Memory Hierarchy and Caches

Madiba Hudson-Quansah

Contents

Chapter 1

Memory Hierarchy

1.1 Memory Hierarchy

The memory hierarchy is an organization of different types of memory in a computer system, based on their speed, size and cost. Starting from the top of the hierarchy, the fastest and most expensive memory is at the top, while the slowest and cheapest memory is at the bottom. The hierarchy is as follows:

Registers - Fast and small memory close to the CPU. Used to store data that is currently being processed.

Cache Memory - Fast memory that stores frequently accessed data. It is smaller than main memory and is used to speed up access to data. Usually computers have 3 levels of cache each with increasing size and latency as you go up in levels, i.e. L1, L2 and L3 cache.

Main Memory - Also known as RAM, it is the primary storage for data and programs that are currently in use. It is slower than cache memory but has a larger capacity.

Secondary Storage - This is non-volatile storage that retains data even when the power is off. It is slower than main memory but has a much larger capacity. Examples include hard drives, SSDs and USB drives.

Tertiary Storage - This is used for long-term storage of data that is not frequently accessed. It is the slowest and cheapest type of storage. Examples include magnetic tapes and optical disks.

1.2 The Structure of Random Access Memory (RAM)

There are two types of RAM:

Static RAM (SRAM) - Requires 6 transistors per bit of memory. It needs low power to retain state, but it is expensive and has a low density. It is used for cache memory.

Dynamic RAM (DRAM) - Requires 1 transistor and 1 capacitor per bit of memory. It needs high power to retain state, but it is cheap and has a high density. It is used for main memory. Must be re-written after being read and must be refreshed periodically by reading and rewriting all the rows in the DRAM.

DRAM supports two types of access:

Row Access - The entire row is accessed and the data is read from the row. The row is then written back to the DRAM.

Column Access - The entire column is accessed and the data is read from the column. The column is then written back to the DRAM.

There are two types of DRAM:

Synchronous DRAM (SDRAM) - A clock is added to the DRAM interface to synchronize operations with the system clock. It is faster than asynchronous DRAM as with improvements in technology the system bus clock improved, increasing the rate of SDRAM operations.

Double Data Rate Synchronous DRAM (DDR) - Is also synchronous but it performs operations on both the rising and falling edges of the clock. It is faster than SDRAM as it doubles the data rate.

1.3 Memory Latency vs Bandwidth

Definition 1.3.1: Memory Latency

The time difference between the time the request for an address is made and the time the data from that address is received. The total latency to a new row/column is the time between opening a new row of memory and accessing the column within that row. Measured in nanoseconds (ns).

Definition 1.3.2: Memory Bandwidth

The rate at which data is transferred between the memory and the CPU. Measured in millions of bytes per second (MB/s).

Chapter 2

Cache Memory

2.1 The Importance of Cache Memory

- The widening gap between CPU and memory latency is a major bottleneck in computer performance.
- Each instruction typically involves at least one memory access making memory access the weakest link in the instruction execution cycle.
- Memory bandwidth limits the instruction execution rate especially for applications that require large amounts of data to be processed.
- Cache memory is used to bridge the gap between CPU and memory latency by storing frequently accessed data in a small, fast memory close to the CPU.
- Cache memory is faster than main memory and is used to speed up access to data.

2.1.1 Typical Memory Hierarchy

Note:-

KiB - 1024 bytes

MiB - 1024 KiB

GiB - 1024 MiB

TiB - 1024 GiB

PiB - 1024 TiB

KB - 1000 bytes

MB - 1000 KB

GB - 1000 MB

TB - 1000 GB

PB - 1000 TB

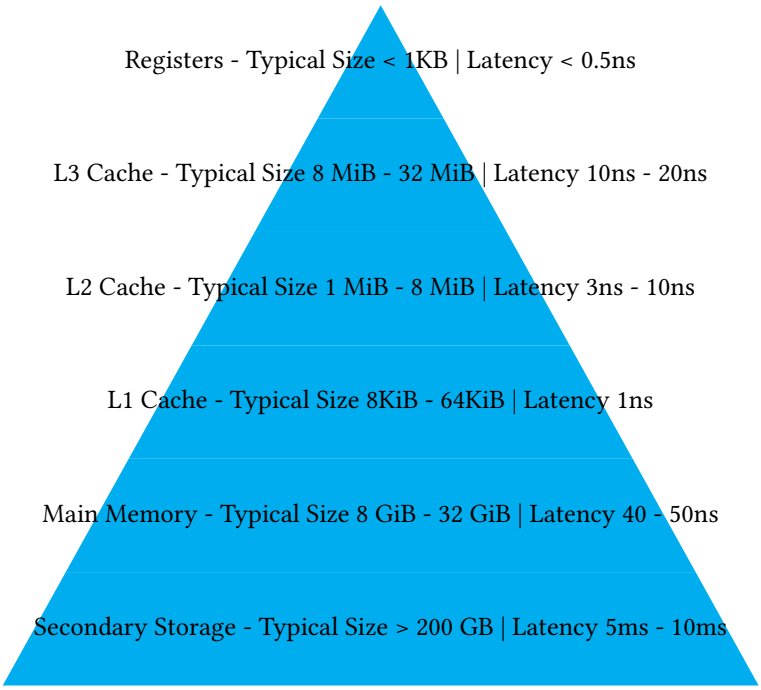

Figure 2.1: Memory Hierarchy

2.2 Locality of Reference

Definition 2.2.1: Locality of Reference

The tendency of a program to access a relatively small portion of its address space at any given time. This means that if a program accesses a particular memory location, it is likely to access nearby memory locations in the near future.

2.2.1 Temporal Locality (Time)

Definition 2.2.2: Temporal Locality

The tendency of a program to access the same memory location multiple times within a short period of time. This means that if a program accesses a particular memory location, it is likely to access that same memory location again in the near future.

2.2.2 Spatial Locality (...and Space)

Definition 2.2.3: Spatial Locality

The tendency of a program to access memory locations that are close to each other. This means that if a program accesses a particular memory location, it is likely to access nearby memory locations in the near future.

2.3 Cache Memory

Cache memory exploits both temporal and spatial locality by keeping frequently accessed data close to the CPU, and storing blocks consisting of multiple bytes of contiguous memory locations respectively.

Data is primary placed in a cache using block placement policies. The most common block placement policies are:

- Direct Mapped Cache
- Set Associative Cache
- Fully Associative Cache

Data is then retrieved from a cache using block identification, where each block is identified via a block tag, address and an index. When a cache miss occurs data is replaced in the cache using a block replacement policy. The most common block replacement policies are:

- Least Recently Used (LRU)
- First In First Out (FIFO)
- Random Replacement

2.3.1 Block Placement Policies

Definition 2.3.1: Cache Block / Cache Line

Unit of data transfer between main memory and a cache. The larger the block size the less tag overhead and the more data that can be transferred at once. However, larger block sizes can lead to more cache misses as the data may not be used immediately. Typically 64 bytes in size

Definition 2.3.2: Block

Unit of data transfer between a cache and memory.

2.3.1.1 Direct Mapped Cache

Definition 2.3.3: Direct Mapped Cache

Each block can only be placed in exactly one location in the cache. Using this policy memory addresses are divided into block addresses and block offsets, identifying the block in memory and the offset to access specific bytes within that block respectively. A block address is then further subdivided into an index and a tag used for direct cache access and the most significant bits of the address respectively, i.e. the index is calculated as

$$\text{Index} = \text{Block Address} \mod \text{Number of Cache Lines}$$

Definition 2.3.4: Cache Hit

A cache hit occurs when the block is found in the cache. The index is used to access the cache block and address tag is compared against the stored tag in the cache. If the tags match, the data is returned from the cache. If the tags do not match, a cache miss occurs and the data is fetched from main memory.

If the number of cache blocks is 2^n then n bits are used for the cache index, if the number of bytes in a block is 2^b then b bits are used for the offset, and depending on the size of the address the remaining bits are used for the tag, i.e. if 32 bit addresses then Tag $= 32 - n - b$. Generally:

$$\text{Tag bits} = \text{Address bits} - \text{Index bits} - \text{Offset bits}$$

The cache data size is therefore 2^{n+b} bytes

For example if a direct mapped cache has 256 blocks, and each block is 16 bytes, assuming 32 bit addresses:

$$\begin{aligned}\text{Index} &= 2^n = 256 \\ &= n = 8 \\ \text{Offset} &= 2^b = 16 \\ &= n = 4 \\ \text{Tag} &= 32 - 8 - 4 \\ &= 20\end{aligned}$$

Therefore the division is: Then computing the tag, index and offset for the address $0x01FFF8AC$ we get:

Tag	Index	Offset
20 bits	8 bits	4 bits

$$\begin{aligned}\text{Tag} &= 0x01FFF \\ \text{Index} &= 0x8A \\ \text{Offset} &= 0xC\end{aligned}$$

2.4 Write Policies

Definition 2.4.1: Write Through

Data is written to both the cache and main memory at the same time. This ensures that the data in main memory is always up to date, but it can slow down the system as both writes must be completed before the CPU can continue processing. Memory always has the latest data which simplifies data coherency. Only a valid bit is required to indicate if the data is valid in the cache.

Definition 2.4.2: Write Back

Data is written to the cache only and is marked as dirty. The data is then written to main memory only when the block is replaced in the cache. This can speed up the system as only one write is required, but it can lead to data inconsistencies if the data in main memory is not updated. Memory may not have the latest data which complicates data coherency. Uses less memory bandwidth as only one write is required. Both valid and modified bits are required to indicate if the data is valid and modified in the cache respectively.

Definition 2.4.3: Modified Bit

A bit used to indicate if the data in the cache has been modified. If the modified bit is set, the data in the cache is different from the data in main memory and must be written back to main memory before the block is replaced.

2.5 Replacement Policies

Direct Mapped caches have no selection alternatives for replacement as each block can only be placed in one location in the cache. However, set associative and fully associative caches have multiple locations for each block and therefore have to select which block to replace when a cache miss occurs. The most common replacement policies are:

Definition 2.5.1: Random Replacement

Candidate blocks are randomly selected for replacement. One counter for all sets (0 to $m - 1$), where m is the number of sets. The counter is incremented for each access and the block with the same index as the counter is replaced. This policy is simple to implement but can lead to poor performance as it does not take into account the usage patterns of the blocks. On a cache miss the block replaced is specified by the counter.

Definition 2.5.2: First In First Out (FIFO)

The oldest block in the cache is replaced. This policy is simple to implement but can lead to poor performance as it does not take into account the usage patterns of the blocks. On a cache miss the block replaced is specified by the index of the block in the cache.

Definition 2.5.3: Least Recently Used (LRU)

The block that has been unused for the longest time is replaced. This policy is more complex to implement as it requires keeping track of the usage patterns of the blocks. On a cache miss the block replaced is specified by the index of the block in the cache. The LRU policy can be implemented using a stack or a counter for each block. With m blocks per set there are $m!$ possible permutations of the blocks. Pure LRU is to costly to implement when $m > 2$, therefore an approximate method is used.