Brief Self Introduction

- **Rahul Bera**
  - Ph.D. Student in SAFARI Research Group
  - Previously worked with Intel Labs and AMD
  - [https://sites.google.com/view/rahulbera/home](https://sites.google.com/view/rahulbera/home)
  - [write2bera@gmail.com](mailto:write2bera@gmail.com) (best way to reach me)

- **Research interests:** Microarchitecture and memory system design
  - High-performance memory system design
    - Prefetchers, Cache Replacement Policies, ...
  - Microarchitectural techniques to reduce/tolerate memory latency
  - Data-driven microarchitectures
    - Machine-learning-assisted microarchitectural predictors and policies
Agenda for Today

- **Part 1: Multi-core issues in Caching**
  - Cache sharing/partitioning
    - Pros and Cons
  - Brief intro to cache coherence

- **Part 2: Prefetching**
  - What, When, Where, How
  - Different types of prefetchers
  - Brief intro to ML-inspired prefetchers
The Memory Hierarchy
Memory Hierarchy

- **Fundamental tradeoff**
  - Fast memory: small
  - Large memory: slow

- **Idea:** Memory hierarchy

- **Latency, cost, size, bandwidth**
Memory Hierarchy Example

```
```

<table>
<thead>
<tr>
<th>capacity</th>
<th>latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>10's of KB</td>
<td>≈ 1ns</td>
</tr>
<tr>
<td>100's of KB</td>
<td>&lt; 5ns</td>
</tr>
<tr>
<td>several MB</td>
<td>≈ 10ns</td>
</tr>
<tr>
<td>several GB</td>
<td>≈ 100ns</td>
</tr>
</tbody>
</table>

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

AMD Ryzen 5000, 2020

[Image of a multi-core system diagram]

Core Count: 8 cores/16 threads
L1 Caches: 32 KB per core
L2 Caches: 512 KB per core
L3 Cache: 32 MB shared

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
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

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/
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?)
Real World Example: Intel Skylake

https://www.makeuseof.com/tag/what-is-cpu-cache/
Real World Example: Apple M1

https://chipsandcheese.com/2022/05/21/igpu-cache-setups-compared-including-m1/
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=PL5Q2soXY2Zi9OhoVQBXYFIZywZXCPI4M&index=17
Lectures on Multi-Core Cache Management

Approaches to Reuse Prediction

1. Group Blocks
   - Use program counter or memory region information.
   - PC 1
   - PC 2

2. Learn group behavior
   - PC 1
   - PC 2

3. Predict reuse
   - PC 1
   - PC 2

- 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_Tqlw8qxOU&list=PL5Q2soXY2Zi9OhoVQBXYFIZywZXCPl4M_&index=17](https://www.youtube.com/watch?v=7_Tqlw8qxOU&list=PL5Q2soXY2Zi9OhoVQBXYFIZywZXCPl4M_&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
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=VJzZbwqBfy8&list=PL5Q2soXY2Zi9xidyIgBxUz7xRPS-wisBN&index=2

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

ld r2, x

P1

Interconnection Network

P2

Main Memory

1000
The Cache Coherence Problem

ld r2, x

P1

1000

P2

1000

ld r2, x

Interconnection Network

Main Memory

x 1000
The Cache Coherence Problem

Id r2, x
add r1, r2, r4
st x, r1

Id r2, x

P1

2000

Interconnection Network

P2

1000

Main Memory

x 1000
The Cache Coherence Problem

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

ld r2, x
Should NOT load 1000
ld r5, x

P1

2000

Interconnection Network

P2

1000

Main Memory
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

- 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 and P2 saw an inconsistent order of operations in memory

A → B → X
A → X

BOTH CANNOT BE CORRECT! (from memory's perspective)
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/onurmutlulelectures
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)
Directory Based Coherence Example (I)

Example directory based scheme

\[ P = 4 \]

\[ \begin{array}{c}
0 \\
0 \\
0 \\
0 \\
0
\end{array} \]

Exclusive bit

No cache has the block

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

\[ \begin{array}{c}
0 \\
0 \\
0 \\
0 \\
0
\end{array} \rightarrow \begin{array}{c}
0 \\
1 \\
0 \\
0 \\
0
\end{array} \]

2. \( P_3 \) takes a read miss

\[ \begin{array}{c}
0 \\
1 \\
0 \\
1 \\
1
\end{array} \]
P2 takes a write miss

→ invalidate P1 & P3’s caches
→ write request → P2 has the exclusive copy of the block now. Set the Exclusive bit

→ P2 can now update the block without notifying any other processor or the directory

→ P2 needs to have a bit in its cache indicating it can perform exclusive updates to that block
→ private/exclusive bit per cache block

P3 takes a write miss

→ Mem Controller requests block from P2
→ Mem Controller gives block to P3
→ P2 invalidates its copy

P2 takes a read miss

→ 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
// initialize large arrays A, B
for (j=0; j<N; j++) {
    index = j*linesize;  // streaming
    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?

Memory Request Buffer

Row size: 8KB, cache block size: 64B
128 (8KB/64B) requests of T0 serviced before T1

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


Memory Performance Attacks: Denial of Memory Service in Multi-Core Systems

Thomas Moscibroda    Onur Mutlu
Microsoft Research
{moscitho,onur}@microsoft.com

http://www.youtube.com/watch?v=VJzZbwgBfy8
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  Stephen W. Keckler  Onur Mutlu†
Department of Computer Sciences  The University of Texas at Austin  {bgrot, skeckler@cs.utexas.edu}
†Computer Architecture Laboratory (CALCM)  Carnegie Mellon University  onur@cmu.edu
Energy Comparison of Memory Technologies
The Problem: Energy

- Faster is more energy-efficient
  - SRAM, \(\sim 5\) pJ
  - DRAM, \(\sim 40-140\) pJ
  - PCM-DIMM (Intel Optane DC DIMM), \(\sim 80-540\) pJ
  - PCM-SSD, \(\sim 120\) µJ
  - Flash memory, \(\sim 250\) µJ
  - Hard Disk, \(\sim 60\) 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>~5 pJ</td>
<td>~1.25 pJ</td>
<td></td>
</tr>
<tr>
<td>SRAM</td>
<td>KByte~MByte</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>~40-140 pJ</td>
<td>~10-35 pJ</td>
<td></td>
</tr>
<tr>
<td>PCM-SSD (Intel Optane SSD)</td>
<td>Gigabyte</td>
<td>~6-10 µs</td>
<td>~120 µJ</td>
<td>~30 nJ</td>
<td></td>
</tr>
<tr>
<td>Flash memory</td>
<td>Gigabyte</td>
<td>~50-100 µs</td>
<td>~250 µJ</td>
<td>~61 nJ</td>
<td></td>
</tr>
<tr>
<td>Hard Disk</td>
<td>Terabyte</td>
<td>~10 millisec</td>
<td>~60 mJ</td>
<td>~15 µJ</td>
<td></td>
</tr>
</tbody>
</table>

- **Bigger is slower**
- **Faster is more energy-efficient**

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

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

Address

11...1111100  
11...1111000  
11...1110100  
11...1110000  
11...1110000  

00...00100100  
00...00100000  
00...00011100  
00...00011000  
00...00010100  
00...00010000  
00...00001100  
00...00001000  
00...00000100  
00...00000000  

mem[0xFF...FC]  
mem[0xFF...F8]  
mem[0xFF...F4]  
mem[0xFF...F0]  
mem[0xFF...EC]  
mem[0xFF...E8]  
mem[0xFF...E4]  
mem[0xFF...E0]  
mem[0x00...24]  
mem[0x00...20]  
mem[0x00...1C]  
mem[0x00...18]  
mem[0x00...14]  
mem[0x00...10]  
mem[0x00...0C]  
mem[0x00...08]  
mem[0x00...04]  
mem[0x00...00]  

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)
Direct Mapped Cache Hardware

Memory Address

1. Tag
2. Set
3. Byte Offset

Hit

V Tag

Data

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

Data
Direct Mapped Cache Performance

# MIPS assembly code

```mips
    # 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 = 73
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>00..00</td>
<td>mem[0x00...0C]</td>
</tr>
<tr>
<td>0</td>
<td>00..00</td>
<td>mem[0x00...08]</td>
</tr>
<tr>
<td>0</td>
<td>00..00</td>
<td>mem[0x00...04]</td>
</tr>
</tbody>
</table>

Miss Rate = 3/15 = 20%

Temporal Locality
Compulsory Misses
Direct Mapped Cache: Conflict

# MIPS assembly code

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

$Miss Rate$ = 75%
Direct Mapped Cache: Conflict

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

<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>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>

<table>
<thead>
<tr>
<th>Set</th>
<th>Offset</th>
<th>Memory Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>0000</td>
<td>mem[0x00...04]</td>
</tr>
<tr>
<td>6</td>
<td>0001</td>
<td>mem[0x00...24]</td>
</tr>
<tr>
<td>5</td>
<td>0010</td>
<td>set of 0</td>
</tr>
<tr>
<td>4</td>
<td>0011</td>
<td>set of 0</td>
</tr>
<tr>
<td>3</td>
<td>0100</td>
<td>set of 0</td>
</tr>
<tr>
<td>2</td>
<td>0101</td>
<td>set of 0</td>
</tr>
<tr>
<td>1</td>
<td>0110</td>
<td>set of 0</td>
</tr>
<tr>
<td>0</td>
<td>0111</td>
<td>set of 0</td>
</tr>
</tbody>
</table>

Miss Rate = 10/10 = 100%

Conflict Misses
N-Way Set Associative Cache

Memory Address

Tag Set Offset

Way 1

Way 0

V Tag Data V Tag Data

Way 1

Way 0

Hit

Hit

Hit

Hit

Hit

Hit

Hit

Hit

Hit

Hit
N-way Set Associative Performance

# MIPS assembly code

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

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

Miss Rate = 78
N-way Set Associative Performance

# MIPS assembly code

```
addi $t0, $0, 5

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

Miss Rate = 2/10
= 20%

Associativity reduces conflict misses

<table>
<thead>
<tr>
<th>Way 1</th>
<th>Way 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
<td>Tag</td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00...10</td>
</tr>
<tr>
<td>0</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 = 82
Direct Mapped Cache Performance

```
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: \( B = \frac{C}{b} \)
  - Number of blocks in a set: \( N \)
  - Number of Sets: \( S = \frac{B}{N} \)

<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 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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Set Number
3 (11)  
2 (10)  
1 (01)  
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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Set Number
3 (11)  
2 (10)  
1 (01)  
0 (00)
LRU Replacement

```mips
# MIPS assembly
lw $t0, 0x04($0)
lw $t1, 0x24($0)
lw $t2, 0x54($0)
```

<table>
<thead>
<tr>
<th>Set 0 (00)</th>
<th>Set 1 (01)</th>
<th>Set 2 (10)</th>
<th>Set 3 (11)</th>
</tr>
</thead>
<tbody>
<tr>
<td>V U Tag</td>
<td>V Tag</td>
<td>Data</td>
<td>Data</td>
</tr>
<tr>
<td>0 0</td>
<td>0</td>
<td>0</td>
<td><em>mem[0x00...04]</em></td>
</tr>
<tr>
<td>0 0</td>
<td>0</td>
<td>0</td>
<td>mem[0x00...04]</td>
</tr>
<tr>
<td>0 0</td>
<td>1 0 00...010</td>
<td>mem[0x00...24]</td>
<td>1 00...000</td>
</tr>
<tr>
<td>0 0</td>
<td>0</td>
<td>0</td>
<td>mem[0x00...54]</td>
</tr>
</tbody>
</table>

(a) Way 0: Set 0 (00) - Set 3 (11)

(b) Way 1: Set 0 (00) - Set 3 (11)
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
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
Subblocked (Sectored) Caches

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

++ No need to transfer the entire cache block into the cache
   (A write simply validates and updates a subblock)
++ More freedom in transferring subblocks into the cache (a cache block does not need to be in the cache fully)
   (How many subblocks do you transfer on a read?)

-- More complex design
-- May not exploit spatial locality fully

| v | d | subblock | v | d | subblock | • | • | • | • | v | d | subblock | tag |
Instruction vs. Data Caches

- **Separate or Unified?**

- **Pros and Cons of Unified:**
  + Dynamic sharing of cache space: no overprovisioning that might happen with static partitioning (i.e., separate I and D caches)
  -- Instructions and data can evict/thrash each other (i.e., no guaranteed space for either)
  -- I and D are accessed in different places in the pipeline. Where do we place the unified cache for fast access?

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

- **Outer level caches are almost always unified**

---

*102*
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


Die shot interpretation by Locuza, October 2021

Intel Alder Lake, 2021

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 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

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

<table>
<thead>
<tr>
<th>Cores:</th>
<th>L1 Cache or Scratchpad:</th>
<th>L2 Cache:</th>
</tr>
</thead>
<tbody>
<tr>
<td>144 Streaming Multiprocessors</td>
<td>256KB per SM</td>
<td>60 MB shared</td>
</tr>
<tr>
<td></td>
<td>Can be used as L1 Cache and/or Scratchpad</td>
<td></td>
</tr>
</tbody>
</table>

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

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.

Memory in the NVIDIA H100 GPU

- SM
- Control
- Registers
- Core
- Core
- Core
- Core
- Core
- Core
- Core
- Core
- SM-to-SM
- Direct copy

- 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 $\rightarrow$ exploit temporal locality not well
  - waste cache space and bandwidth/energy
    - if spatial locality is not high

![Graph showing hit rate vs. block size](graph.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**

```
Data store

MUX

byte in block

MUX

Hit?

Logic

=?

=?

=?

=?

Tag store

Address

tag
index
byte in block

4 bits 1 b 3 bits
```
Higher Associativity (3-way)

- 3-way

Tag store

=?

=?

=?

Logic → Hit?

Data store

MUX

MUX

byte in block

Address

tag  index  byte in block

4 bits 1 b 3 bits
Recall: 8-way Fully Associative Cache

Tag store

Data store

Address

tag

byte in block

MUX

MUX

byte in block

5 bits

3 bits
7-way Fully Associative Cache

![Diagram of a 7-way Fully Associative Cache]

- Tag store
  - 7-way associative lookup
  - Logic
  - Hit?

- Data store
  - MUX
  - Address
    - tag (5 bits)
    - byte in block (3 bits)
  - MUX

- byte in block
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
- ...

130
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, ...
Restructuring Data Access Patterns (II)

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

- Also called **Tiling**
Data Reuse: An Example from GPU Computing

- Same memory locations accessed by neighboring threads

Gaussian filter applied on every pixel of an image

```
for (int i = 0; i < 3; i++){
    for (int j = 0; j < 3; j++){
        sum += gauss[i][j] * Image[(i+row-1)*width + (j+col-1)];
    }
}
```
To take advantage of data reuse, we divide the input into tiles that can be loaded into shared memory (scratchpad memory).

```c
__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`

---


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)
Tiled Matrix Multiplication on GPUs
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

Hit/Miss: H H H H M  H H H H H  M M M M M
Time: stall
Misses=4 Stalls=4

Hit/Miss: H M M M  H M M M  H H H H
Time: stall Saved cycles
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

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

https://www.youtube.com/watch?v=55oYBm9cifI&list=PL5Q2soXY2Zi9JXe3ywQMhylk_d5dl-TM7&index=6
Lectures on Cache Optimizations (III)

Fewest Misses ≠ Best Performance

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

Hit/Miss: H H H H M

Time: stall

Belady’s OPT replacement

Misses = 4
Stalls = 4

Hit/Miss: H M M M M

Time: stall

MLP-Aware replacement

Misses = 6
Stalls = 2


9,737 views • Mar 5, 2015

https://www.youtube.com/watch?v=jDHx2K9HxIM&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=PL5Q2soXY2Zi9OhoVQBXYFIzywZXCPl4M_&index=3](https://www.youtube.com/watch?v=OyomXCHNJDA&list=PL5Q2soXY2Zi9OhoVQBXYFIzywZXCPl4M_&index=3)

- Computer Architecture, Fall 2018, Lecture 4a
  - Cache Design (ETH, Fall 2018)
  - [https://www.youtube.com/watch?v=55oYBm9cifI&list=PL5Q2soXY2Zi9JXe3ywQMhlYlk_d5dI-TM7&index=6](https://www.youtube.com/watch?v=55oYBm9cifI&list=PL5Q2soXY2Zi9JXe3ywQMhlYlk_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

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

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
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
   ABCC
   PC 2
   ABCC

2. Learn group behavior
   PC 1
   ABCC
   PC 2
   ABCC

3. Predict reuse
   PC 1
   C→C
   PC 2
   C→C

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

https://www.youtube.com/watch?v=Siz86__PD4w&list=PL5Q2soXY2Zi9JXe3ywQMhylk_d5dl-TM7&index=30
Lectures on Multi-Core Cache Management

- Computer Architecture, Fall 2018, Lecture 18b
  - Multi-Core Cache Management (ETH, Fall 2018)
  - https://www.youtube.com/watch?v=c9FhGRB3HoA&list=PL5Q2soXY2Zi9JXe3ywQMhylk_d5dI-TM7&index=29

- Computer Architecture, Fall 2018, Lecture 19a
  - Multi-Core Cache Management II (ETH, Fall 2018)
  - https://www.youtube.com/watch?v=Siz86__PD4w&list=PL5Q2soXY2Zi9JXe3ywQMhylk_d5dI-TM7&index=30

- Computer Architecture, Fall 2017, Lecture 15
  - Multi-Core Cache Management (ETH, Fall 2017)
  - https://www.youtube.com/watch?v=7_Tqlw8qxOU&list=PL5Q2soXY2Zi9OhoVQBXYFIZywZXCPI4M_&index=17

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

[YouTube Lecture](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

P1

Interconnection Network

P2

ld r2, x

1000

Main Memory

1000
The Cache Coherence Problem

ld r2, x

Interconnection Network

Main Memory
The Cache Coherence Problem

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

ld r2, x

Interconnection Network

Main Memory

P1

2000

P2

1000
The Cache Coherence Problem

P1

Id r2, x
add r1, r2, r4
st x, r1

2000

Interconnection Network

P2

ld r2, x

1000

Should NOT load 1000
ld r5, x

ld r2, x

Main Memory

x 1000
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

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

A → B (set F1=1)
B (req F1=0) sent to mem.
B (req F1=0) stored
F1=1 in memory
Too late!
Mem. completes F1=1

A → X
Y (req F1=0) sent to mem.
Y (req F1=0) stored
F1=1 in memory
Too late!
Mem. completes F1=1

Mem. sends F2 (0) to P1
Mem. sends F3 (0) to P2

P1 is in Critical Section!

https://www.youtube.com/watch?v=Suy09mzTbiQ&list=PL5Q2soXY2Zi9xidylgBxUz7xRPS-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