Report for assignment 4

cs20b012 and cs20b018

1 Measuring latency

We use the RDTSCP assembly instruction from Intel 64 and IA32 instruction set architectures. The assembly instruction reads the timestamp register and the CPU identifier. The value of the timestamp register is stored into the EDX and EAX registers; the value of the CPU id is stored into the ECX register.

The RDTSCP instruction waits until all previous instructions have been executed before reading the counter i.e. everything that is above its call in the source code is executed before the instruction itself is called.

Modern processors support out-of-order execution of the code. The purpose is to optimize the penalties due to the different instruction latencies. But we need to measure the time taken for in order execution of code between the two RDTSCP instructions. The solution is to call a serializing instruction before calling the RDTSC one. This is CPUID which can be executed at any privilege level to serialize instruction execution with no effect on program flow.

By measuring the latency multiple times and taking the average, we can get a more consistent and accurate result.

To prevent our program being migrated to a different core by the OS we run it using the taskset command in Linux.

We also use isolcpus; a boot parameter in Linux to prevent the program from being interrupted. This isolates a CPU core and prevents the OS from scheduling other programs on it.

2 Cache specifications

L1 cache size = 32768 bytes
L1 block size = 64 bytes
L1 associativity = 8

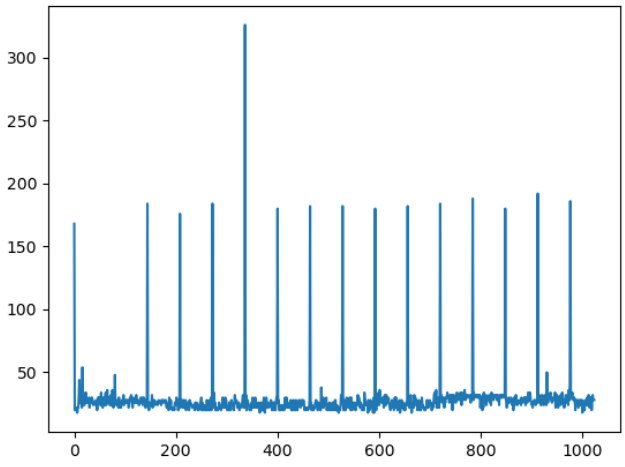
This information can be obtained by using the following command

getconf -a | grep CACHE

3 Measuring cache block size

First we create a character array of size 1024 and initialize the elements of the array randomly. Then we clear the cache using the _mm_clflush() function. Now we perform load operations on each element of the array and determine the access latency using the procedure mentioned in Section 1.

We record the access latency in each case and observe the indices where there is a spike in latency. We do this multiple times and plot the average access times. We obtain the following plot.



The spikes in latency are caused when we access an element of the array which is not present in cache. Whenever we access an element, the entire memory block in which it is present is brought into cache. When we reach the end of the block, we have to bring the next block into cache from memory which has much higher latency and hence causes the spike.

From the plot we obtain a cache block size of 64 bytes. There are occurrences of 128 byte gaps between spikes which can be attributed to processor next-line prefetching.

4 Measuring cache associativity

In order to determine associativity of the cache we need to fill a set completely so that the next access to the set evicts an existing block. This may be observed as a spike in latency.

We create an array whose size is larger than that of the cache. The number of sets in the cache is found to be 64. Since cache block size is 64, accesses to memory addresses 0, 4096, 8192, ... and so on will go to the same cache set.

Whenever we bring a new block into the cache, we measure the access latency for the blocks which have been added before, to check whether any of them have been evicted. If we observe a spike in access latency, then the number of blocks which have been added before the current block is the associativity of the cache.

The results obtained (included as an output file) indicate that the associativity of the cache is 8.