Cache Memory

Main memory is slow. **Cache** is a small high-speed memory that creates the illusion of a fast main memory. Stores data from some **frequently used addresses** (of main memory).

**Cache hit** Data found in cache. Results in data transfer at maximum speed.

**Cache miss** Data not found in cache. Processor loads data from M and copies into cache. This results in extra delay, called **miss penalty**.

**Hit ratio** = percentage of memory accesses satisfied by the cache.

**Miss ratio** = 1 - hit ratio
**Cache Line**

Cache is partitioned into **lines** (also called blocks). Each line has **4-64** bytes in it. During data transfer, a **whole line** is read or written.

Each line has a **tag** that indicates the address in M from which the line has been copied.

<table>
<thead>
<tr>
<th>Index</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>ABC</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>DEF</td>
</tr>
</tbody>
</table>

- **Index**
- **Tag**
- **Data**

<table>
<thead>
<tr>
<th>Index</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>DEF</td>
</tr>
<tr>
<td>1</td>
<td>PQR</td>
</tr>
<tr>
<td>2</td>
<td>ABC</td>
</tr>
<tr>
<td>3</td>
<td>XYZ</td>
</tr>
</tbody>
</table>

- **Index**
- **Data**

**Cache**  

**Main Memory**

Cache hit is detected through **an associative search** of all the tags. Associative search provides a **fast response** to the query:

"**Does this key match with any of the tags?**"

Data is read only if a match is found.
Types of Cache

1. Fully Associative
2. Direct Mapped
3. Set Associative

Fully Associative Cache

"No restriction on mapping from M to C."

Associative search of tags is expensive.

Feasible for very small size caches only.
The secret of success

Program locality.

Cache line replacement

To fetch a new line after a miss, an existing line must be replaced. Two common policies for identifying the victim block are

- LRU (Least Recently Used)
- Random

Estimating Average Memory Access Time

\[
\text{Average memory access time} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}
\]

Assume that

Hit time = 5 ns
Miss rate = 10%
Miss penalty = 100 ns.
The average memory access time = 15 ns.
Better performance at a cheaper price.
**Direct-Mapped Cache**

A given memory block can be mapped into one and only cache line. Here is an example of mapping

<table>
<thead>
<tr>
<th>Cache line</th>
<th>Main memory block</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0, 8, 16, 24, ... 8n</td>
</tr>
<tr>
<td>1</td>
<td>1, 9, 17, 25, ... 8n+1</td>
</tr>
<tr>
<td>2</td>
<td>2, 10, 18, 26, ... 8n+2</td>
</tr>
<tr>
<td>3</td>
<td>3, 11, 19, 27, ... 8n+3</td>
</tr>
</tbody>
</table>

**Advantage**
No need of expensive associative search!

**Disadvantage**
Miss rate may go up due to possible increase of mapping conflicts.
Set-Associative Cache

Two-way Set-associative cache

*N-way set-associative cache*

Each M-block can now be mapped into any one of a set of $N$ C-blocks. The sets are predefined. Let there be $K$ blocks in the cache. Then

$N = 1$  Direct-mapped cache  

$N = K$  Fully associative cache

Most commercial cache have $N= 2, 4, \text{ or } 8$. Cheaper than a fully associative cache. Lower miss ratio than a direct mapped cache.

*But direct-mapped cache is the fastest.*
Address translation: an example

Main memory size = 2 KB
Block size = 8 bytes
Cache size = 64 bytes
Set size = 2
No. of sets in cache = 4

No of sets = 4 = 2^2
Block size = 8 = 2^3

<table>
<thead>
<tr>
<th>Tag</th>
<th>index</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Memory address

To locate an M-block in cache, check the tags in the set \( S = (M\text{-block}) \mod (\text{number of sets}) \) i.e. the index field.
### Specification of a cache memory

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Block size</strong></td>
<td>4-64 byte</td>
</tr>
<tr>
<td><strong>Hit time</strong></td>
<td>1-2 cycle</td>
</tr>
<tr>
<td><strong>Miss penalty</strong></td>
<td></td>
</tr>
<tr>
<td>Access</td>
<td>6-10 cycles</td>
</tr>
<tr>
<td>Transfer</td>
<td>2-22 cycles</td>
</tr>
<tr>
<td><strong>Miss rate</strong></td>
<td>1-20%</td>
</tr>
<tr>
<td><strong>Cache size</strong></td>
<td></td>
</tr>
<tr>
<td>L1</td>
<td>8KB-64KB</td>
</tr>
<tr>
<td>L2</td>
<td>128KB-2 MB</td>
</tr>
<tr>
<td><strong>Cache speed</strong></td>
<td></td>
</tr>
<tr>
<td>L1</td>
<td>0.5 ns (8 GB/sec)</td>
</tr>
<tr>
<td>L2*</td>
<td>0.75 ns (6 GB/sec)</td>
</tr>
</tbody>
</table>

What happens to the cache during a write operation?
Writing into Cache

Case 1. Write hit

Write through
Write into C & M

Write back
Write into C only. Update M only when discarding the block containing x

Q1. Isn’t write-through inefficient?
   Not all cache accesses are for write.

Q2. What about data consistency in write-back cache?
   If M is not shared, then who cares?

Most implementations of write through use a Write Buffer.

How does it work?
Case 2. Write miss

(Store X, X is NOT in C)

Write allocate

Allocate a C-block to X.
Load the block containing X from M to C.
Then write into X in C.

Write around

Write directly into X bypassing C.
A state-of-the-art memory hierarchy: multilevel cache

Reading Operation
- Hit in L1.
- Miss in L1, hit in L2, copy from L2.
- Miss in L1, miss in L2, copy from M.

Write Hit
- Write through: Write in L1, L2, M.
- Write back
  - Write in L1 only. Update L2 when discarding an L1 block. Update M when discarding a L2 block.

Write Miss
- Write-allocate or write-around
Inclusion Property

In a consistent state,

- Every valid L1 block can also be found in L2.
- Every valid L2 block can also be found in M.

Average memory access time =

\[(\text{Hit time})_{L1} + (\text{Miss rate})_{L1} \times (\text{Miss penalty})_{L1}\]

\[(\text{Miss penalty})_{L1} = (\text{Hit time})_{L2} + (\text{Miss rate})_{L2} \times (\text{Miss penalty})_{L2}\]

Performance improves with additional level(s) of cache if we can afford the cost.
**Optimal Size of Cache Blocks**

Large block size supports program locality and reduces the miss rate.

But the miss penalty grows linearly, since more bytes are copied from M to C after a miss.

\[ T_{av} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}. \]

The optimal block size is 8-64 bytes. Usually, I-cache has a higher hit ratio than D-cache. Why?