Reverse Engineering CPU Caches

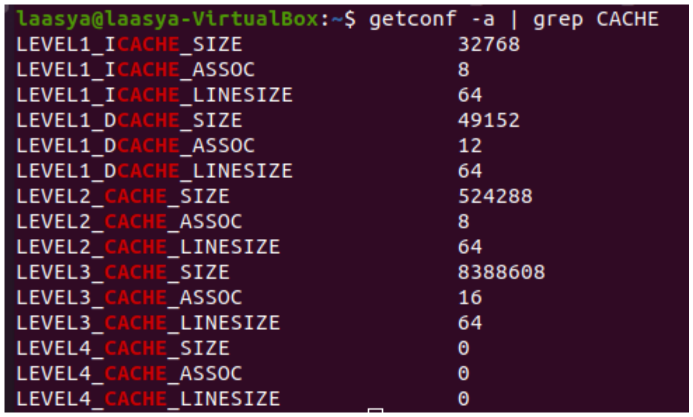
Team Members:
Kothapalli Sai Shre Laasya Raagyni - CS20B043
Saran Kumar - CS20B069

- **Objective:** To identify the cache block size and the associativity of the L1 cache.
- **Concept Used:** We try to find the cache block size and associativity by taking advantage of the fact that the latency for access of different layers of the cache and main memory is different. When we access an element, say a[0], the element, along with the block associated with it in the main memory, is sent to cache memory. By accessing neighbouring elements, we look out for any spike in the time it takes to access these elements. A spike indicates that the element is not present in the cache, hence, we will know the size of the cluster of elements brought into the cache from the main memory when a[0] is accessed. This way, we can deduce the block size of the cache.
- **Instructions Used To Measure The Latency:** We have used RDTSC, RTDSCP and CPUID instructions.

    RDTSC: This instruction reads the value of the timestamp register and stores it into the EDX (first 32 bits into EDX) and EAX (next 32 bits into EAX) registers.

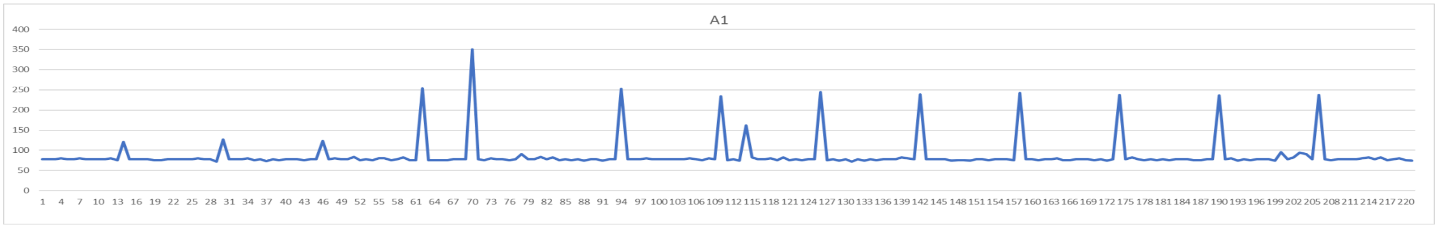
    RDTSCP: This has the same functionality as RDTSC but with an additional feature of storing the CPU id into the ECX register. Due to this, the RDTSCP instruction waits until all the previous instructions are executed before reading the timestamp register.

    CPUID: This instruction is a serialising instruction which makes sure that every preceding instruction is executed before continuing to the next instruction. <u>This ensures that only the function that is between the two RDTSC calls are executed and the latency measured is accurate</u>
- **Implementation of the Instructions for Accuracy:** We used the CPUID instruction before the first RDTSC instruction to avoid out of order execution. As we execute it before the RDTSC instruction, the CPUID instruction is not included in the latency measurement. After storing the timestamp into EDX and EAX registers, we execute the statements whose latency is to be measured. To avoid any overhead due to function calls, we directly wrote the statements instead of a function. After the statements are executed, we read the timestamp register again. This time, instead of RDTSC, we use the RDTSCP instruction to ensure that all the statements previously are executed before the timestamp register is read again. Then we write the values in EDX and EAX onto the memory using the mov instruction. We can be sure that this will be performed only after the timestamp register is read and will not interfere before as both instructions use the same registers. Thus, the CPU will have to wait until the timestamp is read and written into the registers to write the same onto the memory. Then, finally we have a CPUID instruction to make sure that none of the later statements are executed before the RDTSCP instruction. This way, we ensure that the latency measured is only for the desired set of statements and is accurate.
- **Specifications of L1 Cache using system command:** We used the command *$ getconf -a | grep CACHE* in the command prompt in the Linux environment to get the details of all the levels of Cache in the system. It produced the following result:

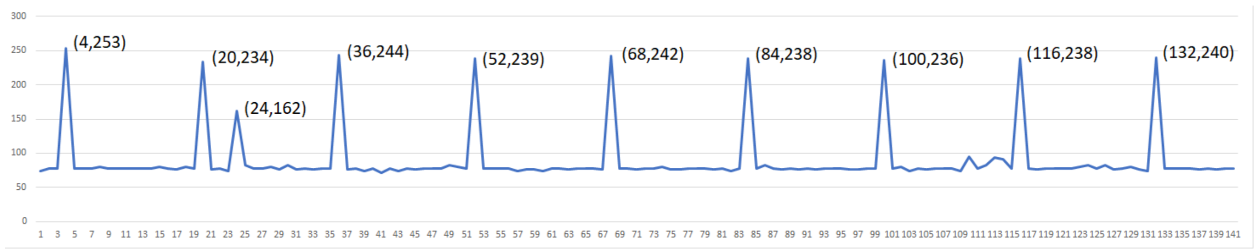


```
laasya@laasya-VirtualBox:~$ getconf -a | grep CACHE
LEVEL1_ICACHE_SIZE                 32768
LEVEL1_ICACHE_ASSOC                8
LEVEL1_ICACHE_LINESIZE             64
LEVEL1_DCACHE_SIZE                 49152
LEVEL1_DCACHE_ASSOC                12
LEVEL1_DCACHE_LINESIZE             64
LEVEL2_CACHE_SIZE                  524288
LEVEL2_CACHE_ASSOC                 8
LEVEL2_CACHE_LINESIZE              64
LEVEL3_CACHE_SIZE                  8388608
LEVEL3_CACHE_ASSOC                 16
LEVEL3_CACHE_LINESIZE              64
LEVEL4_CACHE_SIZE                  0
LEVEL4_CACHE_ASSOC                 0
LEVEL4_CACHE_LINESIZE              0
```

- **Reverse Engineering to find the Cache Block Size:** Using the concept explained previously, we declared an array of size 240 and initialised it to 0. Now, since we had already accessed the elements of the array while initialising the array, we use __*mm*__*clflush()* function to clear the elements of the array in the cache. After this, a for loop was written and in each iteration, the function *timediff()* was called. All the methods to find the latency accurately that was mentioned before is implemented in this function. We pass the array and the index as arguments to this function. In the function, the timestamp is recorded before and after this statement and the latency is returned as an unsigned 64-bit integer. The function accesses an element and it returns the no. of cycles spent for accessing the element. In each interaction we call the function and print out the no. of the clock cycles it each time. Thus, the output had 240 rows with the latency of the index's memory access and the index of the element called. This result was then copied onto a .csv file and used an online plotter to plot the points. It was then observed that there were some spikes and the interval between the spikes indices was 16. As we had initialised an integer array, this means that the interval is of size 64 bytes. From this, we can deduce that the block size of the cache is 64 bytes which matches with the specifications mentioned before.
- The plot of Clock Cycles taken to access the array element vs The index of the array element is shown below.



We can notice the spikes in clock cycles taken at different intervals. We can observe the interval clearly once we enlarge the graph.



From the graph, we can deduce that the block size is 16 x 4 = 64 bytes (since we used an integer array).
This matches with the cache block size as seen from the data obtained using *$ getconf -a | grep CACHE* command.

- **Reverse Engineering to find the Associativity of the Cache:** We shall first fill a particular set of the cache with blocks from the main memory by accessing the appropriate elements of the array. After bringing in a new block, we check if any of the blocks which were previously brought into the cache are evicted by accessing the elements of the array which are present in the blocks brought in. If all the memory accesses have the same latency, this means that there is no need to evict any block. Continue to bring in new blocks to the cache. If there is a spike in the latency, we can say that that particular block was evicted in the previous memory access. This has occurred as the set has reached its capacity. At this point, we know the number of elements that were brought in newly to the cache and thus, we can deduce the associativity.
- **Implementation of the concept in our Code:** To bring in new elements to a particular set, we shall first find the number of sets in the cache and call the elements of the array accordingly.
From the data obtained using *$ getconf -a | grep CACHE* command, we see that
  Cache Size = 49152
  Block Size = 64
  Cache Associativity = 12
We know that
  Cache Size = Block Size * Cache Associativity * Number of Sets
  => Number of Sets = 49152/(64*12) = 64
Offset = 64*64/4 = 1024 (divided by 4 because we are considering int array)
Thus, all the elements of the array whose indices have their interval as a multiple of 1024 (64*64/4) will be called into the same set of the cache.
We then write two for-loops. The outer for-loop is for bringing in new blocks onto the same set of the Cache. The inner loop is for checking the latency of access for all the previously brought blocks in the same set.
As we can see from the data_associativity.txt file which contains the output of the code_associativity.c file, we can see that from 0th until the 11th block, when a new block is brought into the cache, the latency is similar for all the memory accesses of the previously brought-in blocks of the same set. After bringing in the 12th block, we see a sudden spike in the latency of the memory access for the 11th block. This indicates that the 11th block was evicted when the 12th block was brought-in, which resulted in the increased latency for the memory access of the 11th block.
From this, we can conclude that the associativity of the set in the Cache is 12 and thus, the associativity of the cache is 12.
This matches with the cache associativity as seen from the data obtained using *$ getconf -a | grep CACHE* command.

Hence, using Reverse Engineering, we have found the associativity of the cache to be 12 and the block size of the cache to be 64 bytes.

THANK YOU!