medium.com

Taxonomy of Cache Misses - The High Performance Computing Forum - Medium

Gunavaran Brihadiswaran

5-6 minutes

A detailed illustration of different types of cache misses

Photo by Scott Graham on Unsplash

Introduction to Cache Memory

Majority of the modern-day computers have three levels of cache memories namely L1, L2, and L3 caches. Often, L1 and L2 caches reside on-chip while L3 cache resides off-chip. These cache memories play an important role in creating the illusion of having a fast main memory. But, why do they have to create such an illusion?

There has always been a significant gap between the performance of the CPU and the main memory, and as a result, the main memory has become a performance bottleneck. The rate at which CPU processes data is much higher than the rate at which the main memory can provide data. Accessing the main memory is approximately 100 times slower than accessing the L1 cache.

When the CPU needs to access some data, first it would check in the caches. If the data is present, we call it a **cache hit** which would result in faster data access. If the required data is not present in any level of cache, we call it a **cache miss**. Then the data should be fetched from the main memory which is relatively very slow. The cache misses have been categorized into 4 types namely:

- Cold miss (a.k.a. compulsory miss)
- Capacity miss
- Conflict miss
- Coherence miss (true sharing miss and false sharing miss)

1. Cold miss / Compulsory miss

A **cold or compulsory miss** occurs when a piece of data is being accessed for the first time.

Suppose the cache line size is 32B, the size of the cache is 128B (just for the sake of illustration) and we have an array of size 32 with *double* values stored in the main memory (*double array[32]*).

Initially, the cache is empty and the array is stored in the main memory as illustrated in the following diagram.

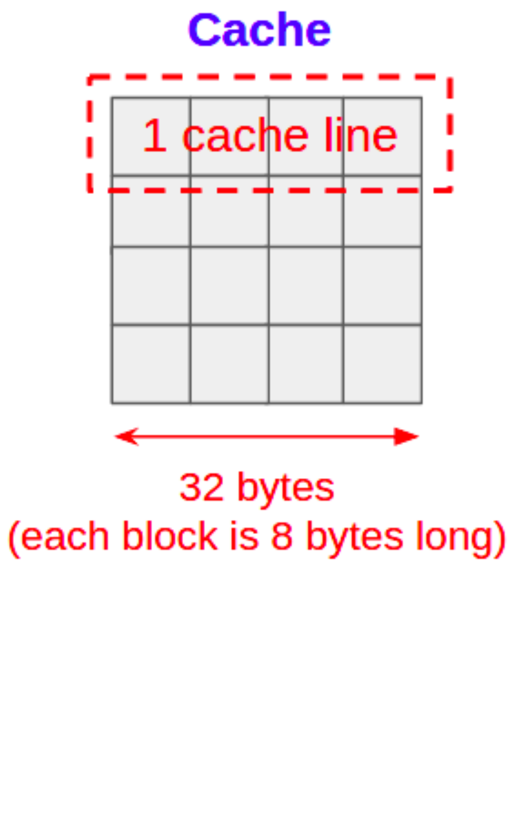
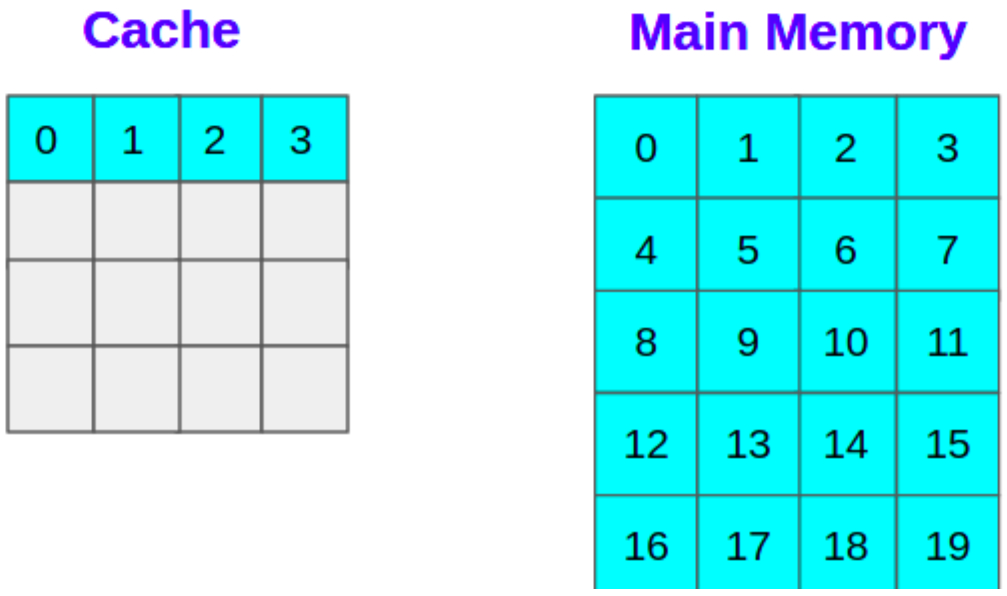


Image by Author

When we try to access *array[0]*, it is not available in the cache since it is the first time *array[0]* is being accessed which would result in a **cold/compulsory miss** and we fetch a cache line worth of data from the main memory to the cache.



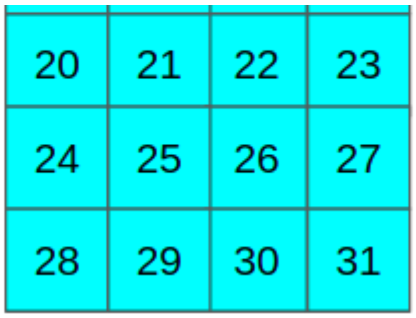


Image by Author

We can reduce **compulsory misses** through prefetching. In the above example, when *array[0]* is being accessed, *array[1]*, *array[2]*, and *array[3]* are prefetched. Later when we access *array[0–3]* we achieve **cache hits**.

2. Capacity Miss

A **capacity miss** occurs when previously fetched data has been removed from the cache because of space limitation.

Let's assume that we have fetched data up to *array[15]*. Now the state of the cache is as follows.

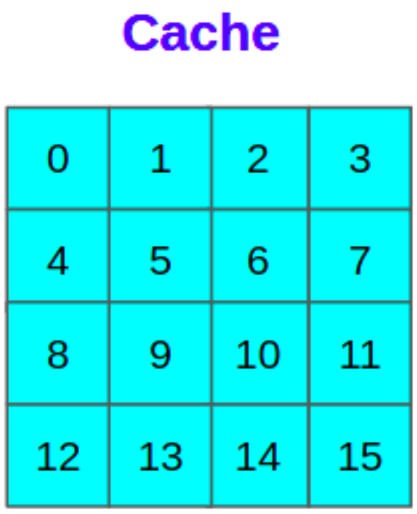


Image by Author

Suppose we want to access *array[16]* next, there is no free space in the cache. If the system follows the LRU (Least Recently Used) protocol, the first cache line will be removed and the new data will be stored.

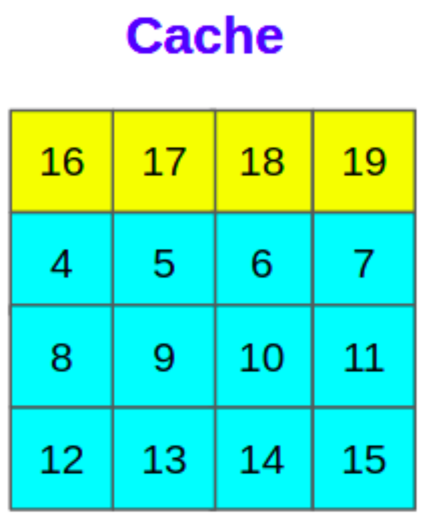


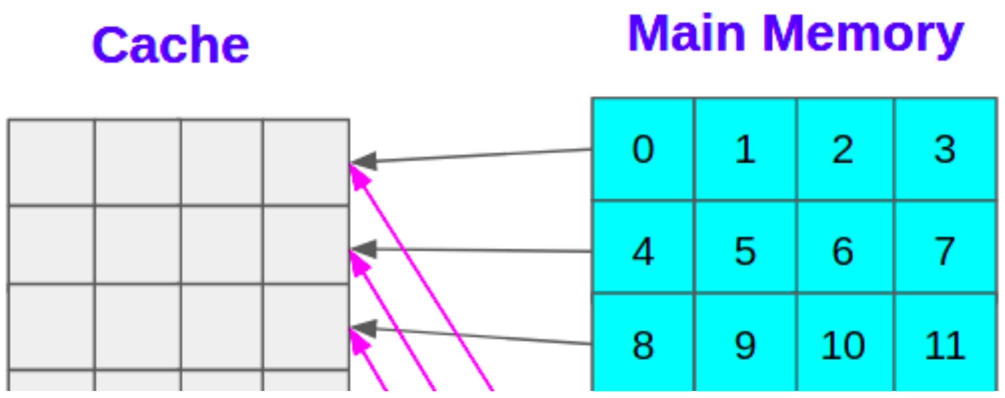
Image by Author

If we want to access *array[0]* again, we end up in a **capacity miss**. Even though we have accessed *array[0]* earlier, it is not there in the cache since it has been evicted from the cache due to the lack of capacity of the cache.

3. Conflict Miss

Conflict misses occur in direct-mapped or set-associative caches. A **conflict miss** occurs when previously accessed data gets removed from the cache even though there is free space left in the cache.

Assuming a direct-mapped cache, the following figure demonstrates how the memory addresses are mapped to the cache.



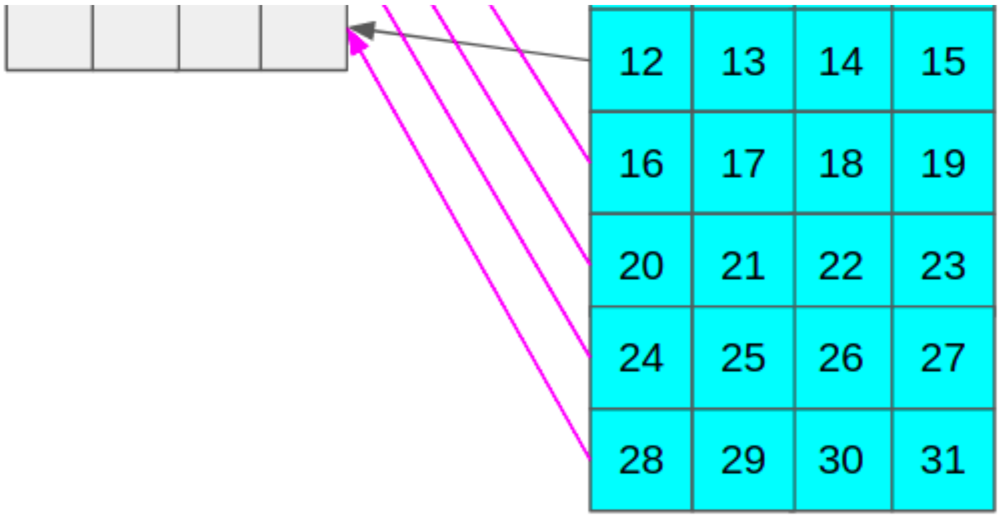


Image by Author

What would happen if we access the array elements in the order *array[0] → array[16] → array[0]*?

When *array[0*] is accessed, the cache would look like:

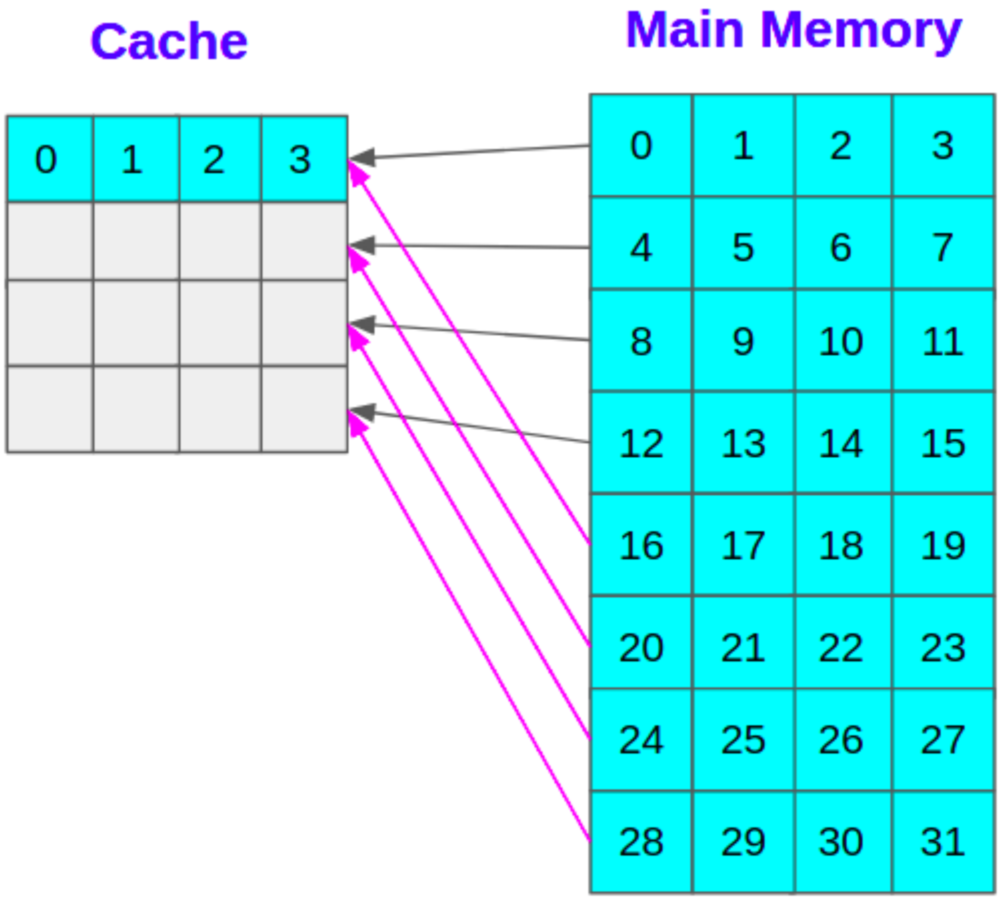


Image by Author

Then, when we try to access *array[16]*, that particular memory address is mapped to the same cache block as *array[0]*. Hence, the previous cache line has to be evicted to make room for the new data.

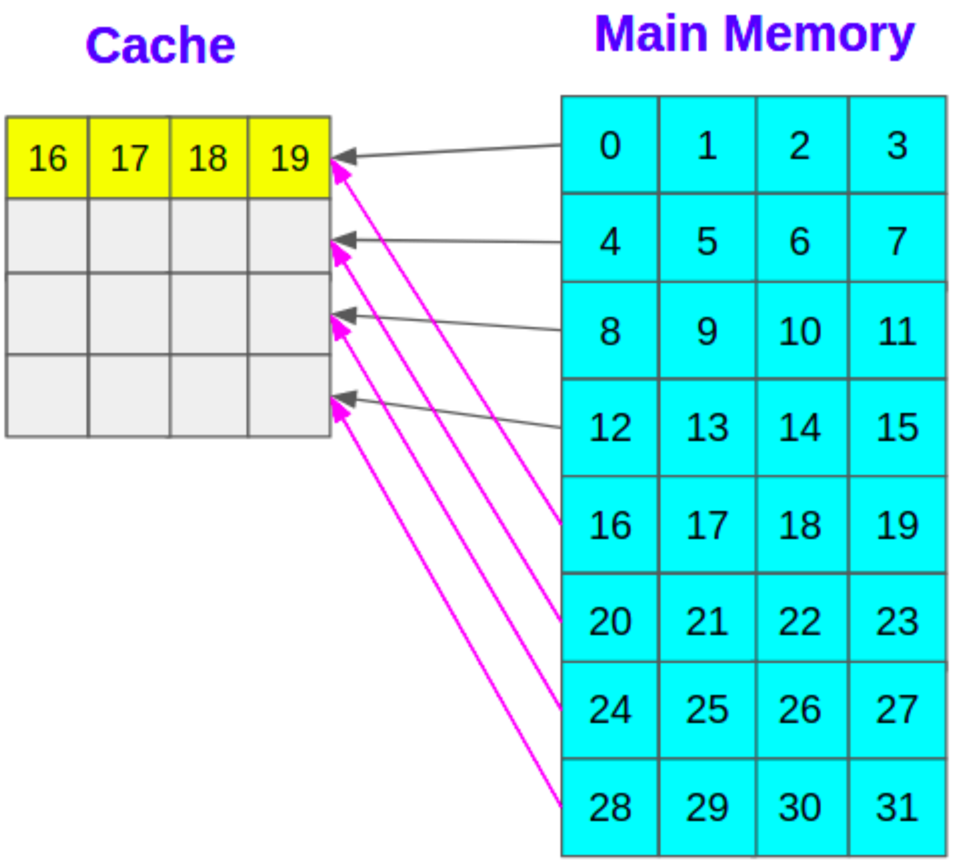


Image by Author

Now when we want to access *array[0]* again, it is not there in the cache which results in a **conflict miss** (because previously accessed *array[0]* has been removed from the cache even though cache is not full).

We can eliminate **conflict misses** by using full-associativity. Full-associativity allows any memory location to be stored anywhere in the cache and as a result, there would never be a conflict.

4. Coherence Miss

Coherence misses are divided into two categories: (1) True sharing miss and (2) False sharing miss. Coherence miss often occurs in a parallel environment.

4.1 True Sharing Miss

Consider a scenario where the system has two processors and both of them require access to *array[0]* one after the other. Processor_1 accesses *array[0]* first. Then Processor_2 accesses *array[0]* and modifies its value. This would invalidate the corresponding cache line in Processor_1. When Processor_1 tries to access *array[0]* again, it would result in a **true sharing miss** since the corresponding cache line has been invalidated**.**

4.2 False Sharing Miss

What would happen if Processor_2 requires access to *array[1]* instead?

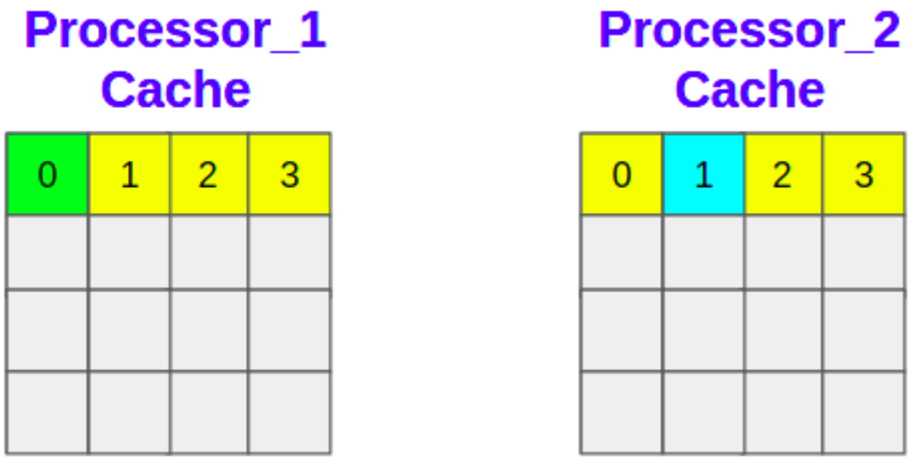


Image by Author

Even though both processes are working with different data, when Processor_2 modifies *array[1]*, the cache line in Processor_1 will be invalidated since *array[0]* and *array[1]* reside in the same cache line. We call it a **false sharing miss** because the data is not actually shared but unfortunately they ended up in the same cache line.

Since cache misses are expensive, arranging data in such a manner to reduce cache misses would provide a performance boost.