CS230 Notes

K.S.Shriram Kumar

May 6, 2024

Part I
Caches

1 Introduction

What is a cache? A cache is can be thought of as a small portion of memory which can be accessed in small amount of time. It is used as an optimisation to work around the limitations of the DRAM's extremely slow speed. It takes a few hundred cycles to access the DRAM, whereas the cache can be accessed in 10's of cycles at most [1].

2 How is it useful?

A cache improves performance by exploiting the spatial and temproal locality of memory accesses. In simple words, when memory is accessed at time 't' it is likely that it will be accesses again in time [t-δ, t + δ]. This is temproal locality. Spatial locality of memory accesses refers to the fact that a memory access to address X means that there is a high probability of an access to memory at addresses [X - Δ,X + Δ]. These observations are based around emprical observations.

To understand them lets take an example. Assume we have an array of size 1000. Odds are that access to the array happens sequentially and this would correspond to spatial locality of memory accesses since array memory addresses are contiguos. Similary, we would repeatedly access the address of a variable `i` within a counter.

Thus if we store spatially, temporally local addresses's corresponding data, they can be directly accessed from cache. This would thus save a lot of processing power and thus improve performance.

[1] Not exact numbers, just to give an idea

2.1 Latency vs Bandwidth

Bandwidth problems are easy to solve by throwing money at the problem. Basically we can always just build more pipelines to pump data through. However Latency problems are not so easy to solve since we can't improve the speed of operations without fundamentally changing the bottle neck component. Cache helps us improve on the latency side which makes it that much more important.

It is important to note that the size of the cache is so much smaller than memory due to the fact that a smaller memory is faster. If we increase cache size, the latency associated with it increases and any benefit from using the cache is lost.

There are also multiple caches of progressively bigger sizes and latencies which finally connect to the DRAM itself. Usually, there are three levels of cache. Any more levels our latency falls so much that a cache access is slower than a DRAM access which makes the cache completely useless.

3 Structure of a cache

3.1 Some terms

- **Line:** A line is a sequence of bytes stored at continuous addresses. A line is of some fixed size (usually 64 bytes).
- **Set:** A set is a group of more than 1 lines.
- **Mapping:** A Mapping refers to the set of lines which data at a particular address can possible reside in inside a cache.
- **Cache hit(or Miss):** When a memory at an address is requested by the CPU, the cache is checked to see if it has the data corresponding to this address. If the requested data is found in the cache it is refered to as a cache hit. Now the data can be directly fetched from the cache. Else it is called a cache miss.

Note that during a Cache miss the CPU accesses the desired data from the DRAM. When this is done the CPU also places this data in the cache. This exploits temporal locality. We know that CPU asked for this data now and thus is likely to ask it again in the near future.

3.2 Organization of a cache

A cache is divided into line sized units and whenever data is fetched it is fetched at the granularity of a line. This exploits spatial locatlity as when a byte is fetched into cache so are bytes nearby it. The lines in a cache are indexed from 0. Each line of the cache stores some number of bytes according to the line size.

When a cache miss occurs, we obtain new data[2] from the DRAM to be placed in the cache. However we need to know which line is to used to hold this data. This is the cache mapping.

[2]The entire line in which the desired byte belong

3.3 Types of Mapping

The Structure of the cache depends on the type of mapping used in a cache.

3.3.1 Direct mapped cache:

In this cache each line in the main memory is mapped to one and only one line in the cache. However since the number of lines in the cache is much lesser than number of lines in main memory, multiple main memory lines map to the same cache line. The Mapping is usually done as
Line number mapped to in cache = Line number % Number lines in cache

Take an arbitrary address X(32-bit). The last $\log_2$ (line size in bytes) which is 5 gives the offset of the corresponding byte within a line. This is called well the *line offset*. The next
$\log_2$ (number of lines in cache) gives the line in the cache this address is mapped to. These bits are called the byte offset. The remaining significant bits are called the 'tag' of the address.

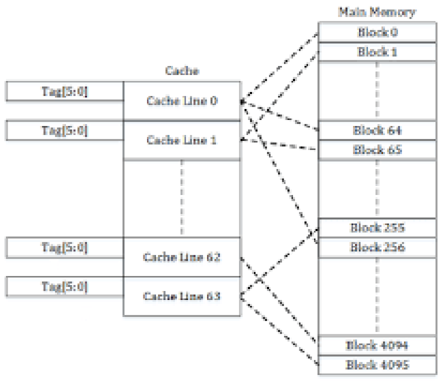

Figure 1: Direct mapped cache

It is clear to see that the tag for a given line in main memory is same. A different tag would mean the addresses are atleast as far apart as the line size and thus can't be in the same line. Thus the tag can identify the line from main memory which is stored in the cache. Since in direct mapped cache a set of few lines, all of which are mapped to the same line could be present in that line. Thus we need to compare the tag bits to identify if the line we desire is present in the cache.

3.3.2 Fully associative cache:

The structure of the address remains the same as before. However unlike in a direct mapped cache, here a line from the main memory can be stored in any line in the cache. Thus we need to compare the tag bits to identify if the line we desire is present in the cache. In a direct mapped cache this needs to only be checked in the corresponding line obtained from the *line offset*. Here However there is no line offset as the line from main memory can be anywhere in the cache. The line offset bits are now included in the tag and to check the presence of a line the tags of all cache lines need to be checked.

This is obviously infeasible as it would require us to have a comparator corresponding to each line of the cache unlike the one comparator needed for direct mapped cache[3].

3.3.3 Set associative cache:

Both fully associative and directly mapped caches are two different extremes of this method. In fully associative caches each line in main memory is mapped to all the lines in the cache. In direct mapping cache, a line in main memory is mapped only to one line in the cache.

In general in a set associative cache each line in the main memory is mapped to one or more lines in the cache. A set is a group of lines to which a line in main memory can be mapped to. In a fully associative cache there is a single set whose size is the same as number of lines in a cache. In a directly mapped cache, there is a set corresponding to each line in the cache and thus number of sets is the number of lines in the cache.

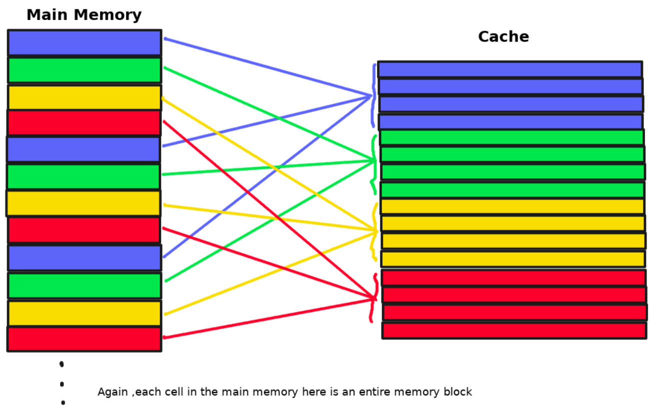
Main Memory
Cache
Again ,each cell in the main memory here is an entire memory block

Figure 2: Set associative cache

Instead of a line offset as in directly mapped caches we have a set offset. A line in main memory having a set offset can be mapped to any of the lines within that set. As can be seen in *Figure* 2. All the blue lines in main memory can be put into any of the lines in the cache.

To figure out if a given address is in our cache we first check if we have a tag hit. The tags of the lines present in the lines corresponding to the set offset of the given address are to be checked. That is in *Figure* 2 if our address maps into the red region, we check all the tags of the cache lines in the red region.

[3]Since only one comparison needed to check if line is in cache

4 Miss me yet?

So what happens during a cache miss. First let's understand what types of misses exist.

1. *Cold Misses:* The first reference to a line in the program definitely constitutes a miss. This is regardless of the how good, what type of cache it is etc.... These are also called compulsory misses.
2. *Conflict miss:* If a cache is ever smaller than main memory (ie) always, we have multiple lines in main memory mapped to same line in cache. Assume we have accessed a line and it was put in cache. It was kicked out by another line sharing same cache line. When the initial line is accessed again we have a conflict miss.
3. *Cache size miss:* Cache is too small to hold this address till we need it again.

It is clear that when we access a memory location who doesn't have an entry, we need to populate the cache with this address. What if the cache is filled with other entries in the possible lines to put in this data? In that case we evict one of the entries and replace it with our data. How to pick the line to replace?

We use cache replacement polices for this.

4.1 Cache replace policies

What would be the optimal policy? Kicking out the line which is not used for the near future is a good choice. The policy is to kick out the line which is going to be accessed at the latest point in the future among all candiates to be kicked out. This is called Belady's policy. This obviously cannot be used since we have no way to predict the future demands of a sequence of instructions.

The key features of an replacement policy is:

- **Insertion:** What to do when a new entry is inserted?
- **Promotion:** What happens to prioties of all entries, which are more likely to be kicked out and how to update this parameter?
- **Eviction:** Which entry to kick out?

An obvious choice would be FIFO(first in first out), but this is bad since an address being used first doesn't imply it is not going to be reused.

The ideal policy is LRU which is relatively easy to implement. At a point we evict the process which was accessed the least recently.