Lecture 24: Memory Hierarchy and Caches

Frank K. Gürkaynak
Mohammad Sadrosadati
Prof. Onur Mutlu

ETH Zürich
Spring 2024
24 May 2024
The Memory Hierarchy
Memory Hierarchy in a Modern System (I)
Memory Hierarchy in a Modern System (II)

Source: https://www.anandtech.com/show/16252/mac-mini-apple-m1-tested
Memory Hierarchy in a Modern System (III)

Apple M1 Ultra System (2022)

https://www.gsmarena.com/apple_announces_m1_ultra_with_20core_cpu_and_64core_gpu-news-53481.php
Memory Hierarchy in an Older System

Intel Pentium Pro, 1995

Processor chip  Level 2 cache chip

Multi-chip module package
Memory Hierarchy in an Older System

https://download.intel.com/newsroom/kits/40thanniversary/gallery/images/Pentium_4_6xx-die.jpg

Intel Pentium 4, 2000
Memory Hierarchy in a Modern System (IV)

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

Memory Hierarchy in a Modern System (V)

IBM POWER10, 2020

Cores:
15-16 cores, 8 threads/core

L2 Caches:
2 MB per core

L3 Cache:
120 MB shared
Memory Hierarchy in a Modern System (VI)

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

Nvidia Ampere, 2020

Ideal Memory

- Zero access time (latency)
- Infinite capacity
- Zero cost
- Infinite bandwidth (to support multiple accesses in parallel)
- Zero energy
The Problem

- Ideal memory’s requirements oppose each other

- Bigger is slower
  - Bigger $\rightarrow$ Takes longer to determine the location

- Faster is more expensive
  - Memory technology: SRAM vs. DRAM vs. SSD vs. Disk vs. Tape

- Higher bandwidth is more expensive
  - Need more banks, more ports, more channels, higher frequency or faster technology
The Problem

- **Bigger is slower**
  - SRAM, < 1KByte, sub-nanosec
  - SRAM, KByte~MByte, ~nanosec
  - DRAM, Gigabyte, ~50 nanosec
  - PCM-DIMM (Intel Optane DC DIMM), Gigabyte, ~300 nanosec
  - PCM-SSD (Intel Optane SSD), Gigabyte ~Terabyte, ~6-10 µs
  - Flash memory, Gigabyte~Terabyte, ~50-100 µs
  - Hard Disk, Terabyte, ~10 millisec

- **Faster is more expensive (monetary cost and chip area)**
  - SRAM, < 0.3$ per Megabyte
  - DRAM, < 0.006$ per Megabyte
  - PCM-DIMM (Intel Optane DC DIMM), < 0.004$ per Megabyte
  - PCM-SSD, < 0.002$ per Megabyte
  - Flash memory, < 0.00008$ per Megabyte
  - Hard Disk, < 0.00003$ per Megabyte
  - These sample values (circa ~2023) scale with time

- **Other technologies have their place as well**
  - FeRAM, MRAM, RRAM, STT-MRAM, memristors, ... (not mature yet)
### The Problem (Table View)

<table>
<thead>
<tr>
<th>Memory Device</th>
<th>Capacity</th>
<th>Latency</th>
<th>Cost per Megabyte</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRAM</td>
<td>&lt; 1 KByte</td>
<td>sub-nanosec</td>
<td>&lt; 0.3$</td>
</tr>
<tr>
<td>SRAM</td>
<td>KByte~MByte</td>
<td>~nanosec</td>
<td>&lt; 0.3$</td>
</tr>
<tr>
<td>DRAM</td>
<td>Gigabyte</td>
<td>~50 nanosec</td>
<td>&lt; 0.006$</td>
</tr>
<tr>
<td>PCM-DIMM (Intel Optane DC DIMM)</td>
<td>Gigabyte</td>
<td>~300 nanosec</td>
<td>&lt; 0.004$</td>
</tr>
<tr>
<td>PCM-SSD (Intel Optane SSD)</td>
<td>Gigabyte ~Terabyte</td>
<td>~6-10 µs</td>
<td>&lt; 0.002$</td>
</tr>
<tr>
<td>Flash memory</td>
<td>Gigabyte ~Terabyte</td>
<td>~50-100 µs</td>
<td>&lt; 0.00008$</td>
</tr>
<tr>
<td>Hard Disk</td>
<td>Terabyte</td>
<td>~10 millisec</td>
<td>&lt; 0.00003$</td>
</tr>
</tbody>
</table>

Bigger is slower

Faster is more expensive ($$$ and chip area)

These sample values (circa ~2023) scale with time
The Problem (Table View): Energy

<table>
<thead>
<tr>
<th>Memory Device</th>
<th>Capacity</th>
<th>Latency</th>
<th>Cost per Megabyte</th>
<th>Energy per access</th>
<th>Energy per byte access</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRAM</td>
<td>&lt; 1 KByte</td>
<td>sub-nanosec</td>
<td></td>
<td>~5 pJ</td>
<td>~1.25 pJ</td>
</tr>
<tr>
<td>SRAM</td>
<td>KByte~MB yte</td>
<td>~nanosec</td>
<td>&lt; 0.3$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DRAM</td>
<td>Gigabyte</td>
<td>~50 nanosec</td>
<td>&lt; 0.006$</td>
<td>~40-140 pJ</td>
<td>~10-35 pJ</td>
</tr>
<tr>
<td>PCM-DIMM (Intel Optane DC DIMM)</td>
<td>Gigabyte</td>
<td>~300 nanosec</td>
<td>&lt; 0.004$</td>
<td>~80-540 pJ</td>
<td>~20-135 pJ</td>
</tr>
<tr>
<td>PCM-SSD (Intel Optane SSD)</td>
<td>Gigabyte~Terabyte</td>
<td>~6-10 µs</td>
<td>&lt; 0.002$</td>
<td>~120 µJ</td>
<td>~30 nJ</td>
</tr>
<tr>
<td>Flash memory</td>
<td>Gigabyte~Terabyte</td>
<td>~50-100 µs</td>
<td>&lt; 0.00008$</td>
<td>~250 µJ</td>
<td>~61 nJ</td>
</tr>
<tr>
<td>Hard Disk</td>
<td>Terabyte</td>
<td>~10 millisec</td>
<td>&lt; 0.00003$</td>
<td>~60 mJ</td>
<td>~15 µJ</td>
</tr>
</tbody>
</table>

Faster is more energy-efficient

Bigger is slower

Faster is more expensive
($$$ and chip area)

These sample values (circa ~2023) scale with time

Disclaimer: Take the energy values with a grain of salt as there are different assumptions
Aside: The Problem (2011 Version)

- **Bigger is slower**
  - SRAM, 512 Bytes, sub-nanosec
  - SRAM, KByte~MByte, ~nanosec
  - DRAM, Gigabyte, ~50 nanosec
  - Hard Disk, Terabyte, ~10 millisecond

- **Faster is more expensive (monetary cost and chip area)**
  - SRAM, < 10$ per Megabyte
  - DRAM, < 1$ per Megabyte
  - Hard Disk < 1$ per Gigabyte
  - These sample values (circa ~2011) scale with time

- **Other technologies have their place as well**
  - Flash memory (mature), PC-RAM, MRAM, RRAM (not mature yet)
Why Memory Hierarchy?

- We want **both** fast and large

- But, we cannot achieve both with a single level of memory

- Idea: Have *multiple levels of storage* (progressively bigger and slower as the levels are farther from the processor) and ensure most of the data the processor needs is kept in the fast(er) level(s)
The Memory Hierarchy

Move what you use here

With good locality of reference, memory appears **as fast as** and **as large as**

Back up everything here

- Fast small
- Large but slow

Faster per byte

Cheaper per byte
Memory Hierarchy

- Fundamental tradeoff
  - Fast memory: small
  - Large memory: slow
- Idea: Memory hierarchy

- Latency, cost, size, bandwidth
Memory Hierarchy Example

Locality

- One’s recent past is a very good predictor of their near future

- **Temporal Locality:** If you just did something, it is very likely that you will do the same thing again soon
  - since you are here today, there is a good chance you will be here again and again regularly

- **Spatial Locality:** If you did something, it is very likely you will do something similar/related (in space)
  - every time I find you in this room, you are probably sitting close to the same people AND/OR in closeby seats
Memory Locality

- A “typical” program has a lot of locality in memory references
  - typical programs are composed of “loops”

- Temporal: A program tends to reference the same memory location many times and all within a small window of time

- Spatial: A program tends to reference nearby memory locations within a window of time
  - most notable examples:
    1. instruction memory references → mostly sequential/streaming
    2. references to arrays/vectors → often streaming/strided
Caching Basics: Exploit Temporal Locality

- **Idea:** Store recently accessed data in automatically-managed fast memory (called cache)
- **Anticipation:** same mem. location will be accessed again soon

- **Temporal locality** principle
  - Recently accessed data will be again accessed in the near future
  - This is what Maurice Wilkes had in mind:
    - “The use is discussed of a fast core memory of, say 32000 words as a slave to a slower core memory of, say, one million words in such a way that in practical cases the effective access time is nearer that of the fast memory than that of the slow memory.”
Caching Basics: Exploit Spatial Locality

- **Idea:** Store data in addresses adjacent to the recently accessed one in automatically-managed fast memory
  - Logically divide memory into equal-size blocks
  - Fetch to cache the accessed block in its entirety

- **Anticipation:** nearby memory locations will be accessed soon

- **Spatial locality** principle
  - Nearby data in memory will be accessed in the near future
    - E.g., sequential instruction access, array traversal
  - This is what IBM 360/85 implemented
    - 16 Kbyte cache with 64 byte blocks
The Bookshelf Analogy

- Book in your hand
- Desk
- Bookshelf
- Boxes at home
- Boxes in storage

- Recently-used books tend to stay on desk
  - Comp Arch books, books for classes you are currently taking
  - Until the desk gets full
- Adjacent books in the shelf needed around the same time
  - If I have organized/categorized my books well in the shelf
Caching in a Pipelined Design

- The cache needs to be tightly integrated into the pipeline
  - Ideally, access in 1-cycle so that load-dependent operations do not stall
- High frequency pipeline → Cannot make the cache large
  - But, we want a large cache AND a pipelined design
- Idea: Cache hierarchy

![Diagram of cache hierarchy]

- CPU
- RF
- Level1 Cache
- Level 2 Cache
- Main Memory (DRAM)
A Note on Manual vs. Automatic Management

- **Manual**: Programmer manages data movement across levels -- too painful for programmers on substantial programs
  - “core” vs “drum” memory in the 1950s
  - done in embedded processors (on-chip scratchpad SRAM in lieu of a cache), GPUs (called “shared memory”), ML accelerators, ...

- **Automatic**: Hardware manages data movement across levels, transparently to the programmer
  - ++ programmer’s life is easier
  - the average programmer doesn’t need to know about caches
    - You don’t need to know how big the cache is and how it works to write a “correct” program! (What if you want a “fast” program?)
Caches and Scratchpad in a Modern GPU

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

Caches and Scratchpad in a Modern GPU

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

Caches and Scratchpad in a Modern GPU

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

Nvidia Hopper, 2022

https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/
Cerebras’s Wafer Scale Engine (2019)

- The largest ML accelerator chip
- 400,000 cores
- 18 GB of on-chip memory
- 9 PB/s memory bandwidth

Cerebras WSE
1.2 Trillion transistors
46,225 mm²

Largest GPU
21.1 Billion transistors
815 mm²

https://www.anandtech.com/show/14758/hot-chips-31-live-blogs-cerebras-wafer-scale-deep-learning
https://www.cerebras.net/cerebras-wafer-scale-engine-why-we-need-big-chips-for-deep-learning
Scratchpad Memory in Cerebras WSE

- **Scratchpad Memory**
  - Highly parallel and distributed scratchpad SRAM memory with 2D mesh interconnection fabric across tiles
  - 16-byte read and 8-byte write single-cycle latency
  - 48 KB scratchpad in each tile, totaling 18 GB on the full chip
  - No shared memory

Cerebras’s Wafer Scale Engine-2 (2021)

- The largest ML accelerator chip
- 850,000 cores
- 40 GB of on-chip memory
- 20 PB/s memory bandwidth

Cerebras WSE-2
2.6 Trillion transistors
46,225 mm²

https://cerebras.net/product/#overview

Largest GPU
54.2 Billion transistors
826 mm²
NVIDIA Ampere GA100
A Historical Perspective

Magnetic Drum Memory
Main Memory of 1950s-1960s

Magnetic Core Memory
Main Memory of 1960s-1970s
Using smaller cores and wires, the memory density of core slowly increased, and by the late 1960s a density of about 32 kilobits per cubic foot (about 0.9 kilobits per litre) was typical. However, reaching this density required extremely careful manufacture, which was almost always carried out by hand in spite of repeated major efforts to automate the process. The cost declined over this period from about $1 per bit to about 1 cent per bit. The introduction of the first semiconductor memory chips in the late 1960s, which initially created static random-access memory (SRAM), began to erode the market for core memory. The first successful dynamic random-access memory (DRAM), the Intel 1103, followed in 1970. Its availability in quantity at 1 cent per bit marked the beginning of the end for core memory.[1]
Automatic Management in Memory Hierarchy


Slave Memories and Dynamic Storage Allocation

M. V. WILKES

Summary

The use is discussed of a fast core memory of, say, 32,000 words as a slave to a slower core memory of, say, one million words in such a way that in practical cases the effective access time is nearer that of the fast memory than that of the slow memory.

- “By a slave memory I mean one which automatically accumulates to itself words that come from a slower main memory, and keeps them available for subsequent use without it being necessary for the penalty of main memory access to be incurred again.”
Historical Aside: Other Cache Papers

  http://dl.acm.org/citation.cfm?id=366800

Cache in 1962 (Bloom, Cohen, Porter)
A Modern Memory Hierarchy

Memory Abstraction

Register File
32 words, sub-nsec

L1 cache
~10s of KB, ~nsec

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

L3 cache,
many MBs, even more nsec

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

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

manual/compiler
register spilling

automatic
HW cache
management

automatic
demand
paging
Hierarchical Latency Analysis

- A given memory hierarchy level $i$ has **intrinsic access time** of $t_i$
- It also has **perceived access time** $T_i$ that is longer than $t_i$
- Except for the outer-most hierarchy level, when looking for a given address there is
  - a chance (hit-rate $h_i$) you “hit” and access time is $t_i$
  - a chance (miss-rate $m_i$) you “miss” and access time $t_i + T_{i+1}$
  - $h_i + m_i = 1$
- Thus
  $$T_i = h_i \cdot t_i + m_i \cdot (t_i + T_{i+1})$$
  $$T_i = t_i + m_i \cdot T_{i+1}$$

$h_i$ and $m_i$ are defined to be the hit-rate and miss-rate of only the references that missed at $L_{i-1}$
Hierarchy Design Considerations

- Recursive latency equation
  \[ T_i = t_i + m_i \cdot T_{i+1} \]
- The goal: achieve desired \( T_1 \) within allowed cost
- \( T_i \approx t_i \) is desirable

- Keep \( m_i \) low
  - increasing capacity \( C_i \) lowers \( m_i \), but beware of increasing \( t_i \)
  - lower \( m_i \) by smarter cache management (replacement::anticipate what you don’t need, prefetching::anticipate what you will need)

- Keep \( T_{i+1} \) low
  - faster outer hierarchy levels can help, but beware of increasing cost
  - introduce intermediate hierarchy levels as a compromise
Intel Pentium 4 Example

Intel Pentium 4 Example

https://download.intel.com/newsroom/kits/40thanniversary/gallery/images/Pentium_4_6xx-die.jpg
Intel Pentium 4 Example

- 90nm P4, 3.6 GHz

- L1 D-cache
  - $C_1 = 16$ kB
  - $t_1 = 4$ cyc int / 9 cycle fp

- L2 D-cache
  - $C_2 = 1024$ kB
  - $t_2 = 18$ cyc int / 18 cyc fp

- Main memory
  - $t_3 = \sim 50$ns or 180 cyc

- Notice
  - best case latency is not 1
  - worst case access latencies are into 500+ cycles

\[ T_i = t_i + m_i \cdot T_{i+1} \]

- if $m_1=0.1$, $m_2=0.1$
  - $T_1=7.6$, $T_2=36$

- if $m_1=0.01$, $m_2=0.01$
  - $T_1=4.2$, $T_2=19.8$

- if $m_1=0.05$, $m_2=0.01$
  - $T_1=5.00$, $T_2=19.8$

- if $m_1=0.01$, $m_2=0.50$
  - $T_1=5.08$, $T_2=108$
Cache Basics and Operation
Cache

- Any structure that “memoizes” used (or produced) data
  - to avoid repeating the long-latency operations required to reproduce/fetch the data from scratch
  - e.g., a web cache

- Most commonly in the processor design context: an automatically-managed memory structure
  - e.g., memoize in fast SRAM the most frequently or recently accessed DRAM memory locations to avoid repeatedly paying for the DRAM access latency
Conceptual Picture of a Cache

A key question: How to map chunks of the main memory address space to blocks in the cache?

Which location in cache can a given “main memory chunk” be placed in?
Logical Organization of a Cache (II)

- A key question: How to map chunks of the main memory address space to blocks in the cache?
  - Which location in cache can a given “main memory chunk” be placed in?

---

Caching Basics

- **Block (line):** Unit of storage in the cache
  - Memory is logically divided into blocks that map to potential locations in the cache

- On a reference:
  - **HIT:** If in cache, use cached data instead of accessing memory
  - **MISS:** If not in cache, bring block into cache
    - May have to evict some other block

- Some important cache design decisions
  - **Placement:** where and how to place/find a block in cache?
  - **Replacement:** what data to remove to make room in cache?
  - **Granularity of management:** large or small blocks? Subblocks?
  - **Write policy:** what do we do about writes?
  - **Instructions/data:** do we treat them separately?
Cache Abstraction and Metrics

- Cache hit rate = (# hits) / (# hits + # misses) = (# hits) / (# accesses)
- Average memory access time (AMAT)
  = (hit-rate * hit-latency) + (miss-rate * miss-latency)
- Important Aside: *Is reducing AMAT always beneficial for performance?*
A Basic Hardware Cache Design

- We will start with a basic hardware cache design

- Then, we will examine a multitude of ideas to make it better (i.e., higher performance)
Blocks and Addressing the Cache

- Main memory logically divided into fixed-size chunks (**blocks**).
- **Cache** can house only a **limited** number of blocks.
Blocks and Addressing the Cache

- Main memory logically divided into fixed-size chunks *(blocks)*
- **Cache** can house only a **limited** number of blocks

- Each **block address** maps to a potential location in the cache, determined by the **index bits** in the address
  - used to index into the tag and data stores

- Cache access:
  1) index into the tag and data stores with index bits in address
  2) check valid bit in tag store
  3) compare tag bits in address with the stored tag in tag store

- If the stored tag is valid and matches the tag of the block, then the block is in the cache (cache hit)
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
Direct-Mapped Cache: Placement and Access

- Assume byte-addressable main memory: 256 bytes, 8-byte blocks → 32 blocks in mem
- Assume cache: 64 bytes, 8 blocks
  - Direct-mapped: A block can go to only one location
  - Blocks with same index contend for the same cache location
    - Cause conflict misses when accessed consecutively

| Block: 00000 | Block: 00001 |
| Block: 00010 | Block: 00011 |
| Block: 00100 | Block: 00101 |
| Block: 00110 | Block: 00111 |
| Block: 01000 | Block: 01001 |
| Block: 01010 | Block: 01011 |
| Block: 01100 | Block: 01101 |
| Block: 01110 | Block: 01111 |
| Block: 10000 | Block: 10001 |
| Block: 10010 | Block: 10011 |
| Block: 10100 | Block: 10101 |
| Block: 10110 | Block: 10111 |
| Block: 11000 | Block: 11001 |
| Block: 11010 | Block: 11011 |
| Block: 11100 | Block: 11101 |
| Block: 11110 | Block: 11111 |
Direct-Mapped Caches

- **Direct-mapped cache:** Two blocks in memory that map to the same index in the cache cannot be present in the cache at the same time
  - One index $\rightarrow$ one entry

- Can lead to 0% hit rate if more than one block accessed in an interleaved manner map to the same index
  - Assume addresses A and B have the same index bits but different tag bits
  - A, B, A, B, A, B, A, B, ... $\rightarrow$ conflict in the cache index
  - All accesses are conflict misses
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
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
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
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)

- 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
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
  - 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
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.


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

| 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 $\rightarrow$ **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

Source: https://www.anandtech.com/show/16252/mac-mini-apple-m1-tested
Deeper and Larger Cache Hierarchies

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


Die shot interpretation by Locuza, October 2021

Intel Alder Lake, 2021
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

AMD’s 3D Last Level Cache (2021)

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
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

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

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

Deeper and Larger Cache Hierarchies

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/
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.
Memory in the NVIDIA H100 GPU

- **SM (Streaming Multiprocessor)**: Control, Registers, Core, SM-to-SM, Shared Memory, L1 Cache, Constant Cache
- L2 Cache: 60 MB
- Global Memory: 80 GB, 3 TB/s

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

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

![Graph](hit_rate_vs_block_size.png)
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

  - 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

**Diagram Description**

- **Tag store**
  - Four comparators labeled with question marks (=?)
  - Logic block connected to comparators
  - Hit? output

- **Data store**
  - Four data blocks connected to comparators

- **MUX**
  - Two MUX blocks, with one labeled "byte in block"
  - Address input

**Addressponents**

- Tag: 4 bits
- Index: 1 bit
- Byte in block: 3 bits
Higher Associativity (3-way)

- 3-way

![Diagram of 3-way associativity]

- Tag store
- Data store
- Logic
- MUX

Address

<table>
<thead>
<tr>
<th>tag</th>
<th>index</th>
<th>byte in block</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 bits</td>
<td>1 b</td>
<td>3 bits</td>
</tr>
</tbody>
</table>
Recall: 8-way Fully Associative Cache

Tag store

Data store

Address

5 bits

3 bits

byte in block

byte in block
7-way Fully Associative Cache

Tag store

Data store

Address

5 bits 3 bits

byte in block
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

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

Better code

```plaintext
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, ...
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
Research Opportunities
Research Opportunities

If you are interested in **doing research** in Computer Architecture, Security, Systems & Bioinformatics:

- Email me and Prof. Mutlu with your interest
- Take the seminar course and the “Computer Architecture” course
- Do readings and assignments on your own & **talk with us**

There are **many exciting projects and research positions**, e.g.:

- Novel memory/storage/computation/communication systems
- New execution paradigms (e.g., in-memory computing)
- Hardware security, safety, reliability, predictability
- GPUs, TPUs, FPGAs, PIM, heterogeneous systems, ...
- Security-architecture-reliability-energy-performance interactions
- Architectures for genomics/proteomics/medical/health/AI/ML
- A limited list is here: [https://safari.ethz.ch/theses/](https://safari.ethz.ch/theses/)
- [https://people.inf.ethz.ch/omutlu/projects.htm](https://people.inf.ethz.ch/omutlu/projects.htm)
Bachelor’s Seminar in Computer Architecture

- Fall 2024 (offered every Fall and Spring Semester)
- 2 credit units

- Rigorous seminar on fundamental and cutting-edge topics in computer architecture
- Critical paper presentation, review, and discussion of seminal and cutting-edge works in computer architecture
  - We will cover many ideas & issues, analyze their tradeoffs, perform critical thinking and brainstorming

- Participation, presentation, synthesis report, lots of discussion
- You can register for the course online
- https://safari.ethz.ch/architecture_seminar
## Fall 2021 Lectures/Schedule

<table>
<thead>
<tr>
<th>Week</th>
<th>Date</th>
<th>Livestream</th>
<th>Lecture</th>
<th>Readings</th>
<th>Assignments</th>
</tr>
</thead>
<tbody>
<tr>
<td>W1</td>
<td>23.09 Thu.</td>
<td>YouTube Live</td>
<td>L1a: Course Logistics (PDF) (PPT)</td>
<td>Suggested</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>L1b: Introduction and Basics (PDF) (PPT)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>L1c: Architectural Design Fundamentals (PDF) (PPT)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>YouTube Video</td>
<td></td>
<td></td>
</tr>
<tr>
<td>W2</td>
<td>30.09 Thu.</td>
<td>YouTube Live</td>
<td>L2: GateKeeper (PDF) (PPT)</td>
<td>Suggested</td>
<td></td>
</tr>
<tr>
<td>W3</td>
<td>07.10 Thu.</td>
<td>YouTube Live</td>
<td>L3: RowClone (Processing using DRAM) (PDF) (PPT)</td>
<td>Suggested</td>
<td></td>
</tr>
<tr>
<td>W4</td>
<td>14.10 Thu.</td>
<td>YouTube Live</td>
<td>L4: Memory Channel Partitioning (PDF) (PPT)</td>
<td>Suggested</td>
<td></td>
</tr>
<tr>
<td>W5</td>
<td>4.11 Thu.</td>
<td>YouTube Live</td>
<td>S1.1: Bottleneck Identification and Scheduling in Multithreaded Applications, ASPLOS 2012 (PDF) (PPT)</td>
<td>Mentioned</td>
<td></td>
</tr>
<tr>
<td>W6</td>
<td>11.11 Thu.</td>
<td></td>
<td>S2.1: Profiling a Warehouse-Scale Computer, ISCA 2015 (PDF) (PPT)</td>
<td>Mentioned</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>S2.2: Understanding Sources of Inefficiency in General-Purpose Chips, ISCA 2016 (PDF) (ODP) (PPT)</td>
<td>Mentioned</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>S3.2: Branch Runahead: An Alternative to Branch Prediction for Impossible to Predict Branches, MICRO 2021 (PDF) (PPT)</td>
<td>Mentioned</td>
<td></td>
</tr>
<tr>
<td>W9</td>
<td>02.12 Thu.</td>
<td>YouTube Live</td>
<td>S5.1: Quantifying Server Memory Frequency Margin and Using It to Improve Performance in HPC Systems, ISCA 2021 (PDF) (PPT)</td>
<td>Mentioned</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>S5.2: SIMDRA: An End-to-End Framework for Bit-Serial SIMD Computing in DRAM, ASPLOS 2021 (PDF) (KEY)</td>
<td>Mentioned</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>S6.2: BlockHammer: Preventing RowHammer at Low Cost by Blacklisting Rapidly-Accessed DRAM Rows, HPCA 2021 (PDF) (PPT)</td>
<td>Mentioned</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>S8.2: SquiggleFilter: An Accelerator for Portable Virus Detection, MICRO 2021 (PPT) (PDF)</td>
<td>Mentioned</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>S8.3: Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks, ASPLOS 2018 (PPT) (PDF)</td>
<td>Mentioned</td>
<td></td>
</tr>
<tr>
<td>Week</td>
<td>Date</td>
<td>Livestream</td>
<td>Lecture</td>
<td>Readings</td>
<td>Assignments</td>
</tr>
<tr>
<td>------</td>
<td>--------</td>
<td>-------------</td>
<td>-------------------------------------------------------------------------</td>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>W1</td>
<td>24.02  Thu.</td>
<td>YouTube Live</td>
<td>L1a: Course Logistics [PDF] [PPT]</td>
<td>Suggested</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>L1b: Introduction and Basics [PDF] [PPT]</td>
<td>Suggested</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>L1c: Architectural Design Fundamentals [PDF] [PPT]</td>
<td>Suggested</td>
<td></td>
</tr>
<tr>
<td>W2</td>
<td>03.03  Thu.</td>
<td>YouTube Live</td>
<td>L2: Memory-Centric Computing [PDF] [PPT]</td>
<td>Suggested</td>
<td></td>
</tr>
<tr>
<td>W3</td>
<td>10.03  Thu.</td>
<td>YouTube Live</td>
<td>L3: Memory-Centric Computing II [PDF] [PPT]</td>
<td>Suggested</td>
<td></td>
</tr>
<tr>
<td>W4</td>
<td>17.03  Thu.</td>
<td>YouTube Live</td>
<td>L4: Memory-Centric Computing III [PDF] [PPT]</td>
<td>Suggested</td>
<td></td>
</tr>
<tr>
<td>W5</td>
<td>24.03  Thu.</td>
<td>YouTube Live</td>
<td>L5: Accelerating Genome Analysis [PDF] [PPT]</td>
<td>Suggested</td>
<td></td>
</tr>
<tr>
<td>W6</td>
<td>31.03  Thu.</td>
<td>YouTube Live</td>
<td>L6a: Rethinking Virtual Memory I [PDF] [PPT]</td>
<td>Suggested</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>L6b: Rethinking Virtual Memory II [PDF] [PPT]</td>
<td>Suggested</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>S1.2: SISA: Set-Centric Instruction Set Architecture for Graph Mining on Processing-in-Memory Systems, MICRO 2021 [PDF] [PPT]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>W8</td>
<td>14.04  Thu.</td>
<td>YouTube Live</td>
<td>S2.1: Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors, ISCA 2014 [PDF] [PPT]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>S2.2: Uncovering In-DRAM RowHammer Protection Mechanisms: A New Methodology, Custom RowHammer Patterns, and Implications, MICRO 2021 [PDF] [PPT]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>W9</td>
<td>28.04  Thu.</td>
<td>YouTube Live</td>
<td>S3.1: ProSE: The Architecture and Design of a Protein Discovery Engine, ASPLOS 2022 [PDF] [PPT]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>S3.2: GenBlore: a high-performance in-storage processing system for genome sequence analysis, ASPLOS 2022 [PDF] [PPT]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>W10</td>
<td>05.05  Thu.</td>
<td>YouTube Live</td>
<td>S4.1: Focusing processor policies via critical-path prediction, ISCA 2001 [PDF] [PPT]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>S4.2: Pythia: A Customizable Hardware Prefetching Framework Using Online Reinforcement Learning, MICRO 2021 [PDF]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>W11</td>
<td>12.05  Thu.</td>
<td>YouTube Live</td>
<td>S5.1: Ten Lessons From Three Generations Shaped Google’s TPUv4I, ISCA 2021 [PDF] [PPT]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>S5.2: Google Neural Network Models for Edge Devices: Analyzing and Mitigating Machine Learning Inference Bottlenecks, PACT 2021 [PDF]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>W12</td>
<td>19.05  Thu.</td>
<td>YouTube Live</td>
<td>S6.1: Hash, Don’t Cache (the Page Table), SIGMETRICS 2016 [PDF] [PPT]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>S6.2: Processing-In-Memory Enabled Graphics Processors for 3D Rendering, HPCA 2017 [PDF]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>W13</td>
<td>02.06  Thu.</td>
<td>YouTube Live</td>
<td>S7.1: QUAC-TRNG: High-Throughput True Random Number Generation Using Quadruple Row Activation in Commodity DRAM Chips, ISCA 2021 [PDF] [PPT]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>S7.2: A Deeper Look into RowHammer’s Sensitivities: Experimental Analysis of Real DRAM Chips and Implications on Future Attacks and Defenses, MICRO 2021 [PDF] [PPT]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>S7.3: A2: Analog malicious hardware, IEEE Symposium on Security and Privacy 2018 [PDF] [PPT]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Research Opportunities

- If you are interested in doing research in Computer Architecture, Security, Systems & Bioinformatics:
  - Email me and Prof. Mutlu with your interest
  - Take the seminar course and the “Computer Architecture” course
  - Do readings and assignments on your own & talk with us

- There are many exciting projects and research positions, e.g.:
  - Novel memory/storage/computation/communication systems
  - New execution paradigms (e.g., in-memory computing)
  - Hardware security, safety, reliability, predictability
  - GPUs, TPUs, FPGAs, PIM, heterogeneous systems, ...
  - Security-architecture-reliability-energy-performance interactions
  - Architectures for genomics/proteomics/medical/health/AI/ML
  - A limited list is here: [https://safari.ethz.ch/theses/](https://safari.ethz.ch/theses/)

[https://people.inf.ethz.ch/omutlu/projects.htm](https://people.inf.ethz.ch/omutlu/projects.htm)
SAFARI Introduction & Research

Computer architecture, HW/SW, systems, bioinformatics, security, memory

Think BIG, Aim HIGH!

https://www.youtube.com/watch?v=mV2OuB2djEs
Lecture 24: Memory Hierarchy and Caches

Frank K. Gürkaynak
Mohammad Sadrosadati
Prof. Onur Mutlu

ETH Zürich
Spring 2024
24 May 2024
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) 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
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:
- Misses = 4
- Stalls = 4

MLP-Aware replacement:
- Misses = 6
- Stalls = 2
Recommended: MLP-Aware Cache Replacement

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


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
  - ...
Victim Cache: Reducing Conflict Misses


**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
Lectures on Cache Optimizations (II)

Peripheral Logic for True Multiporting

CPU OR I/O DEVICE
"L"

DATA

ADDRESS

R/W

SEMAPHORE SELECT

SEMAPHORE CELLS

DUAL/PORT RAM MEMORY CELLS

L ADDRESS DECODER

R ADDRESS DECODER

DATA

ADDRESS

R/W

CPU OR I/O DEVICE
"R"

https://www.youtube.com/watch?v=55oYBm9cifI&list=PL5Q2soXY2Zi9JXe3ywQMhylk_d5dl-TM7&index=6

1,437 views • Sep 29, 2018

Onur Mutlu Lectures
16.3K subscribers
Lectures on Cache Optimizations (III)

Fewest Misses ≠ Best Performance

Hit/Miss: H H H H M M M M M
Time: stall
Belady’s OPT replacement
Misses = 6
Stalls = 2

Hit/Miss: H M M M M M H H H H
Time: stall
MLP-Aware replacement
Misses = 6
Stalls = 2

9,737 views • Mar 5, 2015

https://www.youtube.com/watch?v=jDHx2K9HxlM&list=PL5PHm2jkkXmi5Cxxl7b3JCL1TWybTDtKq&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=PL5Q2soXY2Zi9OhoVQBXJYFIZywZXCPI4M&_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

https://www.youtube.com/onurmutlulectures
Multi-Core Issues in Caching
Caches in a Multi-Core System
Caches in a Multi-Core System

Source: https://www.anandtech.com/show/16252/mac-mini-apple-m1-tested
Caches in a Multi-Core System

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

Intel Alder Lake, 2021
Caches in a Multi-Core System

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

Caches in a Multi-Core System

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/ggAYMx34euU
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

Caches in a Multi-Core System

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

Nvidia Ampere, 2020

Caches in a Multi-Core System

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 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: Problem with Shared Caches

Example: Problem with Shared Caches

Example: Problem with Shared Caches

Resource Sharing vs. Partitioning

- Sharing improves throughput
  - 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
Computer Architecture
Lecture 15:
Multi-Core Cache Management

Prof. Onur Mutlu
ETH Zürich
Fall 2017
15 November 2017

https://www.youtube.com/watch?v=7_Tqlw8gxOU&list=PL5Q2soXY2Zi9OhQVQBXYFZWzwXZCPl4M_&index=17
Page Coloring

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

Approaches to Reuse Prediction

1. Group Blocks
   - PC 1: ABST
   - PC 2: ABST

2. Learn group behavior
   - PC 1: ABST
   - PC 2: ABST

3. Predict reuse
   - PC 1: A → C
   - PC 2: B → C

1. Same group → same reuse behavior
2. No control over number of high-reuse blocks
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](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](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=PL5Q2soXY2Zi9OhoVQBXYFIZywZXCPI4M_&index=17](https://www.youtube.com/watch?v=7_Tqlw8gxOU&list=PL5Q2soXY2Zi9OhoVQBXYFIZywZXCPI4M_&index=17)

[https://www.youtube.com/onurmutlulectures](https://www.youtube.com/onurmutlulectures)
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

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

https://www.youtube.com/onurmutlulectures
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

P1

Interconnection Network

P2

1000

Main Memory

x 1000
The Cache Coherence Problem

$$P_1$$

$$P_2$$

Id r2, x

$$1000$$

Interconnection Network

Main Memory

Id r2, x

$$1000$$
The Cache Coherence Problem

P1

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

2000

P2

ld r2, x
1000

Interconnection Network

1000

Main Memory
The Cache Coherence Problem

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

ld r2, x
Should NOT load 1000

ld r5, x
Hardware Cache Coherence

- Basic idea:
  - A processor/cache broadcasts its write/update to a memory location to all other processors
  - Another processor/cache that has the location either updates or invalidates its local copy
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
Lecture on Memory Ordering & Consistency

For P1:
A appeared to happen before X

For P2:
X appeared to happen before A

P1's VIEW
A → B → X
A → X

P2's VIEW
X → Y → A
X → A

Both cannot be correct!
(from memory's perspective)

P1 and P2
Saw an inconsistent order of operations
in memory

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

https://www.youtube.com/watch?v=Suy09mzTbiQ&list=PL5Q2soXY2Zi9idyIlGbxBz7xRPS-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=T9WlyzeaII&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

https://www.youtube.com/onurmutlulecutes
Additional Slides: Cache Coherence
Two Cache Coherence Methods

- How do we ensure that the proper caches are updated?

- **Snoopy Bus** [Goodman ISCA 1983, Papamarcos+ ISCA 1984]
  - Bus-based, **single point of serialization** *for all memory requests*
  - Processors observe other processors’ actions
    - E.g.: P1 makes “read-exclusive” request for A on bus, P0 sees this and invalidates its own copy of A

- **Directory** [Censier and Feautrier, IEEE ToC 1978]
  - **Single point of serialization per block**, distributed among nodes
  - Processors make explicit requests for blocks
  - Directory tracks which caches have each block
  - Directory coordinates invalidations and updates
    - E.g.: P1 asks directory for exclusive copy, directory asks P0 to invalidate, waits for ACK, then responds to P1
Directory Based Coherence

- Idea: A logically-central directory keeps track of where the copies of each cache block reside. Caches consult this directory to ensure coherence.

- 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 and arrange the supply of data
  - On a write: invalidate all caches that have the block and reset their bits
  - Have an “exclusive bit” associated with each block in each cache (so that the cache can update the exclusive block silently)
Example directory based scheme

\( P = 4 \)

\[ \begin{array}{c|cccc|c}
\hline
 & 0 & 0 & 0 & 0 & 0 \\
\hline
\end{array} \]

Exclusive bit

No cache has the block

1. \( P_1 \) takes a read miss to block A

\[ \begin{array}{c|cccc|c}
0 & 0 & 0 & 0 & 0 & 0 \\
\hline
\end{array} \rightarrow \begin{array}{c|cccc|c}
0 & 1 & 0 & 0 & 0 & 0 \\
\hline
\end{array} \]

2. \( P_3 \) takes a read miss

\[ \begin{array}{c|cccc|c}
0 & 1 & 0 & 0 & 0 & 0 \\
\hline
\end{array} \]
3. P2 takes a write miss
   \( \rightarrow \) Invalidate P1 & P3's caches
   \( \rightarrow \) Write request \( \rightarrow \) P2 has the exclusive copy of the block
   now. Set the Exclusive bit.
   \( \rightarrow \) P2 can now update the block without notifying any other processor or the directory.
   \( \rightarrow \) P2 needs to have a bit in its cache indicating it can perform exclusive updates to that block.
   \( \rightarrow \) Private/exclusive bit per cache block.

4. P3 takes a write miss
   \( \rightarrow \) Mem Controller requests block from P2
   \( \rightarrow \) Mem Controller gives block to P3
   \( \rightarrow \) P2 invalidates its copy.

5. P2 takes a read miss
   \( \rightarrow \) P3 supplies it.
Maintaining Coherence

- Need to guarantee that all processors see a consistent value (i.e., consistent updates) for the same memory location.

- Writes to location A by P0 should be seen by P1 (eventually), and all writes to A should appear in some order.

- Coherence needs to provide:
  - **Write propagation**: guarantee that updates will propagate.
  - **Write serialization**: provide a consistent order seen by all processors for the same memory location.

- Need a global point of serialization for this write ordering.
Coherence: Update vs. Invalidate

- How can we *safely update replicated data*?
  - Option 1 (Update protocol): push an update to all copies
  - Option 2 (Invalidate protocol): ensure there is only one copy (local), update it

- **On a Read:**
  - If local copy is Invalid, put out request
  - (If another node has a copy, it returns it, otherwise memory does)
Coherence: Update vs. Invalidate (II)

- **On a Write:**
  - Read block into cache as before

**Update Protocol:**
- Write to block, and simultaneously broadcast written data and address to sharers
- (Other nodes update the data in their caches if block is present)

**Invalidate Protocol:**
- Write to block, and simultaneously broadcast invalidation of address to sharers
- (Other nodes invalidate block in their caches if block is present)
Update vs. Invalidate Tradeoffs

- Which one is better? Update or invalidate?
  - Write frequency and sharing behavior are critical

- Update
  + If sharer set is constant and updates are infrequent, avoids the cost of invalidate-reatquire (broadcast update pattern)
  - If data is rewritten without intervening reads by other cores, updates would be useless
  - Write-through cache policy → bus can become a bottleneck

- Invalidate
  + After invalidation, core has exclusive access rights
  + Only cores that keep reading after each write retain a copy
  - If write contention is high, leads to ping-ponging (rapid invalidation-reatquire traffic from different processors)
Additional Slides: Memory Interference
Inter-Thread/Application Interference

- **Problem:** Threads share the memory system, but memory system does not distinguish between threads’ requests

- **Existing memory systems**
  - Free-for-all, shared based on demand
  - Control algorithms thread-unaware and thread-unfair
  - Aggressive threads can deny service to others
  - Do not try to reduce or control inter-thread interference
Unfair Slowdowns due to Interference

Uncontrolled Interference: An Example

Multi-Core Chip

Shared DRAM Memory System

unfairness
A Memory Performance Hog

// initialize large arrays A, B
for (j=0; j<N; j++) {
    index = rand();
    A[index] = B[index];
    ...
}

// initialize large arrays A, B
for (j=0; j<N; j++) {
    index = j*linesize;
    A[index] = B[index];
    ...
}

STREAM  
- Sequential memory access
- Very high row buffer locality (96% hit rate)
- Memory intensive

RANDOM  
- Random memory access
- Very low row buffer locality (3% hit rate)
- Similarly memory intensive

What Does the Memory Hog Do?

Row size: 8KB, cache block size: 64B

128 (8KB/64B) requests of T0 serviced before T1

DRAM Controllers

- A row-conflict memory access takes significantly longer than a row-hit access

- Current controllers take advantage of the row buffer

- Commonly used scheduling policy (FR-FCFS) [Rixner 2000]*
  (1) **Row-hit first:** Service row-hit memory accesses first
  (2) **Oldest-first:** Then service older accesses first

- This scheduling policy aims to maximize DRAM throughput
  - But, it is unfair when multiple threads share the DRAM system

---

Effect of the Memory Performance Hog

Results on Intel Pentium D running Windows XP
(Similar results for Intel Core Duo and AMD Turion, and on Fedora Linux)

Greater Problem with More Cores

- Vulnerable to denial of service (DoS)
- Unable to enforce priorities or SLAs
- Low system performance

**Uncontrollable, unpredictable system**
Greater Problem with More Cores

- Vulnerable to denial of service (DoS)
- Unable to enforce priorities or SLAs
- Low system performance

Uncontrollable, unpredictable system
Distributed DoS in Networked Multi-Core Systems

Cores connected via packet-switched routers on chip

~5000X latency increase

More on Memory Performance Attacks

Thomas Moscibroda and Onur Mutlu, "Memory Performance Attacks: Denial of Memory Service in Multi-Core Systems"
More on Interconnect Based Starvation

- Boris Grot, Stephen W. Keckler, and Onur Mutlu,
  "Preemptive Virtual Clock: A Flexible, Efficient, and Cost-effective QOS Scheme for Networks-on-Chip"
  

Preemptive Virtual Clock: A Flexible, Efficient, and Cost-effective QOS Scheme for Networks-on-Chip

Boris Grot
Department of Computer Sciences
The University of Texas at Austin
{bgrot, skeckler}@cs.utexas.edu

Stephen W. Keckler

Onur Mutlu†
†Computer Architecture Laboratory (CALCM)
Carnegie Mellon University
onur@cmu.edu
Energy Comparison of Memory Technologies
The Problem: Energy

- Faster is more energy-efficient
  - SRAM, ~5 pJ
  - DRAM, ~40-140 pJ
  - PCM-DIMM (Intel Optane DC DIMM), ~80-540 pJ
  - PCM-SSD, ~120 µJ
  - Flash memory, ~250 µJ
  - Hard Disk, ~60 mJ

- Other technologies have their place as well
  - MRAM, RRAM, STT-MRAM, memristors, ... (not mature yet)
## The Problem (Table View): Energy

<table>
<thead>
<tr>
<th>Memory Device</th>
<th>Capacity</th>
<th>Latency</th>
<th>Cost per Megabyte</th>
<th>Energy per access</th>
<th>Energy per byte access</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRAM</td>
<td>&lt; 1 KByte</td>
<td>sub-nanosec</td>
<td></td>
<td>~5 pJ</td>
<td>~1.25 pJ</td>
</tr>
<tr>
<td>SRAM</td>
<td>KByte~MByte</td>
<td>~nanosec</td>
<td>&lt; 0.3$</td>
<td>~40-140 pJ</td>
<td>~10-35 pJ</td>
</tr>
<tr>
<td>DRAM</td>
<td>Gigabyte</td>
<td>~50 nanosec</td>
<td>&lt; 0.006$</td>
<td>~80-540 pJ</td>
<td>~20-135 pJ</td>
</tr>
<tr>
<td>PCM-DIMM (Intel Optane DC DIMM)</td>
<td>Gigabyte</td>
<td>~300 nanosec</td>
<td>&lt; 0.004$</td>
<td>~120 µJ</td>
<td>~30 nJ</td>
</tr>
<tr>
<td>PCM-SSD (Intel Optane SSD)</td>
<td>Gigabyte~Terabyte</td>
<td>~6-10 µs</td>
<td>&lt; 0.002$</td>
<td>~250 µJ</td>
<td>~61 nJ</td>
</tr>
<tr>
<td>Flash memory</td>
<td>Gigabyte~Terabyte</td>
<td>~50-100 µs</td>
<td>&lt; 0.00008$</td>
<td>~60 mJ</td>
<td>~15 µJ</td>
</tr>
<tr>
<td>Hard Disk</td>
<td>Terabyte</td>
<td>~10 millisec</td>
<td>&lt; 0.00003$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Bigger is slower
- Faster is more energy-efficient

Faster is more expensive (and expensive)

These sample values (circa ~2022) scale with time
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

2^30 Word Main Memory

2^3 Word Cache

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

Address

<table>
<thead>
<tr>
<th>Address</th>
<th>11...1111100</th>
<th>11...1111000</th>
<th>11...1110100</th>
<th>11...1110000</th>
<th>11...1110000</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>mem[0xFF...FC]</td>
<td>mem[0xFF...F8]</td>
<td>mem[0xFF...F4]</td>
<td>mem[0xFF...F0]</td>
<td>mem[0xFF...E4]</td>
</tr>
<tr>
<td></td>
<td>mem[0xFF...EC]</td>
<td>mem[0xFF...E8]</td>
<td>mem[0xFF...E4]</td>
<td>mem[0xFF...E0]</td>
<td>mem[0xFF...E0]</td>
</tr>
<tr>
<td>00...00100100</td>
<td>mem[0x00...24]</td>
<td>mem[0x00...20]</td>
<td>mem[0x00...1C]</td>
<td>mem[0x00...18]</td>
<td>mem[0x00...14]</td>
</tr>
<tr>
<td>00...00100000</td>
<td>mem[0x00...10]</td>
<td>mem[0x00...0C]</td>
<td>mem[0x00...08]</td>
<td>mem[0x00...04]</td>
<td>mem[0x00...00]</td>
</tr>
</tbody>
</table>
Direct Mapped Cache Hardware

Memory Address

Tag  Set  Byte Offset

00

27

3

V  Tag  Data

8-entry x (1+27+32)-bit SRAM

Data

Hit
# MIPS assembly code

```assembly
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** = 192
Direct Mapped Cache Performance

# 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:

<table>
<thead>
<tr>
<th>V</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 00...00</td>
<td>mem[0x00...0C]</td>
<td></td>
</tr>
<tr>
<td>1 00...00</td>
<td>mem[0x00...08]</td>
<td></td>
</tr>
<tr>
<td>1 00...00</td>
<td>mem[0x00...04]</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Miss Rate = 3/15 = 20%

Temporal Locality
Compulsory Misses
Direct Mapped Cache: Conflict

### MIPS assembly code

```plaintext
# 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:
```

### Data Tag V

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Tag</th>
<th>Set</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>00...01</td>
<td>001</td>
<td>000</td>
<td>mem[0x00...04]</td>
</tr>
</tbody>
</table>

### Tag Set

<table>
<thead>
<tr>
<th>Set</th>
<th>Byte</th>
<th>Offset</th>
<th>Memory Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>0</td>
<td>01</td>
<td>mem[0x00...24]</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>01</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td></td>
</tr>
</tbody>
</table>

### Miss Rate

\[
\text{Miss Rate} = 194
\]
Direct Mapped Cache: Conflict

Miss Rate = 10/10 = 100%

Conflict Misses

# 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:
```
N-Way Set Associative Cache
N-way Set Associative Performance

```mips
# 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 =

<table>
<thead>
<tr>
<th>Way 1</th>
<th>Way 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>V  Tag Data</td>
<td>V  Tag Data Set 3 Set 2 Set 1 Set 0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1 00...10  mem[0x00...24]</td>
<td>1 00...00  mem[0x00...04]</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
N-way Set Associative Performance

Miss Rate = 2/10
= 20%

Associativity reduces conflict misses

# MIPS assembly code

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

done:
```

Way 1

<table>
<thead>
<tr>
<th>V</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00...10</td>
<td>mem[0x00...24]</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Way 0

<table>
<thead>
<tr>
<th>V</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00...00</td>
<td>mem[0x00...04]</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Set 3
Set 2
Set 1
Set 0
Fully Associative Cache

- No conflict misses
- Expensive to build
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 =**

```
Memory Address  Tag  Set  Offset Offset
00...00 0 11 00
```

```
V  Tag  Data
0 00...00  mem[0x00...0C] mem[0x00...08] mem[0x00...04] mem[0x00...00]
```

```
Hit
00...00 0 11 00
32 32 32 32
```

```
Set 1
Set 0
```

```
Data
```

```
Set 1
Set 0
```

```
Data
```
Direct Mapped Cache Performance

```assembly
loop:
    addi $t0, $0, 5
    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: $S = B / N$

<table>
<thead>
<tr>
<th>Organization</th>
<th>Number of Ways (N)</th>
<th>Number of Sets (S = B/N)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Mapped</td>
<td>1</td>
<td>B</td>
</tr>
<tr>
<td>N-Way Set Associative</td>
<td>1 &lt; N &lt; B</td>
<td>B / N</td>
</tr>
<tr>
<td>Fully Associative</td>
<td>B</td>
<td>1</td>
</tr>
</tbody>
</table>
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

# 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)

<table>
<thead>
<tr>
<th>V</th>
<th>U</th>
<th>Tag</th>
<th>Data</th>
<th>V</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00...010</td>
<td>mem[0x00...24]</td>
<td>1</td>
<td>00...000</td>
<td>mem[0x00...04]</td>
</tr>
</tbody>
</table>

Set 3 (11)  
Set 2 (10)  
Set 1 (01)  
Set 0 (00)

(b)

<table>
<thead>
<tr>
<th>V</th>
<th>U</th>
<th>Tag</th>
<th>Data</th>
<th>V</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>00...010</td>
<td>mem[0x00...24]</td>
<td>1</td>
<td>00...101</td>
<td>mem[0x00...54]</td>
</tr>
</tbody>
</table>

Set 3 (11)  
Set 2 (10)  
Set 1 (01)  
Set 0 (00)
Slides for Future Lectures
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 ordering 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 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?

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
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
  - 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
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.


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 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
Subblock (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
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
- Outer 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 & pipeline structure
  - Small, lower associativity; latency is critical
  - Tag store and data store usually accessed in parallel

- Second- and third-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

Source: https://www.anandtech.com/show/16252/mac-mini-apple-m1-tested
Deeper and Larger Cache Hierarchies

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

AMD’s 3D Last Level Cache (2021)

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

Deeper and Larger Cache Hierarchies

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

Nvidia Ampere, 2020

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

Deeper and Larger Cache Hierarchies

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/
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.
Memory in the NVIDIA H100 GPU

- SM (Streaming Multiprocessor)
  - Control
  - Registers
  - Core
  - Constant Cache
  - Shared Memory
  - L1 Cache

- Direct copy: SM-to-SM

- L2 Cache: 60 MB
- Global Memory: 3 TB/s, 80 GB

≈ 1 cycle
≈ 5 cycles
≈ 5 cycles
≈ 500 cycles

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 (not including tag) capacity
  - bigger 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
Benefit of Larger Caches Widely Varies

- Benefits of cache size widely varies across applications

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 → exploit temporal locality not 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?

```
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

- 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

![Diagram showing the 4-way associative memory structure](image-url)
Higher Associativity (3-way)

- 3-way

![Diagram](image)

- Tag store
- Data store
- MUX
- Logic
- Hit?
- Address

<table>
<thead>
<tr>
<th>tag</th>
<th>index</th>
<th>byte in block</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 bits</td>
<td>1 b</td>
<td>3 bits</td>
</tr>
</tbody>
</table>
Recall: 8-way Fully Associative Cache

Tag store

Data store

Logic

Hit?

Address
tag

5 bits

byte in block

3 bits
7-way Fully Associative Cache

Tag store

Data store

Address

MUX

Tag

MUX

byte in block

Data store

MUX

byte in block

5 bits

3 bits

MUX

Logic

Hit?
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 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
- ...

249
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

```c
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)];
    }
}
```
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: $C = A \times B$
- Consider two input matrices $A$ and $B$ in row-major layout
  - $A$ size is $M \times P$
  - $B$ size is $P \times N$
  - $C$ size is $M \times N$
Naïve Matrix Multiplication (II)

- Naïve implementation of matrix multiplication has poor cache locality

```c
#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
Tiled Matrix Multiplication (II)

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

```c
#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](https://doi.org/10.1145/106972.106981)


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](https://doi.org/10.1016/B978-0-12-811986-0.00005-4)
Restructuring Data Layout (I)

- 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 they are rarely accessed!
Restructuring Data Layout (II)

- **Idea:** separate rarely-accessed 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 accessed?

```c
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;
}
```
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) 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
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/cost into replacement decisions?
- How do we design a hybrid cache replacement policy?


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


- **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)
Lectures on Cache Optimizations (III)

Fewest Misses ≠ Best Performance

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

Time: stall

Belady’s OPT replacement

Misses=4
Stalls=4

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

Time: stall

MLP-Aware replacement

Misses=6
Stalls=2


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=PL5Q2soXY2Zi9OhoVQBXYFIywZXCPl4M_&index=3](https://www.youtube.com/watch?v=OyomXCHNJDA&list=PL5Q2soXY2Zi9OhoVQBXYFIywZXCPl4M_&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](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](https://www.youtube.com/watch?v=jDHx2K9HxlM&list=PL5PHm2jkkXmi5CxxI7b3JCL1TWybTDtKq&index=21)

[https://www.youtube.com/onurmutlulectures](https://www.youtube.com/onurmutlulectures)
Multi-Core Issues in Caching
Caches in a Multi-Core System
Caches in a Multi-Core System

Source: https://www.anandtech.com/show/16252/mac-mini-apple-m1-tested
Caches in a Multi-Core System

Source: https://twitter.com/Locuza_/status/1454152714930331652
Caches in a Multi-Core System

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

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
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

Caches in a Multi-Core System

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

Nvidia Ampere, 2020

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

- 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

https://www.youtube.com/watch?v=7_Tqlw8gxOU&list=PL5Q2soXY2Zl9OhOvQBXyFIZywZXCPl4M&_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
Lectures on Multi-Core Cache Management
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](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](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_TqIw8gxOU&list=PL5Q2soXY2Zi9OhoVQBXYFIZywZXCPI4M&_index=17](https://www.youtube.com/watch?v=7_TqIw8gxOU&list=PL5Q2soXY2Zi9OhoVQBXYFIZywZXCPI4M&_index=17)

[https://www.youtube.com/onurmutlulectures](https://www.youtube.com/onurmutlulectures)
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

---

https://www.youtube.com/watch?v=0nnI807nCkc&list=PL5Q2soXY2Zi9xidylgBxUz7xRPS-wisBN&index=21
Lectures on Memory Resource Management

- **Computer Architecture, Fall 2020, Lecture 11a**
  - MemoryControllers(ETH, Fall 2020)
  - [https://www.youtube.com/watch?v=TeG773OgiMQ&list=PL5Q2soXY2Zi9xidyIgBxUz7xRPS-wisBN&index=20](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](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](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](https://www.youtube.com/watch?v=VJzZbwgBfy8&list=PL5Q2soXY2Zi9xidyIgBxUz7xRPS-wisBN&index=2)

[https://www.youtube.com/onurmutlulectures](https://www.youtube.com/onurmutlulectures)
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

P1

Interconnection Network

P2

Main Memory

ld r2, x

1000
The Cache Coherence Problem

P1

P2

ld r2, x

ld r2, x

1000

1000

Interconnection Network

Main Memory

x

1000
The Cache Coherence Problem

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

ld r2, x

Interconnection Network

Main Memory
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 Memory Ordering & Consistency

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

https://www.youtube.com/watch?v=Suy09mzTbiQ&list=PL5Q2soXY2Zi9xidy1gBxUz7xRPS-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=T9WlyzeeaII&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

https://www.youtube.com/onurmutlulectures