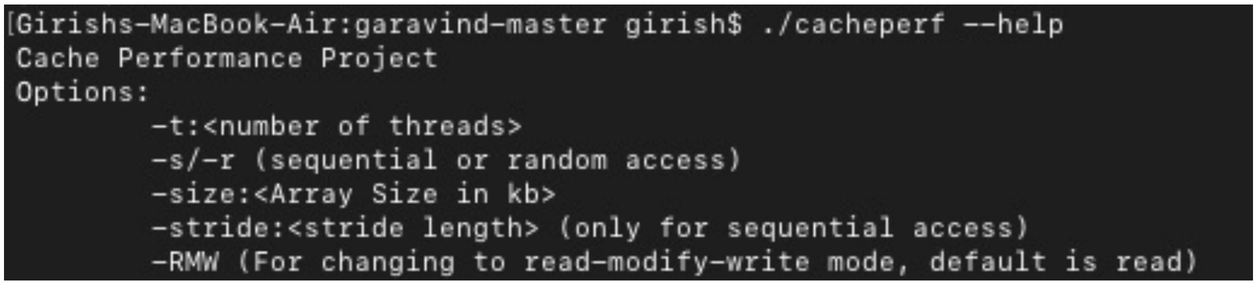
Characterize the cache performance of a CPU under shared and non-shared conditions

Name: Girish Gowtham Aravindan
PID: A53310752

1. **Document your program:**

A. How does your program measure latency?

As the first step, we are getting inputs from the user by providing options as shown below:



```
[Girishs-MacBook-Air:garavind-master girish$ ./cacheperf --help
Cache Performance Project
Options:
        -t:<number of threads>
        -s/-r (sequential or random access)
        -size:<Array Size in kb>
        -stride:<stride length> (only for sequential access)
        -RMW (For changing to read-modify-write mode, default is read)
```

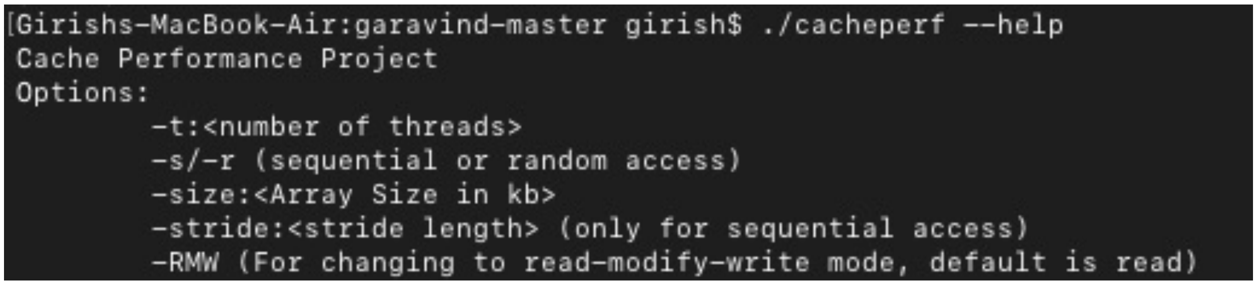
The cacheperf.c file first creates a linked list based on the inputs such as size of the data structure and the type of access (sequential or random access). In sequential access, the pointer will be pointing to a node sequentially based on the stride input given. Default stride value is 1, where the consecutive nodes of the inked list are accessed. In random access, the pointer can access any node in the linked list.

The performance is figured out as an implication of the latency of reading the nodes and read-modify-write of the nodes. The type of operation (Read or RMW) is obtained from the user as an input. The latency is found out by putting a stamp on the clock time at the start of the access and at the end of the access. The difference between the stamps is divided by the number of times the linked list has been accessed to get the latency value as shown below.

```
CPUTime = (1e9 * ( (double)(EndTime - StartTime)/CLOCKS_PER_SEC)) / ReadCount;
```

In RMW, in order to avoid compiler optimizing the loop, the program uses a set of commands as hinted in the project description.

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



```
[Girishs-MacBook-Air:garavind-master girish$ ./cacheperf --help
Cache Performance Project
Options:
        -t:<number of threads>
        -s/-r (sequential or random access)
        -size:<Array Size in kb>
        -stride:<stride length> (only for sequential access)
        -RMW (For changing to read-modify-write mode, default is read)
```

The options available are shown above.

-t:<number of threads>

Gets the number of threads as an input.

-s/-r (sequential or random access)

Gets the type of access. Sets the flag corresponding flag.

-size:<Array Size in kb>

Gets the size of the data structure as an input.

-stride:<stride length> (only for sequential access)

Gets the stride value as an input. Used to decide the nodes interleaved between consecutive node accesses. For example, if the stride is 4, every 4th node is accessed sequentially.

-RMW (For changing to read-modify-write mode, default is read)

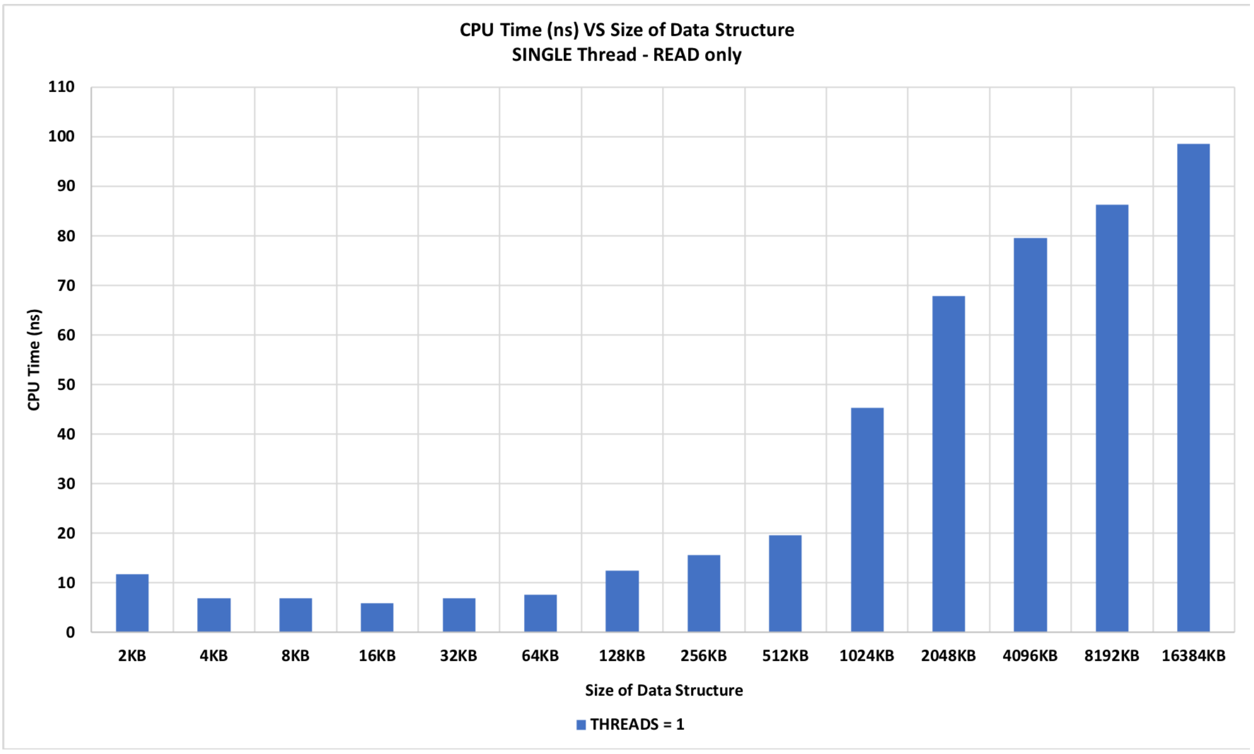
Gets the input for read of RMW operation. Sets the write flag.

C. What limitations does your program have?

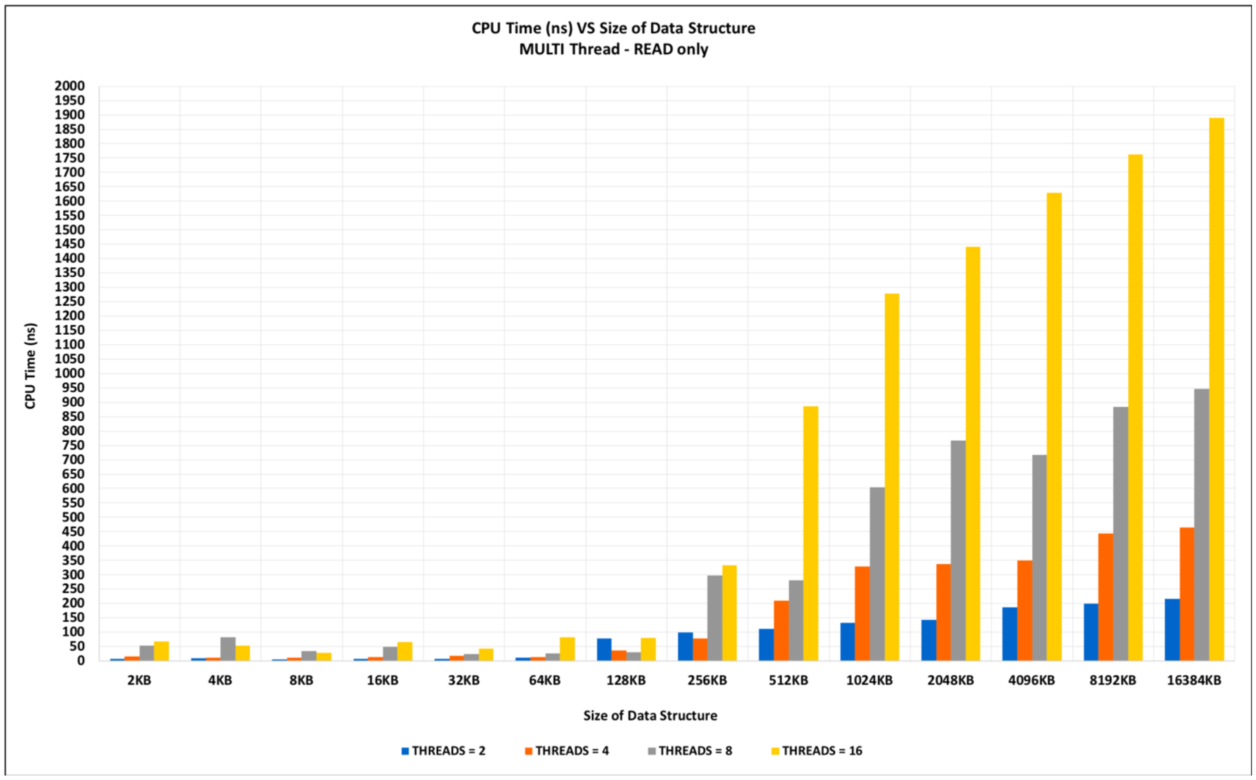
While executing the program for different sizes of data structure and number of threads, it is found that the program execution stalls if the number of threads is greater than 16. It is understandable because the memory allocation is heaped.

2. Create some graphs in your report that show:

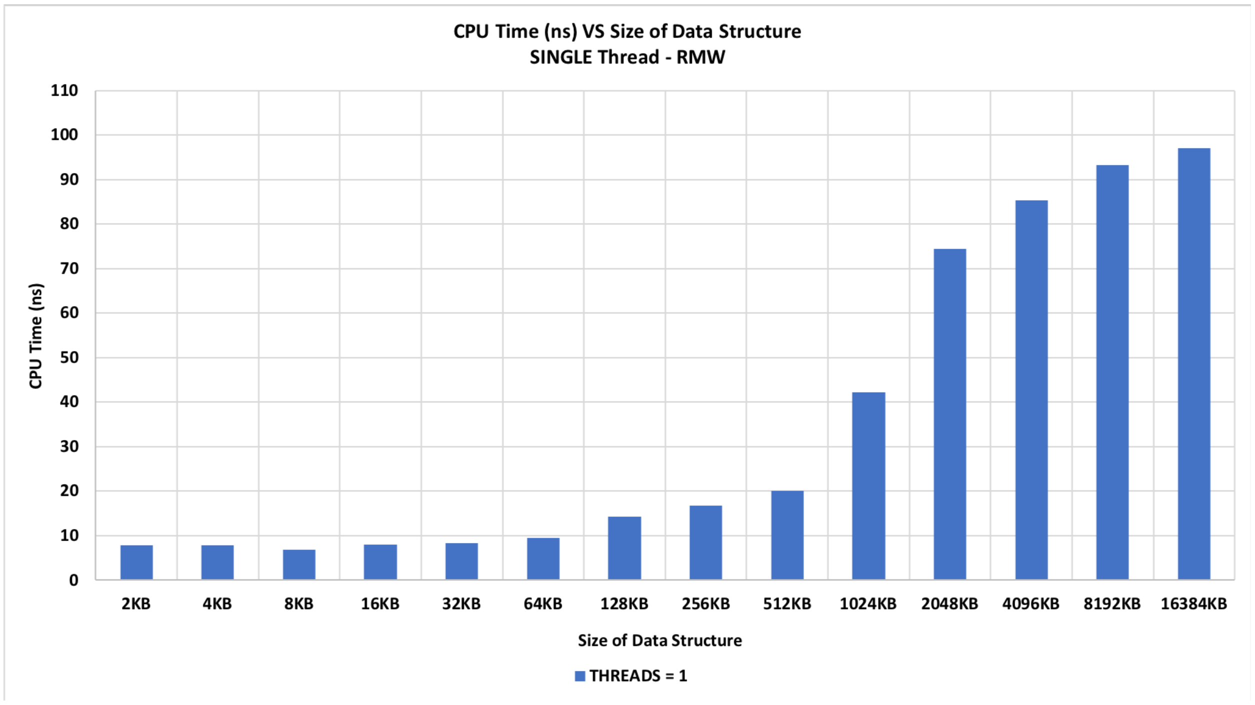
SINGLE Thread – READ only:



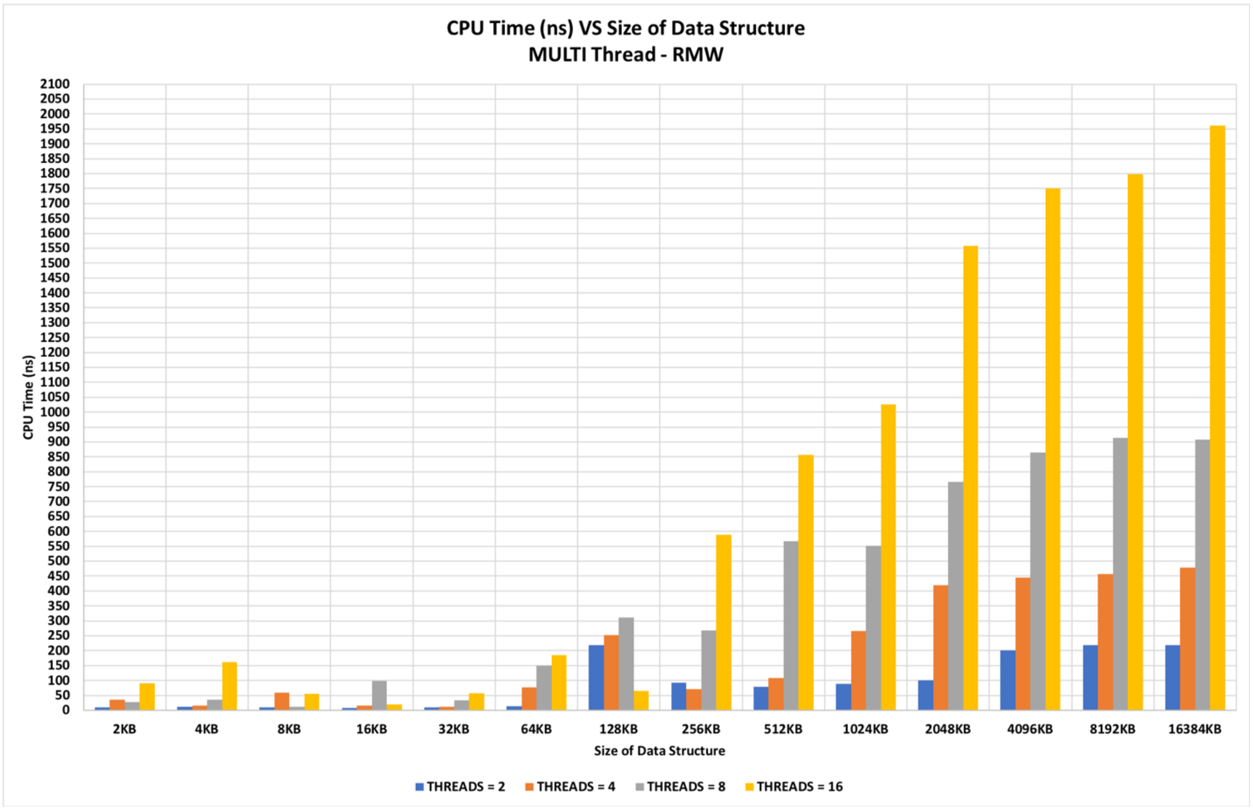
MULTI Thread – READ only:



SINGLE Thread – RMW:



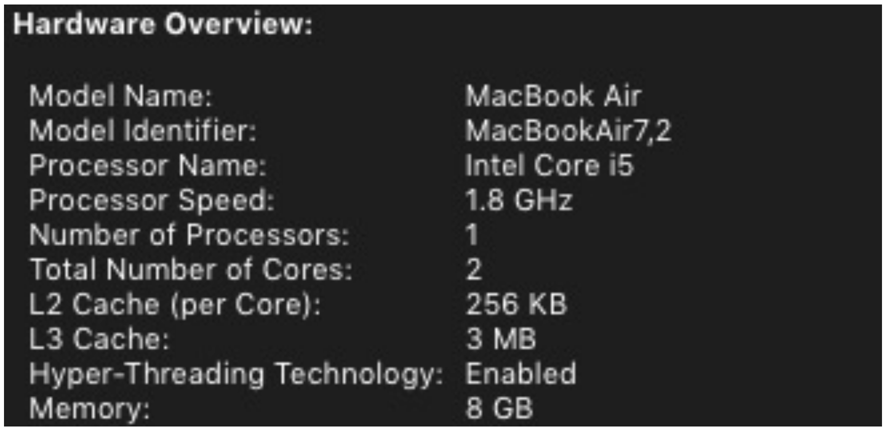
MULTI Thread – RMW:



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.**

The program is run on a machine with the specification shown below:



```
Hardware Overview:

  Model Name:                    MacBook Air
  Model Identifier:              MacBookAir7,2
  Processor Name:                Intel Core i5
  Processor Speed:               1.8 GHz
  Number of Processors:          1
  Total Number of Cores:         2
  L2 Cache (per Core):           256 KB
  L3 Cache:                      3 MB
  Hyper-Threading Technology:    Enabled
  Memory:                        8 GB
```

From the graph of SINGLE thread – READ only, it is seen that till the L2 cache size ie. 256 KB, the latency increases by a very small value. This is because the L1 I-cache, L1 D-cache and the L2 cache are on chip whereas the L3 cache is off chip. So, the latency increases by a large value as the access goes off chip.

B. **Can you see the differently sized caches? Why?**

The caches of different size are seen in the cache hierarchy. The size of the cache increases as the cache level increases. The L1 cache is kept small to reduce the latency of a cache hit and the size of L3 cache is kept large to reduce the cache miss rate.

C. **What size caches would you guess are implemented in this machine based on your graphs? What sized caches does the machine (or datasheet for this processor) indicate it has? Are they the same? If not, do you know why they are different?**

From the graph it is understood that the latency is almost the same till 32KB data structure size and then it increases at a very small margin till 256KB and then after that, the latency became very high. This shows that 32KB, 256KB are L1 and L2 cache sizes. This is same as the cache sizes mentioned in the hardware specs of the machine.

D. **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.**

The RMW graphs are showing the shape of an exponentially increasing function which is the case with read only graphs. The shapes are same because both the RMW and Read only operation involves memory/cache access, but the RMW operation requires memory/cache access twice (once to read and once to write), so the latency is more when compared with read only.

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

Read only looks more stable when compared with RMW as seen while comparing the MULTI thread – READ only and MULTI thread – RMW graphs. In RMW, when the data structure size is 128KB, the latency is greater than that of 256KB and 512KB. The read operation requires only the access to memory/cache once and so, accessing a node in a data structure is complete after the access is done. But for RMW, the access has to happen again in order to write the data back to the node. The graph for RMW is unstable because the latency could go high while waiting for the write back to node to be completed.

F. What other effects may you be observing in the graphs besides raw cache and memory latency?

It is seen that when the program is run while other applications such as safari, mail, calendar, etc. are open, the latency goes high. This is understood because the machine's cache is being used by other programs and so, the data required could be in a L2 cache instead of being in L1 cache.

It is also noted from the graphs that the latency is high for 2KB data structure when compared with the 4KB data structure. This is because the program is executed for the first time. So, we can also say that, had the program for various data structures been executed at different times (not continuously), then the results would have been even more unstable.

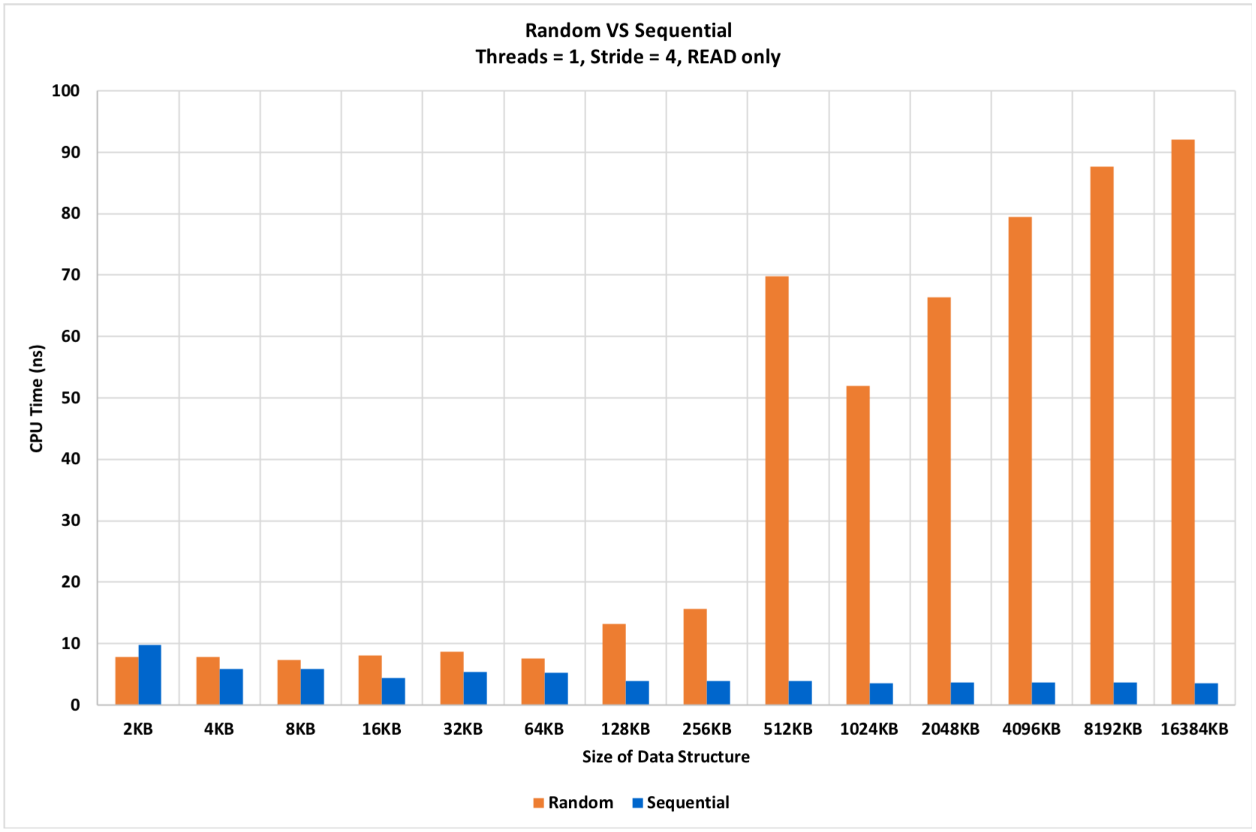
G. What are some ways to validate or sanity check your results?

It is difficult to check exactly the latency numbers because they are in Nano-second scale. But the trend of latency change can be checked and understood by comparing the results with those generated from another machine executing the same program.

Also, the trend can be understood by executing the program after quitting all other applications and comparing the results with those generated by running the program while some heavy applications are kept running. It can be seen that the latency increases in latter.

4. Extra credits

Modify your program and generate data that compares sequential (strided) access to non-sequential (pseudo-random access). Explain any differences you see in the latency results.



It is seen from the graph that the latency for sequential access is almost the same whereas the latency for random access increases exponentially with the size of the data structure. This is understood because when the access is random, a random generator function is used, and the pointer points a random node. But in sequential access, the pointer moves to consecutive nodes based on the stride value. If the cache features prefetching, then the latency gets reduced whereas prefetching is not effective in random access as we do not know the next node that is to be accessed.