2023

Course Project 1: Cache and Memory Performance Profiling

ECSE 4320/6320 ADVANCED COMPUTER SYSTEMS

BARRY, ABDOULA

INSTITUTION: RENSSELAER POLYTECHNIC INSTITUTE

RIN: 662006417

The objective of this project is to gain deeper understanding on cache and memory hierarchy. Following will be a set of experiments designed to quantitatively reveal the following:

(1) The read/write latency of cache and main memory when the queue length is zero (i.e., zero queuing delay)

(2) The maximum bandwidth of the main memory under different data access granularity (i.e., 64B, 256B, 1024B) and different read vs. write intensity ratio (i.e., read-only, write-only, 70:30 ratio, 50:50 ratio)

(3) The trade-off between read/write latency and throughput of the main memory to demonstrate what the queuing theory predicts

(4) The impact of cache miss ratio on the software speed performance (the software is supposed to execute relatively light computations such as multiplication)

(5) The impact of TLB table miss ratio on the software speed performance (again, the software is supposed to execute relatively light computations such as multiplication)

Tool(s):

1- **Intel Memory Latency Checker**
2- **C/C++ (Programming Language)**

Computer Components and Specifications:

System type: 64-bit operating system, x64-based processor

Operating System: Windows 10 Home 64-bit (10.0 Build 19045)

BIOS: F4

Processor: Intel® Core™ i5-10400F CPU @ 2.90GHz (12 CPUs)

Sockets: 1

Cores: 6

Memory: 16384MB RAM; DDR4

L1 cache: 384KB

L2 cache: 1.5MB

L3 cache: 12.0MB

Storage: 1TB

Form factor: DIMM

Speed: 2400 MHz

(1) The read/write latency of cache and main memory when the queue length is zero (i.e., zero queueing delay)

The main thing to note in this prompt is that the queue length is zero. This implies that the system is not executing any processes or requesting information from cache or main memory. The system is idle and can be inferred to be in a state of "zero queueing delay". In other words, this would just be a measurement of the idle latency of the system. We can do this simply with the memory latency checker tool (MLC).

Using the command "mlc –idle_latency" in the command prompt returns the idle memory latency of the platform. See figure 1 for a closer look.

```
C:\Users\abdou\Desktop\FALL23\AdvancedCSYS\mlc\Windows>mlc --idle_latency
Intel(R) Memory Latency Checker - v3.10
Command line parameters: --idle_latency

Using buffer size of 200.000MiB
Each iteration took 198.9 base frequency clocks (       68.6    ns)
```

Figure 1: Command prompt – idle memory latency

As you can see for my system, the idle memory latency is 68.6 ns approximately.

More specifically, to change how much of cache or main memory latency is being measured, I adjusted the buffer size in MiB using the parameter "-bn":

a) mlc –idle_latency
   a. Output:

0.05	1
0.098	3.1
1	12.6
5	14.2
10	23.9
15	46.9
20	62.3
50	67.8
124	68.4
256	68.9
512	68.9
1024	69.1
2048	69.9
4096	70.7

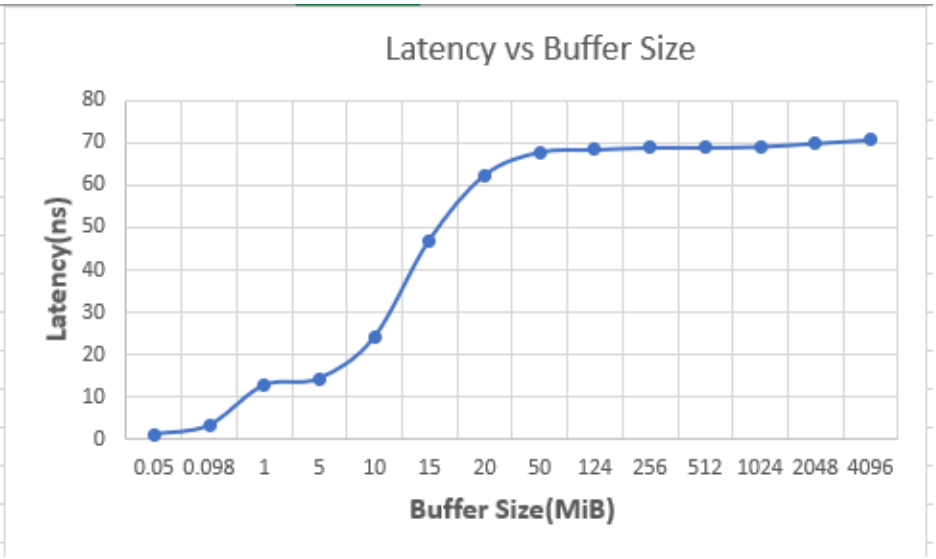

Figure 2: Latency/Buffer Size

The relationship between latency and buffer size shown in figure 2 is such that as the buffer size increases the latency does as well. It is known that the L1 cache size of this system is only 384 KB. Therefore, when the buffer size is smaller than that, memory access is likely happening within the L1 cache. This is supported by the data as this is when latency is the lowest (.05 ns). At a certain point, when the buffer size becomes too large, there is a latency transition point from memory accesses in the cache into main memory. This transition is seen around the 10-15(MiB) buffer mark. From that point, it can be reasonably inferred that access is coming from the main memory as latency sharply increases and plateaus at around 69.9 ns in my system.

(2) The maximum bandwidth of the main memory under different data access granularity (i.e., 64B, 256B, 1024B) and different read vs. write intensity ratio (i.e., read-only, write-only, 70:30 ratio, 50:50 ratio)

To examine this prompt and give it context within the scope of the MLC tool, I used the "—peak_injection_bandwidth" command in mlc which is essentially the same as the maximum bandwidth. Peak injection bandwidth is measured by generating requests from the core at the fastest possible rate. Furthermore, by adding the parameter "-ln", where n is the Byte size, I was able to change the stride length, effectively changing the granularity of the data.

a) mlc –peak_injection_bandwidth
    a. The default granularity is 64B so no "-l" parameter needed
    b. Output:

```
C:\Users\abdou\Desktop\FALL23\AdvancedCSYS\mlc\Windows>mlc --peak_injection_bandwidth
Intel(R) Memory Latency Checker - v3.10
Command line parameters: --peak_injection_bandwidth

Using buffer size of 100.000MiB/thread for reads and an additional 100.000MiB/thread for writes

Measuring Peak Injection Memory Bandwidths for the system
Bandwidths are in MB/sec (1 MB/sec = 1,000,000 Bytes/sec)
Using all the threads from each core if Hyper-threading is enabled
Using traffic with the following read-write ratios
ALL Reads        :      32452.1
3:1 Reads-Writes :      30264.7
2:1 Reads-Writes :      29777.2
1:1 Reads-Writes :      29910.9
Stream-triad like:      30246.3
```

Figure 3: MLC maximum bandwidth at 64B granularity

b) mlc –peak_injection_bandwidth -l128
    a. Output:

```
C:\Users\abdou\Desktop\FALL23\AdvancedCSYS\mlc\Windows>mlc --peak_injection_bandwidth -l128
Intel(R) Memory Latency Checker - v3.10
Command line parameters: --peak_injection_bandwidth -l128

Using buffer size of 100.000MiB/thread for reads and an additional 100.000MiB/thread for writes

Measuring Peak Injection Memory Bandwidths for the system
Bandwidths are in MB/sec (1 MB/sec = 1,000,000 Bytes/sec)
Using all the threads from each core if Hyper-threading is enabled
Using traffic with the following read-write ratios
ALL Reads        :      26993.3
3:1 Reads-Writes :      26335.7
2:1 Reads-Writes :      26066.5
1:1 Reads-Writes :      26419.2
Stream-triad like:      25420.2
```

Figure 4: MLC maximum bandwidth at 128B granularity

c) mlc –peak_injection_bandwidth -l256
   a. Output:

```
C:\Users\abdou\Desktop\FALL23\AdvancedCSYS\mlc\Windows>mlc --peak_injection_bandwidth -l256
Intel(R) Memory Latency Checker - v3.10
Command line parameters: --peak_injection_bandwidth -l256

Using buffer size of 100.000MiB/thread for reads and an additional 100.000MiB/thread for writes

Measuring Peak Injection Memory Bandwidths for the system
Bandwidths are in MB/sec (1 MB/sec = 1,000,000 Bytes/sec)
Using all the threads from each core if Hyper-threading is enabled
Using traffic with the following read-write ratios
ALL Reads        :      22358.5
3:1 Reads-Writes :      24290.9
2:1 Reads-Writes :      24824.3
1:1 Reads-Writes :      25182.8
Stream-triad like:      23527.9
```

Figure 5: MLC maximum bandwidth at 256B granularity

d) mlc –peak_injection_bandwidth -l512
   a. Output:

```
C:\Users\abdou\Desktop\FALL23\AdvancedCSYS\mlc\Windows>mlc --peak_injection_bandwidth -l512
Intel(R) Memory Latency Checker - v3.10
Command line parameters: --peak_injection_bandwidth -l512

Using buffer size of 100.000MiB/thread for reads and an additional 100.000MiB/thread for writes

Measuring Peak Injection Memory Bandwidths for the system
Bandwidths are in MB/sec (1 MB/sec = 1,000,000 Bytes/sec)
Using all the threads from each core if Hyper-threading is enabled
Using traffic with the following read-write ratios
ALL Reads        :      15657.6
3:1 Reads-Writes :      19576.7
2:1 Reads-Writes :      20195.1
1:1 Reads-Writes :      21332.0
Stream-triad like:      18584.2
```

Figure 6: MLC maximum bandwidth at 512B granularity

e) mlc –peak_injection_bandwidth -l1024

a. Output:

```
C:\Users\abdou\Desktop\FALL23\AdvancedCSYS\mlc\Windows>mlc --peak_injection_bandwidth -l1024
Intel(R) Memory Latency Checker - v3.10
Command line parameters: --peak_injection_bandwidth -l1024

Using buffer size of 100.000MiB/thread for reads and an additional 100.000MiB/thread for writes

Measuring Peak Injection Memory Bandwidths for the system
Bandwidths are in MB/sec (1 MB/sec = 1,000,000 Bytes/sec)
Using all the threads from each core if Hyper-threading is enabled
Using traffic with the following read-write ratios
ALL Reads        :      12414.5
3:1 Reads-Writes :      17865.5
2:1 Reads-Writes :      18875.6
1:1 Reads-Writes :      18518.7
Stream-triad like:      15156.9
```

Figure 7: MLC maximum bandwidth at 1024B granularity

f) Output in graph format:

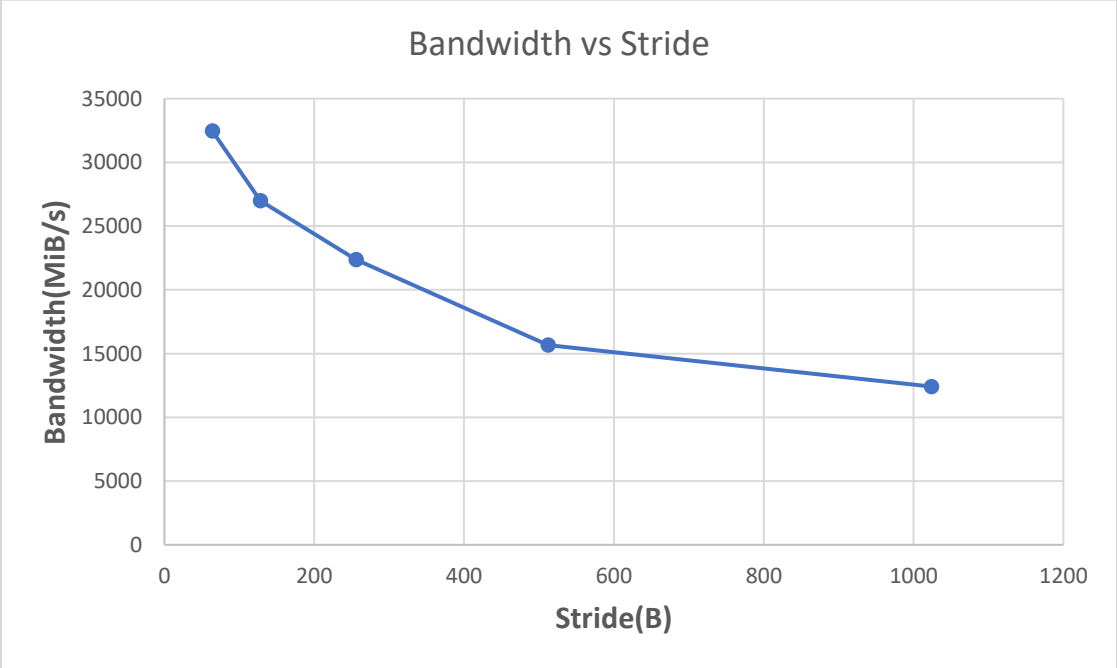

Figure 8: Bandwidth/Data Access Granularity

Increasing the stride length leads to worse memory locality and the memory system cannot fetch data as efficiently when compared to lower stride lengths. We can see this when we compare the outputs of each output granularity. As the stride length increases, the memory accesses become more scattered in memory and leads to more cache misses and higher memory access latencies. These factors contribute to the lower read and write bandwidth numbers seen in the output figures.

Using the “mlc –max_bandwidth” command, the tool will yield the maximum bandwidth of memory with different read/write ratios.

g) mlc –max_bandwidth
   a. Output:

```
C:\Users\abdou\Desktop\FALL23\AdvancedCSYS\mlc\Windows>mlc --max_bandwidth
Intel(R) Memory Latency Checker - v3.10
Command line parameters: --max_bandwidth

Using buffer size of 100.000MiB/thread for reads and an additional 100.000MiB/thread for writes

Measuring Maximum Memory Bandwidths for the system
Will take several minutes to complete as multiple injection rates will be tried to get the best bandwi
dth
Bandwidths are in MB/sec (1 MB/sec = 1,000,000 Bytes/sec)
Using all the threads from each core if Hyper-threading is enabled
Using traffic with the following read-write ratios
ALL Reads        :      31677.15
3:1 Reads-Writes :      30216.96
2:1 Reads-Writes :      29328.03
1:1 Reads-Writes :      29619.34
Stream-triad like:      29984.90
```

Figure 9: MLC maximum bandwidth

h) Graph format:

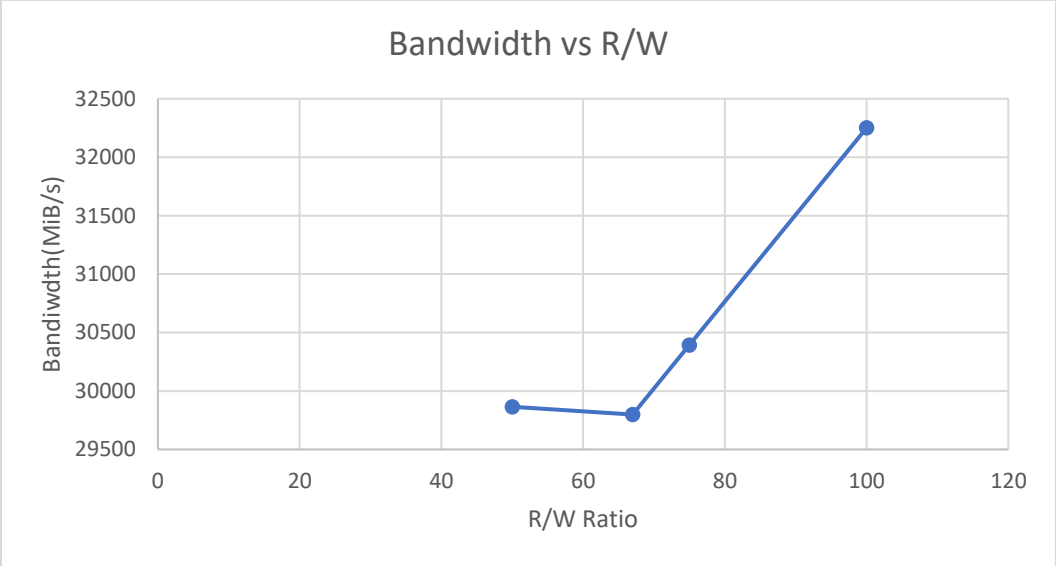

Figure 10: MLC max bandwidth / Read:Write Ratio

The data reveals that as the system shifts from an equal distribution of read and write into 100% read, the bandwidth increases. This is evidence that “writes” are slower than “reads” fundamentally. In read-heavy workloads, the caches are more effective at prefetching and storing frequently accessed data which boosts bandwidth. Conversely, write operations must maintain cache coherence which adds significant overhead that reduces the bandwidth.

(3) The trade-off between read/write latency and throughput of the main memory to demonstrate what the queuing theory predicts

To examine this trade-off, I simply conducted the load latency command to analyze the relationship in graph format.

a) mlc –loaded_latency
    a. Output:

```
C:\Users\abdou\Desktop\FALL23\AdvancedCSYS\mlc\Windows>mlc --loaded_latency
Intel(R) Memory Latency Checker - v3.10
Command line parameters: --loaded_latency

Using buffer size of 100.000MiB/thread for reads and an additional 100.000MiB/thread for writes

Measuring Loaded Latencies for the system
Using all the threads from each core if Hyper-threading is enabled
Using Read-only traffic type
Inject  Latency Bandwidth
Delay   (ns)    MB/sec
==========================
 00000  237.96    31899.1
 00002  240.66    31914.8
 00008  230.44    31708.4
 00015  220.98    31805.2
 00050  191.95    31241.7
 00100  133.14    28888.6
 00200  102.06    21339.1
 00300   91.43    15604.7
 00400   85.25    12260.1
 00500   85.57    10107.4
 00700   82.67     7562.6
 01000   81.49     5600.2
 01300   80.75     4526.1
 01700   81.02     3654.4
 02500   80.77     2747.2
 03500   79.56     2208.5
 05000   80.58     1772.7
 09000   91.60     1226.5
 20000  100.28      865.6
```

Figure 10: MLC loaded latency

b) Graph format:

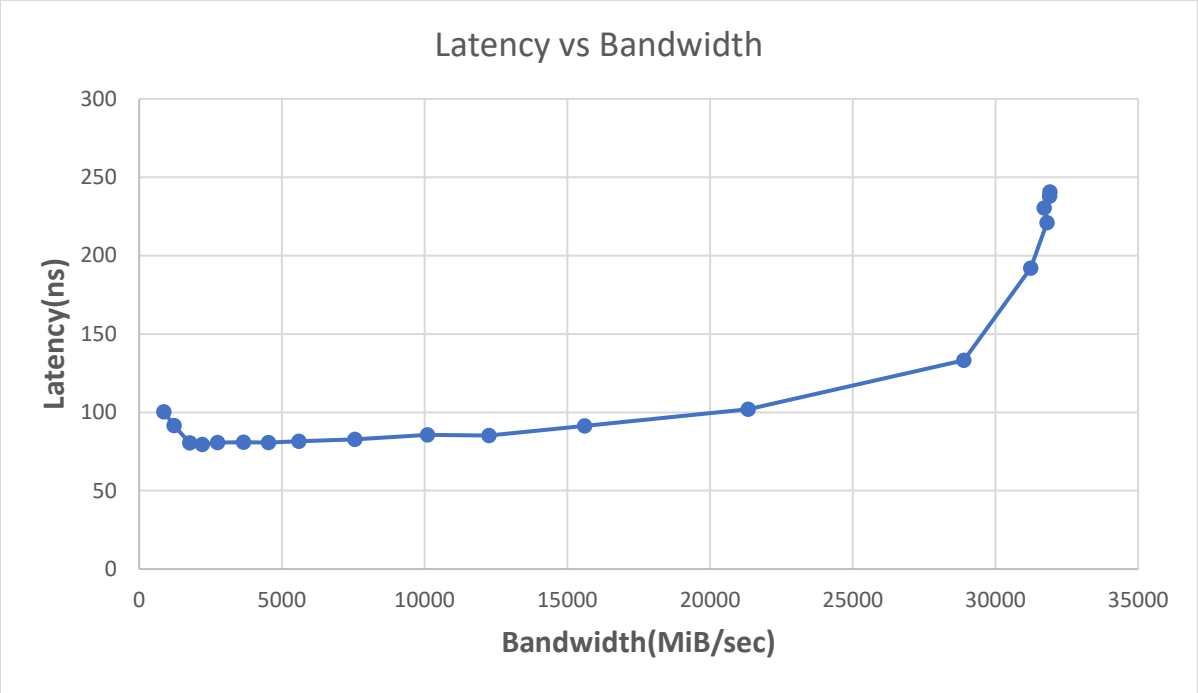

Figure 11: Latency / Bandwidth

The output of the "—loaded_latency" operation tells us a lot about the latency and bandwidth trade-off. As bandwidth reaches its maximum, latency sharply increases as a result. The queuing theory predicts that as a system experiences a higher arrival rate of requests, the average latency will increase. Reducing latency involves minimizing the queueing delay and optimizing the system for quick access to data. However, this comes at a cost to the bandwidth as seen in figure 11.

(4) The impact of cache miss ratio on the software speed performance (the software is supposed to execute relatively light computations such as multiplication)

The function of the C program (found in the part 4 folder) is to allocate an array of a set size and utilize the concept of data access granularity and stride seen in the mlc tool output analysis prior. The data access loops have a set stride size to influence the cache miss ratio of the program. If you access the array with 2 different granularities or stride patterns, memory will be accessed at different intervals. Hence, disrupting the spatial locality of the data and decreasing cache hit rate. The program also measures the time it takes to loop for each pattern and outputs the time in nanoseconds. See figure 12 below for terminal output.

```
PS C:\Users\abdou\Desktop\projects\helloworld> g++ -o main2 cachemiss.c
PS C:\Users\abdou\Desktop\projects\helloworld> ./main2
Stride Access Time: 2900 ns
Stride Access Time (Doubled Stride): 4100 ns
PS C:\Users\abdou\Desktop\projects\helloworld> ./main2
Stride Access Time: 2700 ns
Stride Access Time (Doubled Stride): 5100 ns
PS C:\Users\abdou\Desktop\projects\helloworld> ./main2
Stride Access Time: 2900 ns
Stride Access Time (Doubled Stride): 4700 ns
PS C:\Users\abdou\Desktop\projects\helloworld> ./main2
Stride Access Time: 2800 ns
Stride Access Time (Doubled Stride): 4700 ns
PS C:\Users\abdou\Desktop\projects\helloworld> ./main2
Stride Access Time: 2800 ns
Stride Access Time (Doubled Stride): 5300 ns
PS C:\Users\abdou\Desktop\projects\helloworld> ./main2
Stride Access Time: 2400 ns
Stride Access Time (Doubled Stride): 6000 ns
PS C:\Users\abdou\Desktop\projects\helloworld> ./main2
Stride Access Time: 2900 ns
Stride Access Time (Doubled Stride): 5000 ns
PS C:\Users\abdou\Desktop\projects\helloworld>
```

Figure 12: Terminal Output of cache miss program

Output in Graph format:

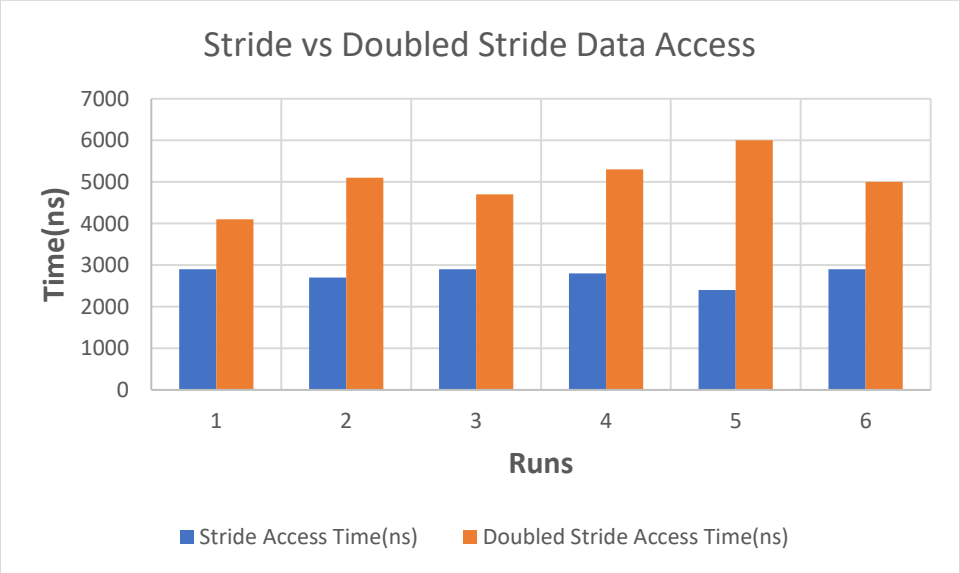

Figure 13: Histogram view of cache miss program

As predicted, when the stride pattern is of a higher granularity, the cache miss rate increases resulting in the longer time execution of the loop. This emphasizes the importance of data locality and cache usage in programming.

(5) The impact of TLB table miss ratio on the software speed performance (again, the software is supposed to execute relatively light computations such as multiplication)

The function of the C program (found in the part 5 folder) is to allocate an array with a set number of elements and access the memory in 2 different ways, sequentially and randomly. It also measures the time taken to do each kind of memory access to see what it reveals about the TLB miss ratio. The reason I chose these 2 methods is because based on topics in lecture, the random-access method is almost 100 percent guaranteed to result in a higher TLB miss ratio compared to the sequential method as this is a non-sequential memory access. This results in the TLB being less effective at predicting and loading non-sequential memory pages. The time is measured in nanoseconds. Figure 14 shows terminal outputs from the program.

```
PS C:\Users\abdou\Desktop\projects\helloworld> g++ -o main1 tlbmiss_1.c
PS C:\Users\abdou\Desktop\projects\helloworld> ./main1
Sequential Access Time: 2431700 ns
Random Access Time: 18093900 ns
PS C:\Users\abdou\Desktop\projects\helloworld> ./main1
Sequential Access Time: 2552700 ns
Random Access Time: 18119000 ns
PS C:\Users\abdou\Desktop\projects\helloworld> ./main1
Sequential Access Time: 2333000 ns
Random Access Time: 18139000 ns
PS C:\Users\abdou\Desktop\projects\helloworld> ./main1
Sequential Access Time: 2245900 ns
Random Access Time: 18230900 ns
PS C:\Users\abdou\Desktop\projects\helloworld> ./main1
Sequential Access Time: 2524100 ns
Random Access Time: 18907800 ns
PS C:\Users\abdou\Desktop\projects\helloworld> ./main1
Sequential Access Time: 2510500 ns
Random Access Time: 18032100 ns
PS C:\Users\abdou\Desktop\projects\helloworld>
```

Figure 14: Terminal Output of tlbmiss program

Output in Graph Format:

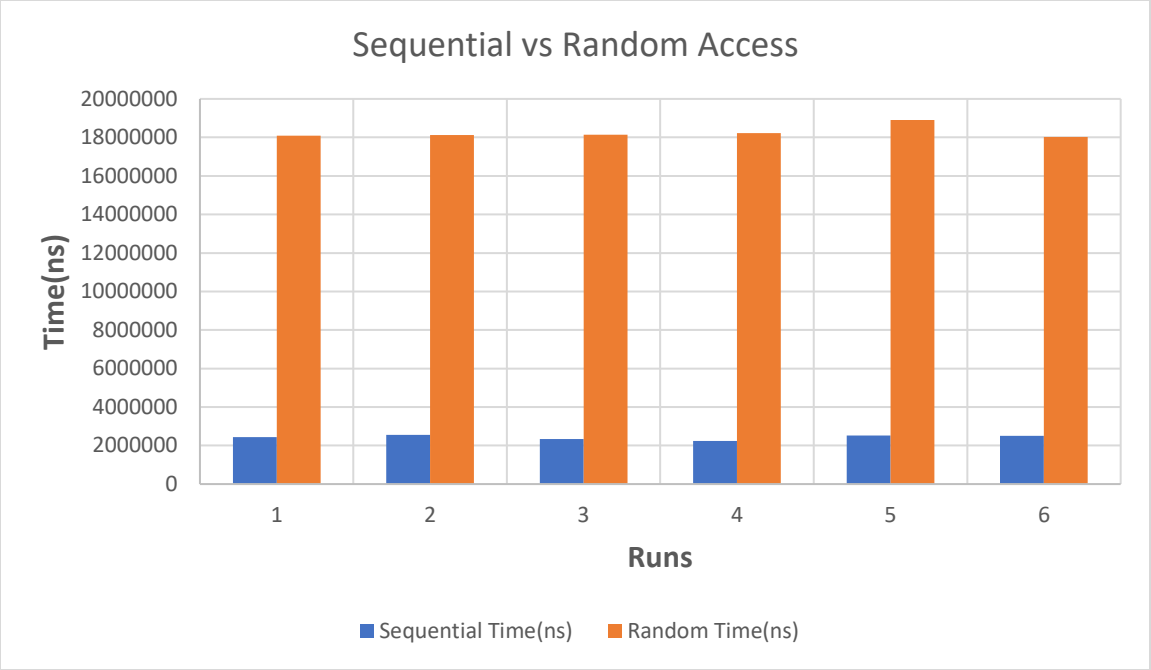

Figure 15: Histogram view of tlbmiss program

It's clear from the above data that the TLB miss rate for random access is much higher represented by the time in ns for each run. As a result, optimizing programs for minimal TLB misses should be a main priority in programming.

Following these experiments, to take a closer look at the cache and main memory, tools such as Valgrind or the Linux perf command which were not available to me, are extremely helpful.