CSE 240B Programming Assignment

Name: Sumiran Shubhi
PID : A53314039

1. **Documentation**
    a) How does your program measure latency?
        - In my programming assignment, I have used the concept of pointer chasing using a linked list. Pointer chasing combined with linked lists provides the flexibility to easily select between sequential access as well as random access.
        - Each node of the linked list is of size 64Bytes. No assumption has been made regarding the block size of any of the caches. Since the block size is always a multiple of 64B, I chose this to be the size of a single node.
        - To measure the latency, I do two types of accesses of these nodes. Read only access and Read/Write access.
        - To ensure that no optimization is performed on the reads or read-modify-writes, I use *assembly instructions and memory fences*. The timer is started after the first memory fence, i.e., before the read is done and stopped before the second memory fence. The difference in the clock time gives us the access time.
        - Multiple threads can be run simultaneously. Latency is calculated as an average of latencies of multiple accesses of each thread.
        - I have taken care to perform the entire access of the linked list nodes, 5 times for each thread. There can be several values affecting the absolute value of latency at a particular moment, hence an average of these gives a better estimate.
        - To figure out the size of caches using the latency pattern , From a wide range of possible cache size, each size is chosen once. A linked list is created whose size is equal to the size we selected. The idea is to fill up the entire cache with the nodes and then access it repeatedly to notice the hit latency. If the selected size is greater than the a particular level of cache, then a spike in the access time will be noticed, because the accesses will result in a miss in the lower level of cache and it will have to be fetched from the higher level. This gives us the latency of the higher levels of cache.
        - The sizes selected were {8,16,32,64,128,256,512,1024,2048,4096,8192,16384}KB. The aim is to observe the size of all levels of caches, including L3 which can be quite big.

b) What options or compile time configuration does your program take and what do those options or compile time configurations do?

- I use the following gcc libraries:
    1) lrt – This library allows us to link the realtime library which is required for functions in time.h
    2) pthread – Similar to above, this option helps the linker in finding symbols from pthread.h.

- I use the following defines to explore latencies under different scenarios (can be seen in ***Makefile***):
    1. *RD_OP/RW_OP* : to enable read-only or read/write mode.
    2. *SIZE_*K* : to select the total size of the linked list that will be accessed in the program. This value is an estimation of possible values of cache sizes.
    3. THREADS_* : This define is used to select the number of threads that the program. Values are {1,2,4,8,16}.

c) What limitations does your program have? (e.g. uses of system dependent functions, can only scale to a maximum of N threads)?

There are a few restrictions and issues that I have faced due to running the program on my laptop which has Processor Intel(R) Core(TM) i5-10210U CPU @ 1.60GHz, 2112 Mhz, 4 Core(s), 8 Logical Processor(s).
Because of these issues I wasn't able to get the best possible result for the latency curves. Below I describe these issues and the possible reasoning why these affect the results that I got:

- I tried Windows Subsystem for Linux (***WSL***) to run my code. No matter which node size or array size I used, the minimum latency was close to 200 ns or more. I believe that using Ubuntu on a Windows laptop introduces some overhead in accesses. Hence, these values weren't accurate.
- Next, I ran the same code using ***Cygwin***, which is lighter and is expected to have lesser overhead. The suspected lowest level cache latency was still between 100-140ns, which does not seem reasonable.
- After that I managed to access a ***linux machine*** and was able to get reasonable values for latency of lowest level of cache, which was between 20 to 50ns.
- Although the values of latencies for sequential access are correct, for random accesses the obtained values are different from the expected values. I have provided the reasoning as well as reference for that in the *Analysis* section.

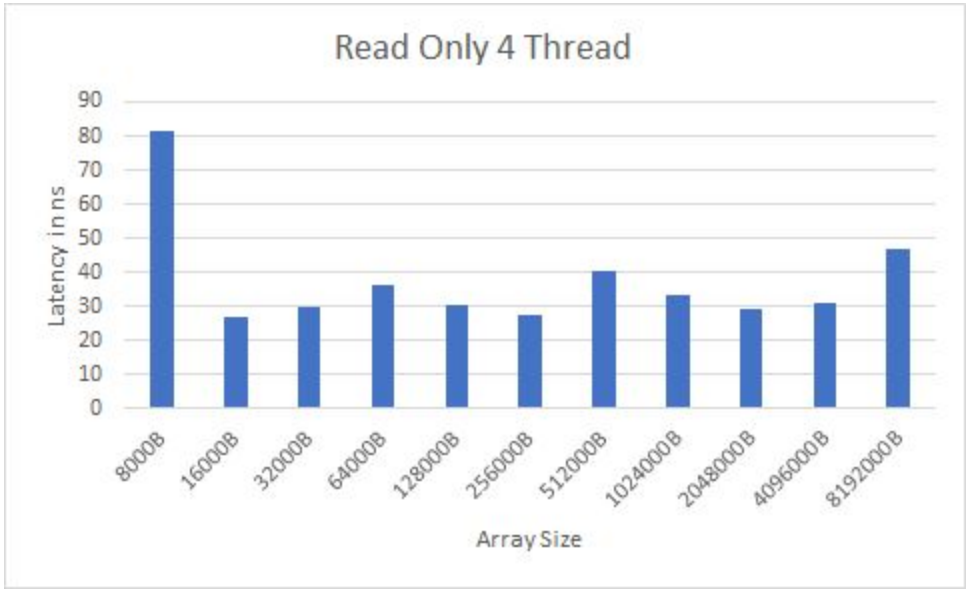
In terms of limitations of the options that the program uses, the number of threads that can be spawned off in Ubuntu is quite high[1]. Hence, it is not a limitation in practical terms. The program uses assembly instructions and memory fences, hence the possibility of any additional overhead is less.

2. Graphs

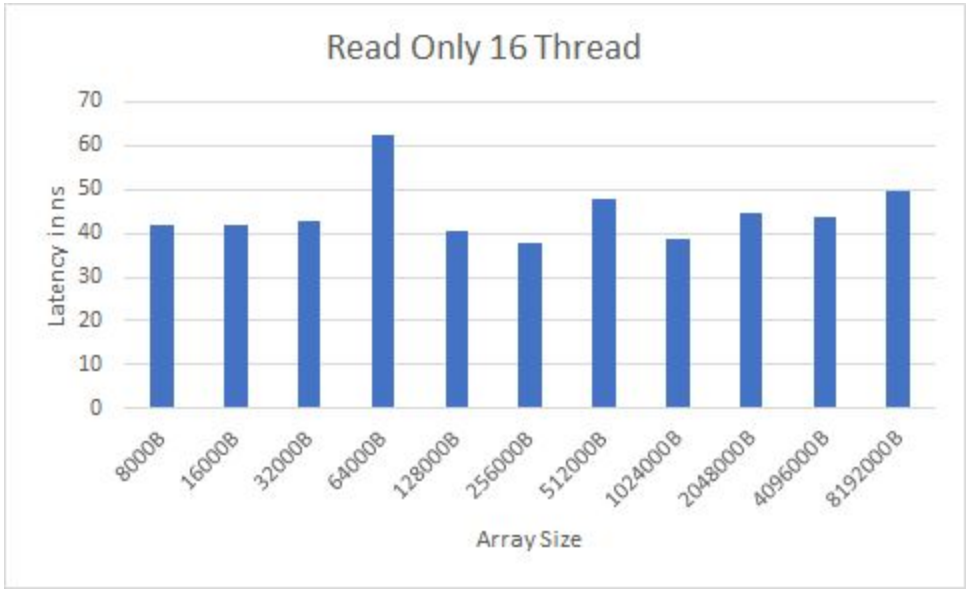
a) Read Only


Read Only 1 Thread
Latency in ns
90
80
70
60
50
40
30
20
10
0
8000B
16000B
32000B
64000B
128000B
256000B
512000B
1024000B
2048000B
4096000B
8192000B
16384000B
Array Size

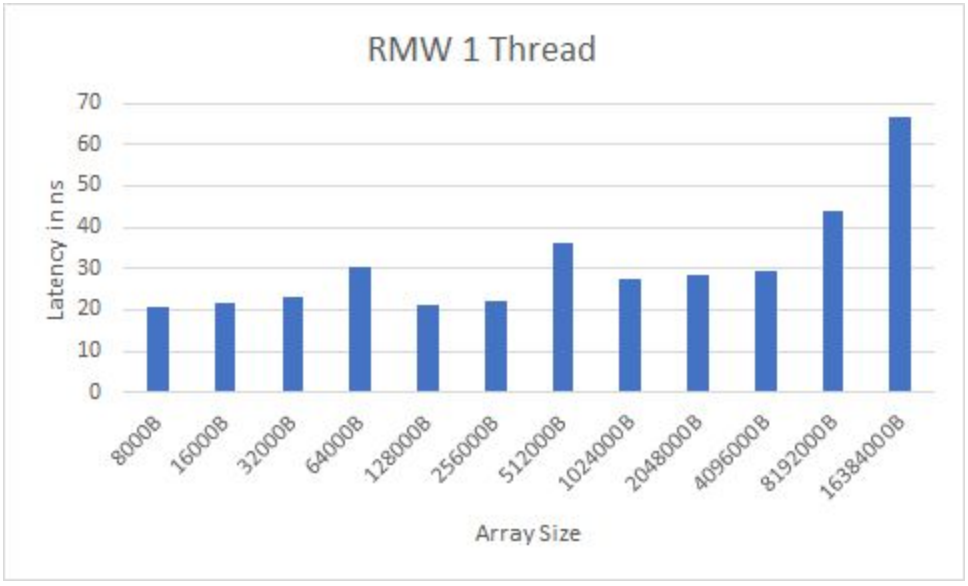

Read Only 2 Thread
Latency in ns
100
90
80
70
60
50
40
30
20
10
0
8000B
16000B
32000B
64000B
128000B
256000B
512000B
1024000B
2048000B
4096000B
8192000B
Array Size

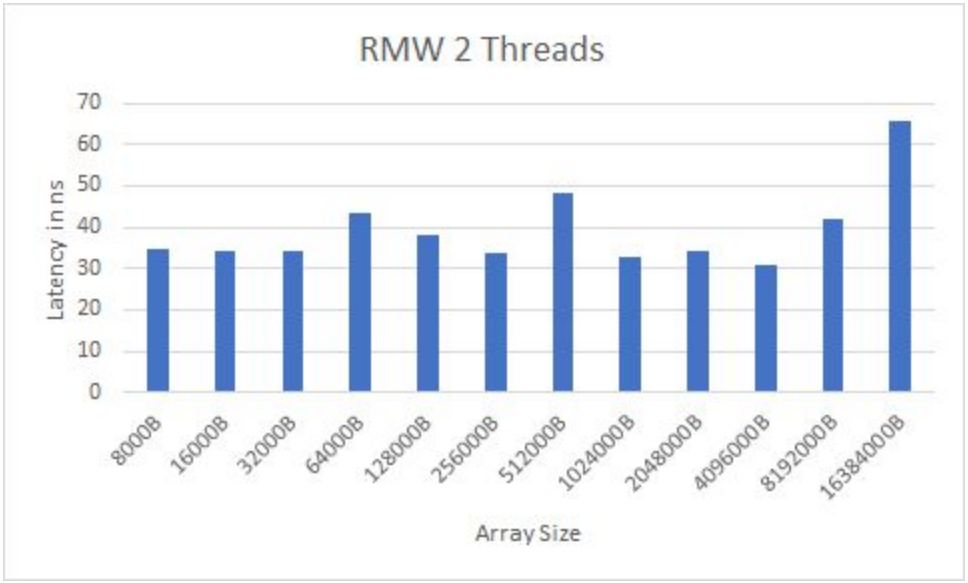

Read Only 4 Thread
Latency in ns
90
80
70
60
50
40
30
20
10
0
8000B
16000B
32000B
64000B
128000B
256000B
512000B
1024000B
2048000B
4096000B
8192000B
Array Size

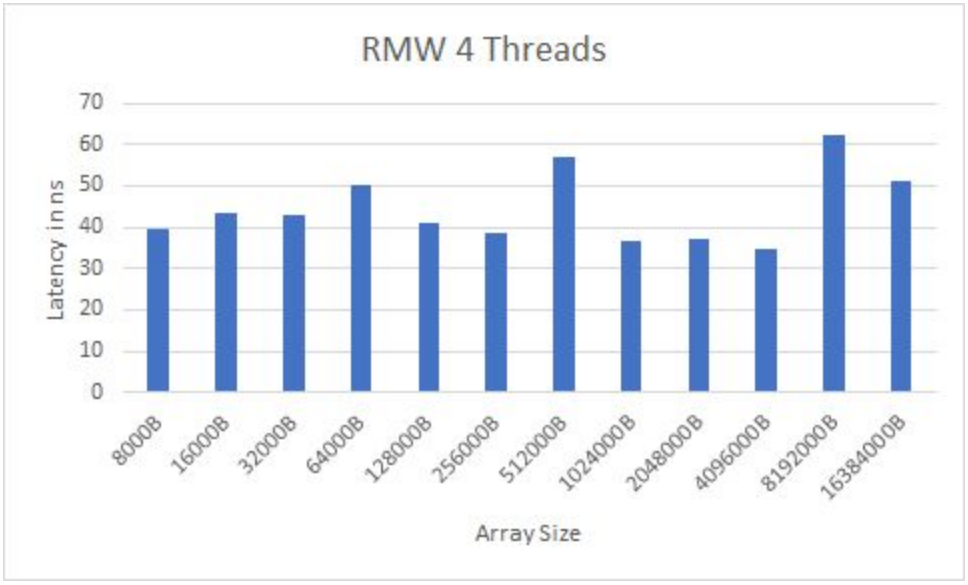

Read Only 8 Thread
Latency in ns
100
90
80
70
60
50
40
30
20
10
0
8000B
16000B
32000B
64000B
128000B
256000B
512000B
1024000B
2048000B
4096000B
8192000B
Array Size

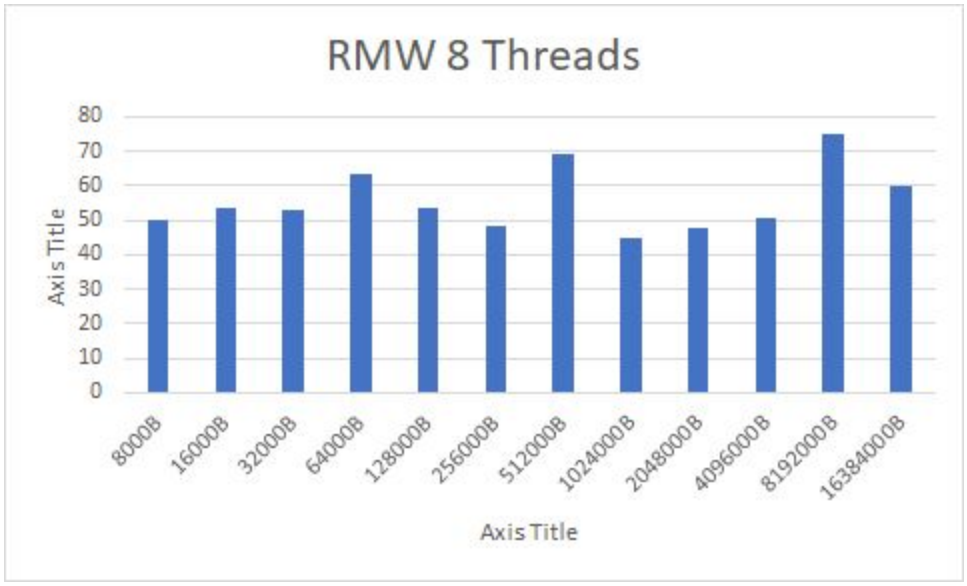

Read Only 16 Thread
Latency in ns
70
60
50
40
30
20
10
0
8000B
16000B
32000B
64000B
128000B
256000B
512000B
1024000B
2048000B
4096000B
8192000B
Array Size

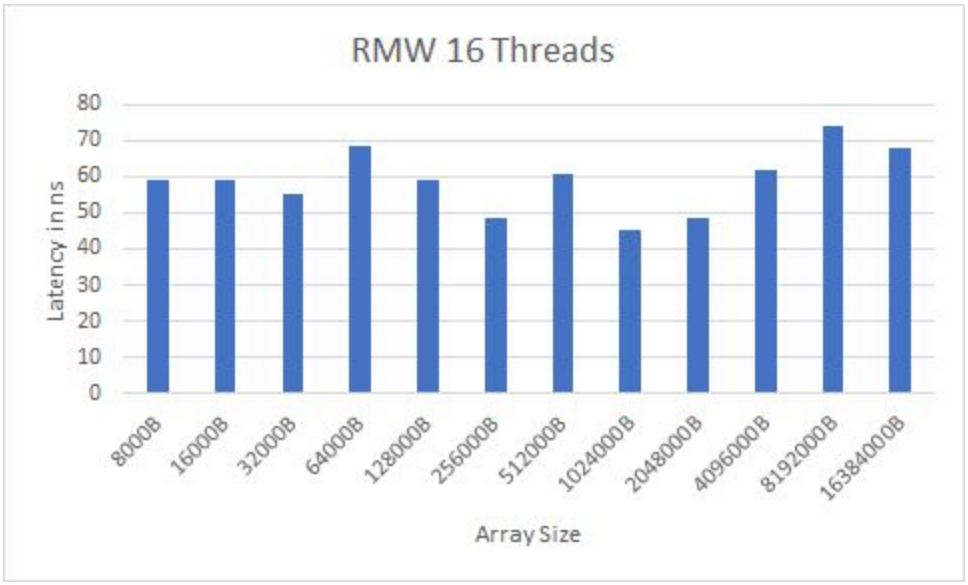
b) Read/Write


RMW 1 Thread
Latency in ns
70
60
50
40
30
20
10
0
8000B
16000B
32000B
64000B
128000B
256000B
512000B
1024000B
2048000B
4096000B
8192000B
16384000B
Array Size

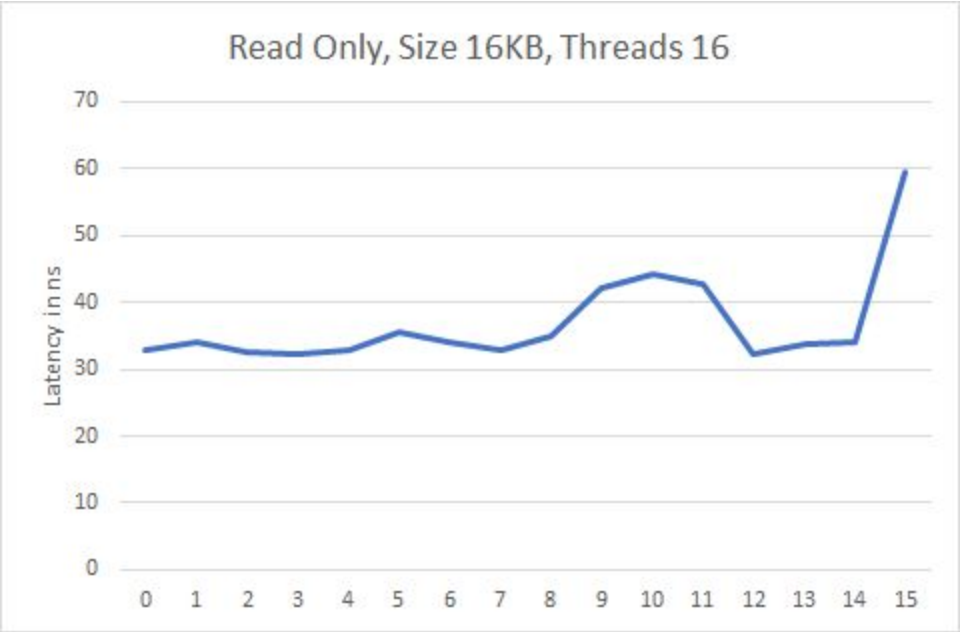

RMW 2 Threads
Latency in ns
70
60
50
40
30
20
10
0
8000B
16000B
32000B
64000B
128000B
256000B
512000B
1024000B
2048000B
4096000B
8192000B
16384000B
Array Size

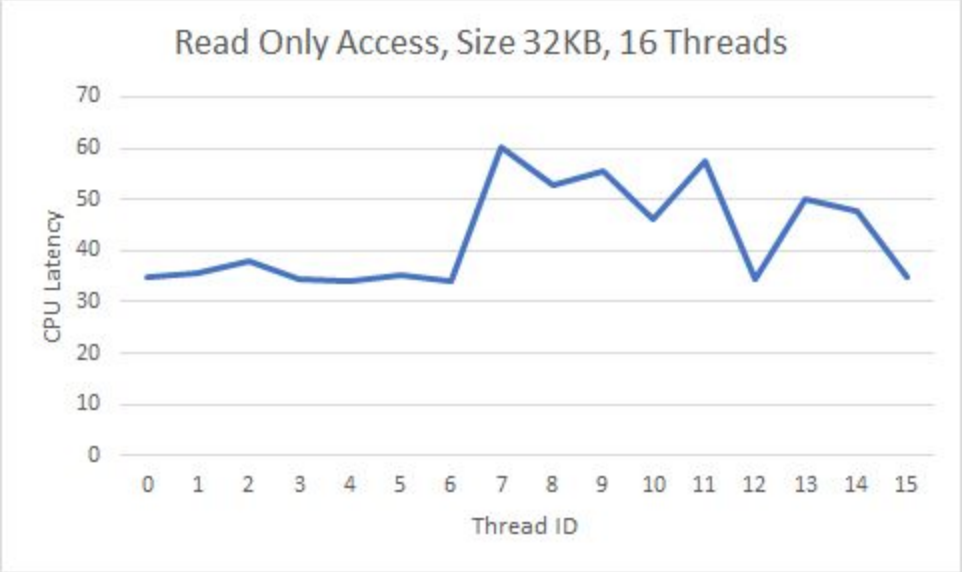

RMW 4 Threads
Latency in ns
70
60
50
40
30
20
10
0
8000B
16000B
32000B
64000B
128000B
256000B
512000B
1024000B
2048000B
4096000B
8192000B
16384000B
Array Size

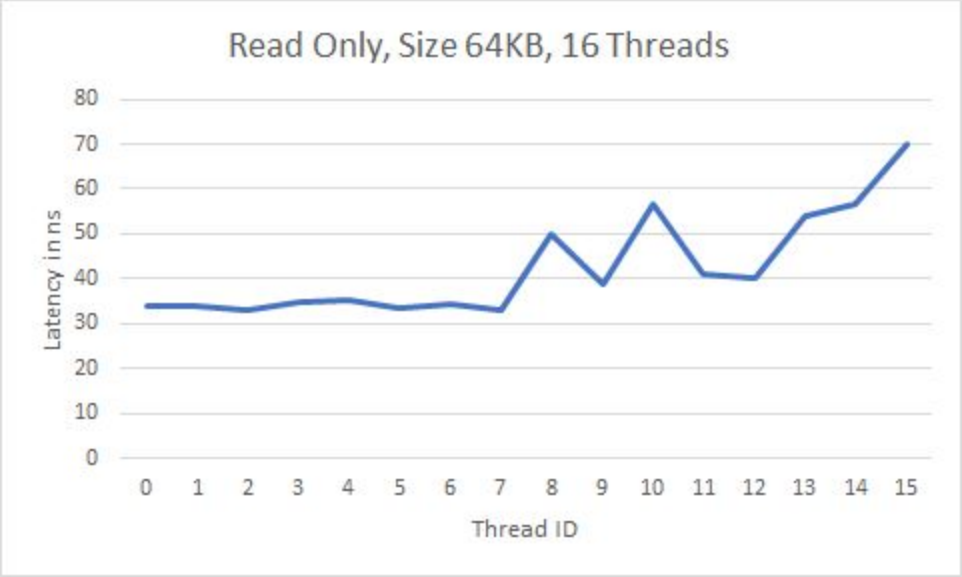

RMW 8 Threads
Axis Title
80
70
60
50
40
30
20
10
0
8000B
16000B
32000B
64000B
128000B
256000B
512000B
1024000B
2048000B
4096000B
8192000B
16384000B
Axis Title

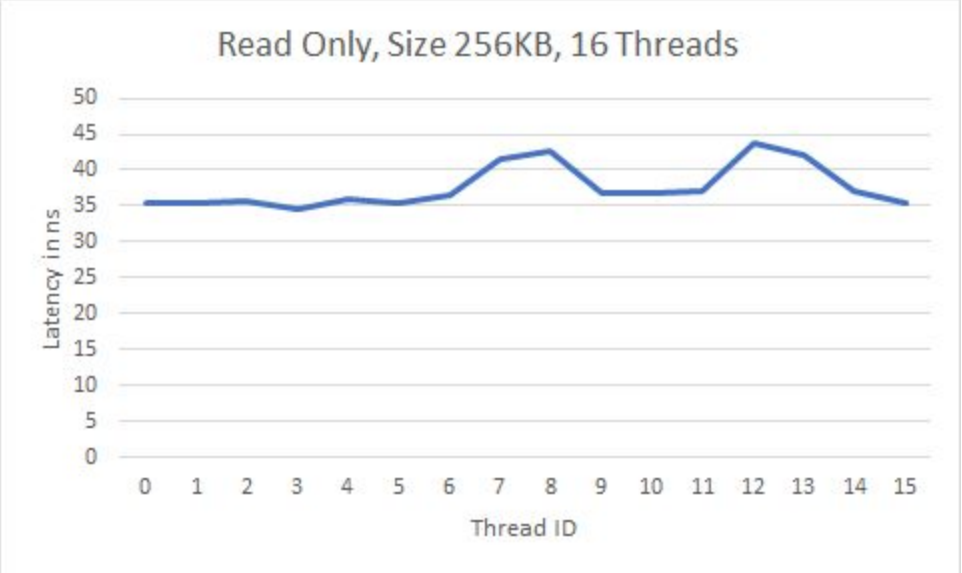

RMW 16 Threads
Latency in ns
80
70
60
50
40
30
20
10
0
8000B
16000B
32000B
64000B
128000B
256000B
512000B
1024000B
2048000B
4096000B
8192000B
16384000B
Array Size

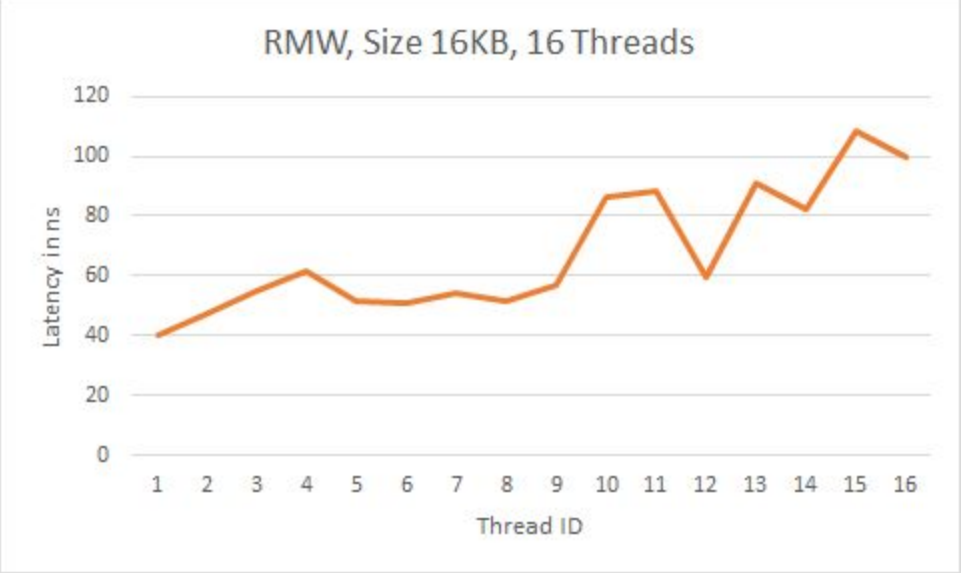
A few other relevant graphs:

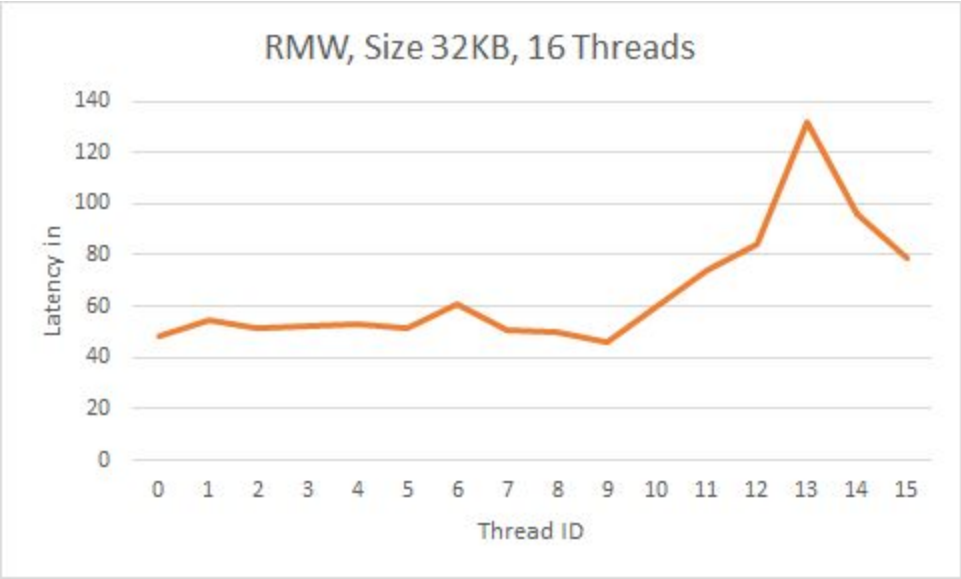


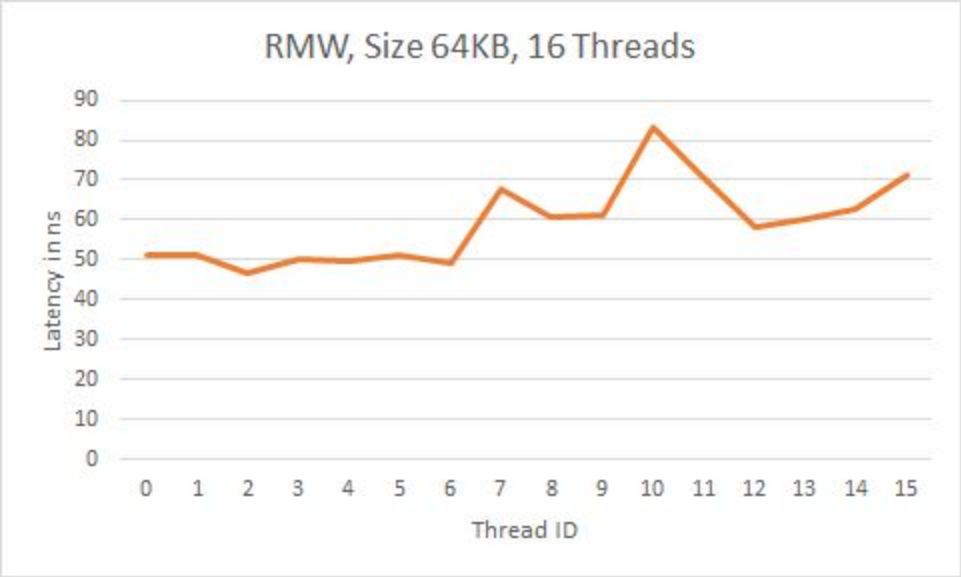


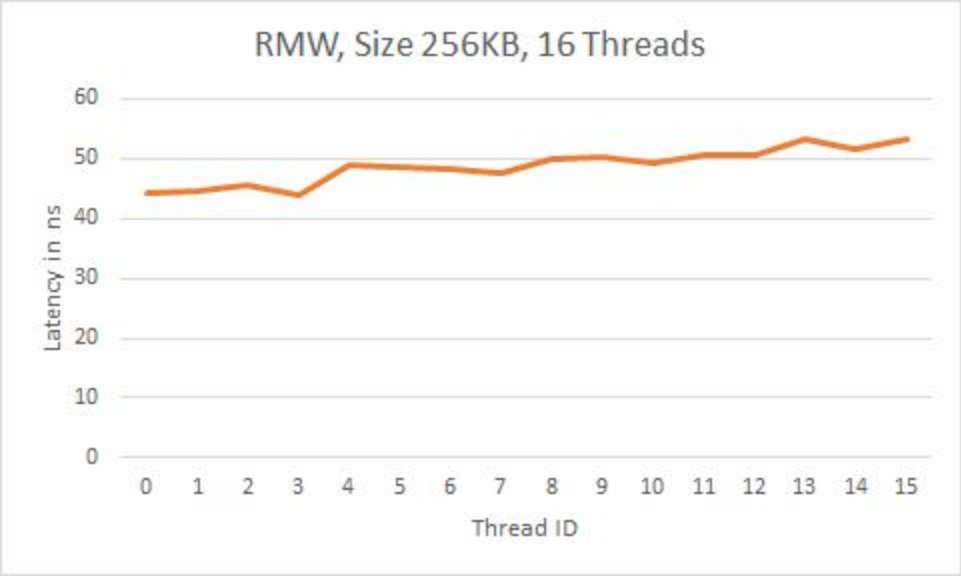

Read Only, Size 64KB, 16 Threads
Latency in ns
80
70
60
50
40
30
20
10
0
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Thread ID


Read Only, Size 256KB, 16 Threads
Latency in ns
50
45
40
35
30
25
20
15
10
5
0
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Thread ID


RMW, Size 16KB, 16 Threads
Latency in ns
120
100
80
60
40
20
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
Thread ID


RMW, Size 32KB, 16 Threads
Latency in
140
120
100
80
60
40
20
0
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Thread ID


RMW, Size 64KB, 16 Threads
Latency in ns
90
80
70
60
50
40
30
20
10
0
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Thread ID


RMW, Size 256KB, 16 Threads
Latency in ns
60
50
40
30
20
10
0
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Thread ID

3. **Analysis**

a) In the read only graphs, what are the significant regions of interest and what do they tell you about the cache and memory hierarchy. Can you see the differently sized caches (why or why not). What size caches would you guess are implemented in this machine based on your Graphs?

Considering the read-only graph, I can see the first spike in access time at 64KB. This means that the size of L1 cache is less than 64KB. Most likely, it is 32KB. The second spike can be seen approximately around 512KB, which points to the possibility of L2 cache having a size of 256KB. This spike is due to the fact that both L1 and L2 caches have resulted in a miss for the data and it needs to be fetched from the L3 cache. The next spike is seen at around 8192KB. This can mean that the size of L3 cache is less than 8192KB.

According to me, the spikes do not stand out because of multiple reasons.

- Even with random accesses, there is some form of hardware prefetching being done, which affects the latency values. I tried to disable hardware prefetching ***using x86 wrmsr*** instruction, however it did not have an effect. According to the discussion in [2], the hypervisor interferes with the impact of disabling hardware prefetching if done this way. I wasn't able to find a workaround for this.
- Several other factors contribute to the unimpressive trends in latency in the graphs. Speculation, runahead, inter-cache movement etc. contribute to this effect.

However, these graphs give a fair idea about the cache hierarchy and sizes, if not the absolute latencies.

b) What sized caches does the machine (or datasheet for this processor) indicate it has?

By running the cpuid instruction, I was able to get the cache configurations. These nearly match with the estimations mentioned in part a.

Cache ID 0:
- Level: 1
- Type: Data Cache
- Sets: 64
- System Coherency Line Size: 64 bytes
- Physical Line partitions: 1
- Ways of associativity: 8
- Total Size: 32768 bytes (32 kb)
- Is fully associative: false
- Is Self Initializing: true
- Maximum number of addressable IDs for logical processors sharing this cache: 2
- Maximum number of addressable IDs for processor cores in the physical package: 8
- Write-Back Invalidate/Invalidate acts upon lower level caches: true
- cache inclusiveness: Cache is not inclusive of lower cache levels
- Complex function for indexing used: false

Cache ID 2:
- Level: 2
- Type: Unified Cache
- Sets: 1024
- System Coherency Line Size: 64 bytes
- Physical Line partitions: 1
- Ways of associativity: 4
- Total Size: 262144 bytes (256 kb)
- Is fully associative: false
- Is Self Initializing: true

- Maximum number of addressable IDs for logical processors sharing this cache: 2
- Maximum number of addressable IDs for processor cores in the physical package: 8
- Write-Back Invalidate/Invalidate acts upon lower level caches: true
- cache inclusiveness: Cache is not inclusive of lower cache levels
- Complex function for indexing used: false

Cache ID 3:
- Level: 3
- Type: Unified Cache
- Sets: 8192
- System Coherency Line Size: 64 bytes
- Physical Line partitions: 1
- Ways of associativity: 12
- Total Size: 6291456 bytes (6144 kb)
- Is fully associative: false
- Is Self Initializing: true
- Maximum number of addressable IDs for logical processors sharing this cache: 16
- Maximum number of addressable IDs for processor cores in the physical package: 8
- Write-Back Invalidate/Invalidate acts upon lower level caches: true
- cache inclusiveness: Cache is inclusive of lower cache levels.
- Complex function for indexing used: true

c) In the read/write graphs, describe the shape of these graphs and contrast them to the read only graphs. Formulate a hypothesis on why the shapes are or are not the same.

a) The latencies in read-write graphs are slightly higher than the ones noted from read-only graphs. Since this operation constitutes a load and a store, the latencies are higher for RMW.
b) From the graphs it can be seen that L2 services more accesses in read-write case than in read-only case. It could be because L2 is a private cache whereas L3 is a unified cache. The modified block can be held in L2 with the dirty bit set and provided to lower caches or cores for each read/write operation. As it can be seen from the cpuid info, L2 is a WB cache, which makes the reasoning valid.

d) Which set (read only) or (read/write) seems more stable? Justify your answer.

The read only graphs look more stable than a read/write case. This is because simultaneous loads can be serviced from the same cache line without much delay as the block can be present in shared permission. However, for a read/write case, two or more cores waiting to access a part of the same block (false sharing) have to wait for the previous store to finish (at least reach a store buffer).

e) What other effects may you be observing in the graphs besides raw cache and memory Latency?

- As mentioned in part a of this question, hardware prefetching can be clearly seen in the way access latencies aren't spiking by a lot and aren't continuously maintaining a higher value. This means that some blocks are prefetched and we cannot correctly estimate the latency due to a miss.
- Another technique at work would be victim cache, which holds the value of recently evicted data and services requests with much lesser latency than expected.
- Load/Store buffers, speculation also contribute to the nature of the graphs.

f) What are some ways to validate or sanity check your results?

I ran cpuid function to get the details of the entire structure of the cache system, including block size and number of sets.

Apart from the actual configurations, there are few published benchmarks that can be used to verify the results obtained, e.g., lat_mem_rd.

4. Extra Credit

b) Compare results from EC2 and TSCC and justify what differences you see. How does virtualization overhead affect your results.

I did not use EC2, however on my own laptop I used WSL that uses Hyper-V architecture to enable virtualization. The overhead is quite significant.

I was getting values between *200ns to 400 ns on WSL*, whereas on a *native Linux machine* I got values between *30 to 80ns* for a certain type of access. However, the optimizations and prefetching effects for both the cases were the same.

References

[1] https://www.geeksforgeeks.org/maximum-number-threads-can-created-within-process-c/

[2] https://software.intel.com/en-us/forums/intel-isa-extensions/topic/785240