Danielle Nnorom
Dennise Kwarteng

Read/Write Latency Deliverable- Task 1:

The read/ write latency of cache and main memory when the queue length is 0 should be in such a way that the computer is faster at getting access to the cache than the main memory, whether that be DRAM or another format. For our program we used the differences between a variable created on the stack and a variable on the heap to highlight some of the difference between the cache and the main memory.

Both the variables were of the type int arrays that were 1000 items in size. We used the built in C++ chronous extension in order to get the measurements between the different parts of the code running in order to figure out the latency for the heap and stack variables. We tested reading and writing to each array separately to see the latencies for the two types of memory interactions. What we found from the experiment was that on average Reading from the cache took around 3.008e-05s while reading from the heap/ main memory took around 2.884e-05s, as Figure 1 can show.

This result was unexpected as from the theory it would make more sense if cache took a shorter time to read than main memory did. The reason for this discrepancy may lie in the way that the CPUs of the computer handle pointers. The variable that holds the pointer to the data is also positioned in the cache of the computer, due to it being a local variable. It is possible that because of this there was less of a chance that the actual data read needed to move between the cache levels resulting in faster reads, due to the program's size. However, this also meant that any time the CPU may have needed to assure cache consistency did not apply to the data in our heap variable as it was directly connected to a space in memory.

The write time for the cache ended up being an average of 2.66e-06s while the average time for the main memory was 2.76e-06s. This result makes sense as there are less steps that are required when trying to access memory from the cache than in main memory. So although with a relatively simple program that was used there is still a difference between accessing memory from the cache and from main memory.

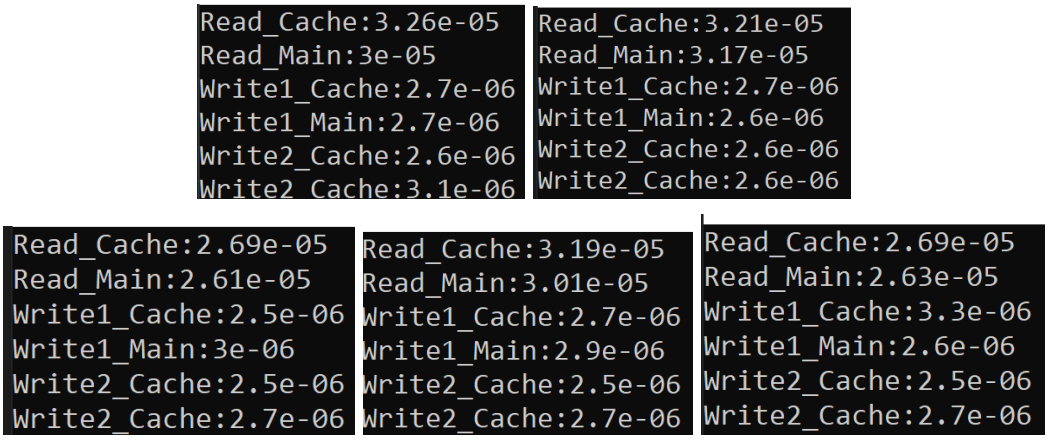

Figure 1: Output Data from the Task 1 Program. Last line is actually "Write2_Main:xxx" data

Danielle Nnorom
Dennise Kwarteng

Bandwidth at different read/write ratios & buffer sizes - Task 2:

Based on the idea that the larger the buffer size for a program the more information can be conveyed at a time leading to better bandwidth in this experiment it was expected that the larger buffer sizes would then generate the lower traffic within the bandwidth. To do this the Intel Memory Latency Checker (mlc) was used in its max_bandwidth format to record the bandwidth of the different read/write ratios with different buffer sizes.

In general the trend for the bandwidth is as the theory suggests. As we increase the buffer size the traffic of the read-write ratios go down. This is because more messages can be processed at the same time and not wait in the queue buffer for a s long. This improves the overall process due to there being more slots for the data to travel in at a time, resulting in the lower traffic times as the buffer size is increased as can be seen in Figures 2 - 6. The same program was run for all the tests during this experiment.

```
adduser@LAPTOP-RPI:/mnt/c/Users/Danielle/Downloads/mlc_v3.10/Windows$ ./mlc.exe --max_bandwidth -b1
Intel(R) Memory Latency Checker - v3.10
Command line parameters: --max_bandwidth -b1

Using buffer size of 0.001MiB/thread for reads and an additional 0.001MiB/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 bandwidth
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        :      202799.49
3:1 Reads-Writes :      506680.35
2:1 Reads-Writes :      510997.97
1:1 Reads-Writes :      306060.64
Stream-triad like:      133994.27
```

Figure 2: Bandwidth measurements via mlc at 1kB

```
Intel(R) Memory Latency Checker - v3.10
Command line parameters: --max_bandwidth -b100

Using buffer size of 0.098MiB/thread for reads and an additional 0.098MiB/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 bandwidth
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        :      247078.24
3:1 Reads-Writes :      318157.09
2:1 Reads-Writes :      295067.82
1:1 Reads-Writes :      453002.42
Stream-triad like:      96255.03
```

Figure 3: Bandwidth measurements via mlc at 100kB

```
adduser@LAPTOP-RPI:/mnt/c/Users/Danielle/Downloads/mlc_v3.10/Windows$ ./mlc.exe --max_bandwidth -b1m
Intel(R) Memory Latency Checker - v3.10
Command line parameters: --max_bandwidth -b1m

Using buffer size of 1.000MiB/thread for reads and an additional 1.000MiB/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 bandwidth
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        :      136735.54
3:1 Reads-Writes :      91764.60
2:1 Reads-Writes :      67708.35
1:1 Reads-Writes :      213826.43
Stream-triad like:      65666.61
```

Figure 4: Bandwidth measurements via mlc at 1MB

```
adduser@LAPTOP-RPI:/mnt/c/Users/Danielle/Downloads/mlc_v3.10/Windows$ ./mlc.exe --max_bandwidth -b10m
Intel(R) Memory Latency Checker - v3.10
Command line parameters: --max_bandwidth -b10m

Using buffer size of 10.000MiB/thread for reads and an additional 10.000MiB/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 bandwidth
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        :      29891.19
3:1 Reads-Writes :      27694.06
2:1 Reads-Writes :      28393.87
1:1 Reads-Writes :      29003.29
Stream-triad like:      28813.37
```

Figure 5: Bandwidth measurements via mlc at 10MB

```
adduser@LAPTOP-RPI:/mnt/c/Users/Danielle/Downloads/mlc_v3.10/Windows$ ./mlc.exe --max_bandwidth -b1g
Intel(R) Memory Latency Checker - v3.10
Command line parameters: --max_bandwidth -b1g

Using buffer size of 1024.000MiB/thread for reads and an additional 1024.000MiB/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 bandwidth
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        :      30998.60
3:1 Reads-Writes :      27534.79
2:1 Reads-Writes :      28309.25
1:1 Reads-Writes :      28701.91
Stream-triad like:      28948.28
```

Figure 6: Bandwidth measurements via mlc at 1GB

Danielle Nnorom
Dennise Kwarteng

Queuing Theory - Task 3:

Queuing Theory states that the more of the bandwidth a program uses exponentially the more latency will be experienced as a result, as seen in Figure 7. Some of the results from the experiment can be seen in Figure 8. This test was done by running the part_2 function and changing the buffer available for the program to run on. The result of this experiment showed that queuing theory does indeed work, by displaying the relationship between the bandwidth and the latency for the program. As less of the bandwidth connection was used by the program the smaller the latency of the program was, which can be seen by the bottom picture in Figure 8. As the buffer size is directly related to the bandwidth size this means that the experiment below the extract bandwidth for those instances are comparable to the bandwidth vs. latency discussion.

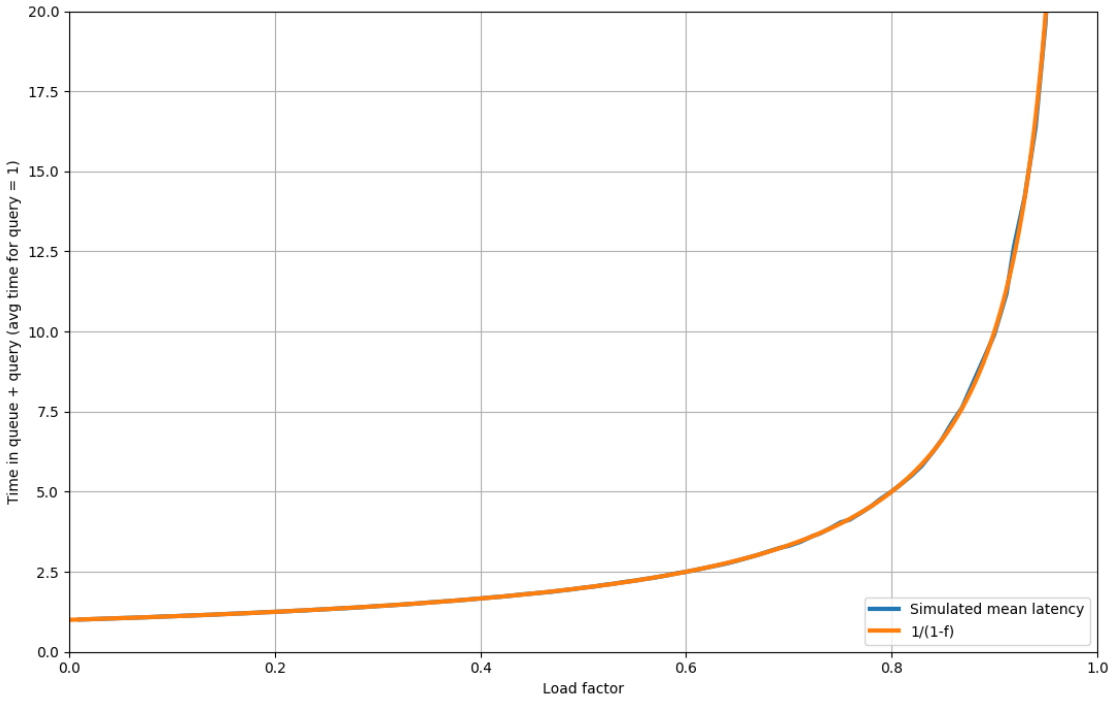

Figure 7: Basic Queuing Theory Relationship Graph

```
adduser@LAPTOP-RPI:/mnt/c/Users/Danielle/Downloads/mlc_v3.10/Windows$ ./mlc.exe --latency_matrix -b100
Intel(R) Memory Latency Checker - v3.10
Command line parameters: --latency_matrix -b100

Using buffer size of 0.098MiB
Measuring idle latencies for random access (in ns)...
                Numa node
Numa node            0
       0          74.8
adduser@LAPTOP-RPI:/mnt/c/Users/Danielle/Downloads/mlc_v3.10/Windows$ ./mlc.exe --bandwidth_matrix -b100
Intel(R) Memory Latency Checker - v3.10
Command line parameters: --bandwidth_matrix -b100

Using buffer size of 0.098MiB/thread for reads and an additional 0.098MiB/thread for writes
Measuring Memory Bandwidths between nodes within 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 Read-only traffic type
                Numa node
Numa node            0
       0        295971.5
```

```
adduser@LAPTOP-RPI:/mnt/c/Users/Danielle/Downloads/mlc_v3.10/Windows$ ./mlc.exe --latency_matrix -b100
Intel(R) Memory Latency Checker - v3.10
Command line parameters: --latency_matrix -b100

Using buffer size of 0.098MiB
Measuring idle latencies for random access (in ns)...
                Numa node
Numa node            0
       0          51.3
adduser@LAPTOP-RPI:/mnt/c/Users/Danielle/Downloads/mlc_v3.10/Windows$ ./mlc.exe --latency_matrix -b10
Intel(R) Memory Latency Checker - v3.10
Command line parameters: --latency_matrix -b10

Using buffer size of 0.010MiB
Measuring idle latencies for random access (in ns)...
                Numa node
Numa node            0
       0         104.1
```

Figure 8: Latency tests at a given Bandwidth

Danielle Nnorom
Dennise Kwarteng

Cache Miss Ratio vs Software Performance - Task 4:

Since memory created on the stack is stored in cache by default, we created a relatively huge matrix on the heap so that there will be cache misses the first time each row in the matrix is accessed. The first test that was performed was with a sequential multiplication. The code would go through each element in the matrix sequentially, and multiply the element by 1. We had it do this 50 different times just for the sequential operation.

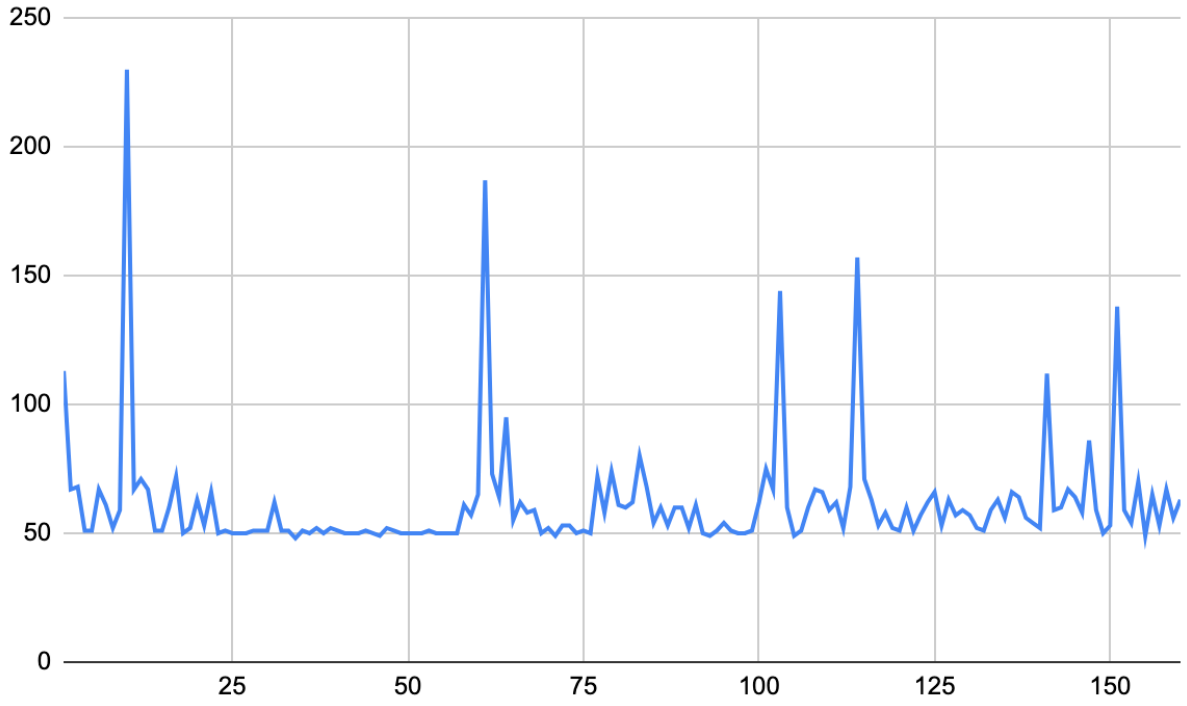

Figure 9: Access Times of Sequentially Accessed Array

The results shown here is the time it takes for each element in the array to be accessed sequentially. There is a lot of noise going on as can be seen with the spikes. Our expectation was to see the spikes appearing for values that are multiples of 30, since the matrix is a 30x30 matrix. The reason being that at those values, the entire row would possibly be inserted into the cache memory, which would result in a longer time for the multiplication operation to be done. This graph here though, shows that for one reason or another, there is a lot of latency at random points in the operation. Seeing as this is the case for the system being tested on, the graph for the random access is expected to be much worse.

Danielle Nnorom
Dennise Kwarteng

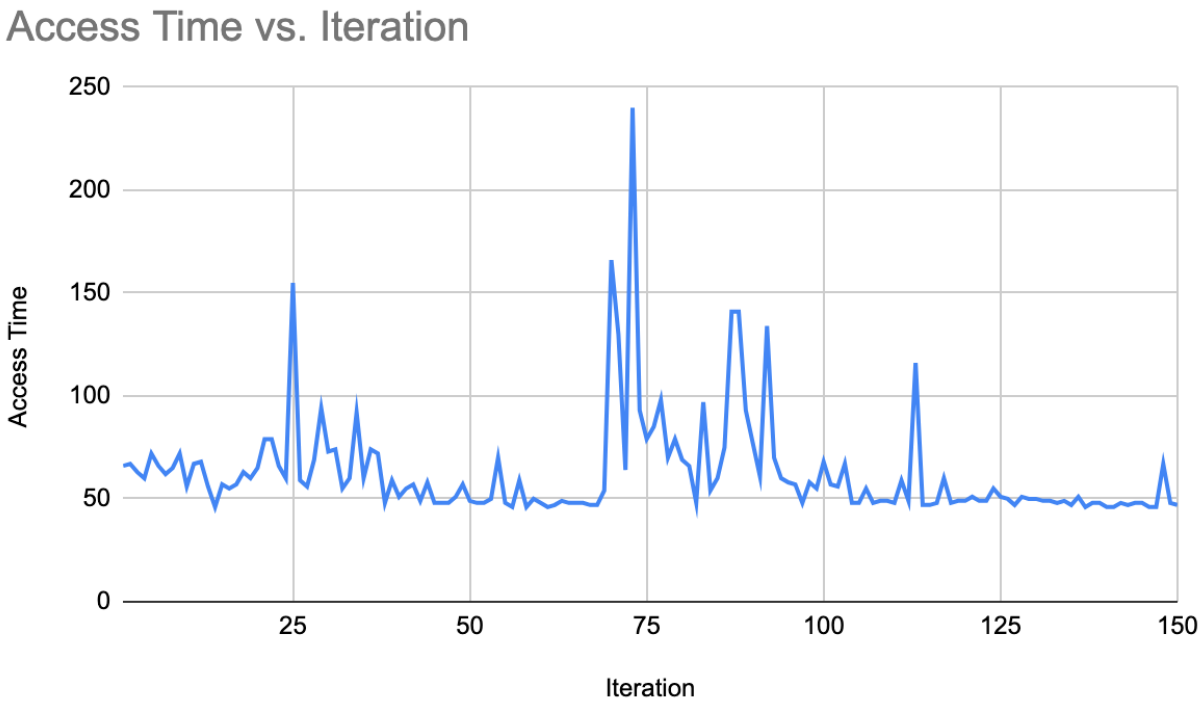

Figure 10: Access Times of Randomly Accessed Array

This graph meets expected behavior. We expected to see very few straight lines. The only areas that we see those would be from the 125th iteration towards the 150th. While unlikely, it is possible that the random number generator repeated indexes from the nearby rows. Below is a graph of some of the indexes that were produced randomly. The sequential (cache hit) test was run again for sanity check and this was the result.

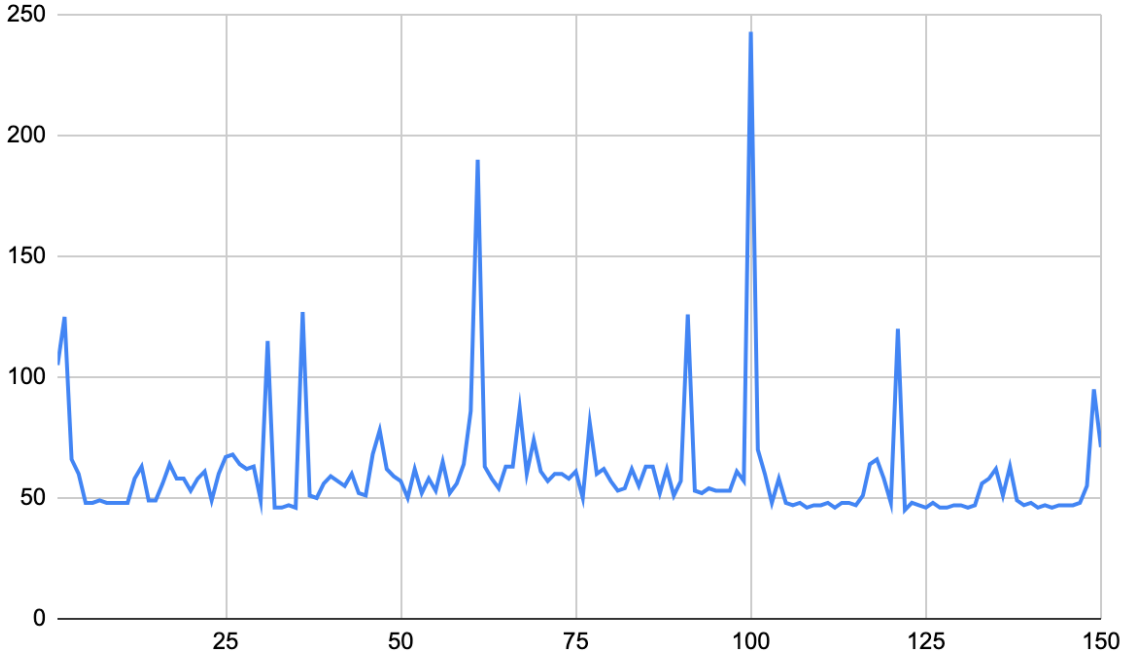

Figure 11: Access Times of Sequentially Accessed Array

As can be seen, there still seems to be latency issues occurring when accessing sequentially. To get an idea of the performance of our system, we decided to loop through the matrix and only access one element, row number 10, column 10, and this is the result.

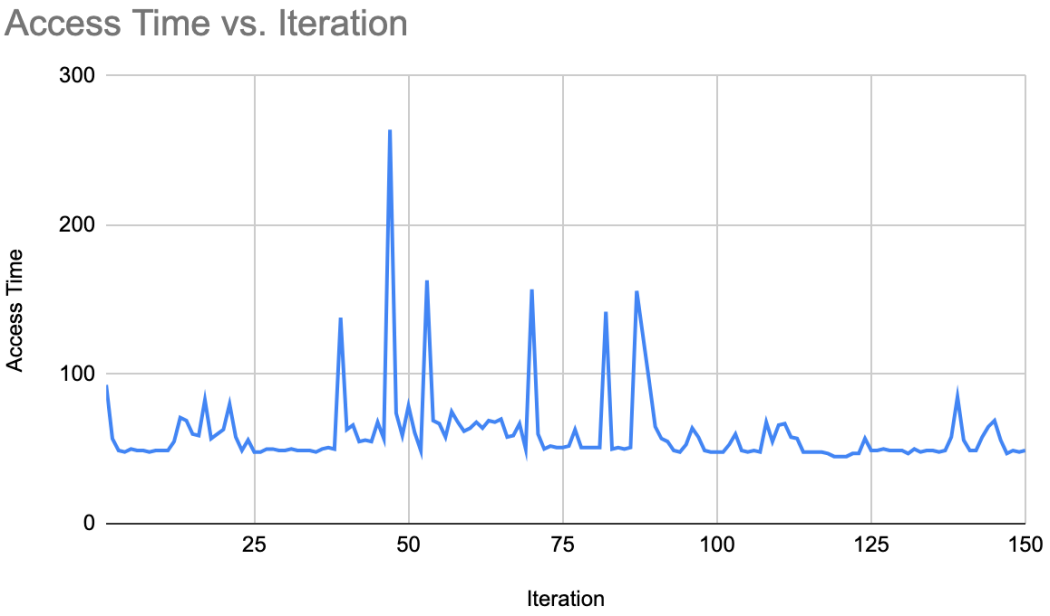

Figure 12: Access Time of One Array Element

As can be seen, even when we access only 1 element and keep the same programs running in the background, the performance of the program is not improving. This leads to the conclusion that there are many other factors that go into the performance of a program. Moreover, sometimes they can't just be optimized by the programmer themselves.

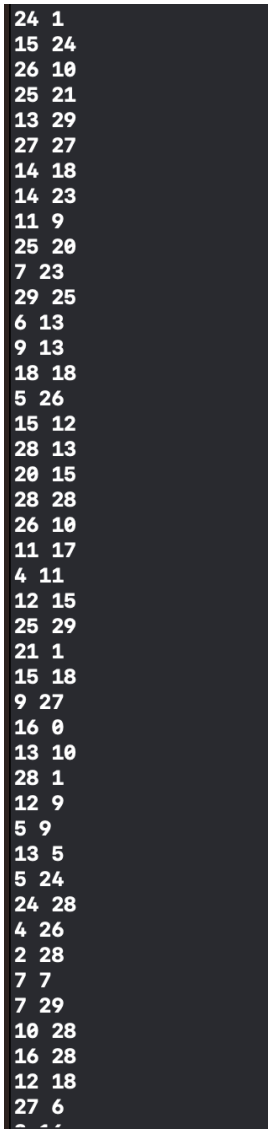

Figure 13: Random Rows and Columns Values

Danielle Nnorom
Dennise Kwarteng

TLB Table - Task 5:

TLB misses typically occur when the CPU's TLB doesn't contain the translation information for a virtual memory address, and it needs to be fetched from the page table. The way the test was conducted was sort of similar to the previous example, where arrays were made and iterated through. For the sake of comparison, one of the tests was performed using a normal array that was iterated through.

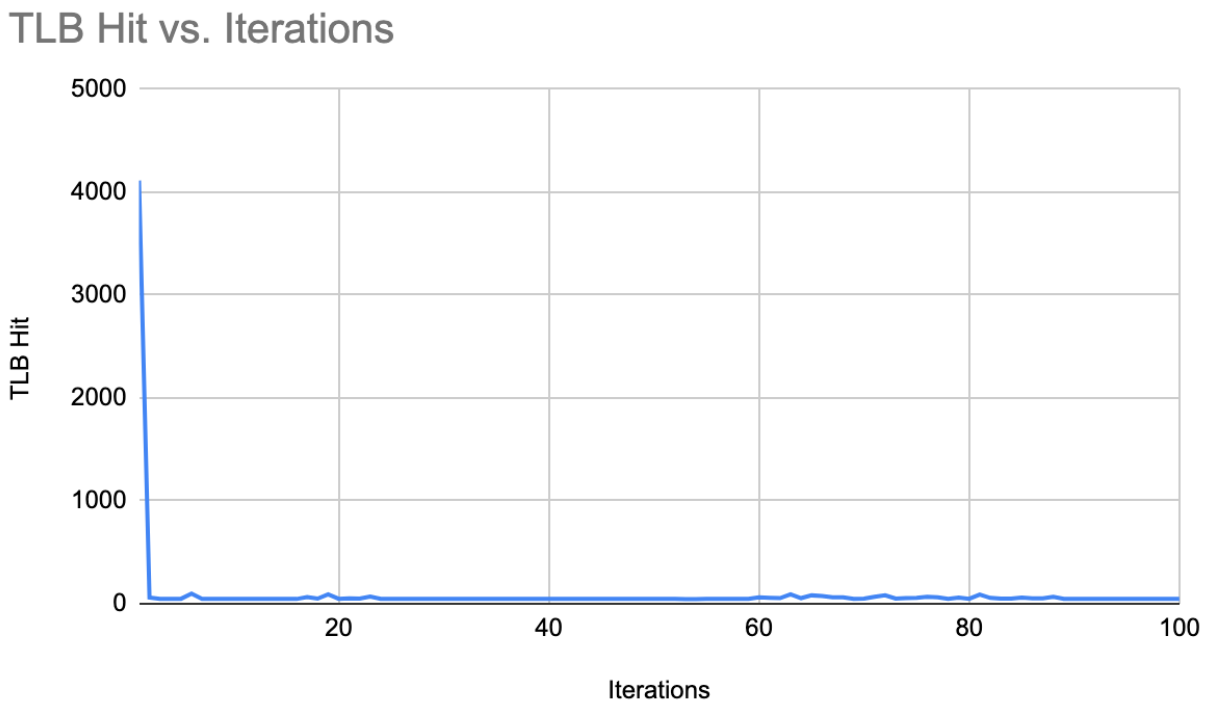

Figure 14: Sequential Array Read Times

As the graph shows, the access time was fairly low throughout with very few spikes. The most notable spike is the very first element. Since the data type is an array made in the heap, the CPU has to fetch the data and place it in the correct locations, resulting in a TLB miss. After that however, access times will normalize.

The second part of the test is the TLB Miss portion. Since the TLB has a maximum size, the test was conducted such that each access of the array would exceed the maximum size of the TLB table, creating a TLB Miss.

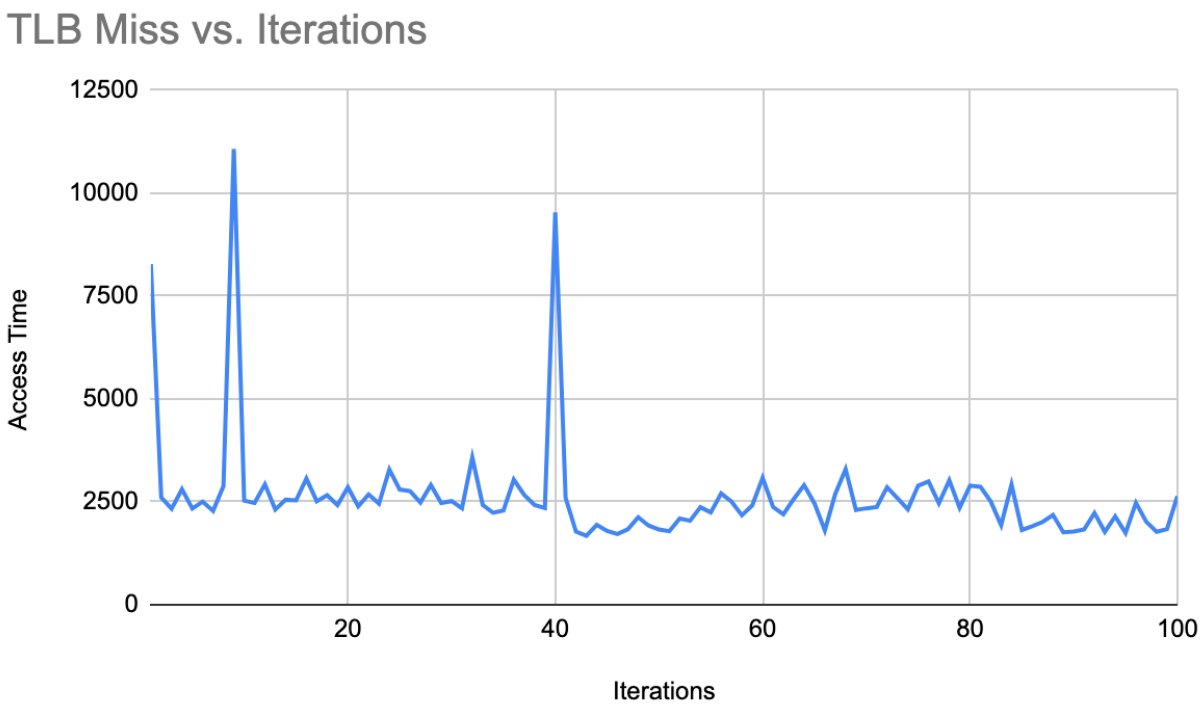

Figure 15: TLB Miss Times

The only part of this graph that does not fall into expected behavior is the peak of each entry. We expected that each spike should have been higher. Nevertheless, considering how the smallest access time of this graph is much higher than the majority of the previous graph, the test was a success. In this test, we purposely access the array in a way that would access the TLB out of bounce, creating a TLB miss. In doing this, the CPU had to go through the process of putting the correct addresses in the TLB table, which takes time to execute.