CS2610: Computer Organization and Architecture Lab

# Assignment-4 (Reverse engineering of CPU)

## Objective: To identify the cache block size and the associativity of the L1 cache

Bhukya Bhavadeep (CS20B015)

Vijaya Sasibind Badde (CS20B087)

### Idea:

We know for a fact that a cache hit takes lesser time than a cache miss. We will use this piece of useful information to proceed.

One way we can do this is to constantly attempt to hit the L1 cache and time every attempt. Every time we encounter a cache miss, since the time taken is noticeably higher, we will note that iteration, and continue hitting the L1 cache.

The second time we encounter a cache miss, we take note of the number of hits it took from the first miss. A cache miss happens when the cache block we’re hitting in is full, and we have to move to another block.

Thus we have our cache block size.

To find the associativity, we hit the cache again and again, as before. After a point, we see that the latency does not change as expected, but remains the same throughout. This is when you have reached the end of your set, so after this, accessing the cache will take the same time.

We can try hitting the cache at specific points in the set, i.e. once in each block. After all the blocks are filled, the time recorded will be the same. The number of times the cache has been hit before this happens will be counted.

Thus we have our associativity.

### Functions used:

To measure the time taken for each cache hit/miss, we can record the time before the after the attempt. This can be accomplished by the \_rdtsc() function (sort for Read Timestamp Counter). This is

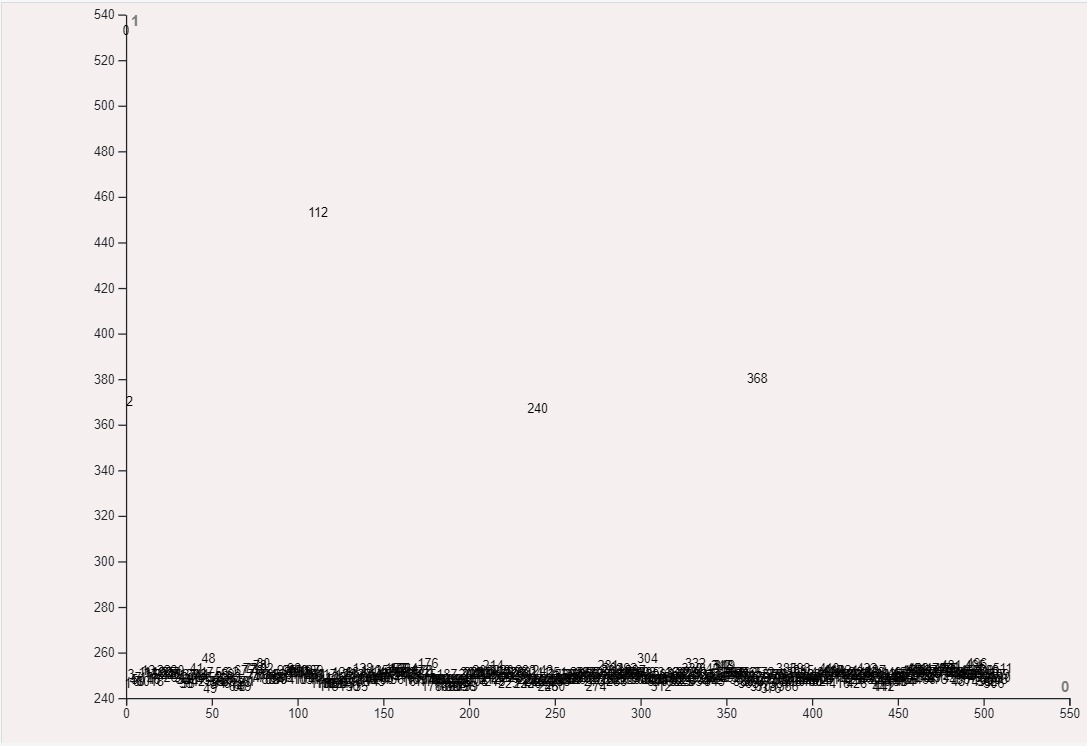
Also we have to make sure we do not accidentally parallelise any operations related to counting time, so we use the \_mm\_mflush() operation to make sure the required operations only after the previous ones are complete.

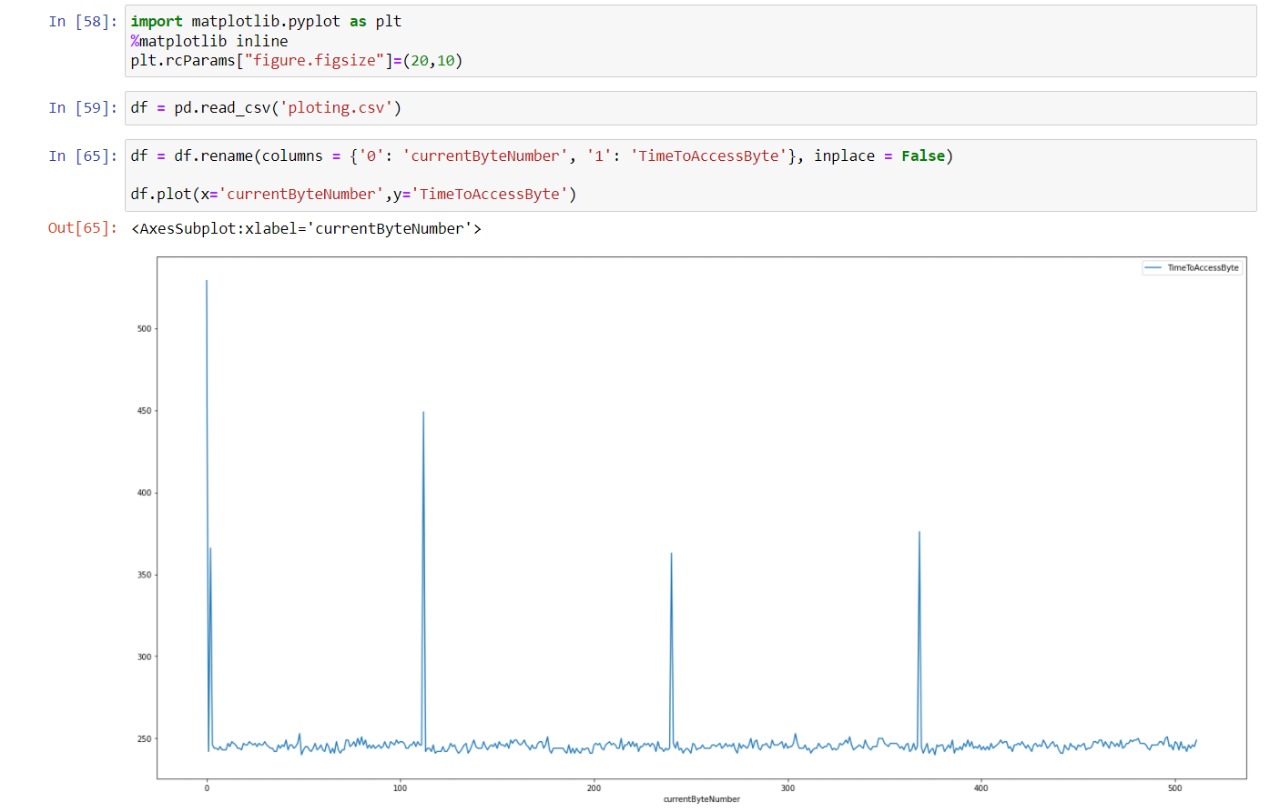
An additional function \_mm\_clflush() (short for Cache Line Flush) clears out a cache line, as the name suggests.

### Block size:

In the first program, we have an array to store the latencies obtained (lat[]), and we have a dynamically allocated array(arr[]) to be used for attempts at hitting the cache. We also have a temporary piece of data (tmp) to use to hit the cache.

Now, we use the process explained above to calculate the time taken for each hit/miss, and repeat the process multiple times to further optimize the values, and remove any noise.





As you can see above, there are high peaks spaced exactly 128 units apart, excluding the first cold hit.

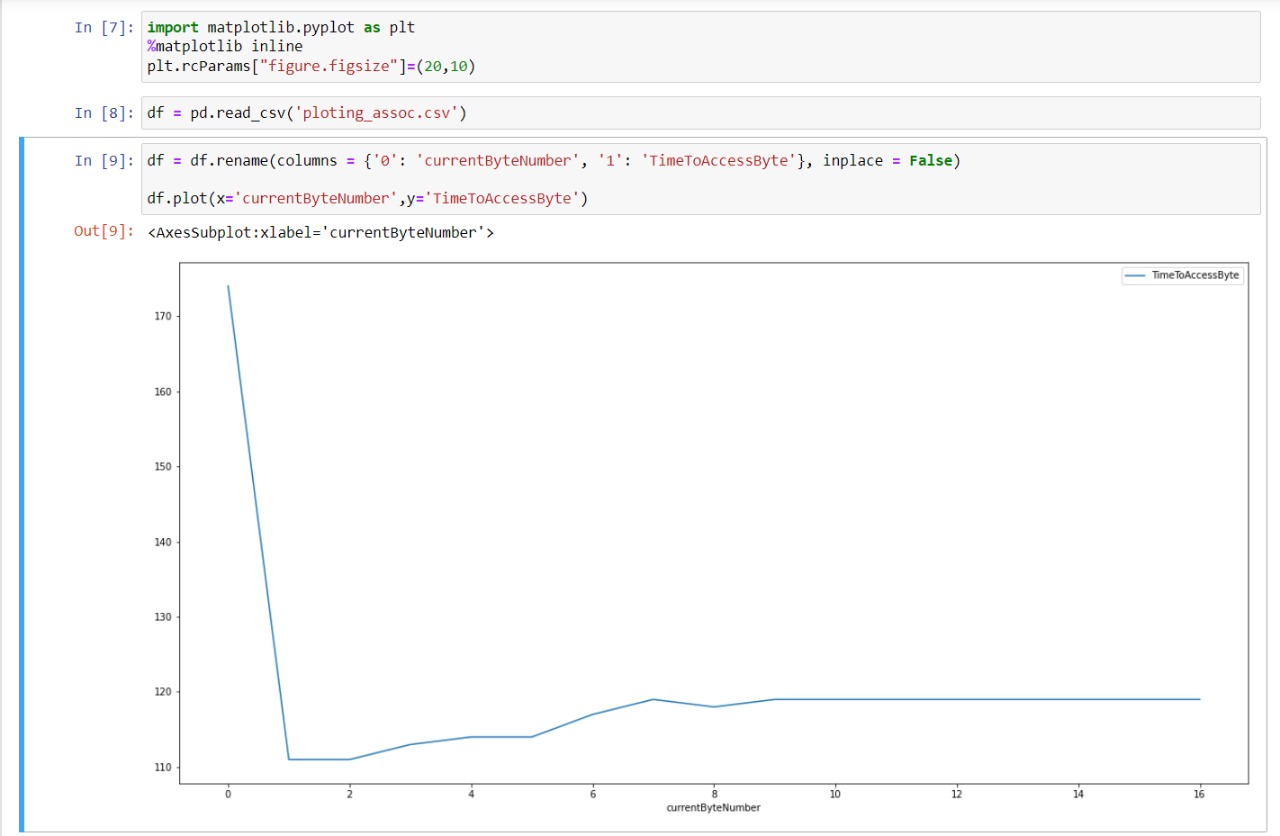
So, from here, we can conclude that the block size is 128 bytes.

### Associativity:

Now that we know our block size, we move on to find the associativity of the cache.

Using the method specified above, and alos the functions and data structures specified, we create a program for this purpose and run it.

The plot for the values obtained look like.

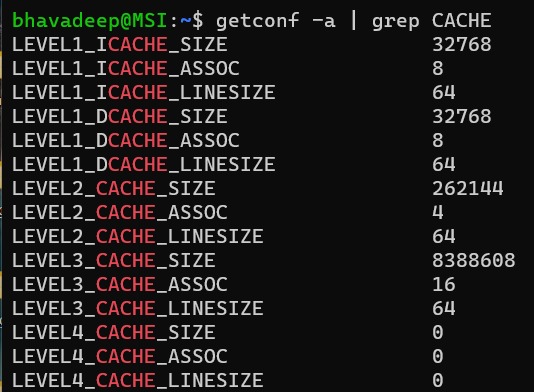


As we can see above, except the first cold hit, the initial latencies start low, and gradually move up, then stagnate at around the 9th hit. This is when we know the L1 cache is full.

So, here, the obtained associativity is 8.

### Verification and conclusion:

Now, we verify our answers. Using the command shown (in Linux) we get the specifications of our cache.



Our values match.

Therefore, we have successfully reverse engineered our L1 cache.