CSE 240B – Parallel Computer Architecture – Programming Assignment

Cache Memory Latency Characterization

1. Documentation of program

The program developed as a part of this assignment measures latency of the entire memory hierarchy of a computer system. Typically, the memory hierarchy of computing machine consists of several levels of caches and a main memory where all accesses end. Modern processors, with several cores, have 3-level cache system. The L1 cache is the most accessed and the closest form of storage to the processor after internal registers. L1 caches sizes are typically in the range of 16 – 64 kBytes depending on the processor. The L1 cache is also split into two, usually of the same size, one for storing instructions and the other for data. We are interested in the data cache since the instruction cache is entirely managed by the processor and is out of bounds for an end-user to access it. The next level is the L2 cache which is a larger version of L1 but unified, where data and instructions are put together. It is important to note that L1 and L2 caches are present in every core, therefore each core will access its own local storage before accessing the shared storage space. The next level is the L3 cache which is usually shared among all cores of the processor. This acts as the final on-chip cached storage before accessing the main memory outside the chip and this also produces a large latency increase as the data must be read or written onto an off-chip location.

The program developed uses pointer chasing technique to simulate accesses to all levels of the memory hierarchy depending on the size of the allocated memory. The program starts with allocating memory of a user-defined size or a program-defined size. The goal is to access this memory as randomly as possible to ensure that the reads and write do not always end at L1 cache. 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 difference in access times can be noticed as the allocated memory changes.

The algorithm is explained below

1) Allocate memory of a pre-defined size (“memory” represents the allocated memory hereafter)
2) Form a chain with the memory elements (random or strided)
    a. Elements are first stored in order

```
for(i=0; i<len; i++)
        memory[i] = i;
```

    b. The indices of the memory are shuffled(random access) or arranged with constant stride(sequential access)

```
if(random)
for(i=0; i<len; i++)
        //random number from uniform distribution
        j = i + uniform(len – i);
```

```
        if(j != i)
              swap(memory[i], memory[j]);    // swap and shuffle


    if(sequential)
    for(i=0; i<len; i++)
          memory[i] = (i * stride) % len;  // arrange with stride
```

c. The shuffled indices are used to make each element of the allocated memory to point to the address of the next element(which is not sequential since the indices are shuffled)

```
    for(i=0; i<len-1; i++)
    //assign address of next element to current element
          memory[i] = &memory[i+1]

    // close chain by assigning last element to address of first
    memory[len-1] = &memory[0];
```

d. The randomized linked list is returned for accessing(or sequential if chosen)

3) Access the shuffled memory by chasing pointers. The memory is now accessed from the first element. The first element will point to the address of a random element which could be several bytes apart from it. This in turn will point to another random location. This access is continued several times. The total time that includes all accesses is returned. The kind of access depends on the set flag/user argument. The options are “Only Read” and “Read-Modify-Write” accesses.

```
if(read)
      Start time;
      While(count > 0)
// Every iteration points to the address of the next location
            pointer = *pointer;
            count = count – 1;
      end time;


if(read-modify-write)
      start time;
      while(count > 0)
            next = *pointer;
            temp = * pointer;      // READ
```

```
            temp = temp & 0x7FFFFFFFFFFFFFFF    //MODIFY
            *pointer = temp       //WRITE
            pointer = next;      // point to next element
            count = count - 1;
        end time;
```

4) 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.

```
//return average access time
average_access_time = (end time - start time)/count;
```

The above algorithm is replicated on multiple threads when specified by the user. 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. OpenMP is used to perform multithreading(used -fopenmp flag while compiling).

The runtime arguments/options taken by the program are:

`-t / --threads` - Takes in number of threads to run the program on. Default is 1 thread.

`-rw / --read-write` - Enables read-modify-write option of the program. Default is read only.

`-s / --stride` - Enables sequential/strided access. Default is random access

`-ss / --strideSize` - Stride size in bytes to use. Used when strided access is enabled

-h / --help - Displays usage of the program

When no arguments are given, the program runs random access with single thread on range of memory sizes starting from ***64 bytes*** and ending with ***67108864 bytes(64 MB)*** with a step that is double the current size.

Running *doit.sh* runs the program for both read only and read-write cases with 1 thread.

Limitations – Although, the program measures latencies reliably for lower number of threads, it might not scale well for threads higher than 16. Since dedicated memory allocation for all the threads is resource intensive for the processor and the memory of the machine, the program could get killed if the heap allocated by the OS for the program is full. The developed program is stable until 16 threads and starts becoming unstable and gives irregular results once 16 threads is reached and for anything beyond it.

2. Graphs

The program was run on two machines. The specifications of the two are given below in Table 1

Specs	**Machine 1**	**Machine 2**
Processor	Intel Xeon E5-2680 v3	Intel Core i7-8750H
Cores & Threads	2 sockets x 12 cores, 24 threads	6 cores, 12 threads
L1	2 x 12 x **32KB**	6 x **32KB**
L2	2 x 12 x **256KB**	6 x **256KB**
L3	2 x **30MB**(shared – 12 cores)	**9MB**(shared – 6 cores)

Machine 1

Figures 1, and 2 show the read latencies with all threads and 1 thread with logarithmic latency scale respectively using machine 1.

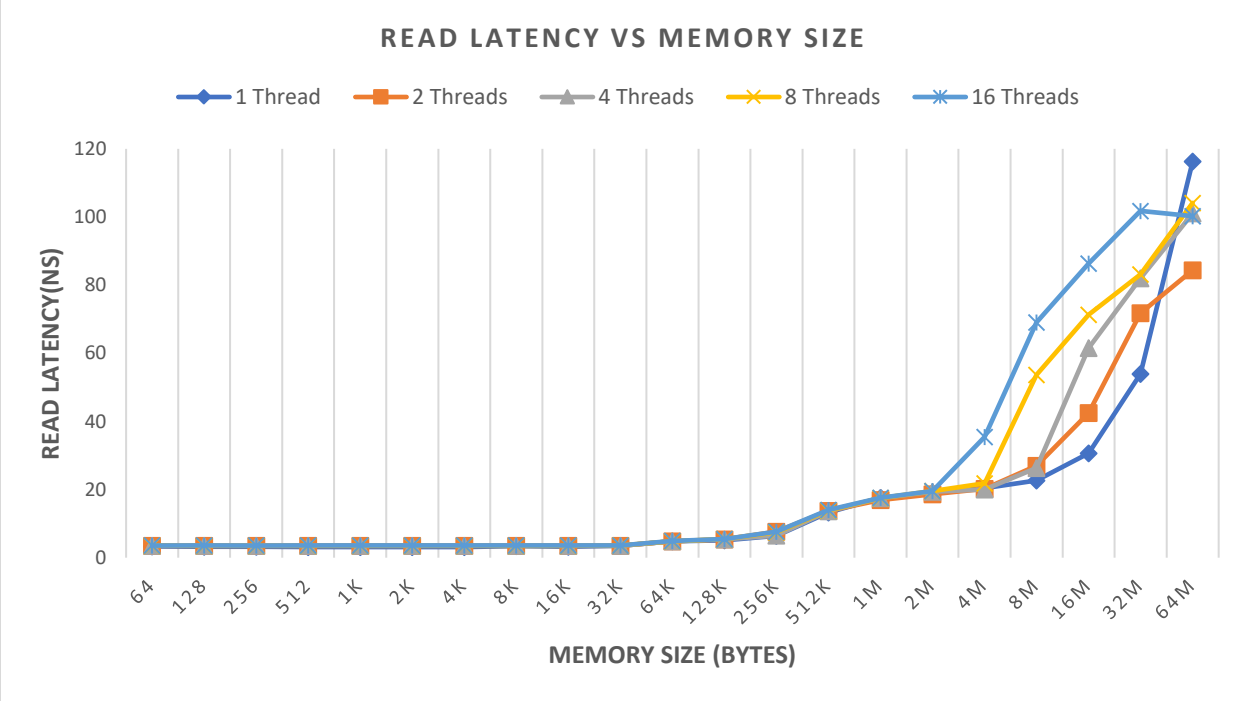

Figure 1 – Read Latency with 1, 2, 4, 8, and 16 threads

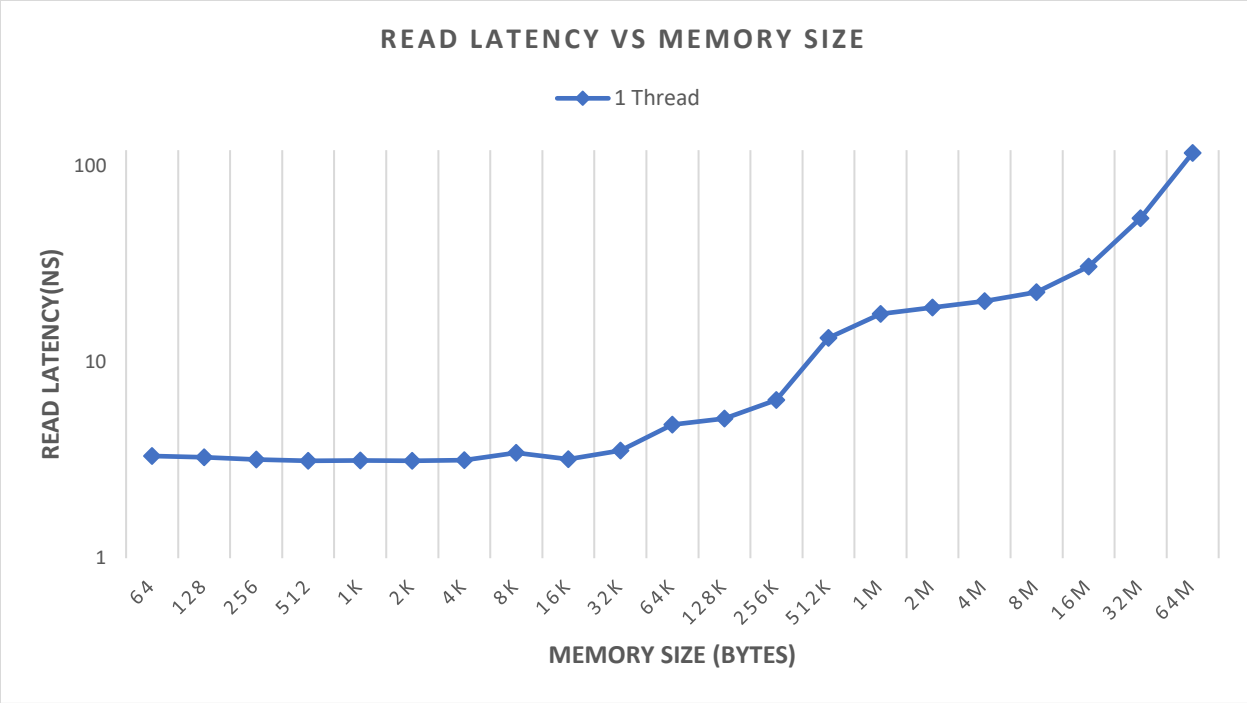

Figure 2 – Read Latency with 1 thread(log scale)

Figures 3, and 4 show the read-write latencies with all threads and 1 thread with logarithmic latency scale respectively.

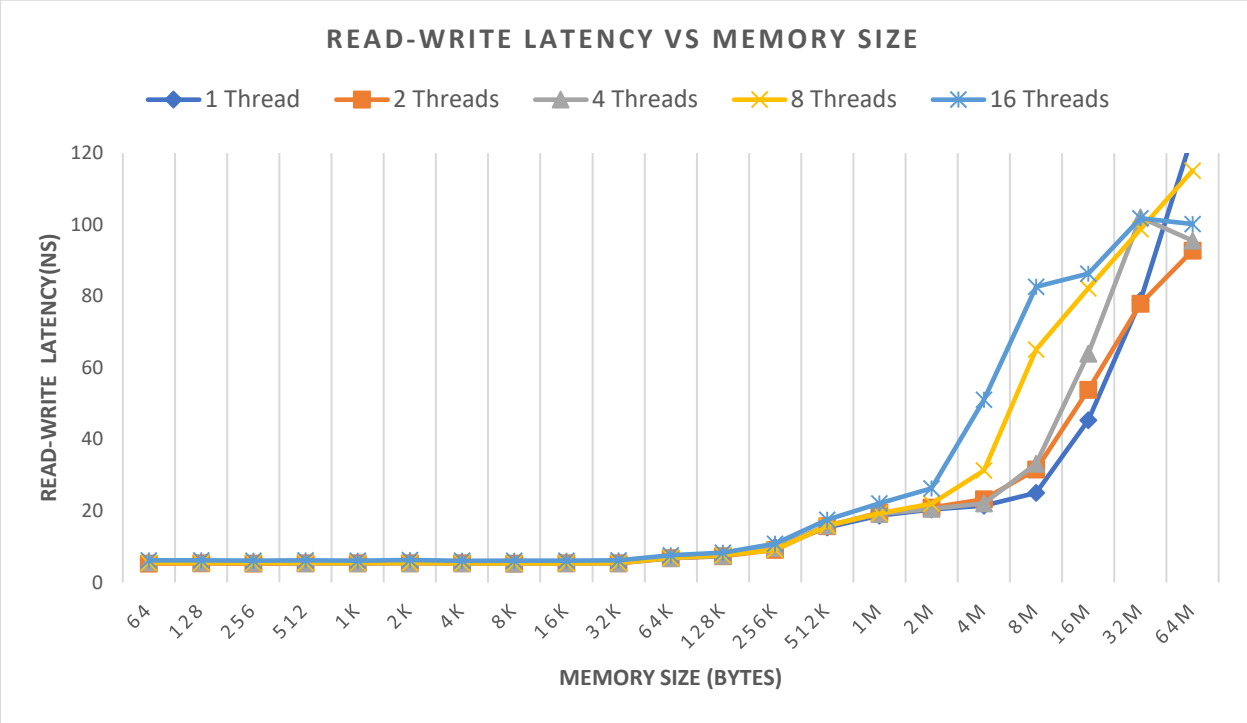

Figure 3 – Read-Write Latency with 1, 2, 4, 8, and 16 threads

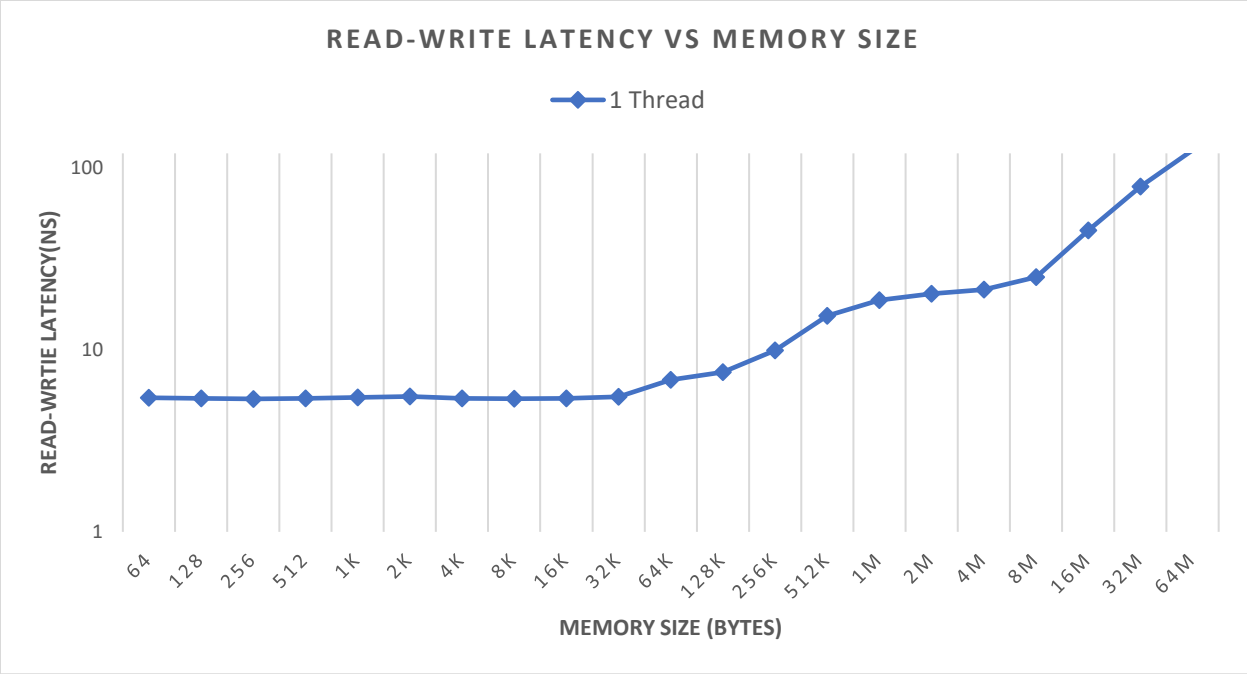

Figure 4 – Read-Write Latency with 1 thread(log scale)

Machine 2

Figures 5, 6, and 7 show the read latencies with all threads, all threads with log scale, and 1 thread with logarithmic latency scale respectively using machine 2.

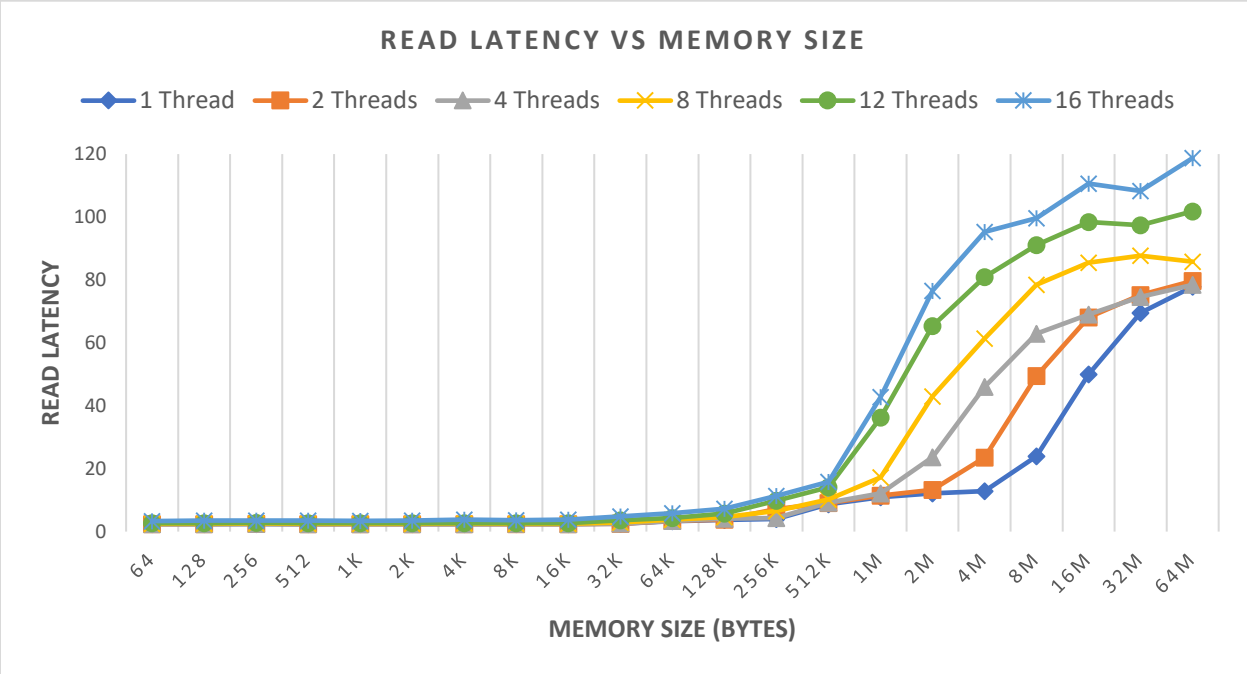

Figure 5 – Read Latency with several threads

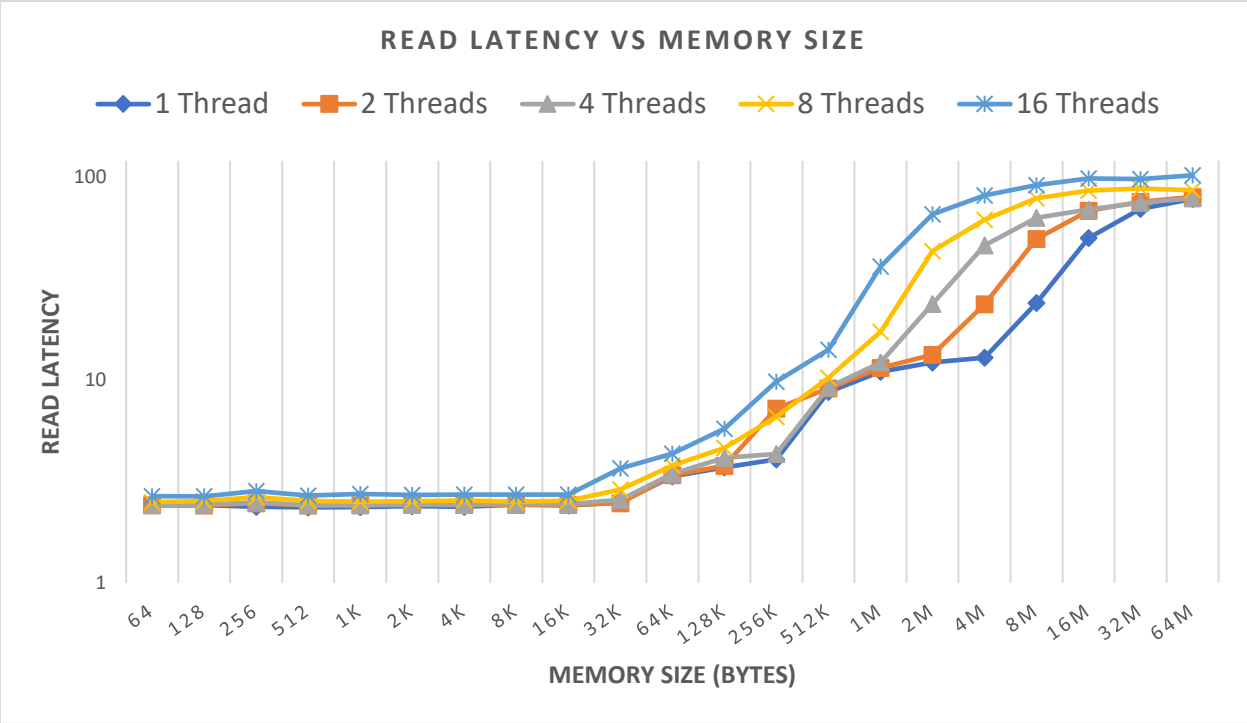

Figure 5 – Read Latency with several threads(log scale)

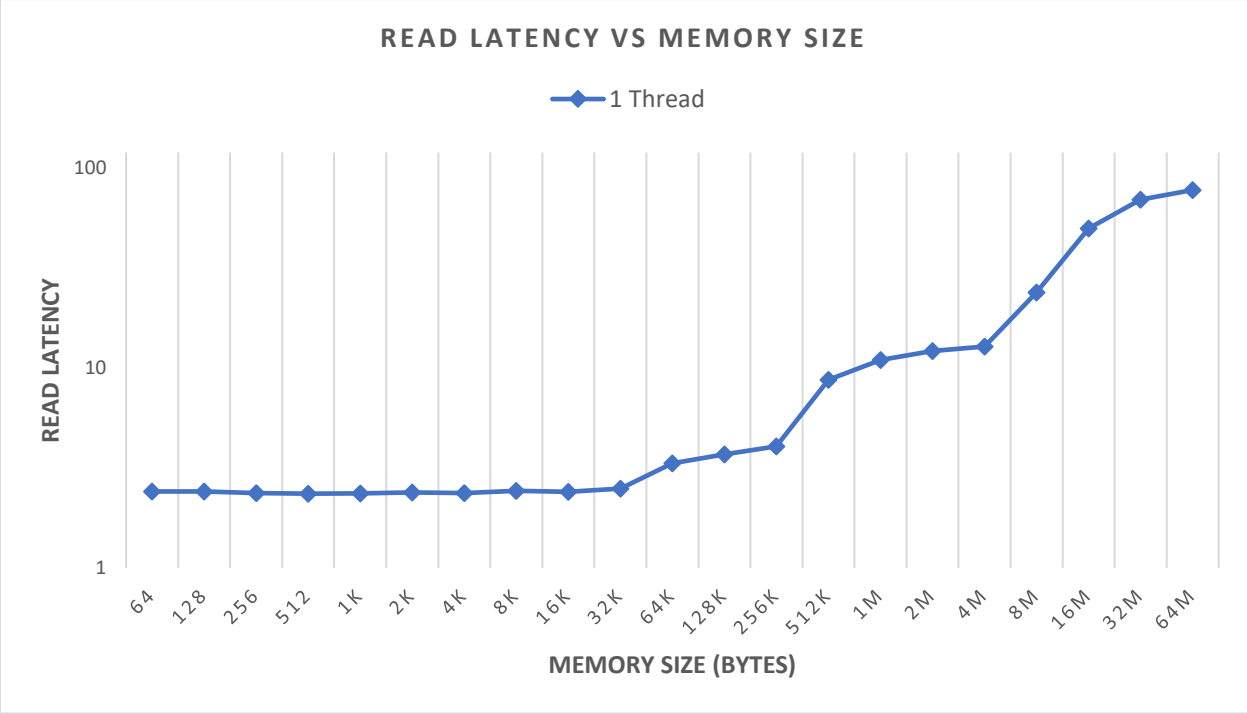

Figure 6 – Read Latency with 1 thread(log scale)

Figures 8, 9, and 10 show the read-write latencies with all threads, all threads with log scale, and 1 thread with logarithmic latency scale respectively using machine 2.

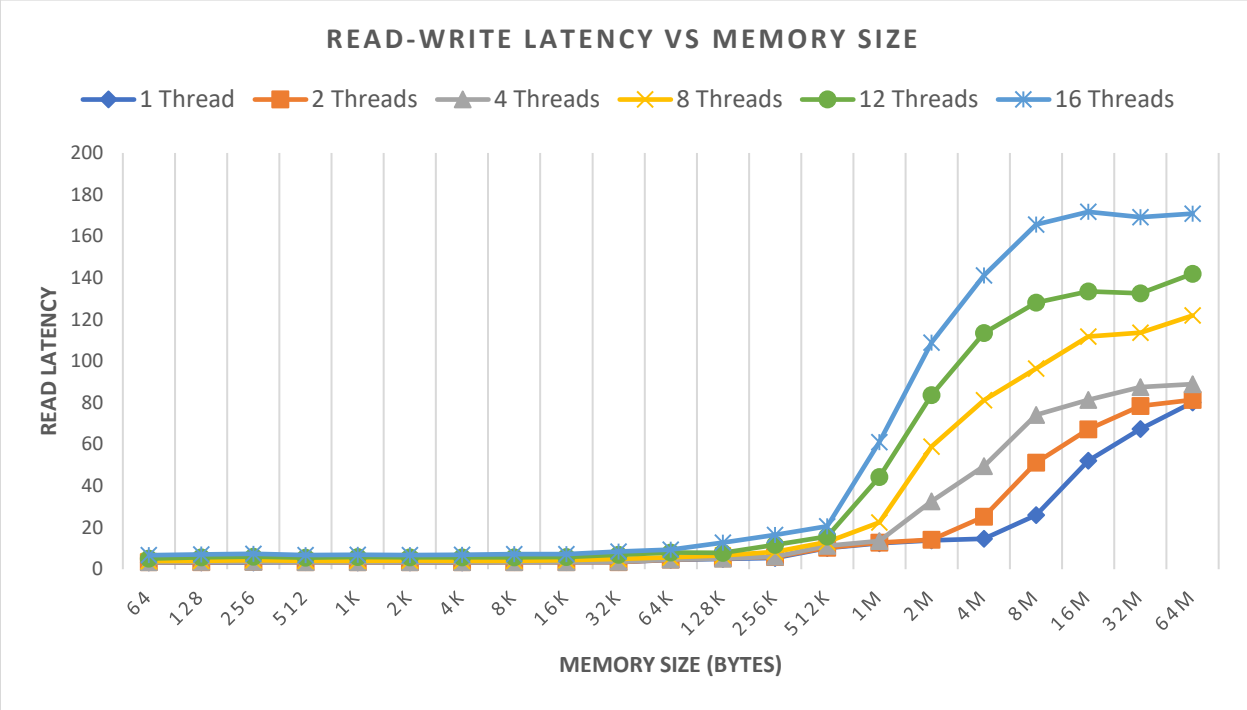

Figure 7 – Read-Write Latency with several threads

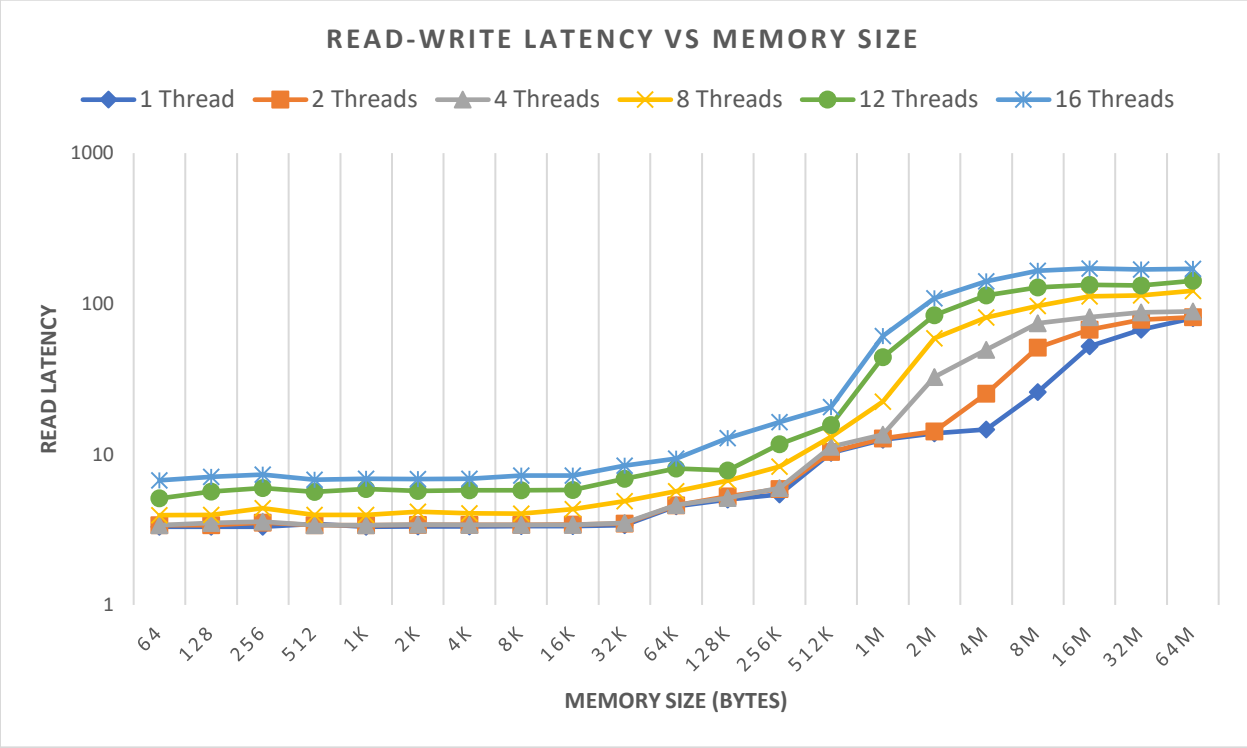

Figure 8 – Read-Write Latency with several threads(log scale)

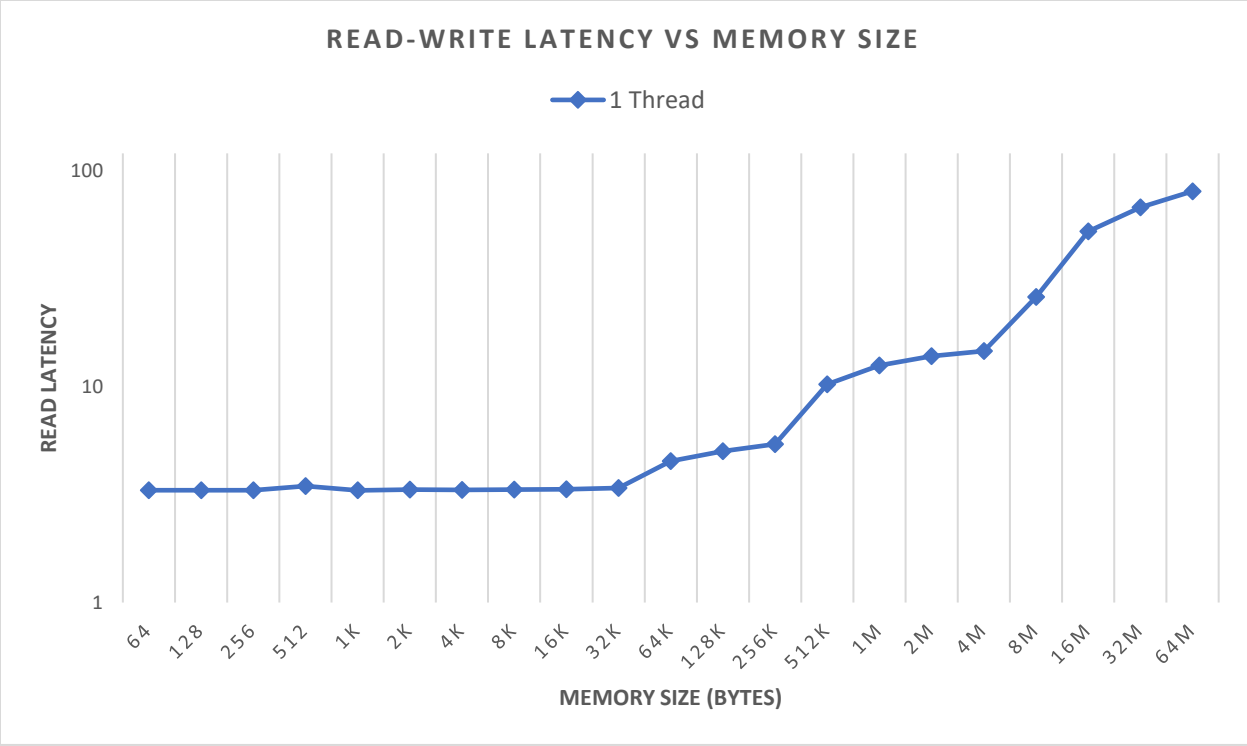

Figure 9 – Read-Write Latency with 1 thread(log scale)

3. Analysis

Read Only

The significant regions of interest are the places where there is an abrupt jump in the latency from the normal. These regions signify that there is more memory allocated than the cache hierarchy components can handle. Figure 2 shows read latencies when the program was run on 1 thread with different memories allocated. Table 1 show the observed latencies. The small jump from about ~3ns to ~5ns at 32KB denotes that the L1 cache is full and accesses fetch data from L2, which is slightly slower. This also implies that the L1 cache size is about 32KB. The next abrupt jump is at 256KB from ~6ns to ~13ns. This denotes that the L1 and L2 caches are filled with data and that the size of L2 is about 256KB. After this point the L3 is also accessed to retrieve any data that is not present in the faster, local caches(L1 and L2). The next interesting region is around 32MB where the latency sees a large jump from ~30ns to ~53ns and then to 116.279ns. 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.

With this observation a reasonable estimate can be made about the cache sizes of the processor system. In this case they are**, L1 – 32KB, L2 – 256KB, and L3 – 32MB**. This is an accurate observation since the actual cache sizes of the machine used to conduct this experiment are **L1 – 32KB, L2 – 256KB, and L3 – 30MB**. 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.

A similar trend can be observed on Machine 2 but this time the differences due to multithreading are more significant and with different latency numbers. Observing Figure 6 and 9 we can estimate that the **L1 size is 32 KB, L2 size is 256KB and L3 size is around 8MB**. This is close to the actual specifications shown in Table 1.

Multithreading results in more interesting results. Since memory is allocated for each thread, the caches are filled sooner. The read latencies until almost 1MB remain the same for all threads. This could be because each core has its own local L1 and L2 caches. Since the L3 is shared and only read, there seems to be a lot of shared data that is read by several threads. Once the memory size increases beyond 1MB, the latencies 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. Once L3 is filled the accesses go to main memory which increases latencies significantly. The reason for the latencies not increasing when the memory size reaches exactly the L3 size is because this machine has 2 L3 caches, each shared among 12 cores of the 24-core processor.

Multithreading in case of machine two shows a linear increase in latency as the number of threads increases and when memories are larger than L2. Since a single L3 is shared among all cores, the latencies rise up pretty fast when compared to machine 1 where two L3 caches are present, both of larger sizes.

Read/Write

The shape of the read/write graph is similar to the read only graph except the latencies are noticeably higher as writing to memory is more time consuming than reading. The regions that distinguish the different caches accessed are also clear in this case. Multithreading read/write program shows a similar result but the effects of multithreading are more defined here. False sharing is a dreaded consequence of any multithreaded program where each core writes to the same cache line in its local cache. This makes the cache invalidate other copies and update memory before execution can proceed. This introduces the significant amount of cycles to the program making it a lot slower than the single-threaded case and extremely slower than the read-only case. Although the difference in very small, the latencies after 256KB memory size branch out with the number of threads. The latencies with 16 threads are clearly higher than the others and this trend becomes clearer as the memory size increases.

Machine 2 produces even more well pronounced results compared to Machine 1 and even Machine 2's read latencies. From figure 8 it can be observed that the number of threads have a clearly distinguishable latency variation. Multithreading in this case shows how latencies monotonically increase with more threads and show the effect of false sharing clearly than Machine 1. False sharing plays a more deteriorating role in case of machine 2 because all the threads end up sharing the same lines in the smaller L3 cache. This is evident from the larger runtimes compared to machine 1 for the same number of threads.

Both read and read/write graphs seem stable with the read/write being slightly more stable with multithreading. The latencies are slightly higher over the entire graph in read/write because writes are costlier than reads. We can see the effects of multithreading clearly in the read/write case as the allocated memory becomes larger than the L3 cache(30MB in case in Machine 1 or 9MB in case of Machine 2) .

Other than latency, we can also predict the write hit mode of the cache. Since the write latencies are almost the same as read(although slightly higher) we can conclude that the caches are write-back. Sanity

check for the program is the observation itself. If the cache sizes estimated using this program are close to the actual hardware specification, the code is correct. Observing the single-threaded latencies on another machine(Intel Core i7-8750H – Machine 2) with **L1 size – 32KB, L2 size – 256KB, and L3 size – 9MB** has resulted in the plot in Figure 6 and Figure 9.

4. Extra Credits

The above program can also be used for sequential or strided memory access. The graphs below show this kind of memory access with increasing stride sizes starting from 0. Zero stride means it is a sequential access of continuous elements stored in the memory. This kind of access is very easy going on the caches as spatial locality helps in significantly reducing access times of sequential elements. As the stride size increases, the number of bytes between two accessed elements increases, and the latency keeps increasing and saturates at a number after which stride size has little to no impact. This is because the accesses are not sequential anymore and the lines are retrieved from a higher level in the memory hierarchy. But this approach is better than a completely random or a pseudo-random approach of accessing elements as it has a slightly higher opportunity to hit one of the caches because the accesses are fixed distances apart. More threads follow the same trends with slightly higher latencies than single threaded. This experiment was again done on both machines. Similar to the random-access cases discussed before, the effects of multithreading is more evident in machine 2, with a small L3 caches, than machine 1 with two larger L3 caches. Both cases experience a saturation in the latencies as the stride sizes increase beyond a certain value.

Machine 1

Figures 10 and 11 shows the read only latencies and Figures 12 and 13 shows the read-write latencies, all four with strided accesses and several threads.

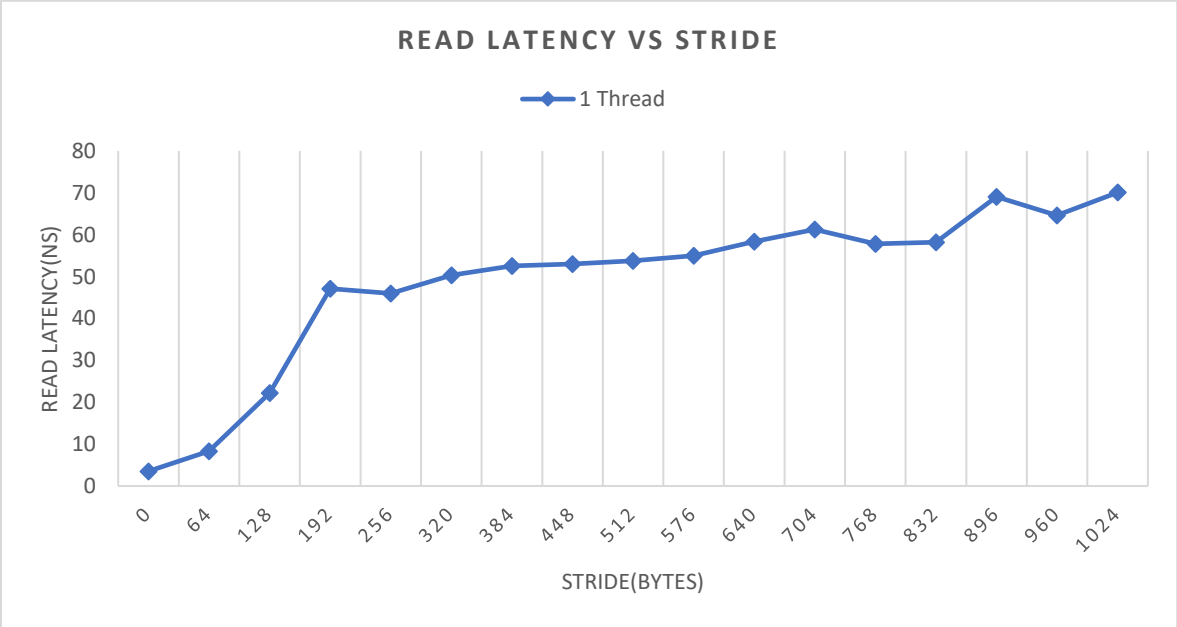

Figure 10 – Read latencies – Single Threaded – Strided access

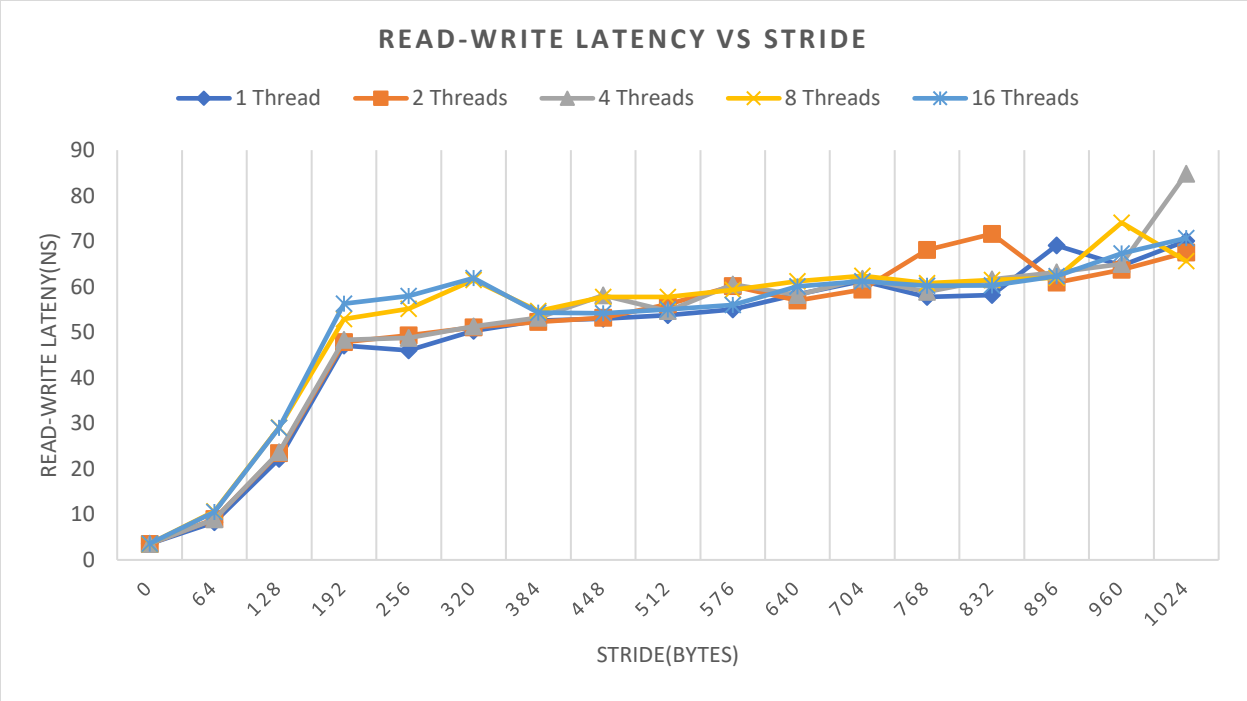

Figure 11 – Read latencies – Multithreaded– Strided access

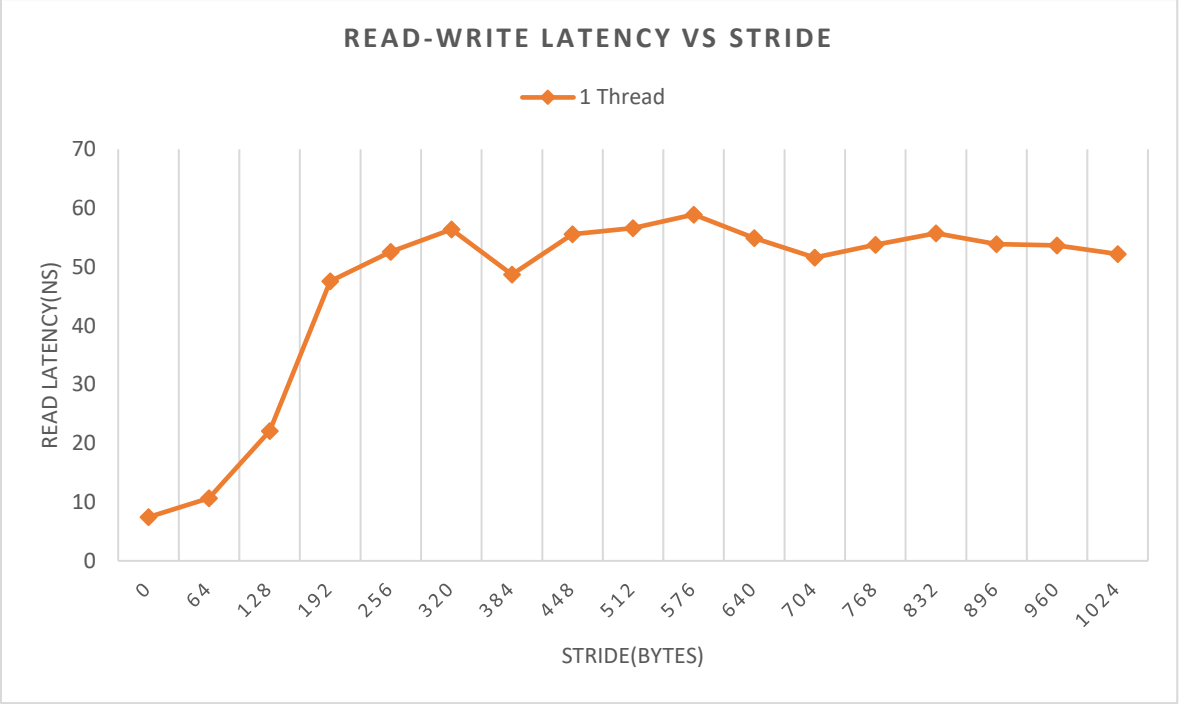

Figure 12 – Read-Write latencies – Single Threaded – Strided access

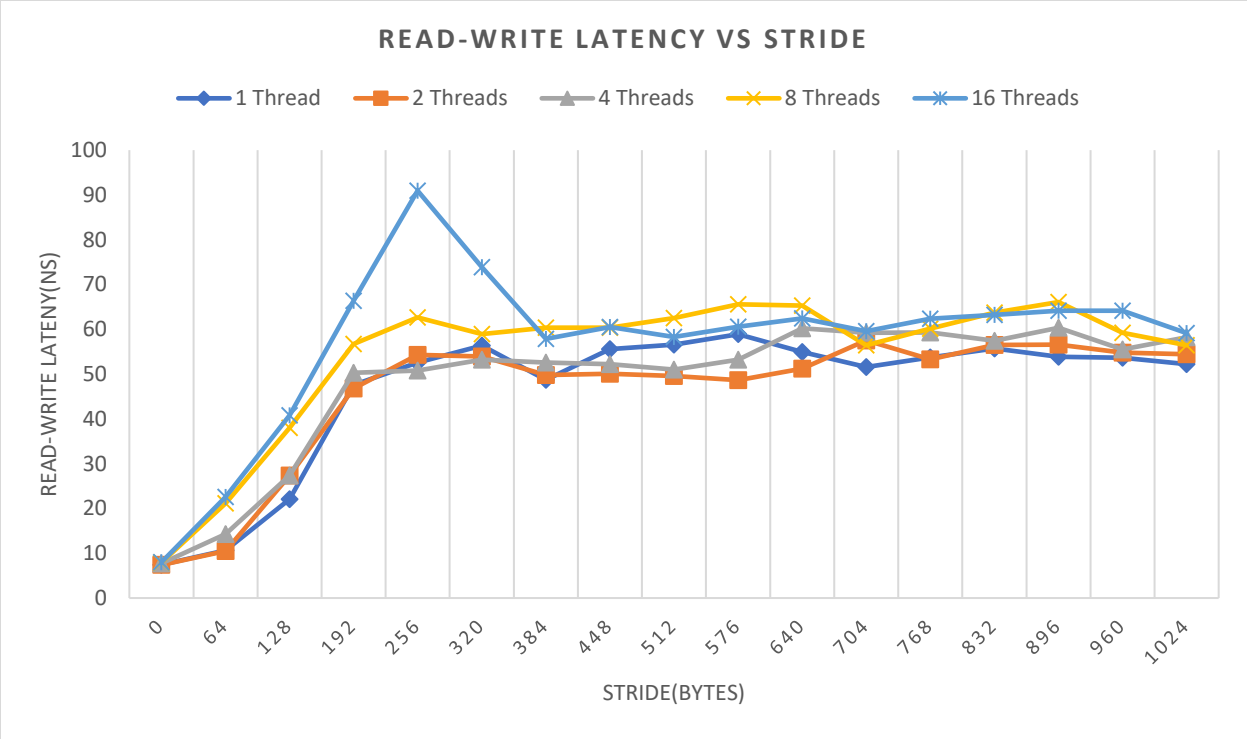

Figure 13 – Read-Write latencies – Multithreaded – Strided access

Machine 2

Figures 14 and 15 shows the read only latencies and Figures 16 and 17 shows the read-write latencies, all four with strided accesses and several threads for Machine 2.

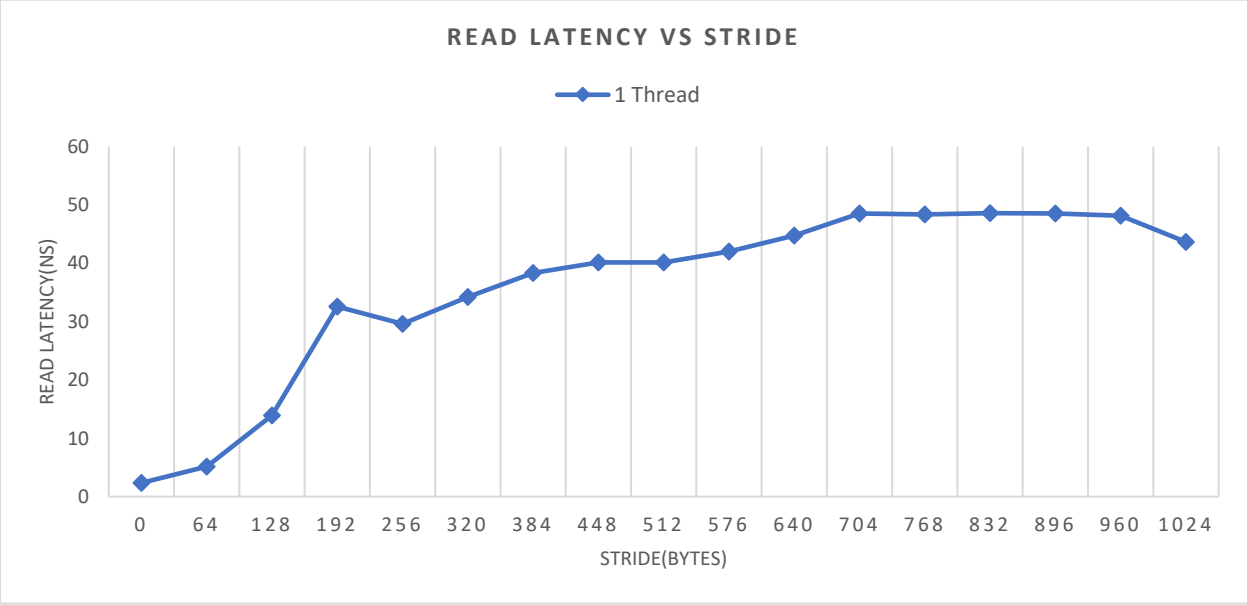

Figure 14 – Read latencies – Single-threaded – Strided access

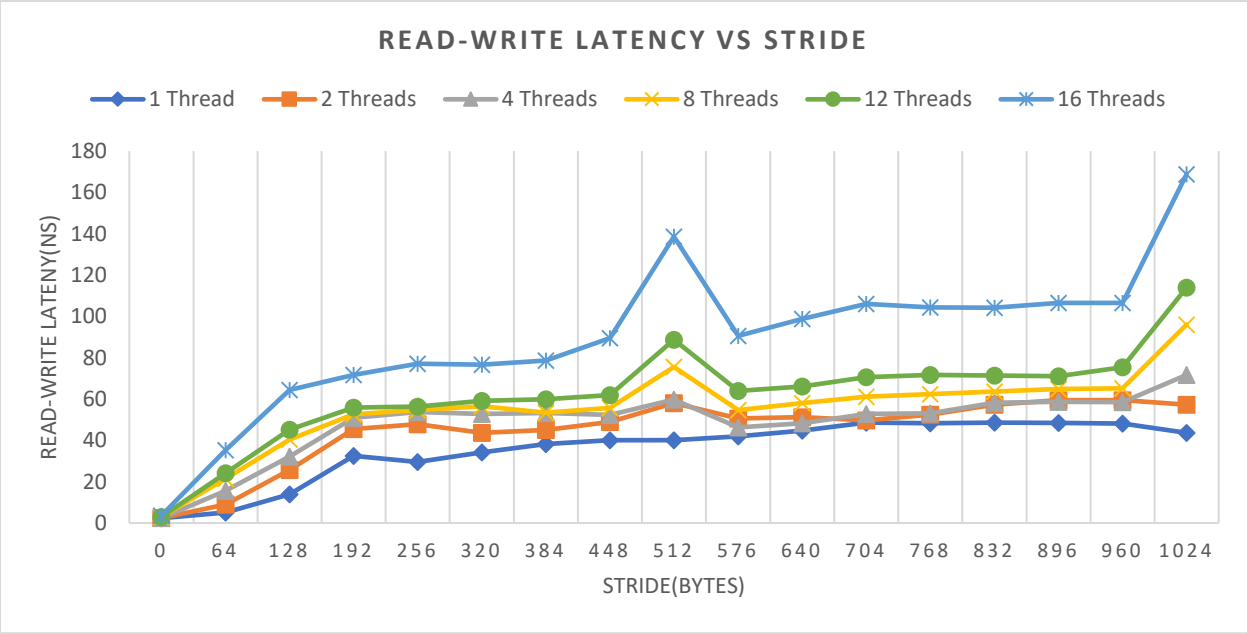

Figure 15 – Read latencies – Multithreaded – Strided access

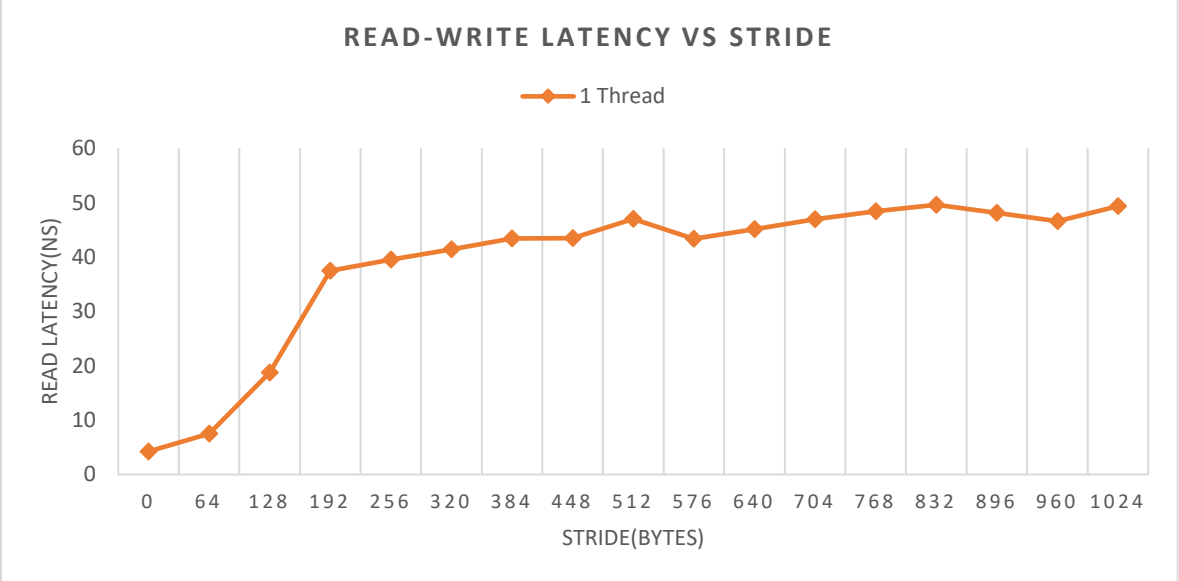

Figure 16 – Read-Write latencies – Single-threaded – Strided access

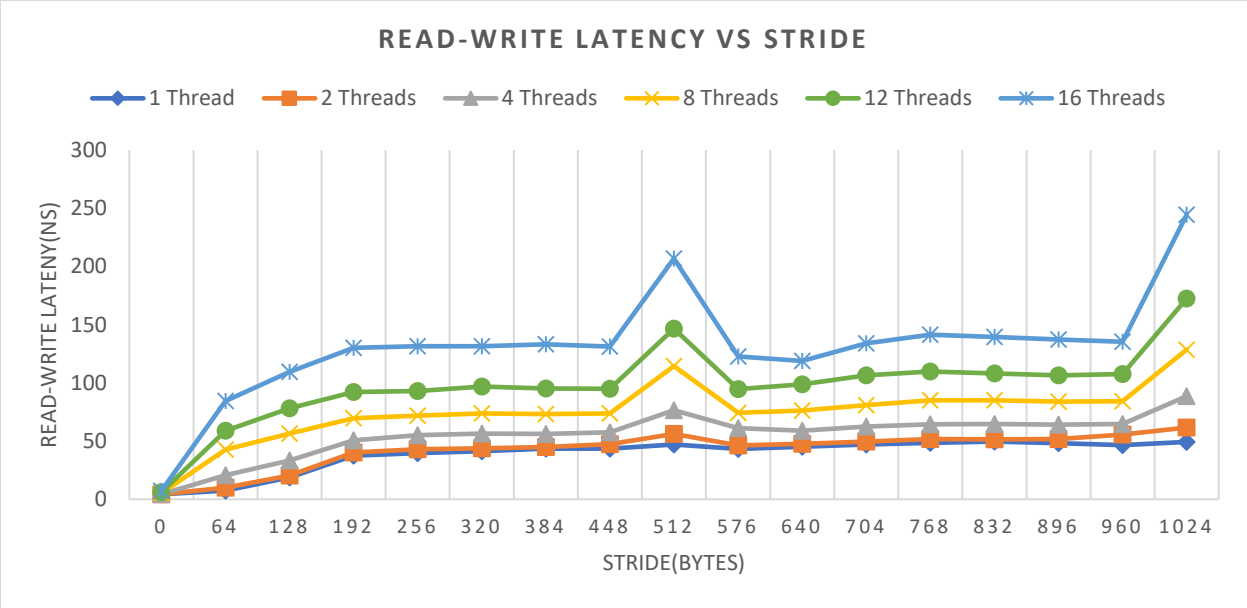

Figure 16 – Read-Write latencies – Multi-threaded – Strided access