CS152: Computer Systems Architecture
Memory System and Caches

Sang-Woo Jun
Winter 2021

Large amount of material adapted from MIT 6.004, “Computation Structures”, Morgan Kaufmann “Computer Organization and Design: The Hardware/Software Interface: RISC-V Edition”, and CS 152 Slides by Isaac Scherson
Eight great ideas

- Design for Moore’s Law
- **Use abstraction to simplify design**
- Make the common case fast
- Performance via parallelism
- Performance via pipelining
- **Performance via prediction**
- Hierarchy of memories
- Dependability via redundancy
Caches are important

“There are only two hard things in computer science:
1. Cache invalidation,
2. Naming things,
3. and off-by-one errors”

Original quote (with only the first two points) by Phil Karlton
I couldn’t find joke source
A modern computer has a hierarchy of memory

- **CPU**
  - Instruction cache
  - Data cache
  - Shared cache

- **DRAM**
  - Low latency (~1 cycle)
  - Small (KBs)
  - Expensive ($1000s per GB)

- **Cost prohibits having a lot of fast memory**
  - Ideal memory:
    - As cheap and large as DRAM (Or disk!)
    - As fast as SRAM
    - ...Working on it!

- **High latency (100s~1000s of cycles)**
  - Large (GBs)
  - Cheap (<$5 per GB)
What causes the cost/performance difference? – SRAM

- SRAM (Static RAM) vs. DRAM (Dynamic RAM)
- SRAM is constructed entirely out of transistors
  - Accessed in clock-synchronous way, just like any other digital component
  - Subject to propagation delay, etc, which makes large SRAM blocks expensive and/or slow

Source: Inductiveload, from commons.wikimedia.org
What causes the cost/performance difference? – DRAM

- DRAM stores data using a capacitor
  - Very small/dense cell
  - A capacitor holds charge for a short while, but slowly leaks electrons, losing data
  - To prevent data loss, a controller must periodically read all data and write it back (“Refresh”)
    - Hence, “Dynamic” RAM
  - Requires fab process separate from processor

- Reading data from a capacitor is high-latency
  - EE topics involving sense amplifiers, which we won’t get into

Source: Dailytech

Note: Old, “trench capacitor” design
What causes the cost/performance difference? – DRAM

- DRAM is typically organized into a rectangle (rows, columns)
  - Reduces addressing logic, which is a high overhead in such dense memory
  - Whole row must be read whenever data in new row is accessed
  - As of 2020, typical row size ~8 KB
- Fast when accessing data in same row, order of magnitude slower when accessing small data across rows
  - Accessed row temporarily stored in DRAM “row buffer”
And the gap keeps growing
Goals of a memory system

- Performance at reasonable cost
  - Capacity of DRAM, but performance of SRAM

- Simple abstraction
  - CPU should be oblivious to type of memory
  - Should not make software/compiler responsible for identifying memory characteristics and optimizing for them, as it makes performance not portable
    - Unfortunately this is not always possible, but the hardware does its best!
Introducing caches

❑ The CPU is (largely) unaware of the underlying memory hierarchy
  o The memory abstraction is a single address space
  o The memory hierarchy automatically stores data in fast or slow memory, depending on usage patterns

❑ Multiple levels of “caches” act as interim memory between CPU and main memory (typically DRAM)
  o Processor accesses main memory through the cache hierarchy
  o If requested address is already in the cache (address is “cached”, resulting in “cache hit”), data operations can be fast
  o If not, a “cache miss” occurs, and must be handled to return correct data to CPU
And the gap keeps growing

<table>
<thead>
<tr>
<th>Year</th>
<th>Processor</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1980</td>
<td>80386</td>
<td></td>
</tr>
<tr>
<td>1985</td>
<td>80486</td>
<td></td>
</tr>
<tr>
<td>1990</td>
<td>Intel x86 (and SRAM)</td>
<td>100,000</td>
</tr>
</tbody>
</table>

Caches introduced to Intel x86 (80386, 80486)
Cache operation

- One of the most intensely researched fields in computer architecture
- Goal is to somehow make to-be-accessed data available in fastest possible cache level at access time
  - Method 1: Caching recently used addresses
    - Works because software typically has “Temporal Locality”: If a location has been accessed recently, it is likely to be accessed (reused) soon
  - Method 2: Pre-fetching based on future pattern prediction
    - Works because software typically has “Spatial Locality”: If a location has been accessed recently, it is likely that nearby locations will be accessed soon
  - Many, many more clever tricks and methods are deployed!

Average Memory Access Time = HitTime + MissRatio × MissPenalty
Basic cache operations

- Unit of caching: “Block” or “Cache line”
  - May be multiple words -- 64 Bytes in modern Intel x86
- If accessed data is present in upper level
  - Hit: access satisfied by upper level
- If accessed data is absent
  - Miss: block copied from lower level
    - Time taken: miss penalty
  - Then accessed data supplied from upper level

How does the memory system keep track of what is present in cache?
A simple solution: “Direct Mapped Cache”

- Cache location determined by address
- Each block in main memory mapped on one location in cache memory (“Direct Mapped”)
  - “Direct mapped”
- Cache is smaller than main memory, so many DRAM locations map to one cache location

\[
\text{Cache address}_{\text{block}} = \text{(main memory address}_{\text{block}} \mod (\text{cache size}_{\text{block}})}
\]

Since cache size is typically power of two, Cache address is lower bits of block address

\[
\text{(Cache address}_{\text{block}}) = \text{(main memory address}_{\text{block}} \mod (\text{cache size}_{\text{block}})}
\]
Selecting index bits

Why do we chose low order bits for index?

- Allows consecutive memory locations to live in the cache simultaneously
- Reduces likelihood of replacing data that may be accessed again in the near future
- Helps take advantage of locality
Tags and Valid Bits

- How do we know which particular block is stored in a cache location?
  - Store block address as well as the data, compare when read
  - Actually, only need the high-order bits (Called the “tag”)

- What if there is no data in a location?
  - Valid bit: 1 = present, 0 = not present
  - Initially 0
Direct Mapped Cache Access

- For cache with $2^W$ cache lines
  - Index into cache with $W$ address bits (the index bits)
  - Read out valid bit, tag, and data
  - If valid bit == 1 and tag matches upper address bits, cache hit!

Example 8-line direct-mapped cache:
Direct Mapped Cache Access Example

- 64-line direct-mapped cache -> 64 indices -> 6 index bits

Example 1: Read memory 0x400C

0x400C = 0100 0000 0000 1100
Tag: 0x40  Index: 0x3
Byte offset: 0x0
-> Cache hit! Data read 0x42424242

Example 2: Read memory 0x4008

0x4008 = 0100 0000 0000 1000
Tag: 0x40  Index: 0x2
Byte offset: 0x0
-> Cache miss! Tag mismatch

<table>
<thead>
<tr>
<th>Valid bit</th>
<th>Tag (24 bits)</th>
<th>Data (32 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x000058</td>
<td>0xDEADBEEF</td>
</tr>
<tr>
<td>1</td>
<td>0x000058</td>
<td>0x00000000</td>
</tr>
<tr>
<td>2</td>
<td>0x000058</td>
<td>0x00000007</td>
</tr>
<tr>
<td>3</td>
<td>0x000040</td>
<td>0x42424242</td>
</tr>
<tr>
<td>4</td>
<td>0x000007</td>
<td>0x6FBA2381</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>63</td>
<td>0x000058</td>
<td>0xF7324A32</td>
</tr>
</tbody>
</table>
Direct Mapped Cache Access Example

- 8-blocks, 1 word/block, direct mapped
- Initial state: All “valid” bits are set to invalid

<table>
<thead>
<tr>
<th>Index</th>
<th>V</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>N</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Direct Mapped Cache Access Example

Cache miss! Main memory read to cache

<table>
<thead>
<tr>
<th>Word addr</th>
<th>Binary addr</th>
<th>Hit/miss</th>
<th>Cache block</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>10 110</td>
<td>Miss</td>
<td>110</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Index</th>
<th>V</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>110</strong></td>
<td><strong>Y</strong></td>
<td><strong>10</strong></td>
<td><strong>Mem[10110]</strong></td>
</tr>
<tr>
<td>111</td>
<td>N</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Direct Mapped Cache Access Example

<table>
<thead>
<tr>
<th>Word addr</th>
<th>Binary addr</th>
<th>Hit/miss</th>
<th>Cache block</th>
</tr>
</thead>
<tbody>
<tr>
<td>26</td>
<td>11 010</td>
<td>Miss</td>
<td>010</td>
</tr>
</tbody>
</table>

Cache miss! Main memory read to cache

<table>
<thead>
<tr>
<th>Index</th>
<th>V</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>Y</td>
<td>11</td>
<td>Mem[11010]</td>
</tr>
<tr>
<td>011</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>Y</td>
<td>10</td>
<td>Mem[10110]</td>
</tr>
<tr>
<td>111</td>
<td>N</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Direct Mapped Cache Access Example

<table>
<thead>
<tr>
<th>Word addr</th>
<th>Binary addr</th>
<th>Hit/miss</th>
<th>Cache block</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>10 110</td>
<td>Hit</td>
<td>110</td>
</tr>
<tr>
<td>26</td>
<td>11 010</td>
<td>Hit</td>
<td>010</td>
</tr>
</tbody>
</table>

Cache hit! No main memory read

<table>
<thead>
<tr>
<th>Index</th>
<th>V</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>Y</td>
<td>11</td>
<td>Mem[11010]</td>
</tr>
<tr>
<td>011</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>Y</td>
<td>10</td>
<td>Mem[10110]</td>
</tr>
<tr>
<td>111</td>
<td>N</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Direct Mapped Cache Access Example

Cache misses result in main memory read

<table>
<thead>
<tr>
<th>Word addr</th>
<th>Binary addr</th>
<th>Hit/miss</th>
<th>Cache block</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>10 000</td>
<td>Miss</td>
<td>000</td>
</tr>
<tr>
<td>3</td>
<td>00 011</td>
<td>Miss</td>
<td>011</td>
</tr>
<tr>
<td>16</td>
<td>10 000</td>
<td>Hit</td>
<td>000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Index</th>
<th>V</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>Y</td>
<td>10</td>
<td>Mem[10000]</td>
</tr>
<tr>
<td>001</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>Y</td>
<td>11</td>
<td>Mem[11010]</td>
</tr>
<tr>
<td>011</td>
<td>Y</td>
<td>00</td>
<td>Mem[00011]</td>
</tr>
<tr>
<td>100</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>Y</td>
<td>10</td>
<td>Mem[10110]</td>
</tr>
<tr>
<td>111</td>
<td>N</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Direct Mapped Cache Access Example

Cache collision results in eviction of old value

What if old value was written to? Written data must be saved to main memory!

<table>
<thead>
<tr>
<th>Word addr</th>
<th>Binary addr</th>
<th>Hit/miss</th>
<th>Cache block</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>10 010</td>
<td>Miss</td>
<td>010</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Index</th>
<th>V</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>Y</td>
<td>10</td>
<td>Mem[10000]</td>
</tr>
<tr>
<td>001</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>Y</td>
<td>10</td>
<td>Mem[10010]</td>
</tr>
<tr>
<td>011</td>
<td>Y</td>
<td>00</td>
<td>Mem[00011]</td>
</tr>
<tr>
<td>100</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>Y</td>
<td>10</td>
<td>Mem[10110]</td>
</tr>
<tr>
<td>111</td>
<td>N</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Write Policies

- **Write Through**: Write is applied to cache, and applied immediately to memory
  - + Simple to implement!
  - - Wastes main memory bandwidth

- **Write Back**: Write is only applied to cache, write is applied only when evicted
  - Cache line has another metadata bit “Dirty” to remember if it has been written
  - + Efficient main memory bandwidth
  - - Complex
  - More common in modern systems
Write Back Example: Cache Hit/Miss

- 64-line direct-mapped cache -> 64 indices -> 6 index bits

- Write 0x9 to 0x480C
  - Tag: 0x48
  - Index: 0x3
  - Byte offset: 0x0
  -> Cache hit!

- Write 0x1 to 0x490C
  - Tag: 0x49
  - Index: 0x3
  - Byte offset: 0x0
  -> Cache miss! (Tag mismatch)

Cache line 3 must be written to main memory, and then apply write to cache
Larger block (cache line) sizes

- Take advantage of spatial locality: Store multiple words per data line
  - Always fetch entire block (multiple words) from memory
  - Another advantage: Reduces size of tag memory!
  - Disadvantage: Fewer indices in the cache -> Higher miss rate!

Example: 4-block, 16-word direct-mapped cache

- 32-bit BYTE address
- Tag bits: 26 (=32-6)
- Index bits: 2 (4 indices)
- Block offset bits: 2 (4 words/block)
- Byte offset bits: 2
Cache miss with larger block

- 64 elements with block size == 4 words
  - 16 cache lines, 4 index bits

- Write 0x9 to 0x483C
  - 0100 1000 0011 1100
    - Tag: 0x48
    - Index: 0x3
    - -> Cache hit!
    - Block offset: 0x3

- Write 0x1 to 0x4938
  - 0100 1001 0011 1000
    - Tag: 0x49
    - Index: 0x3
    - -> Cache miss!
    - Block offset: 0x2
Cache miss with larger block

- Write 0x1 to 0x4938
  - 0100 1001 0011 1000
    - Tag: 0x49  Index: 0x3
    - Block offset: 0x2

- Since D == 1,
  - Write cache line 3 to memory (All four words)
  - Load cache line from memory (All four words)
  - Apply write to cache

Writes/Reads four data elements just to write one!
Block size trade-offs

- Larger block sizes...
  - Take advantage of spatial locality (also, DRAM is faster with larger blocks)
  - Incur larger miss penalty since it takes longer to transfer the block from memory
  - Can increase the average hit time and miss ratio

- $\text{AMAT} = \text{HitTime} + \text{MissPenalty} \times \text{MissRatio}$
Looking back…

- Caches for high performance at low cost
  - Exploits temporal locality in many programs
  - Caches recently used data in fast, expensive memory

- Looked at “direct mapped” caches
  - Cache slot to use was singularly determined by the address in main memory
  - Uses tags and valid bits to correctly match data in cache and main memory

- Cache blocks (or “cache lines”) typically larger than a word
  - Reduces tag size, better match with backing DRAM granularity
  - Exploits spatial locality, up to a certain size (~64 bytes according to benchmarks)

Given a fixed space budget on the chip for cache memory, is this the most efficient way to manage it?
Direct-Mapped Cache Problem: Conflict Misses

- Assuming a 1024-line direct-mapped cache, 1-word cache line
- Consider steady state, after already executing the code once
  - What can be cached has been cached

- Conflict misses:
  - Multiple accesses map to same index!

We have enough cache capacity, just inconvenient access patterns
Other extreme: “Fully associative” cache

- Any address can be in any location
  - No cache index!
  - Flexible (no conflict misses)
  - Expensive: Must compare tags of all entries in parallel to find matching one

- Best use of cache space (all slots will be useful)
- But management circuit overhead is too large
Three types of misses

- Compulsory misses (aka cold start misses)
  - First access to a block

- Capacity misses
  - Due to finite cache size
  - A replaced block is later accessed again

- Conflict misses (aka collision misses)
  - Conflicts that happen even when we have space left
  - Due to competition for entries in a set
  - Would not occur in a fully associative cache of the same total size

Empty space can always be used in a fully associative cache
(e.g., 8 KiB data, 32 KiB cache, but still misses? Those are conflict misses)
Balanced solution: N-way set-associative cache

- Use multiple direct-mapped caches in parallel to reduce conflict misses

- Nomenclature:
  - # Rows = # Sets
  - # Columns = # Ways
  - Set size = #ways = “set associativity” (e.g., 4-way -> 4 lines/set)

- Each address maps to only one set, but can be in any way within the set

- Tags from all ways are checked in parallel
Set-associative cache organization
Spectrum of associativity (For eight total blocks)

- **One-way set-associative (Direct-Mapped)**
  - V D Tag Data

- **Two-way set-associative**
  - V D Tag Data V D Tag Data

- **Four-way set-associative**
  - V D Tag Data V D Tag Data V D Tag Data V D Tag Data

- **Eight-way set-associative (Fully associative)**
  - V D Tag Data V D Tag Data V D Tag Data V D Tag Data V D Tag Data V D Tag Data

Each “Data” is a cache line (~64 bytes), needs another mux layer to get actual word
Associativity example

- Compare caches with four elements
  - Block access sequence: 0, 8, 0, 6, 8

- Direct mapped (Cache index = address mod 4)

<table>
<thead>
<tr>
<th>Block address</th>
<th>Cache index</th>
<th>Hit/miss</th>
<th>Cache content after access</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>miss</td>
<td>Mem[0]</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>miss</td>
<td>Mem[8]</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>miss</td>
<td>Mem[0]</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>miss</td>
<td>Mem[0], Mem[6]</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>miss</td>
<td>Mem[8], Mem[6]</td>
</tr>
</tbody>
</table>

Time
Associativity example

- 2-way set associative (Cache index = address mod 2)

<table>
<thead>
<tr>
<th>Block address</th>
<th>Cache index</th>
<th>Hit/miss</th>
<th>Cache content after access</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Set 0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>miss</td>
<td>Mem[0]</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>miss</td>
<td>Mem[0] Mem[8]</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>hit</td>
<td>Mem[0] Mem[8]</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>miss</td>
<td>Mem[0] Mem[6]</td>
</tr>
</tbody>
</table>

- Fully associative (No more cache index!)

<table>
<thead>
<tr>
<th>Block address</th>
<th>Hit/miss</th>
<th>Cache content after access</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>miss</td>
<td>Mem[0]</td>
</tr>
<tr>
<td>8</td>
<td>miss</td>
<td>Mem[0] Mem[8]</td>
</tr>
<tr>
<td>0</td>
<td>hit</td>
<td>Mem[0] Mem[8]</td>
</tr>
</tbody>
</table>
How Much Associativity?

- Increased associativity decreases miss rate
  - But with diminishing returns

- Simulation of a system with 64KB D-cache, 16-word blocks, SPEC2000
  - 1-way: 10.3%
  - 2-way: 8.6%
  - 4-way: 8.3%
  - 8-way: 8.1%
How much associativity, how much size?

- Highly application-dependent!

For integer portion of SPEC CPU2000

Capacity misses
Conflict misses
Compulsory misses

Associativity implies choices

Direct-mapped

Only one place an address can go
In case of conflict miss, old data is simply evicted

N-way set-associative

Multiple places an address can go
In case of conflict miss, which way should we evict?
What is our “replacement policy”?
Replacement policies

- Optimal policy (Oracle policy):
  - Evict the line accessed furthest in the future
  - Impossible: Requires knowledge of the future!

- Idea: Predict the future from looking at the past
  - If a line has not been used recently, it’s often less likely to be accessed in the near future (temporal locality argument)

- Least Recently Used (LRU): Replace the line that was accessed furthest in the past
  - Works well in practice
  - Needs to keep track of ordering, and discover oldest line quickly

Pure LRU requires complex logic: Typically implements cheap approximations of LRU
Other replacement policies

- LRU becomes very bad if working set becomes larger than cache size
  - “for (i = 0 to 1025) A[i];”, if cache is 1024 elements large, every access is miss

- Some alternatives exist
  - Effective in limited situations, but typically not as good as LRU on average
  - Most recently used (MRU), First-In-First-Out (FIFO), random, etc ...
  - Sometimes used together with LRU
Performance improvements with caches

- Given CPU of CPI = 1, clock rate = 4GHz
  - Main memory access time = 100ns
  - Miss penalty = 100ns/0.25ns = 400 cycles
  - CPI without cache = 400

- Given first-level cache with no latency, miss rate of 2%
  - Effective CPI = 1 + 0.02 \times 400 = 9

- Adding another cache (L2) with 5ns access time, miss rate of 0.5%
  - Miss penalty = 5ns/0.25ns = 20 cycles
  - New CPI = 1 + 0.02 \times 20 + 0.005 \times 400 = 3.4

<table>
<thead>
<tr>
<th></th>
<th>Base</th>
<th>L1</th>
<th>L2</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPI Improvements</td>
<td>400</td>
<td>9</td>
<td>3.4</td>
</tr>
<tr>
<td>IPC improvements</td>
<td>0.0025</td>
<td>0.11</td>
<td>0.29</td>
</tr>
<tr>
<td>Normalized performance</td>
<td>1</td>
<td>44</td>
<td>118</td>
</tr>
</tbody>
</table>
Real-world: Intel Haswell i7

- Four layers of caches (two per-core layers, two shared layers)
  - Larger caches have higher latency
  - Want to achieve both speed and hit rate!

- The layers
  - L1 Instruction & L1 Data: 32 KiB, 8-way set associative
  - L2: 256 KiB, 8-way set associative
  - L3: 6 MiB, 12-way set associative
  - L4: 128 MiB, 16-way set associative eDRAM!
Real-world: Intel Haswell i7

- Cache access latencies
  - L1: 4 - 5 cycles
  - L2: 12 cycles
  - L3: ~30 - ~50 cycles

- For reference, Haswell as 14 pipeline stages
So far...

- What are caches and why we need them

- Direct-mapped cache
  - Write policies
  - Larger block size and implications
  - Conflict and other misses

- Set-associative cache
  - Replacement policies
Cache-aware software example: Matrix-matrix multiply

- Multiplying two \( N \times N \) matrices \((C = A \times B)\)

\[
\begin{align*}
\text{for } (i = 0 \text{ to } N) \\
\text{for } (j = 0 \text{ to } N) \\
\text{for } (k = 0 \text{ to } N) \\
C[i][j] &\leftarrow A[i][k] \times B[k][j]
\end{align*}
\]

2048x2048 on a i5-7400 @ 3 GHz = 63.19 seconds

is this fast?

Whole calculation requires 2K * 2K * 2K = 8 Billion floating-point mult + add

At 3 GHz, ~5 seconds just for the math. Over 1000% overhead!

Assuming IPC=1, true numbers complicated due to superscalar
Overheads in matrix multiplication (1)

- Column-major access makes inefficient use of cache lines
  - A 64 Byte block is read for each element loaded from B
  - 64 bytes read from memory for each 4 useful bytes

- Shouldn’t caching fix this? Unused bits should be useful soon!
  - 64 bytes $\times$ 2048 = 128 KB ... Already overflows L1 cache (~32 KB)

```
for (i = 0 to N)
  for (j = 0 to N)
    for (k = 0 to N)
      C[i][j] += A[i][k] * B[k][j]
```
Overheads in matrix multiplication (1)

- One solution: Transpose B to match cache line orientation
  - Does transpose add overhead? Not very much as it only scans B once

- Drastic improvements!
  - Before: 63.19s
  - After: 10.39s ... 6x improvement!
  - But still not quite ~5s

\[
\begin{align*}
\text{for } (i = 0 \text{ to } N) \\
& \quad \text{for } (j = 0 \text{ to } N) \\
& \quad \quad \text{for } (k = 0 \text{ to } N) \\
& \quad \quad \quad C[i][j] += A[i][k] * B^T[j][k]
\end{align*}
\]
Overheads in matrix multiplication (2)

- Both A and B read N times
  - A re-uses each row before moving on to next
  - B scans the whole matrix for each row of A
  - One row: 2048 * 4 bytes = 8192 bytes fits in L1 cache (32 KB)
  - One matrix: 2048 * 2048 * 4 bytes = 16 MB exceeds in L3 cache (6 MB shared across 4 cores)
  - No caching effect for B!

\[
A \times B^T = C
\]

for \( i = 0 \) to \( N \)
for \( j = 0 \) to \( N \)
for \( k = 0 \) to \( N \)
\[
C[i][j] += A[i][k] \times B^T[j][k]
\]
Overheads in matrix multiplication (2)

- One solution: “Blocked” access
  - Assuming BxB fits in cache,
  - B is read only N/B times from memory
- Performance improvement!
  - No optimizations: 63.19s
  - After transpose: 10.39s
  - After transpose + blocking: 7.35

\[
A \times B^T = C
\]

\[
\text{C1 sub-matrix} = A1 \times B1 + A1 \times B2 + A1 \times B3 \ldots A2 \times B1 \ldots
\]
Aside: Cache oblivious algorithms

- For sub-block size $B \times B \rightarrow N \times N \times (N/B)$ reads. What $B$ do we use?
  - Optimized for L1? (32 KiB for me, who knows for who else?)
  - If $B \times B$ exceeds cache, sharp drop in performance
  - If $B \times B$ is too small, gradual loss of performance

- Do we ignore the rest of the cache hierarchy?
  - Say $B$ optimized for L3,
    - $B \times B$ multiplication is further divided into $T \times T$ blocks for L2 cache
  - $T \times T$ multiplication is further divided into $U \times U$ blocks for L1 cache
  - ... If we don’t, we lose performance

- Class of “cache-oblivious algorithms”
  
  Typically recursive definition of data structures... topic for another day
Aside: Recursive Matrix Multiplication

\[
C_{11} C_{12} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11}B_{11} & A_{11}B_{12} \\ A_{21}B_{11} & A_{21}B_{12} \end{bmatrix} + \begin{bmatrix} A_{12}B_{21} & A_{12}B_{22} \\ A_{22}B_{21} & A_{22}B_{22} \end{bmatrix}
\]

8 multiply-adds of \((n/2) \times (n/2)\) matrices
Recurse down until very small