**Chapter 4: Cache Memory**

Table of Contents

[4.1 Computer Memory System Overview 3](#_Toc48982351)

[Memory Hierarchy 5](#_Toc48982352)

[4.2 Cache Memory Principles 6](#_Toc48982353)

[4.3 Elements of Cache Design 7](#_Toc48982354)

[Cache Addresses 8](#_Toc48982355)

[Cache Size 9](#_Toc48982356)

[Mapping Function 9](#_Toc48982357)

[Direct Mapping 9](#_Toc48982358)

[Associative Mapping 10](#_Toc48982359)

[Set-Associative Mapping 11](#_Toc48982360)

[Memory Address Divisions 12](#_Toc48982361)

[Replacement Algorithms 12](#_Toc48982362)

[Write Policy 13](#_Toc48982363)

[Line Size 15](#_Toc48982364)

[Number of Caches 15](#_Toc48982365)

The computer memory system is used for storage of data. It exhibits the widest range of type, technology, organization, performance and cost of any component of the computer system. No single technology can satisfy all the storage requirements, which results in a typical computer system having a hierarchy of memory subsystems based on different features.

## 4.1 Computer Memory System Overview

Characteristics of Memory Systems

We can classify memory systems based on the following characteristics:

|  |  |
| --- | --- |
| **Location** | Internal (e.g., processor registers, cache, main memory) |
| External (e.g., optical disks, magnetic disks, tapes) |
| **Capacity** | Number of words |
| Number of bytes |
| **Unit of Transfer** | Word |
| Block |
| **Access Method** | Sequential |
| Direct |
| Random |
| Associative |
| **Performance** | Access time |
| Cycle time |
| Transfer rate |
| **Physical Type** | Semiconductor |
| Magnetic |
| Optical |
| Magneto-optical |
| **Physical Characteristics** | Volatile/non-volatile |
| Erasable/non-erasable |
| **Organization** | Memory modules |

Location can refer to internal memory, such as main memory, registers and cache, which can be directly accessed by the processor, or to external memory, such as a disk, tape or pen-drive that are accessible via I/O controllers.

Capacity is usually measured in bits or bytes (1 bytes = 8 bits) or words (groups of bytes) while external memory is usually measured in bytes.

Data can be transferred to and from memory as words, where the word length is equal to the number of electrical lines connected to the memory module, or as blocks, which consists of a group of words.

The method of access can be sequential, where a linear sequence must be followed to reach a location like in memory tapes, direct access, where any location can immediately be accessed like in magnetic disks, random access, where locations are selected randomly like in RAM and registers, or associative access, where a portion of the data itself is used to identify the location where it is stored like in caches.

Performance can be measured based on access time (latency) from the instance of addressing to the instance of availability, memory cycle time, which is the access time plus the time before another access can begin, or transfer rate of data. For the last one, in random access memory, it is equal to 1/cycle time and for non-random access memory it is given by where is the average time to read or write bits, is the average access time and is the transfer rate in bits per second.

Division based on physical types gives us semiconductor memory, magnetic memory, optical memory and magneto-optical memory. We can also divide based on physical characteristics, like whether the memory is volatile (information lost when power is lost) or non-volatile (information retained until deliberately changed, requiring no power) or erasable or not (BIOS is not erasable).

Finally, division based on organization looks at how the memory cell was created, considering IC types and their relative positions.

### Memory Hierarchy

The design constraints of a computer’s memory are based on three questions: how much, how fast and how expensive.

How much is an open ended question. If memory is available, applications will be developed to use it. How fast, depends on whether the memory is able to keep up with the processor. The cost must also have a reasonable relationship with the other components. Of course, these three properties are related. Faster access time means greater cost per bit, greater capacity means smaller cost per bit but slower access time.

Designers would typically like to use larger capacity memory which is also conveniently cheaper, but meeting performance requirements means using faster, more expensive and smaller memory. The way to get past this is to use a hierarchy of different types of memory. As we go down the hierarchy, the cost per bit decreases, capacity increases, access times increases and frequency of access from the processor decreases. The key to the success of this system is the last part, decreasing frequency of access. It makes use of a principle known as the locality of reference. Memory references by the processor for a particular program tend to cluster together, such as when loops or arrays are used. Thus, a large amount of data from that area of the main memory is moved to a smaller, faster memory, so that further accesses to that data, which is likely to be needed soon, is quicker. This is true for multiple levels of memory as well.

## 4.2 Cache Memory Principles

Cache is designed as a mid-point, combining the access time of expensive, fast memory with the large memory size of cheaper, slower memory. The cache contains a copy of a portion of the main memory. When the processor needs some data, the cache is first checked. If the data is available, it is called a cache hit. If not, it is called a cache miss. In the second scenario, a block of data containing the data needed is loaded from the main memory onto the cache, from where the required data is passed to the processor. Due to the phenomenon of locality of reference, there are likely to be future calls to the data in that block, which can then be loaded quickly from the cache.

There are likely to be multiple levels of cache over which the same process is occurring. L1 cache is fast but small, L2 cache is above that and is bigger but slower and so on. Data is loaded onto higher levels of cache and brough onto lower levels as required.

The main memory consists of addressable words, with each word having a unique -bit address. This can be divided into blocks of words each. Thus, . The cache on the other hand, contains blocks, which are called lines. Each line consists of the same words, plus a tag of a few bits and control bits. Notice that this means we can never use all of the space in the cache to store data. The length of the line, not including tag and control bits, is the line size. Since the number of lines is considerably less than the number of memory blocks, only a portion of the blocks can reside in the lines at any time. Each line thus has a tag that helps identify which block is currently stored. The control bits indicate whether or not the data has been modified since the last read operation. If it has, then it needs to be copied onto the main memory. Data is transferred as blocks from the main memory to the cache, and then as words from the cache to the processor.

## 4.3 Elements of Cache Design

When considering how to design cache memory, there are a few things we need to look at:

|  |  |
| --- | --- |
| **Cache Addresses** | Logical |
| Physical |
| **Cache Size** |  |
| **Mapping Function** | Direct |
| Associative |
| Set associative |
| **Replacement Algorithm** | Least recently used (LRU) |
| First in first out (FIFO) |
| Least frequently used (LFU) |
| Random |
| **Write Policy** | Write through |
| Write back |
| **Line Size** |  |
| **Number of Caches** | Single or two level |
| Unified or split |

### 

### Cache Addresses

Many processors support something called virtual memory. Essentially, this allows programs to address memory from a logical point of view, without considering how much memory is physically available. This allows the program to seemingly reside in its entirety in the memory of the system. A memory management unit (MMU) is used to translate the virtual memory addresses into physical memory addresses.

In this scenario, the designer may choose to place the cache between the processor and the MMU. This would allow the cache to use the virtual addresses to store data, which is turn would allow the processor to access the cache directly, without having to go through the MMU. Such a cache is known as a virtual or logical cache, as opposed to a physical cache which stores data using physical addresses form the main memory.

Virtual cache is faster than physical cache since it does not need to wait for the MMU to translate the addresses. The problem however, is that each application is given the same virtual memory space, i.e. addresses that start at . Thus, the same virtual address in two different applications refer to two different physical addresses. This is solved by either flushing the cache memory with each application context switch, or by adding extra bits to each line of the cache to identify which virtual address it refers to.

### Cache Size

Cache size has already been discussed. We want the cache to be small enough so that the cost per bit is low, but large enough so that the average access time is low. Smaller caches are preferred for other reasons too. Larger caches have more gates, which tends to make them slightly slower. The availability of chip and board area also limits cache size. Since the performance of cache is sensitive to the nature of the work being done, it is difficult to choose a single optimum cache size.

### Mapping Function

Since there are fewer cache lines than memory blocks, we need to have an algorithm to map main memory blocks into cache lines, and also to identify which main memory block is currently held in a cache line. The mapping function dictates how the cache is organized. There are three techniques available, direct mapping, associative mapping and set-associative mapping.

Direct Mapping ([Mandatory Video of Indian Dude Explaining Everything](https://www.youtube.com/watch?v=pSarQQTJbDA))

This is the simplest mapping method, where each block from the main memory is mapped onto a single possible cache line. It follows the following expression

where is the cache line number, is the memory block number and is the number of lines in the cache. Essentially, the main memory is divided into sections, with each section having the size of the cache. Thus, the first block from each section can only be put into the first line of the cache, the second block from each section into the second line and so on. If the main memory has 8 blocks number from to and the cache has 4 lines numbered from to , and can only be put in , and can only be put in and so on. The bits on the right side of the main memory block address that match with the addresses of the cache are used to identify which line to put that particular block in, and the bits on the left side of the block address act as a tag to identify which block ( or ) is currently in that cache location. Note that each block also has its data (the words) with it, which was ignored while considering which line to put the block in. When the processor makes requests, the requests include the tag, block address and the word address. Once the correct block is found, the word address is used to identify the word inside the block and return it to the processor.

Direct mapping is relatively simple and inexpensive. The only problem occurs when a program happens to reference words repeatedly that swap into the same cache lines. This causes a low hit ratio since the blocks must be repeatedly swapped, a phenomenon known as ‘thrashing’.

A way to reduce thrashing is to remember the data which was just swapped out of the cache. This is done by storing the data in another cache, known as the victim cache. This allows the data to be read more quickly than if it had had to be loaded from the main memory again.

Associative Mapping ([Video](https://www.youtube.com/watch?v=Pg6bqkekAXY))

Here, a block from main memory can be loaded onto any line of the cache. The cache control logic interprets the memory address as just a tag for a block and the addresses for the words, with no further division and with the tag uniquely identifying a block in main memory. To determine if a block is present in the cache, the control logic must simultaneously examine every line’s tag. Associative mapping also gives us flexibility with which line to replace when new blocks must be loaded. These are discussed a little later, under Replacement Algorithms. The main disadvantage of associate mapping is that complex circuitry is need to parallelly examine the tags of all the lines in the cache.

Set-Associative Mapping([Video](https://www.youtube.com/watch?v=isbdr73Ymug))

This is a compromise between direct mapping and associative mapping. A select number of lines () from the cache are grouped into a set. Blocks from main memory are connected to sets using direct mapping, and inside the sets, associative mapping is used. For example, say we have a 128-byte main memory, with blocks of 8 bytes. Thus, the main memory block addresses are from 0000 to 1111. Let the cache size be 32 bytes, which gives us 4 lines of 8 bytes each, with addresses from 00 to 11. If we use 2-way Set Associative Mapping, then each set has 2 lines (). Since we have only 2 sets, we can use just 1-bit to identify a set, 0 or 1. From the main memory, every block that has a least significant bit of 0, i.e. 0000, 0010, 0100, etc. all go to the first set, while every block that has a least significant bit of 1 go the second set. Now, say we load the data from memory addresses 0001 and 0011 to the cache. Both of these will go to set 2, since their least significant bit is 1. Inside the cache, their most significant bits are used as tags, i.e. 000 and 001 respectively. When the processor makes a request, the request as usual, consists of a block address and a word address. In the cache, the least significant bit of the block address is used to identify which set to check in, and the most significant bits of the block address are used to parallelly check the blocks present in that set. Since the size of a set is small, parallelly checking the blocks takes much less work than it did in Associative Mapping.

### Memory Address Divisions

Take a main memory of 128 bytes with a block size of 8 bytes. Thus, we have 128/8 = 16 blocks. To represent 16 addresses, we will require 4 bits, from 0000 to 1111. Say we have a cache of 32 bytes. Since 32/8 = 4, we have 4 lines in the cache, with addresses from 00 to 11. Since a single block is of 8 bytes, we require 3 bits to identify a word inside a block (23 = 8). The processor will send requests in the format 0100110. Here, the least significant 3 bits identify a word, and the most significant 4 bits are a block address. The block address here is 0100.

Using direct mapping, we know that the block address is divided into a cache line address and a tag. The cache line address here would be 00 and the tag would be 01. The computer goes to cache line 00, verifies that the tag is 01, goes inside that cache line and retrieves the word at the address 110.

For associative mapping, the entire block address is the tag. For set associative mapping, with a 2-way set, we know that the set address is identified using 1 bit. Thus, the set address would be 0, and the tag would be 010.

### Replacement Algorithms

When the cache is filled, some block must be replaced when a new block is brought it. With Direct Mapping, there is no choice, but with Associative and Set-Associative Mapping, an algorithm is needed to determine which block to replace. The algorithm is implemented in hardware to achieve high speeds. There are four replacement algorithms:

* Least Recently Used (LRU): The least recently used block is replaced. This is very easy to implement in 2-Way Set Associative Mapping since there are only 2 blocks in a set. When a block in a set is used, it sets a new, additional bit called the USE bit to 1, and sets the USE bit of the other block to 0. When a block needs to be replaced, the one with a USE bit of 0 is replaced. For fully associative mapping, a separate list of indexes is used. When a block is used, it goes to the front of the list. When replacement must occur, the block at the back of the list is replaced. LRU should give the best hit ratio. This, along with its ease of implementation, makes it the most popular replacement algorithm.
* First in First Out (FIFO): The block that has been in the cache longest is replaced first. This use a circular buffer.
* Least Frequently Used (LFU): The block that is referenced the least is replaced. This could be implemented with the help of a counter.
* Random: A line is picked at random and replaced. Simulation studies show that this method is only slightly inferior to the others.

### Write Policy

When a block in the cache is about to be replaced, we need to check if it has been edited. If the block has not been written onto since it was first read, then it can simply be overwritten. However, if it has been written onto, then then main memory must be updated by writing the line of cache out to the block of memory before overwriting it. There are a variety of write policies with different performance and economic trade-offs.

There are primarily two problems. First, more than one device may have access to the main memory, such as I/O modules. If a word is altered in the cache, the corresponding memory word is invalid and vice versa. Second, if there are multiple processors and a word is altered in the local cache of one processor, that word is invalid in all other processors.

The simplest way to solve these problems called write-through. Any write operations are made directly to the main memory as well as to the cache. Other processors-cache modules monitor traffic to main memory to maintain consistency. The main disadvantage here is that it generates substantial memory traffic and may create a bottleneck.

An alternative technique is called write back, where any alternation to a line in cache causes a dirty bit or use bit to be set. When a block is replaced, the previous data is written onto main memory only if the use bit is set. The problem here is that portions of the main memory may be invalid, and hence access by I/O modules can only be allowed through the cache. This causes a more complex circuitry and potential bottlenecks.

Neither of these techniques solves the problem of the same data being in multiple caches in a bus organization. All the caches have invalid data if even one cache data is changed. A system that prevents this problem is said to maintain cache coherency. Possible approaches include:

* Bus watching with write through: Each cache controller monitors the bus to check whether their shared data has been altered. If it has, it invalidates its own data.
* Hardware transparency: Additional hardware is used to update all caches when main memory is updated.
* Noncacheable memory: A portion of the main memory is chosen and cache is not allowed to fetch data from there. Data is always read or written directly from that portion. Typically, vulnerable data is stored here to avoid inconsistency.

### Line Size

We must also consider the size of a line when designing cache memory. When a block of data is retrieved from the main memory, some words alongside the desired word are also retrieved. This is the line size. As we increase the amount of data being retrieved (and thus the line size), the hit ratio will at first increase due to the principle of locality. Data nearby the data requested is also likely to be needed soon. However, as line size keeps decreasing, we will start to retrieve data that is further away and is less likely to be needed, which will in turn decrease the hit ratio. Two forces come into play. Firstly, larger blocks mean fewer blocks can be stored in the cache, which means blocks will be replaced shortly after being fetched. Secondly, as blocks become larger, each additional word in the block is less likely to be needed in the future. No optimum line size has ever definitively been found.

### Number of Caches

The use of caches originally began with a single one, but recently multiple caches are being used. We now have multilevel caches. As logic density has increased, it has even become possible to have a cache directly on the processor chip. This reduces the need to retrieve data via an eternal bus, and also reduced the distance from the requested data, both of which increase execution speeds. This also frees the bus to support other transfers. This brings up the question of whether we need off-chip caches at all. The answer, mostly, seems to be yes. In the case where data is not found on the on-chip cache, the data would otherwise have to be searched for on the main memory, which would take significantly more time. Instead, we use a secondary level of cache, the L2 cache, which stores more data.

We also have a choice over unified versus split caches. Unified caches store both data and instructions, while split caches are two separate caches, one for data and one for instructions. The split caches exist on the same level, with instructions being searched for in one and data being used from another.