Digital Design & Computer Arch.

Lecture 24a: Advanced Caches

Prof. Onur Mutlu

ETH Zürich
Spring 2021
28 May 2021

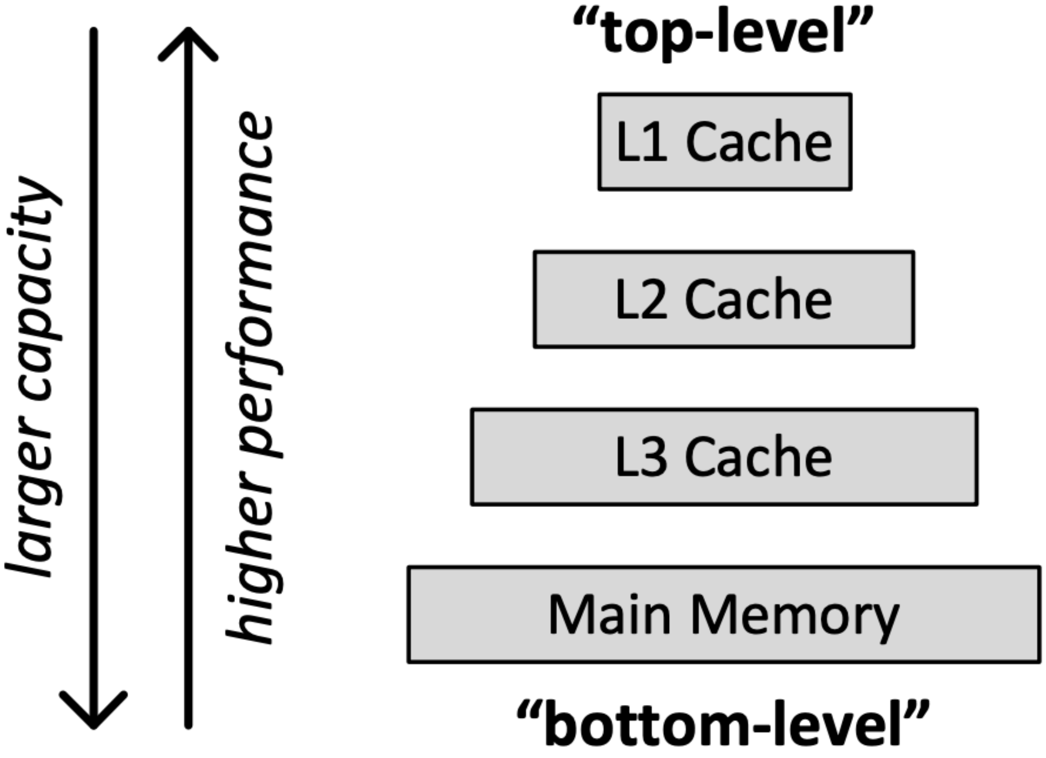
Readings for This Lecture and Next

- 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.

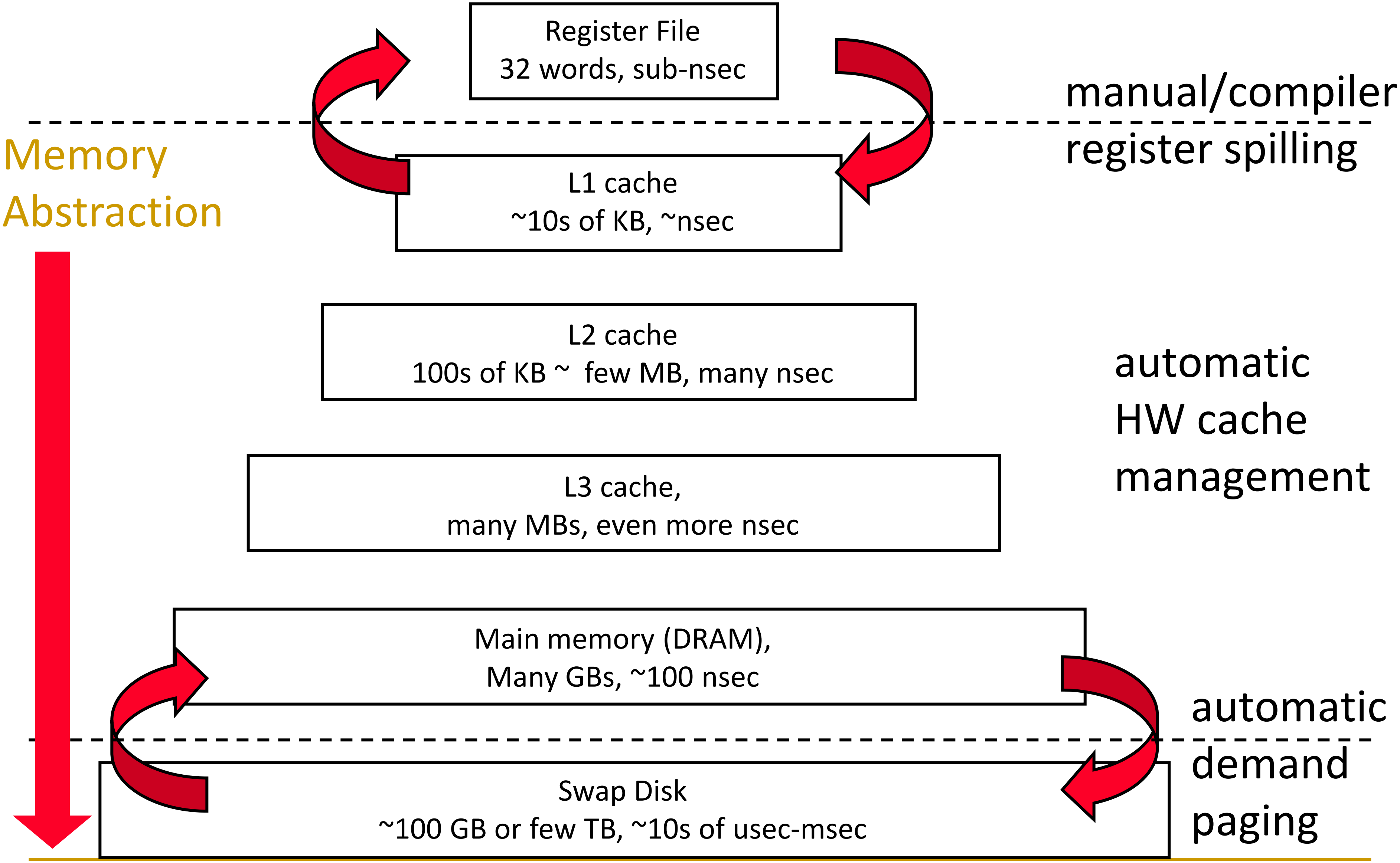
Recall: Memory Hierarchy Example



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



Register File
32 words, sub-nsec

manual/compiler
register spilling

Memory
Abstraction

L1 cache
~10s of KB, ~nsec

L2 cache
100s of KB ~ few MB, many nsec

automatic
HW cache
management

L3 cache,
many MBs, even more nsec

Main memory (DRAM),
Many GBs, ~100 nsec

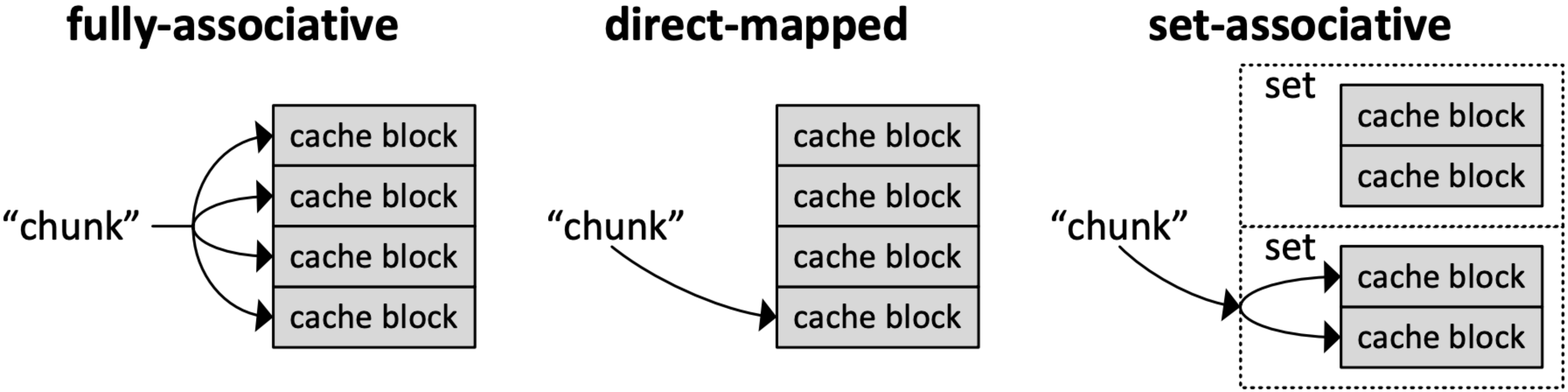
automatic
demand
paging

Swap Disk
~100 GB or few TB, ~10s of usec-msec

Recall: Let's See A Toy 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

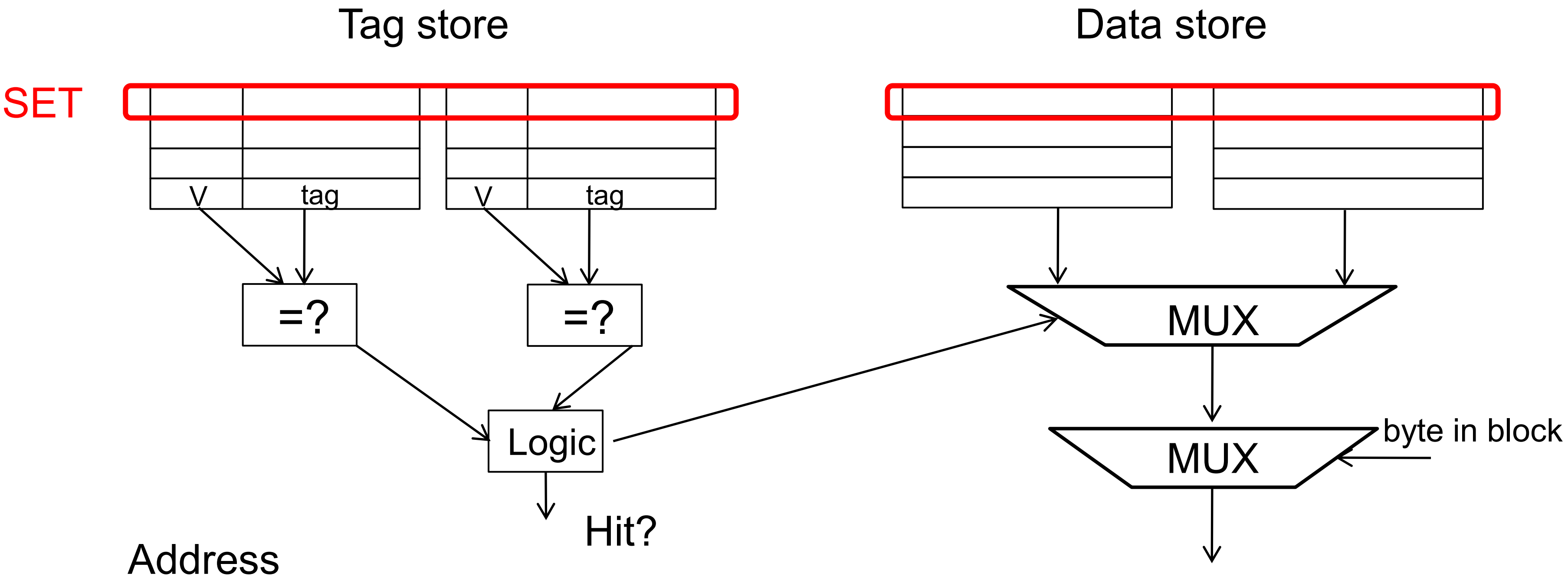
- Toy example: 256-byte memory, 64-byte cache, 8-byte blocks



Kim & Mutlu, "Memory Systems," Computing Handbook, 2014

Recall: Set Associativity

- Addresses N and N+8 always conflict in direct mapped cache
- Instead of having one column of 8, have 2 columns of 4 blocks



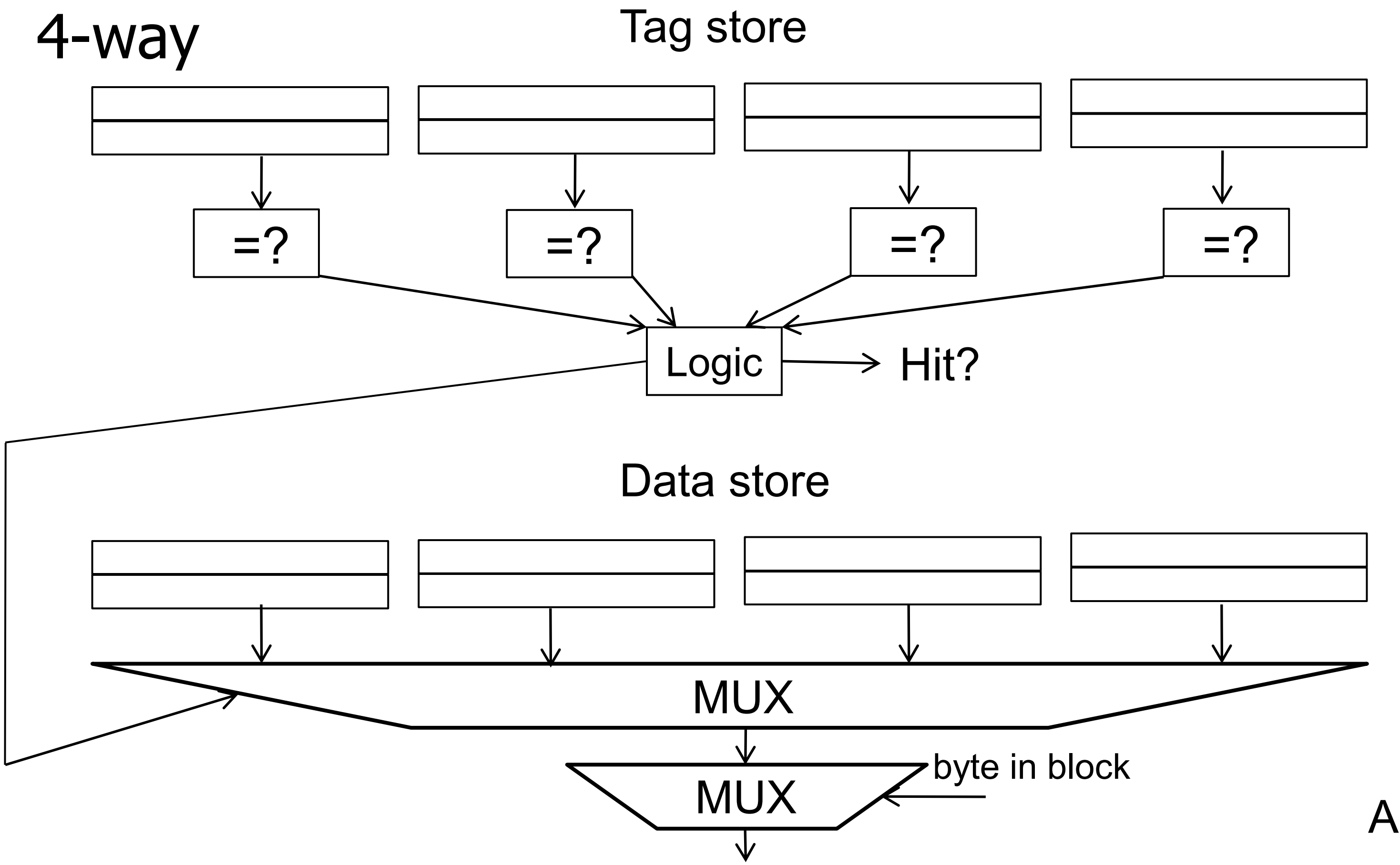
Address

tag	index	byte in block
3 bits	2 bits	3 bits

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

Higher Associativity

- 4-way



Address

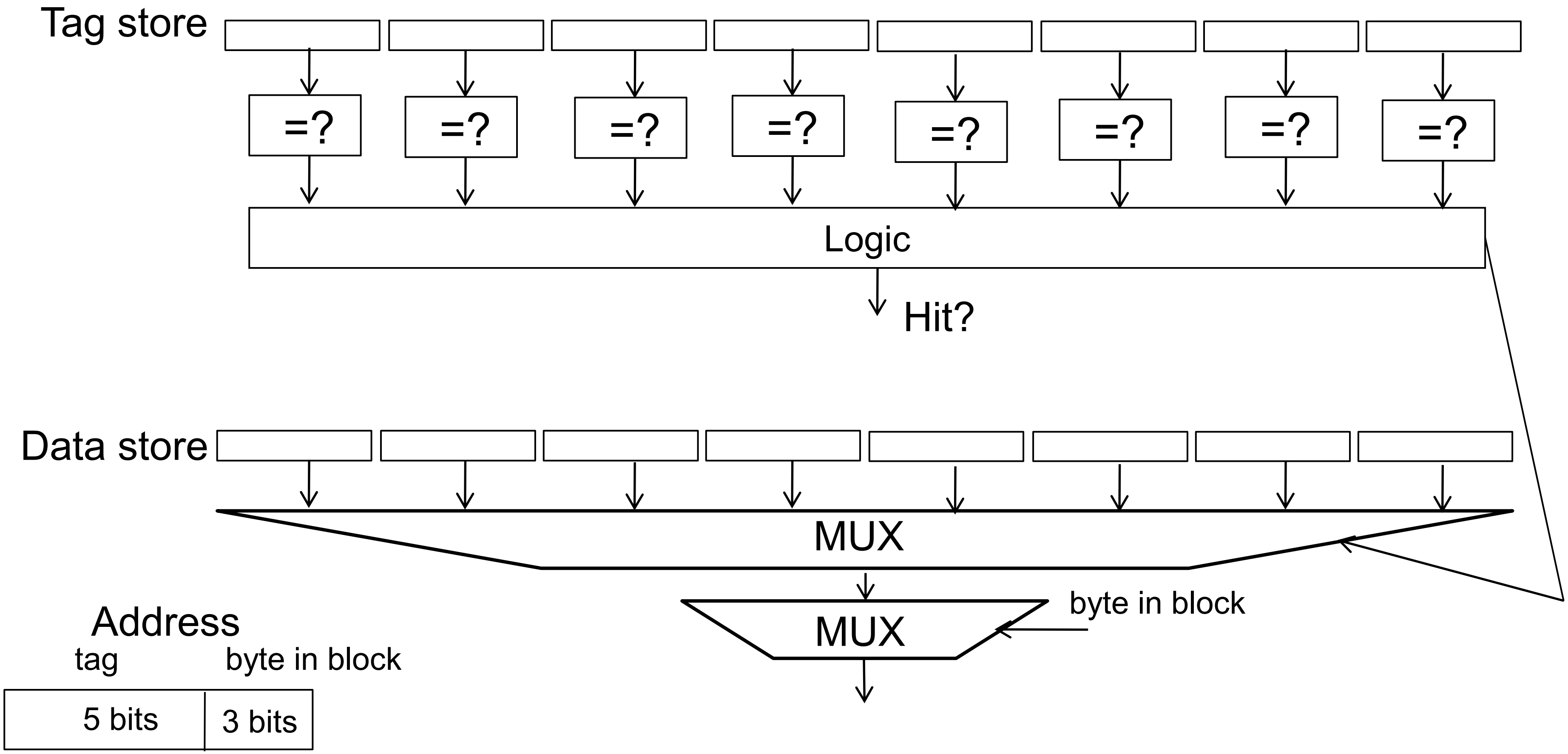
tag	index	byte in block
4 bits	1 b	3 bits

+ Likelihood of conflict misses even lower

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

Full Associativity

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



Recall: Set Associativity (and Tradeoffs)

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

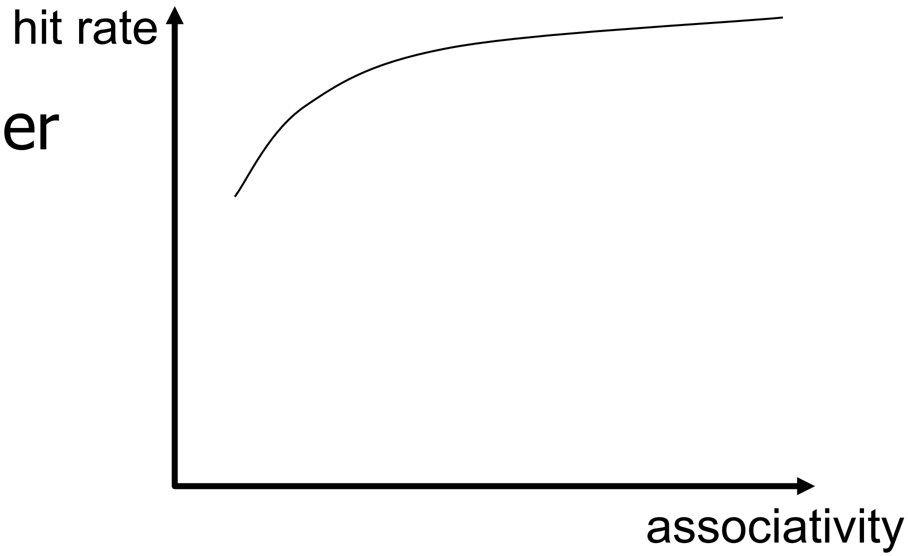
- Higher associativity

  ++ Higher hit rate

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

  -- More expensive hardware (more comparators)

- Diminishing returns from higher associativity



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: 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 ordering of blocks

- Question: 2-way set associative cache:
  - What do you need to implement LRU perfectly?

- Question: 4-way set associative cache:
  - What do you need to implement LRU perfectly?
  - How many different orderings 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?

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?

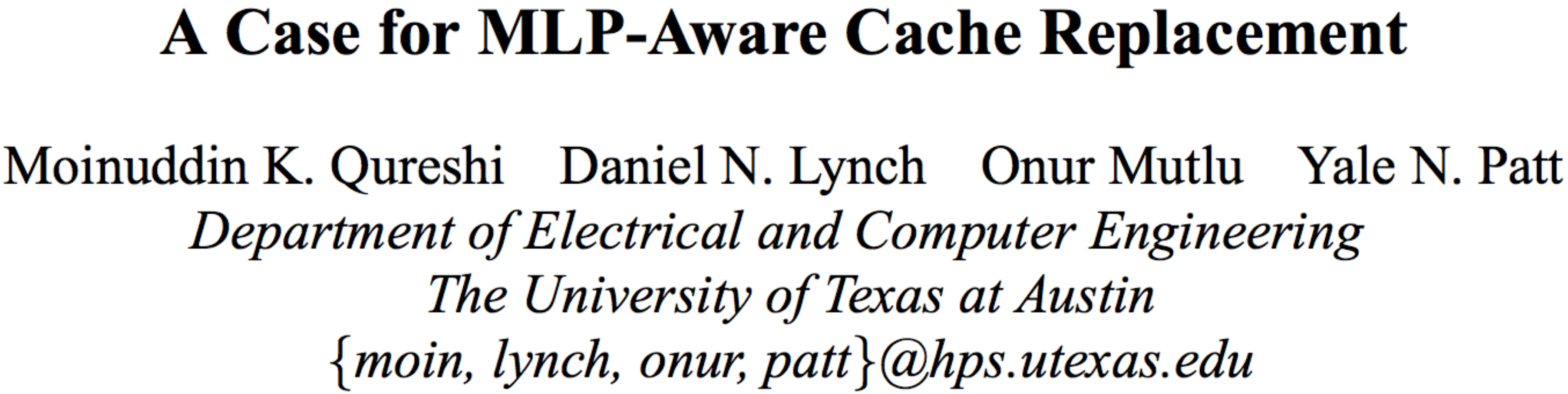
- Belady's OPT
  - Replace the block that is going to be referenced furthest in the future by the program
  - Belady, "A study of replacement algorithms for a virtual-storage computer," IBM Systems Journal, 1966.
  - How do we implement this? Simulate?

- 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: Remote vs. local caches 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

  + 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

  + 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 of them 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 writes 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

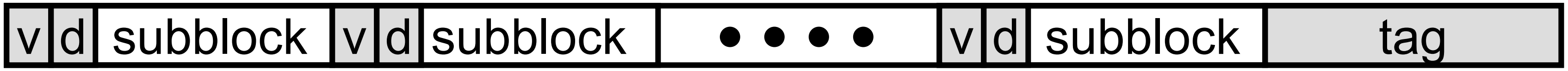
(A write simply validates and updates a subblock)

++ 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

-- May not exploit spatial locality fully



v	d	subblock	v	d	subblock	• • • •	v	d	subblock	tag

Instruction vs. Data Caches

- Separate or Unified?

- Pros and Cons of Unified:

  + Dynamic sharing of cache space: no overprovisioning that might happen with static partitioning (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?

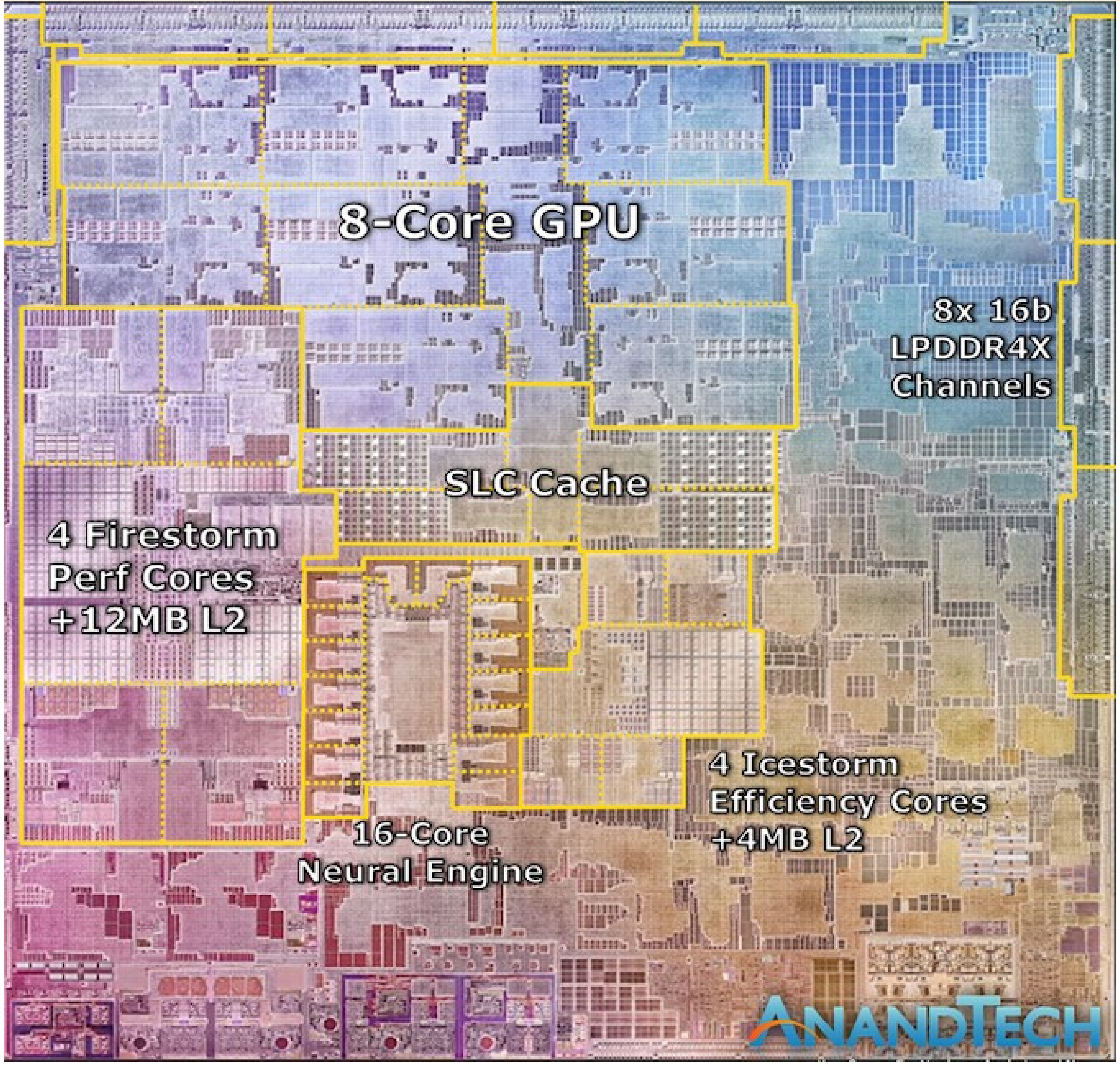
- First level caches are almost always split
  - Mainly for the last reason above – pipeline constraints
- Higher level caches are almost always unified

Multi-level Caching in a Pipelined Design

- First-level caches (instruction and data)
  - Decisions very much affected by cycle time
  - Small, lower associativity; latency is critical
  - Tag store and data store 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

- Serial vs. Parallel access of levels
  - Serial: Second level cache accessed only if first-level misses
  - Second level does not see the same accesses as the first
    - First level acts as a filter (filters some temporal and spatial locality)
    - Management policies are therefore different

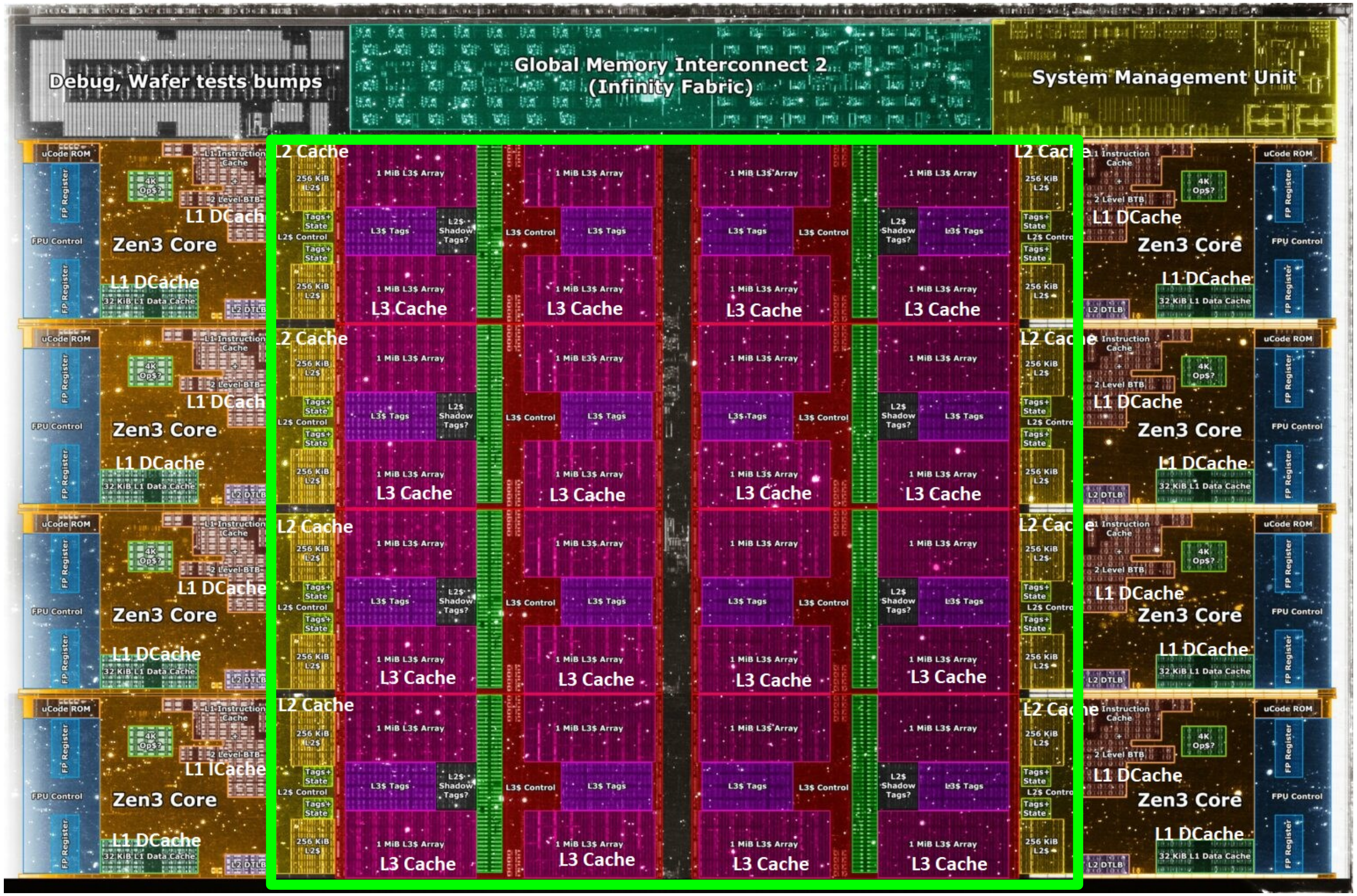
Deeper and Larger Cache Hierarchies



Apple M1, 2021

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

Deeper and Larger Cache Hierarchies



Core Count:

8 cores/16 threads

L1 Caches:

32 KB per core

L2 Caches:

512 KB per core

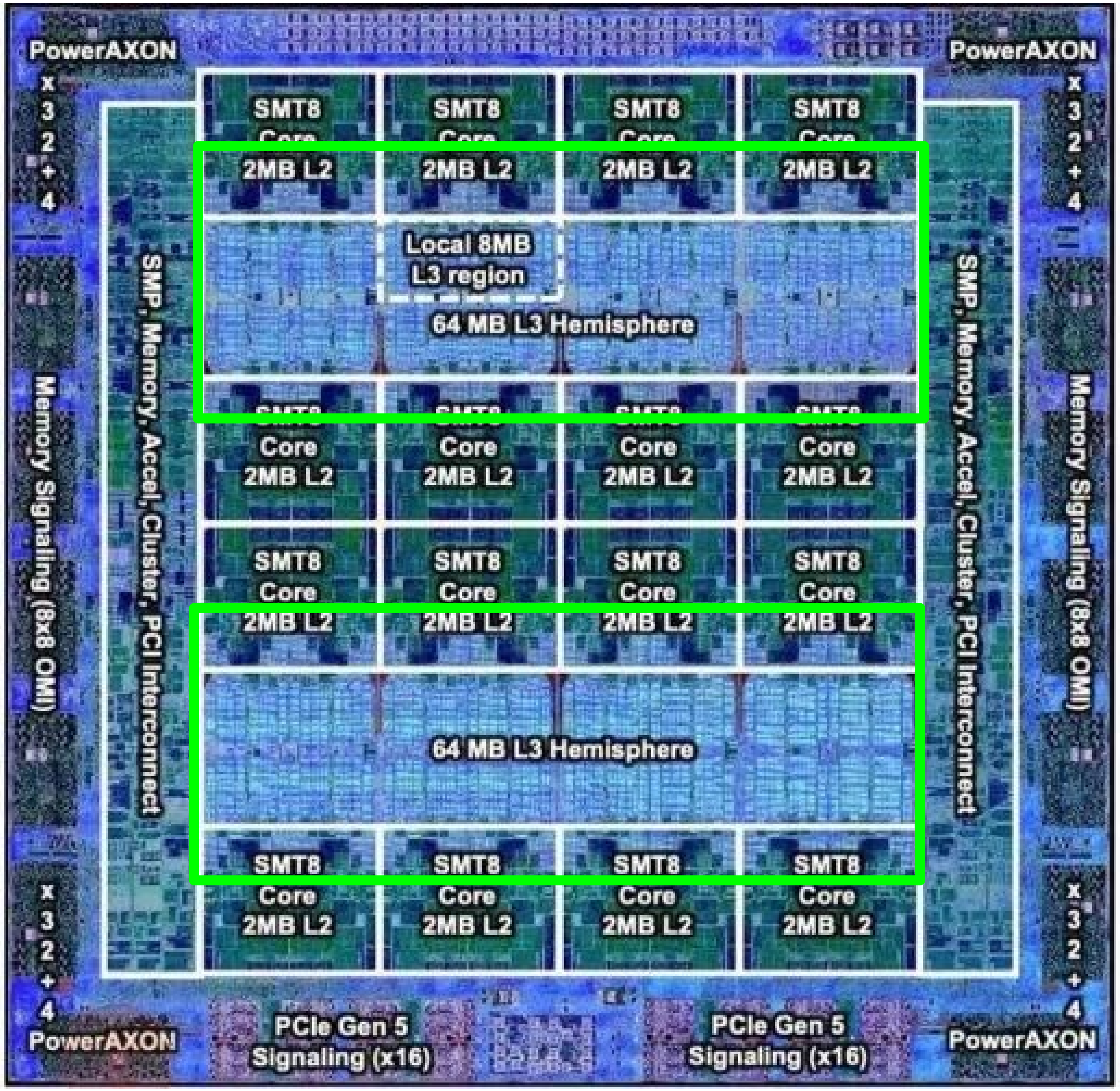
L3 Cache:

32 MB shared

AMD Ryzen 5000, 2020

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

Deeper and Larger Cache Hierarchies


PowerAXON
PowerAXON
PowerAXON
PowerAXON
x 32 + 4
x 32 + 4
x 32 + 4
x 32 + 4
SMT8 Core
2MB L2
Local 8MB L3 region
64 MB L3 Hemisphere
64 MB L3 Hemisphere
SMP, Memory, Accel, Cluster, PCI Interconnect
SMP, Memory, Accel, Cluster, PCI Interconnect
Memory Signaling (8x8 OMI)
Memory Signaling (8x8 OMI)
PCIe Gen 5 Signaling (x16)
PCIe Gen 5 Signaling (x16)

IBM POWER10, 2020

Cores:

15-16 cores,
8 threads/core

L2 Caches:

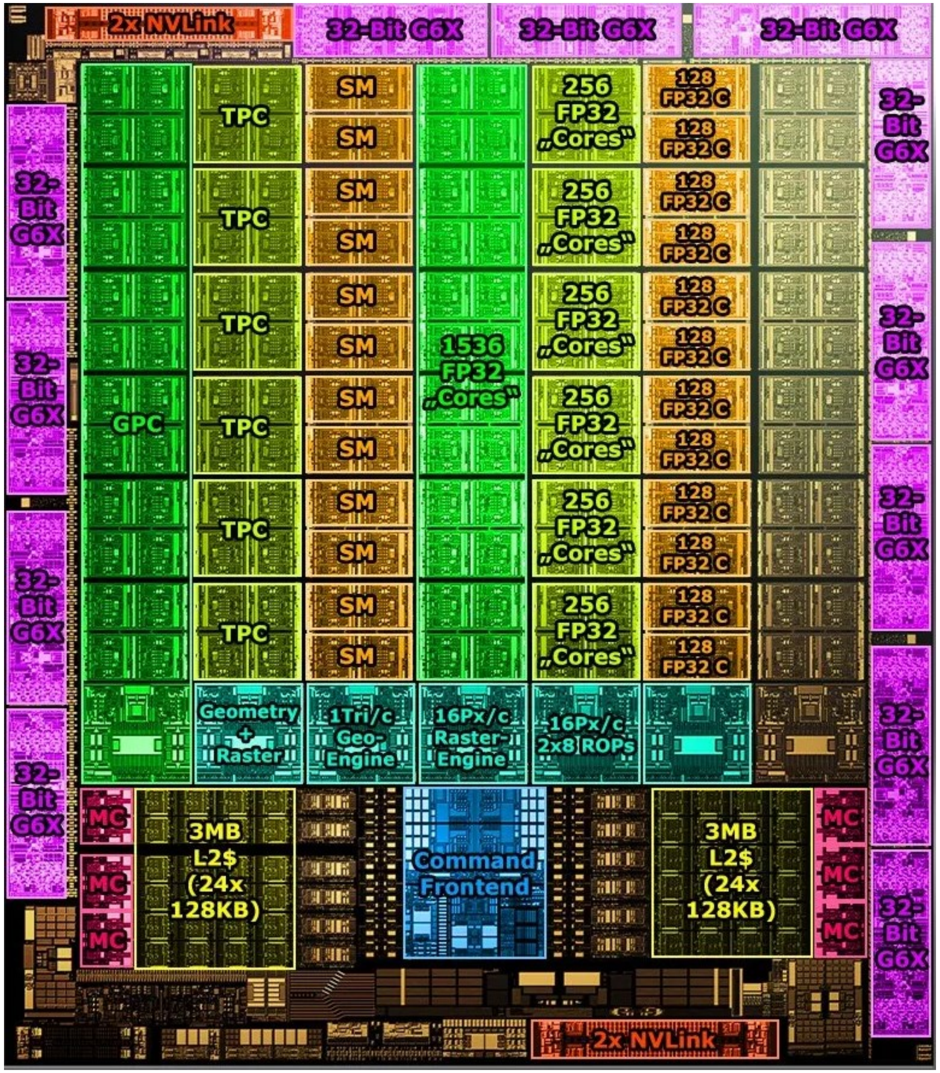
2 MB per core

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

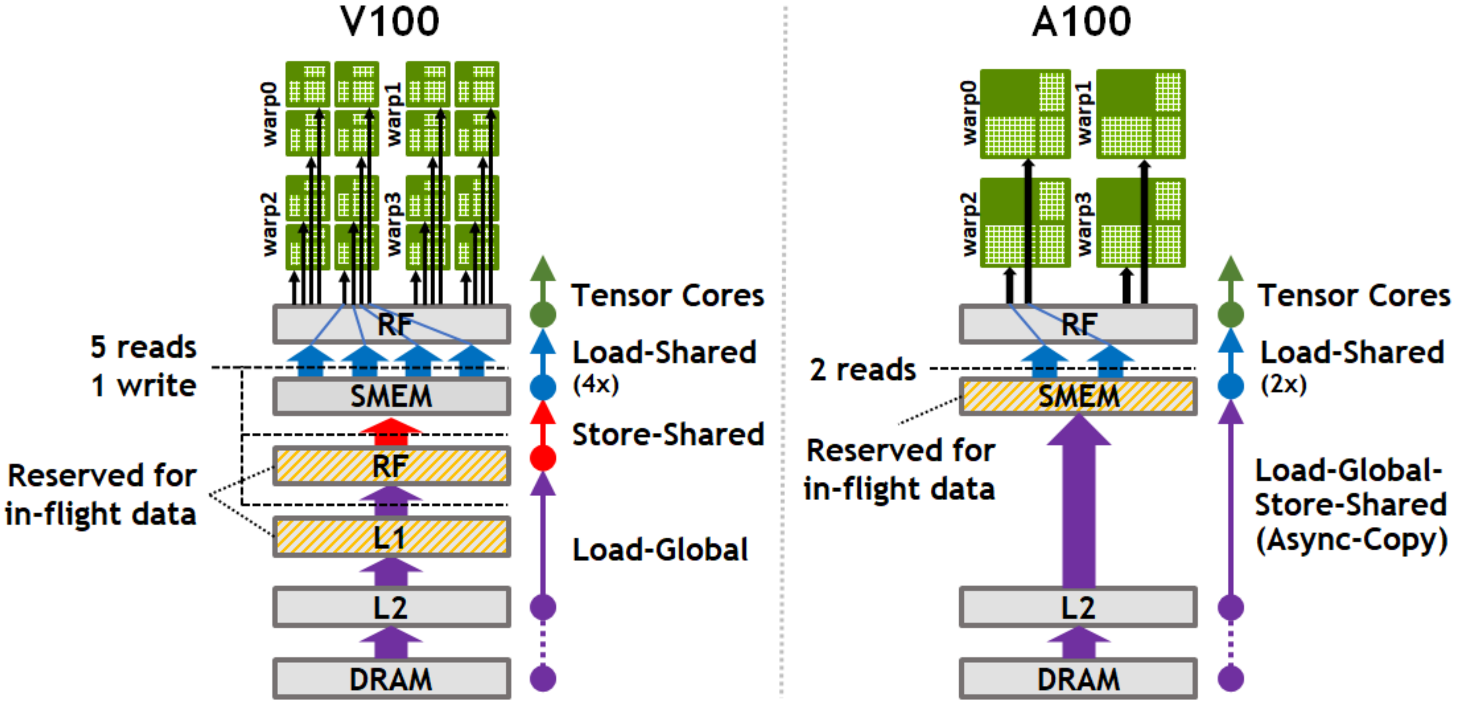
L2 Cache:

40 MB shared

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

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

Cache Performance

Cache Parameters vs. Miss/Hit Rate

- Cache size

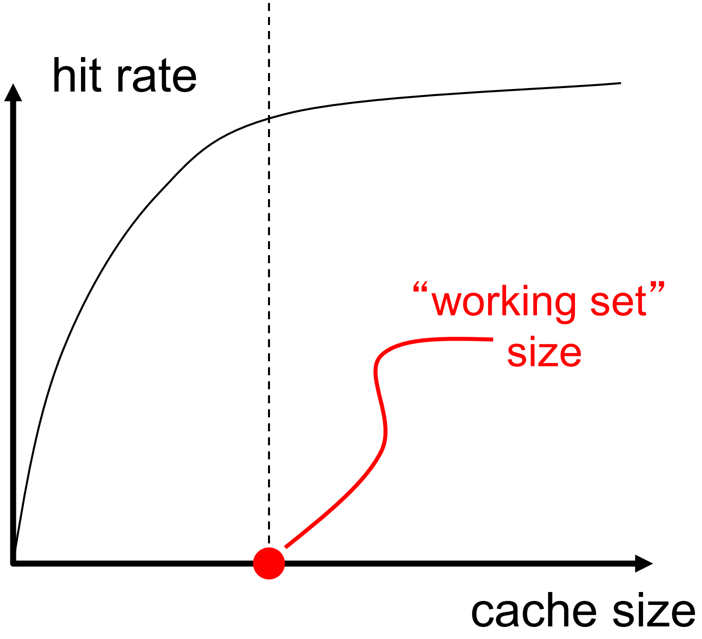
- Block size

- Associativity

- Replacement policy
- Insertion/Placement policy
- Promotion Policy

Cache Size

- Cache size: total data (not including tag) capacity
  - bigger can exploit temporal locality better
  - not ALWAYS better
- Too large a cache adversely affects hit and miss latency
  - smaller is faster => bigger is slower
  - access time may degrade critical path
- Too small a cache
  - doesn't exploit temporal locality well
  - useful data replaced often
- Working set: the whole set of data the executing application references
  - Within a time interval

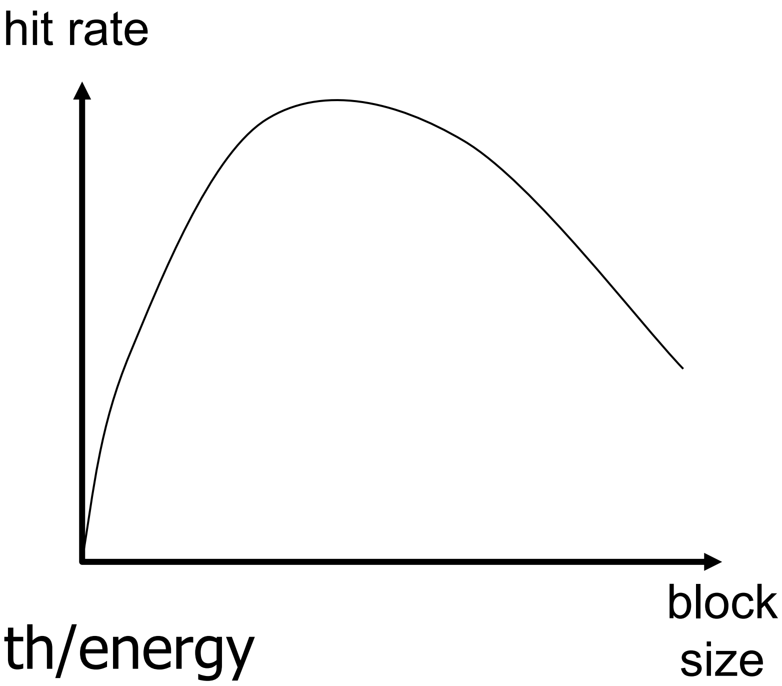


Block Size

- 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
  - don’t exploit spatial locality well
  - have larger tag overhead

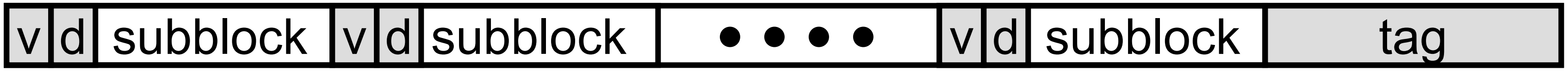
- Too large blocks
  - too few total # of blocks → less temporal locality exploitation
  - waste of cache space and bandwidth/energy if spatial locality is not high



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?

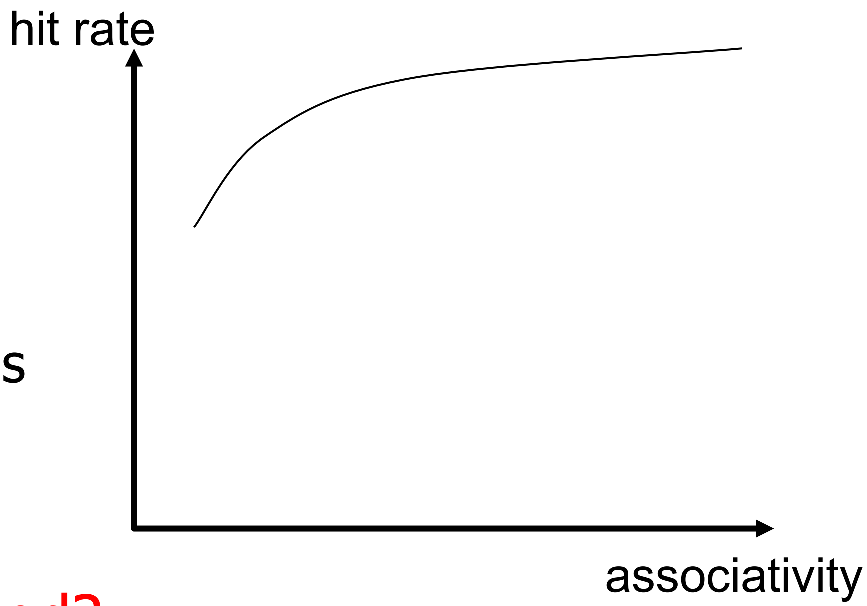


v d subblock v d subblock • • • • v d subblock tag

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 (plus diminishing returns)

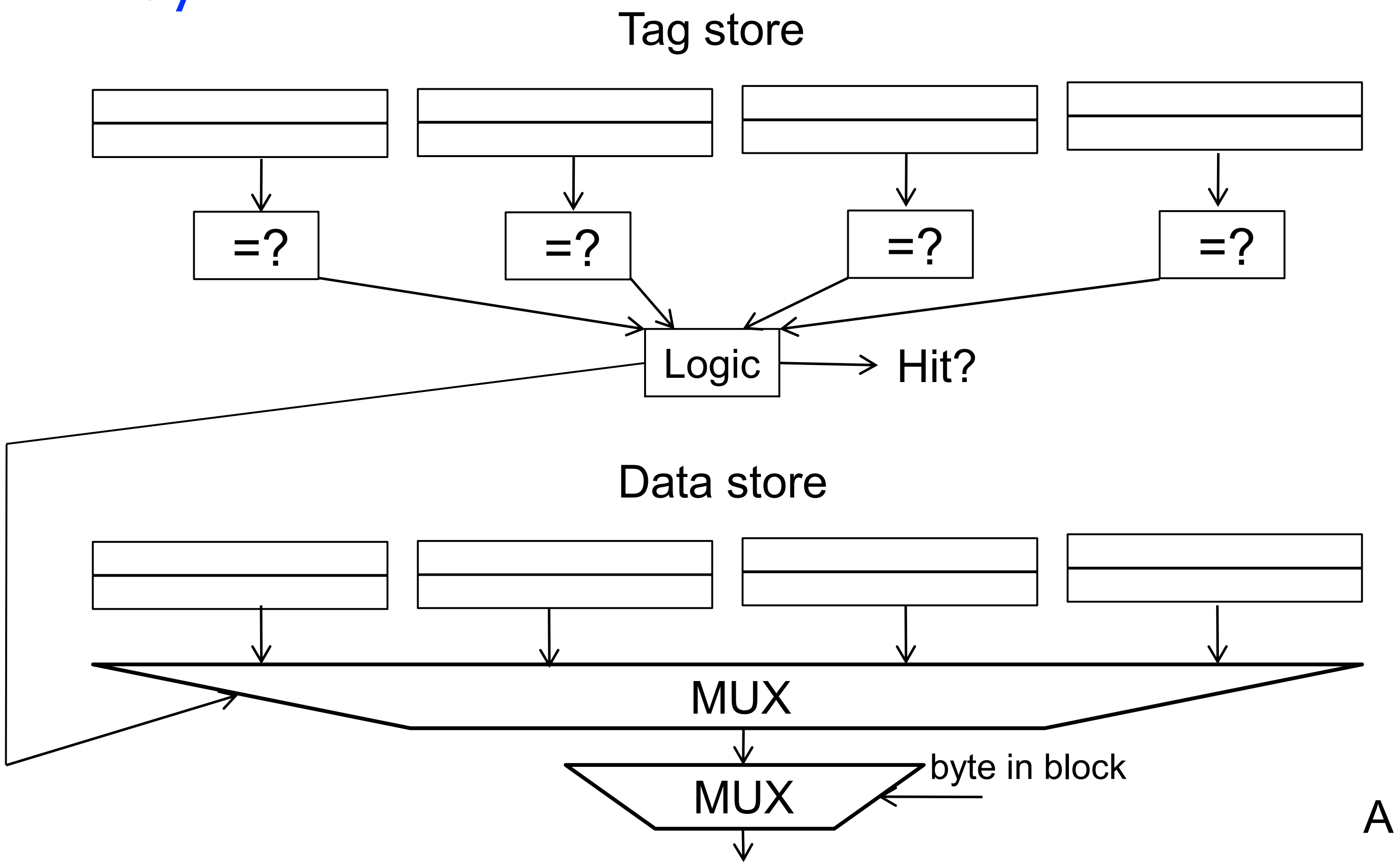


- Smaller associativity
  - lower cost
  - lower hit latency
    - Especially important for L1 caches

- Is power of 2 associativity required?

Recall: Higher Associativity (4-way)

- 4-way

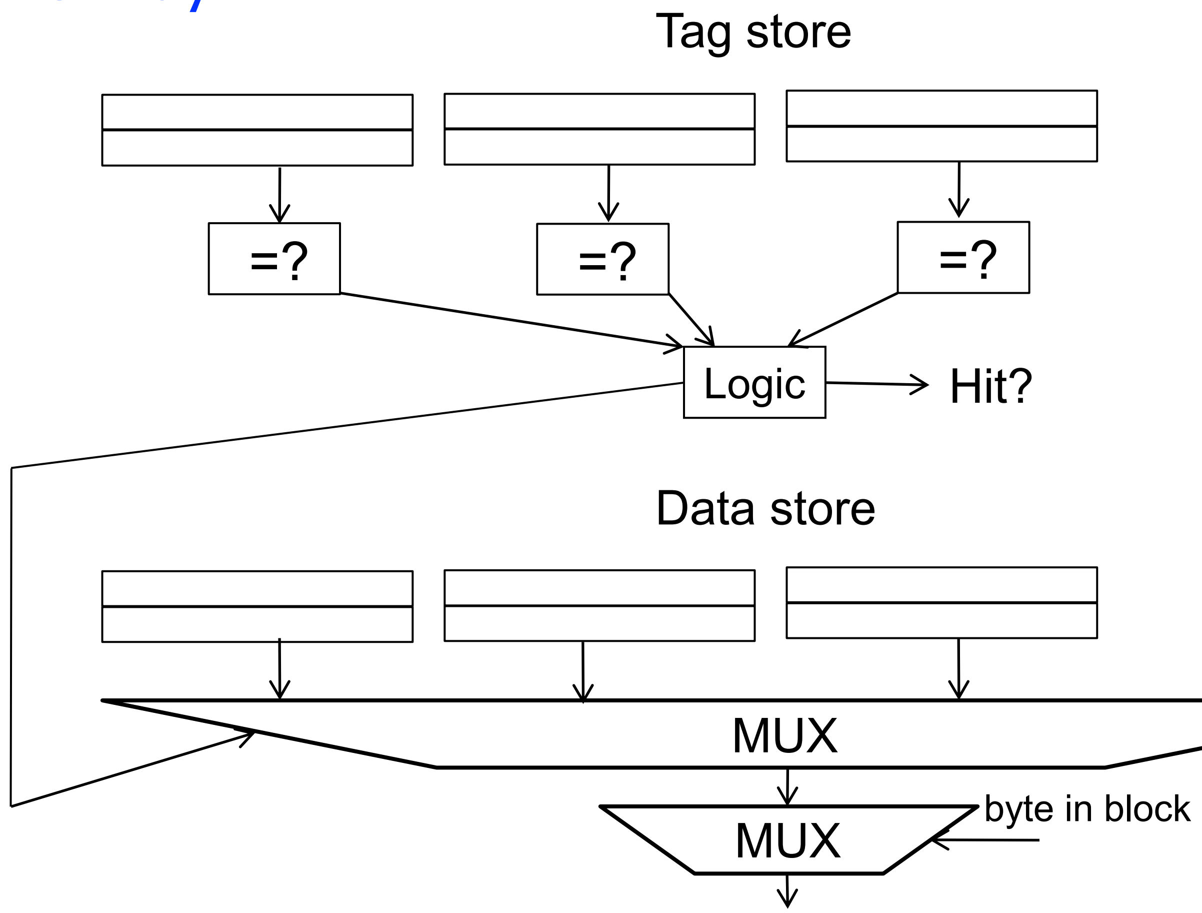


Address

tag	index	byte in block
4 bits	1 b	3 bits

Higher Associativity (3-way)

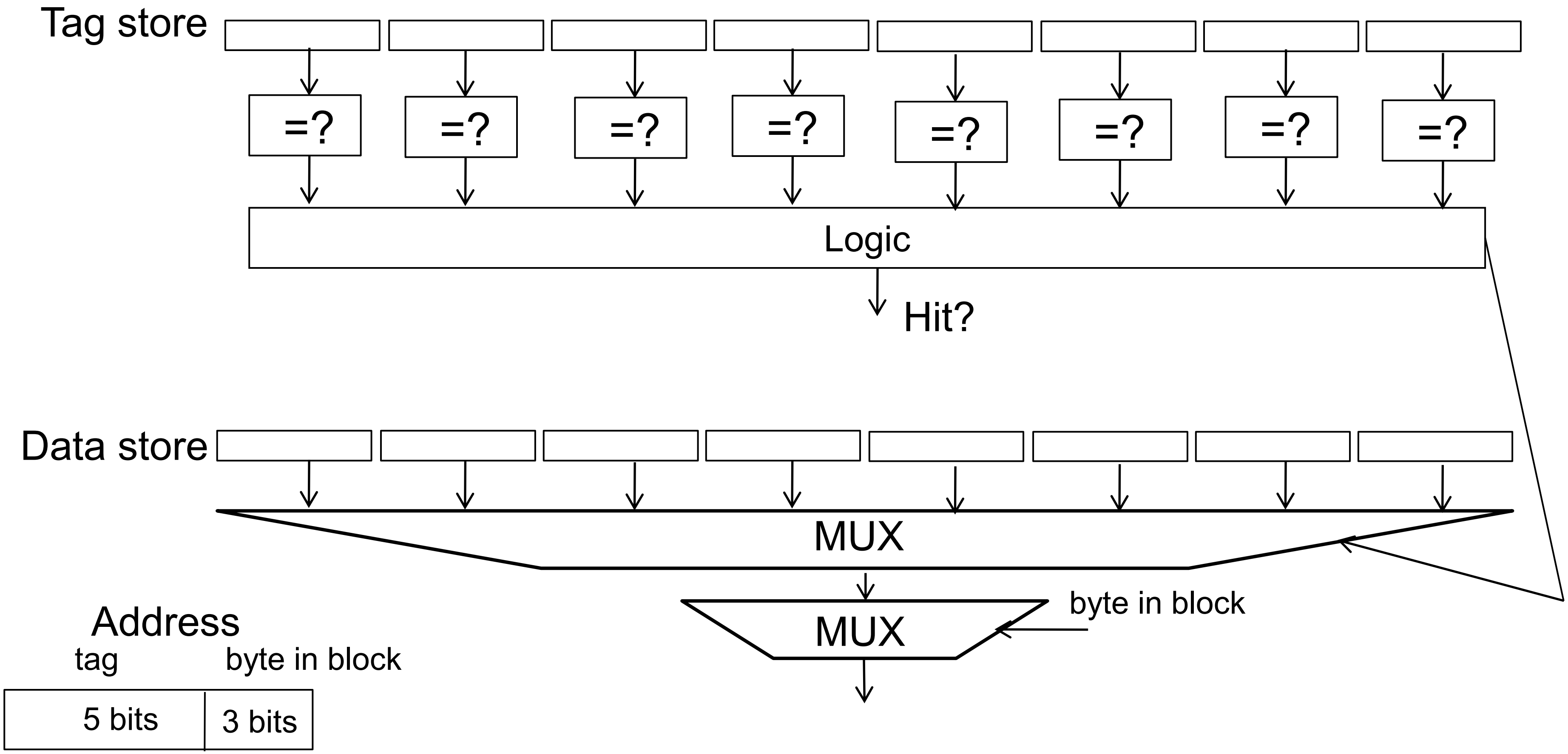
- 3-way



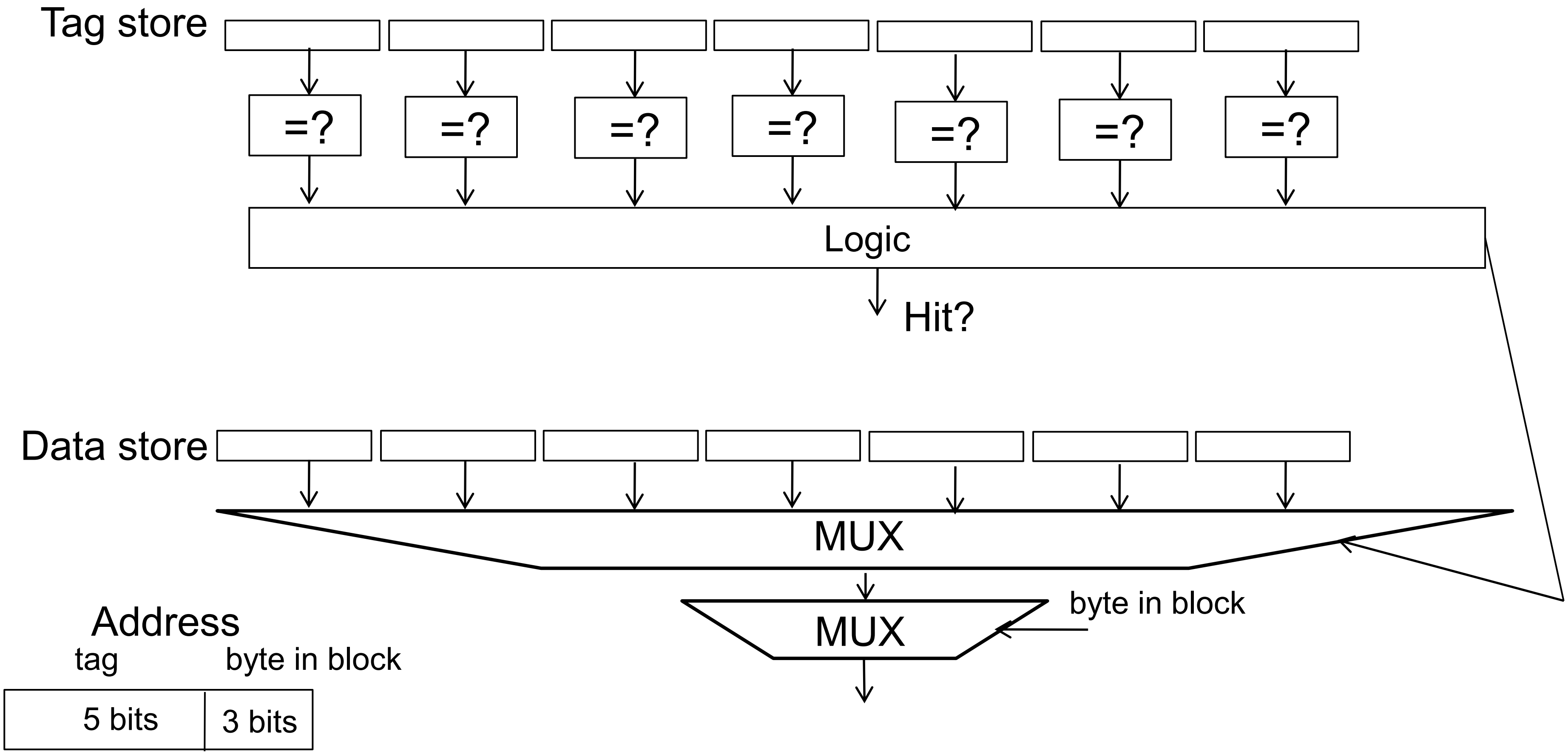
Address

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 should hit unless the cache block is displaced for the reasons below
- Capacity miss
  - cache is too small to hold everything needed
  - 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 cannot help
  - Prefetching can: 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
    - Software hints?
- 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)

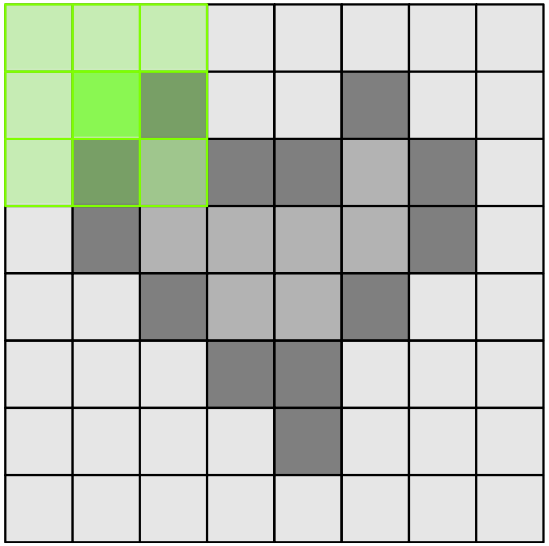
- Blocking
  - Divide loops operating on arrays into computation chunks so that each chunk can hold its data in the cache
  - Avoids cache conflicts between different chunks of computation
  - Essentially: Divide the working set so that each piece fits in the cache

- 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

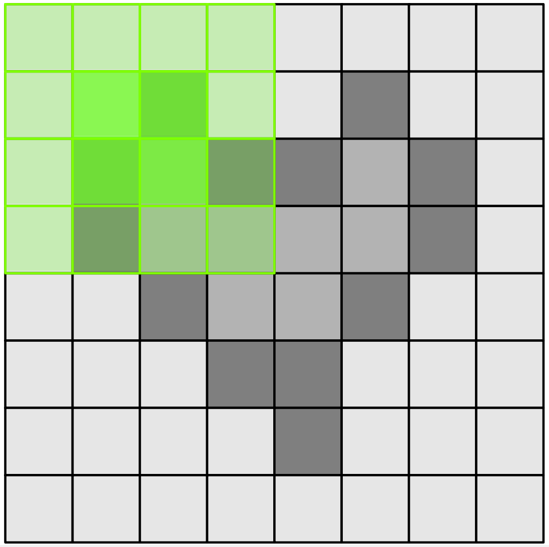


```
for (int i = 0; i < 3; i++){
    for (int j = 0; j < 3; j++){
        sum += gauss[i][j] * Image[(i+row-1)*width + (j+col-1)];
    }
}
```

Lecture 22: GPU Programming (Spring 2018) https://www.youtube.com/watch?v=y40-tY5WJ8A

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)

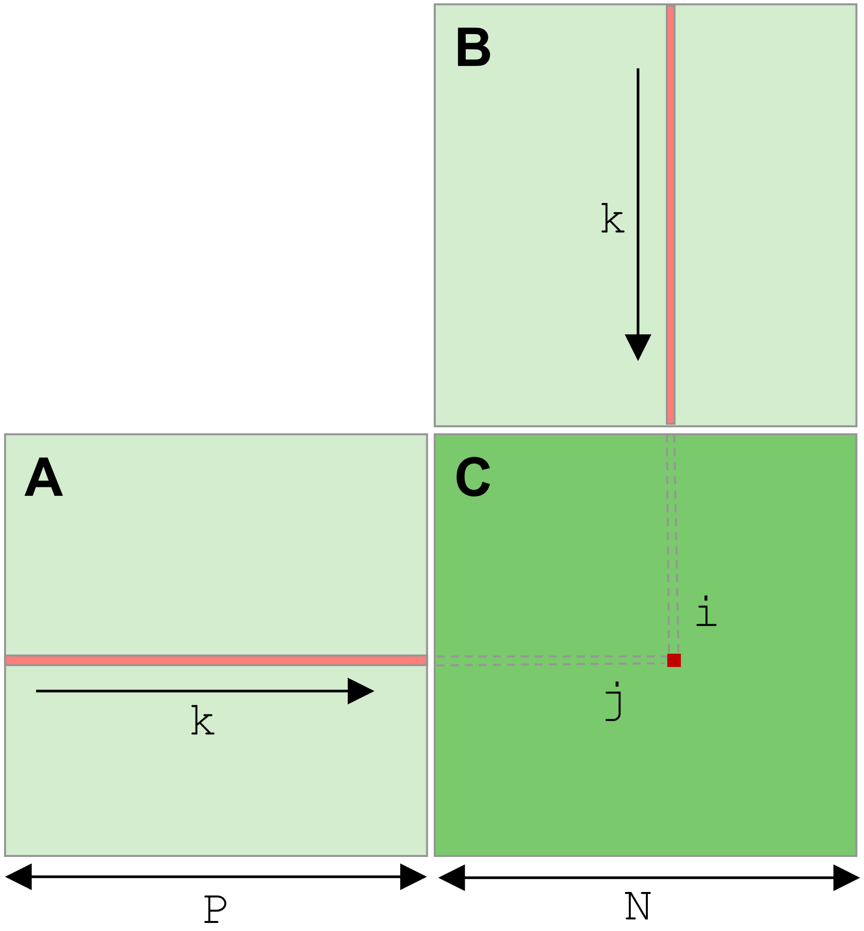


```
__shared__ int l_data[(L_SIZE+2)*(L_SIZE+2)];
…
Load tile into shared memory
__syncthreads();
for (int i = 0; i < 3; i++){
  for (int j = 0; j < 3; j++){
    sum += gauss[i][j] * l_data[(i+l_row-1)*(L_SIZE+2)+j+l_col-1];
  }
}
```

Lecture 22: GPU Programming (Spring 2018) https://www.youtube.com/watch?v=y40-tY5WJ8A

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

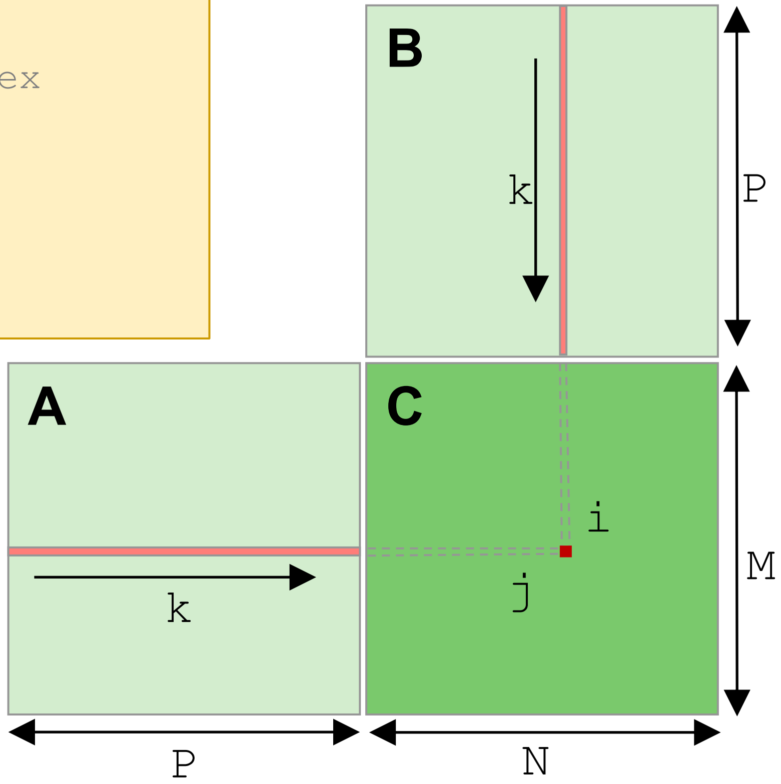


Naïve Matrix Multiplication (II)

- Naïve implementation of matrix multiplication has poor cache locality

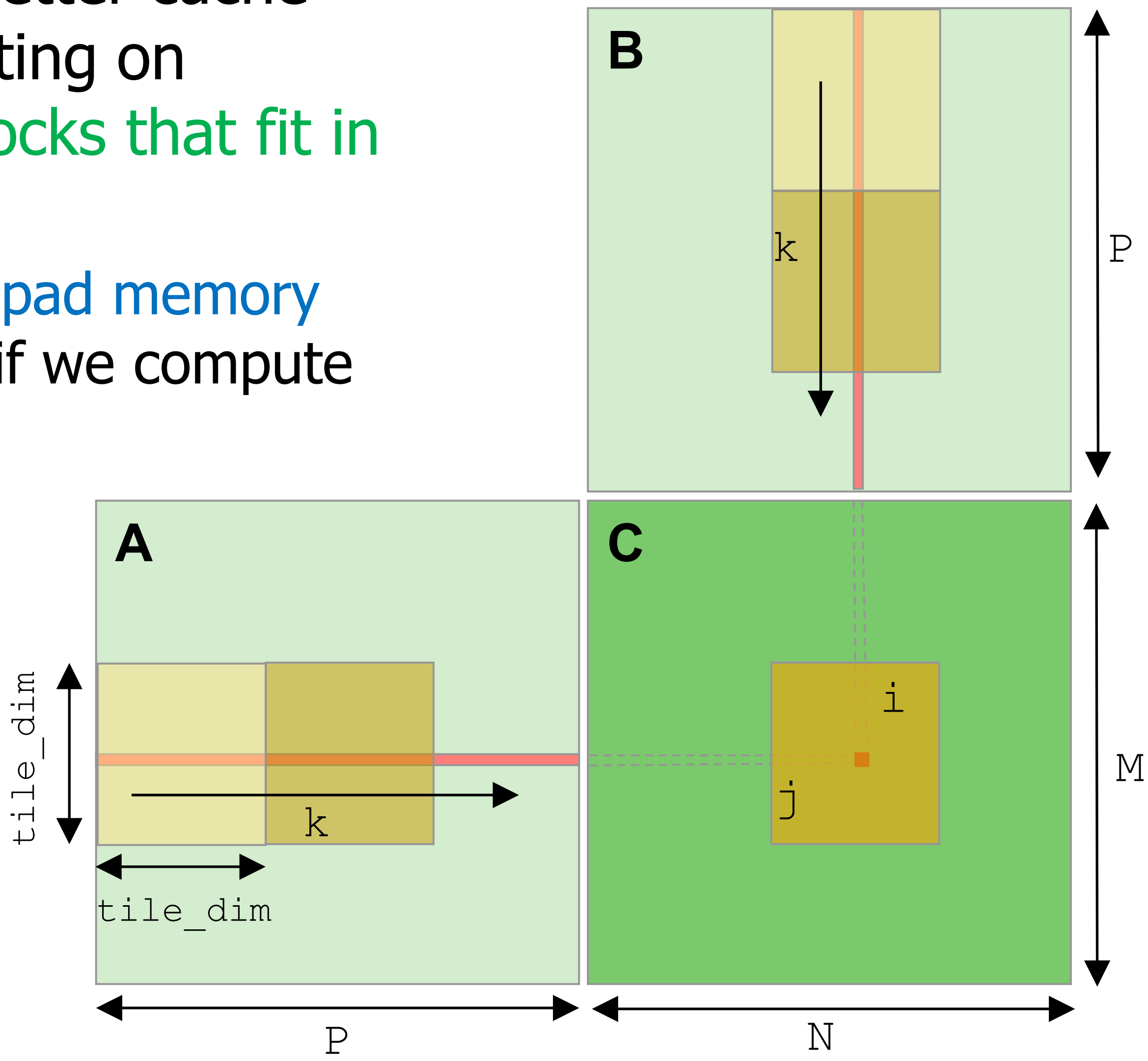
```
#define A(i,j) matrix_A[i * P + j]
#define B(i,j) matrix_B[i * N + j]
#define C(i,j) matrix_C[i * N + j]

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



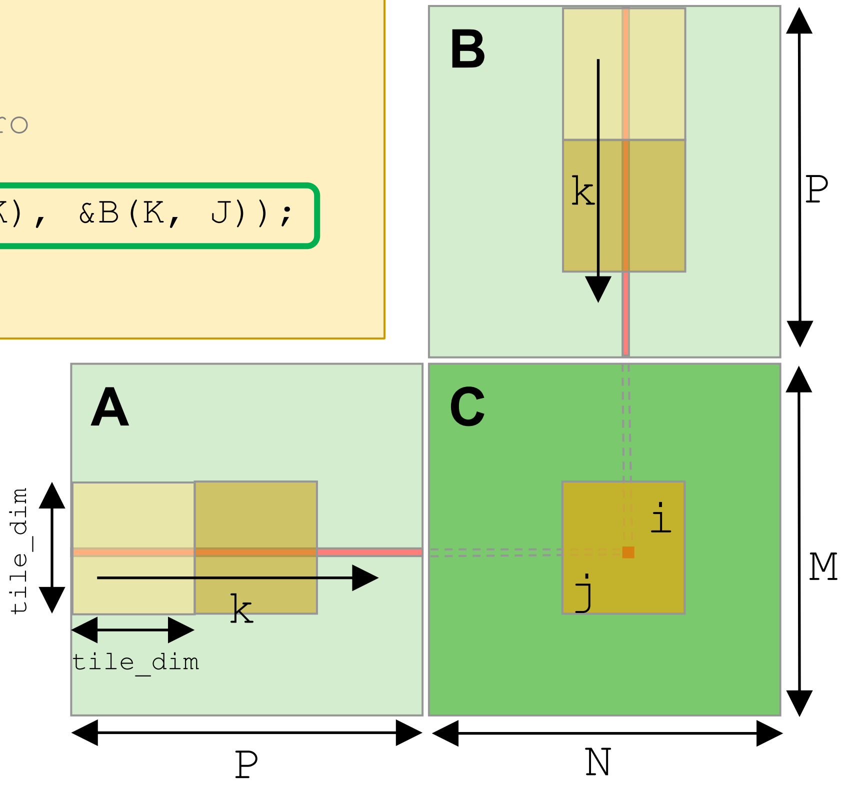
Lam+, "The cache performance and optimizations of blocked algorithms," ASPLOS 1991. https://doi.org/10.1145/106972.106981
Bansal+, "Chapter 15 - Fast Matrix Computations on Heterogeneous Streams," in "High Performance Parallelism Pearls", 2015. https://doi.org/10.1016/B978-0-12-803819-2.00011-2
Kirk & 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)

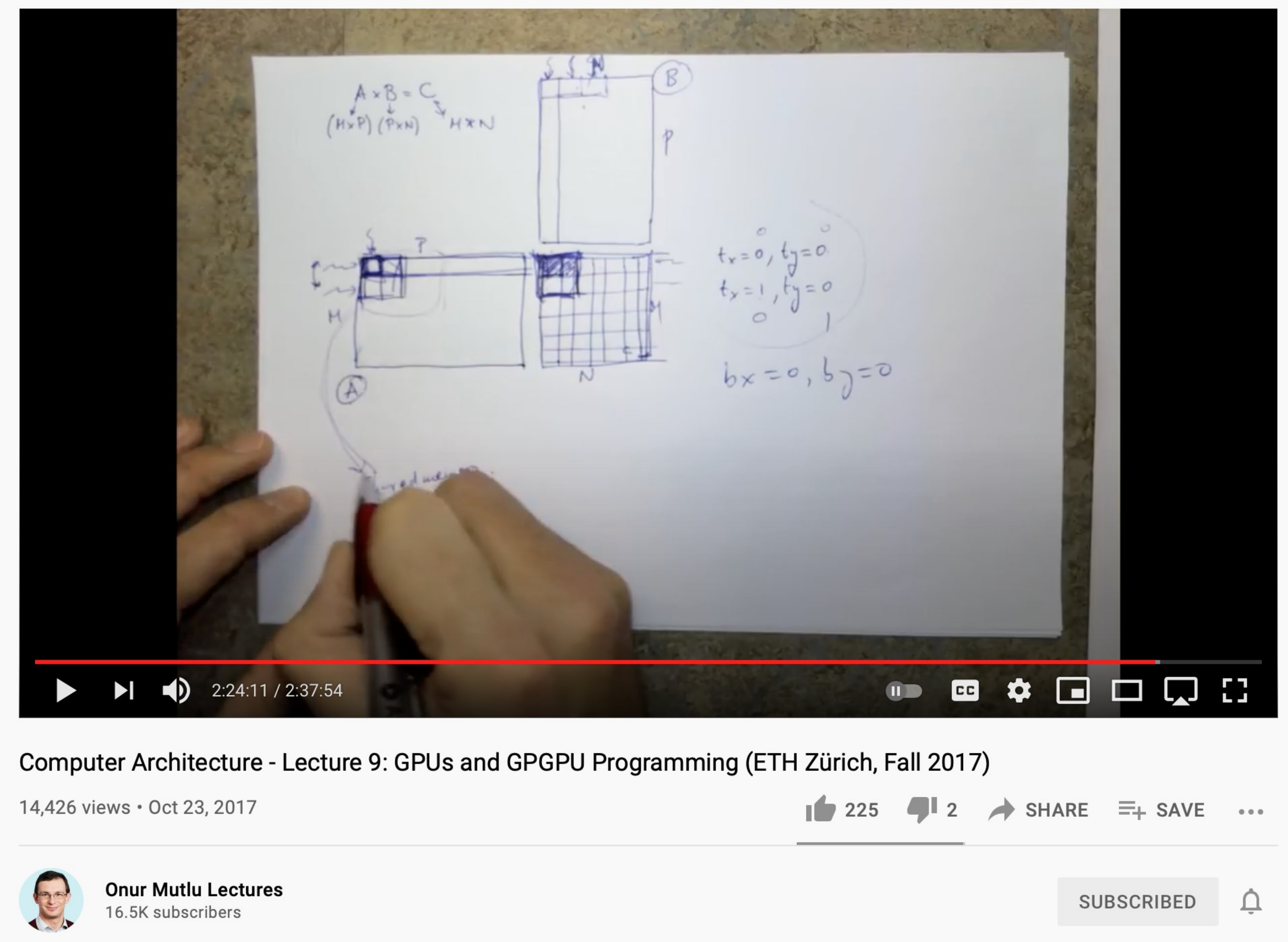
```
#define A(i,j) matrix_A[i * P + j]
#define B(i,j) matrix_B[i * N + j]
#define C(i,j) matrix_C[i * N + j]

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

Onur Mutlu Lectures

16.5K subscribers

Computer Architecture - Lecture 9: GPUs and GPGPU Programming (Fall 2017) https://youtu.be/mgtlbEqn2dA?t=8157

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;
}
```

- Pointer based traversal (e.g., of a linked list)
- Assume a huge linked list (1B nodes) and unique keys
- Why does the code on the left have poor cache hit rate?
    - "Other fields" occupy most of the cache line even though 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 frequently-used fields of a data structure and pack them into a separate data structure

- Who should do this?
  - Programmer
  - Compiler
    - Profiling vs. dynamic
  - Hardware?
  - Who can determine what is frequently used?

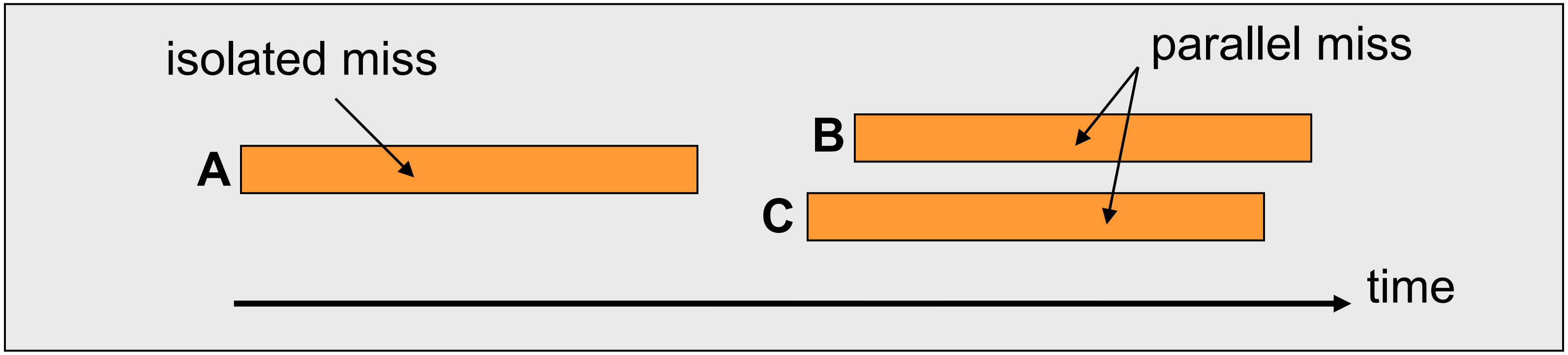
Improving Basic Cache Performance

- 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?
    - Local vs. remote memory
    - What level of cache in the hierarchy?
    - Row hit versus row conflict
    - Queueing delays in the memory controller and the interconnect
    - …
  - How much does the miss stall the processor?
    - Is it overlapped with other latencies?
    - Is the data immediately needed?
    - Is the incoming block going to evict a longer-to-refetch block?
    - …

Memory Level Parallelism (MLP)



- Memory Level Parallelism (MLP) means generating and servicing multiple memory accesses in parallel [Glew' 98]
- Several techniques to 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

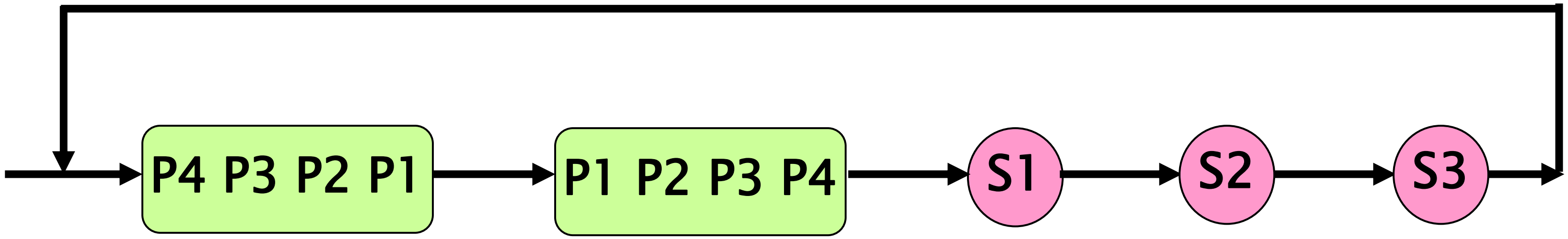
- Traditional cache replacement policies try to reduce miss count

- 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



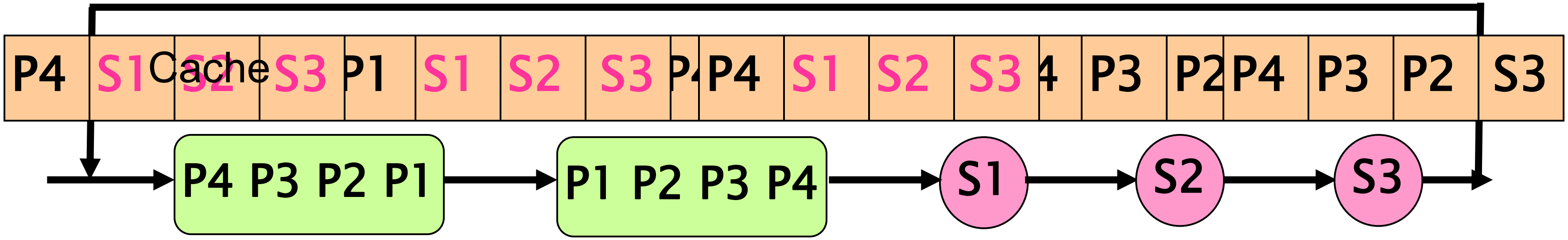
Misses to blocks P1, P2, P3, P4 can be parallel
Misses to blocks S1, S2, and S3 are isolated

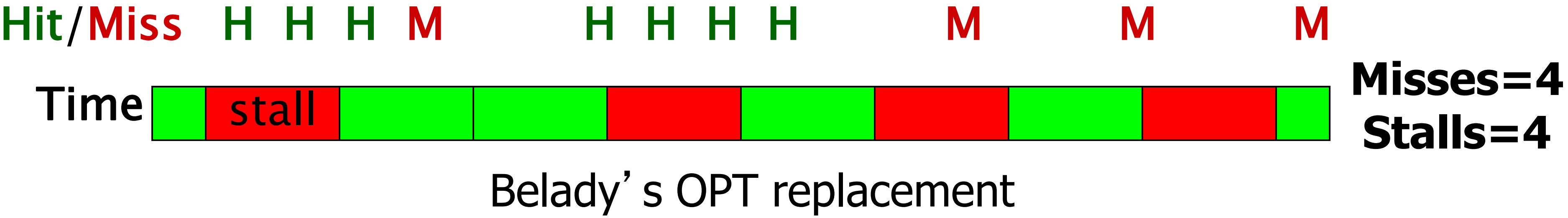
Two replacement algorithms:

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

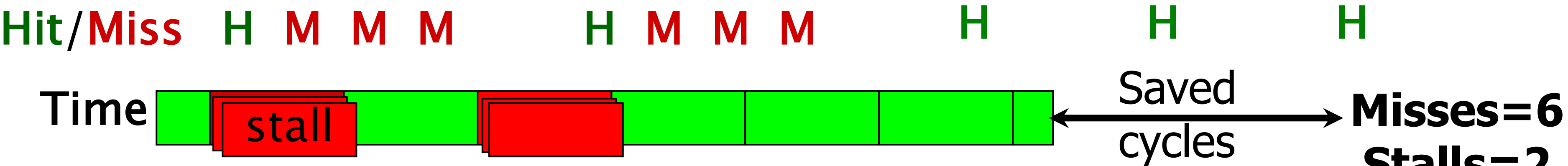
For a fully associative cache containing 4 blocks

Fewest Misses ≠ Best Performance





Belady’s OPT replacement

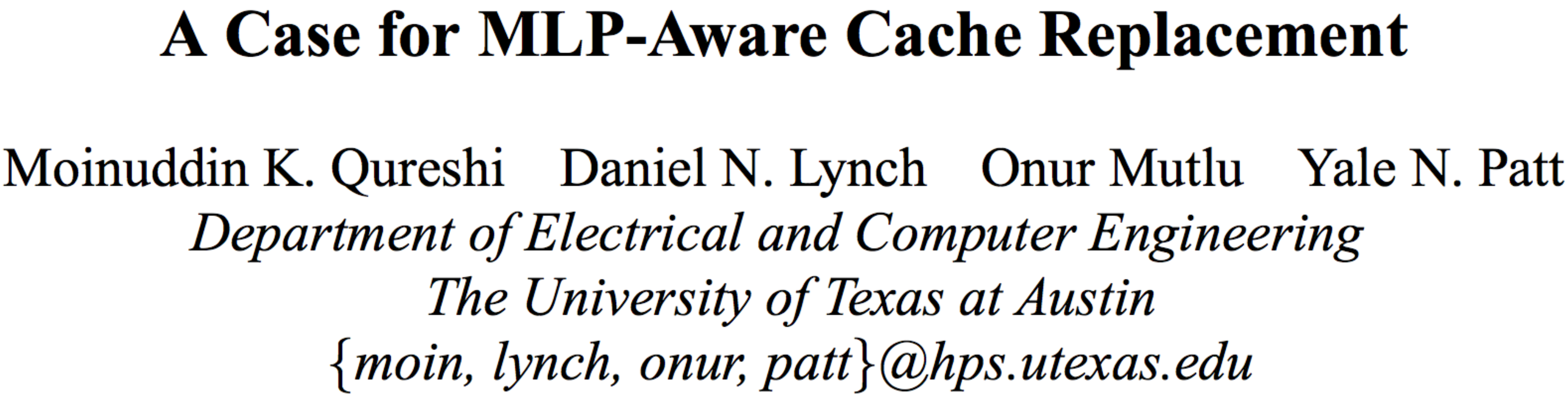


MLP-Aware replacement

Recommended: MLP-Aware Cache Replacement

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

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



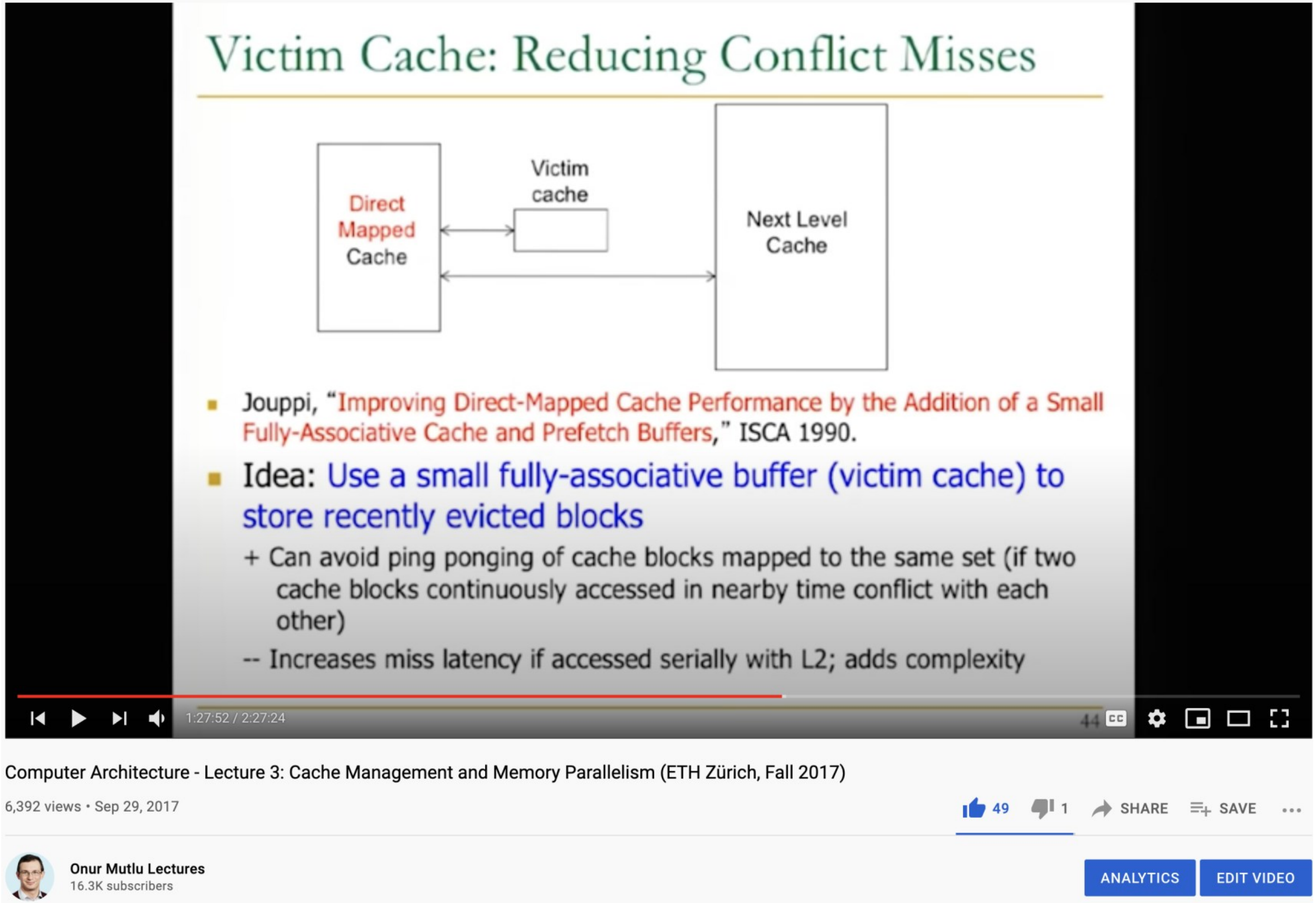
A Case for MLP-Aware Cache Replacement

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)



Victim Cache: Reducing Conflict Misses

- Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," ISCA 1990.
- Idea: Use a small fully-associative buffer (victim cache) to store recently evicted blocks

  + Can avoid ping ponging of cache blocks mapped to the same set (if two cache blocks continuously accessed in nearby time conflict with each other)

  -- Increases miss latency if accessed serially with L2; adds complexity

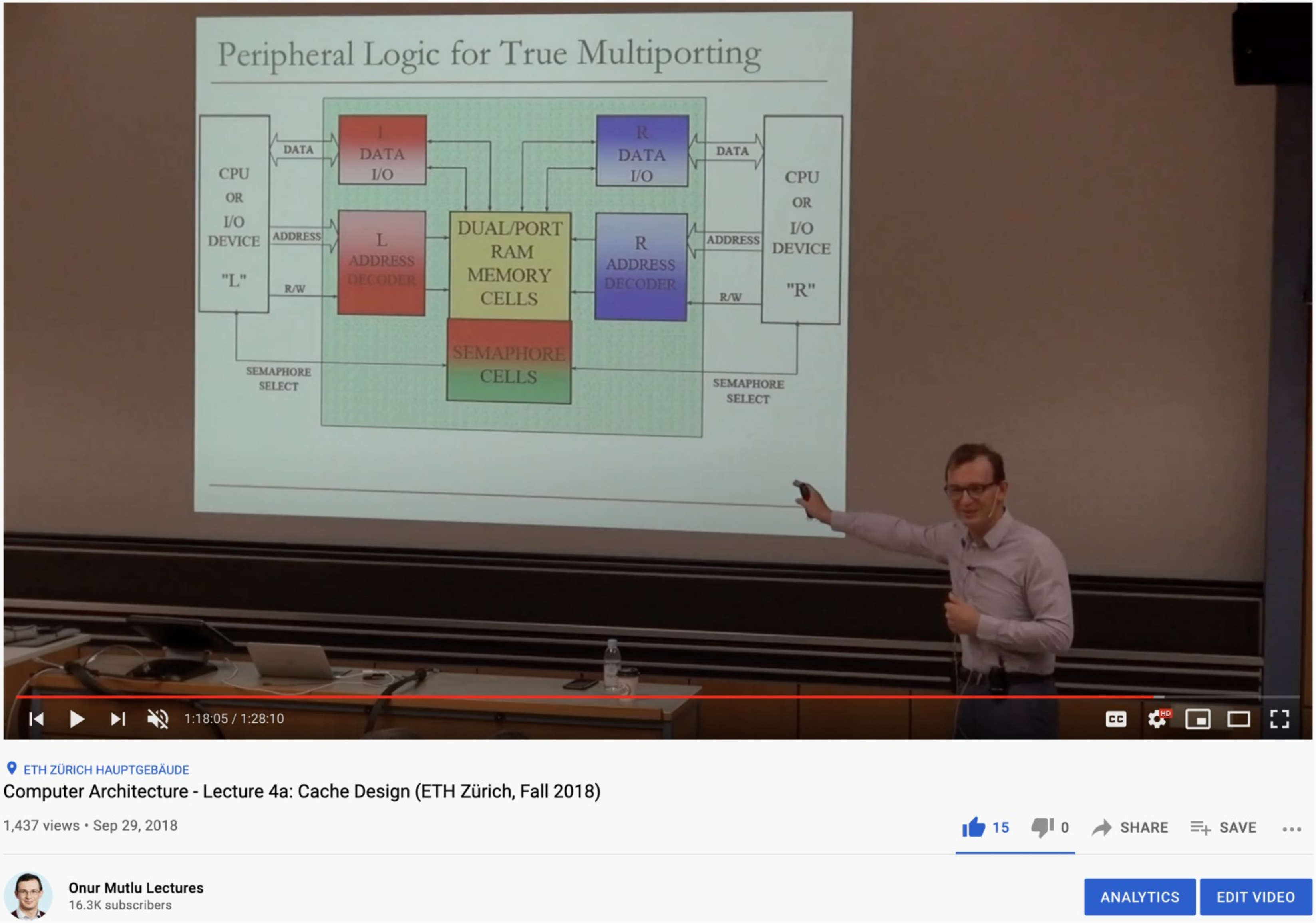
Computer Architecture - Lecture 3: Cache Management and Memory Parallelism (ETH Zürich, Fall 2017)

6,392 views • Sep 29, 2017

Onur Mutlu Lectures
16.3K subscribers

https://www.youtube.com/watch?v=OyomXCHNJDA&list=PL5Q2soXY2Zi9OhoVQBXYFIZywZXCPI4M_&index=3

Lectures on Cache Optimizations (II)



Peripheral Logic for True Multiporting

ETH ZÜRICH HAUPTGEBÄUDE

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

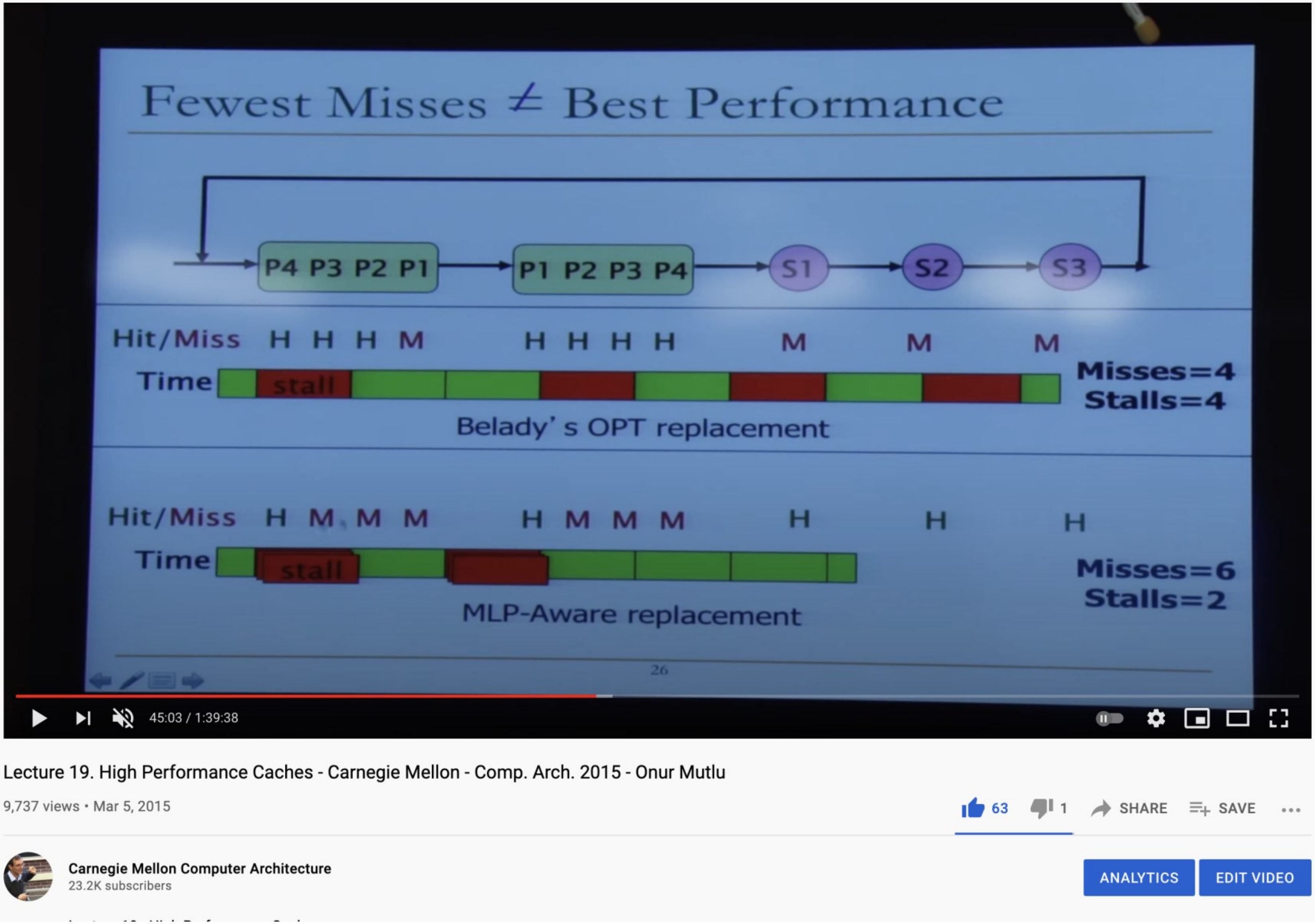
1,437 views • Sep 29, 2018

Onur Mutlu Lectures

16.3K subscribers

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

Lectures on Cache Optimizations (III)



Fewest Misses ≠ Best Performance

P4 P3 P2 P1 → P1 P2 P3 P4 → S1 → S2 → S3

Hit/Miss H H H M H H H H M M M

Time stall

Misses=4
Stalls=4

Belady's OPT replacement

Hit/Miss H M M M H M M M H H H

Time stall

Misses=6
Stalls=2

MLP-Aware replacement

26

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

9,737 views • Mar 5, 2015

Carnegie Mellon Computer Architecture
23.2K subscribers

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

Lecture on Cache Optimizations

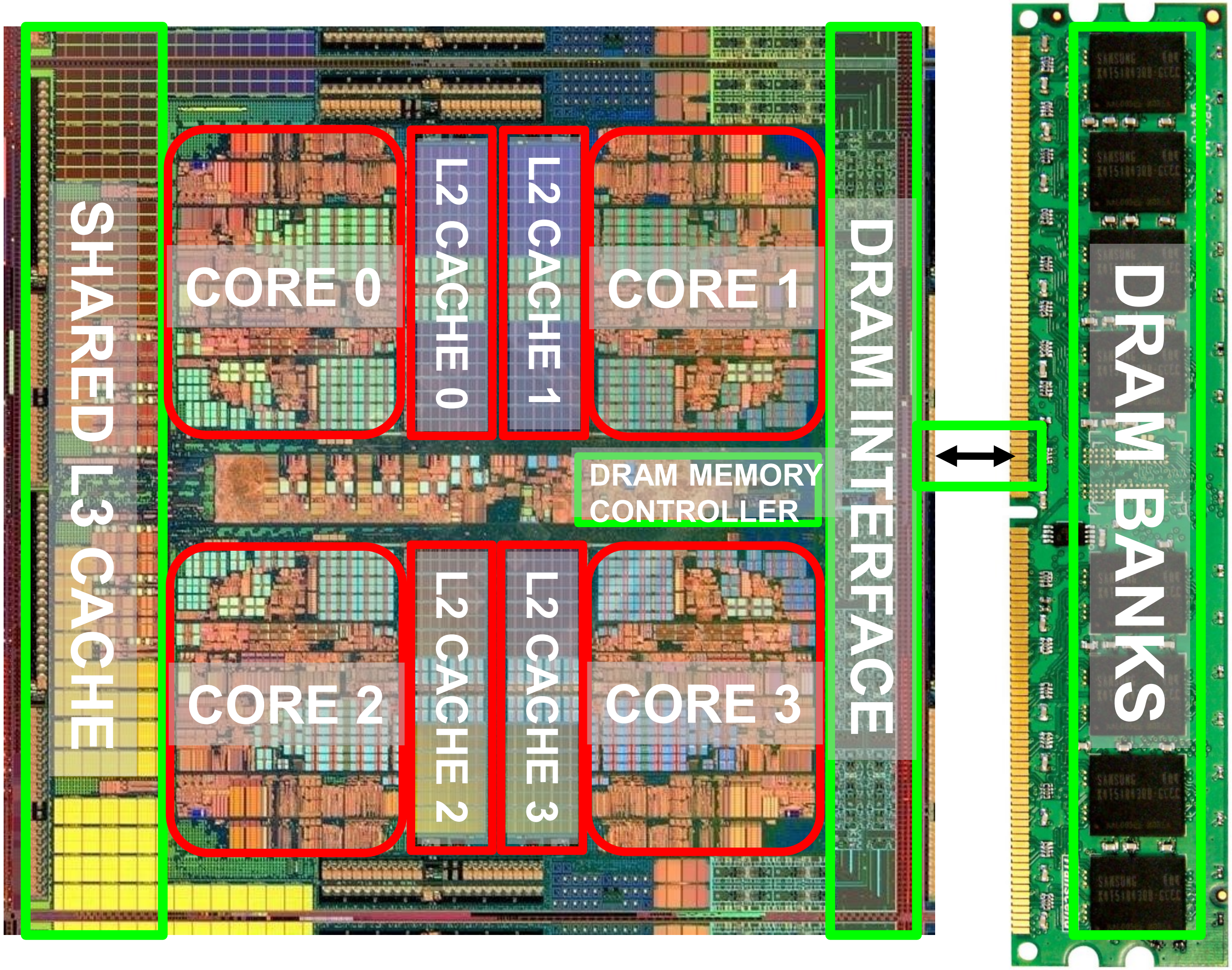
- Computer Architecture, Fall 2017, Lecture 3
  - Cache Management & Memory Parallelism (ETH, Fall 2017)
  - https://www.youtube.com/watch?v=OyomXCHNJDA&list=PL5Q2soXY2Zi9OhoVQBXYFIZywZXCPl4M_&index=3

- Computer Architecture, Fall 2018, Lecture 4a
  - Cache Design (ETH, Fall 2018)
  - https://www.youtube.com/watch?v=55oYBm9cifI&list=PL5Q2soXY2Zi9JXe3ywQMhylk_d5dI-TM7&index=6

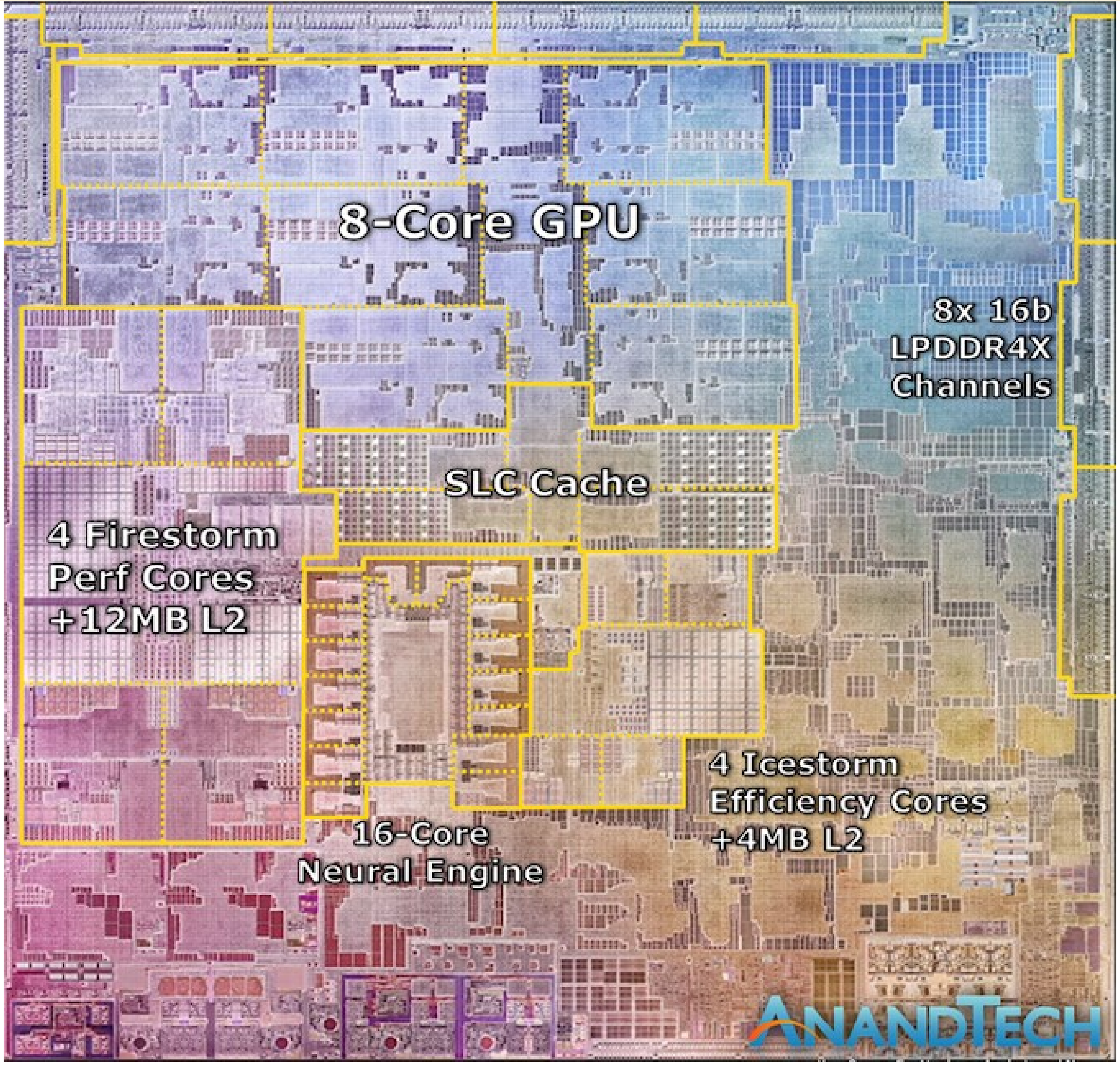
- Computer Architecture, Spring 2015, Lecture 19
  - High Performance Caches (CMU, Spring 2015)
  - https://www.youtube.com/watch?v=jDHx2K9HxlM&list=PL5PHm2jkkXmi5CxxI7b3JCL1TWybTDtKq&index=21

Multi-Core Issues in Caching

Caches in a Multi-Core System



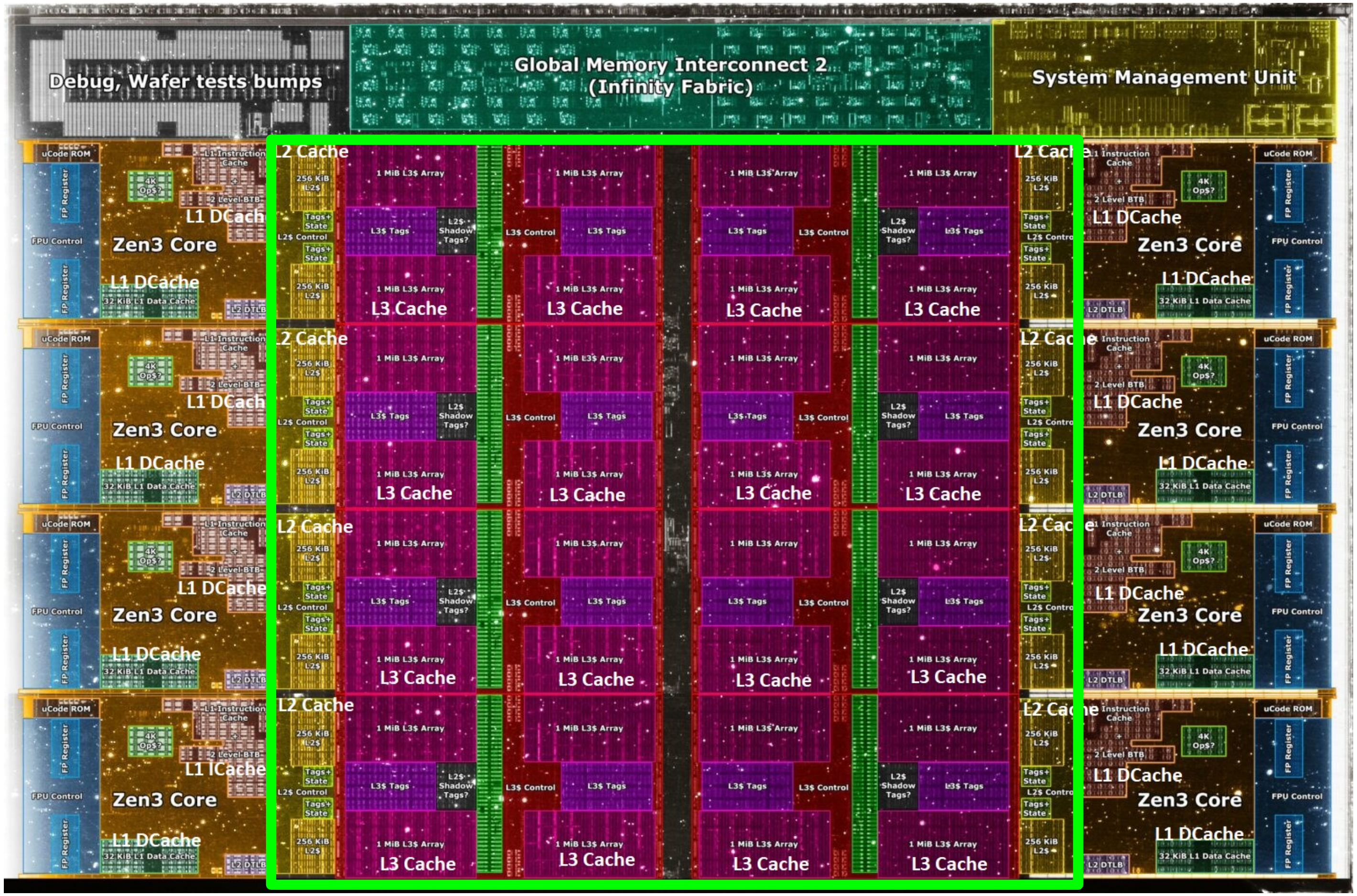
Caches in a Multi-Core System



Apple M1, 2021

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

Caches in a Multi-Core System



AMD Ryzen 5000, 2020

Core Count:

8 cores/16 threads

L1 Caches:

32 KB per core

L2 Caches:

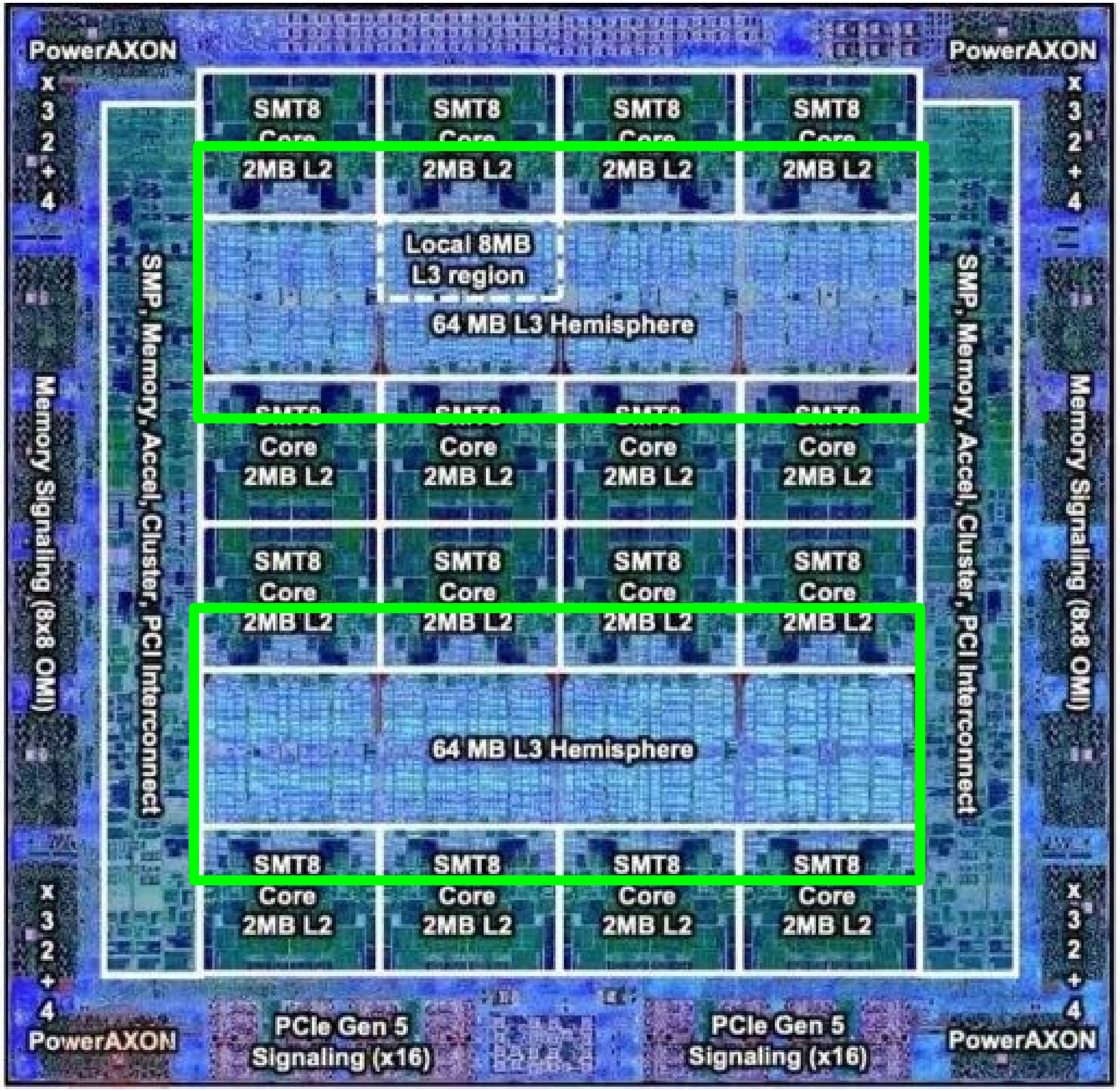
512 KB per core

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


PowerAXON
PowerAXON
x 32 + 4
x 32 + 4
SMT8 Core
2MB L2
Local 8MB L3 region
64 MB L3 Hemisphere
SMT8 Core
2MB L2
SMT8 Core
2MB L2
64 MB L3 Hemisphere
SMT8 Core
2MB L2
SMP, Memory, Accel, Cluster, PCI Interconnect
SMP, Memory, Accel, Cluster, PCI Interconnect
Memory Signaling (8x8 OMI)
Memory Signaling (8x8 OMI)
x 32 + 4
x 32 + 4
PowerAXON
PCIe Gen 5 Signaling (x16)
PCIe Gen 5 Signaling (x16)
PowerAXON

IBM POWER10, 2020

Cores:

15-16 cores,
8 threads/core

L2 Caches:

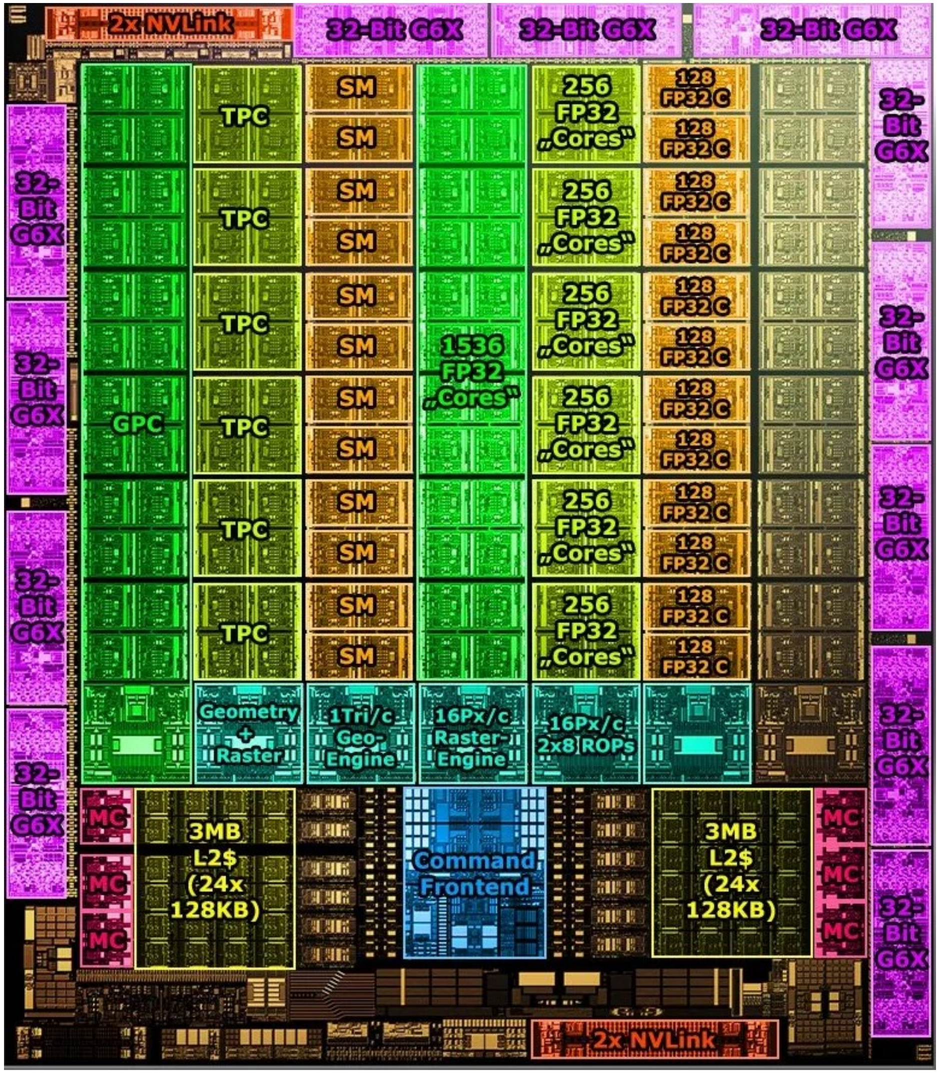
2 MB per core

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 Multi-Core Systems

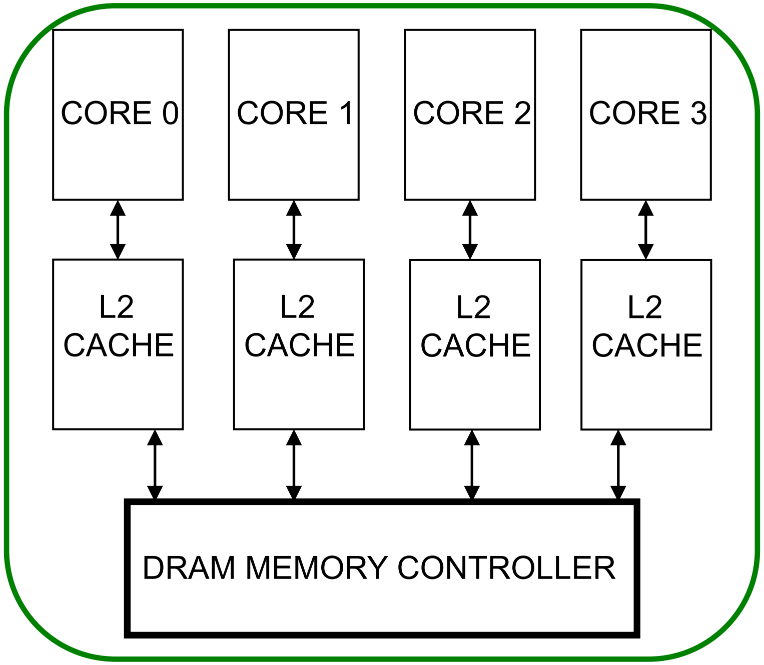
- Cache efficiency becomes even more important in a multi-core/multi-threaded system
  - Memory bandwidth is at premium
  - Cache space is a limited resource across cores/threads

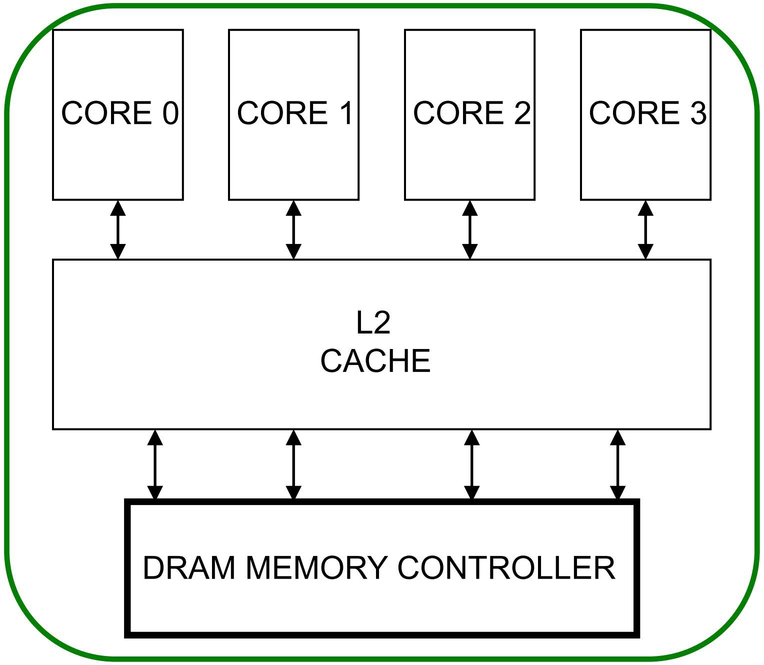
- How do we design the caches in a multi-core system?

- Many decisions
  - Shared vs. private caches
  - How to maximize performance of the entire system?
  - How to provide QoS 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
- 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

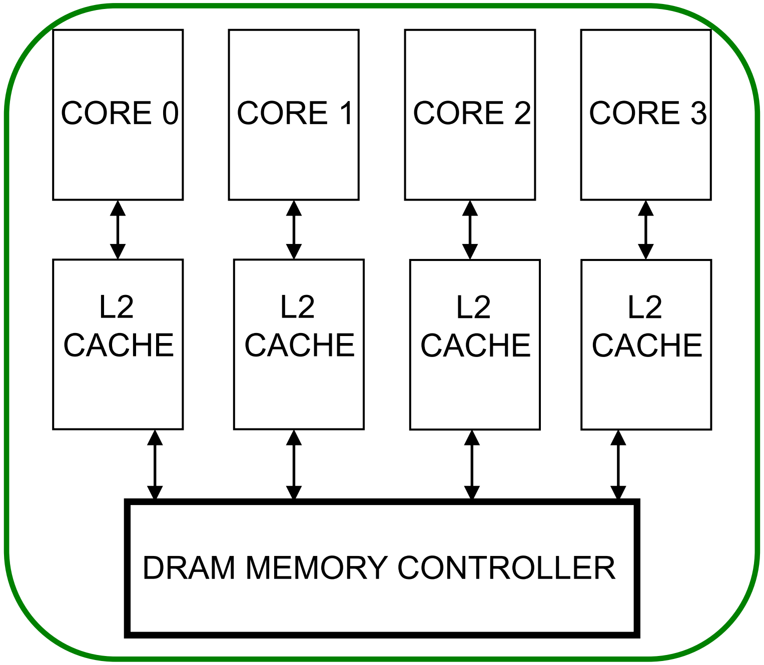
- Resource sharing results in contention for resources
  - When the resource is not idle, another thread cannot use it
  - If space is occupied by one thread, another thread needs to re-occupy it

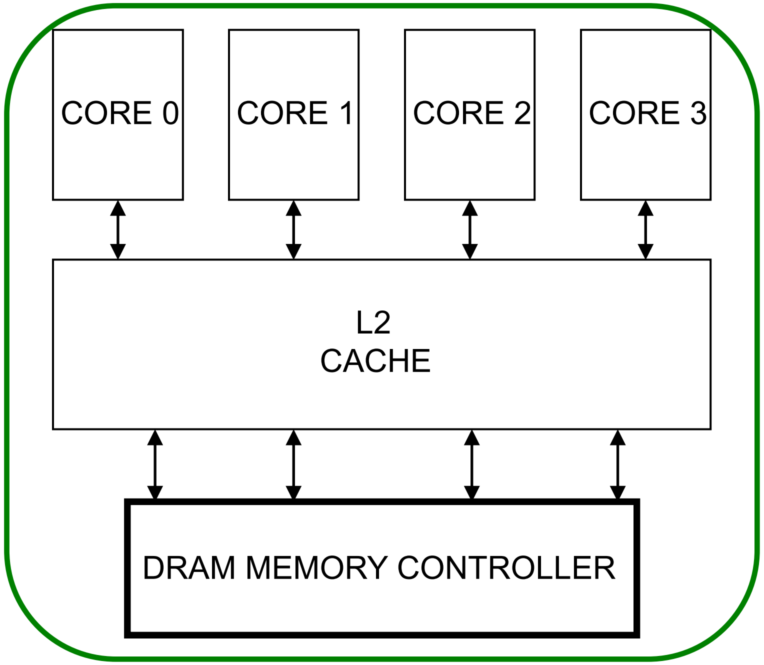
- Sometimes reduces each or some thread's performance
  - Thread performance can be worse than when it is run alone
- Eliminates performance isolation → inconsistent performance across runs
  - Thread performance depends on co-executing threads
- Uncontrolled (free-for-all) sharing degrades QoS
  - Causes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



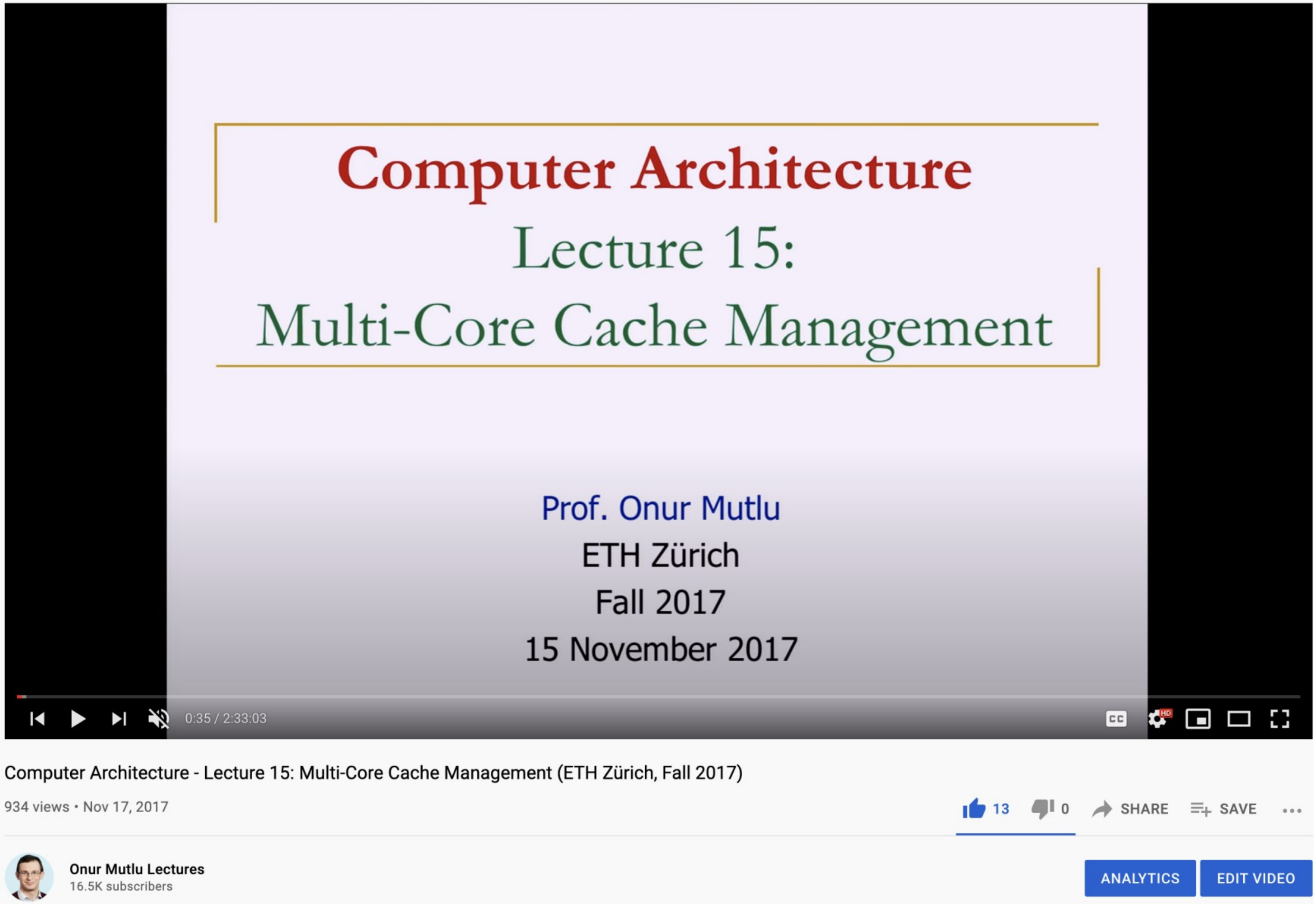


Shared Caches Between Cores

- 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 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?)

Lectures on Multi-Core Cache Management



Computer Architecture

Lecture 15:

Multi-Core Cache Management

Prof. Onur Mutlu

ETH Zürich

Fall 2017

15 November 2017

Computer Architecture - Lecture 15: Multi-Core Cache Management (ETH Zürich, Fall 2017)

934 views • Nov 17, 2017

Onur Mutlu Lectures

16.5K subscribers

https://www.youtube.com/watch?v=7_Tqlw8gxOU&list=PL5Q2soXY2Zi9OhoVQBXYFIZywZXCPI4M_&index=17

Lectures on Multi-Core Cache Management

Page Coloring

- Physical memory divided into colors
- Colors map to different cache sets
- Cache partitioning
  - Ensure two threads are allocated pages of different colors

Computer Architecture - Lecture 18b: Multi-Core Cache Management (ETH Zürich, Fall 2018)

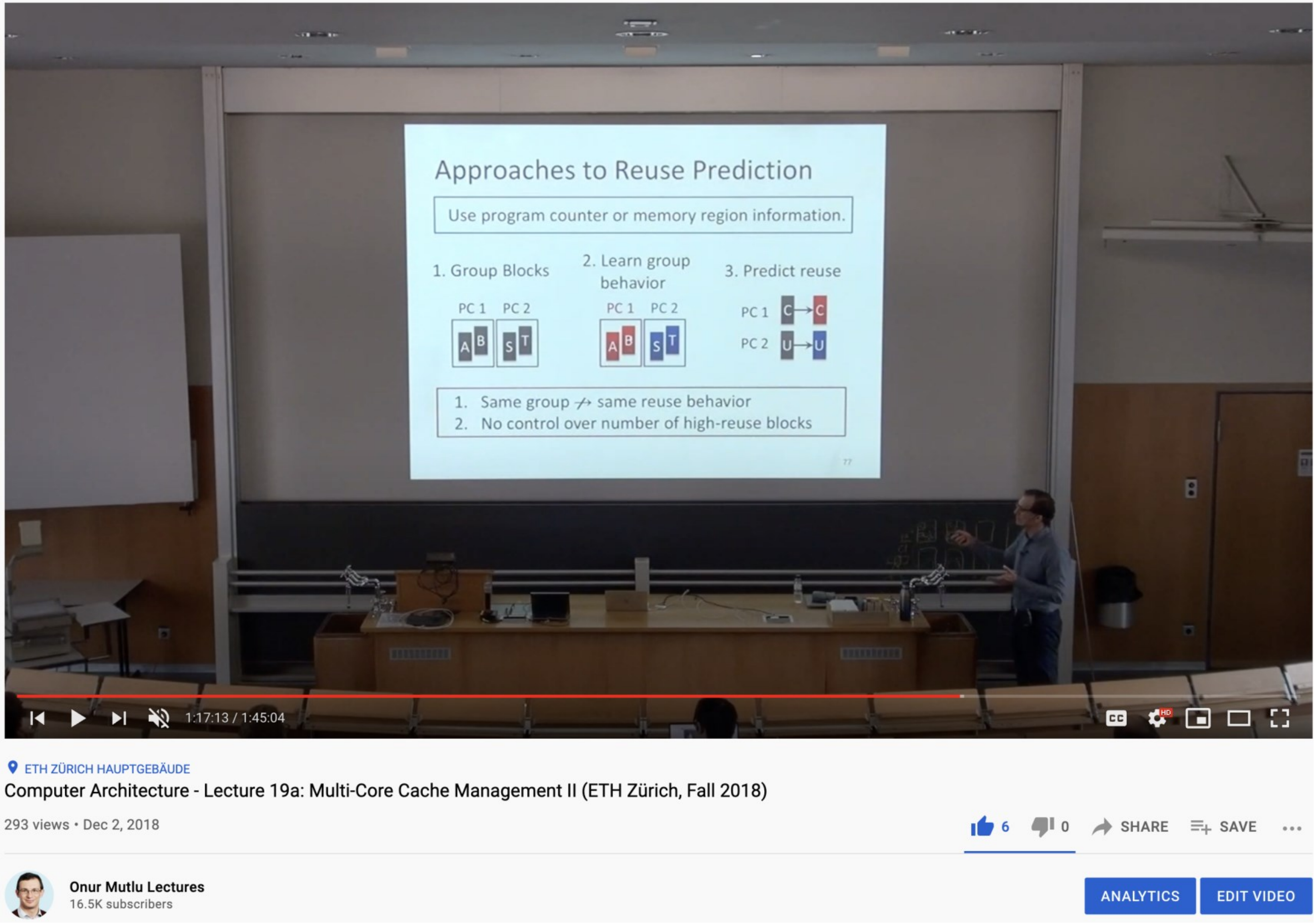
744 views • Nov 23, 2018

Onur Mutlu Lectures

16.5K subscribers

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

Lectures on Multi-Core Cache Management



Approaches to Reuse Prediction

Use program counter or memory region information.

1. Group Blocks
2. Learn group behavior
3. Predict reuse

1. Same group ↛ same reuse behavior
2. No control over number of high-reuse blocks

1:17:13 / 1:45:04

ETH ZÜRICH HAUPTGEBÄUDE

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

293 views • Dec 2, 2018

Onur Mutlu Lectures

16.5K subscribers

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

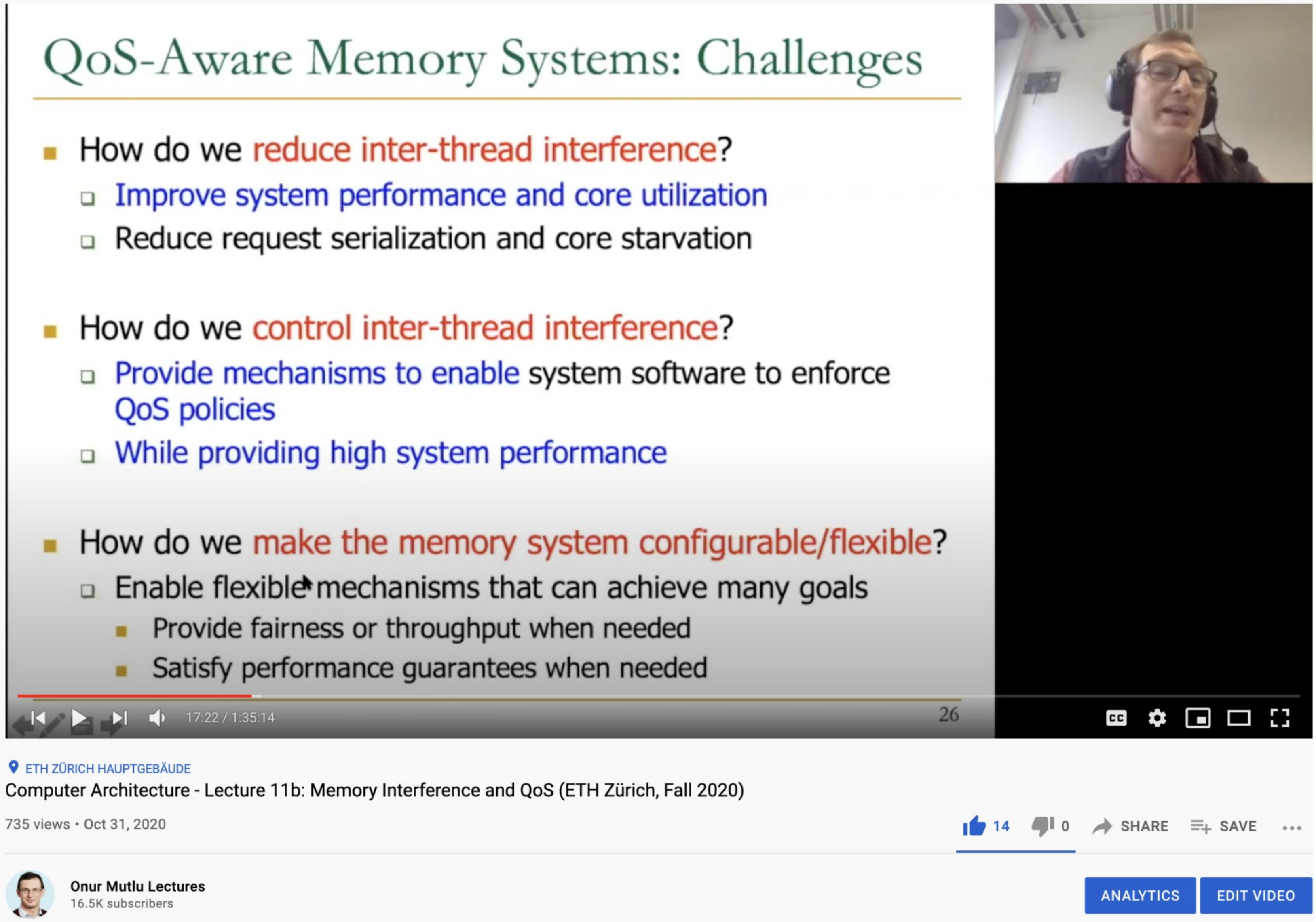
Lectures on Multi-Core Cache Management

- 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



QoS-Aware Memory Systems: Challenges

- How do we reduce inter-thread interference?
  - Improve system performance and core utilization
  - Reduce request serialization and core starvation
- How do we control inter-thread interference?
  - Provide mechanisms to enable system software to enforce QoS policies
  - While providing high system performance
- How do we make the memory system configurable/flexible?
  - Enable flexible mechanisms that can achieve many goals
    - Provide fairness or throughput when needed
    - Satisfy performance guarantees when needed

ETH ZÜRICH HAUPTGEBÄUDE

Computer Architecture - Lecture 11b: Memory Interference and QoS (ETH Zürich, Fall 2020)

735 views • Oct 31, 2020

Onur Mutlu Lectures
16.5K subscribers

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

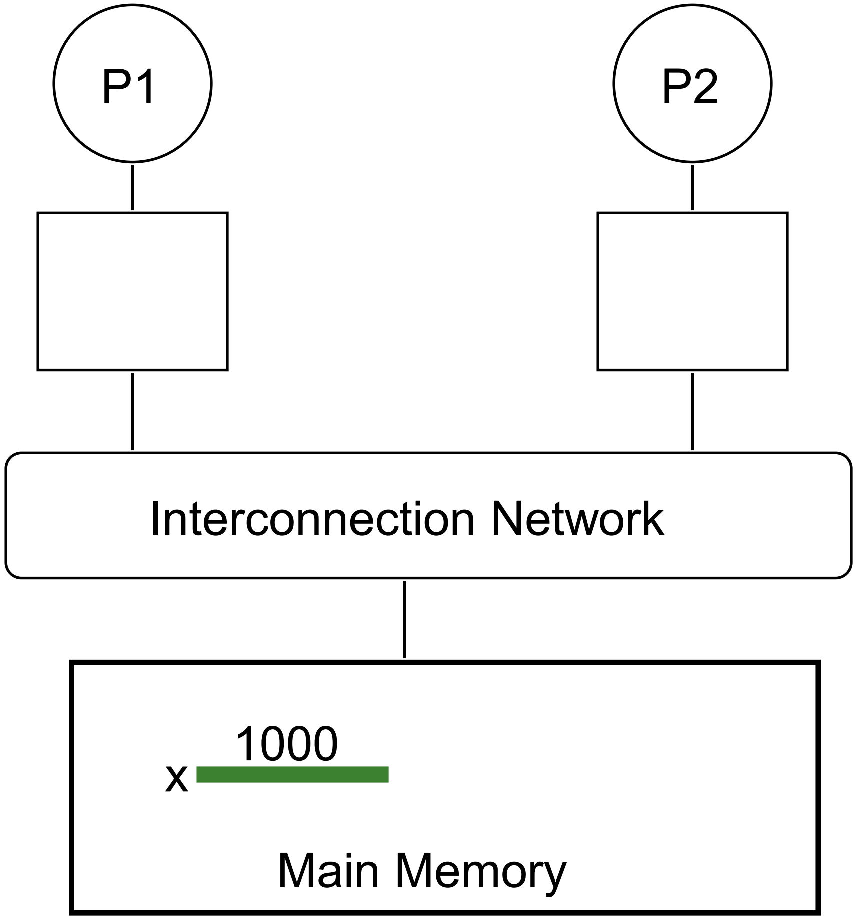
Lectures on Memory Resource Management

- 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

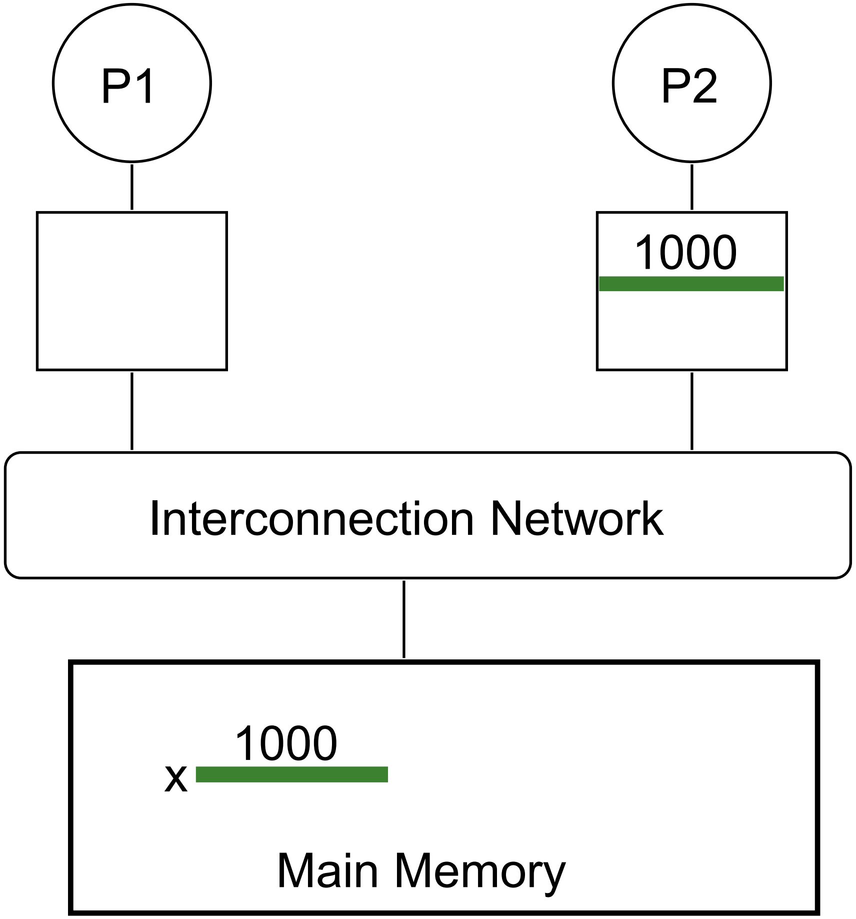
Cache Coherence

Cache Coherence

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

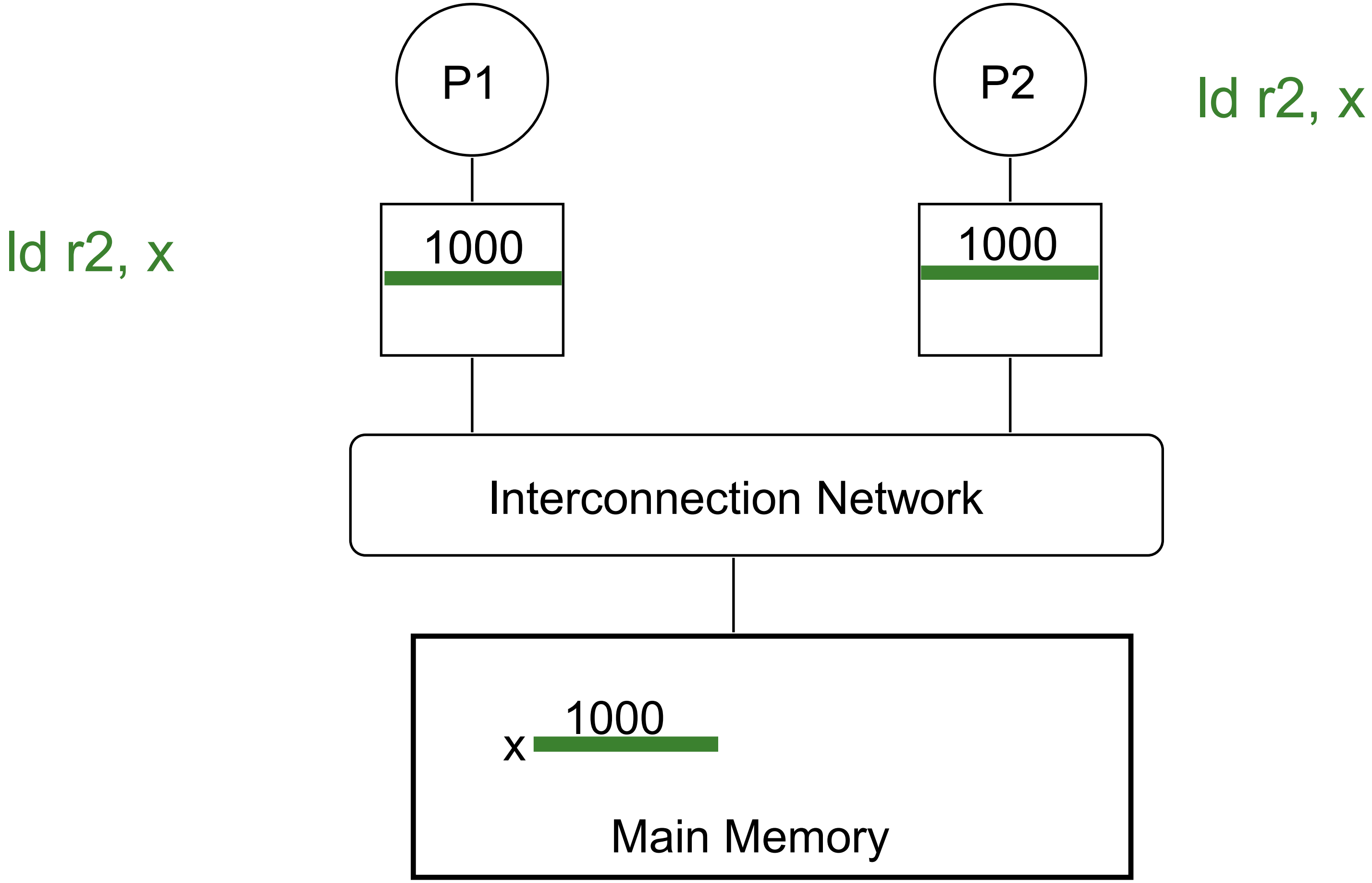


The Cache Coherence Problem



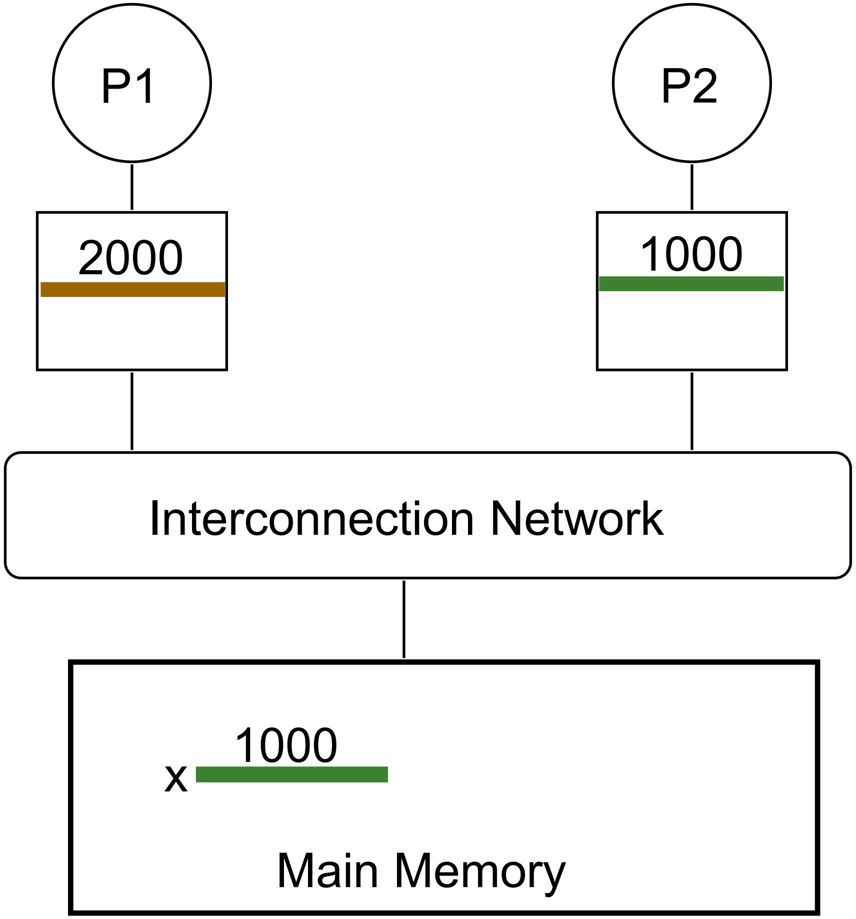
ld r2, x

The Cache Coherence Problem



The Cache Coherence Problem

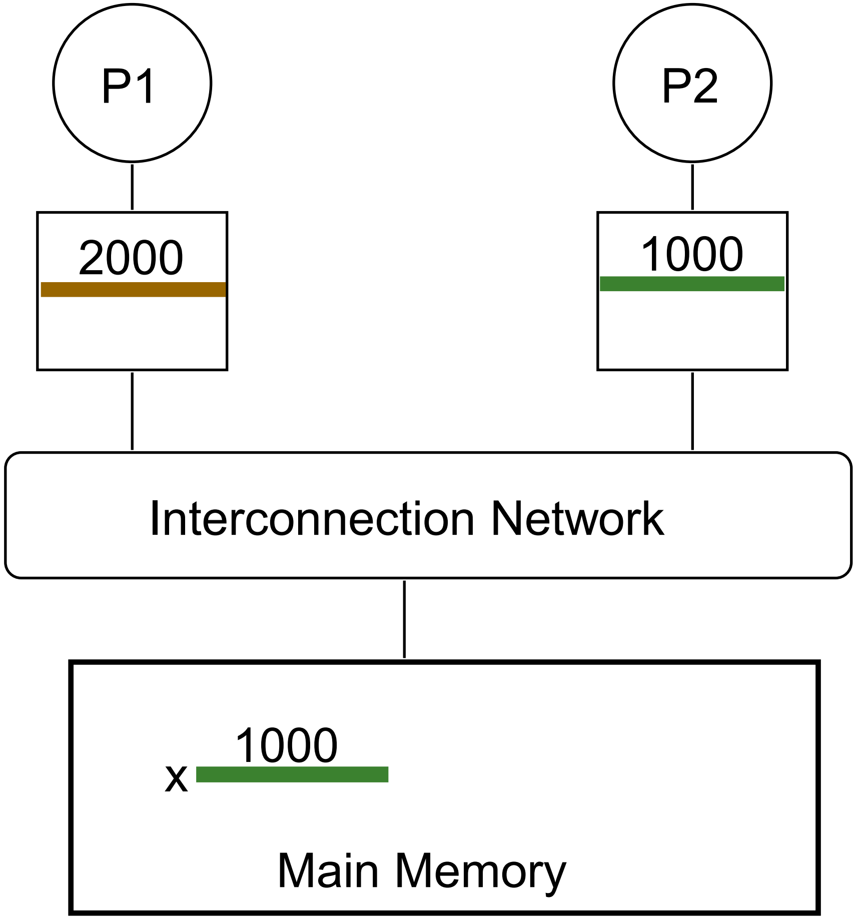
```
ld r2, x
add r1, r2, r4
st x, r1
```



```
ld r2, x
```

The Cache Coherence Problem

```
ld r2, x
add r1, r2, r4
st x, r1
```



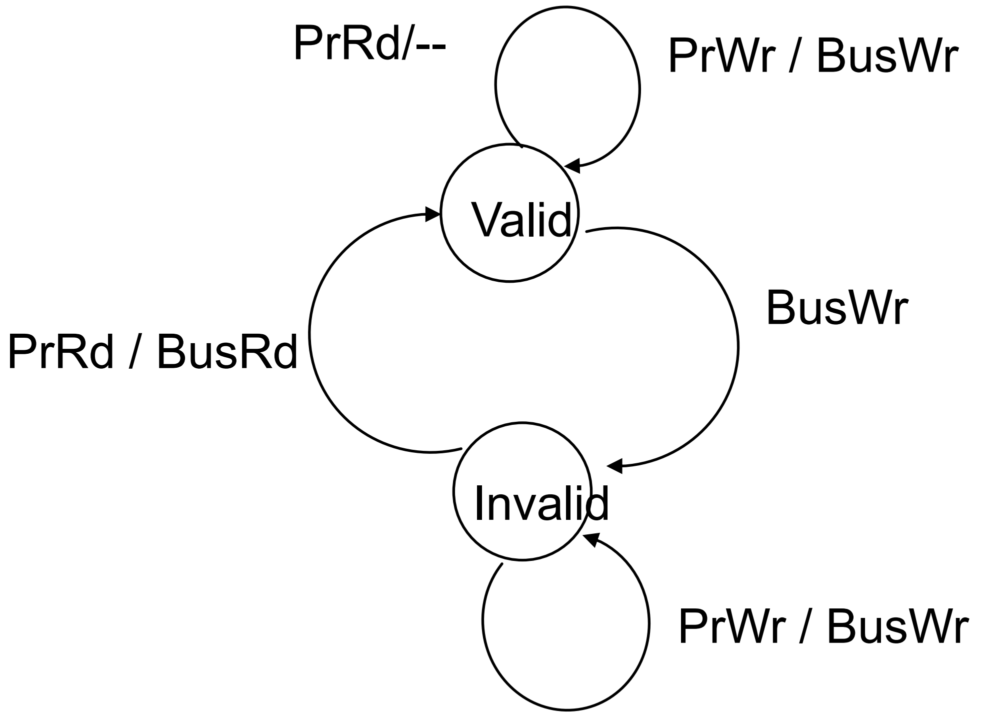
```
ld r2, x
```

Should NOT load 1000

```
ld r5, x
```

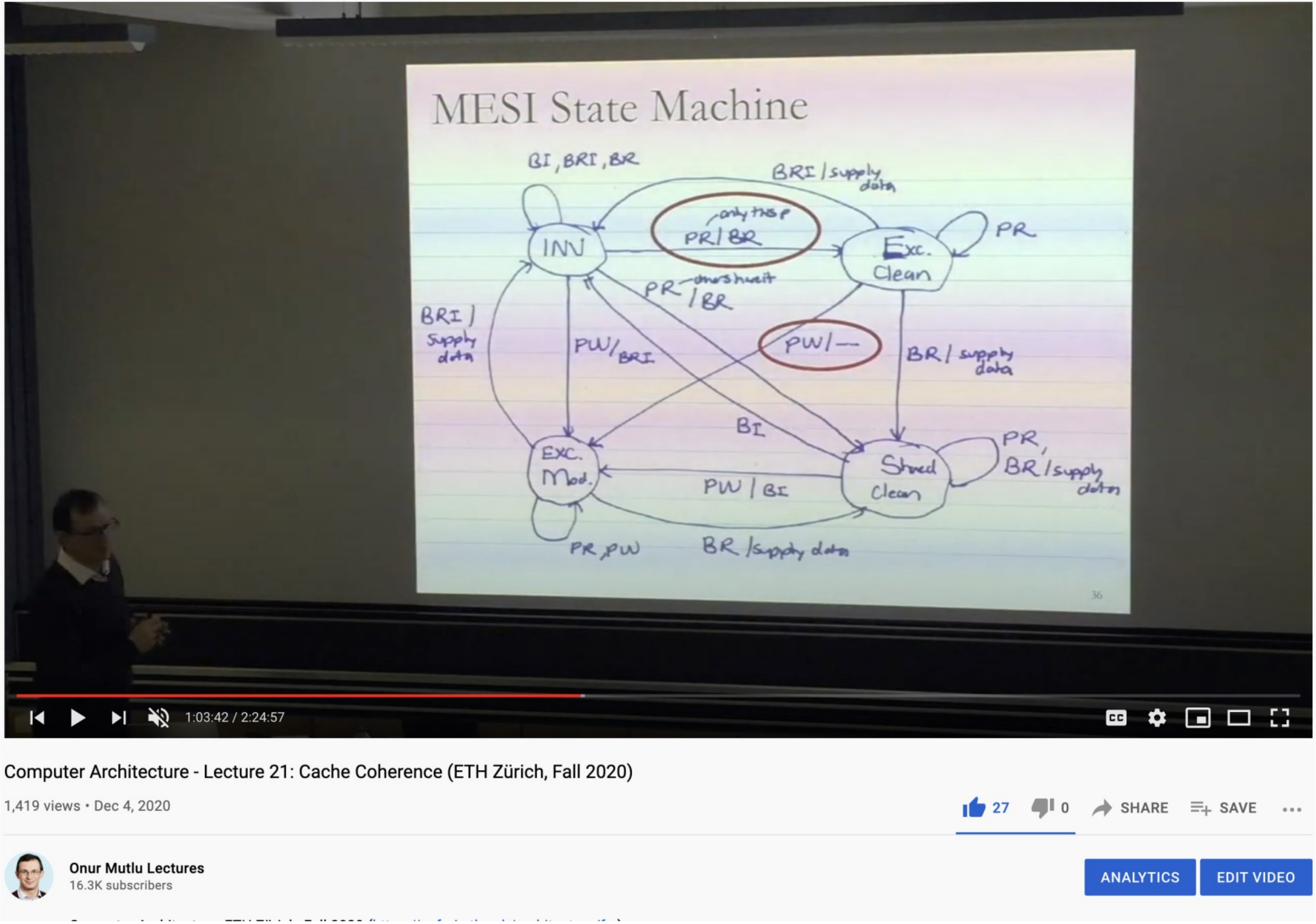
A Very Simple Coherence Scheme (VI)

- 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:



- 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

Lecture on Cache Coherence



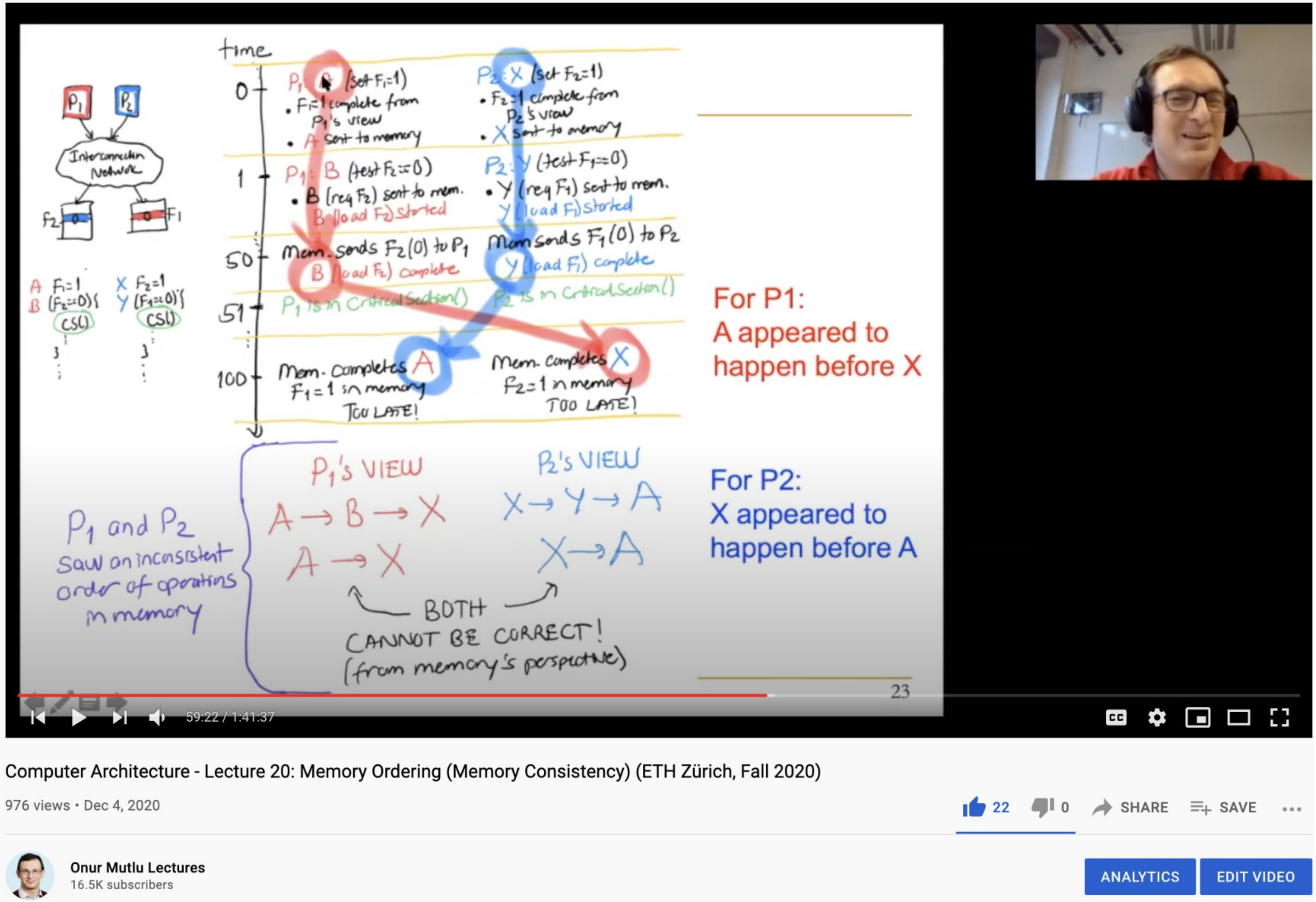
Computer Architecture - Lecture 21: Cache Coherence (ETH Zürich, Fall 2020)

1,419 views • Dec 4, 2020

Onur Mutlu Lectures
16.3K subscribers

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

Lecture on Memory Ordering & Consistency



For P1:
A appeared to happen before X

For P2:
X appeared to happen before A

23

59:22 / 1:41:37

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

976 views • Dec 4, 2020

Onur Mutlu Lectures
16.5K subscribers

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

Prefetching

Prefetching

- Idea: Fetch the data before it is needed (i.e. pre-fetch) by the program

- Why?
  - Memory latency is high. If we can prefetch accurately and early enough we can reduce/eliminate that latency.
  - Can eliminate compulsory cache misses
  - Can it eliminate all cache misses? Capacity, conflict?

- Involves predicting which address will be needed in the future
  - Works if programs have predictable miss address patterns

Prefetching and Correctness

- Does a misprediction in prefetching affect correctness?

- No, prefetched data at a “mispredicted” address is simply not used
- There is no need for state recovery
  - In contrast to branch misprediction or value misprediction

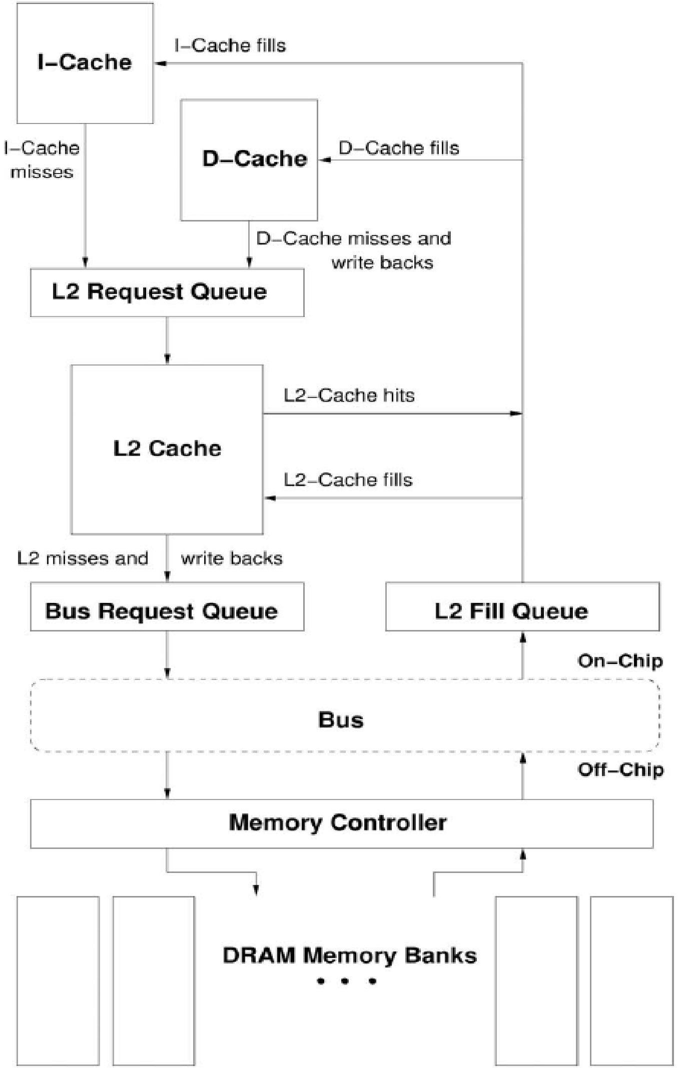
Basics

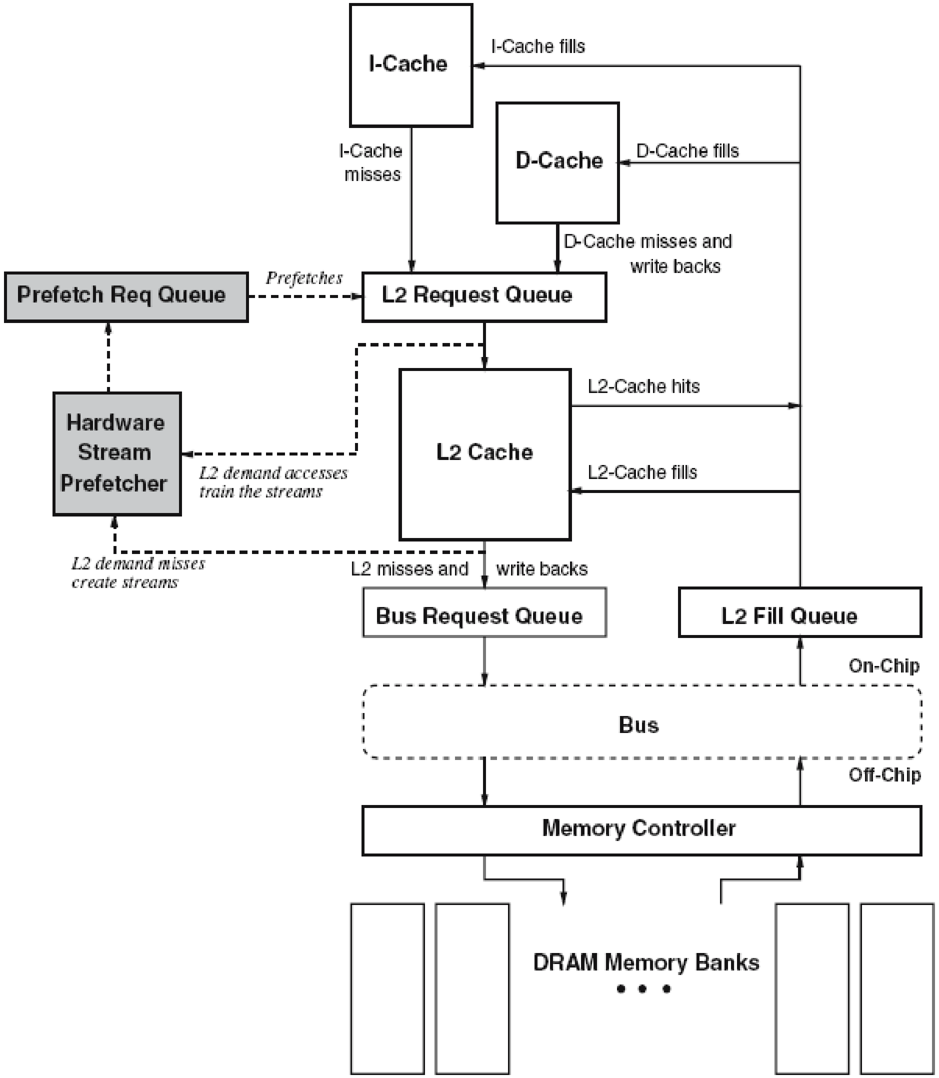
- In modern systems, prefetching is usually done in cache block granularity

- Prefetching is a technique that can reduce both
  - Miss rate
  - Miss latency

- Prefetching can be done by
  - Hardware
  - Compiler
  - Programmer
  - System

How a HW Prefetcher Fits in the Memory System





Prefetching: The Four Questions

- What
  - What addresses to prefetch (i.e., address prediction algorithm)

- When
  - When to initiate a prefetch request (early, late, on time)

- Where
  - Where to place the prefetched data (caches, separate buffer)
  - Where to place the prefetcher (which level in memory hierarchy)

- How
  - How does the prefetcher operate and who operates it (software, hardware, execution/thread-based, cooperative, hybrid)

Challenges in Prefetching: How

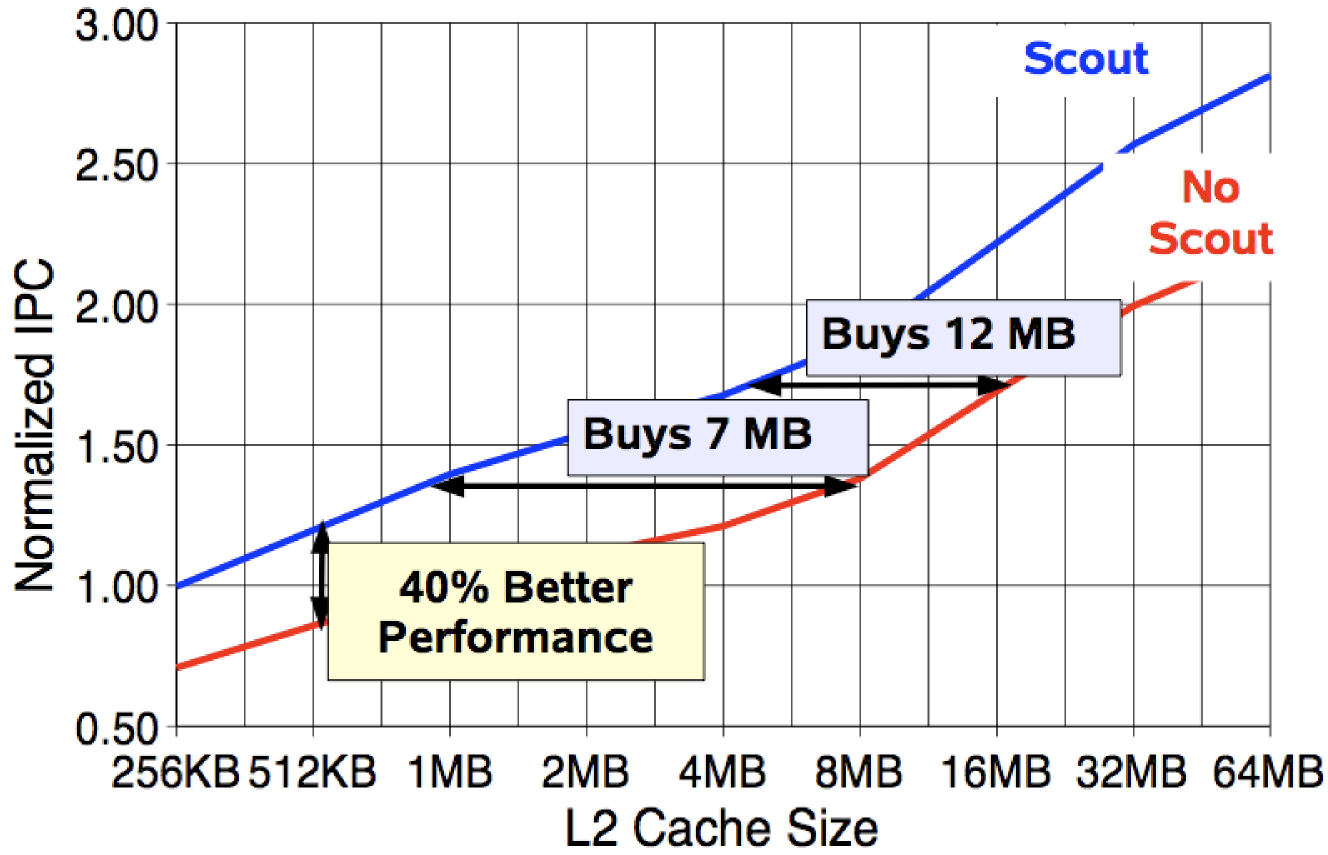
- Software prefetching
  - ISA provides prefetch instructions
  - Programmer or compiler inserts prefetch instructions (effort)
  - Usually works well only for "regular access patterns"

- Hardware prefetching
  - Hardware monitors processor accesses
  - Memorizes or finds patterns/strides
  - Generates prefetch addresses automatically

- Execution-based prefetchers
  - A "thread" is executed to prefetch data for the main program
  - Can be generated by either software/programmer or hardware

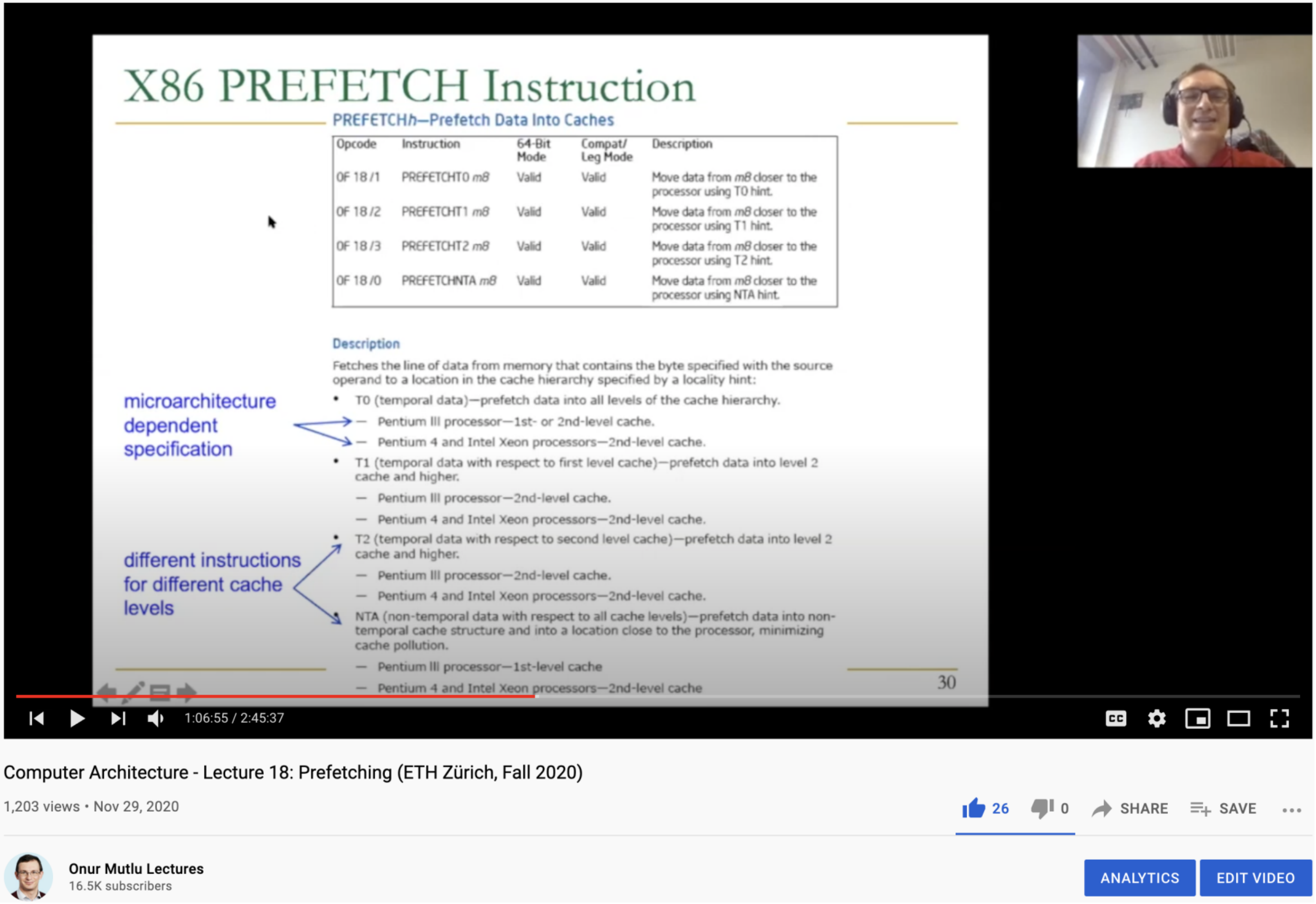
Effect of Runahead Prefetching in Sun ROCK

- Shailender Chaudhry talk, Aug 2008.



Effective prefetching can improve performance and reduce hardware cost

Lectures on Prefetching (I)



X86 PREFETCH Instruction

PREFETCHh—Prefetch Data Into Caches

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F 18 /1	PREFETCHT0 *m8*	Valid	Valid	Move data from *m8* closer to the processor using T0 hint.
0F 18 /2	PREFETCHT1 *m8*	Valid	Valid	Move data from *m8* closer to the processor using T1 hint.
0F 18 /3	PREFETCHT2 *m8*	Valid	Valid	Move data from *m8* closer to the processor using T2 hint.
0F 18 /0	PREFETCHNTA *m8*	Valid	Valid	Move data from *m8* closer to the processor using NTA hint.

Description

Fetches the line of data from memory that contains the byte specified with the source operand to a location in the cache hierarchy specified by a locality hint:

- T0 (temporal data)—prefetch data into all levels of the cache hierarchy.
  - — Pentium III processor—1st- or 2nd-level cache.
  - — Pentium 4 and Intel Xeon processors—2nd-level cache.
- T1 (temporal data with respect to first level cache)—prefetch data into level 2 cache and higher.
  - — Pentium III processor—2nd-level cache.
  - — Pentium 4 and Intel Xeon processors—2nd-level cache.
- T2 (temporal data with respect to second level cache)—prefetch data into level 2 cache and higher.
  - — Pentium III processor—2nd-level cache.
  - — Pentium 4 and Intel Xeon processors—2nd-level cache.
- NTA (non-temporal data with respect to all cache levels)—prefetch data into non-temporal cache structure and into a location close to the processor, minimizing cache pollution.
  - — Pentium III processor—1st-level cache
  - — Pentium 4 and Intel Xeon processors—2nd-level cache

microarchitecture dependent specification

different instructions for different cache levels

30

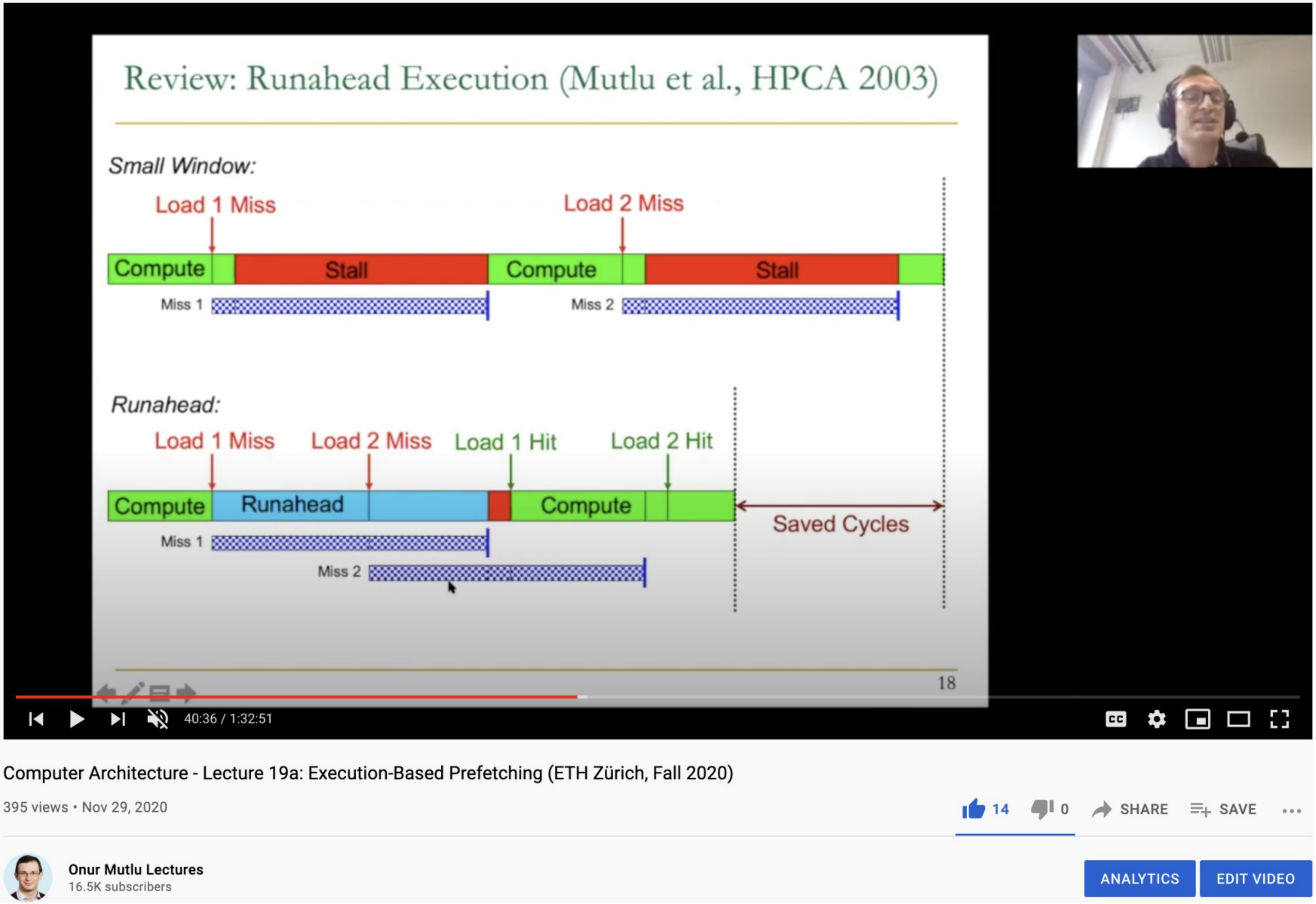
1:06:55 / 2:45:37

Computer Architecture - Lecture 18: Prefetching (ETH Zürich, Fall 2020)

1,203 views • Nov 29, 2020

https://www.youtube.com/watch?v=xZmDyj0g3Pw&list=PL5Q2soXY2Zi9xidyIgBxUz7xRPS-wisBN&index=33

Lectures on Prefetching (II)



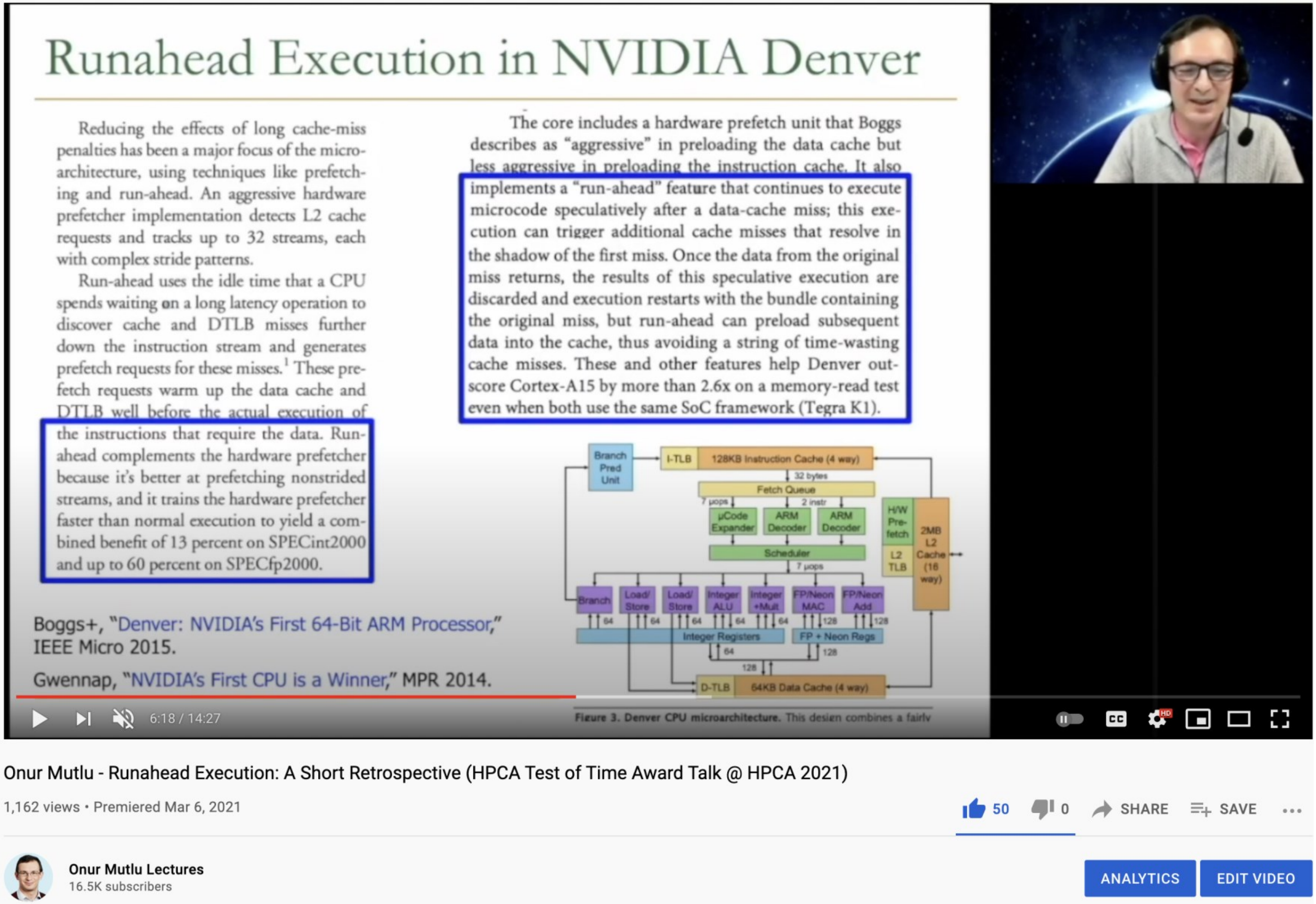
Review: Runahead Execution (Mutlu et al., HPCA 2003)

Computer Architecture - Lecture 19a: Execution-Based Prefetching (ETH Zürich, Fall 2020)

395 views • Nov 29, 2020

https://www.youtube.com/watch?v=zPewo6IaJ_8&list=PL5Q2soXY2Zi9xidyIgBxUz7xRPS-wisBN&index=34

Lectures on Prefetching (III)



Runahead Execution in NVIDIA Denver

Reducing the effects of long cache-miss penalties has been a major focus of the microarchitecture, using techniques like prefetching and run-ahead. An aggressive hardware prefetcher implementation detects L2 cache requests and tracks up to 32 streams, each with complex stride patterns.

Run-ahead uses the idle time that a CPU spends waiting on a long latency operation to discover cache and DTLB misses further down the instruction stream and generates prefetch requests for these misses.[1] These prefetch requests warm up the data cache and DTLB well before the actual execution of the instructions that require the data. Run-ahead complements the hardware prefetcher because it's better at prefetching nonstrided streams, and it trains the hardware prefetcher faster than normal execution to yield a combined benefit of 13 percent on SPECint2000 and up to 60 percent on SPECfp2000.

The core includes a hardware prefetch unit that Boggs describes as "aggressive" in preloading the data cache but less aggressive in preloading the instruction cache. It also implements a "run-ahead" feature that continues to execute microcode speculatively after a data-cache miss; this execution can trigger additional cache misses that resolve in the shadow of the first miss. Once the data from the original miss returns, the results of this speculative execution are discarded and execution restarts with the bundle containing the original miss, but run-ahead can preload subsequent data into the cache, thus avoiding a string of time-wasting cache misses. These and other features help Denver outscore Cortex-A15 by more than 2.6x on a memory-read test even when both use the same SoC framework (Tegra K1).

Figure 3. Denver CPU microarchitecture. This design combines a fairly

Boggs+, "Denver: NVIDIA's First 64-Bit ARM Processor," IEEE Micro 2015.

Gwennap, "NVIDIA's First CPU is a Winner," MPR 2014.

Onur Mutlu - Runahead Execution: A Short Retrospective (HPCA Test of Time Award Talk @ HPCA 2021)

1,162 views • Premiered Mar 6, 2021

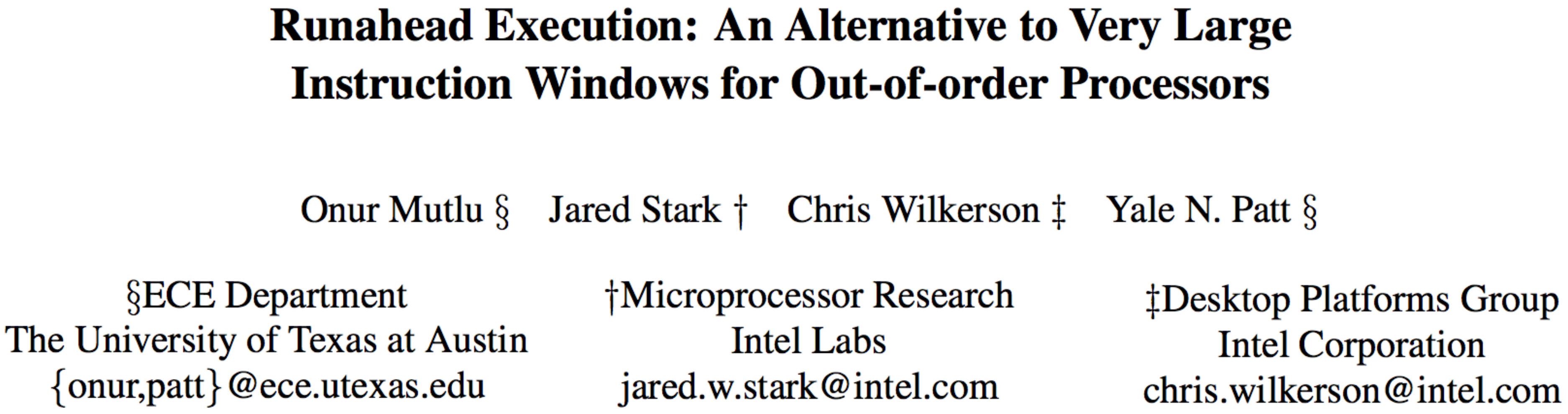
https://www.youtube.com/watch?v=KFCOecRQTIc

Lectures on Prefetching

- Computer Architecture, Fall 2020, Lecture 18
  - Prefetching (ETH, Fall 2020)
  - https://www.youtube.com/watch?v=xZmDyj0g3Pw&list=PL5Q2soXY2Zi9xidyIgBxUz7xRPS-wisBN&index=33
- Computer Architecture, Fall 2020, Lecture 19a
  - Execution-Based Prefetching (ETH, Fall 2020)
  - https://www.youtube.com/watch?v=zPewo6IaJ_8&list=PL5Q2soXY2Zi9xidyIgBxUz7xRPS-wisBN&index=34
- Computer Architecture, Spring 2015, Lecture 25
  - Prefetching (CMU, Spring 2015)
  - https://www.youtube.com/watch?v=ibPL7T9iEwY&list=PL5PHm2jkkXmi5CxxI7b3JCL1TWybTDtKq&index=29
- Computer Architecture, Spring 2015, Lecture 26
  - More Prefetching (CMU, Spring 2015)
  - https://www.youtube.com/watch?v=TUFins4z6o4&list=PL5PHm2jkkXmi5CxxI7b3JCL1TWybTDtKq&index=30

Some Readings on Prefetching

- Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale N. Patt,
  "Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors"
  *Proceedings of the 9th International Symposium on High-Performance Computer Architecture (**HPCA**)*, pages 129-140, Anaheim, CA, February 2003. Slides (pdf)
  One of the 15 computer arch. papers of 2003 selected as Top Picks by IEEE Micro. HPCA Test of Time Award (awarded in 2021).
  [Lecture Slides (pptx) (pdf)]
  [Lecture Video (1 hr 54 mins)]
  [Retrospective HPCA Test of Time Award Talk Slides (pptx) (pdf)]
  [Retrospective HPCA Test of Time Award Talk Video (14 minutes)]



Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors

Onur Mutlu § Jared Stark † Chris Wilkerson ‡ Yale N. Patt §

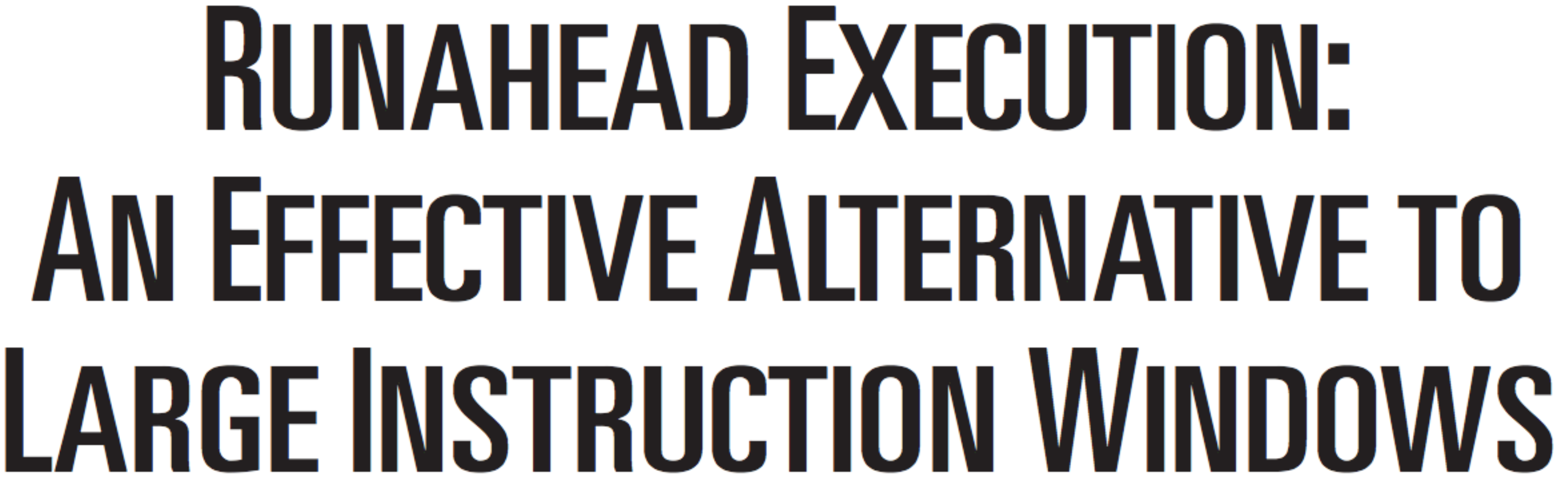
§ECE Department
The University of Texas at Austin
{onur,patt}@ece.utexas.edu

†Microprocessor Research
Intel Labs
jared.w.stark@intel.com

‡Desktop Platforms Group
Intel Corporation
chris.wilkerson@intel.com

Some Readings on Prefetching

- Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale N. Patt, **"Runahead Execution: An Effective Alternative to Large Instruction Windows"** *IEEE Micro, Special Issue: Micro's Top Picks from Microarchitecture Conferences* (***MICRO TOP PICKS***), Vol. 23, No. 6, pages 20-25, November/December 2003.



Runahead Execution:
An Effective Alternative to
Large Instruction Windows

Digital Design & Computer Arch.

Lecture 24a: Advanced Caches

Prof. Onur Mutlu

ETH Zürich
Spring 2021
28 May 2021

Basic Cache Examples:
For You to Study

Cache Terminology

- Capacity (*C*):
  - the number of data bytes a cache stores
- Block size (*b*):
  - bytes of data brought into cache at once
- Number of blocks (*B = C/b*):
  - number of blocks in cache: $B = C/b$
- Degree of associativity (*N*):
  - number of blocks in a set
- Number of sets (*S = B/N*):
  - each memory address maps to exactly one cache set

How is data found?

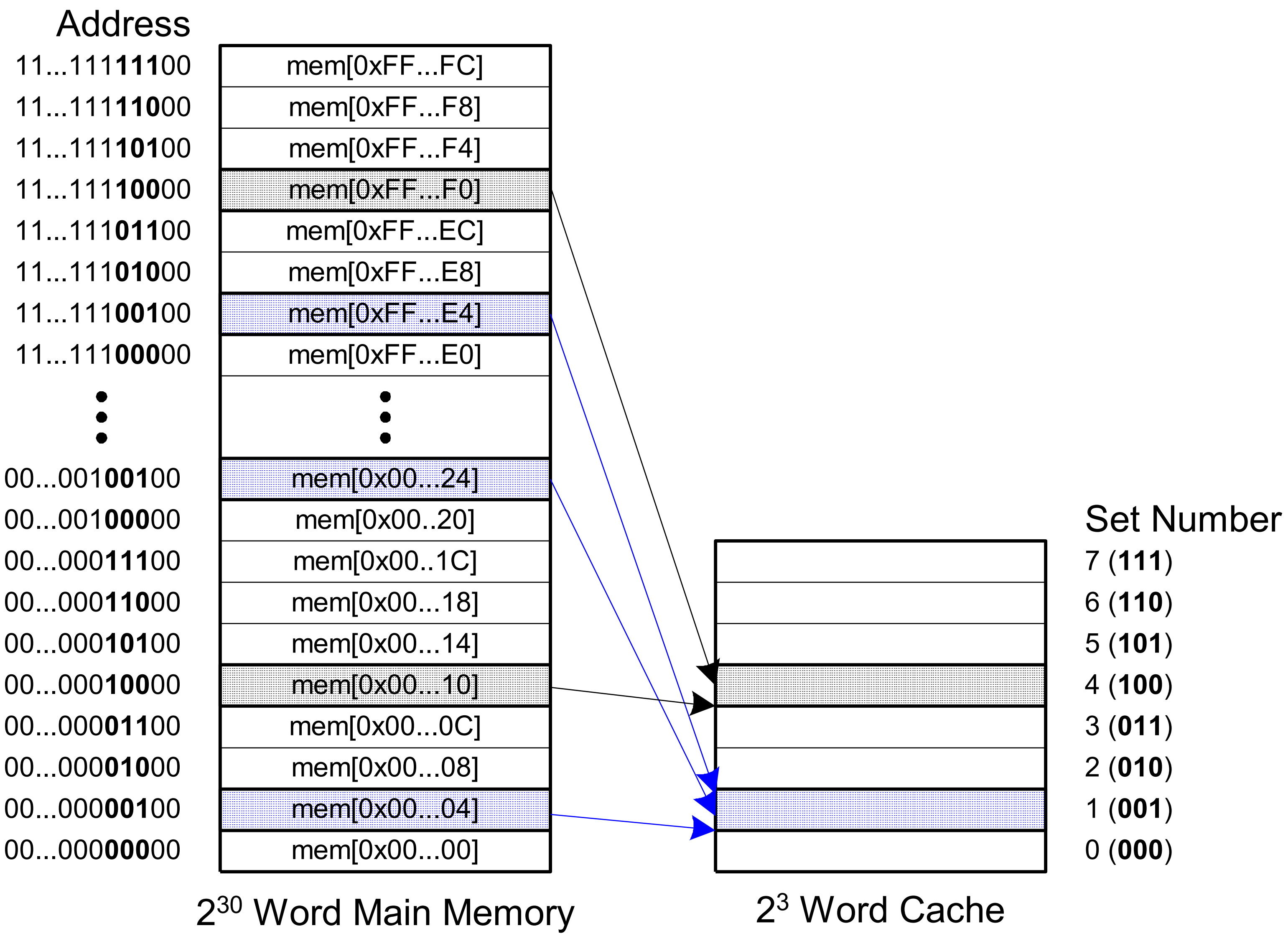
- Cache organized into *S* sets

- Each memory address maps to exactly one set

- Caches categorized by number of blocks in a set:
  - Direct mapped: 1 block per set
  - N-way set associative: N blocks per set
  - Fully associative: all cache blocks are in a single set

- Examine each organization for a cache with:
  - Capacity (*C* = 8 words)
  - Block size (*b* = 1 word)
  - So, number of blocks (*B* = 8)

Direct Mapped Cache



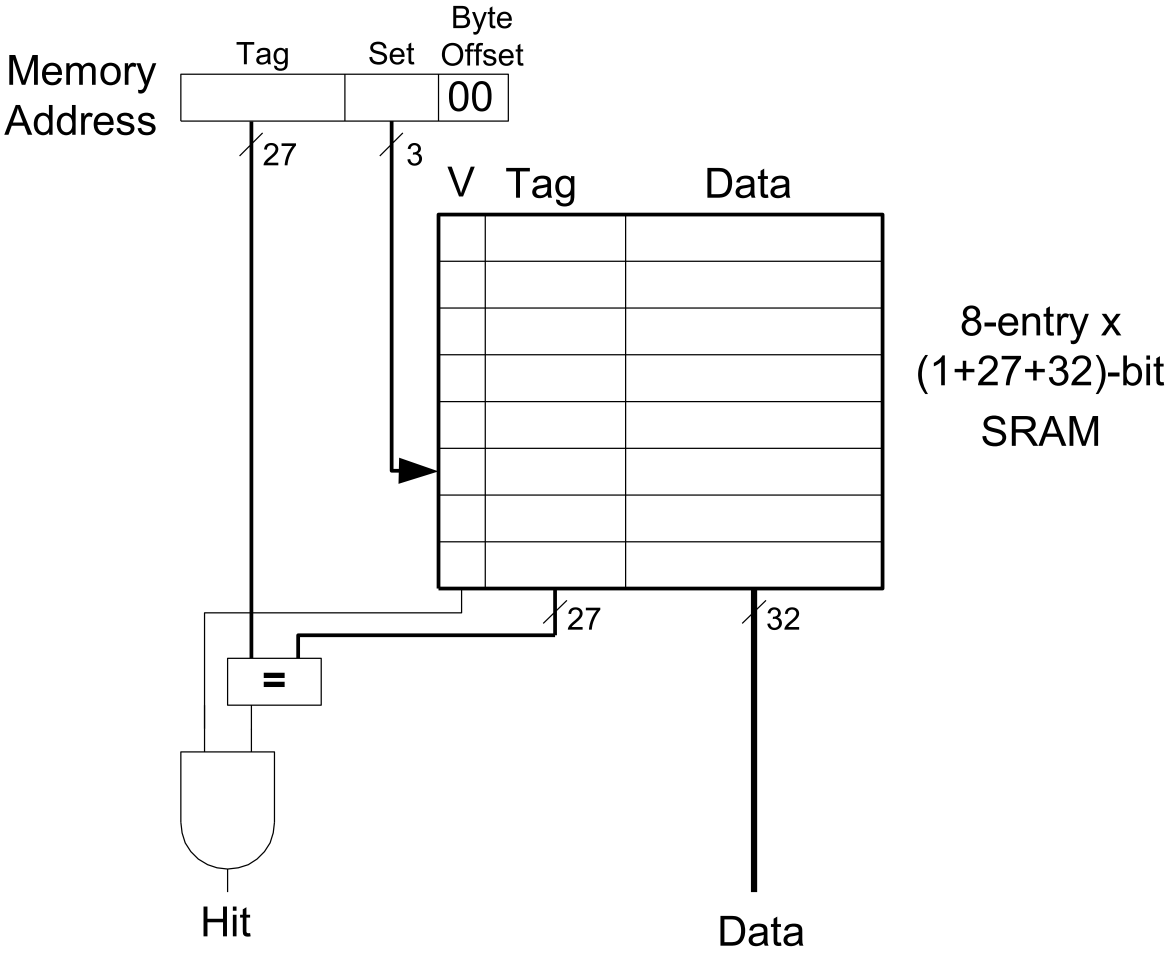
Address	
11...11**111**00	mem[0xFF...FC]
11...11**110**00	mem[0xFF...F8]
11...11**101**00	mem[0xFF...F4]
11...11**100**00	mem[0xFF...F0]
11...11**011**00	mem[0xFF...EC]
11...11**010**00	mem[0xFF...E8]
11...11**001**00	mem[0xFF...E4]
11...11**000**00	mem[0xFF...E0]
⋮	⋮
00...00**100**100	mem[0x00...24]
00...00**100**000	mem[0x00..20]
00...00**011**100	mem[0x00..1C]
00...00**011**000	mem[0x00...18]
00...00**010**100	mem[0x00...14]
00...00**010**000	mem[0x00...10]
00...00**001**100	mem[0x00...0C]
00...00**001**000	mem[0x00...08]
00...00**000**100	mem[0x00...04]
00...00**000**000	mem[0x00...00]

2^{30} Word Main Memory

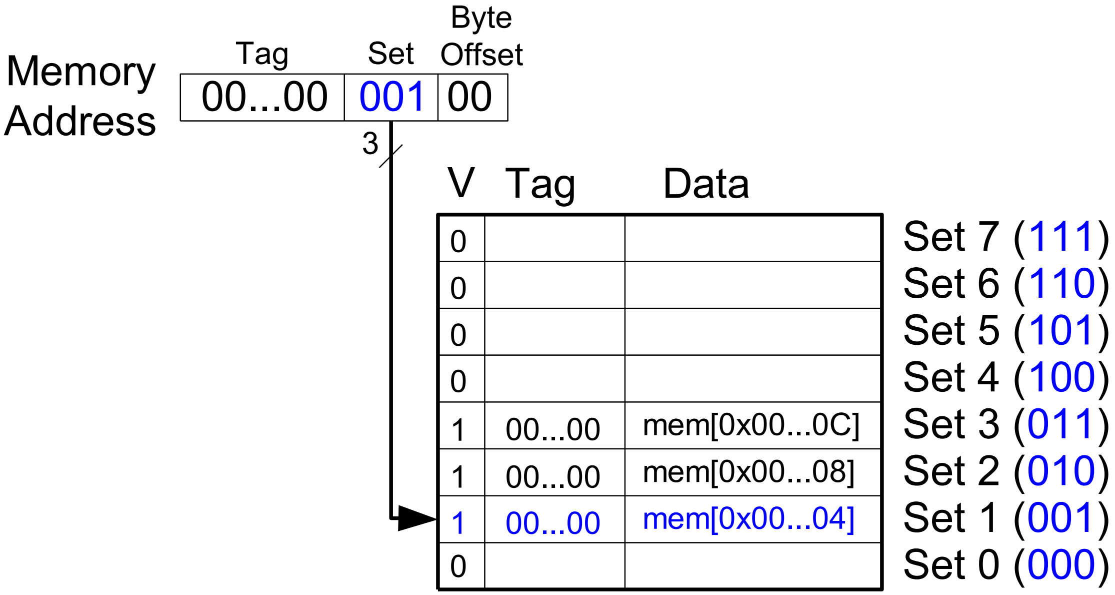
Set Number
7 (**111**)
6 (**110**)
5 (**101**)
4 (**100**)
3 (**011**)
2 (**010**)
1 (**001**)
0 (**000**)

2^3 Word Cache

Direct Mapped Cache Hardware



Direct Mapped Cache Performance

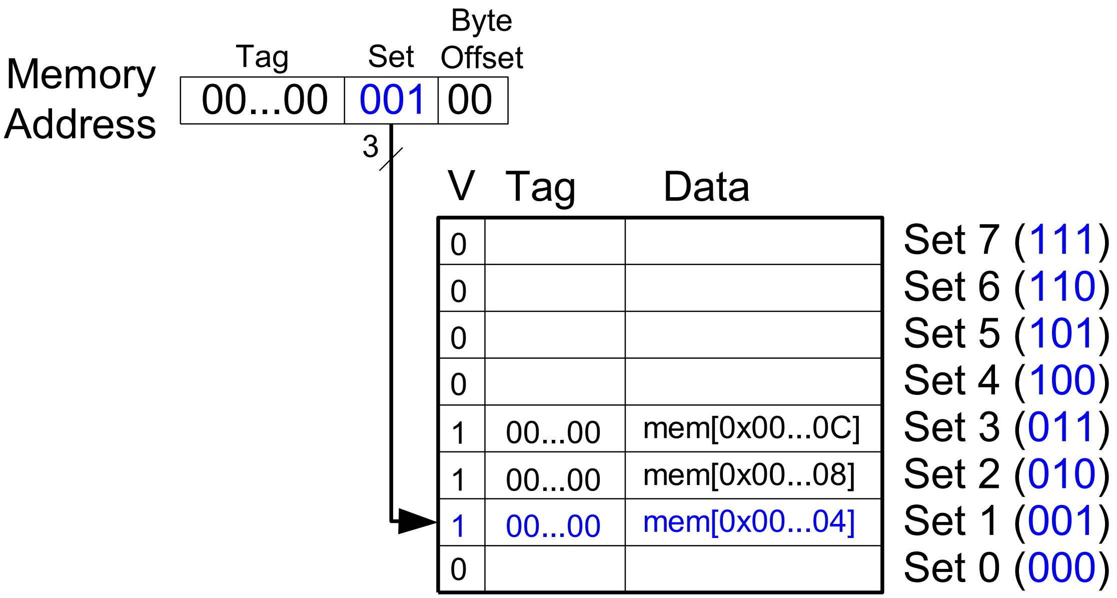


V	Tag	Data	
0			Set 7 (111)
0			Set 6 (110)
0			Set 5 (101)
0			Set 4 (100)
1	00...00	mem[0x00...0C]	Set 3 (011)
1	00...00	mem[0x00...08]	Set 2 (010)
1	00...00	mem[0x00...04]	Set 1 (001)
0			Set 0 (000)

```
# MIPS assembly code
        addi $t0, $0, 5
loop:   beq  $t0, $0, done
        lw   $t1, 0x4($0)
        lw   $t2, 0xC($0)
        lw   $t3, 0x8($0)
        addi $t0, $t0, -1
        j    loop
done:
```

Miss Rate =

Direct Mapped Cache Performance



Memory Address

Tag	Set	Byte Offset
00...00	001	00

V	Tag	Data	
0			Set 7 (111)
0			Set 6 (110)
0			Set 5 (101)
0			Set 4 (100)
1	00...00	mem[0x00...0C]	Set 3 (011)
1	00...00	mem[0x00...08]	Set 2 (010)
1	00...00	mem[0x00...04]	Set 1 (001)
0			Set 0 (000)

```
# MIPS assembly code
        addi $t0, $0, 5
loop:   beq  $t0, $0, done
        lw   $t1, 0x4($0)
        lw   $t2, 0xC($0)
        lw   $t3, 0x8($0)
        addi $t0, $t0, -1
        j    loop
done:
```

Miss Rate = 3/15

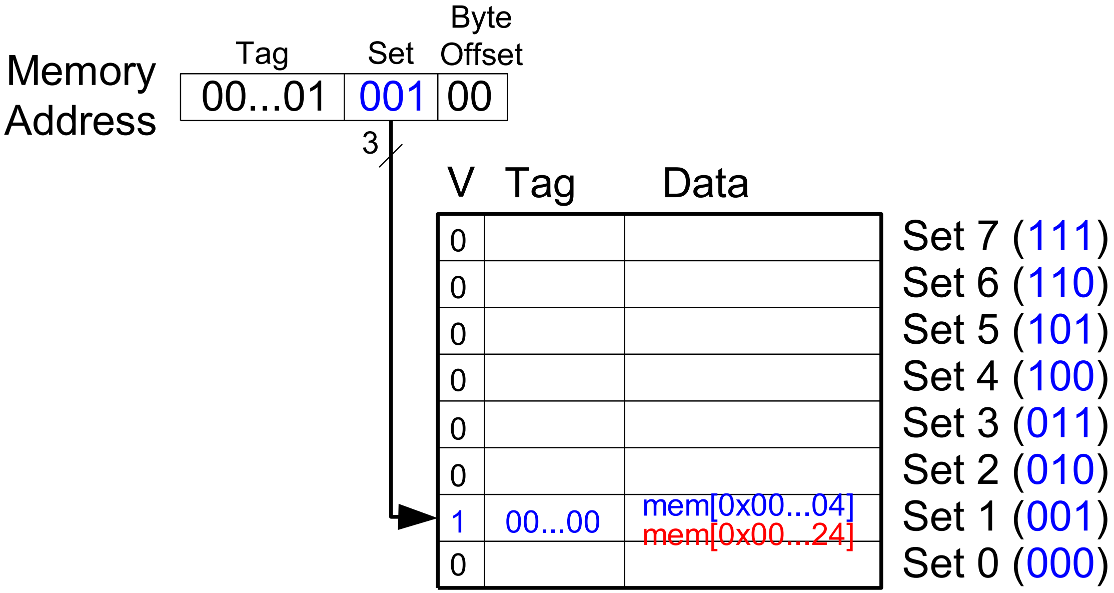
=

20%

Temporal Locality

Compulsory Misses

Direct Mapped Cache: Conflict

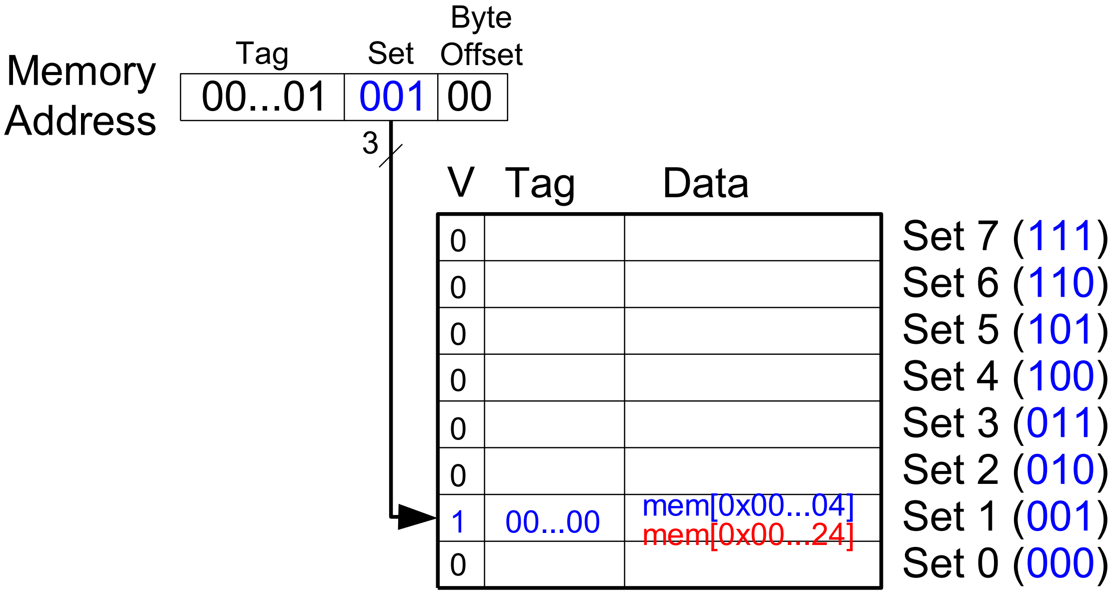


V	Tag	Data	
0			Set 7 (111)
0			Set 6 (110)
0			Set 5 (101)
0			Set 4 (100)
0			Set 3 (011)
0			Set 2 (010)
1	00...00	mem[0x00...04] mem[0x00...24]	Set 1 (001)
0			Set 0 (000)

```
# MIPS assembly code
        addi $t0, $0, 5
loop:   beq  $t0, $0, done
        lw   $t1, 0x4($0)
        lw   $t2, 0x24($0)
        addi $t0, $t0, -1
        j    loop
done:
```

Miss Rate =

Direct Mapped Cache: Conflict

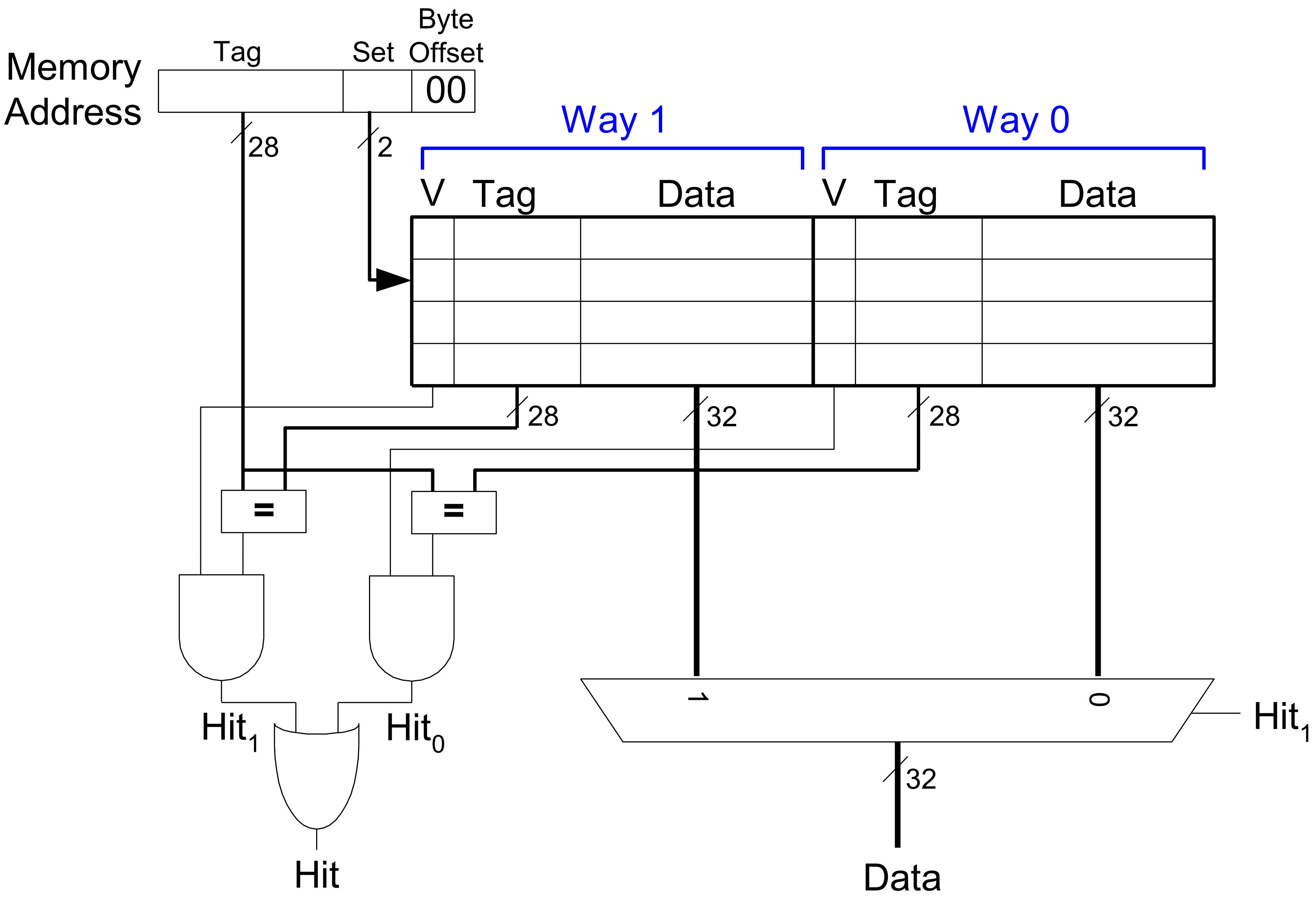


```
# MIPS assembly code
        addi $t0, $0, 5
loop:   beq  $t0, $0, done
        lw   $t1, 0x4($0)
        lw   $t2, 0x24($0)
        addi $t0, $t0, -1
        j    loop
done:
```

Miss Rate = 10/10
= 100%

Conflict Misses

N-Way Set Associative Cache



N-way Set Associative Performance

```
# MIPS assembly code

        addi $t0, $0, 5
loop:   beq  $t0, $0, done
        lw   $t1, 0x4($0)
        lw   $t2, 0x24($0)
        addi $t0, $t0, -1
        j    loop
done:
```

Miss Rate =

Way 1			Way 0			
V	Tag	Data	V	Tag	Data	
0			0			Set 3
0			0			Set 2
1	00...10	mem[0x00...24]	1	00...00	mem[0x00...04]	Set 1
0			0			Set 0

N-way Set Associative Performance

```
# MIPS assembly code

        addi $t0, $0, 5
loop:   beq  $t0, $0, done
        lw   $t1, 0x4($0)
        lw   $t2, 0x24($0)
        addi $t0, $t0, -1
        j    loop
done:
```

Miss Rate = 2/10

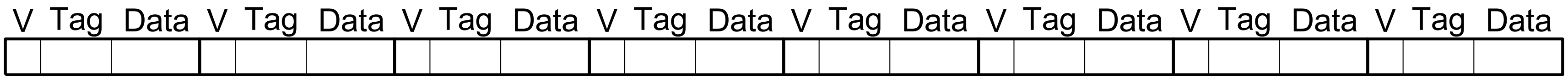
= 20%

Associativity reduces conflict misses

Way 1			Way 0			
V	Tag	Data	V	Tag	Data	
0			0			Set 3
0			0			Set 2
1	00...10	mem[0x00...24]	1	00...00	mem[0x00...04]	Set 1
0			0			Set 0

Fully Associative Cache

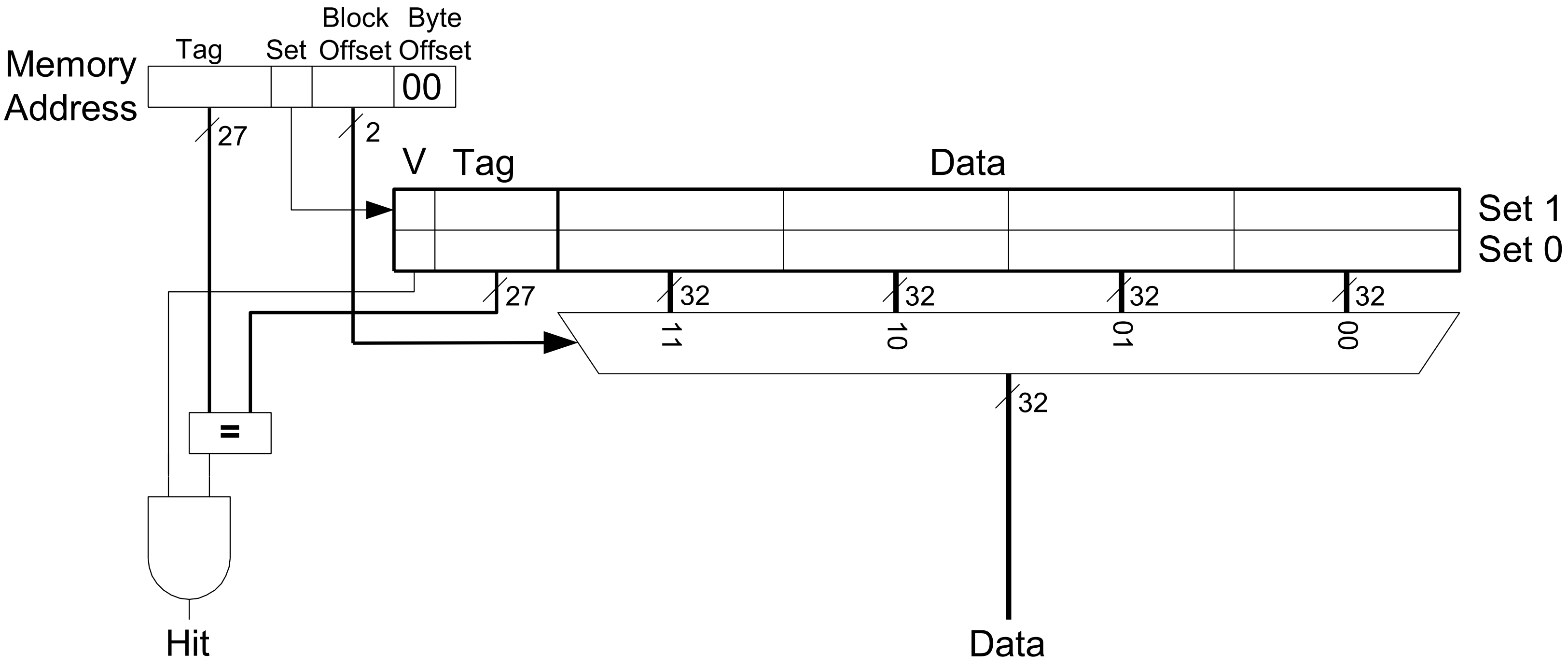
- No conflict misses
- Expensive to build



V Tag Data V Tag Data V Tag Data V Tag Data V Tag Data V Tag Data V Tag Data V Tag Data

Spatial Locality?

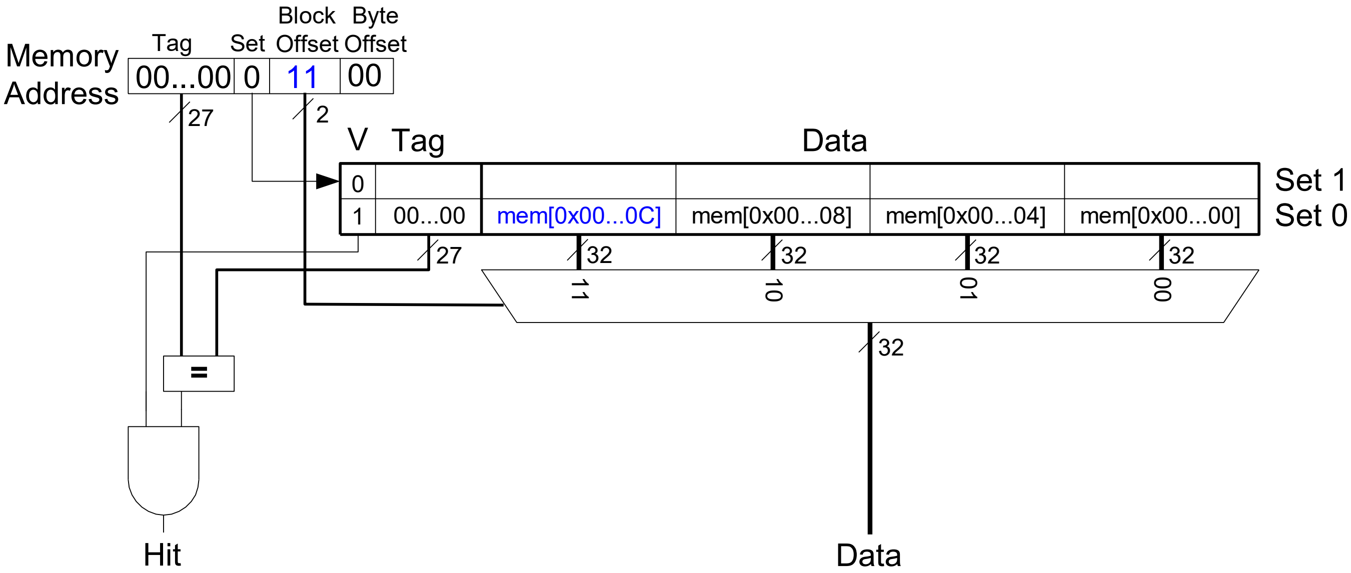
- Increase block size:
  - Block size, ***b* = 4** words
  - *C* = 8 words
  - Direct mapped (1 block per set)
  - Number of blocks, *B* = *C*/*b* = 8/4 = 2



Direct Mapped Cache Performance

```
        addi $t0, $0, 5
loop:   beq  $t0, $0, done
        lw   $t1, 0x4($0)
        lw   $t2, 0xC($0)
        lw   $t3, 0x8($0)
        addi $t0, $t0, -1
        j    loop
done:
```

Miss Rate =



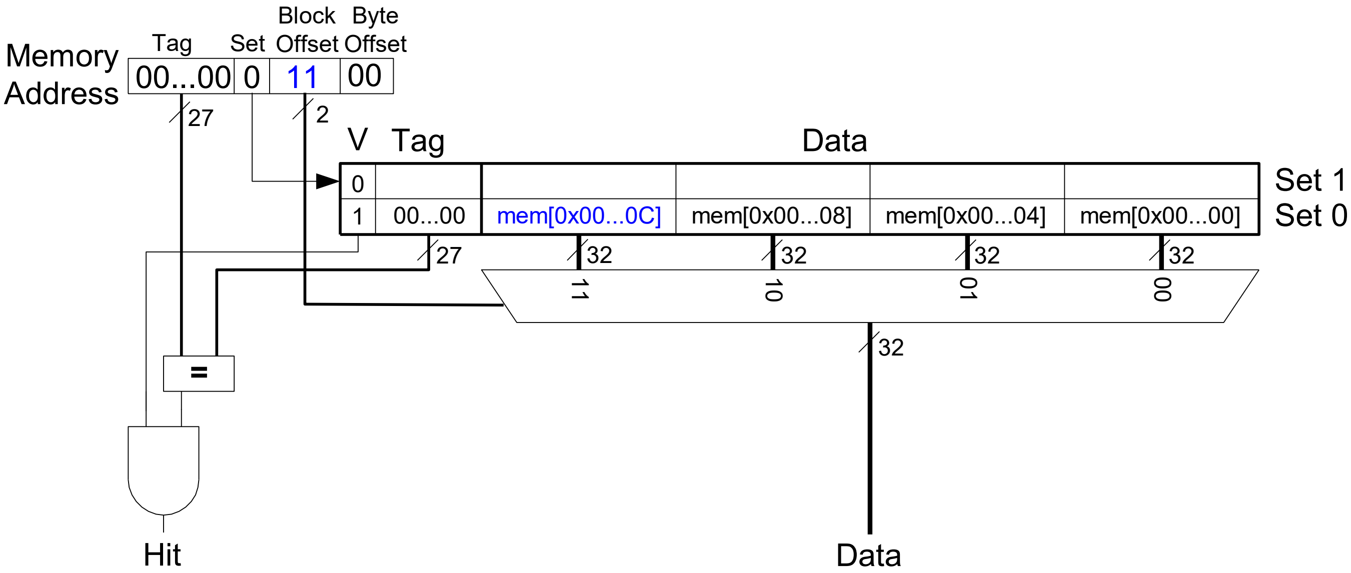
Direct Mapped Cache Performance

```
        addi $t0, $0, 5
loop:   beq  $t0, $0, done
        lw   $t1, 0x4($0)
        lw   $t2, 0xC($0)
        lw   $t3, 0x8($0)
        addi $t0, $t0, -1
        j    loop
done:
```

Miss Rate = 1/15

= 6.67%

Larger blocks reduce compulsory misses through spatial locality



Cache Organization Recap

- Main Parameters
  - Capacity: ***C***
  - Block size: ***b***
  - Number of blocks in cache: ***B*** = *C*/*b*
  - Number of blocks in a set: ***N***
  - Number of Sets: ***S*** = *B*/*N*

Organization	**Number of Ways (N)**	**Number of Sets (S = B/N)**
Direct Mapped	1	B
N-Way Set Associative	1 < N < B	B / N
Fully Associative	B	1

Capacity Misses

- Cache is too small to hold all data of interest at one time
  - If the cache is full and program tries to access data X that is not in cache, cache must evict data Y to make room for X
  - **Capacity miss** occurs if program then tries to access Y again
  - X will be placed in a particular set based on its address

- In a direct mapped cache, there is only one place to put X

- In an associative cache, there are multiple ways where X could go in the set.

- How to choose Y to minimize chance of needing it again?
  - Least recently used (LRU) replacement: the least recently used block in a set is evicted when the cache is full.

Types of Misses

- Compulsory: first time data is accessed
- Capacity: cache too small to hold all data of interest
- Conflict: data of interest maps to same location in cache
- Miss penalty: time it takes to retrieve a block from lower level of hierarchy

LRU Replacement

```
# MIPS assembly

lw $t0, 0x04($0)
lw $t1, 0x24($0)
lw $t2, 0x54($0)
```

(a)

V	U	Tag	Data	V	Tag	Data	Set Number
							3 (**11**)
							2 (**10**)
							1 (**01**)
							0 (**00**)

(b)

V	U	Tag	Data	V	Tag	Data	Set Number
							3 (**11**)
							2 (**10**)
							1 (**01**)
							0 (**00**)

LRU Replacement

```
# MIPS assembly

lw $t0, 0x04($0)
lw $t1, 0x24($0)
lw $t2, 0x54($0)
```

(a)

		Way 1			Way 0		
V	U	Tag	Data	V	Tag	Data	
0	0			0			Set 3 (11)
0	0			0			Set 2 (10)
1	0	00...010	mem[0x00...24]	1	00...000	mem[0x00...04]	Set 1 (01)
0	0			0			Set 0 (00)

(b)

		Way 1			Way 0		
V	U	Tag	Data	V	Tag	Data	
0	0			0			Set 3 (11)
0	0			0			Set 2 (10)
1	1	00...010	mem[0x00...24]	1	00...101	mem[0x00...54]	Set 1 (01)
0	0			0			Set 0 (00)