CS2610 ASSIGNMENT 4 : Reverse Engineering of L1 Cache Memory

Aditya C (CS20B003)
Kruthic Vignesh M (CS20B045)

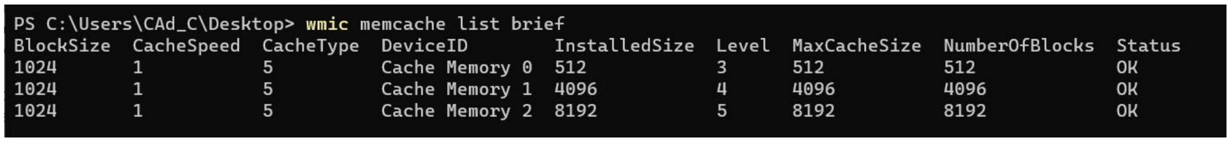
Measuring Latency:

- We used RDTSC (Read Time Stamp Counter) command to find the time taken for data access.
- RDTSC moves the value in the processor's time stamp counter to the EDX:EAX registers. We convert this into a 64-bit integer.
- We get the time stamp counter's values before and after memory access and subtract them. This gives us the number of CPU cycles taken for memory access.
- To ensure the time measured is solely for the memory access, we used the CPUID instruction. CPUID serializes the instruction execution, thus ensuring all the previous instructions are completed before the next instruction is fetched and executed.

Specifications of L1 cache in system:

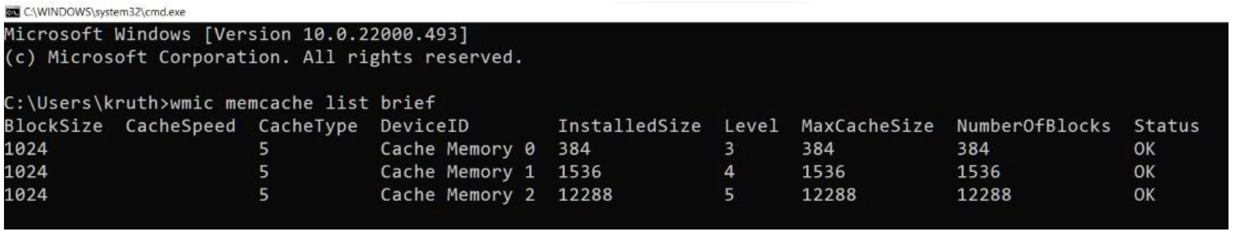
- On windows: `wmic memcache list brief`

  1. *Processor = AMD Ryzen 7 4800H*



```
PS C:\Users\CAd_C\Desktop> wmic memcache list brief
BlockSize  CacheSpeed  CacheType  DeviceID        InstalledSize  Level  MaxCacheSize  NumberOfBlocks  Status
1024       1           5          Cache Memory 0  512            3      512           512             OK
1024       1           5          Cache Memory 1  4096           4      4096          4096            OK
1024       1           5          Cache Memory 2  8192           5      8192          8192            OK
```

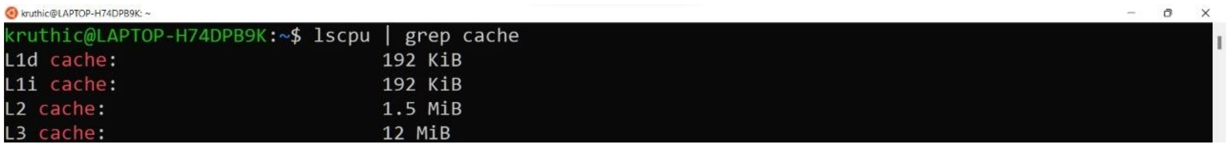
  2. *Processor = 10th Gen Intel i7 10750H*



```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows [Version 10.0.22000.493]
(c) Microsoft Corporation. All rights reserved.

C:\Users\kruth>wmic memcache list brief
BlockSize  CacheSpeed  CacheType  DeviceID        InstalledSize  Level  MaxCacheSize  NumberOfBlocks  Status
1024                   5          Cache Memory 0  384            3      384           384             OK
1024                   5          Cache Memory 1  1536           4      1536          1536            OK
1024                   5          Cache Memory 2  12288          5      12288         12288           OK
```

- On Linux: `lscpu | grep cache`
  Processor = 10th Gen Intel i7 10750H



```
kruthic@LAPTOP-H74DPB9K: ~
kruthic@LAPTOP-H74DPB9K:~$ lscpu | grep cache
L1d cache:                       192 KiB
L1i cache:                       192 KiB
L2 cache:                        1.5 MiB
L3 cache:                        12 MiB
```

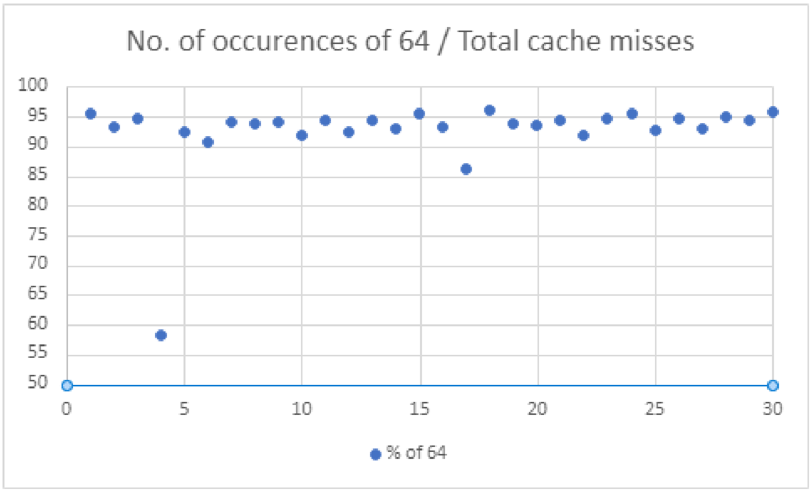
What is the block size obtained by reverse engineering?

64 bytes

Primary method:

- We declared a large array arr, size of the order 2^16, and accessed its elements. Using the latencies for the memory accesses, we classified them into cache hits and cache misses.
- We sequentially moved from arr[0] to arr[N-1] (N = size of the array) and accessed the elements in order. We then computed the number of hits between 2 misses and stored them in an array 'gap'.
- Note that the array "gap" stores the difference between indices corresponding to 2 consecutive cache misses.
- For example, if we incurred cache misses when accessing arr[1024] and arr[1088], then 1088-1024 = 64 gets added to the array gap.
- We computed the array "gap" and obtained the **mode**, which came out to be 64 bytes.

As CPUs may try to optimize for sequential access by doing next-line prefetching, we considered the mode of the array "gap" to be a more relevant parameter to be studied than the mean. However, we also present the mean number of hits between 2 misses in our analysis.



The mean of the array gap was observed to be higher than 64. This can be attributed to next-line prefetching by the CPU (the number of hits between 2 misses doubles if 2 blocks are fetched instead of 1).

Secondary method:

We moved randomly within the array arr (the index accessed was randomized using rand() function) and we computed the number of continuous cache hits before a miss. We computed the mean number of continuous cache hits before a miss. This mean came in the range [45,120].

The block size measured using the primary method (mode of the "gap" array) matches with the actual block size of the cache.

Link to code and output files generated: https://github.com/AdityaC-003/CS2610-Lab

What is the cache associativity obtained by reverse engineering?

8-way associativity

- We declared a large array arr, size of the order 2^25, and accessed blocks that belong to the same cache set.
- We access the i^th block that belongs to a given cache set, and then attempt to access the previously inserted (i-1) blocks of that cache set. If all of the (i-1) accesses are cache hits, then the associativity of the cache is >= i. If there is a cache miss, then the insertion of the i^th block led to one of the previously inserted blocks getting evicted from the cache. Hence, i-1 will be the associativity of the cache.
- On running the above algorithm, we got the associativity to be in the range [6,9], and the mean of the measured values to be close to 8.

Link to code and output files generated: Click Here