7 Caches

- Computer memory is divided into levels, based on speed/cost.

<table>
<thead>
<tr>
<th>Memory technology</th>
<th>access time</th>
<th>$ per Mbyte</th>
<th>In 1998</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRAM</td>
<td>5 - 25 ns</td>
<td>100 - 250</td>
<td></td>
</tr>
<tr>
<td>DRAM</td>
<td>60 - 120 ns</td>
<td>5 - 10</td>
<td></td>
</tr>
<tr>
<td>Magnetic Disk</td>
<td>10,000,000 - 20,000,000 ns</td>
<td>0.1 - 0.2</td>
<td></td>
</tr>
</tbody>
</table>

- Fastest memory is used at the highest level → closest to processor. Slowest memory at the lowest level → must go through failure at all other intermediate levels.
- Fastest memory is most expensive → smallest. Slowest memory is cheapest → largest.
Locality

- The principle of locality makes it possible to have a memory as large as the slowest, which *seems to run* as fast as the highest level (even though the smallest).

- Two types of locality:
  - *Temporal locality*: If a value is *referenced* (needed from memory), it has a high probability of being referenced again soon (time-wise).
  - *Spatial locality*: If a value is referenced in memory at location A, then there is a high probability that values will be referenced soon from A+i (space-wise), where i is small. (if i is zero, this is the same as temporal locality).
The Speed/Size Hierarchy

<table>
<thead>
<tr>
<th>speed</th>
<th>CPU</th>
<th>size</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>fastest</td>
<td>Memory</td>
<td>smallest</td>
<td>highest</td>
</tr>
<tr>
<td>slowest</td>
<td>Memory</td>
<td>largest</td>
<td>lowest</td>
</tr>
</tbody>
</table>
Interface Between Levels

- Memory at level $i$ is never accessed unless the access has failed at level $i+1$. Note: level $i$ is slower and larger than level $i+1$.
  - Therefore, the same access protocol must hold, in principle, between every pair of levels.
  - We consider only a 2-level memory system, and the interface between those two levels. Interfaces between all other pairs of successive levels will use the same principles.
  - *Virtual memory* is an example of interface between two memory levels: the higher level is usually DRAM; the lower level is usually a disk.
  - In this chapter, we consider a level which is one level higher than main memory in the hierarchy and which is called *cache*. 
Graphic Illustration of a 2-level Memory Hierarchy

- Block-unit of transfer

Processor

Data transferred between levels

Cache

Main Memory

Level i

Level i+1

Processor reads/writes data
Mapping to the Cache

- 3 questions to answer:
  - Where is a datum from memory placed in the cache?
  - How do we find it there when we need it?
  - What happens when we look for data in the cache but it is not there?

- Simplest cache organization: *direct mapped*
  - For every memory location, there is exactly one cache location where it can be stored.
Illustration of Direct Mapping

Last 3 bits of memory address determine cache location

= cache block address/index

• Block = Unit of transfer between memory technologies.
For this cache, the lower portion of the address is used to select a cache entry consisting of a data word and a tag. The tag from the cache is compared against the upper portion of the address to determine whether the entry in the cache corresponds to the requested address. Because the cache has $2^{10}$ or 1024 words, and a block size of 1 word, 10 bits are used to index the cache, leaving $32 - 10 - 2 = 20$ bits to be compared against the tag. If the tag and upper 20 bits of the address are equal and the valid bit is on, then the request hits in the cache and the word is supplied to the processor. Otherwise, a miss occurs.
Finding Data in the Cache (Read)

- Each cache location has a tag.
  - Tag = the address bits which are higher order than the cache block address.
  - Tag uniquely distinguishes every memory location which can map to a specific cache block.
  - Processor read address is divided into 2 parts:
    » Lower order part (cache block address) is used to address the correct cache location.
    » Higher order part is compared with tag at that location.

- If tags match, we have a cache hit.
Cache Miss

- If data are not found in the cache (tag does not match requested address), we have a cache *miss*.
  - Requested data must be read from memory.
  - We assume temporal locality -- the data will be used again soon -- so we write it into cache.
  - Datapath must wait for new data to arrive → control must be *stalled* (insert *wait states*).
Example

<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>

a. The initial state of the cache after power-on.

<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>

b. After handling a miss of address (10110<sub>two</sub>).

c. After handling a miss of address (11010<sub>two</sub>).

<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&lt;sub&gt;two&lt;/sub&gt; Memory (10000&lt;sub&gt;two&lt;/sub&gt;)</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&lt;sub&gt;two&lt;/sub&gt; Memory (11010&lt;sub&gt;two&lt;/sub&gt;)</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>Y</td>
<td>10&lt;sub&gt;two&lt;/sub&gt; Memory (10110&lt;sub&gt;two&lt;/sub&gt;)</td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>N</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

d. After handling a miss of address (10000<sub>two</sub>).

e. After handling a miss of address (00100<sub>two</sub>).  

f. After handling a miss of address (10010<sub>two</sub>).  

The cache contents are shown after each reference request that misses with the index and tag fields show in binary. The cache is initially empty with all valid bits (V entry in cache) turned off (N). The processor requests the following addresses: 10110<sub>two</sub> (miss), 11010<sub>two</sub> (miss), 10110<sub>two</sub> (hit), 11010<sub>two</sub> (hit), 10000<sub>two</sub> (miss), 00100<sub>two</sub> (miss), 10000<sub>two</sub> (hit) and 10010<sub>two</sub> (miss). The figures show the cache contents after each miss in the sequence has been handled. When address 10010<sub>two</sub> (18) is referenced, the entry for address 11010<sub>two</sub> (26) must be replaced and a reference to 11010<sub>two</sub> will cause a subsequent miss. Remember that the tag field will contain only the upper portion of the address. The full address of a word contained in cache block / with tag field / for this cache is 8 x j + I, or equivalently the concatenation of the tag field j and the index I. You can see this by looking at the lock address in the Data field of any cache entry and the corresponding index and tag. For example, in cache f above, index 010 has tag 10 and corresponds to address 10010.
Spatial locality - larger block size

- A cache which brings in a single word on each read of memory, does not exploit spatial locality.
  - If a word at location $i$ is required, there is a high probability that words near $i$ will be required soon.
  - It makes sense to bring in more than a single word.
  - The number of words brought into cache on a single read from memory is called the *block size*. 
A 64 KB cache using four-word (16 byte) blocks. The tag field is 16 bits wide and the index field is 12 bits wide, while a 2-bit field (bits 3-2) are used to index the block and select the word from the block using a 4-to-1 multiplexor. In practice, the low-order bits of the address (bits 2 and 3 in this case) are used to enable only those RAMs that contain the desired word. This eliminates the need for the multiplexor. This technique works because the values of the block offset bits are known at the same time as the rest of the address bits.
Mapping address from memory to cache

Address in main memory =
No. of blocks X Block size X Tag +
Block size X Index +
Bytes offset

Example: What cache block will be used for byte address 1240?
- block size = 16 bytes (4 words)
- cache has 64 blocks

Answer: 1240 / 16 = 77 blocks + 8 bytes = 78th block.
There are 64 blocks in cache so block 77 maps to cache block 13.
### Factors affecting cache performance

- A larger blocksize exploits spatial locality; miss rate goes down and fewer accesses to memory required.
- Beyond a certain size, the blocksize does not contribute to more spatial locality.
- If the blocksize is too large, there will not be room in the cache for enough different blocks; blocks will compete for the few spaces and will constantly push each other out → miss rate goes up.
- Miss rate is not the only factor. A large blocksize means the *miss penalty* is higher, since we have to read more from memory. High performance systems use special memory designs to allow a large block to be read from memory quickly.
**Calculate Cache Size**

Cache specifications -
- $2^n$ blocks of words
- word size = address size = $w$ bits
- blocksize = $b$ words

Total cache bits = $2^n \times [b \times w + (w - \lceil n + \log_2(b \times w/8) \rceil) + 1]$

Example: $w = 32$ bits; blocksize = 1 word; cache = 64KB

$\rightarrow 64KB = 16K$ words = $2^{14}$ words

$\rightarrow 2^{14} \times [32 + (32 - \lceil 14 + 2 \rceil) + 1] = 2^{14} \times 49 = 784 \times 2^{10} = 784$ Kbits

$= 98$ Kbytes, i.e., more than 50% overhead for managing cache.
Yet another example

Example: $w = 32$ bits; blocksize = 16 words; cache = 64KB

$\rightarrow 64\text{KB} = 16\text{K words} = 1\text{K blocks} = 2^{10} \text{ blocks}$

$\rightarrow 2^{10} \times [32 \times 16 + (32 - [10 + 6]) + 1] = 2^{10} \times 529 = 529 \text{Kbits}$

$= 66.125 \text{ Kbytes, i.e., much less overhead for managing cache.}$
Full Associative Mapping to the Cache

- A block from memory may be mapped into any cache block,
- Usually it is mapped to the first block found to be free.
- If no such free block in the cache (all blocks are occupied), a search method is activated to find an appropriate block. When found, the block is taken out of the cache and the new memory block is written over.
- The most common searching method is LRU - Least Recently Used, which means that the block that was least recently used (read from or written to) may be taken out. This heuristic is based on the locality principle.
- Advantages: better usage of the cache by taking out blocks only when they are supposedly unneeded and only when no more room is available. This method is considered more sophisticated.
- Disadvantages: LRU is much more time consuming than a simple direct mapping. Moreover, since a block may sit anywhere in the cache, the tag must contain (almost) all the address bits. All the cache needs to be searched for a block.
Set Associative Mapping to the Cache

- The cache is organized into sets of blocks, each of which is composed of more than one block.
- Mapping is done in two steps:
  - First, a block is *directly* mapped into a unique set, according to the *index* field.
  - Then, the block may be placed into *any* one of the set blocks.
- A set associative cache with sets containing *n* blocks each is called an *n-way* set associative cache.
- This method is considered a combination (and maybe a compromise) between direct mapping and full associative mapping.
- In fact, a 1-way set associative cache is actually a direct mapped cache.
- Similarly, a set associative cache, where the entire cache is only one set, is in fact a full associative cache.
- Increasing the set associativity, usually decreases the miss rate.
- Advantage: enjoys (hopefully) the best from the two worlds.
- Disadvantage: more complex. All the blocks in the set are searched for a match. (but if set is small, can be done in parallel.) (Set containing a block = Block number modulo numbers of sets in the cache.)
Cache Write-through policy

- If word is in cache (write "hit"), write cache.
- What about main memory? Cache will change but memory stays the same? Memory data can become stale.
  - This word may be removed, and later read again from memory ⇒ result is incorrect data (stale).
  - Write-through policy: update memory on every cache write.
  - Problem with write-through: write to memory occurs frequently (every write), hence average memory access time is poor.
  - If blocksize is 1, there is no write miss; The new block overrides the old one.
  - If blocksize > 1, when a write miss occurs. We cannot write just one word from cache to memory. The new block must be read first to the cache. We change the word we need and only then the write can be done.
Write-through Overhead

- Write-through is required for every write.
- Example calculation:
  - Program has 11% Store instructions.
  - Average CPI is 1.2.
  - Write to memory requires 10 cycles.

  → Effective CPI = 1.2 + .11 x 10 = 2.3
  - CPI has almost doubled ⇒ performance is 100% worse!

- Write-through systems use *write buffering* to reduce this problem
  - A buffer is inserted between the processor and memory.
  - The buffer can hold a small number (1-8) of write requests; a write request includes the data and address to be written.
  - After a word has been sent to the buffer, the processor does not wait for the write to complete; it continues with other instructions.
Cache Write-back Policy

- Write-back policy: update memory only when needed.
  - Suppose a block $b_1$ is to be read from memory into cache block $b$. If cache block $b$ contains memory block $b_2$ and $b$ was updated since the last load of $b_2$ into the cache, then write-back block $b$ into memory block $b_2$ and then load memory block $b_1$ into cache block $b$.
  - A possible problem with write-back: if too many cache misses occur, write to memory might occur too frequently (both for read and write occasions), hence average memory access time is poor.
  - Write-back overhead: more complicated - needs to keep a dirty flag for each block.
  - Regardless of blocksize, old block must be written (updated) before new block may override it.

- Write-back is NOT required for every write.
- Write-back is done only when a cache block is to be "taken out" of the cache in favor of another block, either for reading this other block or for writing into it.
- Write-back is mostly used in virtual memory systems, where reading or writing from/to a lower level is costly.
Write-back - write miss

- For blocksize = 1 word:
  - Write the block from the cache to the memory (update memory).
  - Write the new word to the appropriate location in the cache.

- For blocksize > 1 word:
  - Write the block from the cache to the memory (update memory).
  - Read the new block from memory to the cache.
  - Write the new word to the new block in the cache.
Example

Assume the following:

- Instruction cache miss rate: 5%.
- Data cache miss rate: 10%.
- CPI = 4 for no memory stalls (no misses).
- Miss penalty: 12 cycles.

**How much faster** would be a machine with a perfect cache, that never misses?

**Solution**: Given that instruction count and clock cycle are the same in both models, only CPI has to be compared.

Let \( I \) be the instruction count of the program for which statistics (percentage of each instruction type) are known.

- Percentage of **Load/Store** instructions is 33% (22% + 11%).
- Total penalty (in terms of CPI) for **instruction** fetches: \( I \times 0.05 \times 12 = 0.6 \times I \).
- Total penalty (in terms of CPI) for **data** access: \( I \times 0.33 \times 0.10 \times 12 = 0.4 \times I \).
- Total penalty (in terms of CPI): \( 0.6 \times I + 0.4 \times I = 1 \times I \).

**Conclusion**: the total penalty is 1 more cycle for each instruction, so if CPI is 4 for a perfect cache, then for the given cache, CPI is 5.

**Answer**: a machine with a perfect cache would be of 25% faster (5/4).