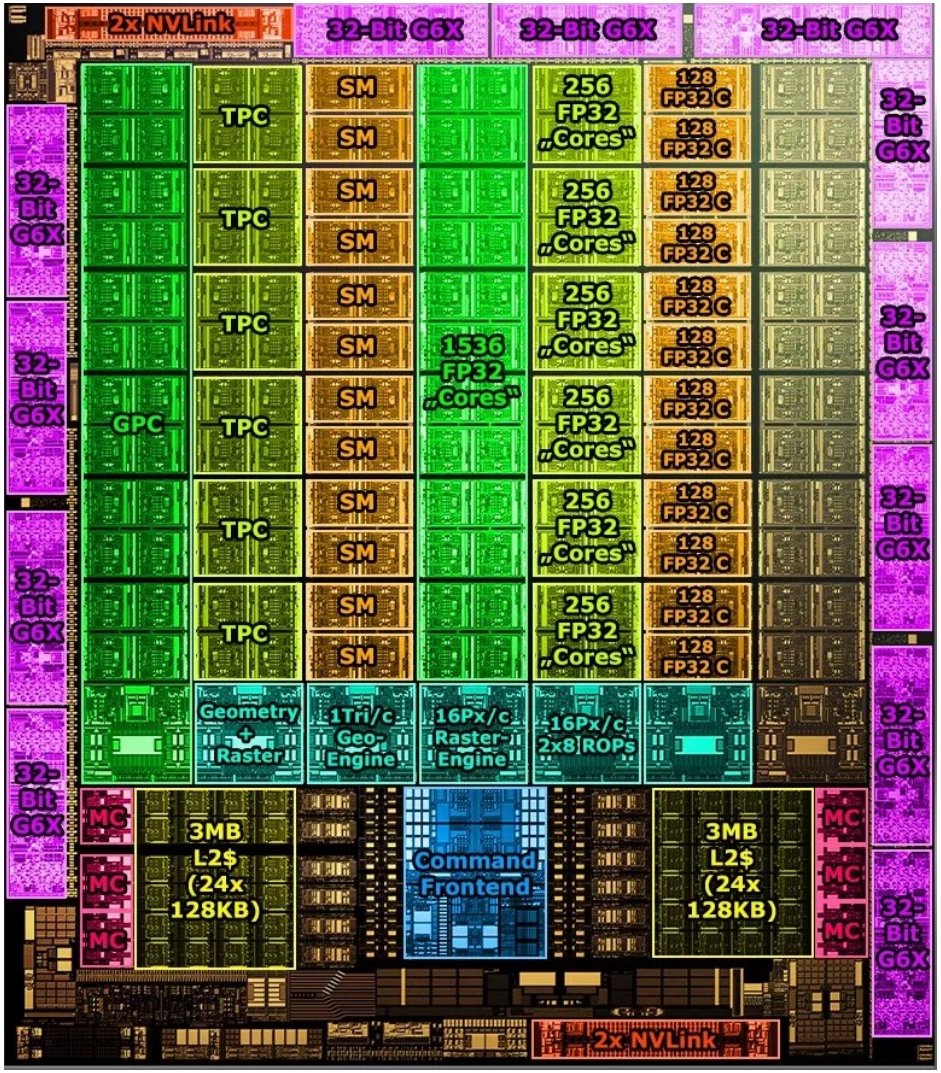
2 MB per core

L3 Cache:

120 MB shared

https://www.it-techblog.de/ibm-power10-prozessor-mehr-speicher-mehr-tempo-mehr-sicherheit/09/2020/

Deeper and Larger Cache Hierarchies



Nvidia Ampere, 2020

Cores:

128 Streaming Multiprocessors

L1 Cache or Scratchpad:

192KB per SM

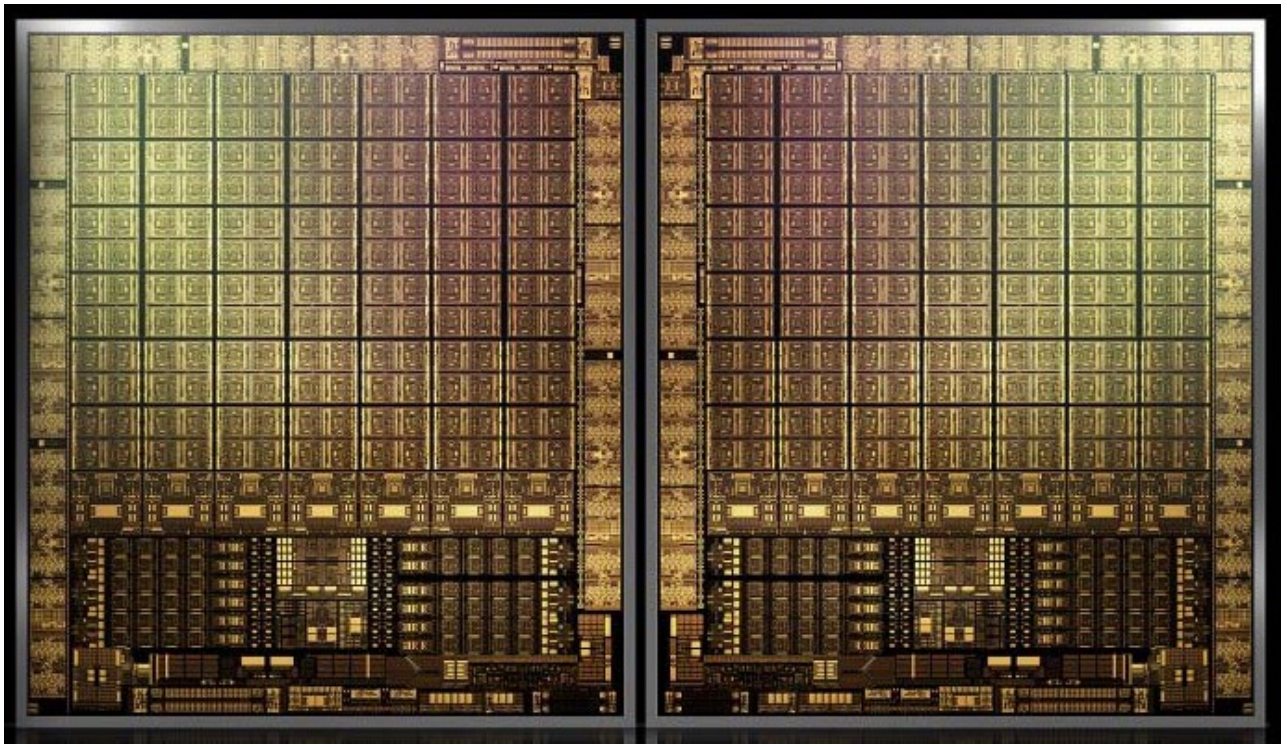
Can be used as L1 Cache and/or Scratchpad

L2 Cache:

40 MB shared

https://www.tomshardware.com/news/infrared-photographer-photos-nvidia-ga102-ampere-silicon

Deeper and Larger Cache Hierarchies



Nvidia Hopper, 2022

Cores:

144 Streaming Multiprocessors

L1 Cache or Scratchpad:

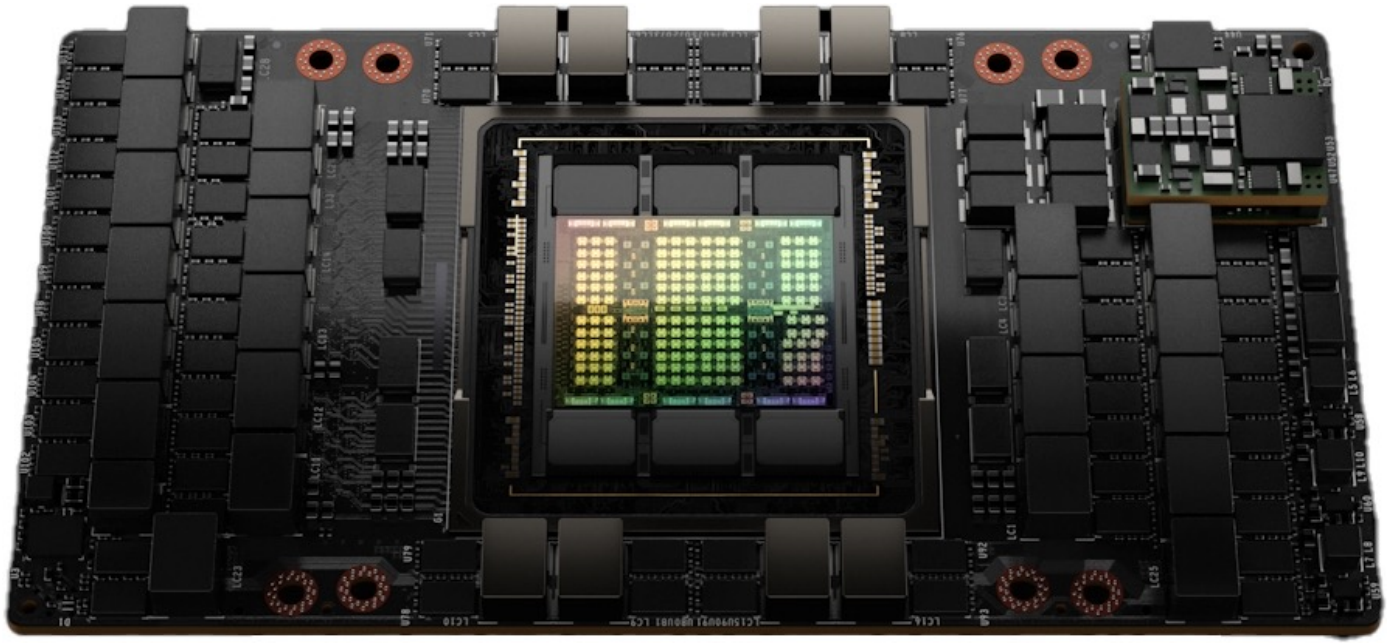
256KB per SM
Can be used as L1 Cache and/or Scratchpad

L2 Cache:

60 MB shared

https://wccftech.com/nvidia-hopper-gpus-featuring-mcm-technology-tape-out-soon-rumor/

Deeper and Larger Cache Hierarchies



Nvidia Hopper, 2022

Cores:

144 Streaming Multiprocessors

L1 Cache or Scratchpad:

256KB per SM

Can be used as L1 Cache and/or Scratchpad

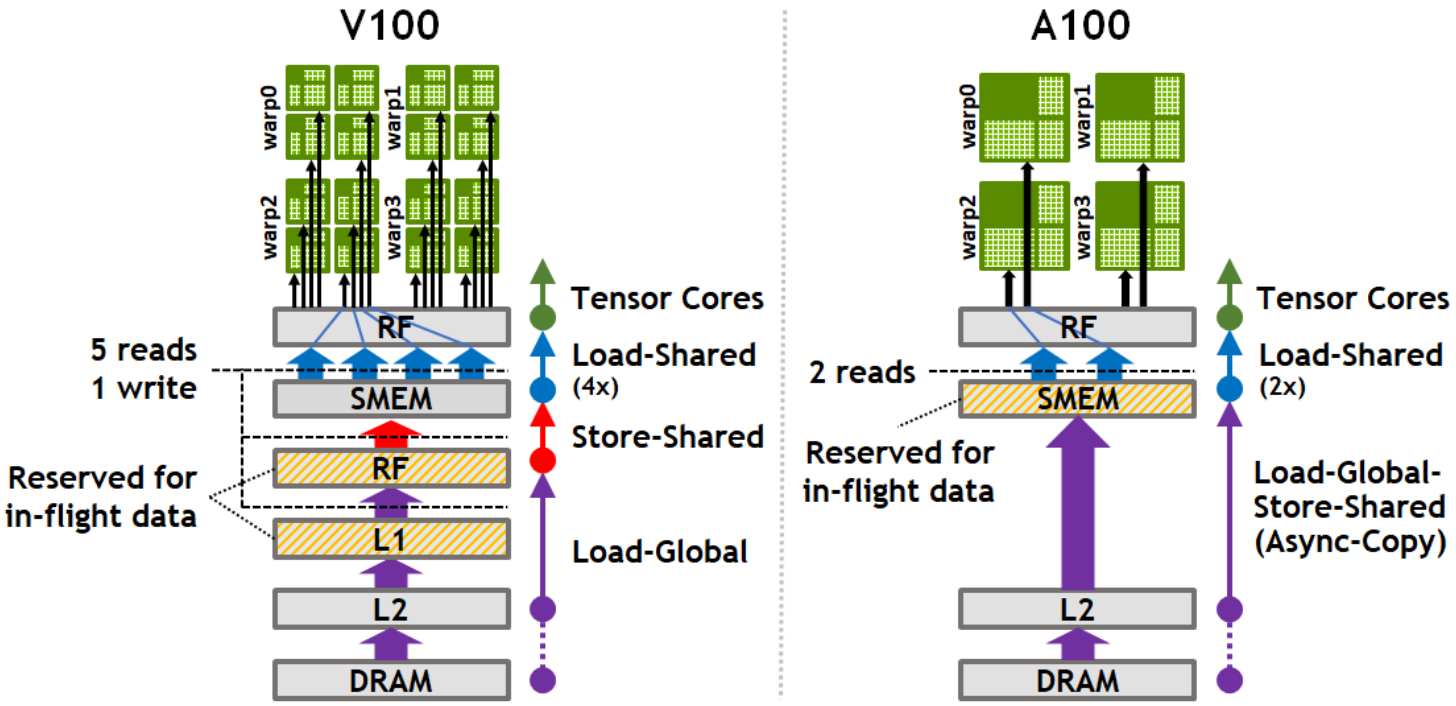
L2 Cache:

60 MB shared

https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/

NVIDIA V100 & A100 Memory Hierarchy

- Example of data movement between GPU global memory (DRAM) and GPU cores.

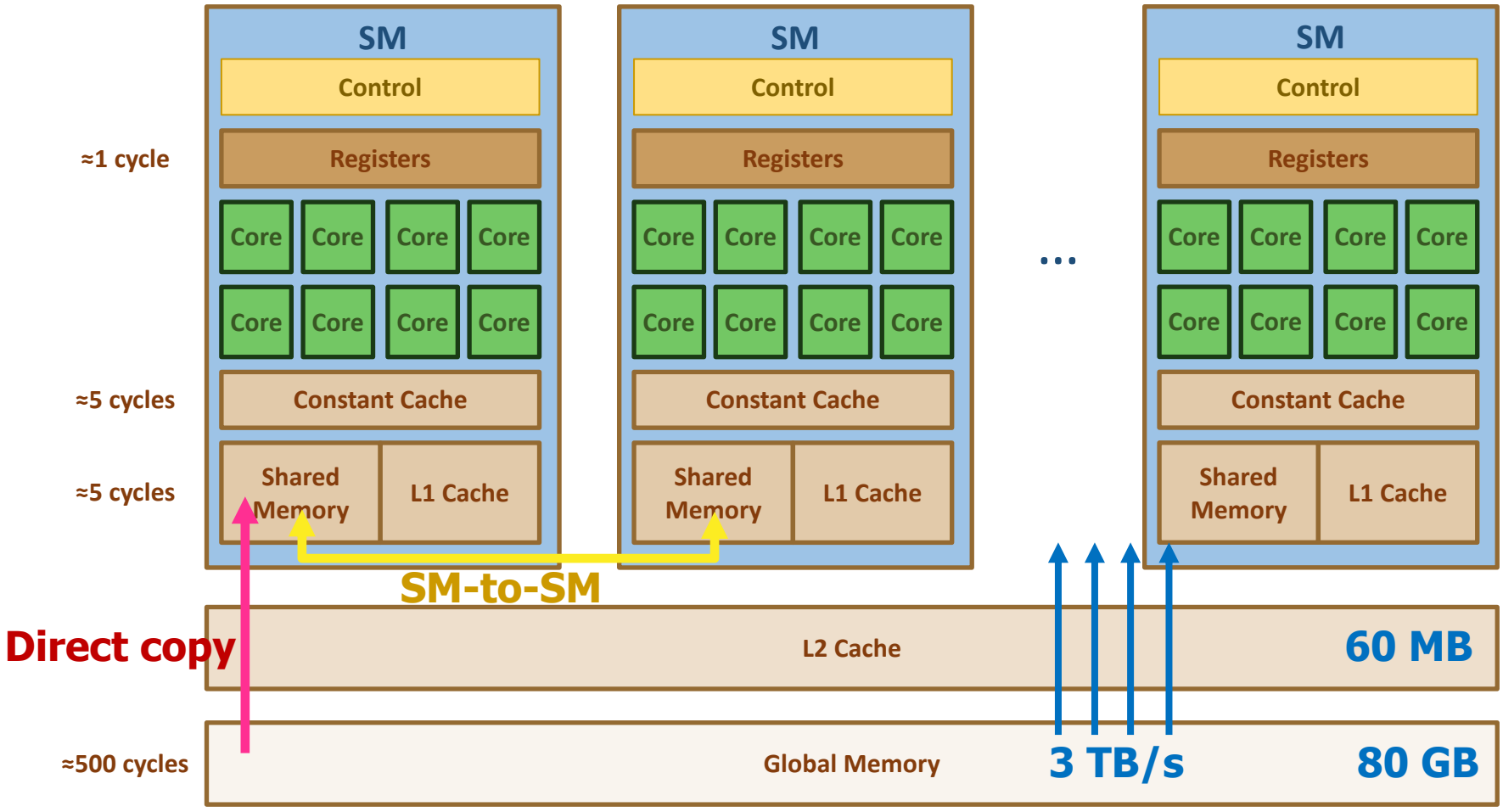


A100 improves SM bandwidth efficiency with a new load-global-store-shared asynchronous copy instruction that bypasses L1 cache and register file (RF). Additionally, A100's more efficient Tensor Cores reduce shared memory (SMEM) loads.

A100 feature:
Direct copy from L2 to scratchpad, bypassing L1 and register file.

https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf

Memory in the NVIDIA H100 GPU



Slide credit: Izzat El Hajj

Multi-Level Cache Design Decisions

- Which level(s) to place a block into (from memory)?
- Which level(s) to evict a block to (from an inner level)?
- Bypassing vs. non-bypassing levels
- Inclusive, exclusive, non-inclusive hierarchies
  - Inclusive: a block in an inner level is always included also in an outer level → simplifies cache coherence
  - Exclusive: a block in an inner level does not exist in an outer level → better utilizes space in the entire hierarchy
  - Non-inclusive: a block in an inner level may or may not be included in an outer level → relaxes design decisions

Cache Performance

Cache Parameters vs. Miss/Hit Rate

- Cache size
- Block size
- Associativity
- Replacement policy
- Insertion/Placement policy
- Promotion Policy

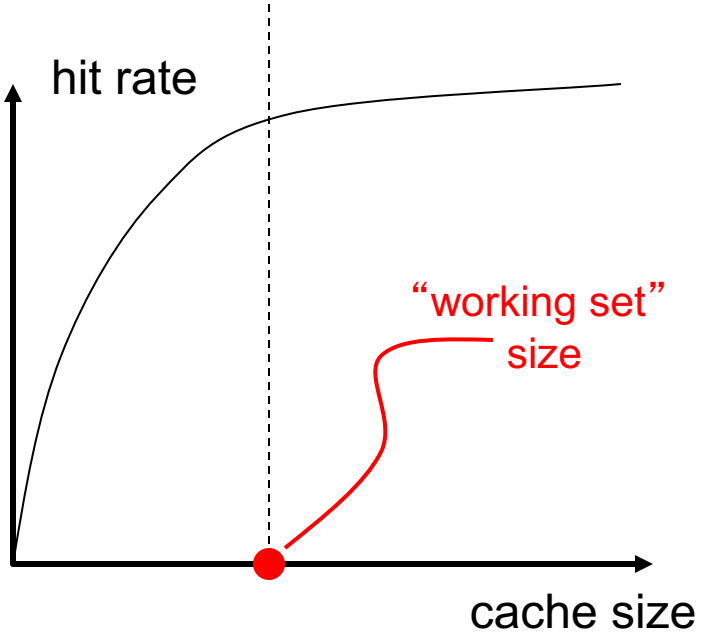
Cache Size

- Cache size: total data capacity (not including tag store)
  - bigger cache can exploit temporal locality better

- Too large a cache adversely affects hit and miss latency
  - bigger is slower

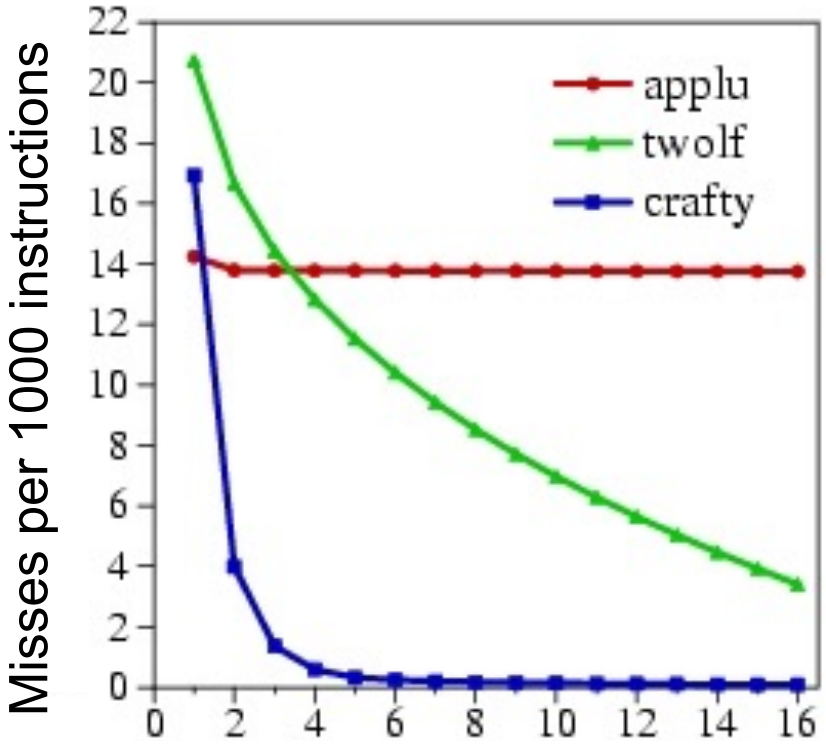
- Too small a cache
  - does not exploit temporal locality well
  - useful data replaced often

- Working set: entire set of data the executing application references
  - Within a time interval



Benefits of Larger Caches Widely Varies

- Benefits of cache size widely varies across applications



Low Cache Utility

High Cache Utility

Saturating Cache Utility

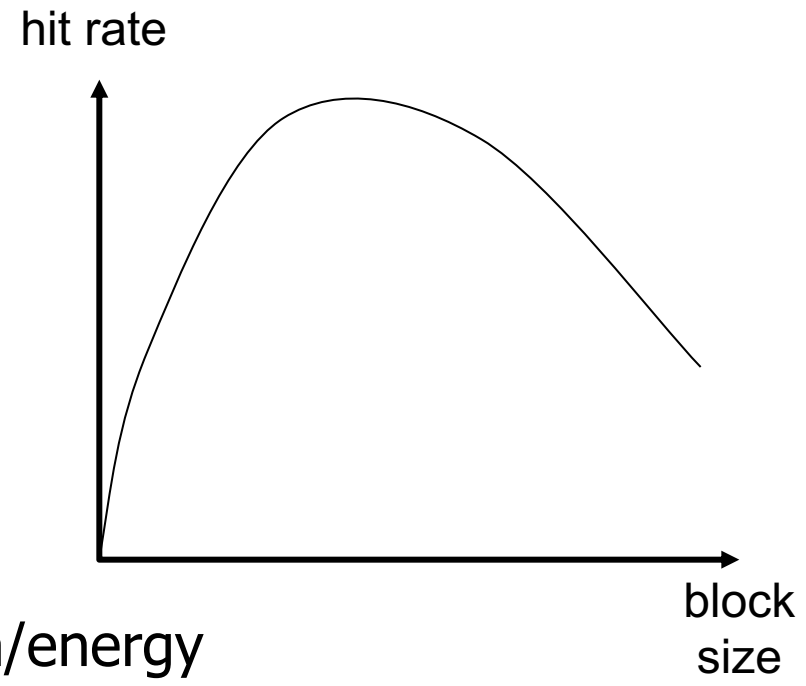
Qureshi and Patt, "Utility-Based Cache Partitioning," MICRO 2006.

Block Size

- Block size is the data that is associated with an address tag
  - not necessarily the unit of transfer between hierarchies
    - Sub-blocking: A block divided into multiple pieces (each w/ V/D bits)

- Too small blocks
  - do not exploit spatial locality well
  - have larger tag overhead

- Too large blocks
  - too few total blocks → do not exploit temporal locality well
  - waste cache space and bandwidth/energy if spatial locality is not high



Large Blocks: Critical-Word and Subblocking

- Large cache blocks can take a long time to fill into the cache
  - Idea: Fill cache block critical-word first
  - Supply the critical data to the processor immediately

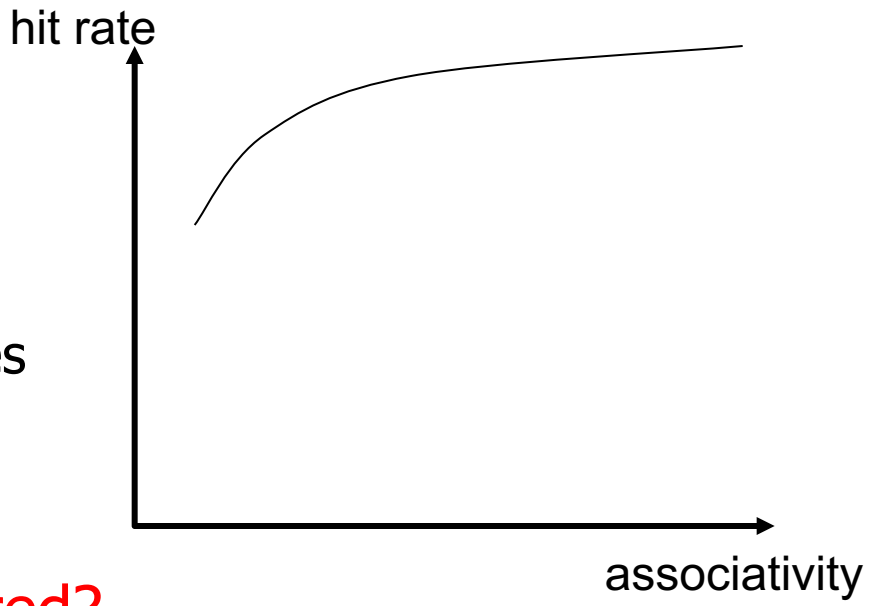
- Large cache blocks can waste bus bandwidth
  - Idea: Divide a block into subblocks
  - Associate separate valid and dirty bits for each subblock
  - Recall: When is this useful?

Associativity

- How many blocks can be present in the same index (i.e., set)?

- Larger associativity
  - lower miss rate (reduced conflicts)
  - higher hit latency and area cost

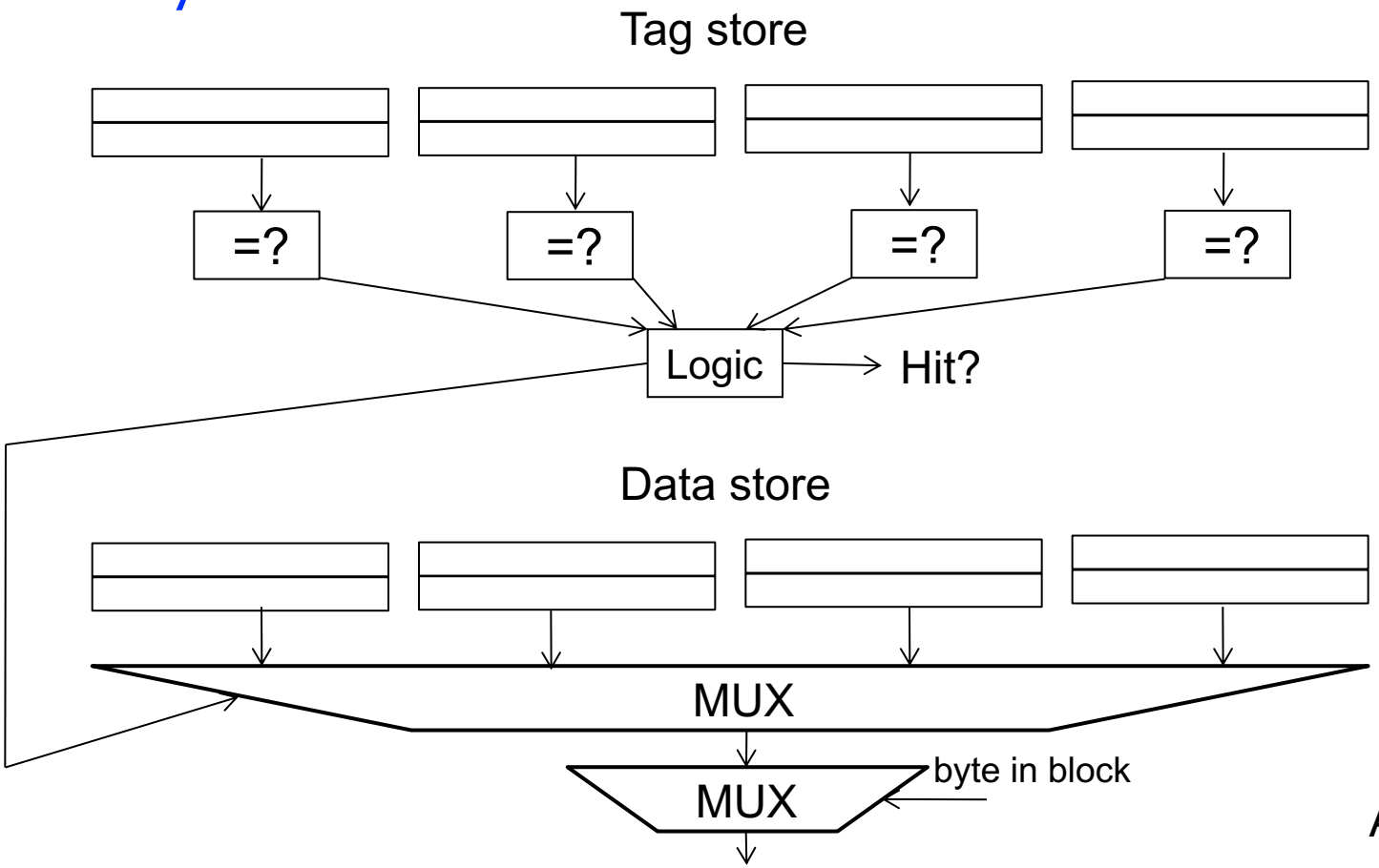
- Smaller associativity
  - lower cost
  - lower hit latency
    - Especially important for L1 caches



- Is power of 2 associativity required?

Recall: Higher Associativity (4-way)

- 4-way

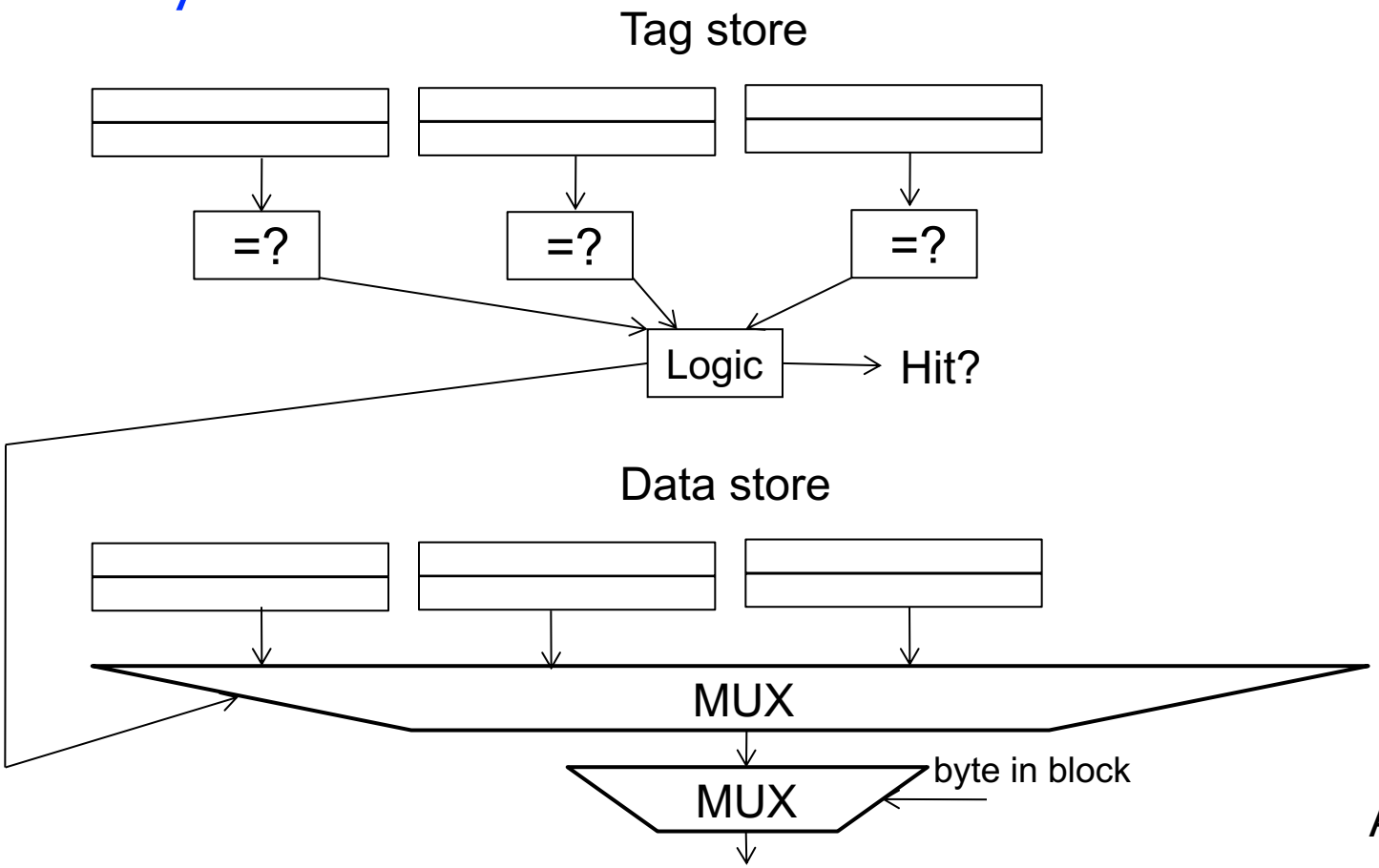


Address

tag	index	byte in block
4 bits	1 b	3 bits

Higher Associativity (3-way)

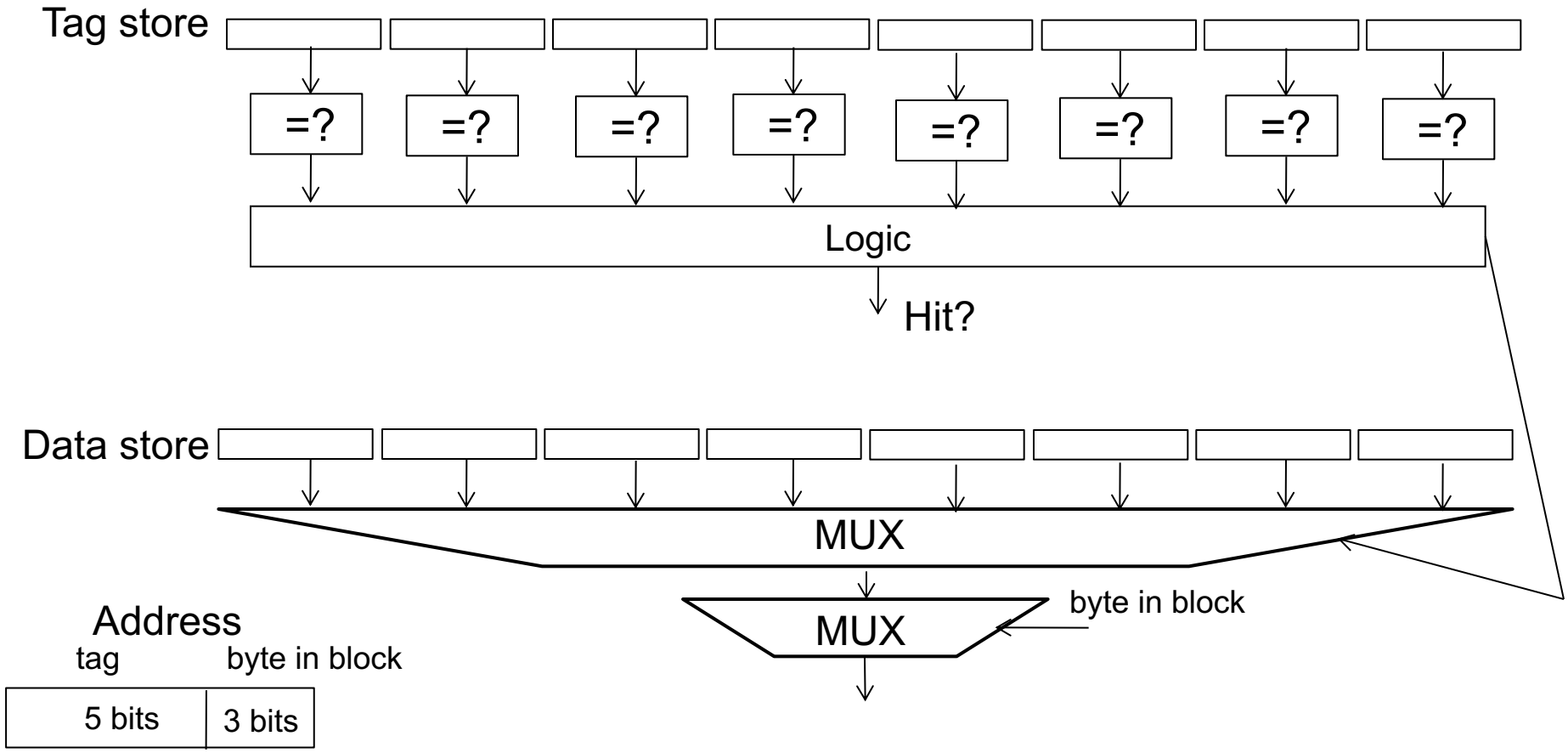
- 3-way



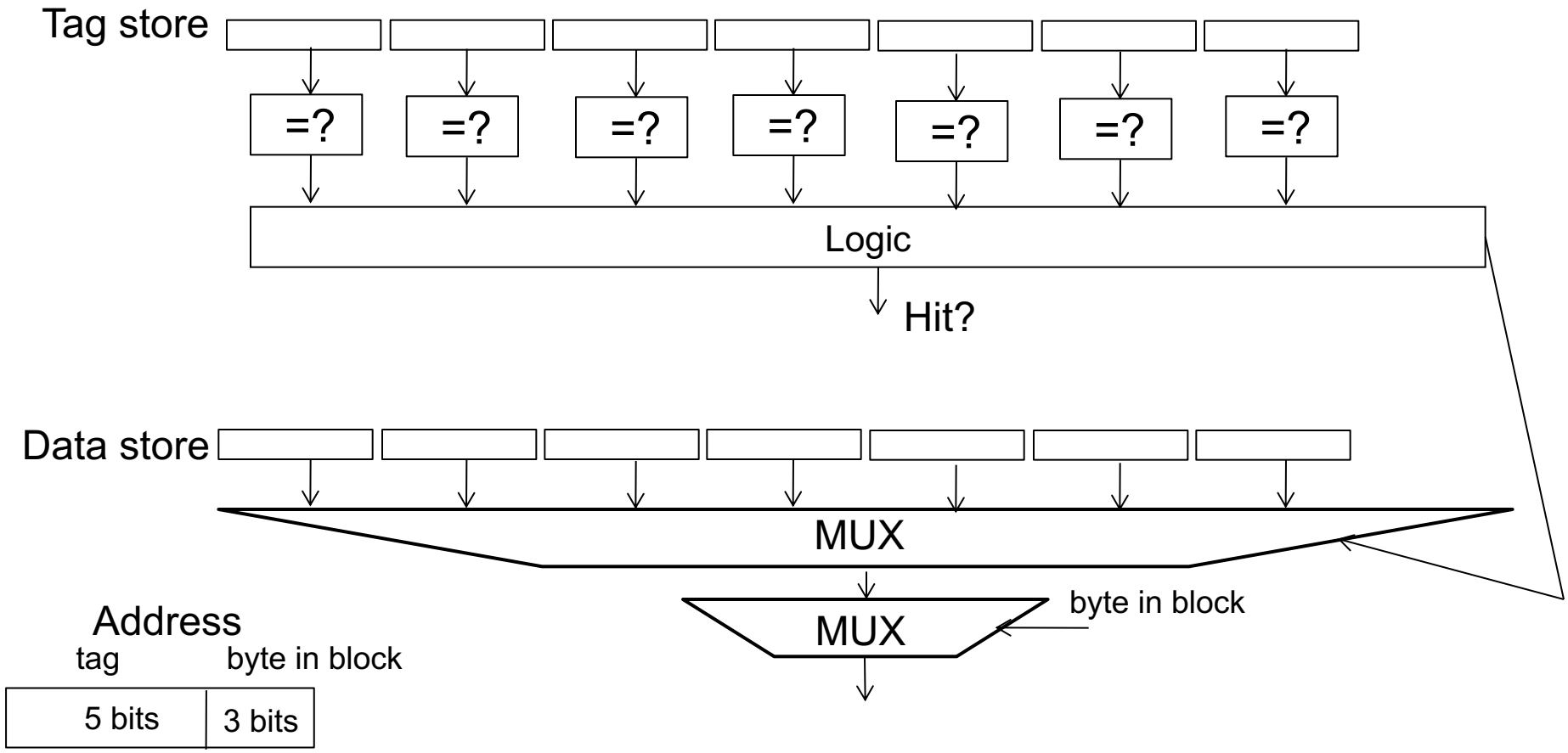
Address

tag	index	byte in block
4 bits	1 b	3 bits

Recall: 8-way Fully Associative Cache



7-way Fully Associative Cache



Classification of Cache Misses

- Compulsory miss
  - first reference to an address (block) always results in a miss
  - subsequent references to the block should hit in cache unless the block is displaced from cache for the reasons below
- Capacity miss
  - cache is too small to hold all needed data
  - defined as the misses that would occur even in a fully-associative cache (with optimal replacement) of the same capacity
- Conflict miss
  - defined as any miss that is neither a compulsory nor a capacity miss

How to Reduce Each Miss Type

- Compulsory
  - Caching (only accessed data) cannot help; larger blocks can
  - Prefetching helps: Anticipate which blocks will be needed soon
- Conflict
  - More associativity
  - Other ways to get more associativity without making the cache associative
    - Victim cache
    - Better, randomized indexing into the cache
    - Software hints for eviction/replacement/promotion
- Capacity
  - Utilize cache space better: keep blocks that will be referenced
  - Software management: divide working set and computation such that each "computation phase" fits in cache

How to Improve Cache Performance

- Three fundamental goals
- Reducing miss rate
  - Caveat: reducing miss rate can reduce performance if more costly-to-refetch blocks are evicted
- Reducing miss latency or miss cost
- Reducing hit latency or hit cost
- The above three **together** affect performance

Improving Basic Cache Performance

- Reducing miss rate
  - More associativity
  - Alternatives/enhancements to associativity
    - Victim caches, hashing, pseudo-associativity, skewed associativity
  - Better replacement/insertion policies
  - Software approaches
- Reducing miss latency/cost
  - Multi-level caches
  - Critical word first
  - Subblocking/sectoring
  - Better replacement/insertion policies
  - Non-blocking caches (multiple cache misses in parallel)
  - Multiple accesses per cycle
  - Software approaches

Software Approaches for Higher Hit Rate

- Restructuring data access patterns
- Restructuring data layout

- Loop interchange
- Data structure separation/merging
- Blocking
- …

Restructuring Data Access Patterns (I)

- Idea: Restructure data layout or data access patterns
- Example: If column-major
  - x[i+1,j] follows x[i,j] in memory
  - x[i,j+1] is far away from x[i,j]

Poor code

```
for i = 1, rows
    for j = 1, columns
        sum = sum + x[i,j]
```

Better code

```
for j = 1, columns
    for i = 1, rows
        sum = sum + x[i,j]
```

- This is called loop interchange
- Other optimizations can also increase hit rate
  - Loop fusion, array merging, …

Restructuring Data Access Patterns (II)

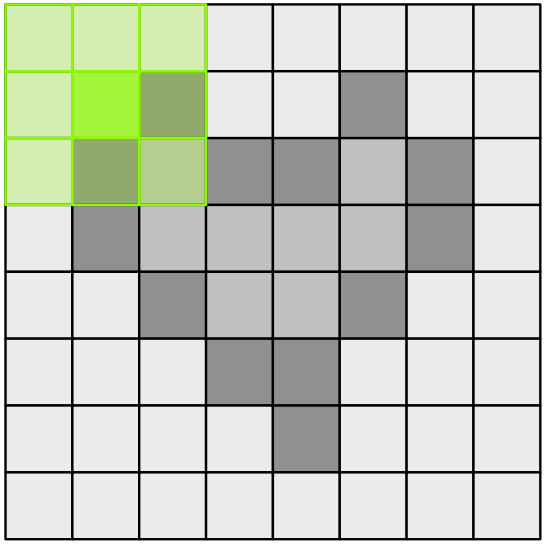
- Blocking
  - Divide loops operating on arrays into computation chunks so that each chunk can hold its data in the cache
  - Avoids cache conflicts between different chunks of computation
  - Essentially: Divide the working set so that each piece fits in the cache

- Also called Tiling

Data Reuse: An Example from GPU Computing

- Same memory locations accessed by neighboring threads

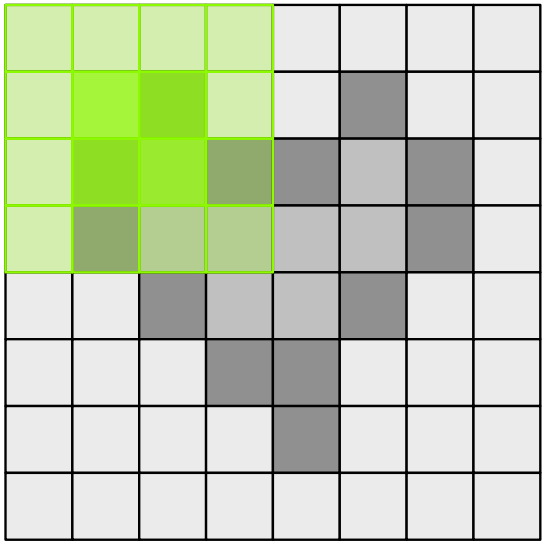
Gaussian filter applied on every pixel of an image



```
for (int i = 0; i < 3; i++){
    for (int j = 0; j < 3; j++){
        sum += gauss[i][j] * Image[(i+row-1)*width + (j+col-1)];
    }
}
```

Data Reuse: Tiling in GPU Computing

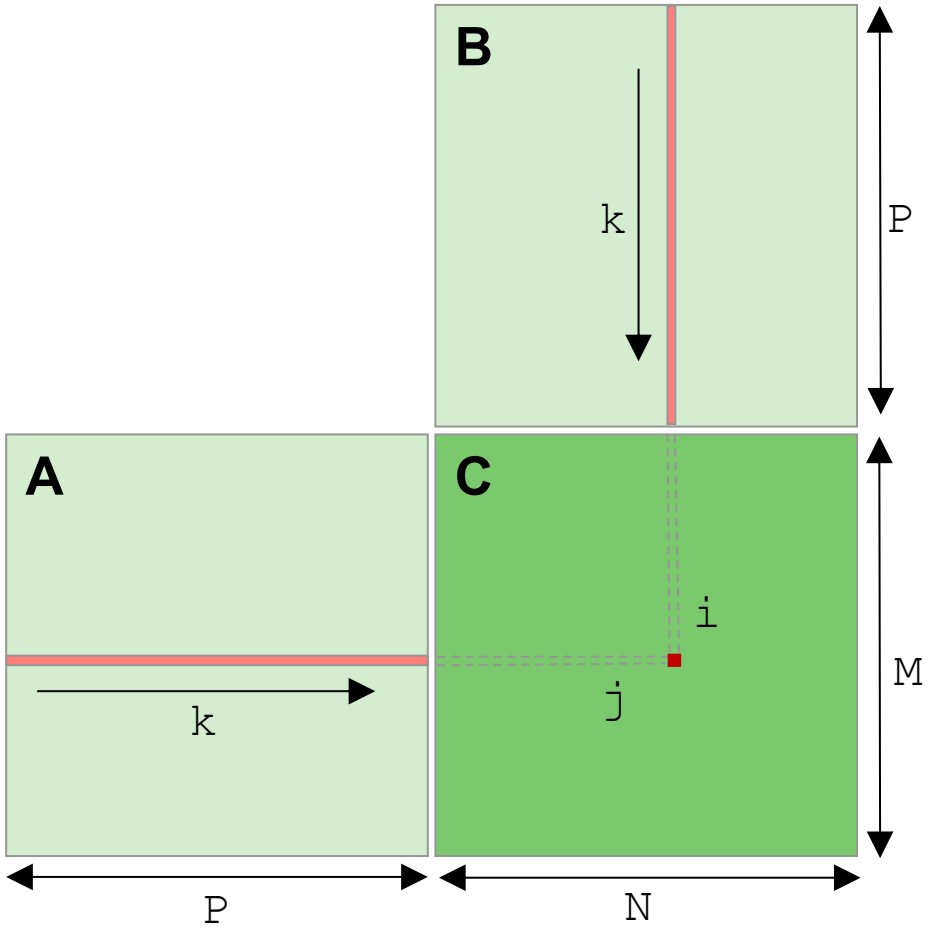
- To take advantage of data reuse, we divide the input into tiles that can be loaded into shared memory (scratchpad memory)



```
__shared__ int l_data[(L_SIZE+2)*(L_SIZE+2)];
…
Load tile into shared memory
__syncthreads();
for (int i = 0; i < 3; i++){
  for (int j = 0; j < 3; j++){
    sum += gauss[i][j] * l_data[(i+l_row-1)*(L_SIZE+2)+j+l_col-1];
  }
}
```

Naïve Matrix Multiplication (I)

- Matrix multiplication: C = A x B
- Consider two input matrices A and B in row-major layout
  - A size is M x P
  - B size is P x N
  - C size is M x N

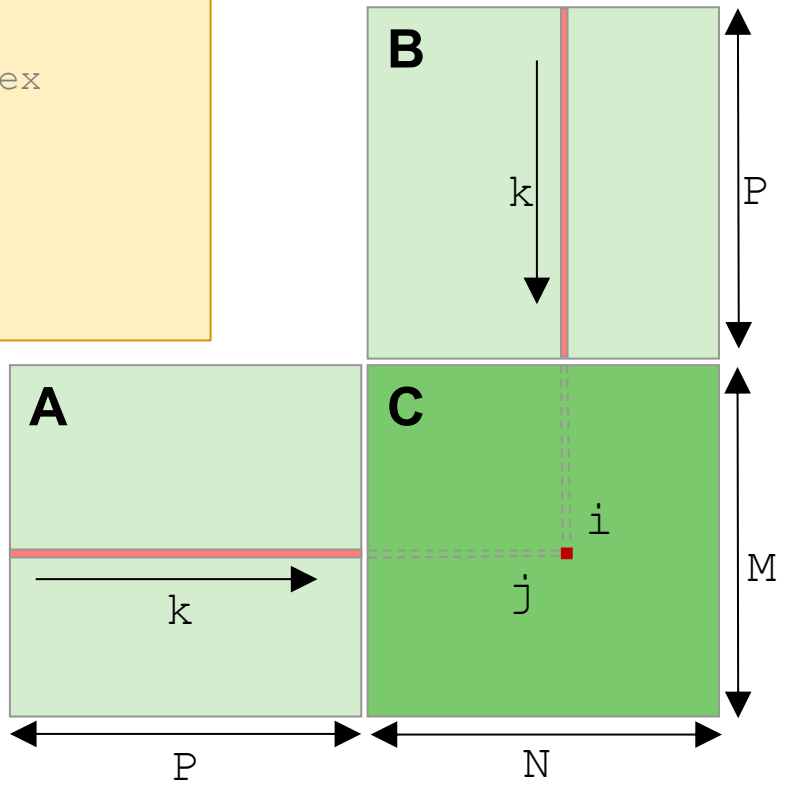


Naïve Matrix Multiplication (II)

- Naïve implementation of matrix multiplication has poor cache locality

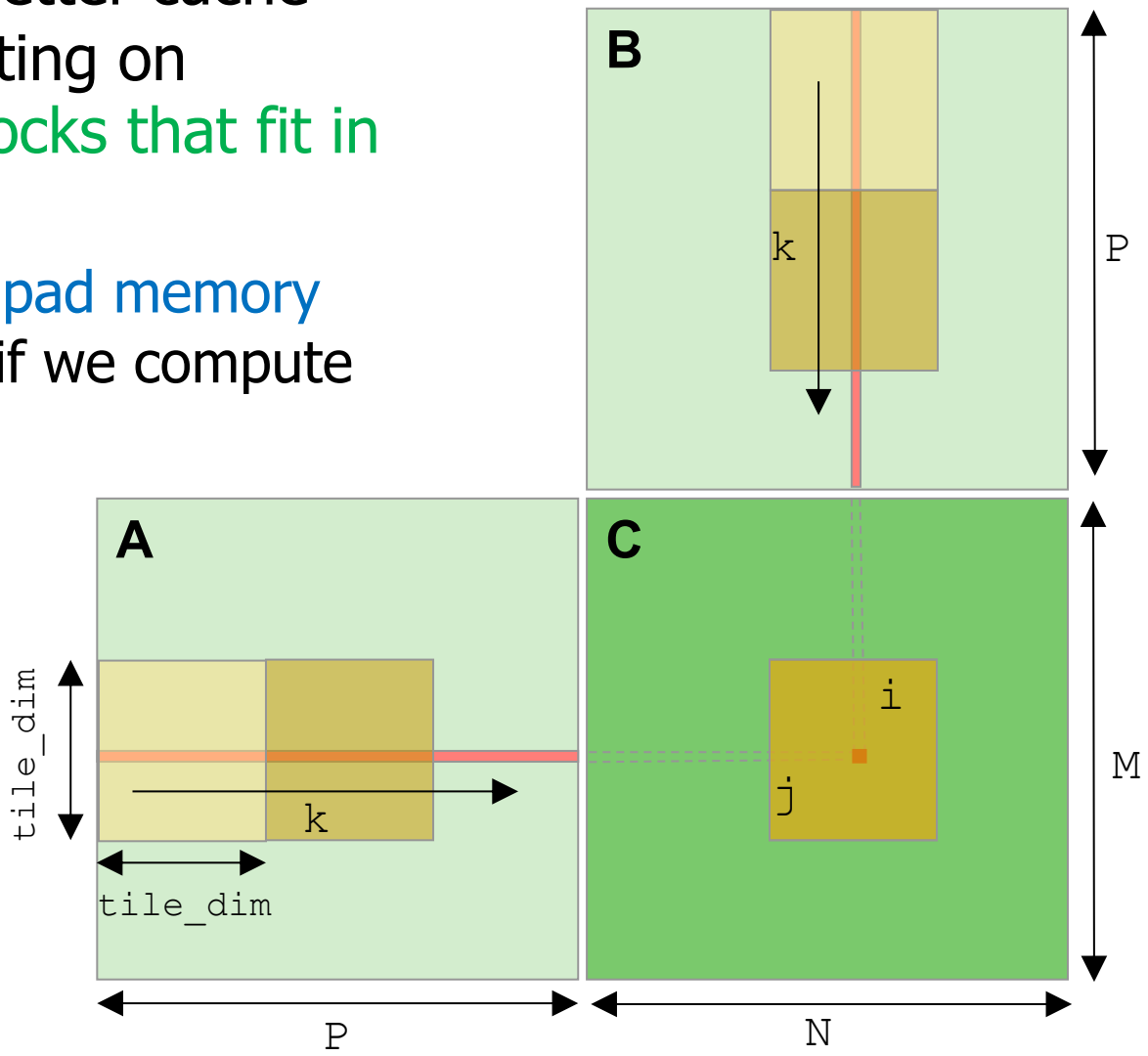
```
#define A(i,j) matrix_A[i * P + j]
#define B(i,j) matrix_B[i * N + j]
#define C(i,j) matrix_C[i * N + j]

for (i = 0; i < M; i++){ // i = row index
    for (j = 0; j < N; j++){ // j = column index
        C(i, j) = 0; // Set to zero
        for (k = 0; k < P; k++) // Row x Col
            C(i, j) += A(i, k) * B(k, j);
    }
}
```

Consecutive accesses to B are far from each other, in different cache lines.
Every access to B is likely to cause a cache miss



Tiled Matrix Multiplication (I)

- We can achieve better cache locality by computing on smaller tiles or blocks that fit in the cache
  - Or in the scratchpad memory and register file if we compute on a GPU



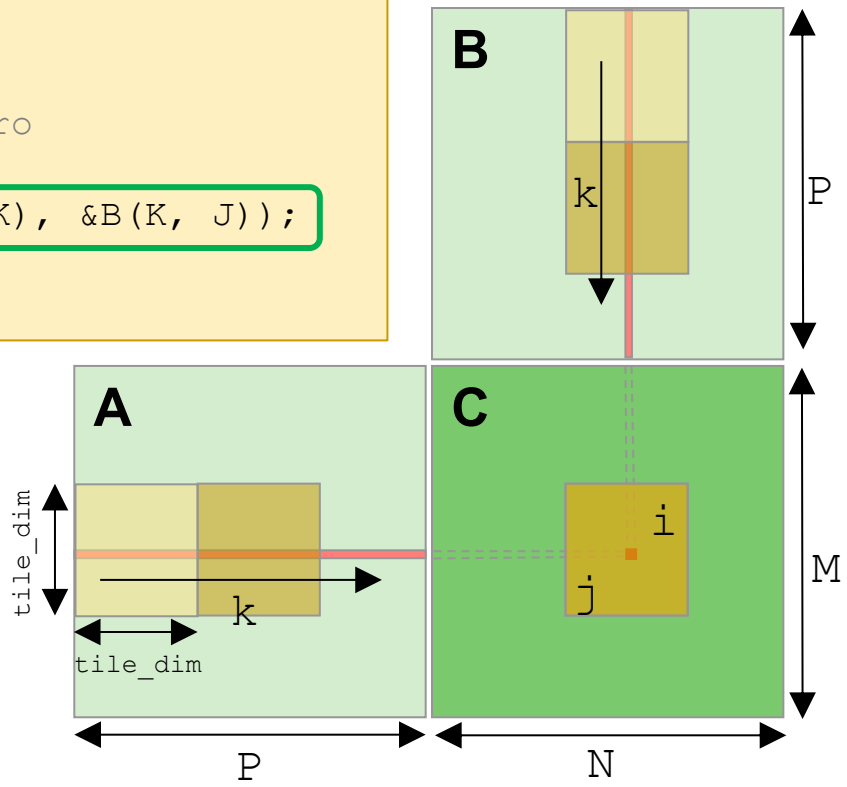
Lam+, "The cache performance and optimizations of blocked algorithms," ASPLOS 1991. https://doi.org/10.1145/106972.106981
Bansal+, "Chapter 15 - Fast Matrix Computations on Heterogeneous Streams," in "High Performance Parallelism Pearls", 2015. https://doi.org/10.1016/B978-0-12-803819-2.00011-2
Kirk & Hwu, "Chapter 5 - Performance considerations," in "Programming Massively Parallel Processors (Third Edition)", 2017. https://doi.org/10.1016/B978-0-12-811986-0.00005-4

Tiled Matrix Multiplication (II)

- Tiled implementation operates on submatrices (tiles or blocks) that fit fast memories (cache, scratchpad, RF)

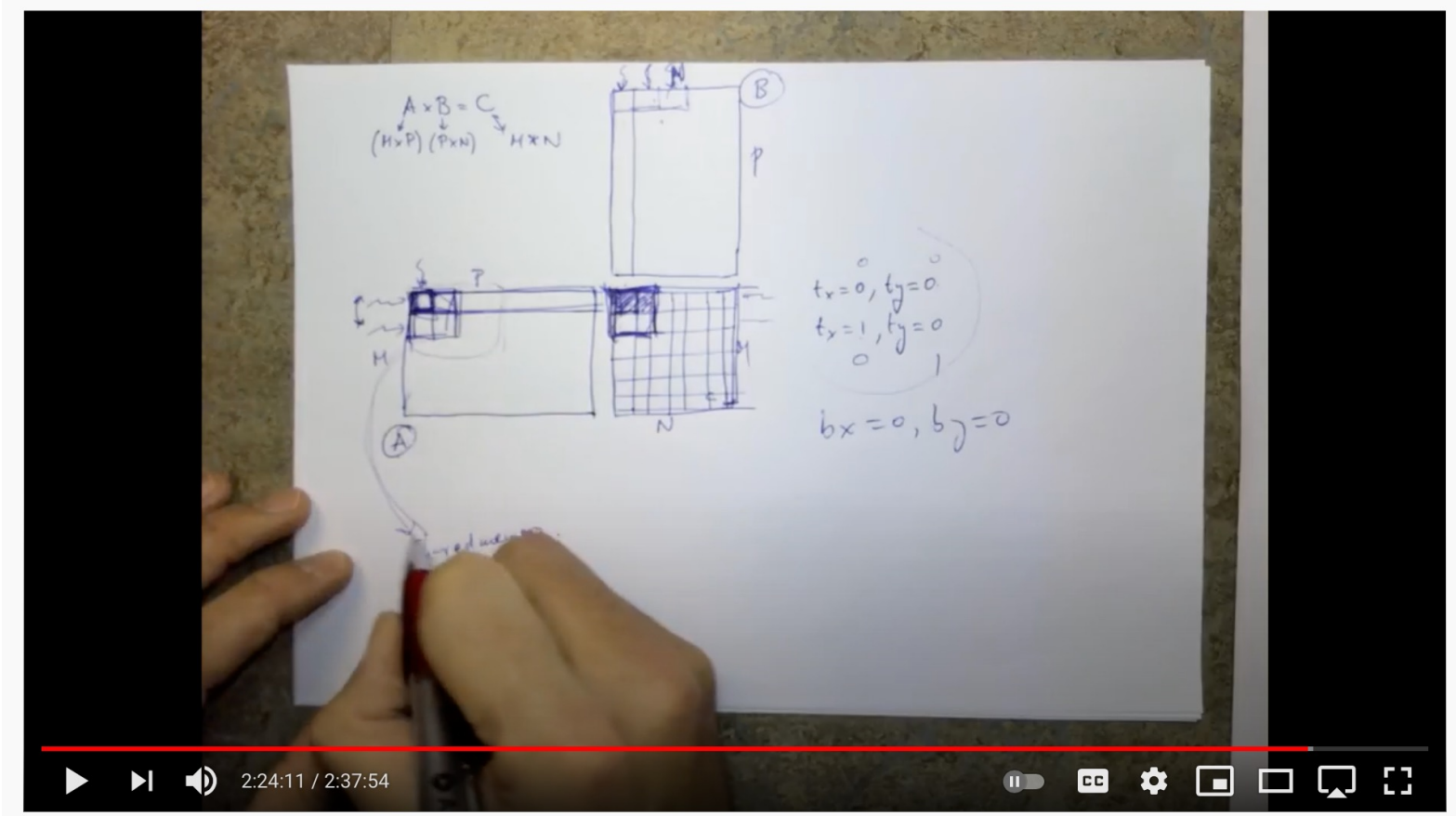
```
#define A(i,j) matrix_A[i * P + j]
#define B(i,j) matrix_B[i * N + j]
#define C(i,j) matrix_C[i * N + j]

for (I = 0; I < M; I += tile_dim){
    for (J = 0; J < N; J += tile_dim){
        Set_to_zero(&C(I, J)); // Set to zero
        for (K = 0; K < P; K += tile_dim)
            Multiply_tiles(&C(I, J), &A(I, K), &B(K, J));
    }
}
```

Multiply small submatrices (tiles or blocks) of size `tile_dim x tile_dim`



Lam+, "The cache performance and optimizations of blocked algorithms," ASPLOS 1991. https://doi.org/10.1145/106972.106981
Bansal+, "Chapter 15 - Fast Matrix Computations on Heterogeneous Streams," in "High Performance Parallelism Pearls", 2015. https://doi.org/10.1016/B978-0-12-803819-2.00011-2
Kirk & Hwu, "Chapter 5 - Performance considerations," in "Programming Massively Parallel Processors (Third Edition)", 2017. https://doi.org/10.1016/B978-0-12-811986-0.00005-4

Tiled Matrix Multiplication on GPUs



Computer Architecture - Lecture 9: GPUs and GPGPU Programming (ETH Zürich, Fall 2017)

14,426 views • Oct 23, 2017 225 2 SHARE SAVE

Onur Mutlu Lectures
16.5K subscribers

SUBSCRIBED

Restructuring Data Layout (I)

```
struct Node {
    struct Node* next;
    int key;
    char [256] name;
    char [256] school;
}

while (node) {
    if (node→key == input-key) {
        // access other fields of node
    }
    node = node→next;
}
```

Frequently accessed

Rarely accessed

Rarely accessed

Frequently accessed

- Assume pointer-based traversal (e.g., of a linked list)
- Assume a huge linked list (1B nodes) and unique keys
- Does the code on the left have poor or good cache hit rate?
- Poor hit rate: Why?
  - "Other fields" occupy most of the cache block even though they are rarely accessed!

Restructuring Data Layout (II)

```
struct Node {
    struct Node* next;
    int key;
    struct Node-data* node-data;
}
```

```
struct Node-data {
    char [256] name;
    char [256] school;
}
```

```
while (node) {
    if (node→key == input-key) {
        // access node→node-data
    }
    node = node→next;
}
```

- Idea: Separate rarely-accessed fields of a data structure and pack them into a separate data structure
- General idea: Separate frequently-accessed (hot) data from rarely-accessed (cold) data so that they are not in the same cache block
- Who should do this?
  - Programmer
  - Compiler
    - Profiling vs. dynamic
  - Hardware?
  - Who can determine what is frequently accessed?

Improving Basic Cache Performance

- Reducing miss rate
  - More associativity
  - Alternatives/enhancements to associativity
    - Victim caches, hashing, pseudo-associativity, skewed associativity
  - Better replacement/insertion policies
  - Software approaches
- Reducing miss latency/cost
  - Multi-level caches
  - Critical word first
  - Subblocking/sectoring
  - Better replacement/insertion policies
  - Non-blocking caches (multiple cache misses in parallel)
  - Multiple accesses per cycle
  - Software approaches

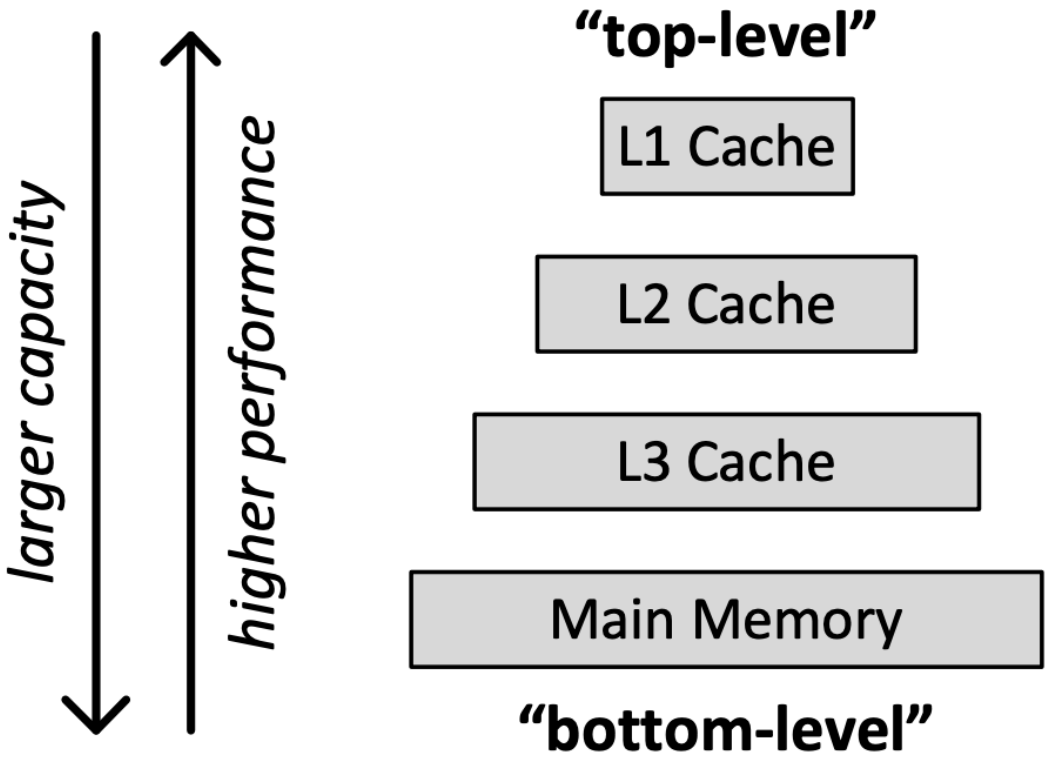
Readings for This Week and Last Week

- Memory Hierarchy and Caches

- Required
  - H&H Chapters 8.1-8.3
  - Refresh: P&P Chapter 3.5
  - Kim & Mutlu, "Memory Systems," Computing Handbook, 2014.
    - https://people.inf.ethz.ch/omutlu/pub/memory-systems-introduction_computing-handbook14.pdf

- Recommended
  - An early cache paper by Maurice Wilkes
    - Wilkes, "Slave Memories and Dynamic Storage Allocation," IEEE Trans. On Electronic Computers, 1965.

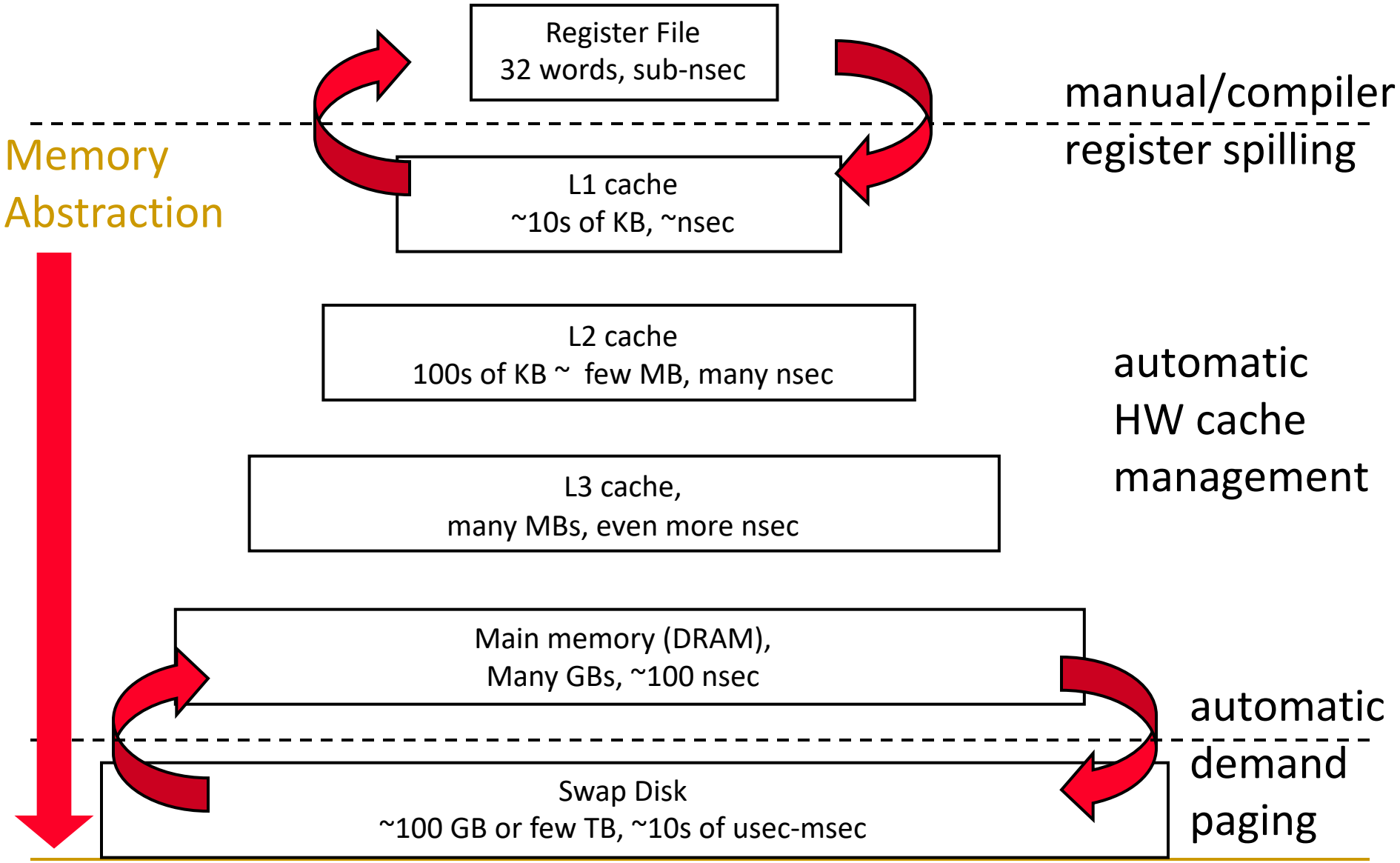
Recall: Memory Hierarchy Example



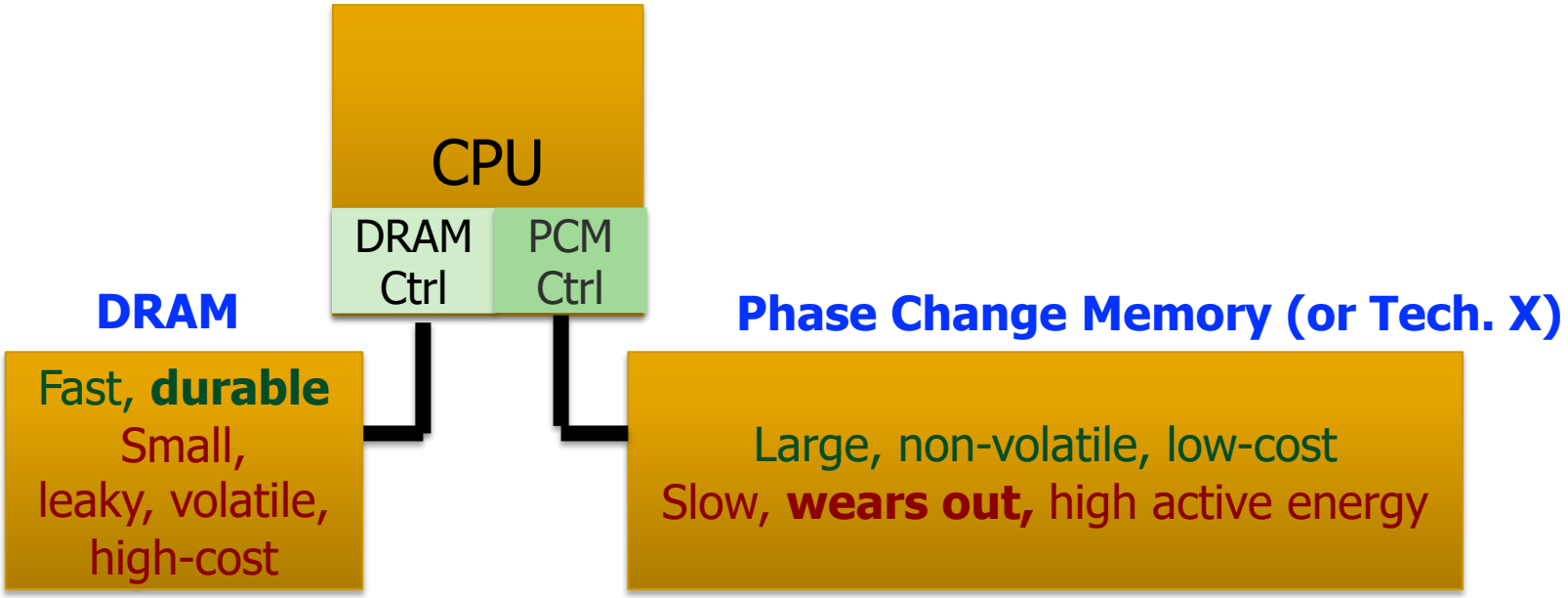
capacity	latency
10's of KB	≈ 1ns
100's of KB	< 5ns
several MB	≈ 10ns
several GB	≈ 100ns

Kim & Mutlu, "Memory Systems," Computing Handbook, 2014
https://people.inf.ethz.ch/omutlu/pub/memory-systems-introduction_computing-handbook14.pdf

Recall: A Modern Memory Hierarchy



Hybrid Main Memory Extends the Hierarchy



Hardware/software manage data allocation & movement
to achieve the best of multiple technologies

Meza+, "Enabling Efficient and Scalable Hybrid Memories," IEEE Comp. Arch. Letters, 2012.

Yoon+, "Row Buffer Locality Aware Caching Policies for Hybrid Memories," ICCD 2012 Best Paper Award.

Recall: Issues in Set-Associative Caches

- Think of each block in a set having a "priority"
  - Indicating how important it is to keep the block in the cache
- Key issue: How do you determine/adjust block priorities?
- There are three key decisions in a set:
  - Insertion, promotion, eviction (replacement)

- **Insertion**: What happens to priorities on a cache fill?
  - Where to insert the incoming block; whether or not to insert the block
- **Promotion**: What happens to priorities on a cache hit?
  - Whether and how to change block priority
- **Eviction/replacement**: What happens to priorities on a cache miss?
  - Which block to evict and how to adjust priorities

Recall: Multi-Level Cache Design Decisions

- Which level(s) to place a block into (from memory)?
- Which level(s) to evict a block to (from an inner level)?
- Bypassing vs. non-bypassing levels
- Inclusive, exclusive, non-inclusive hierarchies
  - Inclusive: a block in an inner level is always included also in an outer level → simplifies cache coherence
  - Exclusive: a block in an inner level does not exist in an outer level → better utilizes space in the entire hierarchy
  - Non-inclusive: a block in an inner level may or may not be included in an outer level → relaxes design decisions

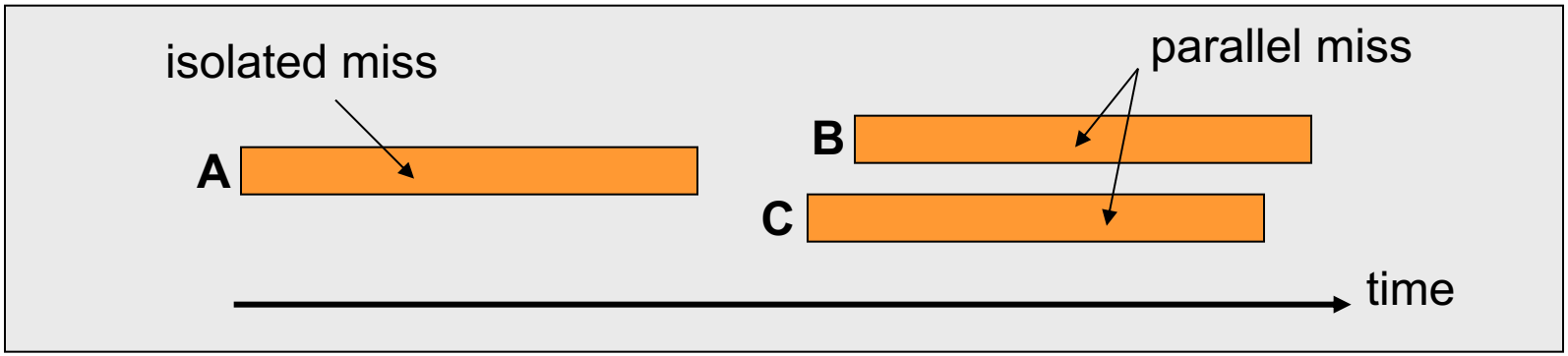
Improving Basic Cache Performance

- Reducing miss rate
  - More associativity
  - Alternatives/enhancements to associativity
    - Victim caches, hashing, pseudo-associativity, skewed associativity
  - Better replacement/insertion policies
  - Software approaches
- Reducing miss latency/cost
  - Multi-level caches
  - Critical word first
  - Subblocking/sectoring
  - Better replacement/insertion policies
  - Non-blocking caches (multiple cache misses in parallel)
  - Multiple accesses per cycle
  - Software approaches

Miss Latency/Cost

- What is miss latency or miss cost affected by?
  - Where does the miss get serviced from?
    - What level of cache in the hierarchy?
    - Row hit versus row conflict in DRAM (bank/rank/channel conflict)
    - Queueing delays in the memory controller and the interconnect
    - Local vs. remote memory (chip, node, rack, remote server, …)
    - …
  - How much does the miss stall the processor?
    - Is it overlapped with other latencies?
    - Is the data immediately needed by the processor?
    - Is the incoming block going to evict a longer-to-refetch block?
    - …

Memory Level Parallelism (MLP)



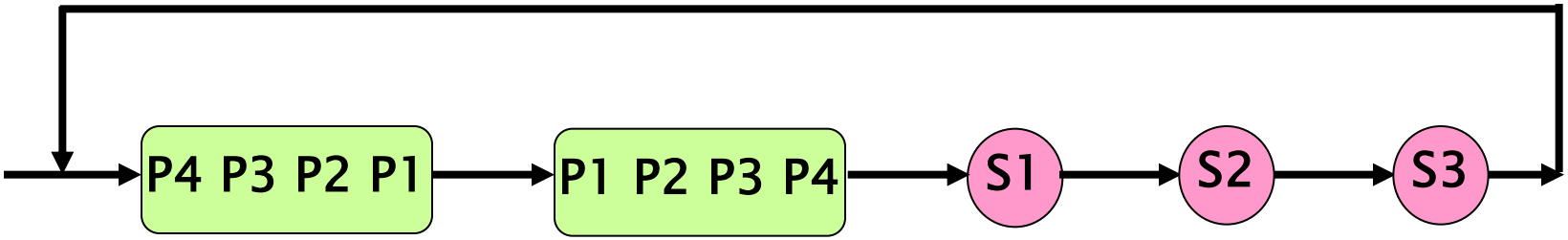
- Memory Level Parallelism (MLP) is the notion of generating and servicing multiple memory accesses in parallel [Glew'98]
- Several techniques improve MLP (e.g., out-of-order execution)
- MLP varies. Some misses are isolated and some parallel

How does this affect cache replacement?

Traditional Cache Replacement Policies

- Traditional cache management policies try to reduce miss count
- Implicit assumption: Reducing miss count reduces memory-related stall time
- Misses with varying cost/MLP breaks this assumption!
- Eliminating an isolated miss helps performance more than eliminating a parallel miss
- Eliminating a higher-latency miss could help performance more than eliminating a lower-latency miss

An Example



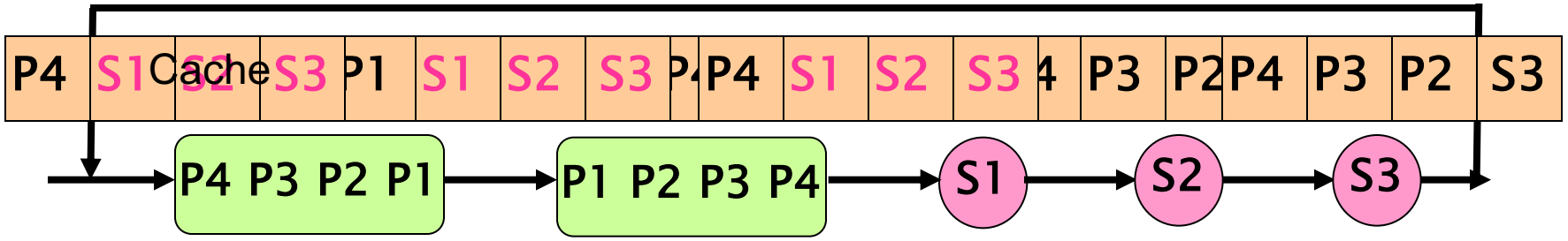
Misses to blocks P1, P2, P3, P4 can be serviced in parallel
Misses to blocks S1, S2, and S3 are isolated (i.e., serviced serially)

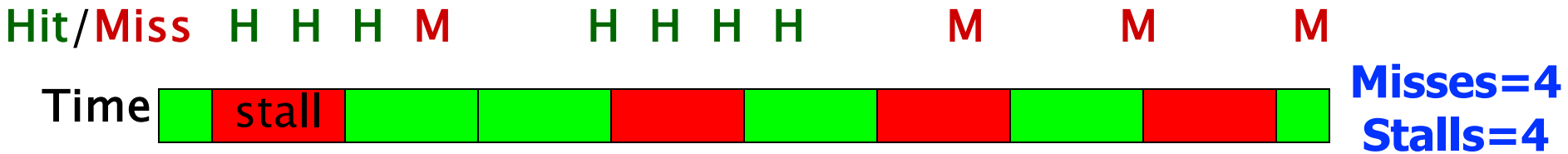
Two replacement algorithms:

1. Minimizes miss count (Belady's OPT)
2. Reduces isolated miss (MLP-Aware)

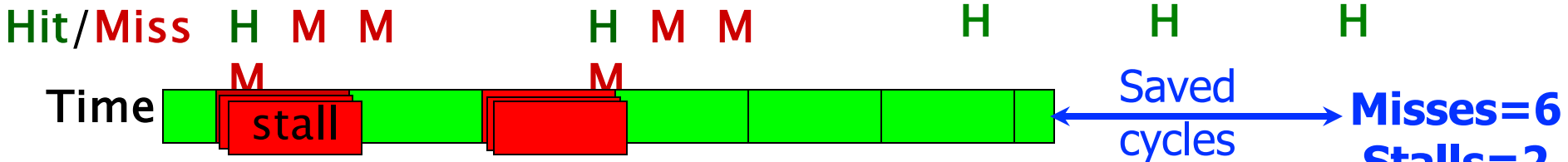
For a fully associative cache containing 4 blocks

Fewest Misses ≠ Best Performance





Belady's OPT replacement



MLP-Aware replacement

Recommended: MLP-Aware Cache Replacement

- How do we incorporate MLP/cost into replacement decisions?
- How do we design a hybrid cache replacement policy?

- Qureshi et al., "A Case for MLP-Aware Cache Replacement," ISCA 2006.

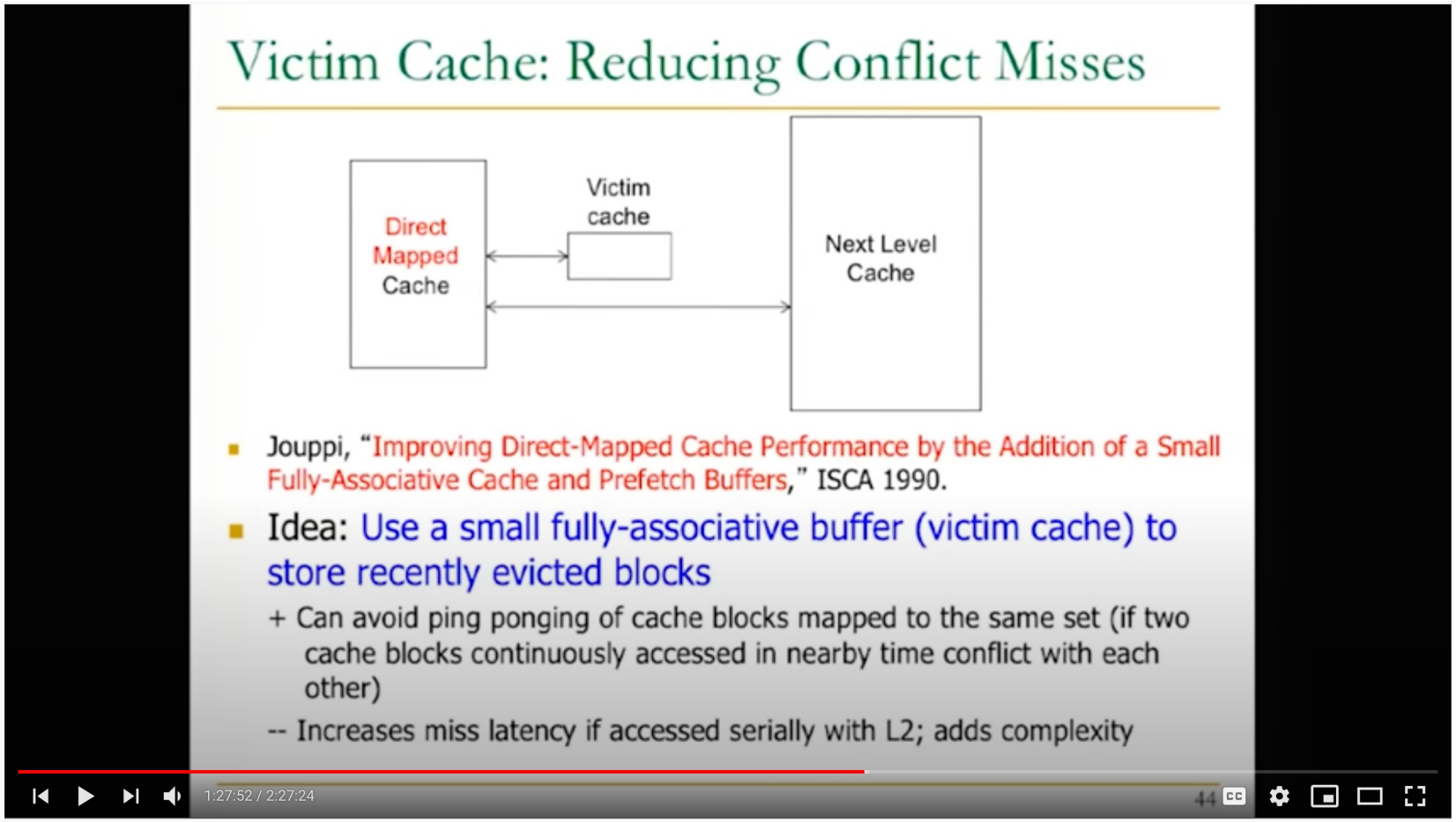
A Case for MLP-Aware Cache Replacement

Moinuddin K. Qureshi Daniel N. Lynch Onur Mutlu Yale N. Patt
Department of Electrical and Computer Engineering
The University of Texas at Austin
{*moin, lynch, onur, patt*}*@hps.utexas.edu*

Improving Basic Cache Performance

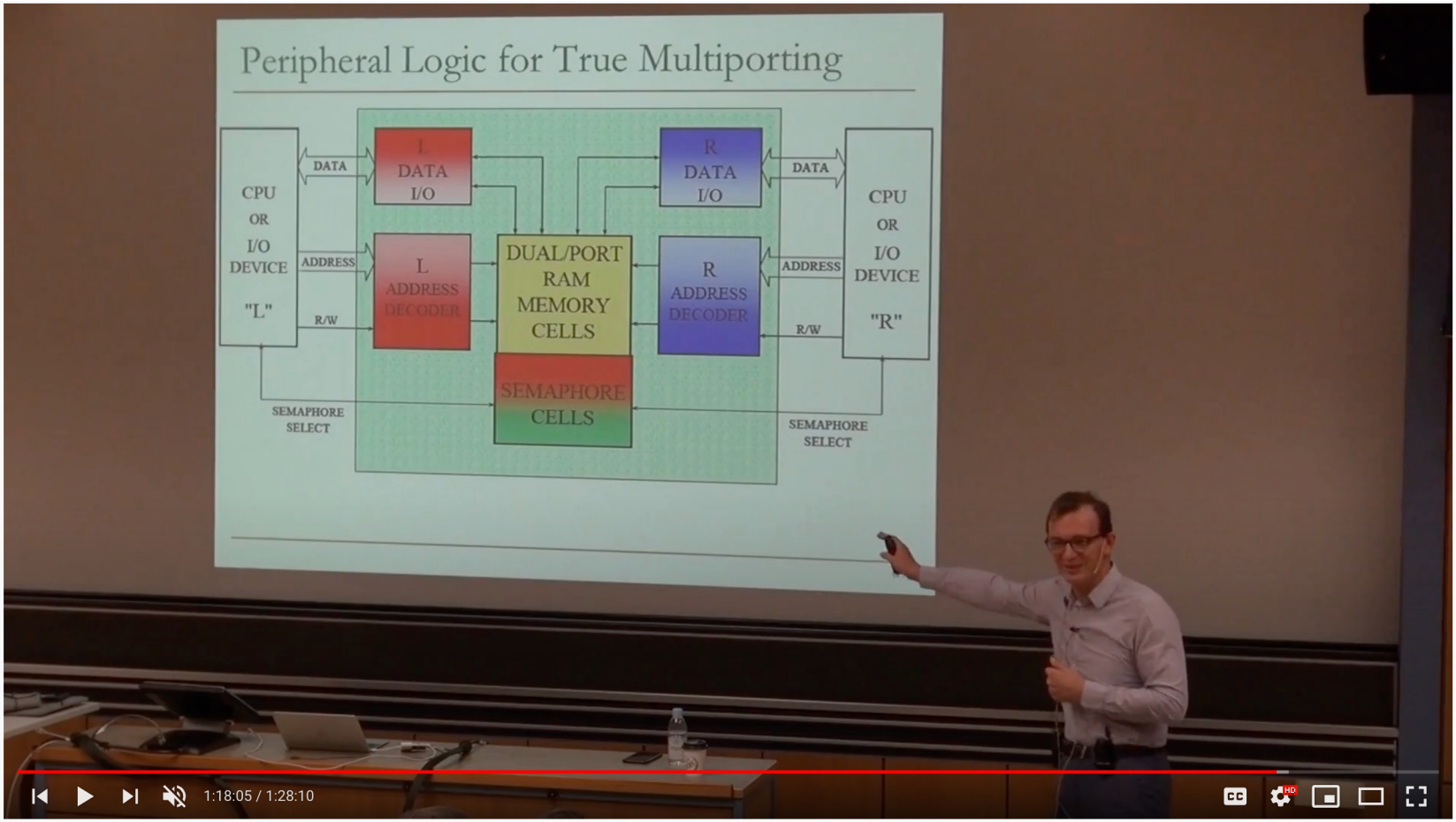
- Reducing miss rate
  - More associativity
  - Alternatives/enhancements to associativity
    - Victim caches, hashing, pseudo-associativity, skewed associativity
  - Better replacement/insertion policies
  - Software approaches
  - …
- Reducing miss latency/cost
  - Multi-level caches
  - Critical word first
  - Subblocking/sectoring
  - Better replacement/insertion policies
  - Non-blocking caches (multiple cache misses in parallel)
  - Multiple accesses per cycle
  - Software approaches
  - …

Lectures on Cache Optimizations (I)



https://www.youtube.com/watch?v=OyomXCHNJDA&list=PL5Q2soXY2Zi9OhoVQBXYFIZywZXCPl4M_&index=3

Lectures on Cache Optimizations (II)



ETH ZÜRICH HAUPTGEBÄUDE

Computer Architecture - Lecture 4a: Cache Design (ETH Zürich, Fall 2018)

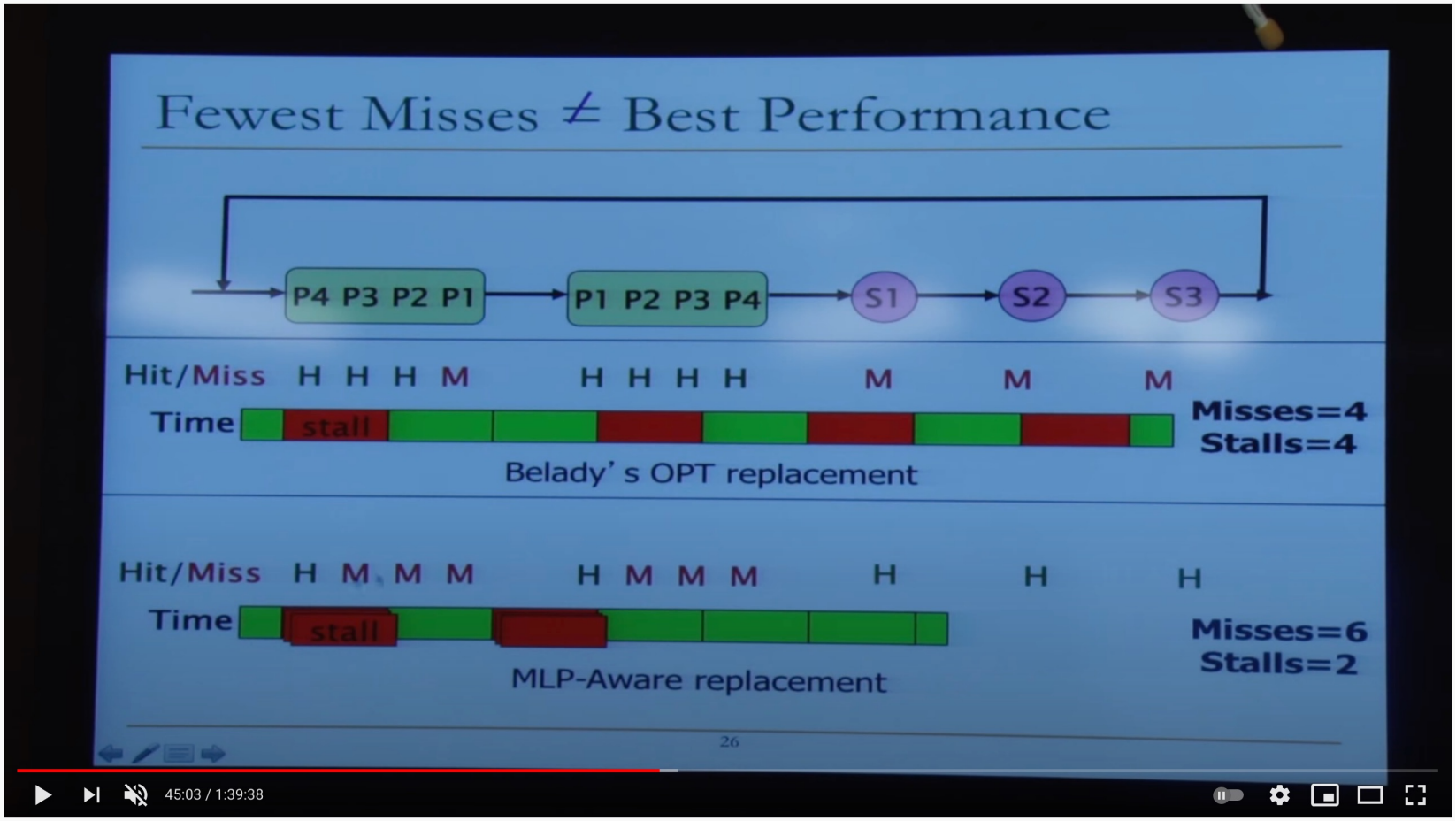
1,437 views • Sep 29, 2018 15 0 SHARE SAVE

Onur Mutlu Lectures
16.3K subscribers

ANALYTICS EDIT VIDEO

https://www.youtube.com/watch?v=55oYBm9cifI&list=PL5Q2soXY2Zi9JXe3ywQMhylk_d5dI-TM7&index=6

Lectures on Cache Optimizations (III)



Lecture 19. High Performance Caches - Carnegie Mellon - Comp. Arch. 2015 - Onur Mutlu

9,737 views • Mar 5, 2015

63 1 SHARE SAVE

Carnegie Mellon Computer Architecture
23.2K subscribers

ANALYTICS EDIT VIDEO

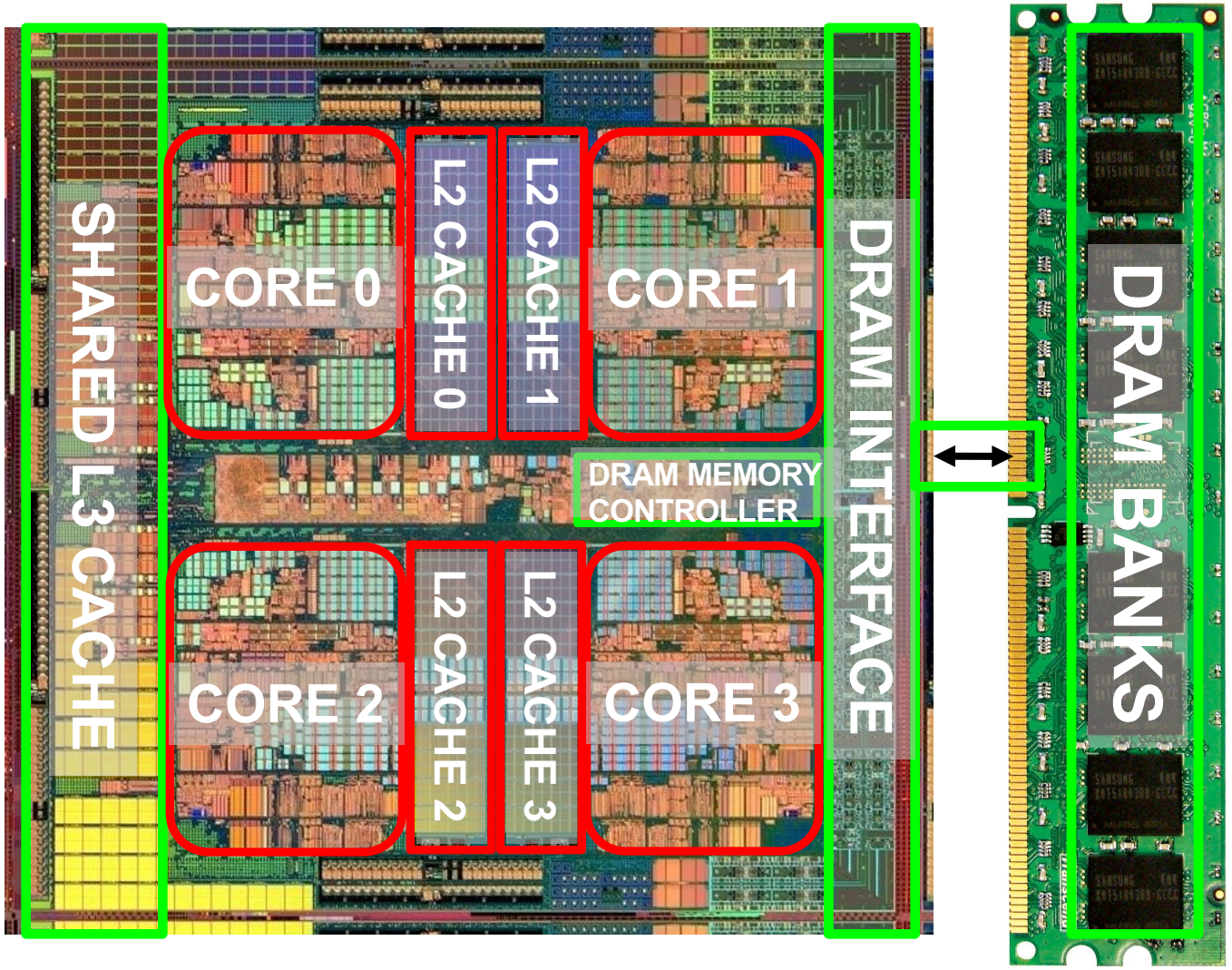
https://www.youtube.com/watch?v=jDHx2K9HxlM&list=PL5PHm2jkkXmi5CxxI7b3JCL1TWybTDtKq&index=21

Lectures on Cache Optimizations

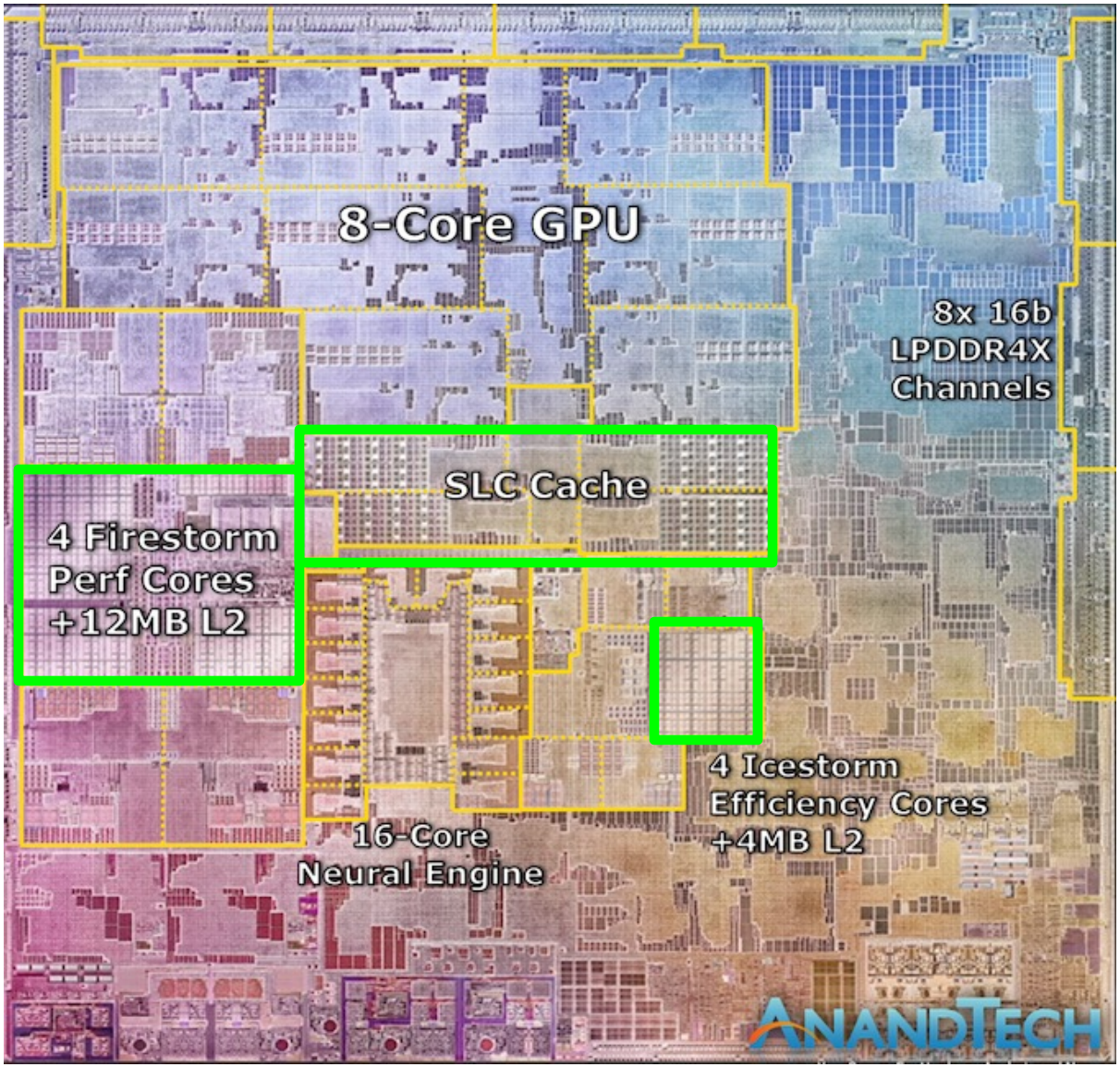
- Computer Architecture, Fall 2017, Lecture 3
  - Cache Management & Memory Parallelism (ETH, Fall 2017)
  - https://www.youtube.com/watch?v=OyomXCHNJDA&list=PL5Q2soXY2Zi9OhoVQBXYFIZywZXCPl4M_&index=3
- Computer Architecture, Fall 2018, Lecture 4a
  - Cache Design (ETH, Fall 2018)
  - https://www.youtube.com/watch?v=55oYBm9cifI&list=PL5Q2soXY2Zi9JXe3ywQMhylk_d5dI-TM7&index=6
- Computer Architecture, Spring 2015, Lecture 19
  - High Performance Caches (CMU, Spring 2015)
  - https://www.youtube.com/watch?v=jDHx2K9HxlM&list=PL5PHm2jkkXmi5CxxI7b3JCL1TWybTDtKq&index=21

Multi-Core Issues in Caching

Caches in a Multi-Core System



Caches in a Multi-Core System



Apple M1, 2021

Source: https://www.anandtech.com/show/16252/mac-mini-apple-m1-tested

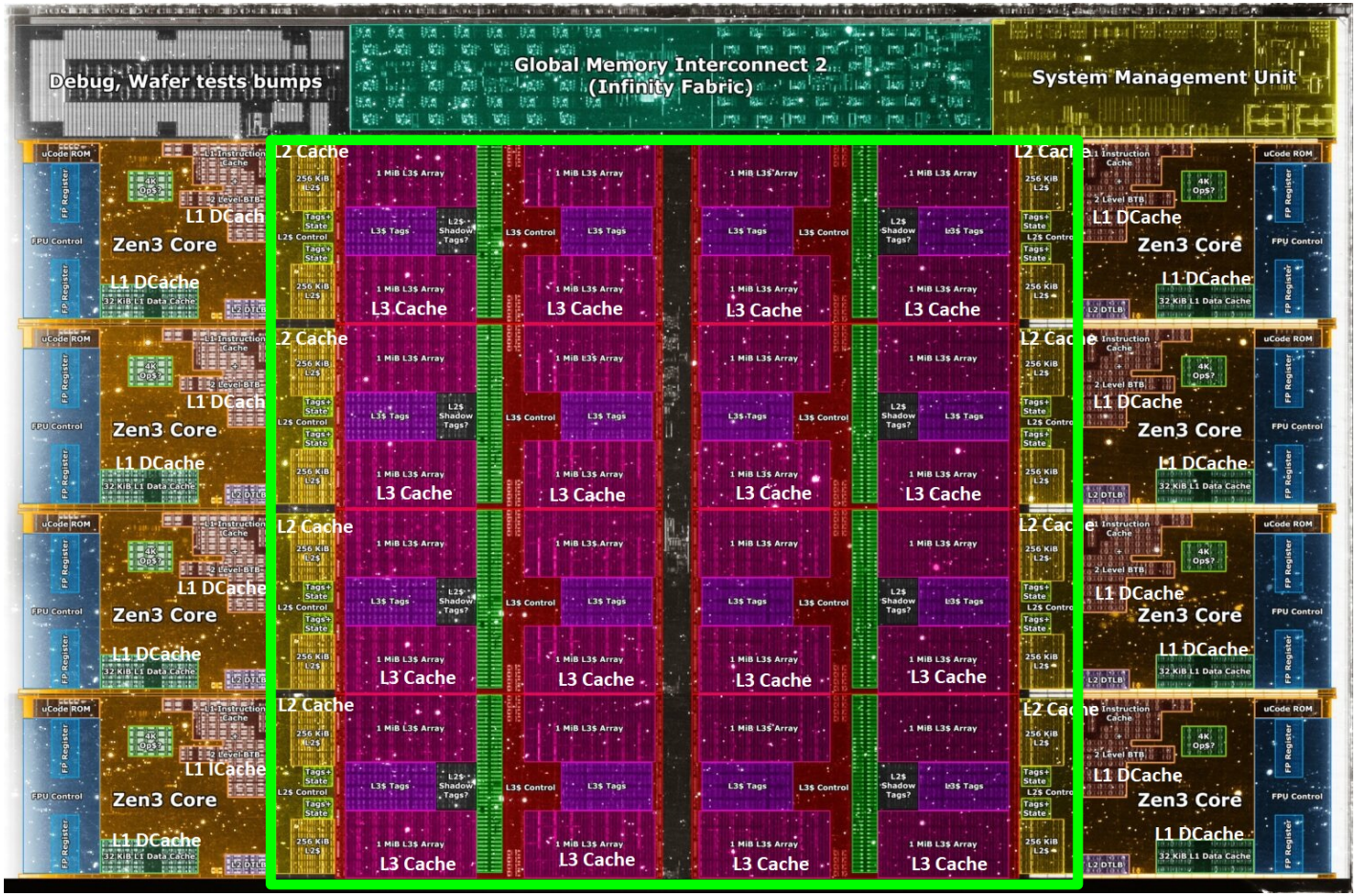
Caches in a Multi-Core System

10nm ESF=Intel 7 Alder Lake die shot (~209mm²) from Intel: https://www.intel.com/content/www/us/en/newsroom/news/12th-gen-core-processors.html
Die shot interpretation by Locuza, October 2021

Intel Alder Lake, 2021

Source: https://twitter.com/Locuza_/status/1454152714930331652

Caches in a Multi-Core System



AMD Ryzen 5000, 2020

Core Count:

8 cores/16 threads

L1 Caches:

32 KB per core

L2 Caches:

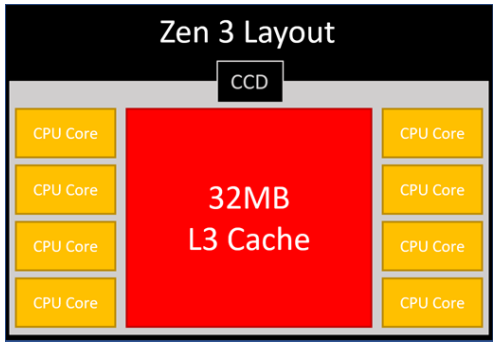
512 KB per core

L3 Cache:

32 MB shared

https://wccftech.com/amd-ryzen-5000-zen-3-vermeer-undressed-high-res-die-shots-close-ups-pictured-detailed/

Caches in a Multi-Core System

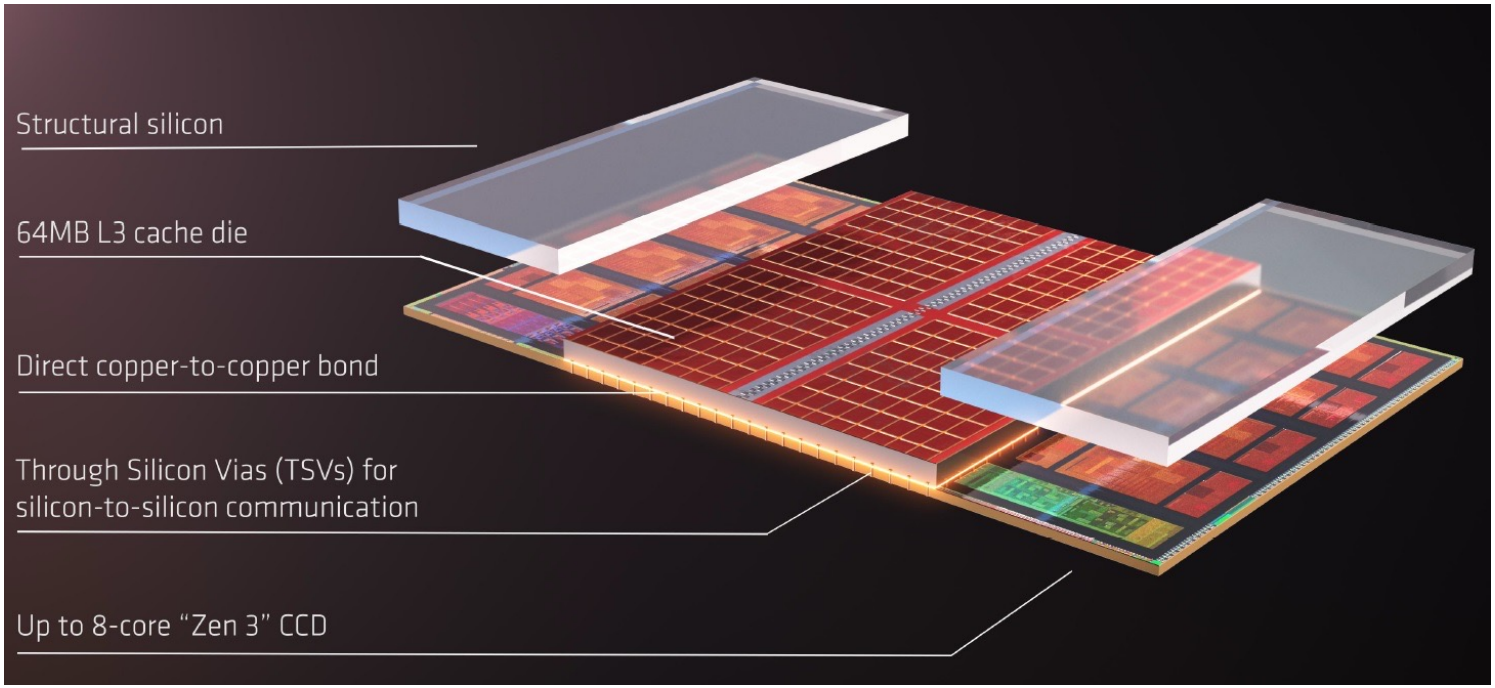


https://community.microcenter.com/discussion/5134/comparing-zen-3-to-zen-2

AMD increases the L3 size of their 8-core Zen 3 processors from 32 MB to 96 MB

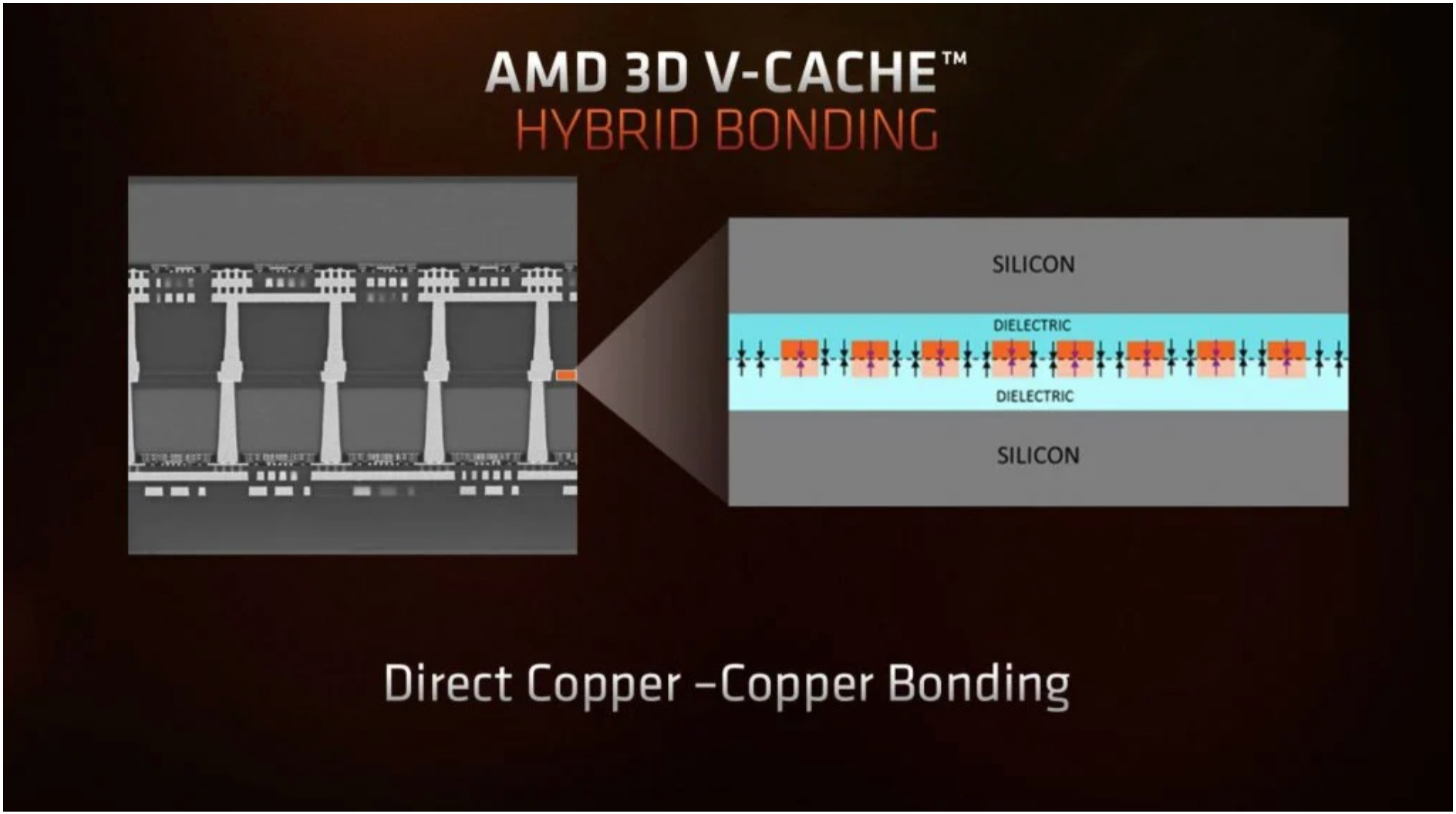
Additional 64 MB L3 cache die stacked on top of the processor die

- Connected using Through Silicon Vias (TSVs)
- Total of 96 MB L3 cache



https://youtu.be/gqAYMx34euU
https://www.tech-critter.com/amd-keynote-computex-2021/

3D Stacking Technology: Example

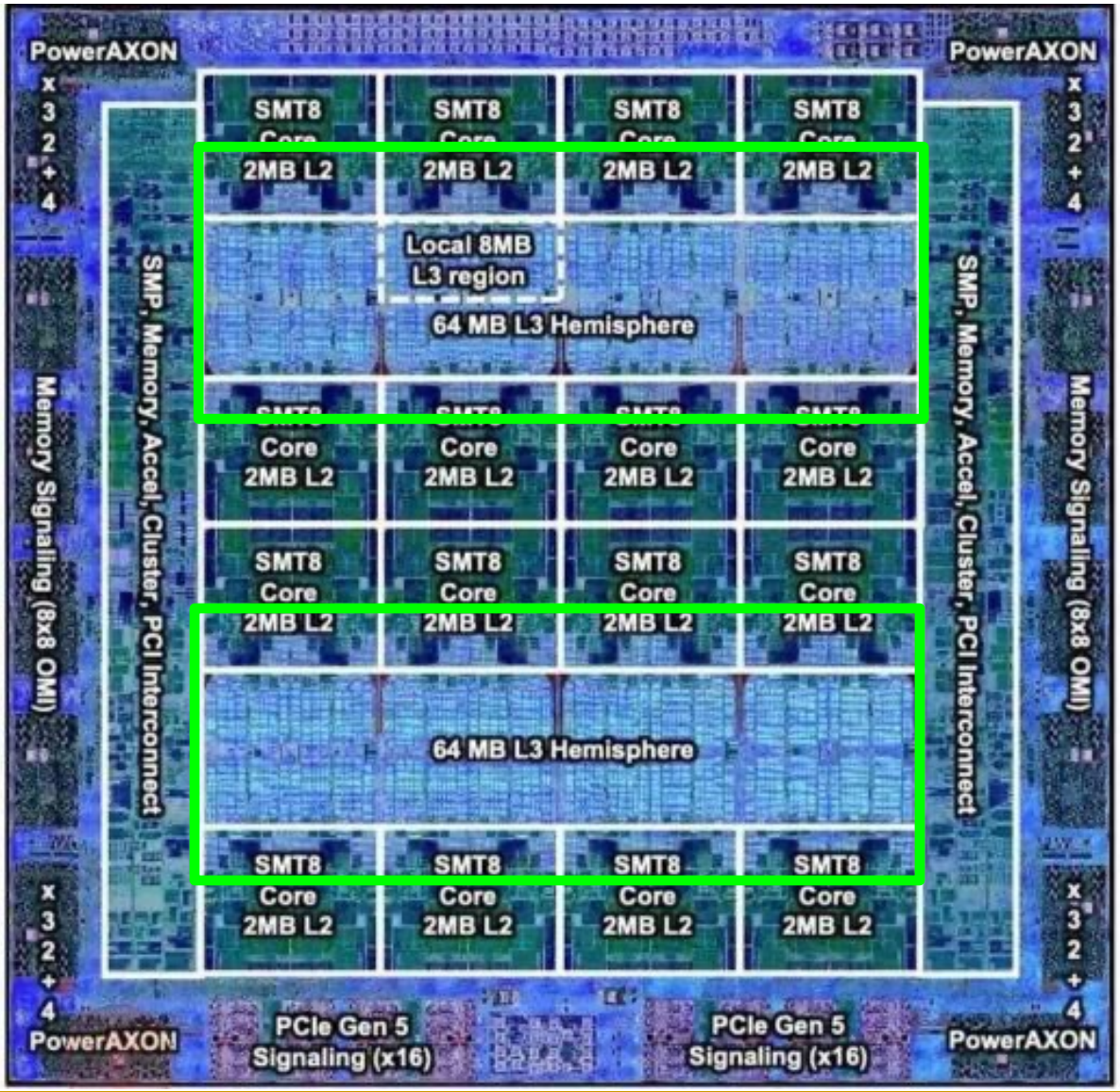


AMD Ryzen 7 5800X3D: The 3D V-Cache in detail (4)

Source: AMD

https://www.pcgameshardware.de/Ryzen-7-5800X3D-CPU-278064/Specials/3D-V-Cache-Release-1393125/

Caches in a Multi-Core System



IBM POWER10, 2020

Cores:

15-16 cores,
8 threads/core

L2 Caches:

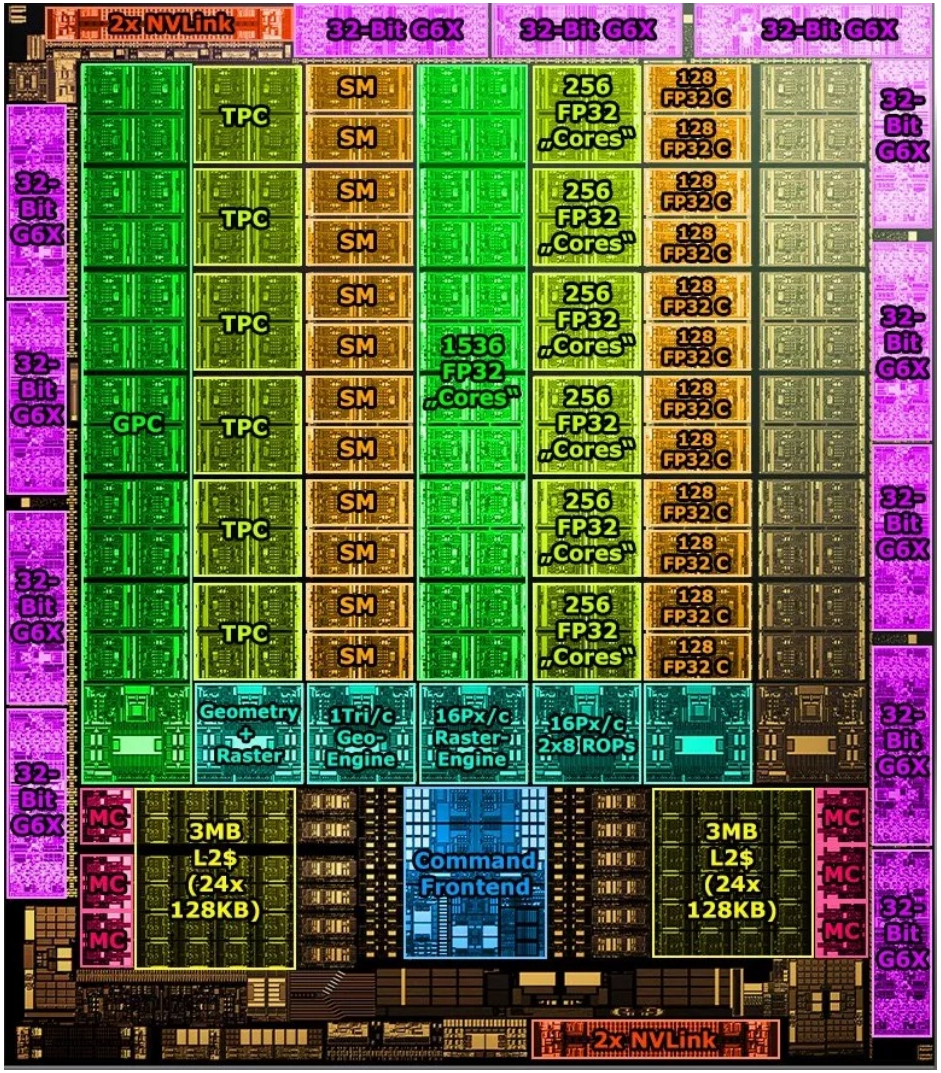
2 MB per core

L3 Cache:

120 MB shared

https://www.it-techblog.de/ibm-power10-prozessor-mehr-speicher-mehr-tempo-mehr-sicherheit/09/2020/

Caches in a Multi-Core System



Nvidia Ampere, 2020

Cores:

128 Streaming Multiprocessors

L1 Cache or Scratchpad:

192KB per SM

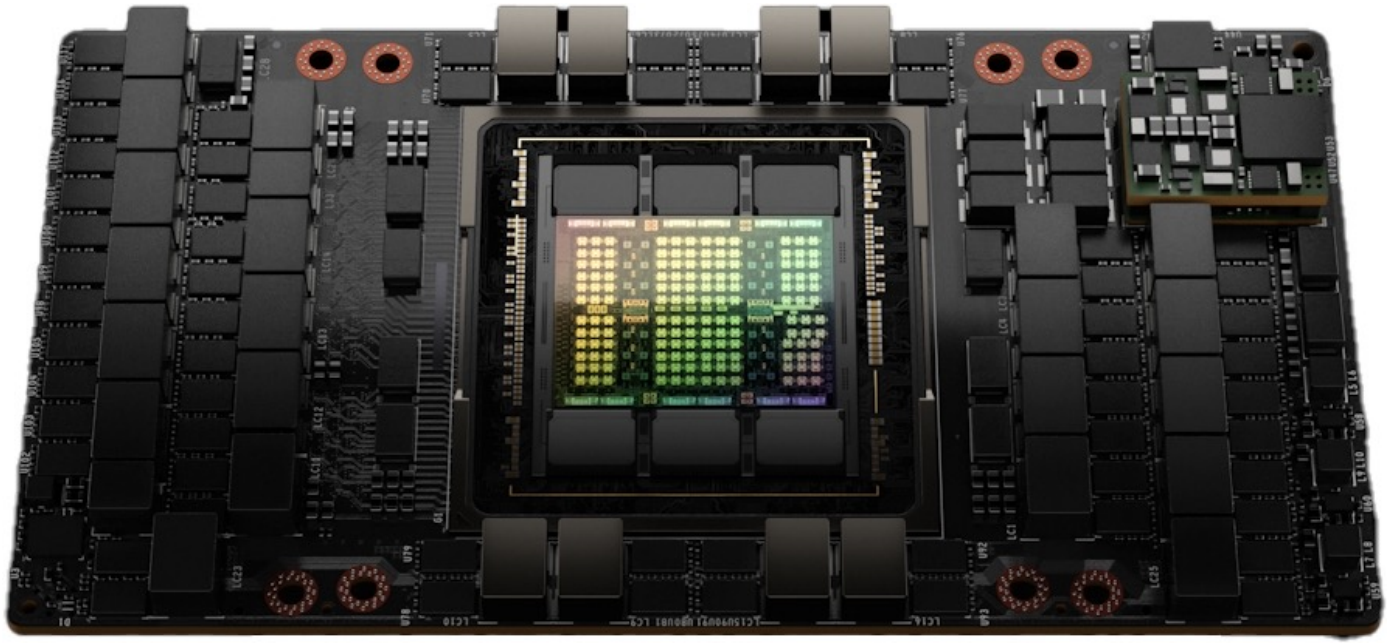
Can be used as L1 Cache and/or Scratchpad

L2 Cache:

40 MB shared

https://www.tomshardware.com/news/infrared-photographer-photos-nvidia-ga102-ampere-silicon

Caches in a Multi-Core System



Nvidia Hopper, 2022

Cores:

144 Streaming Multiprocessors

L1 Cache or Scratchpad:

256KB per SM

Can be used as L1 Cache and/or Scratchpad

L2 Cache:

60 MB shared

https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/

Caches in Multi-Core Systems

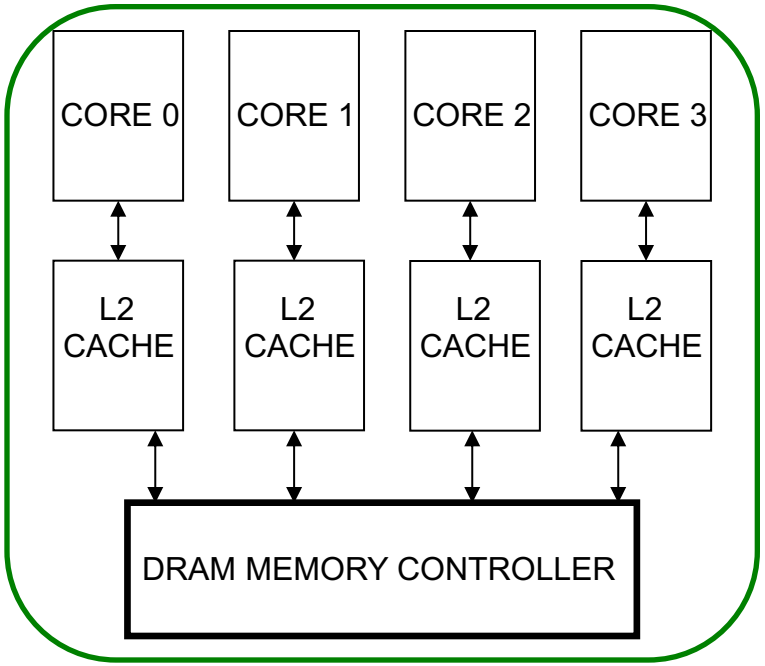
- Cache efficiency becomes even more important in a multi-core/multi-threaded system
  - Memory bandwidth is at premium
  - Cache space is a limited resource across cores/threads

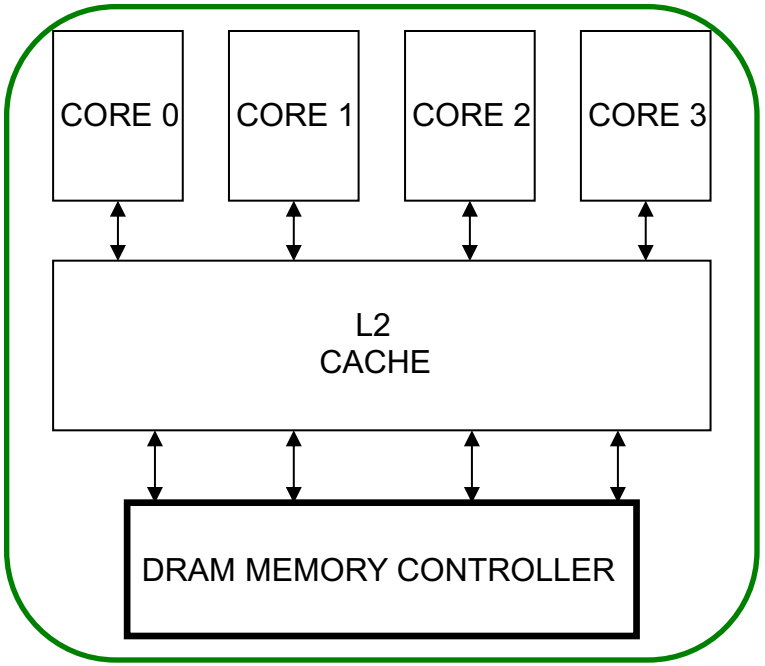
- How do we design the caches in a multi-core system?

- Many decisions and questions
  - Shared vs. private caches
  - How to maximize performance of the entire system?
  - How to provide QoS & predictable perf. to different threads in a shared cache?
  - Should cache management algorithms be aware of threads?
  - How should space be allocated to threads in a shared cache?
  - Should we store data in compressed format in some caches?
  - How do we do better reuse prediction & management in caches?

Private vs. Shared Caches

- Private cache: Cache belongs to one core (a shared block can be in multiple caches)
- Shared cache: Cache is shared by multiple cores





Resource Sharing Concept and Advantages

- Idea: Instead of dedicating a hardware resource to a hardware context, allow multiple contexts to use it
  - Example resources: functional units, pipeline, caches, buses, memory, interconnects, storage
- Why?

+ Resource sharing improves utilization/efficiency → throughput
  - When a resource is left idle by one thread, another thread can use it; no need to replicate shared data
+ Reduces communication latency
  - For example, data shared between multiple threads can be kept in the same cache in multithreaded processors
+ Compatible with the shared memory programming model

Resource Sharing Disadvantages

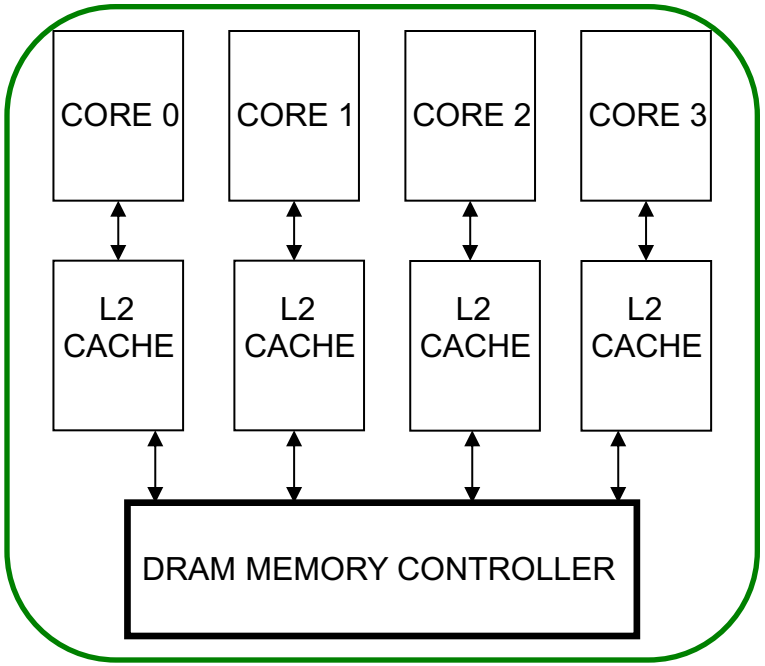
- Resource sharing results in contention for resources
  - When the resource is not idle, another thread cannot use it
  - If space is occupied by one thread, another thread needs to re-occupy it

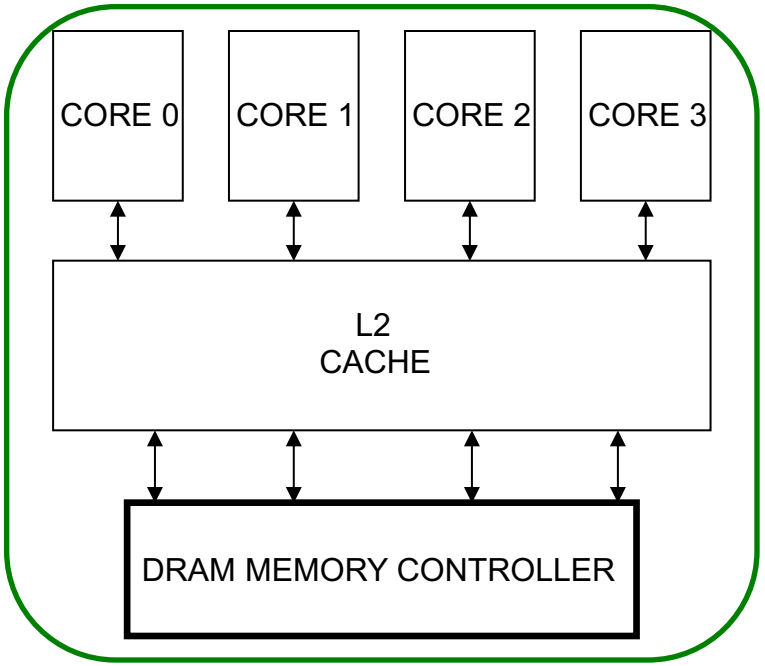
- Sometimes reduces each or some thread's performance
  - Thread performance can be worse than when it is run alone
- Eliminates performance isolation → inconsistent performance across runs
  - Thread performance depends on co-executing threads
- Uncontrolled (free-for-all) sharing degrades quality of service
  - Causes unfairness, starvation

Need to efficiently and fairly utilize shared resources

Private vs. Shared Caches

- Private cache: Cache belongs to one core (a shared block can be in multiple caches)
- Shared cache: Cache is shared by multiple cores



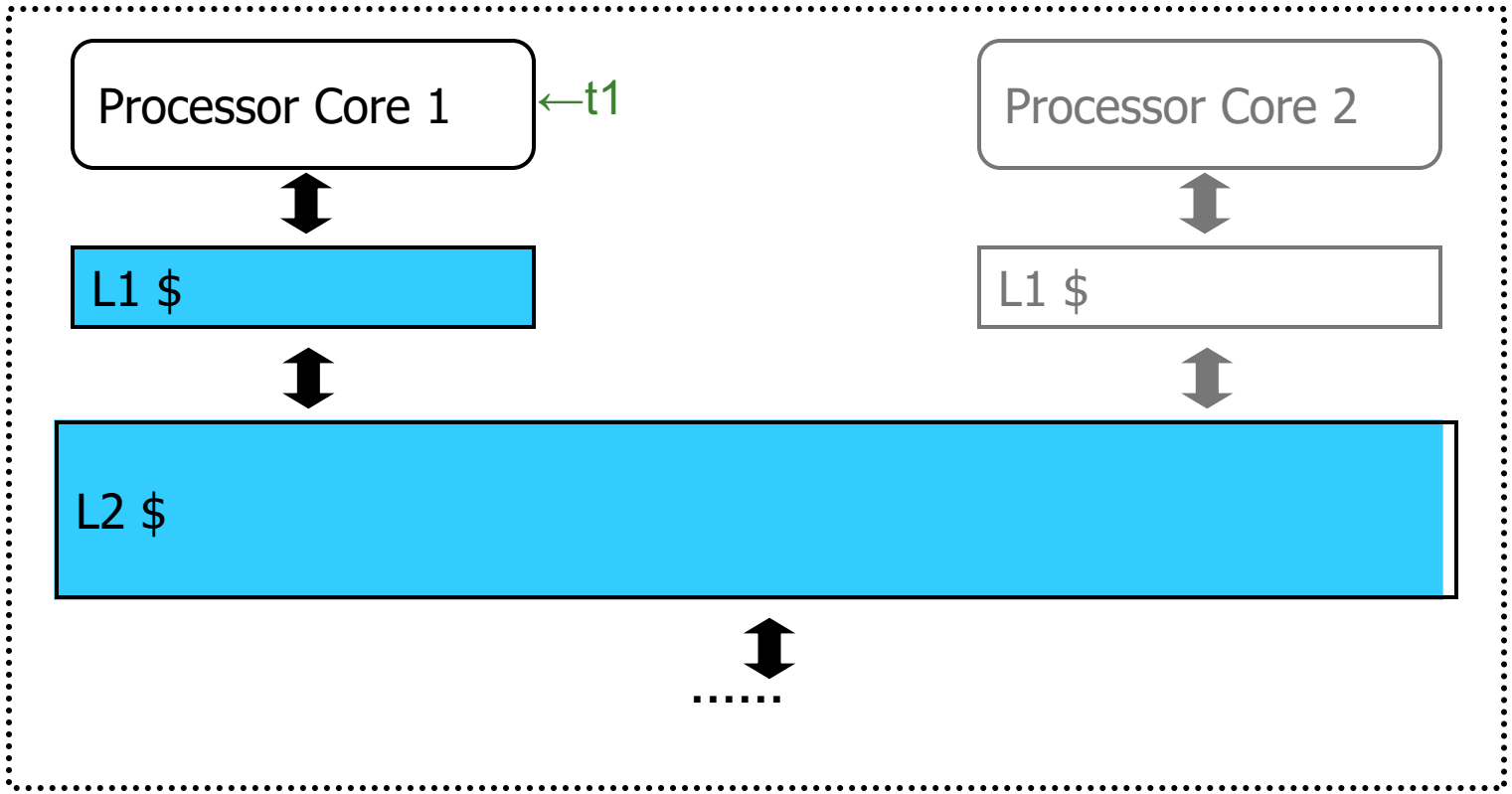


Shared Caches Between Cores

- Advantages:
  - High effective capacity
  - Dynamic partitioning of available cache space
    - No fragmentation due to static partitioning
    - If one core does not utilize some space, another core can
  - Easier to maintain coherence (a cache block is in a single location)

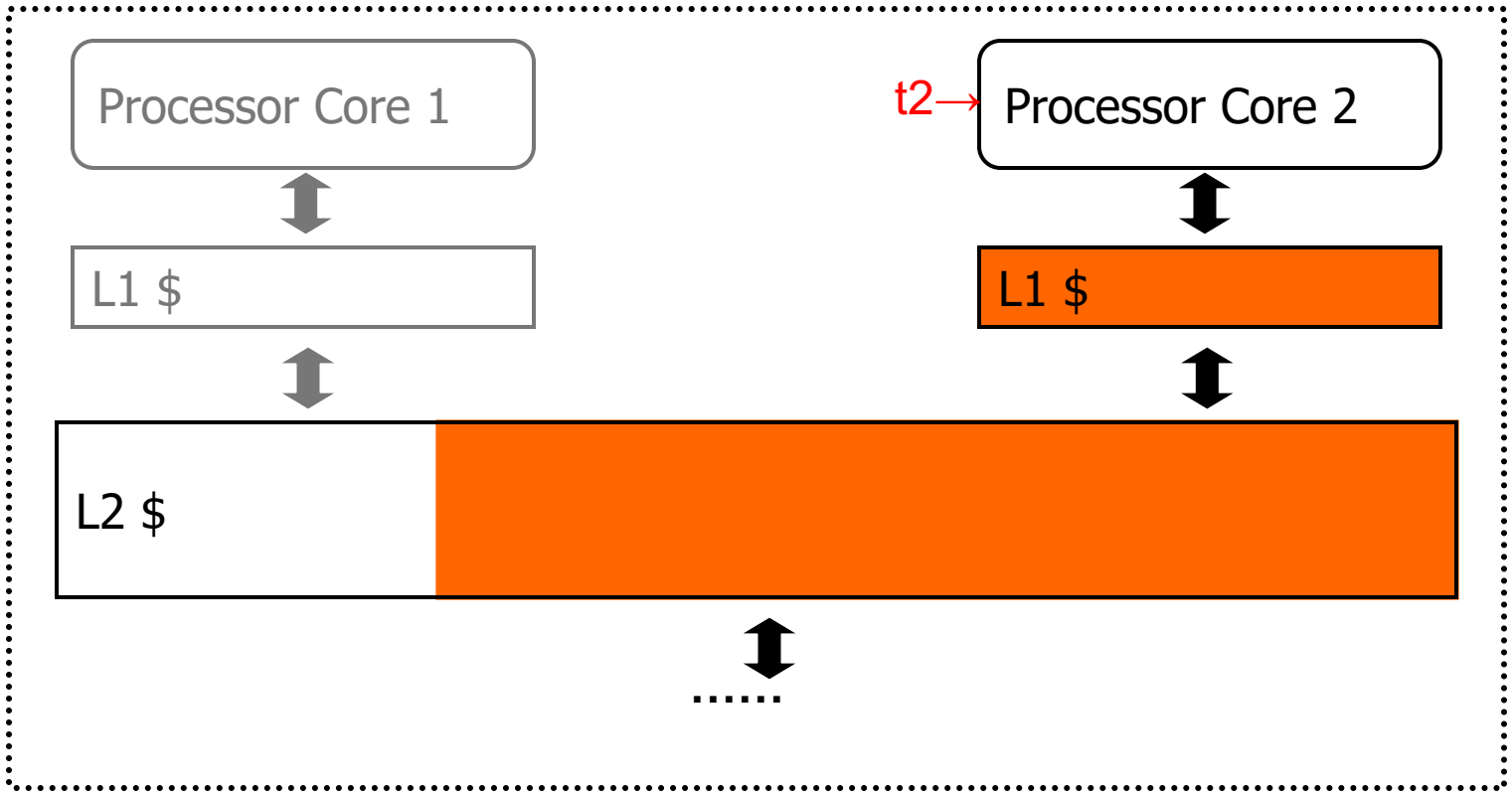
- Disadvantages
  - Slower access (cache is not tightly coupled with the core)
  - Cores incur conflict misses due to other cores' accesses
    - Misses due to inter-core interference
    - Some cores can destroy the hit rate of other cores
  - Guaranteeing a minimum level of service (or fairness) to each core is harder (how much space, how much bandwidth?)

Example: One Problem with Shared Caches



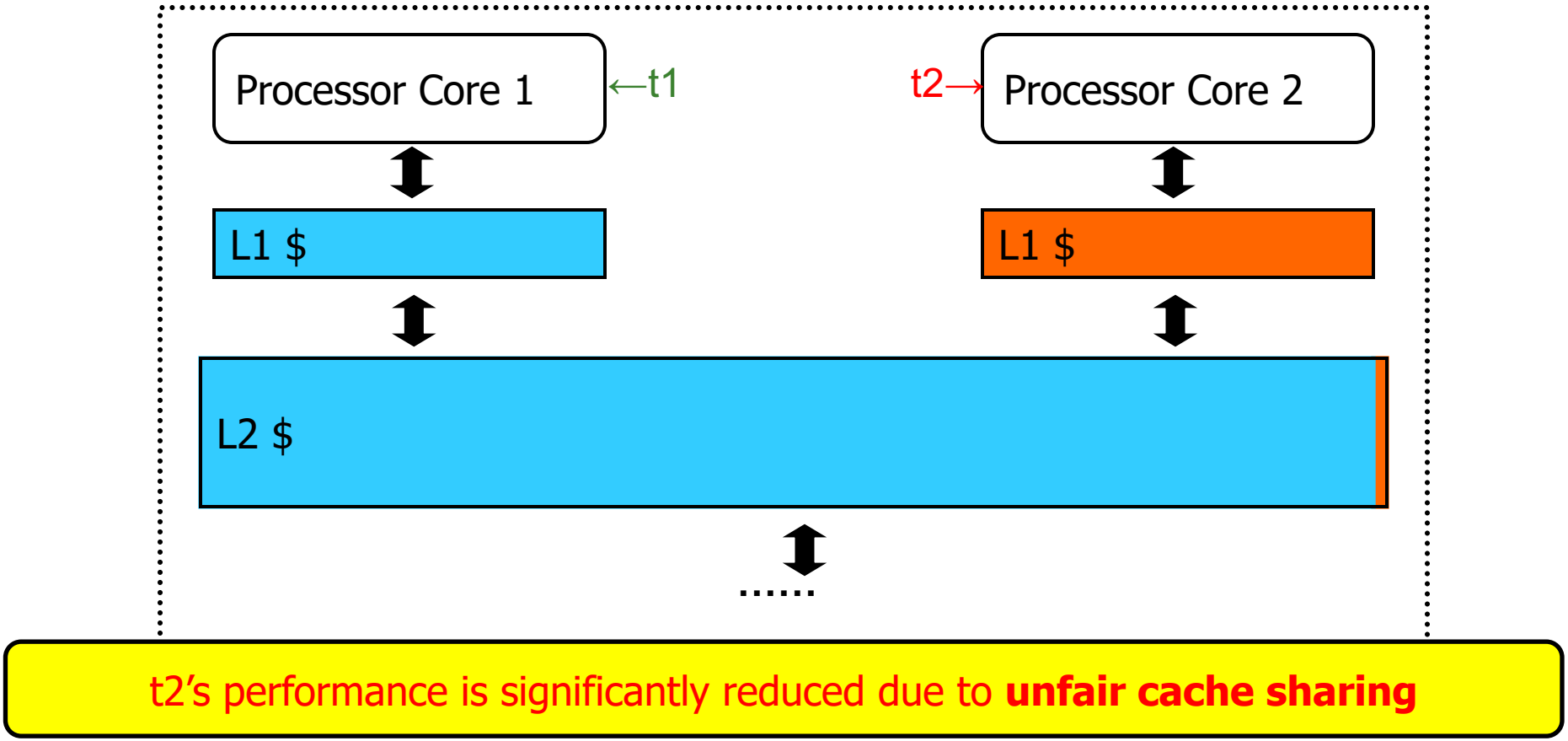
Kim et al., "Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture," PACT 2004.

Example: One Problem with Shared Caches



Kim et al., "Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture," PACT 2004.

Example: One Problem with Shared Caches

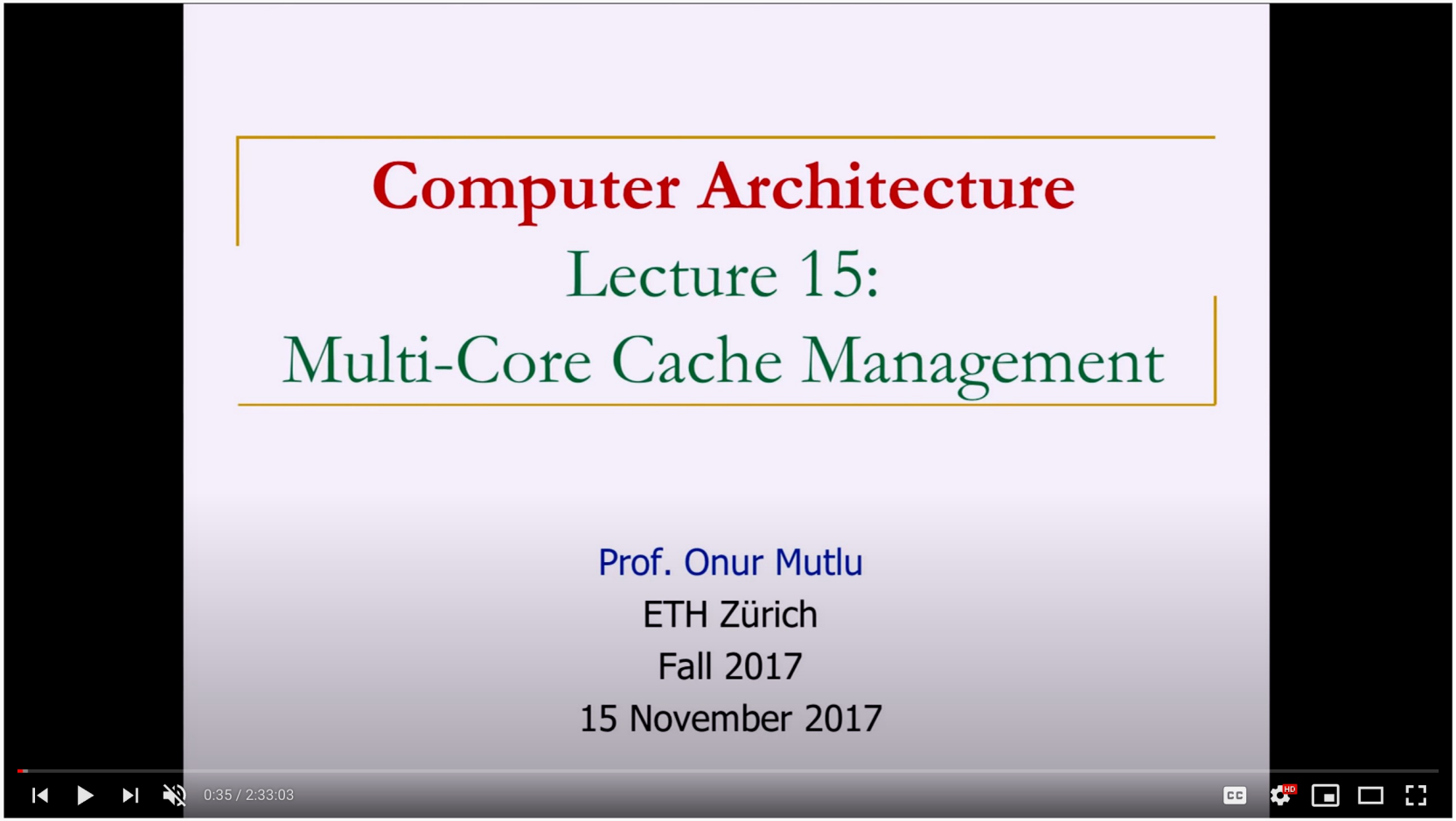


Kim et al., "Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture," PACT 2004.

Resource Sharing vs. Partitioning

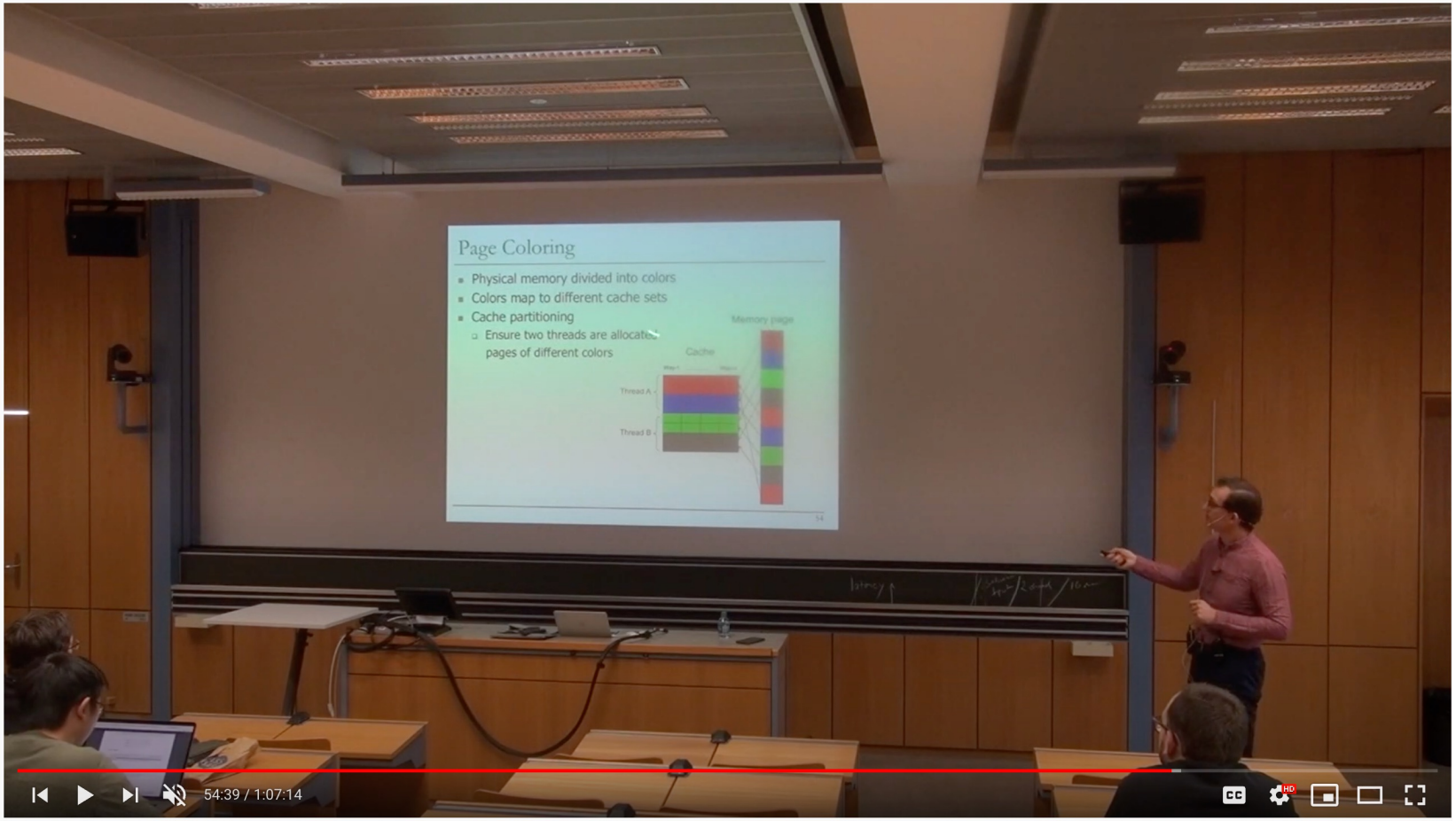
- Sharing improves resource utilization
  - Better utilization of space
- Partitioning provides performance isolation (predictable performance)
  - Dedicated space
- Can we get the benefits of both?
- Idea: Design shared resources such that they are efficiently utilized, controllable and partitionable
  - No wasted resource + QoS mechanisms for threads

Lectures on Multi-Core Cache Management



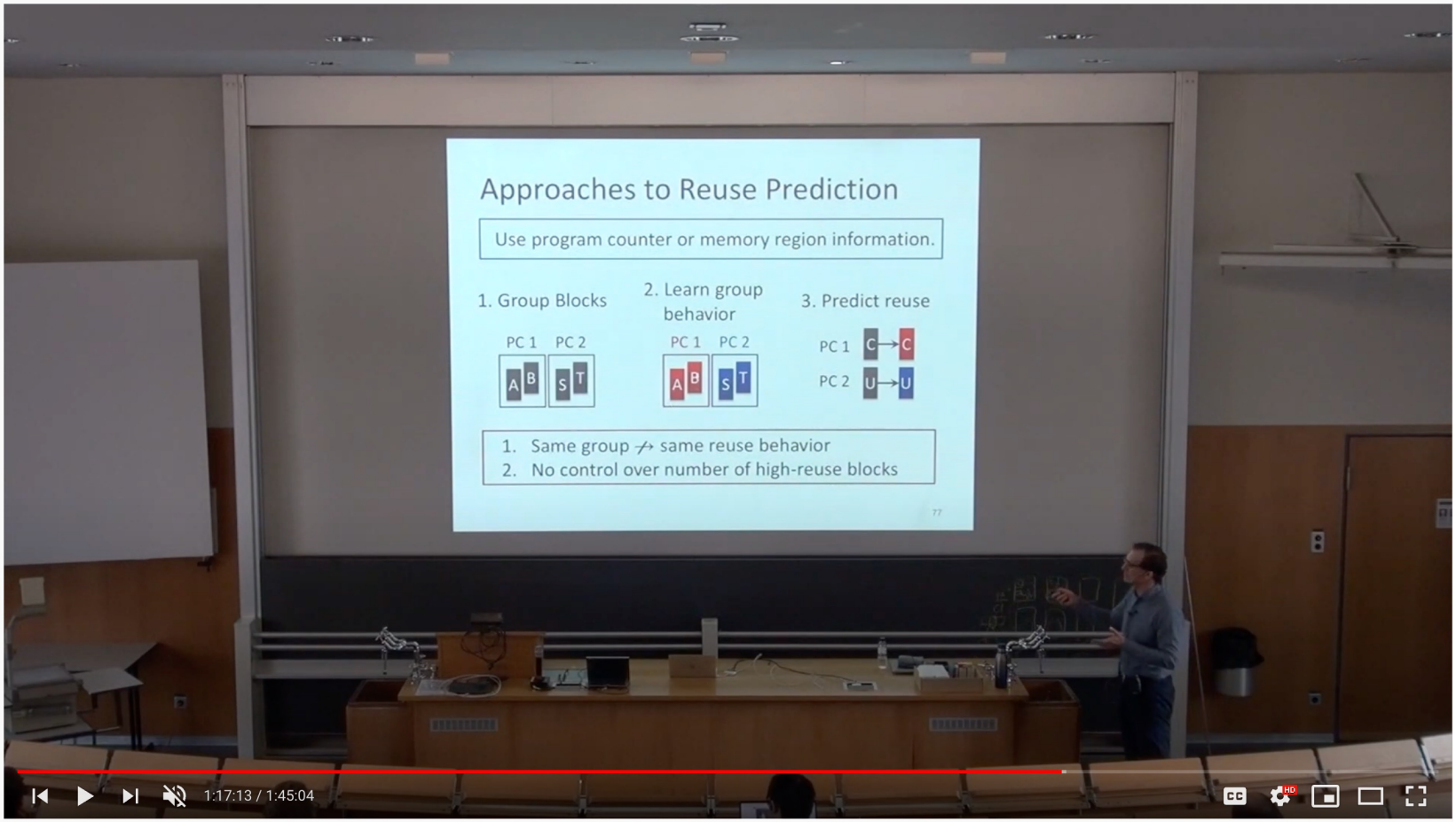
https://www.youtube.com/watch?v=7_Tqlw8gxOU&list=PL5Q2soXY2Zi9OhoVQBXYFIZywZXCPl4M_&index=17

Lectures on Multi-Core Cache Management



https://www.youtube.com/watch?v=c9FhGRB3HoA&list=PL5Q2soXY2Zi9JXe3ywQMhylk_d5dI-TM7&index=29

Lectures on Multi-Core Cache Management



ETH ZÜRICH HAUPTGEBÄUDE

Computer Architecture - Lecture 19a: Multi-Core Cache Management II (ETH Zürich, Fall 2018)

293 views • Dec 2, 2018

6 0 SHARE SAVE ...

Onur Mutlu Lectures
16.5K subscribers

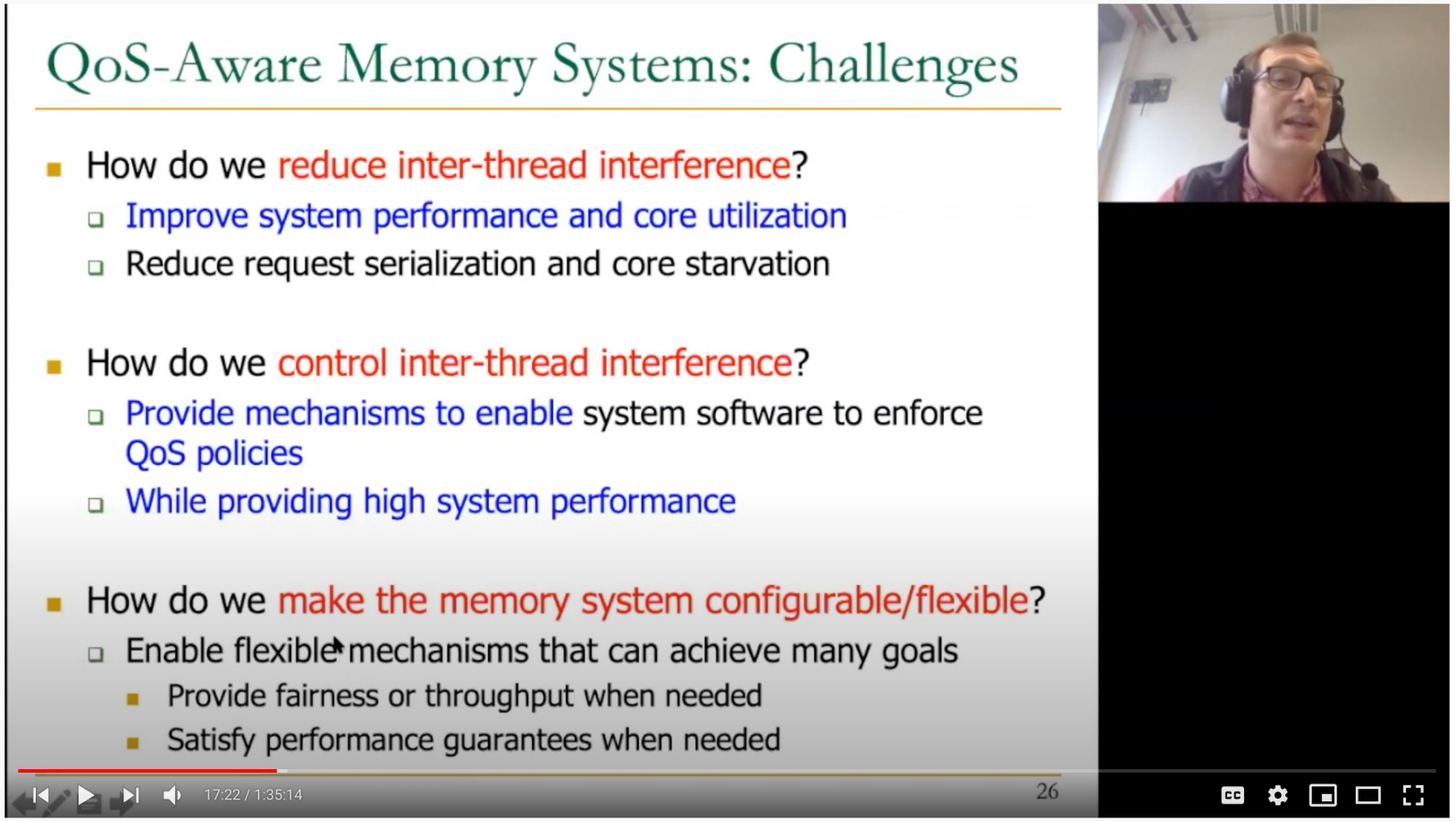
ANALYTICS EDIT VIDEO

https://www.youtube.com/watch?v=Siz86__PD4w&list=PL5Q2soXY2Zi9JXe3ywQMhylk_d5dI-TM7&index=30

Lectures on Multi-Core Cache Management

- Computer Architecture, Fall 2018, Lecture 18b
  - Multi-Core Cache Management (ETH, Fall 2018)
  - https://www.youtube.com/watch?v=c9FhGRB3HoA&list=PL5Q2soXY2Zi9JXe3ywQMhylk_d5dI-TM7&index=29
- Computer Architecture, Fall 2018, Lecture 19a
  - Multi-Core Cache Management II (ETH, Fall 2018)
  - https://www.youtube.com/watch?v=Siz86__PD4w&list=PL5Q2soXY2Zi9JXe3ywQMhylk_d5dI-TM7&index=30
- Computer Architecture, Fall 2017, Lecture 15
  - Multi-Core Cache Management (ETH, Fall 2017)
  - https://www.youtube.com/watch?v=7_Tqlw8gxOU&list=PL5Q2soXY2Zi9OhoVQBXYFIZywZXCPl4M_&index=17

Lectures on Memory Resource Management



https://www.youtube.com/watch?v=0nnI807nCkc&list=PL5Q2soXY2Zi9xidyIgBxUz7xRPS-wisBN&index=21

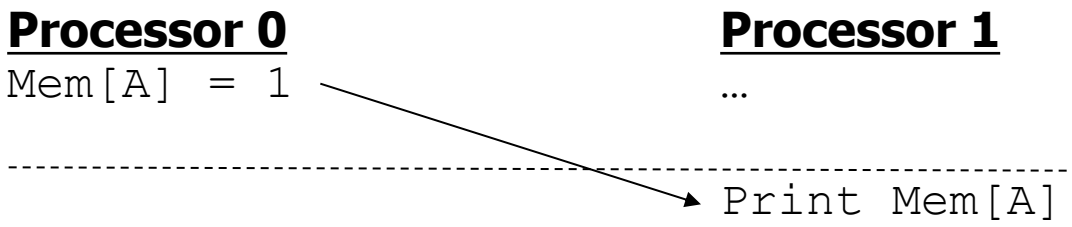
Lectures on Memory Resource Management

- Computer Architecture, Fall 2020, Lecture 11a
  - Memory Controllers (ETH, Fall 2020)
  - https://www.youtube.com/watch?v=TeG773OgiMQ&list=PL5Q2soXY2Zi9xidyIgBxUz7xRPS-wisBN&index=20
- Computer Architecture, Fall 2020, Lecture 11b
  - Memory Interference and QoS (ETH, Fall 2020)
  - https://www.youtube.com/watch?v=0nnI807nCkc&list=PL5Q2soXY2Zi9xidyIgBxUz7xRPS-wisBN&index=21
- Computer Architecture, Fall 2020, Lecture 13
  - Memory Interference and QoS II (ETH, Fall 2020)
  - https://www.youtube.com/watch?v=Axye9VqQT7w&list=PL5Q2soXY2Zi9xidyIgBxUz7xRPS-wisBN&index=26
- Computer Architecture, Fall 2020, Lecture 2a
  - Memory Performance Attacks (ETH, Fall 2020)
  - https://www.youtube.com/watch?v=VJzZbwgBfy8&list=PL5Q2soXY2Zi9xidyIgBxUz7xRPS-wisBN&index=2

Cache Coherence

Shared Memory Model

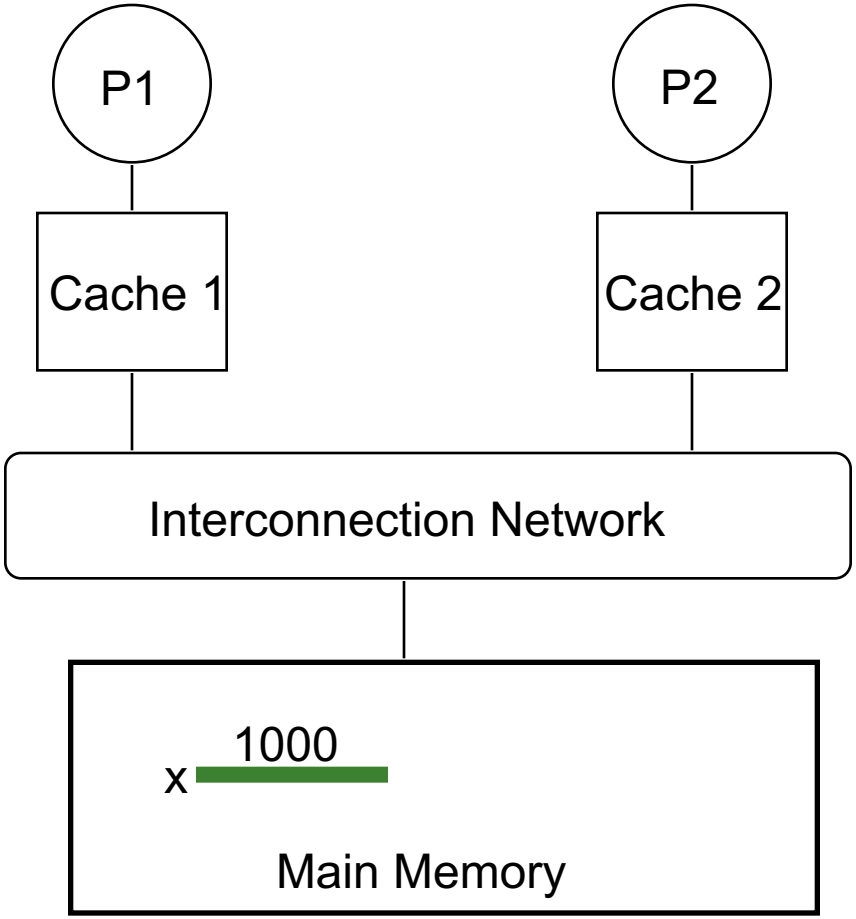
- Threads in parallel programs communicate through *shared memory*
- Thread 0 writes to an address, followed by Thread 1 reading
  - This implies communication between the two



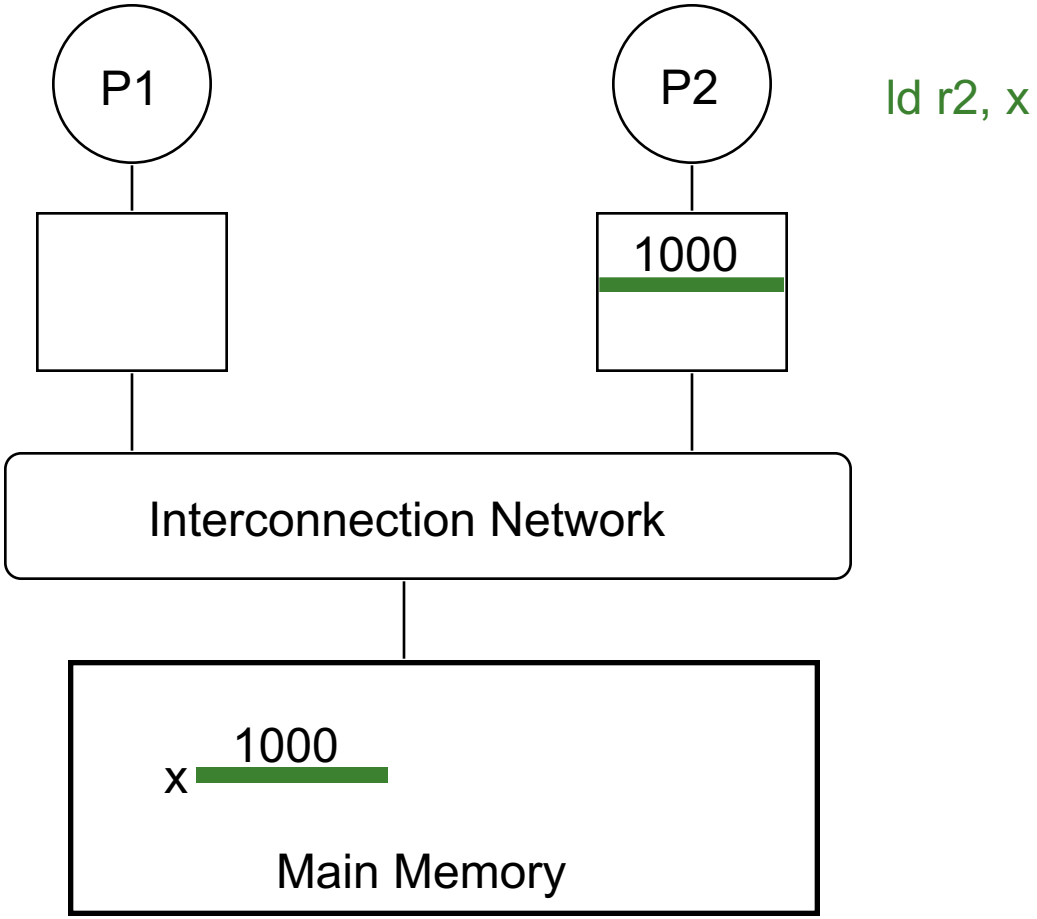
- Each read should receive the value last written by any processor
  - This requires synchronization between threads (i.e., what does last written mean?)
- What if Mem[A] is cached (at either processor)?

Cache Coherence

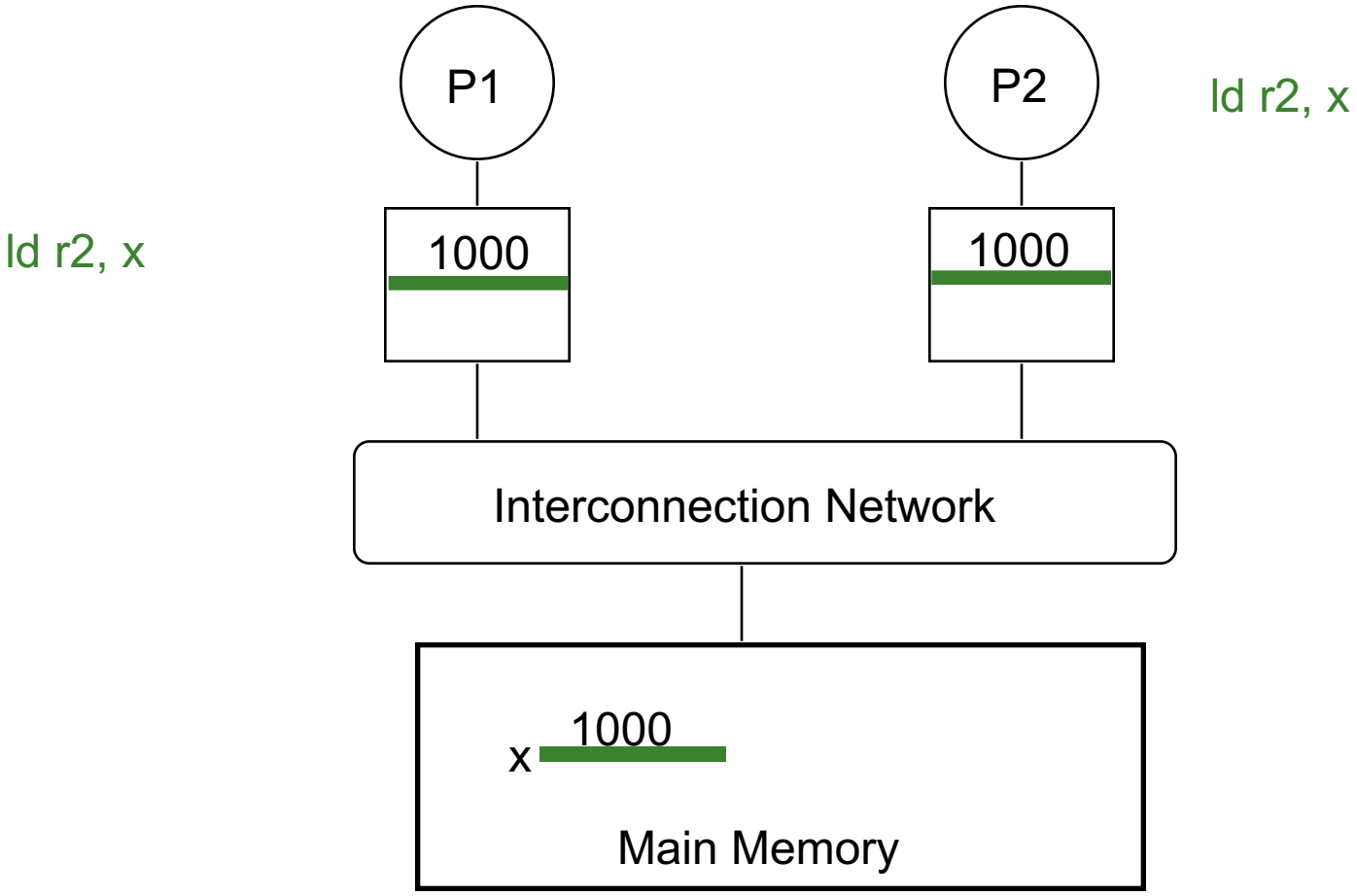
- Basic question: If multiple processors cache the same block, how do they ensure they all see a consistent state?



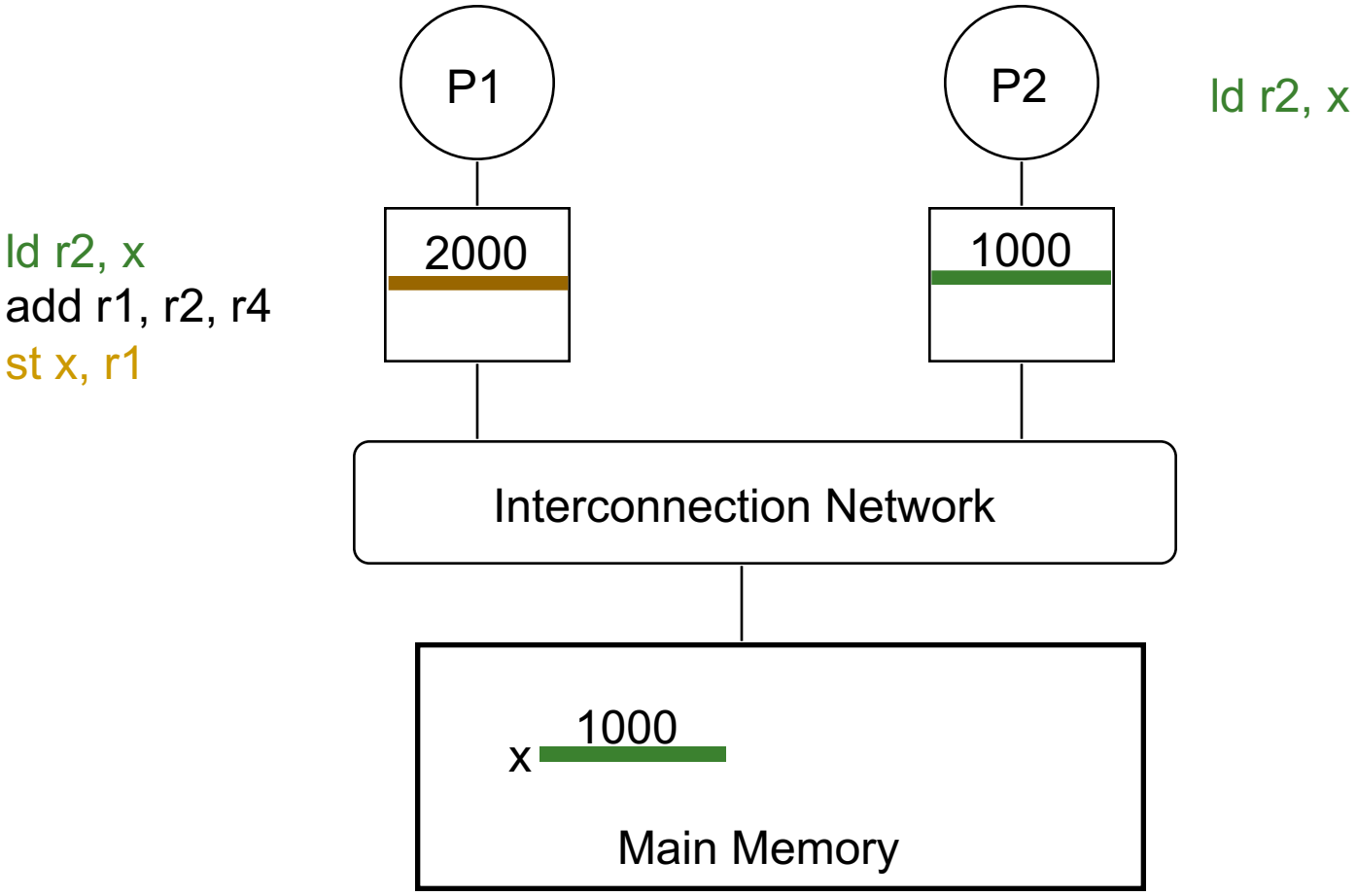
The Cache Coherence Problem



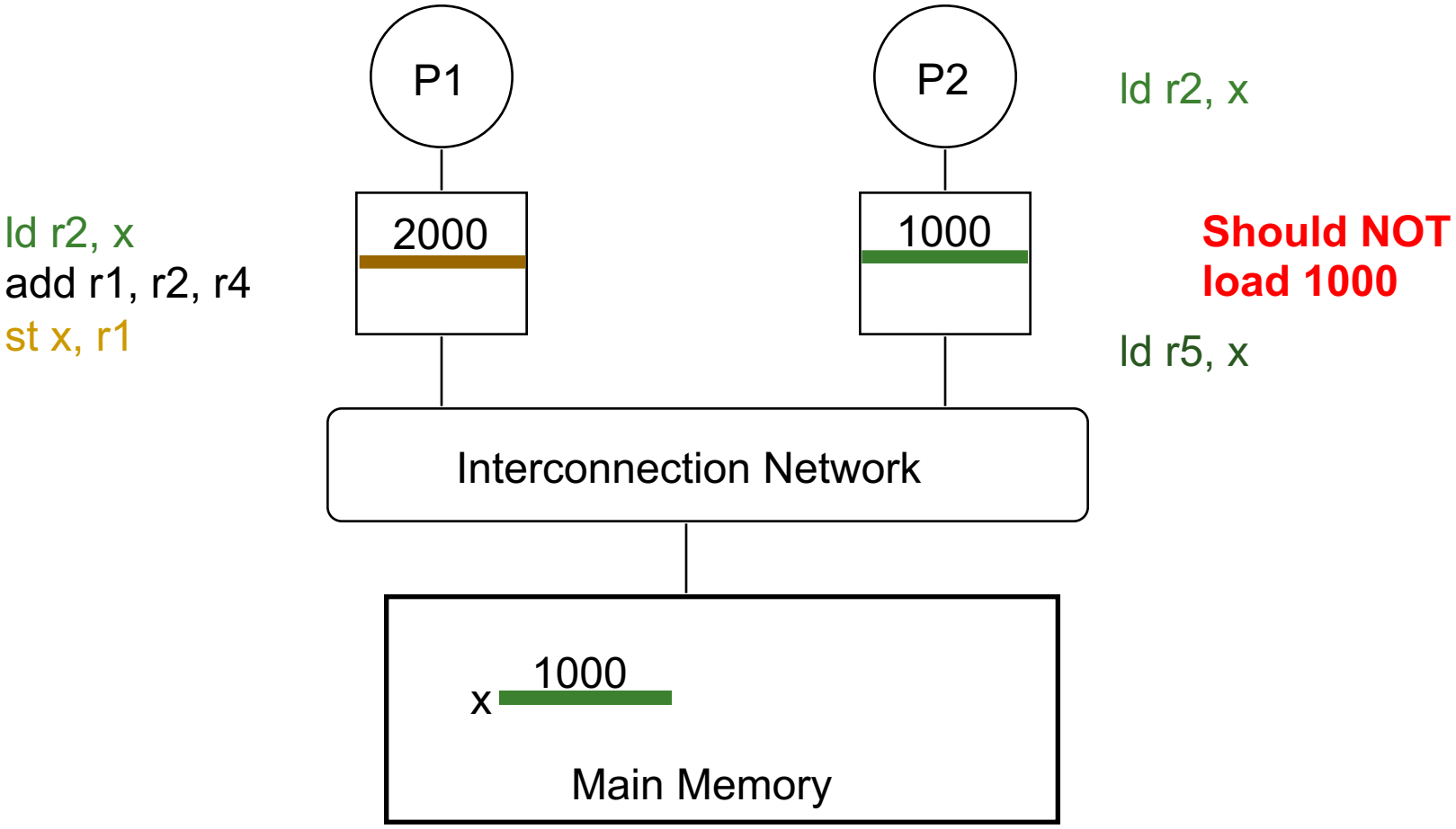
The Cache Coherence Problem



The Cache Coherence Problem

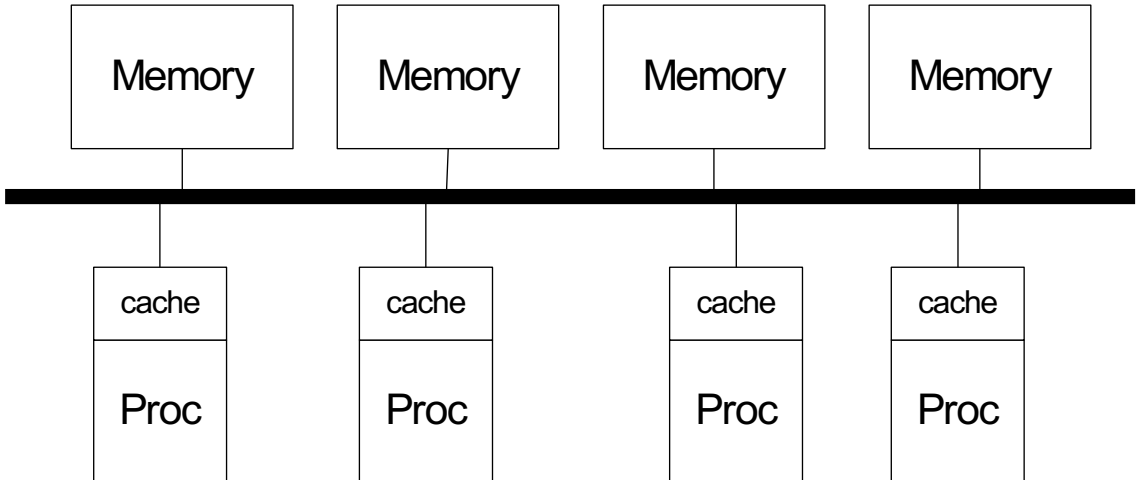


The Cache Coherence Problem



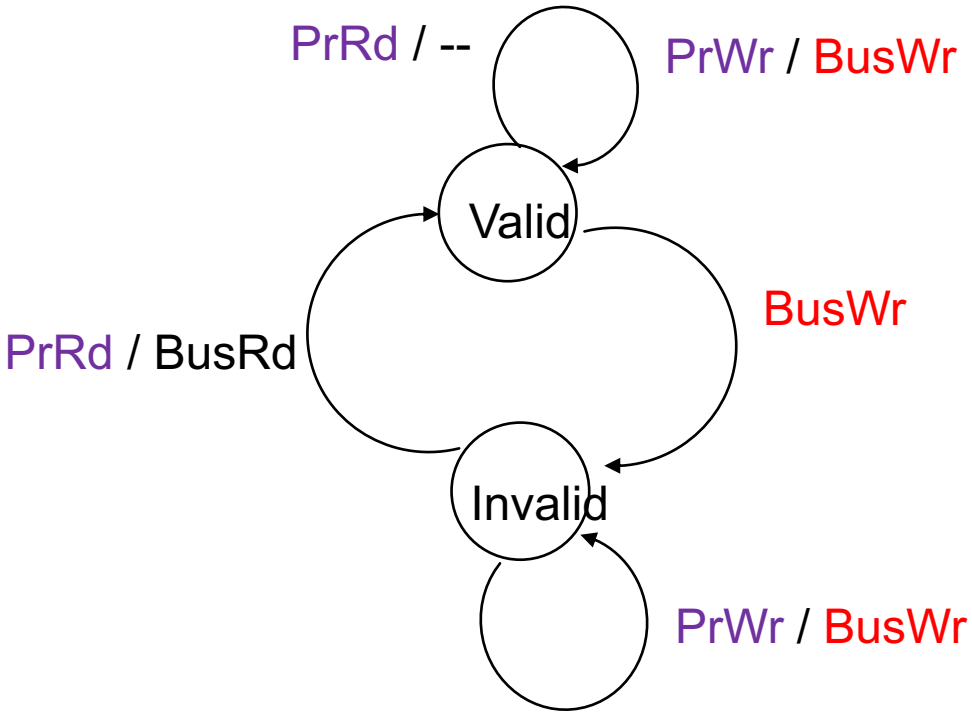
Broadcast-Based Hardware Cache Coherence

- Basic idea:
  - A processor/cache broadcasts its write/update to a cache block to all other processors
  - Another processor/cache that has the block either invalidates or updates its local copy of the block



A Very Simple Coherence Scheme (VI)

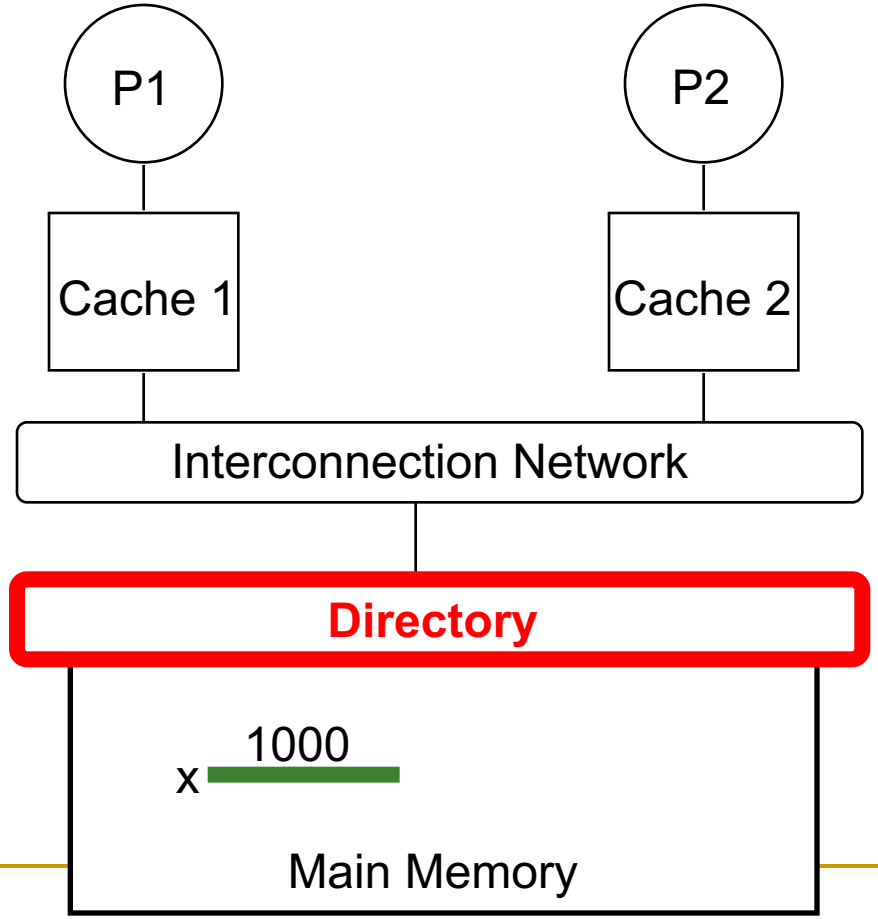
- Idea: All caches "snoop" (observe) each other's write/read operations. If a processor writes to a block, all others invalidate the block.
- A simple protocol:



- Assumption: Write-through, no-write-allocate cache
- Actions of the local processor on the cache block: PrRd, PrWr
- Actions that are broadcast on the bus for the block: BusRd, BusWr

Directory-Based Cache Coherence

- Idea: A central directory keeps track of which caches contain every possible cache block. Every cache consults this directory to ensure data is kept coherent.

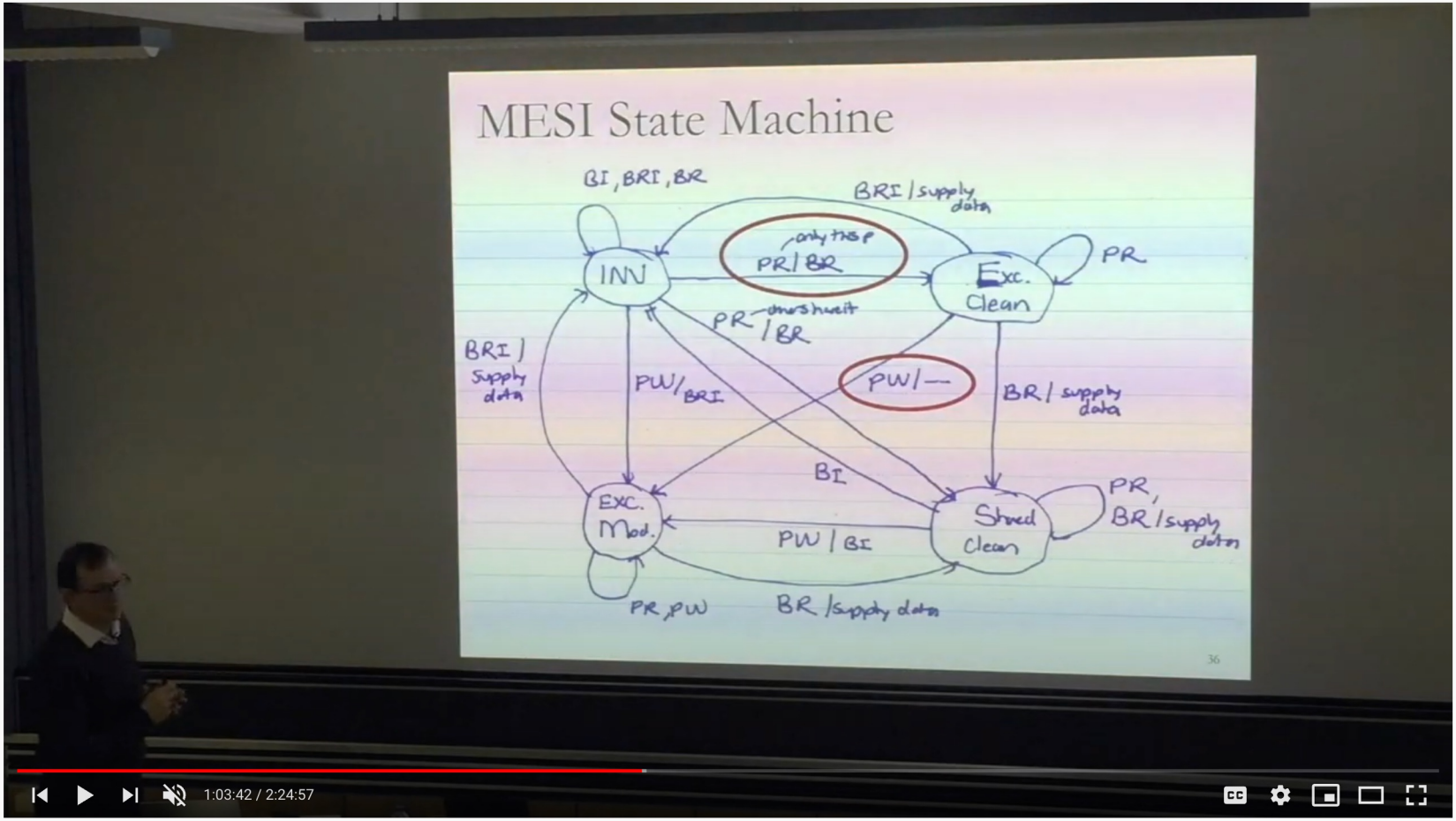


Directory-Based Cache Coherence

- Idea: A central directory keeps track of which caches contain every possible cache block. Every cache consults this directory to ensure data is kept coherent.

- An example mechanism:
  - For each cache block in memory, store P+1 bits in directory
    - One bit for each cache, indicating whether the block is in cache
    - Exclusive bit: indicates that a cache has the only copy of the block and can update it without notifying others
  - On a read: set the cache's bit in the directory entry for the block and arrange the supply of the block into the cache
  - On a write: invalidate the block in all caches that have the block and reset their bits in directory entry for the block
  - Have an "exclusive bit" associated with each block in each cache (so that the cache can update the exclusive block silently)

Lecture on Cache Coherence



Computer Architecture - Lecture 21: Cache Coherence (ETH Zürich, Fall 2020)

1,419 views • Dec 4, 2020

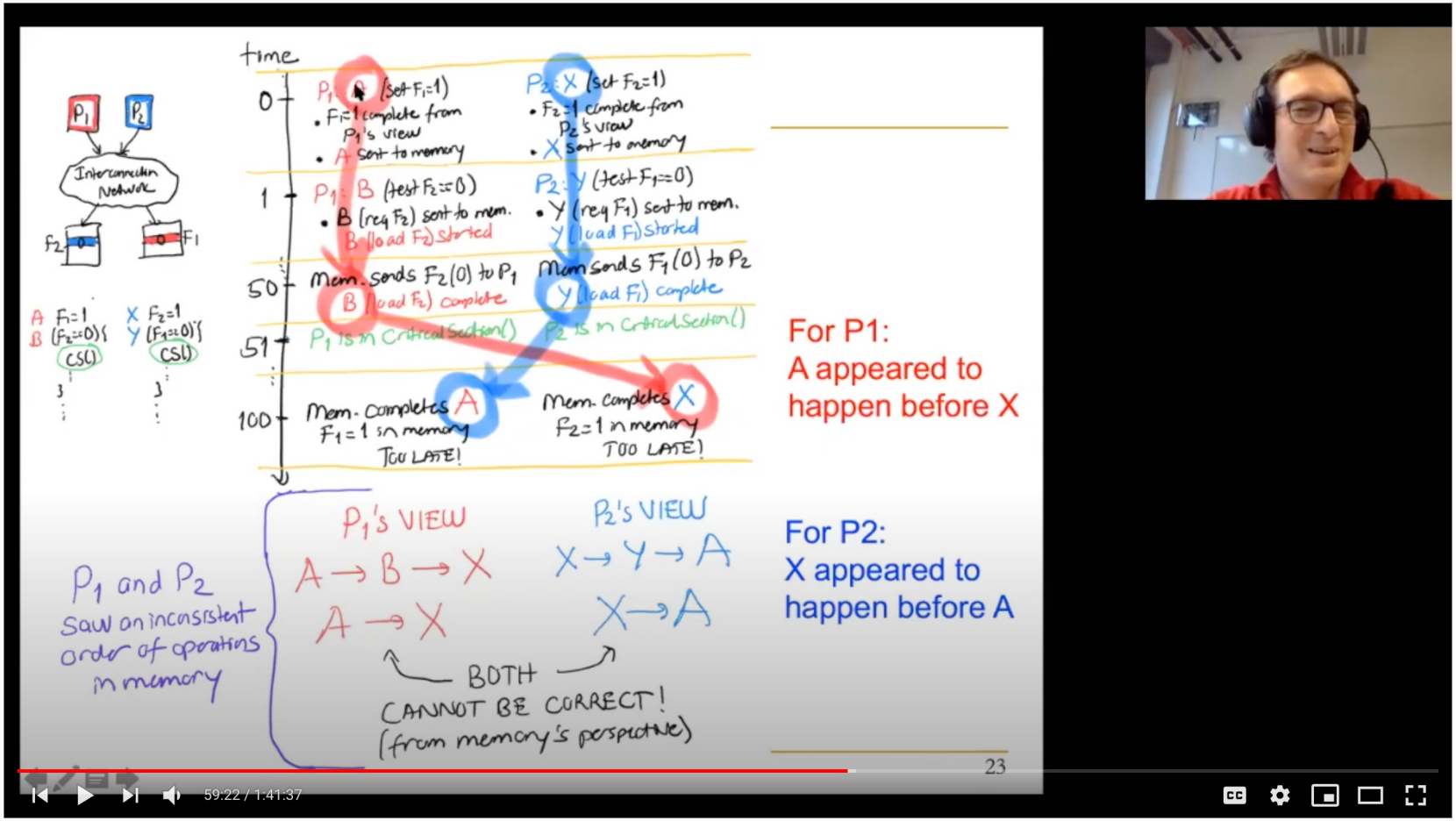
27 0 SHARE SAVE

Onur Mutlu Lectures
16.3K subscribers

ANALYTICS EDIT VIDEO

https://www.youtube.com/watch?v=T9WlyezeaII&list=PL5Q2soXY2Zi9xidyIgBxUz7xRPS-wisBN&index=38

Lecture on Memory Ordering & Consistency



Computer Architecture - Lecture 20: Memory Ordering (Memory Consistency) (ETH Zürich, Fall 2020)

976 views • Dec 4, 2020

22 0 SHARE SAVE ...

Onur Mutlu Lectures
16.5K subscribers

ANALYTICS EDIT VIDEO

https://www.youtube.com/watch?v=Suy09mzTbiQ&list=PL5Q2soXY2Zi9xidyIgBxUz7xRPS-wisBN&index=37

Lecture on Cache Coherence & Consistency

- Computer Architecture, Fall 2020, Lecture 21
  - Cache Coherence (ETH, Fall 2020)
  - https://www.youtube.com/watch?v=T9WlyezeaII&list=PL5Q2soXY2Zi9xidyIgBxUz7xRPS-wisBN&index=38
- Computer Architecture, Fall 2020, Lecture 20
  - Memory Ordering & Consistency (ETH, Fall 2020)
  - https://www.youtube.com/watch?v=Suy09mzTbiQ&list=PL5Q2soXY2Zi9xidyIgBxUz7xRPS-wisBN&index=37
- Computer Architecture, Spring 2015, Lecture 28
  - Memory Consistency & Cache Coherence (CMU, Spring 2015)
  - https://www.youtube.com/watch?v=JfjT1a0vi4E&list=PL5PHm2jkkXmi5CxxI7b3JCL1TWybTDtKq&index=32
- Computer Architecture, Spring 2015, Lecture 29
  - Cache Coherence (CMU, Spring 2015)
  - https://www.youtube.com/watch?v=X6DZchnMYcw&list=PL5PHm2jkkXmi5CxxI7b3JCL1TWybTDtKq&index=33

Computer Architecture

Lecture 32: Cache Design and Management

Prof. Onur Mutlu
ETH Zürich

Fall 2023

14 February 2024