

CS2610 - Lab Assignment 4

Hemesh D J (CS20B031) & Bersilin C (CS20B013)

Latency Measurement

We used the following **assembly instructions** and **methods** for measuring the latency:

- **RDTSC, RDTSCP:** Measures code execution time in terms of clock cycles.
- **CPUID:** Serializes instruction execution with no effect on program flow, except that the EAX, EBX, ECX and EDX registers are modified.
- **_MM_CLFLUSH:** Flushes the block to which the pointer is pointing, from all levels of cache.
- In order to **prevent prefetching**, we accessed the memory in random order instead of sequentially accessing it.
- We measured the number of accesses **between consecutive misses** for finding the block size.
- We took care of slight variations in the number of accesses between consecutive misses by taking its **average**.

L1 Cache Specifications

The L1 cache present in our system (**11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz**) has the following specifications:

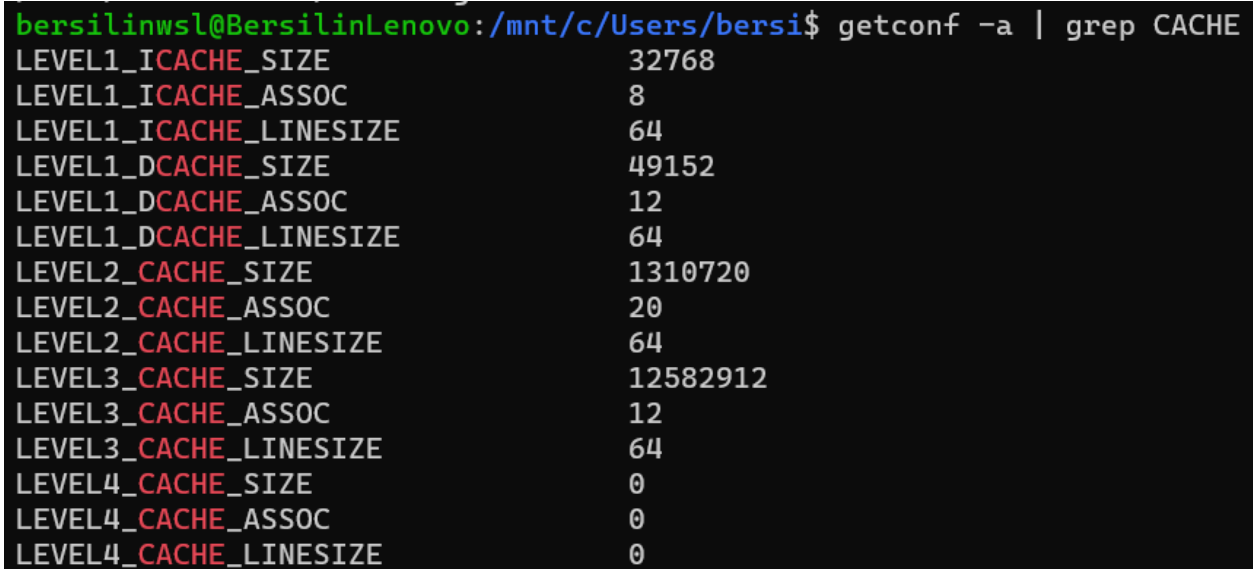
(The following data is for a single core, the **total number of cores = 4**)

- **L1 Cache:**
 - Size of L1 Instruction Cache: **32KB** (32786 Bytes)
 - Size of L1 Data Cache: **48KB** (49152 Bytes)

-
- Associativity of L1 Instruction Cache: **8**
 - Associativity of L1 Data Cache: **12**
 - Block Size of L1 Instruction Cache: **64 Bytes**
 - Block Size of L1 Data Cache: **64 Bytes**

The actual configuration of the cache in the system processor was found using the **“getconf -a | grep CACHE”** command in the **Linux terminal**. This command gave the cache size, associativity and the line size (block size) of 4 different types of cache present in the processor namely **L1 Data Cache, L1 Instruction Cache, L2 Data Cache, L3 Data Cache**.

An example image of the output is given below:



```
bersilinwsl@BersilinLenovo:/mnt/c/Users/bersi$ getconf -a | grep CACHE
LEVEL1_ICACHE_SIZE          32768
LEVEL1_ICACHE_ASSOC         8
LEVEL1_ICACHE_LINESIZE      64
LEVEL1_DCACHE_SIZE          49152
LEVEL1_DCACHE_ASSOC         12
LEVEL1_DCACHE_LINESIZE      64
LEVEL2_CACHE_SIZE           1310720
LEVEL2_CACHE_ASSOC          20
LEVEL2_CACHE_LINESIZE       64
LEVEL3_CACHE_SIZE           12582912
LEVEL3_CACHE_ASSOC          12
LEVEL3_CACHE_LINESIZE       64
LEVEL4_CACHE_SIZE           0
LEVEL4_CACHE_ASSOC          0
LEVEL4_CACHE_LINESIZE       0
```

Cache Block Size

The cache block size inferred from our observations is **64 Bytes**.

Method

While **accessing one byte** of memory the processor **fetches the whole block** of data to different levels of the cache memory. So accessing **spatially local elements** of that particular byte of data will end up as a **hit in the cache** only if it is within the block size of the cache memory i.e. **|Access Byte - Fetched Byte| = Block size**. If the accessed byte fails to meet the previous criteria it ends up as a **miss in the cache**.

We exploit this property to find the cache block size by accessing different byte locations and observing the access time. We then compute the **average time between two consecutive misses** in the cache.

The problem with **sequential access** of byte locations is that the CPU does **next-line(block) prefetching** which causes the access to the next few blocks to end up as a hit in the cache. In order to prevent this, we use **random order access** instead of sequential access. We implemented this by simply **randomly permuting** an array of **0 to N-1** and accessing it in that order.

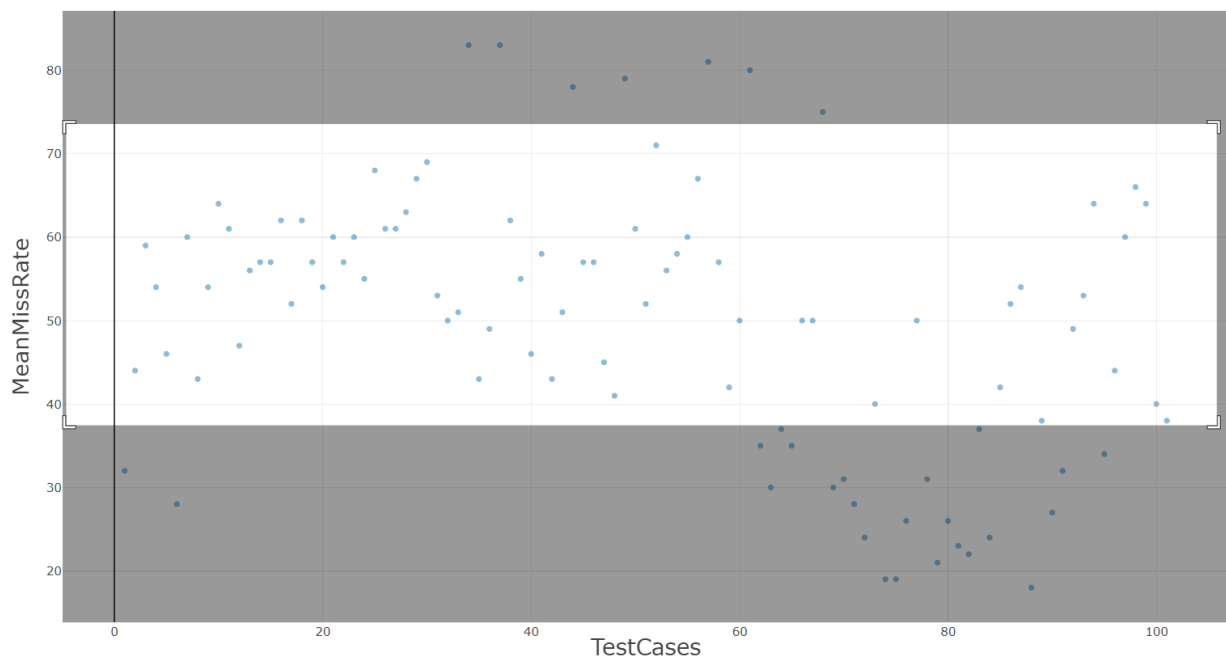
We use the **assembly instructions** mentioned previously to accurately measure latency.

After finding out the access latency for the permutation of memory accesses we built a **classification data analysis** model with **biasing parameter** as **1.5*(avg_access_latency)**. We then take the average of those accesses whose latency was higher than the biasing parameter, **sort them**, find the **difference between the consecutive elements** and find their **average**. This gives us the approximate value of the cache block size. We repeat this process for **k** times with

different access patterns each time to obtain data for the mean and median of the assumed block size.

The above-randomized access pattern and latency measurement work because **we aren't restricting the access variable** whose block to be brought into the cache memory to be **(64*n-1)** and this gives us a better estimate for the cache size in both the worst and best case.

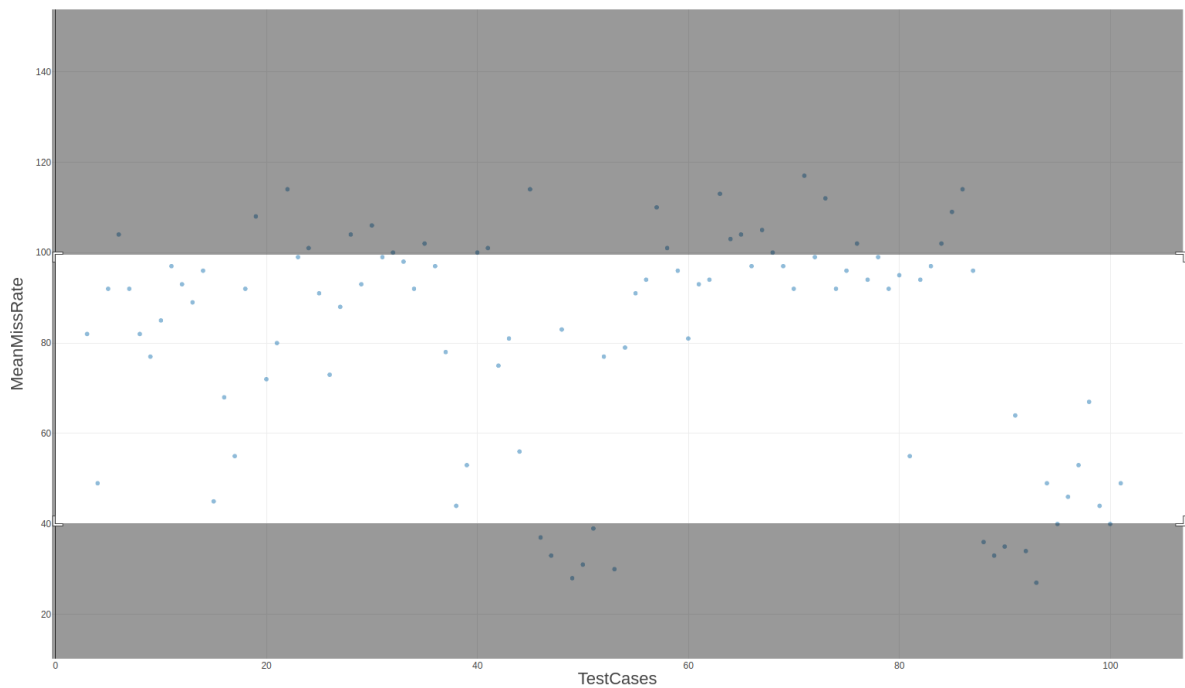
Plot_1



Provided Above is a graph of '**MeanMissRate**' against '**TestCasesNumber**'.

For 101 tests we found out the cache block size and plotted one of our tests for which: **Mean** = 49, **Median** = 55

Plot_2



Provided Above is a graph of '**MeanMissRate**' against '**TestCasesNumber**'.

For **101 tests** we found out the **cache block size** and plotted one of our tests for which: **Mean** = 84, **Median** = 92

Conclusion

Theoretical cache size matches the original cache size of **64B**.

Associativity of L1 Cache

The number of sets in the L1 cache is **256**. Cache Block Size = **64B**

Set Number = (Byte Address / Block Size) modulo (**# of sets**)

So if we access **byte addresses** that are **multiples of $64 * 256$** they are guaranteed to be fetched into only one particular set of the cache.

We exploit this property of cache to find associativity of the cache to form an **algorithm** given below:

- Let's say that cache has an associativity **a**.
- Now **choose** an integer **k** such that **$2a > k > a$** .
- Now access **k distinct Byte addresses** which are multiples of $64 * 256$.
- All these accesses will be **cache misses** and those blocks will be fetched into the cache into the **same set**.
- While fetching the **(a + 1)th** block, the set must be completely full, thereby **evicting a block** that is already present.
- The pattern continues for the rest of the accesses also.
- We have some **(k - a)** Byte addresses evicted and the rest present in the cache.
- **Re-access** the exact same Byte addresses.
- The cache should show a hit for **exactly a**.

This is the algorithm we had written but unfortunately, we weren't able to get accurate results from its execution.