Memory Hierarchy: Motivation

- The gap between CPU performance and main memory speed has been widening with higher performance CPUs creating performance bottlenecks for memory access instructions.

- The memory hierarchy is organized into several levels of memory or storage with the smaller, more expensive, and faster levels closer to the CPU: registers, then primary or Level 1 (L1) Cache, then possibly one or more secondary cache levels (L2, L3…), then main memory, then mass storage (virtual memory).

- Each level of the hierarchy is a subset of the level below: data found in a level is also found in the level below but at a lower speed.

- Each level maps addresses from a larger physical memory/storage level to a smaller level above it.

- This concept is greatly aided by the principal of locality both temporal and spatial which indicates that programs tend to reuse data and instructions that they have used recently or those stored in their vicinity leading to working set of a program.
From Recent Technology Trends

<table>
<thead>
<tr>
<th></th>
<th>Capacity</th>
<th>Speed (latency)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logic:</td>
<td>2x in 3 years</td>
<td>2x in 3 years</td>
</tr>
<tr>
<td>DRAM:</td>
<td>4x in 3 years</td>
<td>2x in 10 years</td>
</tr>
<tr>
<td>Disk:</td>
<td>4x in 3 years</td>
<td>2x in 10 years</td>
</tr>
</tbody>
</table>

**DRAM**

<table>
<thead>
<tr>
<th>Year</th>
<th>Size</th>
<th>Cycle Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1980</td>
<td>64 Kb</td>
<td>250 ns</td>
</tr>
<tr>
<td>1983</td>
<td>256 Kb</td>
<td>220 ns</td>
</tr>
<tr>
<td>1986</td>
<td>1 Mb</td>
<td>190 ns</td>
</tr>
<tr>
<td>1989</td>
<td>4 Mb</td>
<td>165 ns</td>
</tr>
<tr>
<td>1992</td>
<td>16 Mb</td>
<td>145 ns</td>
</tr>
<tr>
<td>1995</td>
<td>64 Mb</td>
<td>120 ns</td>
</tr>
</tbody>
</table>

**1000:1 2:1**
Memory Hierarchy: Motivation
Processor-Memory (DRAM) Performance Gap

Processor-Memory Performance Gap:
(grows 50% / year)

μProc
60%/yr.

DRAM
7%/yr.

Performance

To illustrate the performance impact, assume a pipelined RISC CPU with CPI = 1 using non-ideal memory. 

Over an 10 year period, ignoring other factors, the cost of a full memory access in terms of number of wasted instructions:

<table>
<thead>
<tr>
<th>Year</th>
<th>CPU speed MHZ</th>
<th>CPU cycle ns</th>
<th>Memory Access ns</th>
<th>Minimum CPU cycles or instructions wasted</th>
</tr>
</thead>
<tbody>
<tr>
<td>1986:</td>
<td>8</td>
<td>125</td>
<td>190</td>
<td>190/125 = 1.5</td>
</tr>
<tr>
<td>1988:</td>
<td>33</td>
<td>30</td>
<td>175</td>
<td>175/30 = 5.8</td>
</tr>
<tr>
<td>1991:</td>
<td>75</td>
<td>13.3</td>
<td>155</td>
<td>155/13.3 = 11.65</td>
</tr>
<tr>
<td>1994:</td>
<td>200</td>
<td>5</td>
<td>130</td>
<td>130/5 = 26</td>
</tr>
<tr>
<td>1996:</td>
<td>300</td>
<td>3.33</td>
<td>100</td>
<td>110/3.33 = 33</td>
</tr>
</tbody>
</table>
Memory Hierarchy: Motivation
The Principle Of Locality

- Programs usually access a relatively small portion of their address space (instructions/data) at any instant of time (program working set).

- Two Types of locality:
  - Temporal Locality: If an item is referenced, it will tend to be referenced again soon.
  - Spatial locality: If an item is referenced, items whose addresses are close will tend to be referenced soon.

- The presence of locality in program behavior, makes it possible to satisfy a large percentage of program access needs using memory levels with much less capacity than program address space.
Levels of The Memory Hierarchy

Part of The On-chip
CPU Datapath
16-256 Registers

One or more levels (Static RAM):
Level 1: On-chip 16-64K
Level 2: On or Off-chip 128-512K
Level 3: Off-chip 128K-8M

Dynamic RAM (DRAM)
16M-16G

Interface:
SCSI, RAID,
IDE, 1394,
4G-100G

Farther away from
The CPU
Lower Cost/Bit
Higher Capacity
Increased Access
Time/Latency
Lower Throughput
A Typical Memory Hierarchy
(With Two Levels of Cache)

- Faster
- Larger Capacity

Processor

Control

Datapath

Registers

On-Chip Level
One Cache

L₁

Second Level
Cache (SRAM)

L₂

Main Memory

(DRAM)

Virtual Memory,
Secondary Storage

(Disk)

Tertiary Storage

(Tape)

Speed (ns): 1s 10s 100s 10,000,000s (10s ms) 10,000,000,000s (10s sec)

Size (bytes): 100s Ks Ms Gs Ts
Memory Hierarchy Operation

If an instruction or operand is required by the CPU, the levels of the memory hierarchy are searched for the item starting with the level closest to the CPU (Level 1, L₁ cache):

− If the item is found, it’s delivered to the CPU resulting in a cache hit without searching lower levels.
− If the item is missing from an upper level, resulting in a cache miss, then the level just below is searched.
− For systems with several levels of cache, the search continues with cache level 2, 3 etc.
− If all levels of cache report a miss then main memory is accessed for the item.

  • CPU ↔ cache ↔ memory: Managed by hardware.
− If the item is not found in main memory resulting in a page fault, then disk (virtual memory) is accessed for the item.
  • Memory ↔ disk: Managed by hardware and the operating system.
Memory Hierarchy: Terminology

- **A Block**: The smallest unit of information transferred between two levels.
- **Hit**: Item is found in some block in the upper level (example: Block X)
  - **Hit Rate**: The fraction of memory access found in the upper level.
  - **Hit Time**: Time to access the upper level which consists of RAM access time + Time to determine hit/miss
- **Miss**: Item needs to be retrieved from a block in the lower level (Block Y)
  - **Miss Rate** = 1 - (Hit Rate)
  - **Miss Penalty**: Time to replace a block in the upper level + Time to deliver the block the processor
- **Hit Time << Miss Penalty**

![Memory Hierarchy Diagram]

**Diagram**:
- **Upper Level Memory**: Blk X
- **Lower Level Memory**: Blk Y
- **To Processor**: From Processor
Cache Concepts

- Cache is the first level of the memory hierarchy once the address leaves the CPU and is searched first for the requested data.

- If the data requested by the CPU is present in the cache, it is retrieved from cache and the data access is a cache hit otherwise a cache miss and data must be read from main memory.

- On a cache miss a block of data must be brought in from main memory to into a cache block frame to possibly replace an existing cache block.

- The allowed block addresses where blocks can be mapped into cache from main memory is determined by cache placement strategy.

- Locating a block of data in cache is handled by cache block identification mechanism.

- On a cache miss the cache block being removed is handled by the block replacement strategy in place.

- When a write to cache is requested, a number of main memory update strategies exist as part of the cache write policy.
Cache Design & Operation Issues

Q1: Where can a block be placed cache?  
(Block placement strategy & Cache organization)  
• Fully Associative, Set Associative, Direct Mapped.

Q2: How is a block found if it is in cache?  
(Block identification)  
• Tag/Block.

Q3: Which block should be replaced on a miss?  
(Block replacement policy)  
• Random, Least Recently Used (LRU).

Q4: What happens on a write?  
(Cache write policy)  
• Write through, write back.
We will examine:

- Cache Placement Strategies
  - Cache Organization.
- Locating A Data Block in Cache.
- Cache Replacement Policy.
- What happens on cache Reads/Writes.
- Cache write strategies.
- Cache write miss policies.
- Cache performance.
Cache Organization & Placement Strategies

Placement strategies or mapping of a main memory data block onto cache block frame addresses divide cache into three organizations:

1. **Direct mapped cache**: A block can be placed in one location only, given by:
   
   \[(\text{Block address}) \mod \text{ (Number of blocks in cache)}\]

2. **Fully associative cache**: A block can be placed anywhere in cache.

3. **Set associative cache**: A block can be placed in a restricted set of places, or cache block frames. A set is a group of block frames in the cache. A block is first mapped onto the set and then it can be placed anywhere within the set. The set in this case is chosen by:
   
   \[(\text{Block address}) \mod \text{ (Number of sets in cache)}\]

   If there are \(n\) blocks in a set the cache placement is called \(n\)-way set-associative.
Cache Organization: Direct Mapped Cache

A block can be placed in one location only, given by:

(Block address) MOD (Number of blocks in cache)

In this case: (Block address) MOD (8)

8 cache block frames

32 memory blocks cacheable

(cache address) MOD (100) = 101

(cache address) MOD (8)
Direct Mapped Cache

Example

1024 Blocks
Each block = one word
Can cache up to
$2^{32}$ bytes of memory
Direct Mapped Cache Example

4K blocks
Each block = four words

Takes better advantage of spatial locality
Cache Organization:
Set Associative Cache

One-way set associative
(direct mapped)

Block  Tag  Data
0
1
2
3
4
5
6
7

Two-way set associative

Set  Tag  Data  Tag  Data
0
1
2
3

Four-way set associative

Set  Tag  Data  Tag  Data  Tag  Data  Tag  Data
0
1

Eight-way set associative (fully associative)

Tag  Data  Tag  Data  Tag  Data  Tag  Data  Tag  Data  Tag  Data  Tag  Data  Tag  Data
Cache Organization Example

Fully associative: block 12 can go anywhere

Direct mapped: block 12 can go only into block 4 (12 mod 8)

Set associative: block 12 can go anywhere in set 0 (12 mod 4)

FIGURE 5.2 This example cache has eight block frames and memory has 32 blocks.
Locating A Data Block in Cache

- Each block frame in cache has an address tag.
- The tags of every cache block that might contain the required data are checked or searched in parallel.
- A valid bit is added to the tag to indicate whether this entry contains a valid address.
- The address from the CPU to cache is divided into:
  - A block address, further divided into:
    - An index field to choose a block set in cache.
      (no index field when fully associative).
    - A tag field to search and match addresses in the selected set.
  - A block offset to select the data from the block.

Block Address

- Tag
- Index

Block Offset
Address Field Sizes

Physical Address Generated by CPU

Block Address

Tag

Index

Block Offset

Block offset size = \log_2(\text{block size})

Index size = \log_2(\text{Total number of blocks/associativity})

Tag size = address size - index size - offset size
Four-Way Set Associative Cache: MIPS Implementation Example

256 sets
1024 block frames
Cache Organization/Addressing Example

• Given the following:
  – A single-level L₁ cache with 128 cache block frames
    • Each block frame contains four words (16 bytes)
  – 16-bit memory addresses to be cached (64K bytes main memory or 4096 memory blocks)

• Show the cache organization/mapping and cache address fields for:
  • Fully Associative cache.
  • Direct mapped cache.
  • 2-way set-associative cache.
Cache Example: Fully Associative Case

- Block Address = 12 bits
- Tag = 12 bits
- Block offset = 4 bits

All 128 tags must be checked in parallel by hardware to locate a data block.
Cache Example: Direct Mapped Case

- Block offset = 4 bits
- Index = 7 bits
- Tag = 5 bits
- Block Address = 12 bits

Only a single tag must be checked in parallel to locate a data block in parallel.
Cache Example: 2-Way Set-Associative

- Block offset = 4 bits
- Block Address = 12 bits
- Tag = 6 bits
- Index = 6 bits

Main Memory

Valid bits not shown

Two tags in a set must be checked in parallel to locate a data block
Calculating Number of Cache Bits Needed

• How many total bits are needed for a direct-mapped cache with 64 KBytes of data and one word blocks, assuming a 32-bit address?
  – 64 Kbytes = 16 K words = $2^{14}$ words = $2^{14}$ blocks
  – Block size = 4 bytes => offset size = 2 bits,
  – #sets = #blocks = $2^{14}$ => index size = 14 bits
  – Tag size = address size - index size - offset size = 32 - 14 - 2 = 16 bits
  – Bits/block = data bits + tag bits + valid bit = 32 + 16 + 1 = 49
  – Bits in cache = #blocks x bits/block = $2^{14} x 49 = 98$ Kbytes

• How many total bits would be needed for a 4-way set associative cache to store the same amount of data?
  – Block size and #blocks does not change.
  – #sets = #blocks/4 = ($2^{14}$)/4 = $2^{12}$ => index size = 12 bits
  – Tag size = address size - index size - offset = 32 - 12 - 2 = 18 bits
  – Bits/block = data bits + tag bits + valid bit = 32 + 18 + 1 = 51
  – Bits in cache = #blocks x bits/block = $2^{14} x 51 = 102$ Kbytes

• Increase associativity => increase bits in cache
Calculating Cache Bits Needed

• How many total bits are needed for a direct-mapped cache with 64 KBytes of data and 8 word blocks, assuming a 32-bit address (it can cache $2^{32}$ bytes in memory)?
  - 64 Kbytes = $2^{14}$ words = $(2^{14})/8 = 2^{11}$ blocks
  - block size = 32 bytes
    => offset size = block offset + byte offset = 5 bits,
  - #sets = #blocks = $2^{11}$ => index size = 11 bits
  - tag size = address size - index size - offset size = 32 - 11 - 5 = 16 bits
  - bits/block = data bits + tag bits + valid bit = $8 \times 32 + 16 + 1 = 273$ bits
  - bits in cache = #blocks x bits/block = $2^{11} \times 273 = 68.25$ Kbytes
• Increase block size => decrease bits in cache.
When a cache miss occurs the cache controller may have to select a block of cache data to be removed from a cache block frame and replaced with the requested data, such a block is selected by one of two methods:

- **Random**:  
  - Any block is randomly selected for replacement providing uniform allocation.  
  - Simple to build in hardware.  
  - The most widely used cache replacement strategy.

- **Least-recently used (LRU)**:  
  - Accesses to blocks are recorded and the block replaced is the one that was not used for the longest period of time.  
  - LRU is *expensive* to implement, as the number of blocks to be tracked increases, and is usually approximated.
Cache Read/Write Operations

• Statistical data suggest that reads (including instruction fetches) dominate processor cache accesses (writes account for 25% of data cache traffic).

• In cache reads, a block is read at the same time while the tag is being compared with the block address. If the read is a hit the data is passed to the CPU, if a miss it ignores it.

• In cache writes, modifying the block cannot begin until the tag is checked to see if the address is a hit.

• Thus in cache writes, tag checking cannot take place in parallel, and only the specific data requested by the CPU can be modified.

• Cache is classified according to the write and memory update strategy in place as: write through, or write back.
Cache Write Strategies

1 Write Though: Data is written to both the cache block and to a block of main memory.
   - The lower level always has the most updated data; an important feature for I/O and multiprocessing.
   - Easier to implement than write back.
   - A write buffer is often used to reduce CPU write stall while data is written to memory.

2 Write back: Data is written or updated only to the cache block frame. The modified cache block is written to main memory when it’s being replaced from cache.
   - Writes occur at the speed of cache.
   - A status bit called a dirty bit, is used to indicate whether the block was modified while in cache; if not the block is not written to main memory.
   - Uses less memory bandwidth than write through.
Cache Write Miss Policy

• Since data is usually not needed immediately on a write miss two options exist on a cache write miss:

Write Allocate:
The cache block is loaded on a write miss followed by write hit actions.

No-Write Allocate:
The block is modified in the lower level (lower cache level, or main memory) and not loaded into cache.

While any of the above two write miss policies can be used with either write back or write through:

• Write back caches use write allocate to capture subsequent writes to the block in cache.

• Write through caches usually use no-write allocate since subsequent writes still have to go to memory.
## Miss Rates for Caches with Different Size, Associativity & Replacement Algorithm

### Sample Data

<table>
<thead>
<tr>
<th>Associativity:</th>
<th>2-way</th>
<th></th>
<th>4-way</th>
<th></th>
<th>8-way</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2-way</td>
<td>4-way</td>
<td>8-way</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Size</td>
<td>LRU</td>
<td>Random</td>
<td>LRU</td>
<td>Random</td>
<td>LRU</td>
</tr>
<tr>
<td>16 KB</td>
<td>5.18%</td>
<td>5.69%</td>
<td>4.67%</td>
<td>5.29%</td>
<td>4.39%</td>
</tr>
<tr>
<td>64 KB</td>
<td>1.88%</td>
<td>2.01%</td>
<td>1.54%</td>
<td>1.66%</td>
<td>1.39%</td>
</tr>
<tr>
<td>256 KB</td>
<td>1.15%</td>
<td>1.17%</td>
<td>1.13%</td>
<td>1.13%</td>
<td>1.12%</td>
</tr>
</tbody>
</table>
Cache Performance

For a CPU with a single level (L1) of cache and no stalls for cache hits:

With ideal memory

CPU time = (CPU execution clock cycles +
Memory stall clock cycles) x clock cycle time

Memory stall clock cycles =
(Reads x Read miss rate x Read miss penalty) +
(Writes x Write miss rate x Write miss penalty)

If write and read miss penalties are the same:

Memory stall clock cycles =
Memory accesses x Miss rate x Miss penalty
Cache Performance

CPUtime = Instruction count x CPI x Clock cycle time

\[ \text{CPUtime} = \text{Instruction Count} \times (\text{CPI}_{\text{execution}} + \text{Mem Stall cycles per instruction}) \times \text{Clock cycle time} \]

Mem Stall cycles per instruction =

\[ \text{Mem accesses per instruction} \times \text{Miss rate} \times \text{Miss penalty} \]

CPUtime = IC x (CPI_{execution} + Mem accesses per instruction x Miss rate x Miss penalty) x Clock cycle time

Misses per instruction = Memory accesses per instruction x Miss rate

CPUtime = IC x (CPI_{execution} + Misses per instruction x Miss penalty) x Clock cycle time
Cache Performance Example

- Suppose a CPU executes at Clock Rate = 200 MHz (5 ns per cycle) with a single level of cache.
- CPI_{execution} = 1.1
- Instruction mix: 50% arith/logic, 30% load/store, 20% control
- Assume a cache miss rate of 1.5% and a miss penalty of 50 cycles.

\[
\text{CPI} = \text{CPI}_{\text{execution}} + \text{mem stalls per instruction}
\]

\[
\text{Mem Stalls per instruction} = \text{Mem accesses per instruction} \times \text{Miss rate} \times \text{Miss penalty}
\]

\[
\text{Mem accesses per instruction} = 1 + .3 = 1.3
\]

\[
\text{Mem Stalls per instruction} = 1.3 \times .015 \times 50 = 0.975
\]

\[
\text{CPI} = 1.1 + .975 = 2.075
\]

The ideal CPU with no misses is \(2.075/1.1 = 1.88\) times faster
Cache Performance Example

• Suppose for the previous example we double the clock rate to 400 MHZ, how much faster is this machine, assuming similar miss rate, instruction mix?

• Since memory speed is not changed, the miss penalty takes more CPU cycles:

  Miss penalty = 50 x 2 = 100 cycles.

  CPI = 1.1 + 1.3 x .015 x 100 = 1.1 + 1.95 = 3.05

  Speedup = \( \frac{\text{CPI}_{\text{old}} \times C_{\text{old}}}{\text{CPI}_{\text{new}} \times C_{\text{new}}} \)
  = \( \frac{2.075 \times 2}{3.05} = 1.36 \)

The new machine is only 1.36 times faster rather than 2 times faster due to the increased effect of cache misses.

→ CPUs with higher clock rate, have more cycles per cache miss and more memory impact on CPI.
3 Levels of Cache

CPU

L1 Cache
Hit Rate= $H_1$, Hit time = 1 cycle

L2 Cache
Hit Rate= $H_2$, Hit time = $T_2$ cycles

L3 Cache
Hit Rate= $H_3$, Hit time = $T_3$ cycles

Main Memory

Memory access penalty, $M$
3-Level Cache Performance

\[ \text{CPUtime} = \text{IC} \times (\text{CPI}_{\text{execution}} + \text{Mem Stall cycles per instruction}) \times C \]

Mem Stall cycles per instruction = Mem accesses per instruction \times Stall cycles per access

- For a system with 3 levels of cache, assuming no penalty when found in L\(_1\) cache:

Stall cycles per memory access =

\[
[\text{miss rate L}_1] \times [\text{Hit rate L}_2 \times \text{Hit time L}_2
\]
\[
+ \text{Miss rate L}_2 \times (\text{Hit rate L}_3 \times \text{Hit time L}_3
\]
\[
+ \text{Miss rate L}_3 \times \text{Memory access penalty}) ] =
\]

\[
[1 - H_1] \times [H_2 \times T_2
\]
\[
+ (1-H_2) \times (H_3 \times (T_2 + T_3)
\]
\[
+ (1 - H_3) \times M) ]
\]
Three Level Cache Example

- CPU with $CPI_{\text{execution}} = 1.1$ running at clock rate = 500 MHZ
- 1.3 memory accesses per instruction.
- $L_1$ cache operates at 500 MHZ with a miss rate of 5%
- $L_2$ cache operates at 250 MHZ with miss rate 3%, ($T_2 = 2$ cycles)
- $L_3$ cache operates at 100 MHZ with miss rate 1.5%, ($T_3 = 5$ cycles)
- Memory access penalty, $M=100$ cycles. Find CPI.

With single $L_1$, $CPI = 1.1 + 1.3 \times .05 \times 100 = 7.6$

$$CPI = CPI_{\text{execution}} + \text{Mem Stall cycles per instruction}$$

$$\text{Mem Stall cycles per instruction} = \text{Mem accesses per instruction} \times \text{Stall cycles per access}$$

Stall cycles per memory access = $[1 - H_1] \times [H_2 \times T_2 + (1-H_2) \times (H_3 \times (T_2 + T_3)) + (1 - H_3) \times M]$ 

$$= [.05] \times [.97 \times 2 + (.03) \times (.985 \times (2+5) + .015 \times 100)]$$

$$= .05 \times [1.94 + .03 \times (6.895 + 1.5)]$$

$$= .05 \times [1.94 + .274] = .11$$

- $CPI = 1.1 + 1.3 \times .11 = 1.24$