Computer Architecture II
Caches

Syed Asad Alam
School of Computer Science and Statistics

Introduction

Main Computer Components

- Main components of a computer?
### Our Needs from Memory

- Why do we need memory? → To store data and program (von Neumann)
- How much memory do we need? → As much as possible
- How fast should it be? → As fast as possible
- How costly should it be? → As low as possible
- A set of contradictory requirements!!!

### Solving the Contradictory Requirements

- By creating an illusion
- Analogy → Doing grocery for your home

- Second analogy → Reading books in a library
Underlying Principle

- Two underlying principles
  - Locality
  - Having a hierarchy

Types of Locality

- A data is more probable to be accessed more than once → Temporal locality (Buying a number of same items)
- A data next to an already read data is more probable to be accessed soon → Spatial locality (Fetching similar books lying next to the one you fetched)
- A combination of these two localities help create the illusion

Memory Hierarchy
Access Time – Example

- Processor with two levels of memory
- Level 1 → 1000 words, 0.011µs access time
- Level 2 → 100,000 words, 0.11µs access time
- 95% of memory accesses in level 1, what is the average access time?

How does the Processor know?

- Does the processor know the location of data?
- Typically looks for data in the upper level memory in the hierarchy
- If present, reads the data → Hit
- If not, copies data from a lower level memory into the higher level memory → Miss
- Data copied only between two adjacent levels, in blocks of data

Hits and Misses

- Hit ratio = \( \frac{\text{Number of hits}}{\text{Number of memory accesses}} \)
- Higher the hit ratio, better the performance of the memory hierarchy
- Hit time → Time to access a higher level of memory
- Miss time/penalty → Time to access a lower level of memory
- Which one is longer?
- Miss time > Hit time, but why?
  \[ T_{\text{av}} = h \times T_{\text{cache}} + (1 - h) \times T_{\text{miss}} \]
- Small changes in hit ratio [as \( h \rightarrow 1 \)] → Amplified by \( \frac{T_{\text{miss}}}{T_{\text{cache}}} \)
- If ratio is 10 → ↓ 1% in \( h \rightarrow \uparrow 10\% \) in \( T_{\text{av}} \)
Why Block of Data?

- Why not just the exact data needed?
- To exploit spatial locality
- Should we keep the data once used?
- Yes → To exploit temporal locality

![Image](chart.png)

Cache Memory – Principles

Need of cache?

- CPU is fast
- Main memory is slow
- Solution
  - Make main memory fast → Expensive
  - Small capacity of fast memory, named cache, between CPU and main memory → Making use of the principle of locality
- Is cache visible to programmer? → No, only some CPU registers and main memory visible to programmer

![Image](chart2.png)
**Location of Cache**

- Earlier cache were off-chip
- Currently all levels are on the same chip as the CPU

**Levels of cache?**

**Structure of Cache and Main Memory**
Structure of Cache and Main Memory

- $M = 2^n / K$ blocks in memory
- $m$ blocks in cache $\rightarrow$ Lines
- Each cache line has tag and control bits (not shown)
- Line size (w/o tag/ctrl) = block size with $m << M$
- Data to be read copied from main memory to cache
- Purpose of tag bits? $\rightarrow$ Identify the memory block

Cache Read Operation

Cache Memory – Design
Design Parameters

- Addressing and mapping function
- Size
- Replacement algorithm
- Write policy
- Block size
- Number of caches

Cache Size

- Small enough to reduce cost
- Large enough to reduce the access time
- Large caches are slower than small caches
- No real optimum

<table>
<thead>
<tr>
<th>Processor</th>
<th>Type</th>
<th>Year of Introduction</th>
<th>1 cache</th>
<th>2 caches</th>
<th>3 caches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Xeon</td>
<td>Haswell</td>
<td>2020</td>
<td>0.8 x 1.5B</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>AMD Ryzen</td>
<td>7000</td>
<td>2021</td>
<td>0.9</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Apple M1</td>
<td>2020</td>
<td>0.9B-1.0B</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Intel Core i7</td>
<td>6700K</td>
<td>2015</td>
<td>0.9</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Intel Core i5</td>
<td>6600K</td>
<td>2015</td>
<td>0.9</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Intel Core i3</td>
<td>6100</td>
<td>2015</td>
<td>0.9</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>AMD Fusion E2</td>
<td>2016</td>
<td>0.8</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>AMD A6-3200</td>
<td>2012</td>
<td>0.8B</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Intel Core i7-2600K</td>
<td>2011</td>
<td>0.8B</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Intel Core i7-2600</td>
<td>2011</td>
<td>0.8B</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Intel Core i5-2500K</td>
<td>2011</td>
<td>0.8B</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Intel Core i5-2500</td>
<td>2011</td>
<td>0.8B</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Intel Core i5-2400</td>
<td>2011</td>
<td>0.8B</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>AMD Athlon II X2</td>
<td>2010</td>
<td>0.8B</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>AMD Athlon II X3</td>
<td>2010</td>
<td>0.8B</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>AMD Athlon II X4</td>
<td>2010</td>
<td>0.8B</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>
Cache Mapping Function

- Memory size > Cache size
- Where does a memory block read be placed in cache?
- What is the mapping between memory block and cache line
- To decide, we need a mapping function

Three types of mapping function:
- Direct
- Associative
- Set associative

Direct Mapping

- A fixed mapping mechanism
- A number of memory blocks map to the same cache line
- But how?
- Let's do an example

Direct Mapping – Example

<table>
<thead>
<tr>
<th>Memory</th>
<th>Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

First 8 blocks:
- Memory size = 32 words or bytes
- Block size = 1 word or byte
- Cache size = 8 words or bytes
- Where does the next 8 blocks map?
  → Same set of line
Direct Mapping – Example

\[ i = j \mod m \]

- **Cache contents**

<table>
<thead>
<tr>
<th>Cache line no.</th>
<th>Mem. block no.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0, 8, 16, 24</td>
</tr>
<tr>
<td>1</td>
<td>1, 9, 17, 25</td>
</tr>
<tr>
<td>2</td>
<td>2, 10, 18, 26</td>
</tr>
<tr>
<td>3</td>
<td>3, 11, 19, 27</td>
</tr>
<tr>
<td>4</td>
<td>4, 12, 20, 28</td>
</tr>
<tr>
<td>5</td>
<td>5, 13, 17, 29</td>
</tr>
<tr>
<td>6</td>
<td>6, 14, 16, 30</td>
</tr>
<tr>
<td>7</td>
<td>7, 15, 19, 31</td>
</tr>
</tbody>
</table>

Direct Mapping – Decoding the Address

Mapping can be identified looking at the address bit positions

- 32 word memory → How many bits for address?
  \[ \log_2 32 = 5 \text{ bits} \]
- Addresses for the first 8 blocks
  \[
  00000, 00001, 00010, 00011, 00100, 00101, 00110, 00111
  \]
- Address for the next 8 blocks
  \[
  01000, 01010, 01011, 01100, 01110, 01101, 01111
  \]
- Addresses for the first 8 blocks
  \[
  00000, 00001, 00010, 00011, 00100, 00101, 00110, 00111
  \]
- Address for the next 8 blocks
  \[
  01000, 01010, 01011, 01100, 01110, 01101, 01111
  \]
- Address for the next 8 blocks starting from 16?

Direct Mapping – Address Fields

- Addresses for the first 8 blocks
  \[
  00000, 00001, 00010, 00011, 00100, 00101, 00110, 00111
  \]
- Address for the next 8 blocks
  \[
  01000, 01010, 01011, 01100, 01101, 01110, 01111
  \]
- The last three bits identify the cache line to which a memory block maps

  - Role of first 2 bits?
  - How do you recognize which block is occupying a cache line?
    - By looking at the first 2 bits
    - Tag bits
    - Tag stored together with the copied data
Direct Mapping – Anything left?

- Assumed 1 word block
- To exploit spatial locality, blocks contain more than 1 word
- After finding the cache line where data resides and matching the tag, how do we recognize the word within the block?
- Extra bits needed for the address

### Example?

<table>
<thead>
<tr>
<th>Tag</th>
<th>Line</th>
<th>Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>r</td>
<td>w</td>
</tr>
</tbody>
</table>

Direct Mapping – Operation

---

Direct Mapping – Summary

- Address length = \((s + w)\) bits
- Number of addressable units = \(2^{s+w}\) words or bytes
- Block size = line size = \(2^w\) words or bytes
- Number of blocks in main memory = \(M = \frac{2^s}{2^w} = 2^s\)
- Number of lines in cache = \(m = 2^r\)
- Size of cache = \(2^{r+w}\) words or bytes

---

\(^1\) Comp. Org. Arch. by W. Stallings
Direct Mapping – Example

Consider a machine with a byte addressable main memory of $2^{16}$ bytes and block size of 8 bytes. Assume a direct mapped cache consisting of 32 lines is used with this machine.

- How is a 16-bit memory divided into tag, line and byte fields?
- Into what line would bytes with the following addresses be stored?
  - 0001000100011011
  - 1100001100110100
  - 1101000000011101
  - 1010101010101010
- Suppose the byte with address 0001010000110101 is stored in the cache. What are the addresses of the other bytes?
- How many total bytes of memory can be stored in the cache at a given time?

Direct Mapping – Another Example

A very simple computer operates with main memory that is capable to store 1024 bytes. In order to speed up the processing, a cache memory having total size of 32 bytes is employed in between the CPU and main memory. It is known that 1 word is equivalent to 1 byte.

A CPU operation requires access to the memory addresses in the following order:
- 1
- 0
- 7
- 4
- 1
- 6
- 25
- 32
- 5
- 8
- 13
- 1
- 3
- 11
- 17

Suppose that direct mapping is used with 1-byte blocks.

- Explain the structure of the main memory address in terms of tag, index and word.
- Suppose the cache is empty at the beginning. By showing the relevant steps, determine the hits and misses from the address references and evaluate the final content of the cache.

Direct Mapping – Pros and Cons

- Pros:
  - Simple to implement
  - Need to compare only one tag
- Cons:
  - A number of locations are mapped to one fixed location
  - Multiple access to blocks mapped to the same line will cause constant misses and blocks will be continually swapped
- Solution?
  - Allow the data block to be placed any where in the cache
  - Associative Mapping
Associative Mapping

- A memory block mapped to any cache line
- Which part of the address field not needed?
- The line field
- Why do we need such a large tag field? → Tag field identifies a unique block among the whole cache

<table>
<thead>
<tr>
<th>Tag</th>
<th>Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s$</td>
<td>$w$</td>
</tr>
</tbody>
</table>

Associative Mapping – Operation

- Comparison of a large number of tags
- Increase in cache size as tag needs to be stored with the cache
Associative Mapping – Summary

- Address length = \( (s + w) \) bits
- Number of addressable units = \( 2^{s+w} \) words or bytes
- Block size = line size = \( 2^w \) words or bytes
- Number of blocks in main memory = \( M = \frac{2^s}{2^w} = 2^s \)
- Number of lines in cache = Non-deterministic from the address size
- Size of tag = \( s \) words or bytes

Set Associative Mapping

- Flexible yet fixed → A compromise between direct and associative mapped cache
- How?
  - Divide the cache into sets
  - A cache set consists of a number of lines
  - A memory block maps to only one set (the fixed part)
  - Within each set, a memory block can be placed anywhere (the flexible part)

Example

- Memory size = 32 words or bytes
- Block size = 1 word or byte
- Cache size = 8 words or bytes
- Set associativity = 2-way\((k\text{-way})\) set associative (Each set has two cache lines)
- How many total sets? \( \rightarrow 4 \) sets \((v)\)
- How many bits required to identify the set? \( \rightarrow 2 \) bits
Set Associative Mapping – Decoding the Address

Mapping can be identified looking at the address bit positions:

- 32 word memory → How many bits for address? 
  \( \log_2 32 = 5 \) bits
- Addresses for the first 8 blocks: 00000, 00001, 00010, 00011, 00100, 00101, 00110, 00111
- Address for the next 8 blocks: 01000, 01001, 01010, 01011, 01100, 01101, 01110, 01111

Another Perspective for Set Mapping

- Memory Block  | Cache Set
  0  | 0
  1  | 1
  2  | 2
  3  | 3
  4  | 0
  5  | 1
  6  | 2
  7  | 3

  The operation, sounds similar?
  Modulo operation
  Cache set no. \( (i) = \text{Mem block no.}(j) \mod \text{Number of sets}(v) \)

Cache Contents

\[ i = j \mod v \]

- Cache contents
  
<table>
<thead>
<tr>
<th>Cache set no.</th>
<th>Mem. block no.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0, 4, 8, 12, 16, 20, 24, 28</td>
</tr>
<tr>
<td>1</td>
<td>1, 5, 9, 13, 17, 21, 25, 29</td>
</tr>
<tr>
<td>2</td>
<td>2, 6, 10, 14, 18, 22, 26, 30</td>
</tr>
<tr>
<td>3</td>
<td>3, 7, 11, 15, 19, 23, 27, 31</td>
</tr>
</tbody>
</table>
Set Associative – Operation

![Diagram of cache and main memory interaction]

Set Associative – Summary

- Address length = \((s + w)\) bits
- Number of addressable units = \(2^{s+w}\) words or bytes
- Block size = cache size = \(2^w\) words or bytes
- Number of blocks in main memory = \(M = 2^s = 2^s\)
- Number of lines in set = \(k\)
- Number of set = \(v = 2^d\)
- Number of lines in cache = \(m = kv = k \times 2^d\)
- Size of cache = \(k \times 2^{s+w}\) words or bytes

![Diagram of cache structure with labels]

Replacement Algorithm

- In associative mapping → One set (or whole cache) has multiple memory blocks
- What if the set is full? Where does the new data go? Which one does it replace?
- Replacement algorithm needed → Any suggestions?
  - Least recently used (LRU) → Implemented using a use bit (easy for 2-way) or List of indexes. More bits needed for large \(k\) (algo. proposed by Maruyama [IBM] uses \(k^2\) its)
  - Pseudo LRU → One bit for each way, set when a way accessed
  - First-in First-out (FIFO) → Implemented as circular buffer
  - Least frequently used (LFU) → Implemented using counters
  - Not recently used (NRU) → Implemented using referenced and modified bits
  - Random
Implementing LRU – Maruyama [IBM]

Consider → 1,2,0,1,3

<table>
<thead>
<tr>
<th>access line 1</th>
<th>access line 2</th>
<th>access line 0</th>
<th>access line 1</th>
<th>access line 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0</td>
<td>0 1 1 1</td>
<td>0 0 0 0</td>
<td>0 0 1 1</td>
<td>0 0 1 0</td>
</tr>
<tr>
<td>1 0 0 1</td>
<td>1 0 0 1</td>
<td>0 0 0 0</td>
<td>1 0 1 1</td>
<td>1 0 1 0</td>
</tr>
<tr>
<td>0 0 0 0</td>
<td>0 1 1 1</td>
<td>0 0 0 0</td>
<td>0 0 1 1</td>
<td>0 0 1 0</td>
</tr>
<tr>
<td>LRU lines 0, 2 and 3</td>
<td>LRU lines 0 and 3</td>
<td>LRU line 3</td>
<td>LRU line 3</td>
<td>LRU line 2</td>
</tr>
</tbody>
</table>

Notes

Replacement Algorithm – A Comparison

<table>
<thead>
<tr>
<th>Associativity</th>
<th>Two-way</th>
<th>Four-way</th>
<th>Eight-way</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>LRU</td>
<td>Random</td>
<td>FIFO</td>
</tr>
<tr>
<td>32 KI</td>
<td>39.4</td>
<td>39.3</td>
<td>39.3</td>
</tr>
<tr>
<td>64 KI</td>
<td>39.4</td>
<td>39.3</td>
<td>39.3</td>
</tr>
<tr>
<td>256 KI</td>
<td>39.4</td>
<td>39.3</td>
<td>39.3</td>
</tr>
</tbody>
</table>

Figure 8.4: Data cache misses per 1000 instructions comparing least recently used, random, and first in, first out replacements for several sizes and associativities. There is little difference between LRU and random for the largest size cache. With LRU consistently outperforming the others for smaller caches. FIFO generally outperforms random in the smaller cache sizes. These data were collected for a block size of 64 bytes for the X86 architecture using SPECTRE benchmark. Raw and transformed output from SPEC2000: (app, gam, gcc, gzip, mcf, and perl+ and perf) and five are from SPEC2000: (input, art, equake, lucas, and weld). We will use this computer and these benchmarks in most figures in this appendix.
Cache Coherency

- Different data in the cache and memory
- How?
  - Cache modified by processor but not written back to memory
  - An I/O with direct access to memory modifies it
  - Individual cores having individual caches (L1/L2 per core, L3 shared)
- Another related problem → Write miss

Write Policy

- To deal with cache coherency → Need to develop a writing policy
- Two options
  - All write operations to cache are also made to the memory → Write through
  - All write operations to memory are only made when the block is replaced → Write back
Write Through

Advantages
- Memory always valid
- Any other processor cache can maintain consistency if it has the same block

Disadvantages
- High memory traffic
- Writing to memory is slow → Slows down the processor

Write miss
- Use a write buffer
- Write to the main memory independent of the processor

Write Back

Advantages
- Improves performance when processor writes very frequently and at a faster rate
- Almost always fast

Disadvantages
- Complex to implement
- Need to check whether the cache line is modified before replacement
- A cache coherency protocol is needed to maintain consistency

Write miss
- Write allocate

Block/Line Size

- Why do we need multi-word blocks → To exploit spatial locality
- Can we increase the block size arbitrarily? → No, initially hit ratio will increase but then decrease
- But why?
  - Large blocks reduce the total number of blocks and each block will be replaced much more frequently
  - Larger block means each additional word is farther from the requested word and less likely to be referenced

![Graph showing hit ratio vs. block size](image-url)
Number of Caches

- Initially only a single cache
- Extra levels of cache introduced → Reducing miss-rate/miss-penalty
- For a two level cache
  - L1 focus on minimizing hit time → Smaller L1, larger miss-rate, smaller miss-penalty (as compared to single-level cache)
  - L2 focus on minimizing miss-rate/miss-penalty → Larger L2, larger hit time (but not important, Why?)
- Increase in design complexity
- Data relationship between L1 and L2? Two options
  - L1 has a subset of L2 → Allows for different block sizes (Itanium2 has 64-byte L1 blocks and 128-byte L2 blocks)
  - L1 and L2 are exclusive → L1 miss results in a swap of blocks b/w L1 and L2, prevents wasting space in L2 cache (Important when difference in size b/w L1 and L2 not big (AMD))
- For multi-core, L1 and L2 per core, L3 shared

Unified/Split Cache

- Typically L1 is split in instruction and data
- Advantages → Eliminates contention between instruction/data, enables fetching instructions ahead of time
- L2 and L3 are unified
- Advantages → Higher hit rate as load between instruction and data automatically balanced

Cache Example

```
### Cache Example

![Cache Example Image]

### Cache Evolution

<table>
<thead>
<tr>
<th>Processor</th>
<th>Type</th>
<th>Year of Introduction</th>
<th>L1 Cache</th>
<th>L2 Cache</th>
<th>L3 Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM RISC</td>
<td>Mainframe</td>
<td>1990</td>
<td>256 KB</td>
<td>1 MB</td>
<td></td>
</tr>
<tr>
<td>DEC VAX</td>
<td>Mainframe</td>
<td>1978</td>
<td>4 KB</td>
<td>16 KB</td>
<td></td>
</tr>
<tr>
<td>IBM S/36</td>
<td>Mainframe</td>
<td>1993</td>
<td>2 MB</td>
<td>4 MB</td>
<td></td>
</tr>
<tr>
<td>IBM RISC</td>
<td>Mainframe</td>
<td>1990</td>
<td>256 KB</td>
<td>1 MB</td>
<td></td>
</tr>
<tr>
<td>Intel 486</td>
<td>PC</td>
<td>1989</td>
<td>8 KB</td>
<td>8 KB</td>
<td>16 KB</td>
</tr>
<tr>
<td>Sparc SC</td>
<td>SPARC</td>
<td>1986</td>
<td>128 KB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RISC 6010</td>
<td>RISC</td>
<td>1999</td>
<td>128 KB</td>
<td>256 KB</td>
<td>1 MB</td>
</tr>
<tr>
<td>DEC Alpha</td>
<td>Mainframe</td>
<td>1990</td>
<td>256 KB</td>
<td>1 MB</td>
<td></td>
</tr>
<tr>
<td>PowerPC</td>
<td>Mainframe</td>
<td>1990</td>
<td>1 MB</td>
<td>2 MB</td>
<td></td>
</tr>
<tr>
<td>PowerPC</td>
<td>Microserver</td>
<td>1990</td>
<td>1 MB</td>
<td>2 MB</td>
<td></td>
</tr>
<tr>
<td>CISC RISC</td>
<td>Mainframe</td>
<td>1985</td>
<td>256 KB</td>
<td>1 MB</td>
<td></td>
</tr>
<tr>
<td>DEC Alpha</td>
<td>Mainframe</td>
<td>1990</td>
<td>256 KB</td>
<td>1 MB</td>
<td></td>
</tr>
<tr>
<td>PowerPC</td>
<td>Microserver</td>
<td>1990</td>
<td>1 MB</td>
<td>2 MB</td>
<td></td>
</tr>
<tr>
<td>RISC 6010</td>
<td>RISC</td>
<td>1999</td>
<td>128 KB</td>
<td>256 KB</td>
<td>1 MB</td>
</tr>
<tr>
<td>RISC 6010</td>
<td>RISC</td>
<td>1999</td>
<td>128 KB</td>
<td>256 KB</td>
<td>1 MB</td>
</tr>
</tbody>
</table>

### ARM Cache Evolution

<table>
<thead>
<tr>
<th>Core</th>
<th>Cache Type</th>
<th>Cache Size (kB)</th>
<th>Cache Line Size (words)</th>
<th>Associativity</th>
<th>Location</th>
<th>Write Buffer Size (words)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM946EJ-S</td>
<td>Unified</td>
<td>4</td>
<td>4- way</td>
<td>Logical</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>ARM946EJ-S</td>
<td>Split</td>
<td>16/4-32/12</td>
<td>64- way</td>
<td>Logical</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>ARM1136JF-S</td>
<td>Split</td>
<td>4-128/12-128/32</td>
<td>64- way</td>
<td>Logical</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>ARM1176JF-S</td>
<td>Split</td>
<td>4-128/12-128/32</td>
<td>64- way</td>
<td>Logical</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>StrongARM</td>
<td>Split</td>
<td>16/4-32/12</td>
<td>64- way</td>
<td>Logical</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>Mid Range</td>
<td>Split</td>
<td>32/32-32</td>
<td>64- way</td>
<td>Logical</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>ARM1136JF-S</td>
<td>Split</td>
<td>4-64/64-64/64</td>
<td>64- way</td>
<td>Physical</td>
<td>32</td>
<td>32</td>
</tr>
</tbody>
</table>
How can we measure performance? A number of ways
- Instruction count
- Miss rate
- Average memory access time
- Execution time
Types of Misses

- Three types of misses
- Compulsory
  - Misses caused by first access
- Capacity
  - Misses when cache is full
- Conflict
  - Misses when multiple blocks compete for the same cache line
  - Occurs in direct and set-associative mapped caches

What happens on a Miss?

- Hit → Great, data is copied to processor
- Total time → Hit time
- Miss
  - Processor is stalled for a few cycles
  - Data/instruction fetched from memory and copied into cache
  - Miss penalty → Time to copy data from memory to cache

\[
T_{av} = h \cdot T_c + m(T_c + T_{mp})
\]

\[
T_{av} = T_c + mT_{mp}
\]

Example

Which has the lower miss rate: a 16KB instruction cache with a 16KB data cache or a 32KB unified cache? Assume miss per 1000 instructions for 16 KB instruction and data cache and 32KB unified cache to be 3.82, 4.9 and 43.3, respectively. Assume 36% of the instructions are data transfer instructions. 74% of memory references are for instructions and 26% are for data. Assume a hit takes 1 clock cycle and the miss penalty is 200 clock cycles. A load or store hit takes 1 extra clock cycle on a unified cache if there is only one cache port to satisfy two simultaneous requests. What is the average memory access time in each case? Assume write-through caches with a write buffer and ignore stalls due to the write buffer.
CPU Execution Time

- Two components
  - Program cycles (Cycles to execute a program)
  - Stall cycles (Cycles for which processor waits for data)

\[ T = \text{Program cycles} \times \tau \]
\[ T = (\text{Program cycles} + \text{Memory stall cycles (MSC)}) \times \tau \]

Memory Stall Cycles

- Two major components
  - Number of misses
  - Miss penalty in terms of clock cycles

\[ \text{MSC} = \text{Number of misses} \times \text{MP} \]
\[ \text{MSC} = \text{Memory accesses} \times \text{Miss rate} \times \text{MP} \]
\[ \text{MSC} = I_c \times \frac{\text{Memory access}}{\text{Instruction}} \times (1 - h) \times \text{MP} \]

\[ T = (I_c \times \text{CPI} + I_c \times \text{No. of mem. ref. per instr.} \times (1 - h) \times \text{MP}) \times \tau \]

Example

Assume that the cache miss penalty is 200 clock cycles, and all instructions normally take 1.0 clock cycles (ignoring memory stalls). Assume that the average miss rate is 2%, there is an average of 1.5 memory references per instruction, and the average number of cache misses per 1000 instructions is 30. What is the impact on performance when behavior of the cache is included? Calculate the impact using both misses per instruction and miss rate.
Effect of Two Level Cache

- What is miss penalty of L1?
- Time to either access L2 or access memory (if miss in L2)

\[ T_{av} = T_{h1} + (1 - h_1) \times T_{m1} \]
\[ T_{av} = T_{h1} + (1 - h_1) \times (T_{h2} + (1 - h_2) \times T_{m2}) \]

Summary of Performance Equations

Six Basic Cache Optimizations

\[ T_{av} = T_c + mT_{mp} \]

- Three categories of optimization
  - Reducing the miss rate \( \rightarrow \) Larger block size, larger cache size, higher associativity
  - Reducing the miss penalty \( \rightarrow \) Multilevel caches, giving reads priority over reads
  - Reducing the hit time \( \rightarrow \) Avoiding address translation when indexing the cache
Larger Block Size

![Graph showing miss rate versus block size for five different-sized caches.](image)

Figure B.10 Miss rate versus block size for five different-sized caches. Note that miss rate actually goes up if the block size is too large relative to the cache size. Each line represents a cache of different size. Figure B.11 shows the data used to plot these lines. Unfortunately, SPEC2000 traces would take too long if block size were included, so these data are based on SPEC92 on a DECstation 5000 (Gae et al. 1993).

Larger Caches

- **Advantage**: More data in cache reduces miss rate
- **Disadvantage**: Longer hit time and higher cost and power

Higher Associativity

<table>
<thead>
<tr>
<th>Cache size (KB)</th>
<th>Degree associative</th>
<th>Total miss rate</th>
<th>Compulsory</th>
<th>Capacity</th>
<th>Conflict</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1-way</td>
<td>0.09%</td>
<td>0.01%</td>
<td>0.07%</td>
<td>72%</td>
</tr>
<tr>
<td>4</td>
<td>2-way</td>
<td>0.17%</td>
<td>0.01%</td>
<td>0.07%</td>
<td>95%</td>
</tr>
<tr>
<td>4</td>
<td>4-way</td>
<td>0.171%</td>
<td>0.01%</td>
<td>0.07%</td>
<td>99%</td>
</tr>
<tr>
<td>4</td>
<td>8-way</td>
<td>0.171%</td>
<td>0.01%</td>
<td>0.070%</td>
<td>100%</td>
</tr>
</tbody>
</table>

![Table showing miss rate components.](image)

Figure B.13 Average memory access time using miss rates in Figure B.8 for parameters in the example. Solid-like type means that this time is higher than the number to the left, that is, higher associativity increases average memory access time.
Multilevel Caches

\[ T_{av} = T_{h1} + (1 - h1) \times T_{m1} \]
\[ T_{av} = T_{h1} + (1 - h1) \times (T_{h2} + (1 - h2) \times T_{m2}) \]

- Local miss rate → Number of misses divided by memory access to a particular cache
- Global miss rate → Number of misses divided by the total number of memory access
- Better to use misses per instruction

MSC = Misses per instruction,\(_L1\) × \(T_{h1}\) + Misses per instruction,\(_L2\) × \(T_{m2}\)

Example

Suppose that in 1000 memory references there are 40 misses in the first-level cache and 20 misses in the second-level cache. What are the various miss rates? Assume the miss penalty from the L2 cache to memory is 200 clock cycles, the hit time of the L2 cache is 10 clock cycles, the hit time of L1 is 1 clock cycle, and there are 1.5 memory references per instruction. What is the average memory access time and average stall cycles per instruction? Ignore the impact of writes.
Giving Priority to Read Misses

- **Write through**
  - Read miss needs the updated value in write buffer
- **Solution**
  - Wait until buffer is empty
  - Check the contents of the buffer for conflicts (prioritizes reads)
- **Write back**
  - Read miss replaces a dirty block
- **How to proceed?**
  - Write dirty block -> Read memory
  - Write dirty block to buffer -> Read memory -> Write memory (prioritizes reads)

Avoid Address Translation when Indexing Cache

- **Address translation** → Slows down cache access
- **Solution** → Virtual caches (work directly with logical/virtual address)
- **Problems**
  - Page level protection
  - Process switch
  - Aliasing
- **Solution** → Virtually indexed, physically tagged (VIPT) cache
- **Limitation** → Size limited to page size times the associativity of cache
- **Real world** → Intel’s (almost all) L1 caches are VIPT. AMD’s Zen also has VIPT

Example

Figure 8.27: The overall picture of a hypothetical memory hierarchy going from virtual address to L2 cache address. The page size is 4 Kbytes. The L1 is hierarchically associated with the L2. The L2 cache is a direct-mapped LRU replacement policy. Pages are tagged at the L1 and hypervisor or only at the L2. Both L1 and L2 have tags. The virtual addresses hold both logical and physical addresses.
Cache Trace Analysis

Simulating Real Cache Behaviour

- Simulation → To understand the behaviour and calculate estimates of performance numbers of cache before implementing them
- Needed → Address trace of expected workload
- Goal → Achieve actual miss rate
- Problem → Enough trace to negate compulsory misses

Cache Trace Length Requirement

- Cache size: 32KB
- Four-way set associative with 16 byte lines
- Number of sets: 512
- Goal: Miss ratio of 1%
- Trace length → To achieve 4 misses per set: 400 hits per set or 200,000 references
- Trace length for 100 misses per set?
- Trace length for 4 × bigger cache → 8 ×
  - 128KB → 1.6M
  - 512KB → 12.8M
  - 2MB → 102M
Multiple Analysis Per Run

**Idea** → Single simulation run for one pass for $K$-way cache can produce data for several cache simulations for $k$-way caches, where $k < K$

**Applicable with LRU replacement policy**

4-way cache directory (for one set) maintained with a LRU policy

**Implementation** → A vector to keep track of hits of 1–$K$ way cache

If hit at position $i$ → Increment hits[$i$]

Total hits for $k$-way → sum(hits[$i$])
Trace Stripping – Filtering

- Perform analysis on the full trace for the following cache
  - Fixed line size: \( L \)
  - Number of sets: \( N \)
  - One-way associative cache (direct mapped)
- Output → Reduced trace with only addresses that produce misses in the \( N \)-sets, one-way set associative cache
- Trace length → 10% of the original if total miss ratio of 10 percent or less
- Number of misses → Same
- Number of hits, hit/miss ratio → Not same, correct value can be calculated using the original length and total number of misses

Same technique → Applicable if \( N \) varied as powers of 2

Number of misses on the full trace == Number of misses on the reduced trace

addresses which map onto set 0 in the 4 set cache, will map onto set 0 or 4 in the 8 set cache.

Writing A Small Simulation Software

- How to find LRU cache line?

```c
int init_line = 0;
for (int i = 1; i < k; i++)
    if (t[i].cnt < t[init_line].cnt)
        init_line = i;
    t[init_line].tag = xtag;
    t[init_line].cnt = cnt;
```
You are now able to
- Explain why and how caches work
- Explain the organization and operation of caches
- Calculate hits, misses and the 3 Cs given an address trace and cache organization
- Know the difference between virtual and physical caches
- Analyze cache performance parameters
- Get an idea about how to write cache simulation software