CSE 240B – Parallel Computer Architecture
Measuring Cache Performance of a CPU

Utkarsh Singh (A53267107)
usingh@ucsd.edu

1 Introduction

In this project, the aim was to design a program that is capable of measuring the latency as well as sizes of the caches present in any computer. More specifically, we try and observe different cache latency and the main memory latency for single thread and multi thread processes for both read only and read/write accesses.

The program measures the latency of the entire memory hierarchy of a computer system. In this assignment, we are concerned with measuring latency of the data cache. Typically, the memory hierarchy of computing machine consists of several levels of caches and a main memory. In the modern processor, we generally have 3-level cache system. The L1 cache resides closest to the processor and the latency is minimum to access the data stored in it. L1 caches sizes are typically in the range of 16 – 64 KB depending on the processor. In the memory hierarchy, we have L2 cache which is the intermediate storage unit and slightly larger than L1 cache. It is a unified cache shared by all the cores on chip. L1 and L2 caches are present in every core, therefore each core will access its own local storage before accessing the shared storage space. Finally, L3 cache forms the final on-chip cached storage before we access the main memory. It would have the largest latency when compared to all the cache level, however still magnitude smaller than accessing main memory where the data must be read or written onto an off-chip location. L3 cache is usually shared among all cores of the processor.

2 Code Description

The concept used here to measure the cache latency is called Pointer Chasing. It is similar to a linked list. In pointer chasing, one element stored in an array points to the next element to be accessed. For example, if a[0] = 3, the next element accessed will be a[3]. The catch here is that once we access almost all of the elements of the array, the last element points to the first element of the array forming a loop. Using this concept and varying array sizes, we can measure the latency of each cache and the main memory. We first start off with a small array size, say of 64B or 128B. In this case, all the data will be fetched from the main memory on first access and since the entire array fits in the L1 cache, it will give us the latency of the L1 cache. As the array size increases, the data will not be just in the L1 cache, it will start getting stored in L2 as well. At this point, if we further increase the array sizes, we see an increase in the memory access time. This will increase sharply as the array sizes becomes large enough so that it penetrates the L3 and the main memory. This can be observed in the graphs given further in this report.

The program works for both single thread and multi-threads. The program creates an array of integers which are accessed by the threads. The array of integers is created similar to a linked list as stated above. A linked-list or a chain is formed in a way that shuffles the addresses of the entire memory such that no memory access is sequential. Once randomized, the memory is accessed for fixed number of times. The access time is recorded, and this gives an approximation of the amount of time it takes to access each element of the memory. The algorithm for the code is described in Fig 1.

The above algorithm is run on each thread. The number of threads and whether only read access is required or both read and write access is required is configured during run-time. Dedicated memory is allocated for all threads, which can be 1, 2, 4, 8, or 16 in number, and every thread is made to access memory simultaneously.

The number of threads decide the size of array and the number of threads that would be created while the program is being executed. OpenMP is used to perform multi-threading. The compilation flags used by the program are –std=c++11 for 2011 C++ standard and -fopenmp to enable multi-threading via OpenMP. Another variable "rw" is determined by whether the user wants just read permission or both read and write permissions.

```
Step 1: Allocate memory of a pre-defined size

Step 2: Store sequential values in the data array
        for(i=0; i<size; i++)
            memory[i] = i
            memory[size-1] = &memory[0] //close chain by assigning last element to
               address of first

Step 3: Shuffle the memory (i.e. data array)
        for(i=size-1; i>0; i--)
            k = rand()%(i+1) //random number from uniform distribution
            swap(temp, memory[k]) // swap and shuffle

Step 4: Access the shuffled memory depending on whether only read permissions are
   required or both read and write permissions are required and calculate the average
   time taken per access. The total time that includes all accesses is returned.
        if(read)
            start_count()
            while(count--)
                pointer = *pointer
            end_count()

        if(read-write)
            start_count()
            while(count--)
                next = *pointer
                temp = * pointer //Read the memory location
                temp = temp & 0x7FFFFFFFFFFFFFFF //Technique to turnoff compiler
                   optimization
                *pointer = temp //Write to the memory location
                pointer = next //Point to next memory location
            end_count()

Step 5: The average access time is total time divided by the number of accesses. This is
    the resulting memory access time for the given allocated memory size.
        average_access_time = (end_count() minus start_count())/count //return average
           access time
```

Figure 1: Description of Code

There are a few flaws in the program. Firstly, it is possible that one of the elements in the array points to itself. For example, if a[3] = 3, the program will keep accessing a[3]. In this case, once a[3] is brought into the L1 cache, the final latency will be that of the L1 cache. Secondly, this program might not scale well for threads higher than 16. This is because we have a cache line of 64 bytes, only 16 4-byte integers can be stored in one line. Thus, for false sharing, we can run only 16 threads at a time. This limitation depends on the underlying hardware specifications of the machine. A machine that can handle more memory allocation simultaneously will scale better with more threads.

3 Machine Configuration

I had the access to Comet machine on XSEDE portal. I have done all the testing on the machine. The specifications of are given below in Table 1

Specs	Machine
Processor	Intel Xeon E5-2680 v3
Cores & Threads	2 sockets x 12 cores, 24 threads
L1	2 x 12 x 32KB
L2	2 x 12 x 256KB
L3	2 x 30MB(shared – 12 cores)

Table 1: Machine Configuration

4 Graphs

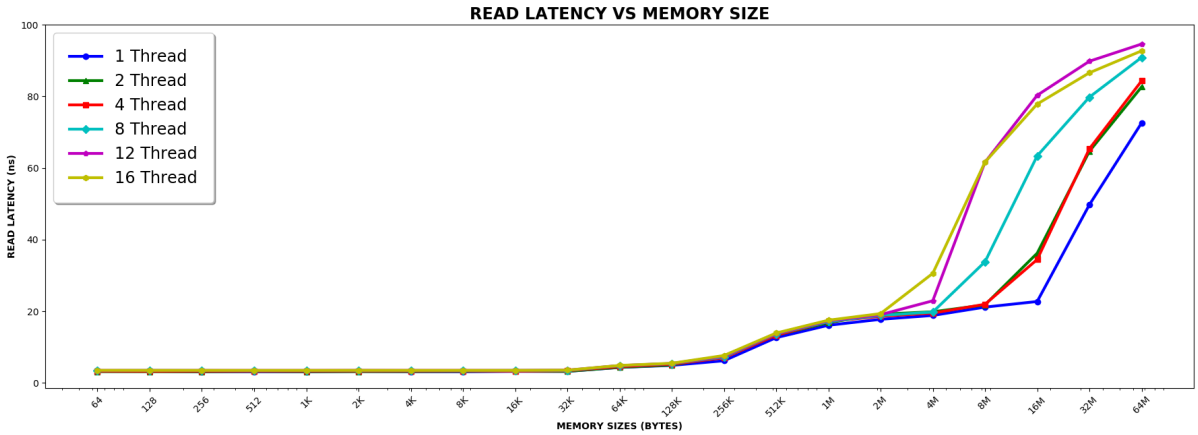

Figure 2: Read only Latency with 1, 2, 4, 8, 12 and 16 threads

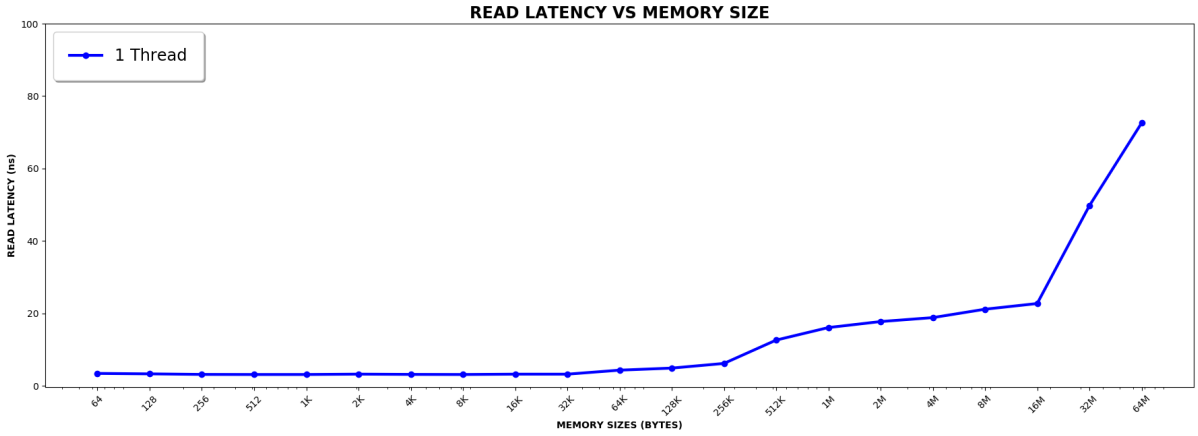

Figure 3: Read only Latency with 1 thread

5 Analysis

5.1 Read Only Memory

Figure 2 shows read latency when the program was run on 1 thread with different memories allocated. In the read only graph, Fig. 2, the most interesting regions are between (1.) 32kB 64kB, (2.) 256kB 1MB, and (3.) beyond 16MB. The small in latency around 32kB signifies L1 cache has been fully utilized and accesses fetch data from L2, which is slightly slower. It further proves a point that the L1 cache size is about 32KB.

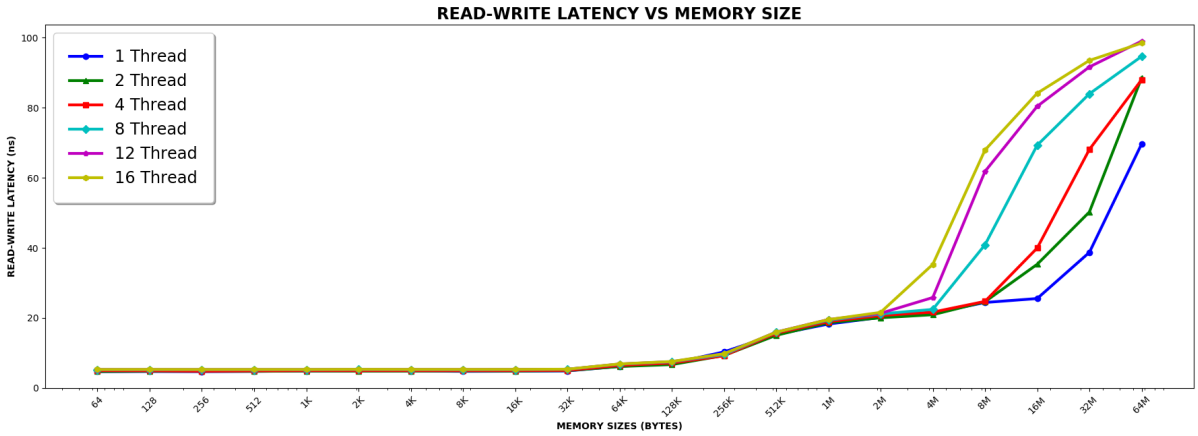

Figure 4: Read-Write Latency with 1, 2, 4, 8, 12 and 16 threads

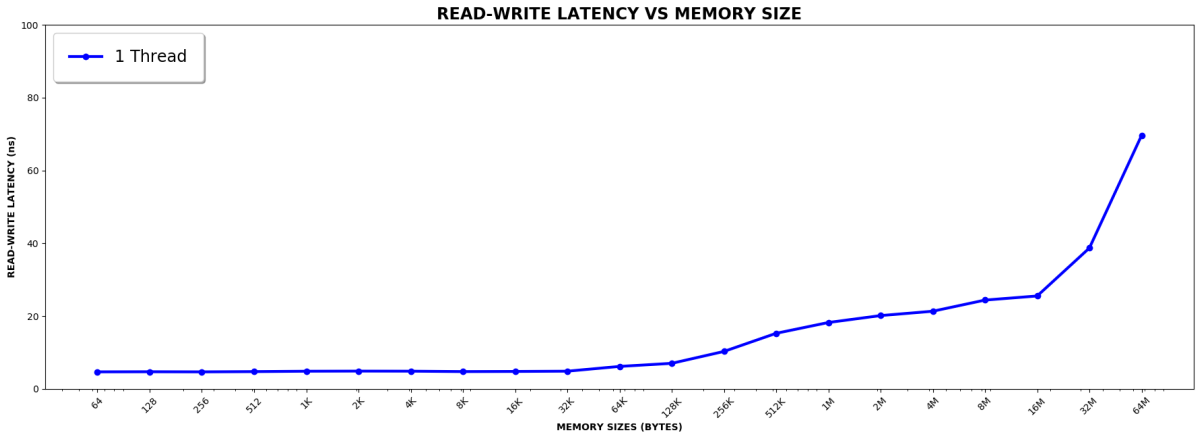

Figure 5: Read-Write Latency with 1 thread

The next abrupt jump happens at 256KB. This denotes that the L1 and L2 caches are filled with data and that the size of L2 is about 256KB. Once we go above 1 MB, we start fetching data from L3 cache. The next interesting region is around 32MB where the latency sees a large jump from 22ns to 75.5ns. This means that all caches are filled, and data is accessed from main memory which is significantly slower than the L3 cache. With this information we can estimate the size of L3 to be somewhere around 16MB − 32MB. In all three regions, there is a steep increase in the memory access latency. This tells us that the computer implements 3 levels of cache. Due to this increasing latency, we can clearly see the differently sized caches. As per the data sheet of the processor (i.e. Table 1), it has a 32 kB L1 cache, a 256 kB L2 cache and a 20 MB L3 cache which matches with the speculated sizes above. Here the L1 and L2 caches are local to each core while the L3 cache is shared among 12 cores of the processor Intel Xeon E5-2680 v3 that runs at 2.50 GHz.

Moreover, we performed the study for multi-threaded program as well. In the multi-threading program, memory is allocated for each thread, and thus the caches start to fill sooner. The interesting result is that he read latency until almost 1MB remain the same for all threads. Since, L2 cache is unified cache being shared by multiple threads, so latency remains similar for the threads up to 1MB. However, once the memory size increases beyond 1MB, the latency start increasing with the number of threads. This is because all the threads start to access unique lines of data that make the L3 cache lines to get evicted more frequently than before. The latency further increases as soon we start accessing main memory.

5.2 Read Write Memory

In the read/write access graphs, the latency is almost the same or slightly higher than that of the read only accesses when the data array is quite small. As the size of the array increases, the latency increases sharply as compared to the read only access latency. While storing data, cache properties such as write-back and write-through, inclusivity, etc. come into picture. Thus, while storing data, the latency might increase due to the listed factors. This changes the shape and the steepness of the graph.

The read/write set seems more stable. While doing a read-modify-write operation, the processor gives the memory in 'OWNED' state in case of MOESI or in 'EXCLUSIVE' state in case of MESI. Either ways, a broadcast is not required while writing into that particular piece of memory, i.e., it can silently change its state. Read/Write could have a little more latency since write could have some additional cost, but overall, it would be more stable than read only memory accesses.

Besides raw cache and memory latency, we observe cache properties such as write-back/write-through and inclusivity. If the cache is a write-through cache, the memory latency for read/write access would be significantly higher since every time a piece of data is written in the cache, it has to be written to the main memory.

6 Extra Credit

The extra credit required comparison of random access and sequential/strided access. The program has been modified for sequential access as well for memory size of 64 MB. The graphs attached below provide the insight how memory access time increase with stride. Starting from zero stride. which basically means we are accessing contiguous elements stored in memory leads to very less latency even for 64MB memory size. It happens because of utilization of spatial locality which basically helps in significantly reducing access times of sequential elements. As we could infer from the graphs, as the stride size increases, we start to access elements stored far away in memory (i.e. spatial distance increases between them), and thus latency keeps increasing util it saturates after which stride size has very less impact. This is because the accesses are not sequential anymore and the lines are retrieved from a higher level in the memory hierarchy. The program starts to behave as we had been doing random access of elements. However, this approach still might lead to improvement in performance in comparison to totally random approach because we are dealing with fixed distance apart elements and compiler might learn this feature and pre-fetch the elements in memory. Moreover, multi threaded program follow the same trends with slightly higher latency's than single threaded program. The graphs obtained during this experiment are shown in subsequent pages.

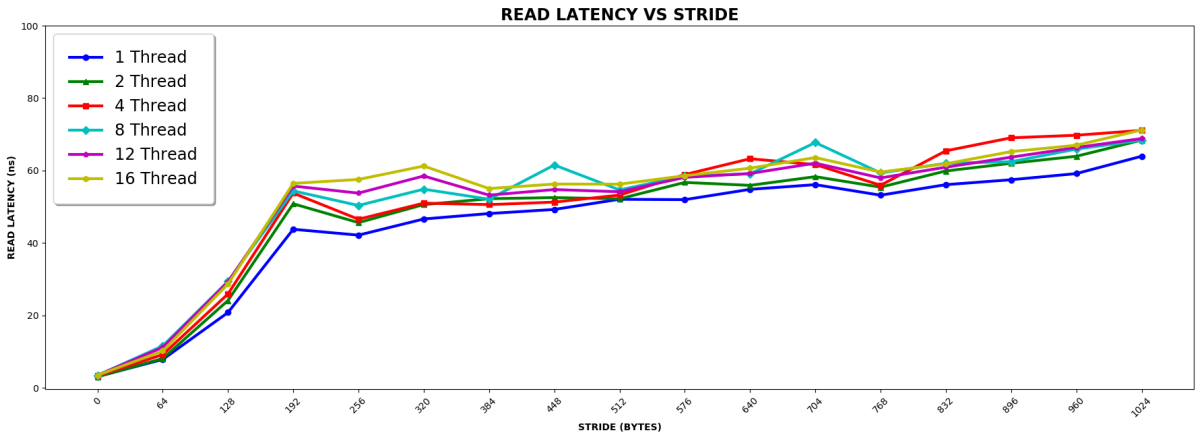

Figure 6: Read latency - Multi Threaded - Strided Access

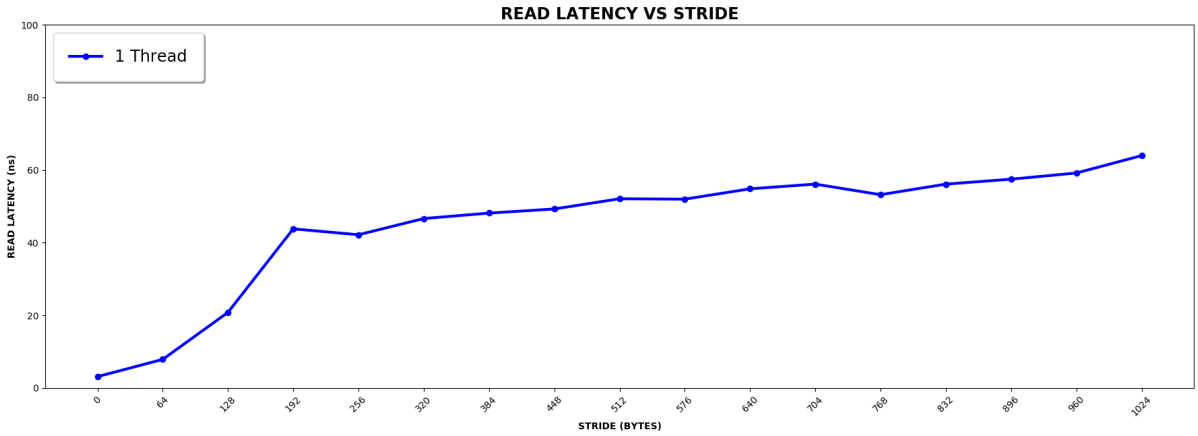

Figure 7: Read latency – Single Threaded – Strided Access

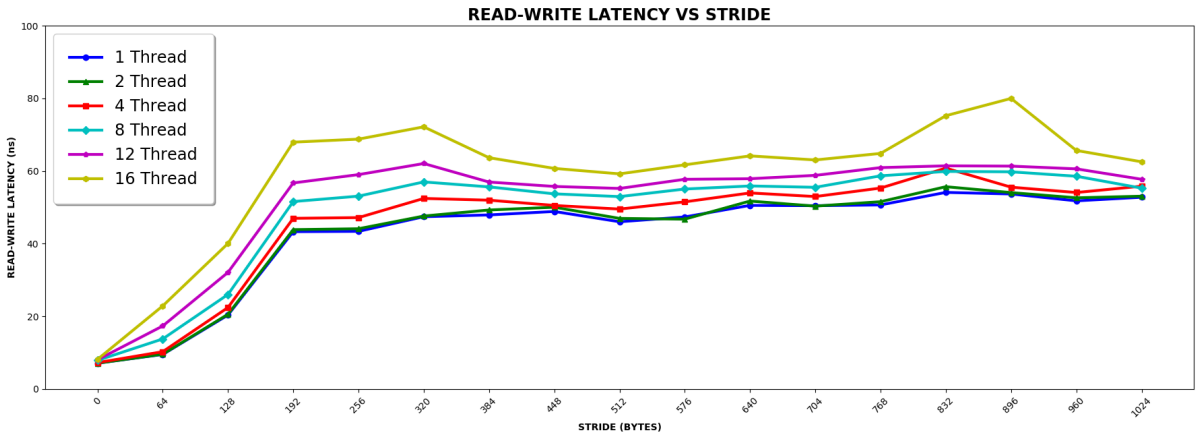

Figure 8: Read-Write Latency - Multi Threaded - Strided Access

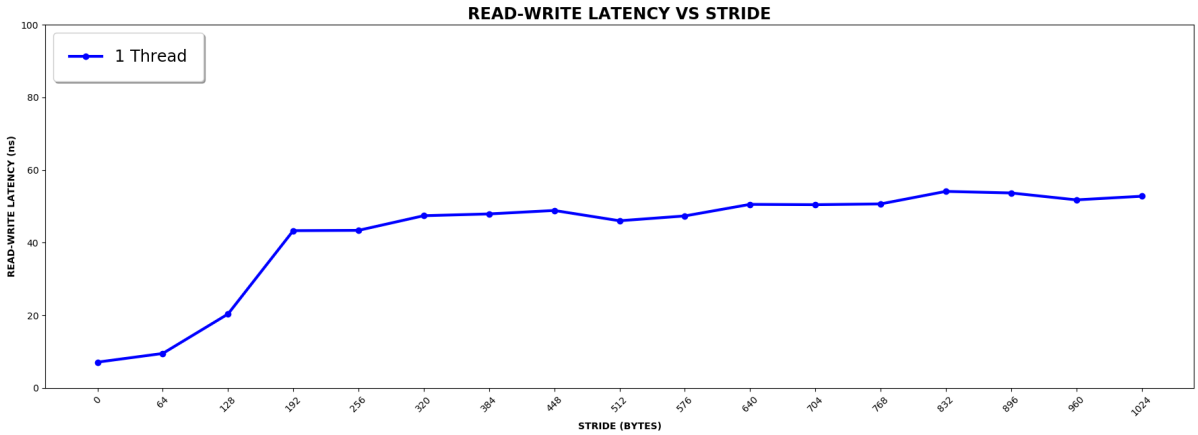

Figure 9: Read-Write Latency – Single Threaded – Strided Access