Course Project #1: Cache and Memory Performance Profiling

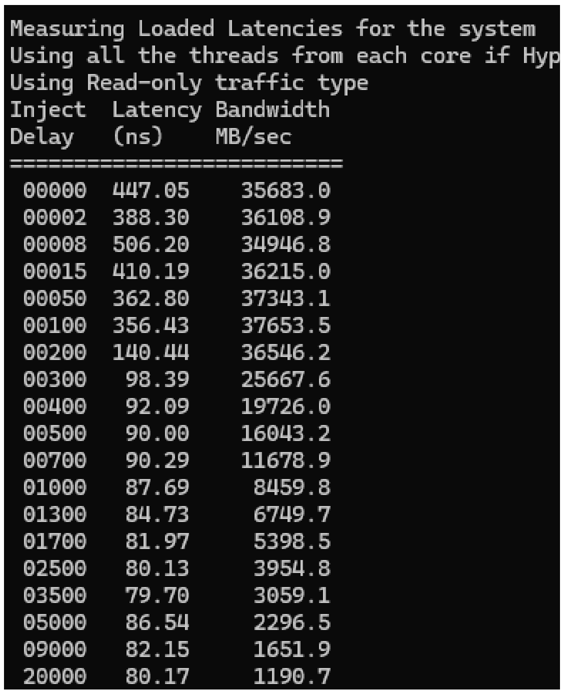
Due date: Sept. 25
Charles Clarke

The objective of this project is to gain deeper understanding of cache and memory hierarchy in modern computers.

You should design a set of experiments that will 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)

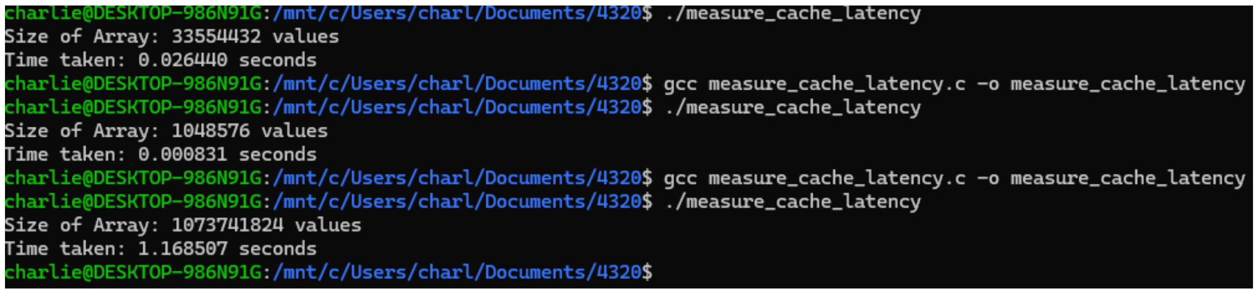
In order to get started with this project, I downloaded WSL on my desktop computer and installed Intel MLC in order to do latency checks. In order to check the latency of the cache and main memory when the queue length was zero, I checked the latency under low system load, when no major programs were taking up resources.



```
Measuring Loaded Latencies for the system
Using all the threads from each core if Hyp
Using Read-only traffic type
Inject  Latency Bandwidth
Delay   (ns)    MB/sec
==========================
 00000  447.05    35683.0
 00002  388.30    36108.9
 00008  506.20    34946.8
 00015  410.19    36215.0
 00050  362.80    37343.1
 00100  356.43    37653.5
 00200  140.44    36546.2
 00300   98.39    25667.6
 00400   92.09    19726.0
 00500   90.00    16043.2
 00700   90.29    11678.9
 01000   87.69     8459.8
 01300   84.73     6749.7
 01700   81.97     5398.5
 02500   80.13     3954.8
 03500   79.70     3059.1
 05000   86.54     2296.5
 09000   82.15     1651.9
 20000   80.17     1190.7
```

I checked the latency of main memory (DRAM) using the command 'sudo ./mlc --loaded_latency' which outputs Inject Delay, Latency, and Bandwidth. As the injected display increases, different levels of memory congestion are simulated. At 0ns delay, the system is fully loaded, with no artificially added delay, giving a baseline number for latency of the DRAM. This was about 400ns. The results can be seen on the left. As the injected delay increases, it simulates the system under less load, allowing more time between requests, and thus lowering the needed bandwidth. This table ultimately shows that under light memory loads, memory access becomes faster, as there isn't much pressure on the memory controller. There's an obvious tradeoff between latency and bandwidth here, as when the system is fully loaded with no delay, bandwidth is maximized. However latency suffers due to high contention.

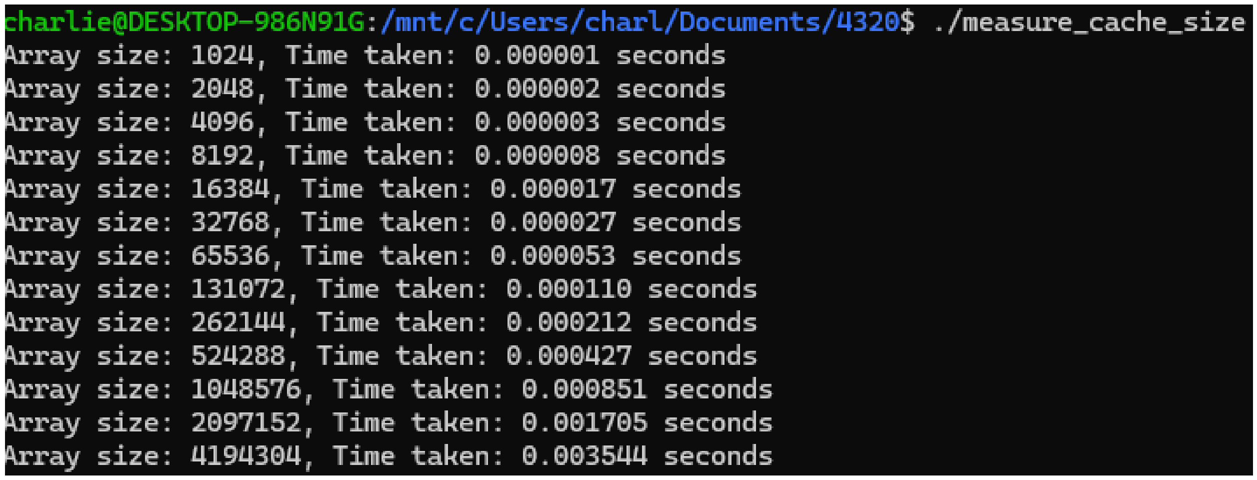
In order to target the cache memory latency I wrote a simple C program that would initialize an array. The program records the time taken using clock(), and after it iterates over the array, the time stops.The latency measurement occurs during the access of array elements, so if the array fits in the cache, the access times will be much lower than when the cache is full. The times can be seen below, and the code used can be accessed within the git repository.



```
charlie@DESKTOP-986N91G:/mnt/c/Users/charl/Documents/4320$ ./measure_cache_latency
Size of Array: 33554432 values
Time taken: 0.026440 seconds
charlie@DESKTOP-986N91G:/mnt/c/Users/charl/Documents/4320$ gcc measure_cache_latency.c -o measure_cache_latency
charlie@DESKTOP-986N91G:/mnt/c/Users/charl/Documents/4320$ ./measure_cache_latency
Size of Array: 1048576 values
Time taken: 0.000831 seconds
charlie@DESKTOP-986N91G:/mnt/c/Users/charl/Documents/4320$ gcc measure_cache_latency.c -o measure_cache_latency
charlie@DESKTOP-986N91G:/mnt/c/Users/charl/Documents/4320$ ./measure_cache_latency
Size of Array: 1073741824 values
Time taken: 1.168507 seconds
charlie@DESKTOP-986N91G:/mnt/c/Users/charl/Documents/4320$
```

As seen above, the first array at 2^25 values lands in the middle of the other two arrays, 2^20 and 2^30. The smallest array being exponentially quicker than the largest array as expected based on how much the cache can hold.

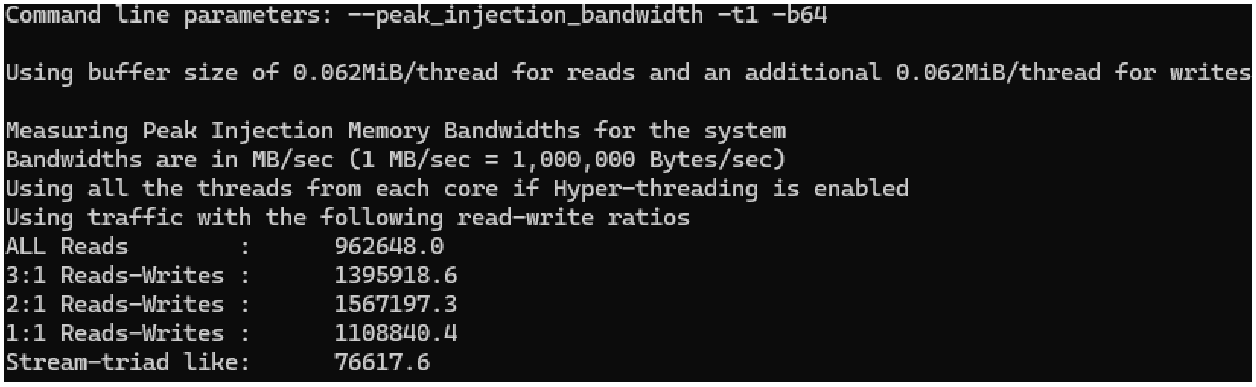
I next conducted another test to determine the relative size of each cache by looping through arrays of various sizes. I could then check where dramatic time increases happen to note the approximate size of individual caches. The results of this can be seen below.



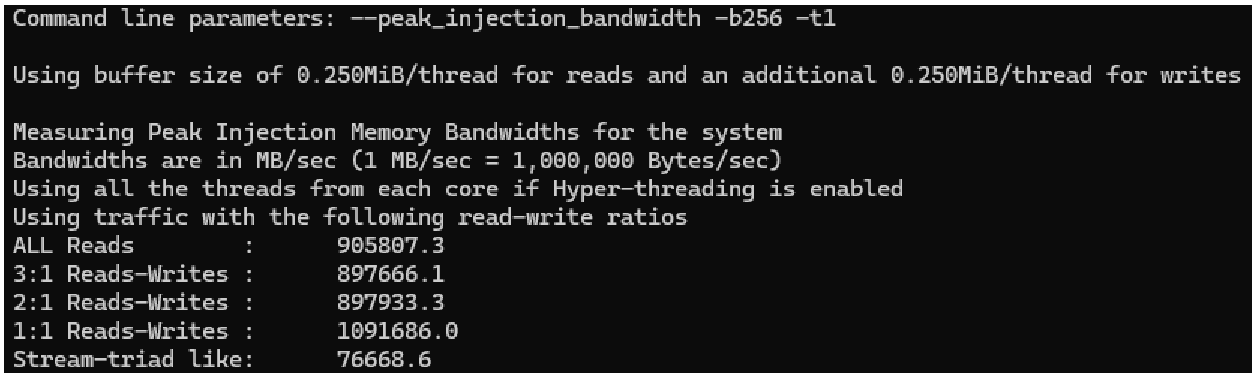
```
charlie@DESKTOP-986N91G:/mnt/c/Users/charl/Documents/4320$ ./measure_cache_size
Array size: 1024, Time taken: 0.000001 seconds
Array size: 2048, Time taken: 0.000002 seconds
Array size: 4096, Time taken: 0.000003 seconds
Array size: 8192, Time taken: 0.000008 seconds
Array size: 16384, Time taken: 0.000017 seconds
Array size: 32768, Time taken: 0.000027 seconds
Array size: 65536, Time taken: 0.000053 seconds
Array size: 131072, Time taken: 0.000110 seconds
Array size: 262144, Time taken: 0.000212 seconds
Array size: 524288, Time taken: 0.000427 seconds
Array size: 1048576, Time taken: 0.000851 seconds
Array size: 2097152, Time taken: 0.001705 seconds
Array size: 4194304, Time taken: 0.003544 seconds
```

(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)

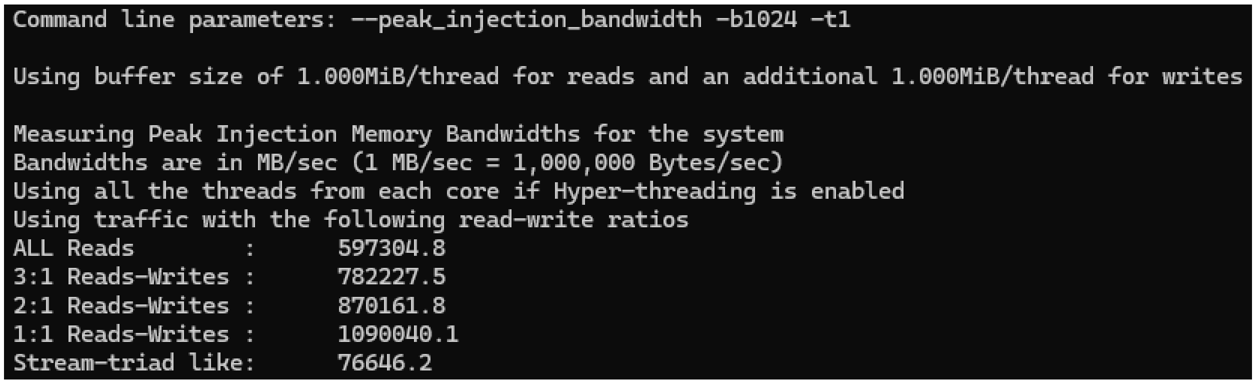
Intel MLC possesses a great way to find all of this information with just a few commands, so that's what I did. "Peak_injection_bandwidth" prints peak memory bandwidths of the platform for various read-write ratios which is just what I'm looking for. Along with the ratios, The granularity can be specified using the command "-bn" where n is the buffer size in KiB. The outputs at different granularities can all be seen below.



```
Command line parameters: --peak_injection_bandwidth -t1 -b64

Using buffer size of 0.062MiB/thread for reads and an additional 0.062MiB/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        :      962648.0
3:1 Reads-Writes :      1395918.6
2:1 Reads-Writes :      1567197.3
1:1 Reads-Writes :      1108840.4
Stream-triad like:      76617.6
```



```
Command line parameters: --peak_injection_bandwidth -b256 -t1

Using buffer size of 0.250MiB/thread for reads and an additional 0.250MiB/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        :      905807.3
3:1 Reads-Writes :      897666.1
2:1 Reads-Writes :      897933.3
1:1 Reads-Writes :      1091686.0
Stream-triad like:      76668.6
```



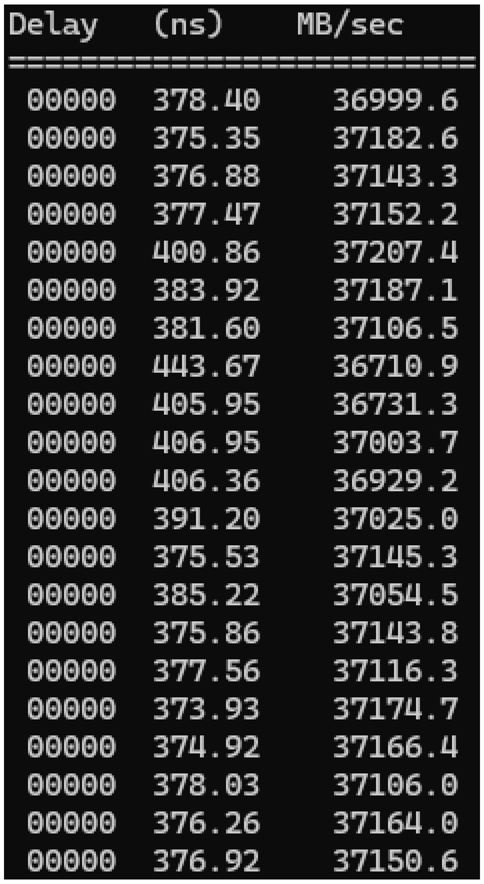
```
Command line parameters: --peak_injection_bandwidth -b1024 -t1

Using buffer size of 1.000MiB/thread for reads and an additional 1.000MiB/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        :      597304.8
3:1 Reads-Writes :      782227.5
2:1 Reads-Writes :      870161.8
1:1 Reads-Writes :      1090040.1
Stream-triad like:      76646.2
```

As seen from these tests, the bandwidth gets slower as the granularity increases which makes sense because granularity is the breaking down of larger tasks into smaller ones. Smaller accesses fit better into the cache hierarchy leading to fewer misses. Also if the latency is lower per operation, the system can process more requests in a similar amount of time. Another observation made from these results is that the bandwidth is the highest when there is a 50/50 ratio between read and write operations. This is also expected, because a balanced resource usage allows the memory to use its full capacity for both operations without overwhelming one side of the memory controller.

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

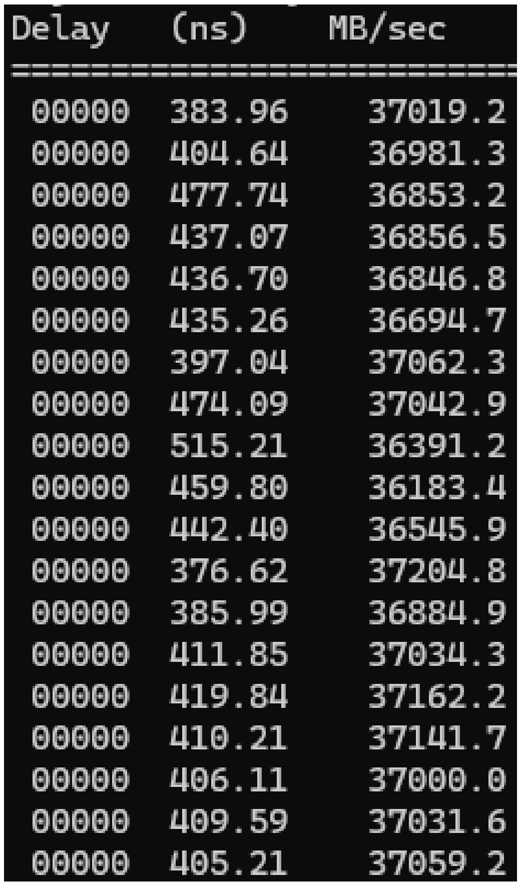
In order to work through this problem, I need to design an experiment that will give me varying amounts of read/write latency, either by altering the ratio between them, or by increasing the amount of data processed. I can then use MLC to determine how long these read and write operations take. After calculating the throughput data, I can compare it to the latency. This should give me an inverse relationship between latency and throughput of the main memory to demonstrate what queueing theory predicts. After running MLC on two programs, one with 100% read operations, and the next with 100% write operations, you can see the difference in latency overall between the two, which is very interesting. This can be seen below.

Read only



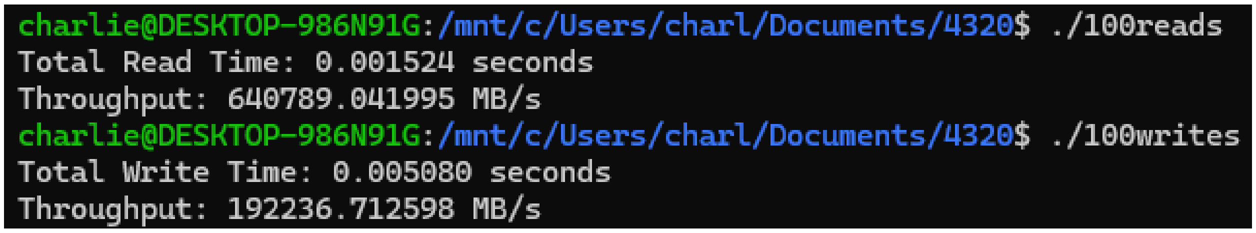
```
Delay   (ns)    MB/sec
==========================
 00000  378.40    36999.6
 00000  375.35    37182.6
 00000  376.88    37143.3
 00000  377.47    37152.2
 00000  400.86    37207.4
 00000  383.92    37187.1
 00000  381.60    37106.5
 00000  443.67    36710.9
 00000  405.95    36731.3
 00000  406.95    37003.7
 00000  406.36    36929.2
 00000  391.20    37025.0
 00000  375.53    37145.3
 00000  385.22    37054.5
 00000  375.86    37143.8
 00000  377.56    37116.3
 00000  373.93    37174.7
 00000  374.92    37166.4
 00000  378.03    37106.0
 00000  376.26    37164.0
 00000  376.92    37150.6
```

Write Only



```
Delay   (ns)    MB/sec
==========================
 00000  383.96    37019.2
 00000  404.64    36981.3
 00000  477.74    36853.2
 00000  437.07    36856.5
 00000  436.70    36846.8
 00000  435.26    36694.7
 00000  397.04    37062.3
 00000  474.09    37042.9
 00000  515.21    36391.2
 00000  459.80    36183.4
 00000  442.40    36545.9
 00000  376.62    37204.8
 00000  385.99    36884.9
 00000  411.85    37034.3
 00000  419.84    37162.2
 00000  410.21    37141.7
 00000  406.11    37000.0
 00000  409.59    37031.6
 00000  405.21    37059.2
```

As you can see, the latency of the write only program is generally higher all around. This makes sense because the read operation is less intensive than the write operation. Along with this latency check, I used these programs to output their throughput as well, the results can be seen down below.

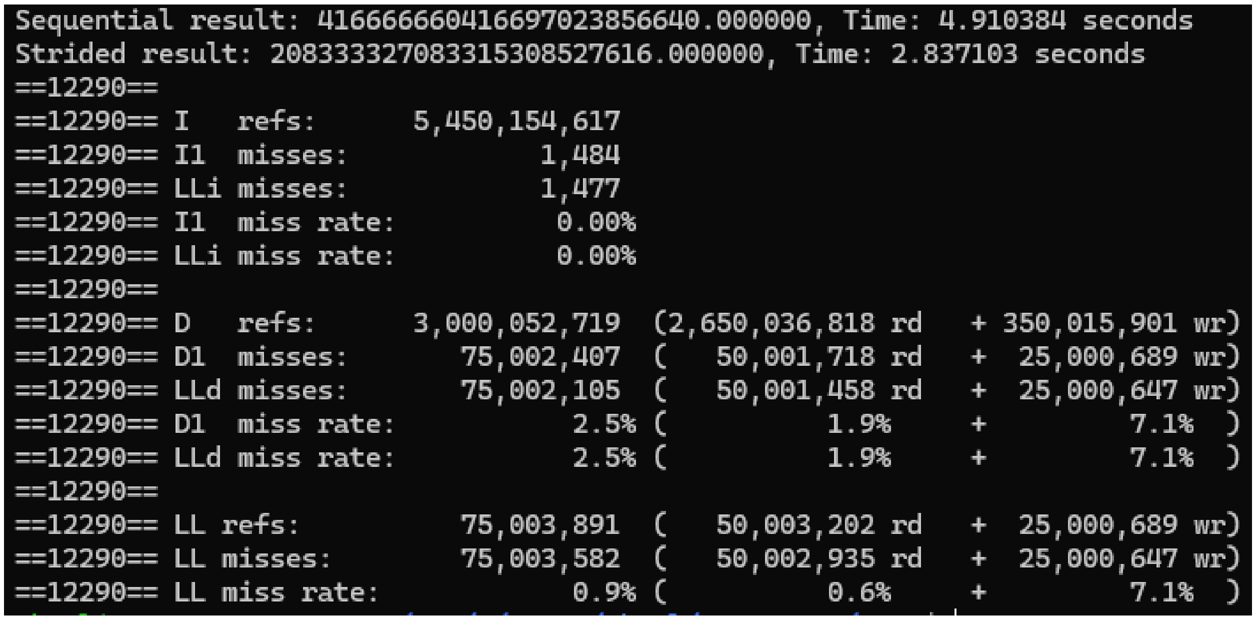


```
charlie@DESKTOP-986N91G:/mnt/c/Users/charl/Documents/4320$ ./100reads
Total Read Time: 0.001524 seconds
Throughput: 640789.041995 MB/s
charlie@DESKTOP-986N91G:/mnt/c/Users/charl/Documents/4320$ ./100writes
Total Write Time: 0.005080 seconds
Throughput: 192236.712598 MB/s
```

This just shows how much less throughput there is when writing data at a higher latency compared to reading data at a lower latency. From the same array size, we had a throughput decrease of about 70% from read to write which is very significant.

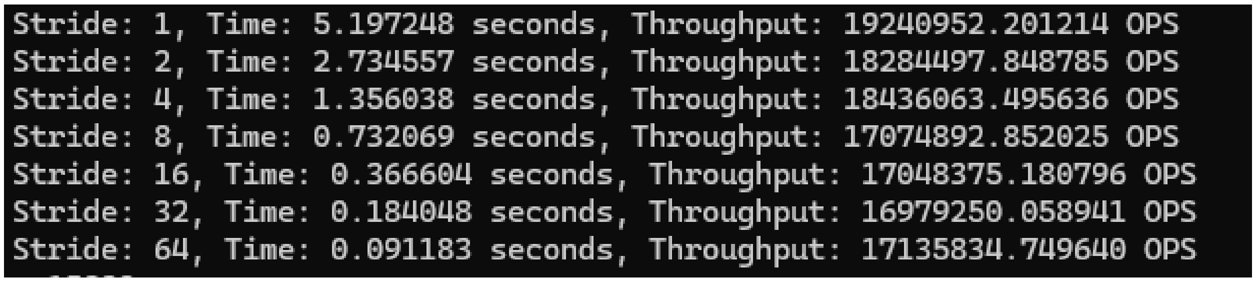
(4) the impact of cache miss ratio on the software speed performance (the software is supposed to execute relatively light computations such as multiplication)

In order to determine the impact of cache miss ratio on the software speed performance I first created a simple multiplication program to target the memory access speed. My program will have two scenarios, one where there is a low cache miss ratio, ensuring data is accessed sequentially. The next scenario will have poor cache access ratio, by reaching the memory in a strided pattern that exceeds cache line size. By changing this stride size to force cache misses, I can increase the miss ratio. The code I'm using to simulate this can be seen in my GIT repository. When the stride size is set low to just 2, the result can be seen below, with a last level miss rate of 7.1%.



```
Sequential result: 41666666041669702385664О.000000, Time: 4.910384 seconds
Strided result: 20833332708331530852761б.000000, Time: 2.837103 seconds
==12290==
==12290== I   refs:      5,450,154,617
==12290== I1  misses:            1,484
==12290== LLi misses:            1,477
==12290== I1  miss rate:          0.00%
==12290== LLi miss rate:          0.00%
==12290==
==12290== D   refs:      3,000,052,719  (2,650,036,818 rd   + 350,015,901 wr)
==12290== D1  misses:       75,002,407  (   50,001,718 rd   +  25,000,689 wr)
==12290== LLd misses:       75,002,105  (   50,001,458 rd   +  25,000,647 wr)
==12290== D1  miss rate:           2.5% (          1.9%     +         7.1%  )
==12290== LLd miss rate:           2.5% (          1.9%     +         7.1%  )
==12290==
==12290== LL refs:          75,003,891  (   50,003,202 rd   +  25,000,689 wr)
==12290== LL misses:        75,003,582  (   50,002,935 rd   +  25,000,647 wr)
==12290== LL miss rate:            0.9% (          0.6%     +         7.1%  )
```

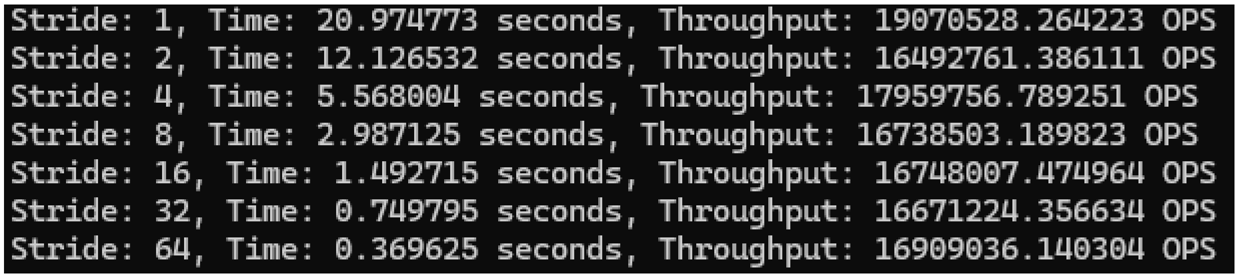
However once I up that value to something crazy like 64 or 128, the last level miss rate bumps up over a percent to around 8.3%. This shows that as the stride increases, the miss rate increases which makes sense, however did this have any noticeable effect on the software speed performance. The strided pattern is going to make the program finish quicker, because it’s accessing fewer elements, so I’m really looking at operations per second. I added some code to my program that would show OPS for different stride sizes. The results of this test can be seen below.



```
Stride: 1, Time: 5.197248 seconds, Throughput: 19240952.201214 OPS
Stride: 2, Time: 2.734557 seconds, Throughput: 18284497.848785 OPS
Stride: 4, Time: 1.356038 seconds, Throughput: 18436063.495636 OPS
Stride: 8, Time: 0.732069 seconds, Throughput: 17074892.852025 OPS
Stride: 16, Time: 0.366604 seconds, Throughput: 17048375.180796 OPS
Stride: 32, Time: 0.184048 seconds, Throughput: 16979250.058941 OPS
Stride: 64, Time: 0.091183 seconds, Throughput: 17135834.749640 OPS
```

As you can see above, as the stride size increases, the time the program takes to complete is less. This is obviously expected though. What I was excited to see was the change in throughput, it seems to decrease by large amounts at different stride sizes. From 1 to 2 and 4, and then to 8-64. The OPS seems to be separated into groups. Next I will try to increase the array size by a factor of 10 or so to see what happens at a larger scale.

The factor of 10 took so long I had to abort the command. So I just increased the array size by 4 times. A somewhat similar result was seen. This is shown below.



```
Stride: 1, Time: 20.974773 seconds, Throughput: 19070528.264223 OPS
Stride: 2, Time: 12.126532 seconds, Throughput: 16492761.386111 OPS
Stride: 4, Time: 5.568004 seconds, Throughput: 17959756.789251 OPS
Stride: 8, Time: 2.987125 seconds, Throughput: 16738503.189823 OPS
Stride: 16, Time: 1.492715 seconds, Throughput: 16748007.474964 OPS
Stride: 32, Time: 0.749795 seconds, Throughput: 16671224.356634 OPS
Stride: 64, Time: 0.369625 seconds, Throughput: 16909036.140304 OPS
```

Here the OPS is really only higher when stride length is 1. Also it spikes up at 4 for some reason. I’m not sure at all why this is.

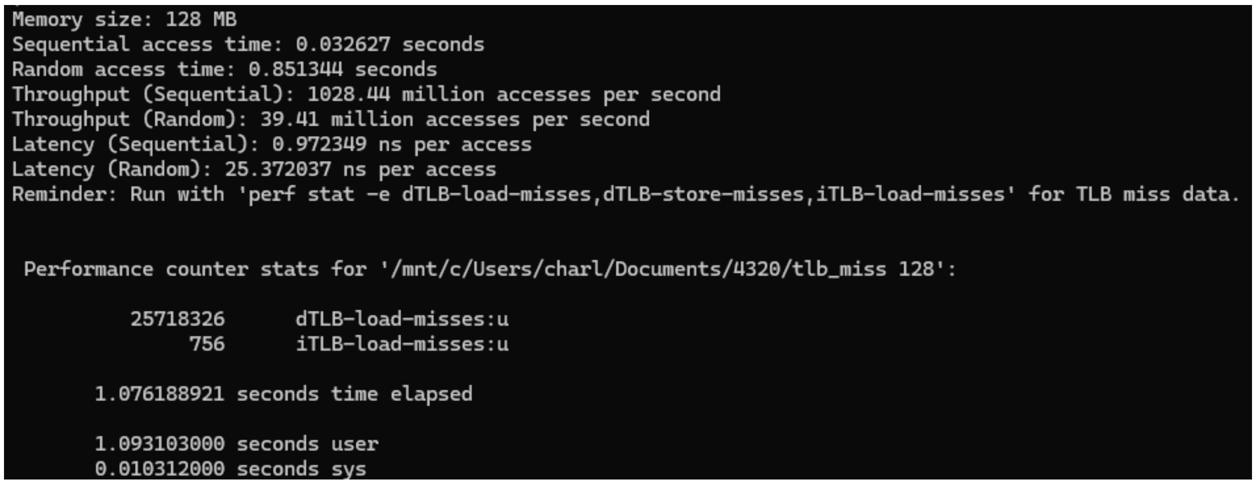
In general however this experiment has shown that as cache miss ratio increases, system speed performance drops. I artificially increased cache miss ratio using my multiplication program with increasing sized gaps that are between data accessed. As the stride size or the cache miss rate increases, the throughput decreases, which is represented as operations per second. This is very closely related to system speed performance.

(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)

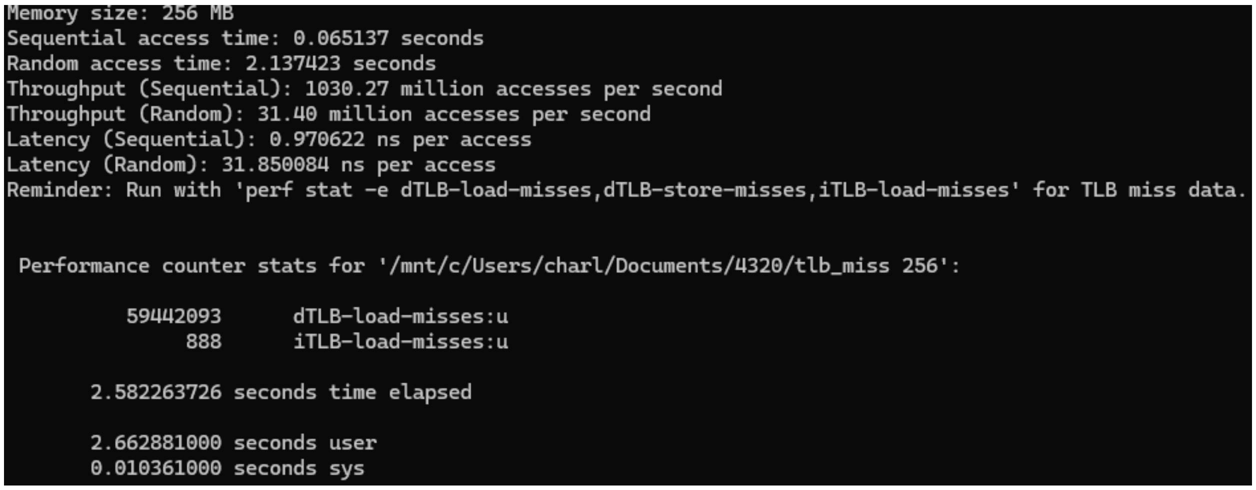
This is a similar setup to the last experiment. Except this time I figured out how to get perf working on my machine. The general overview of the experiment is quite simple. First of all, TLB or translation lookaside buffer is a cache that stores recent virtual to physical memory address translation. A miss occurs when a virtual address cannot be found in the TLB. Once this happens, a page table lookup is necessary, which is slower. If I can artificially alter the TLB miss ratio, then I can determine its impact on the performance of my system.

I will make this happen by using a similar multiplication program to the precious experiments. By storing a large amount of values in an array, and then accessing these values in a random pattern to stress the TLB. By modifying the size of this array, I could compare the throughput to determine the effect of TLB miss ratio on system speed performance.

This is definitely my favorite experiment of them all, probably because I finally got perf to work, and it's a life changer. By increasing the size of the array, the TLB miss rate continues to increase, and the throughput continues to decrease. The numbers are so beautiful I put them in a table and a graph. My code outputs and data can be seen below.



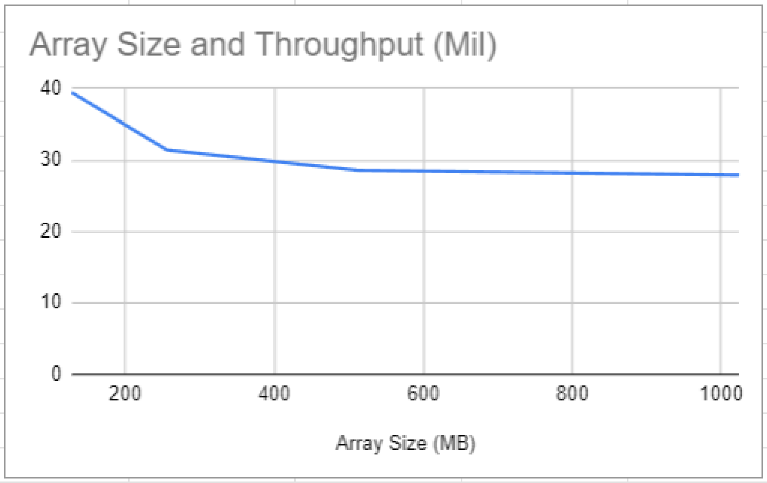
```
Memory size: 128 MB
Sequential access time: 0.032627 seconds
Random access time: 0.851344 seconds
Throughput (Sequential): 1028.44 million accesses per second
Throughput (Random): 39.41 million accesses per second
Latency (Sequential): 0.972349 ns per access
Latency (Random): 25.372037 ns per access
Reminder: Run with 'perf stat -e dTLB-load-misses,dTLB-store-misses,iTLB-load-misses' for TLB miss data.


 Performance counter stats for '/mnt/c/Users/charl/Documents/4320/tlb_miss 128':

         25718326      dTLB-load-misses:u
              756      iTLB-load-misses:u

       1.076188921 seconds time elapsed

       1.093103000 seconds user
       0.010312000 seconds sys
```



```
Memory size: 256 MB
Sequential access time: 0.065137 seconds
Random access time: 2.137423 seconds
Throughput (Sequential): 1030.27 million accesses per second
Throughput (Random): 31.40 million accesses per second
Latency (Sequential): 0.970622 ns per access
Latency (Random): 31.850084 ns per access
Reminder: Run with 'perf stat -e dTLB-load-misses,dTLB-store-misses,iTLB-load-misses' for TLB miss data.


 Performance counter stats for '/mnt/c/Users/charl/Documents/4320/tlb_miss 256':

         59442093      dTLB-load-misses:u
              888      iTLB-load-misses:u

       2.582263726 seconds time elapsed

       2.662881000 seconds user
       0.010361000 seconds sys
```

Above you can see that throughput decreases as the size of the memory array increases. Also with a quick calculation of dTLB misses divided by array size we get the TLB miss ratio.

Array Size (MB)	Throughput (Mil)	TLB Miss Rate
128	39.4	0.766
256	31.4	0.866
512	28.57	0.946
1024	27.9	0.975



Above is the throughput plotted against the array size and you can clearly see a downward trend in the system speed performance as the array size goes up, increasing the TLB miss rate.

Overall these experiments turned out to be very interesting and helped me dig deeper into computer systems as a whole. They also taught me a whole lot about the command line and MLC.

The Intel Memory Latency Checker is a useful tool: Google or ask ChatGPT about “Intel Memory Latency Checker”
The Linux “perf” command can gather lots of CPU runtime information such as cache miss ratio and TLB miss ratio,
and you can Google or ask ChatGPT to learn more.