CSE 240B Programming Assignment Report

- How does your program measure latency?

Ans: The program measures latency by first creating an array of linked list elements that are randomized or ordered sequentially based on the options presented to the program. For read latency the linked list is traversed through entirely and time to do it is logged. The read latency will be (time taken/size of linked list). Similarly for read/write latency a series of dependent RMW operations are performed where the data element of the previous linked list is used in conjunction with the current data element. Volatile elements are used to prevent any optimizing of code. The for loop does have a 'if' condition to check for the last linked list but that is considered to be negligible as the branch predictor would be able to easily predict the conditional.

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

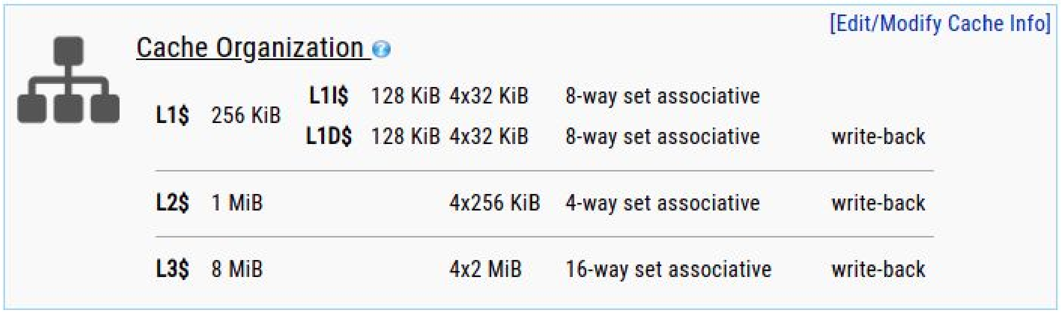
Ans: Main options used are `OPTS=-g -O2 -Werror -pthread` . -g is for debugging, O2 for compiler optimizations, Werror for warnings as errors and -pthread to link the pthread library.

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

Ans: The program can have as many threads as necessary (using a pthread **ptr to create an array of pthreads). Similarly the size of the array is also dynamic as the program takes that as a command line argument. Due to assigning them as 64 bit integers, thread size and array size are **theoretically** limited. As it uses pthreads, this program can only work in linux derived operating systems.

Computer Characteristics:

The computer on which these experiments were run has an i5-8300h processor. Memory hierarchy is as follows:

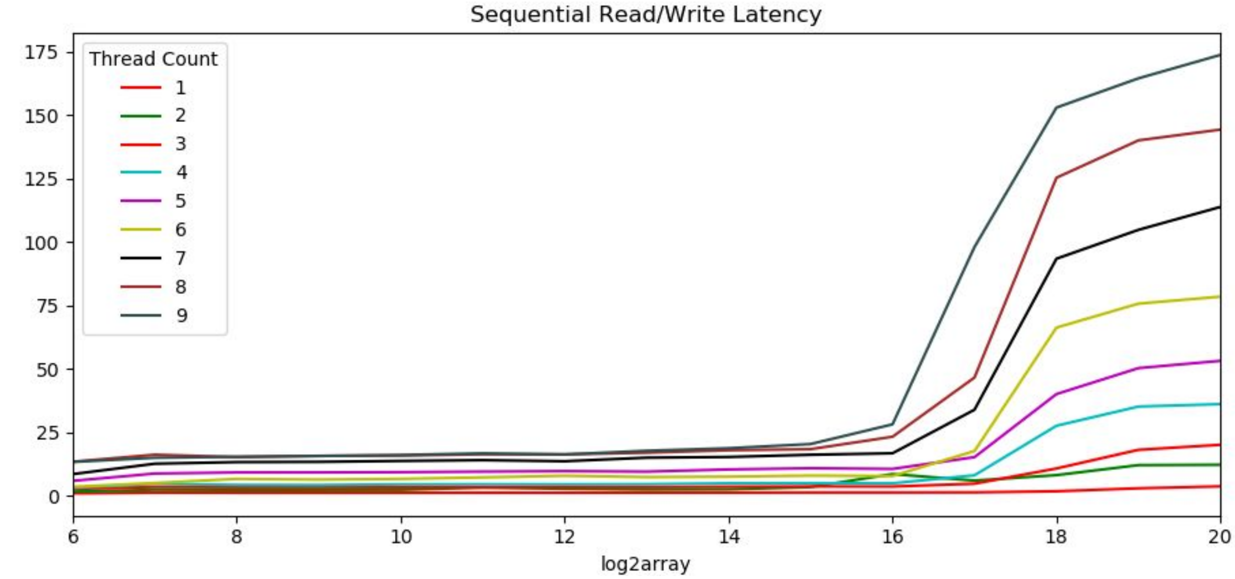


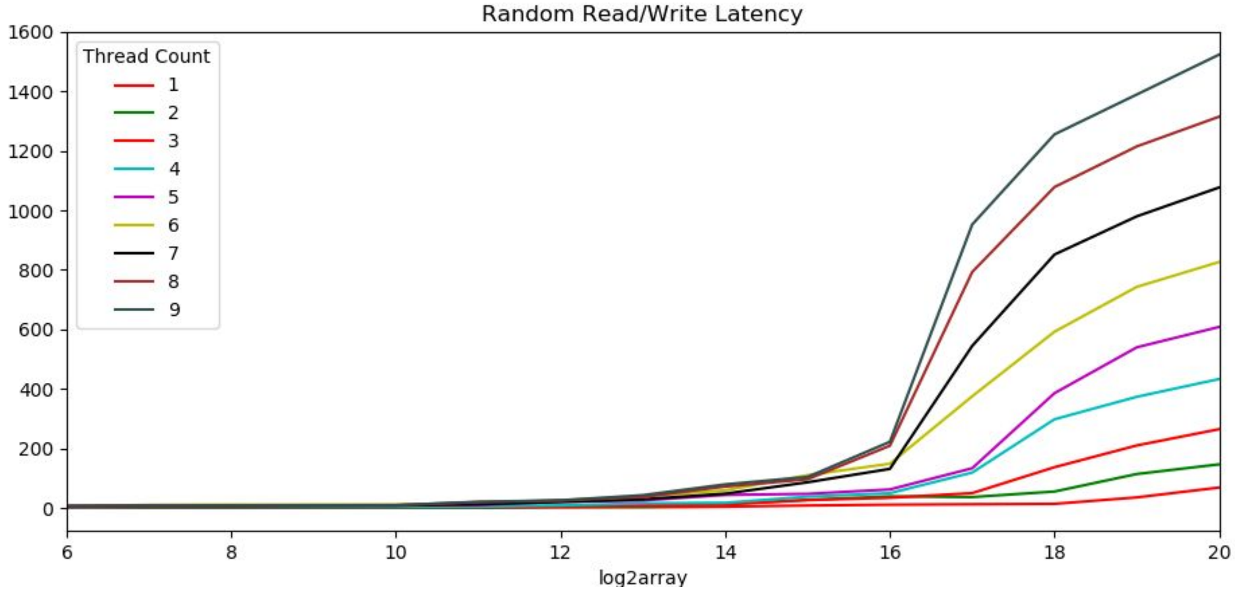
Cache Organization

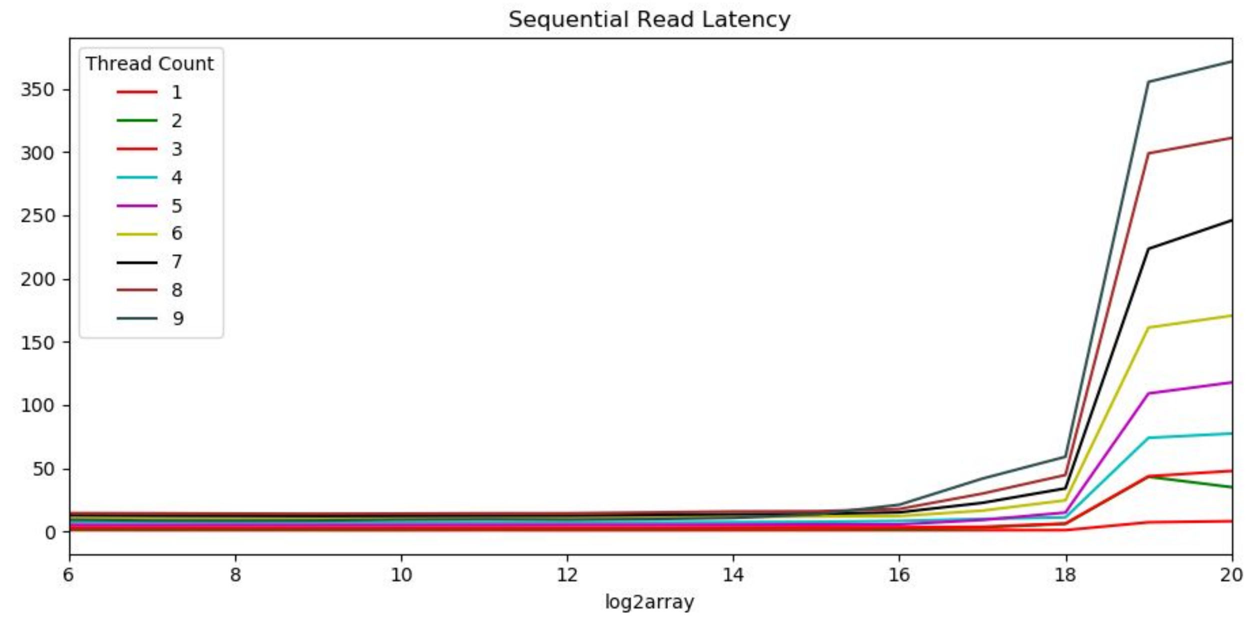
[Edit/Modify Cache Info]

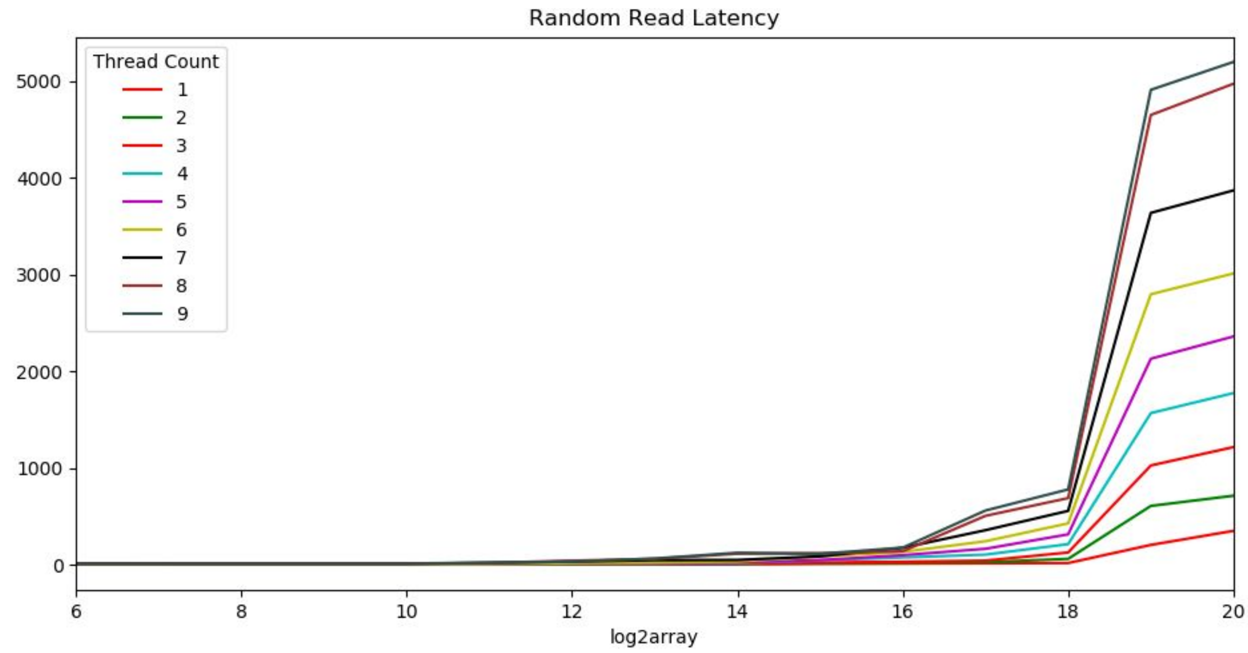
L1$	256 KiB	L1I$	128 KiB	4x32 KiB	8-way set associative	
		L1D$	128 KiB	4x32 KiB	8-way set associative	write-back
L2$	1 MiB			4x256 KiB	4-way set associative	write-back
L3$	8 MiB			4x2 MiB	16-way set associative	write-back

Graphs


Sequential Read/Write Latency
Thread Count
1
2
3
4
5
6
7
8
9
0
25
50
75
100
125
150
175
6
8
10
12
14
16
18
20
log2array

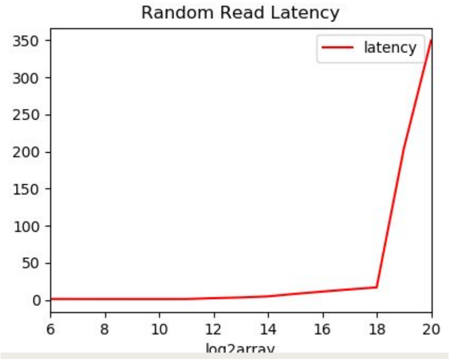

Random Read/Write Latency
Thread Count
1
2
3
4
5
6
7
8
9
0
200
400
600
800
1000
1200
1400
1600
6
8
10
12
14
16
18
20
log2array





Analysis

In the read only graphs, the significant regions of interest lie in the 16+ log2array part of the x axis. As we know, L1 dcache size is around 32*8*1024 (2^18) and L2 dcache is around 256*8*1024. When running the experiments we get this for 1 thread:



This indicates that for array sizes of 2^18 or 262144 we get negligible change but on log2 19 sized arrays there is significant overhead. This shows that the program is able to spot the L1 cache in the log2 range.

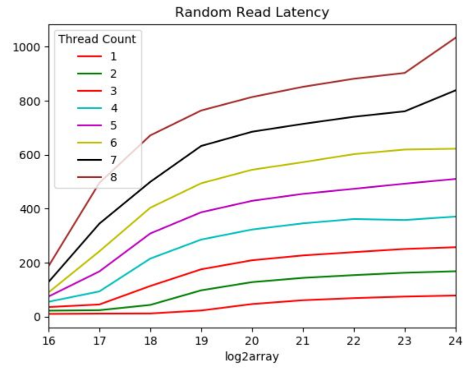
As seen in multiple threads, we can see the effect of thrashing among multi-threaded operations as effective latency increases as we increase threads. This also leads to an overall jump in latencies in all threads.

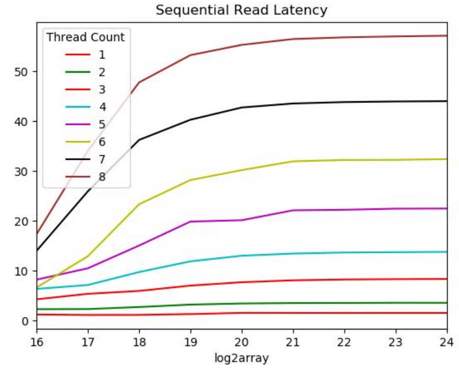
It is hard to observe the L3 cache but easy to say that as memory size increases in the L2 cache latencies increase linearly due to handling of data between L1 and L2. The rate of increase in latencies between arrays can give us information on whether the program is accessing higher caches.

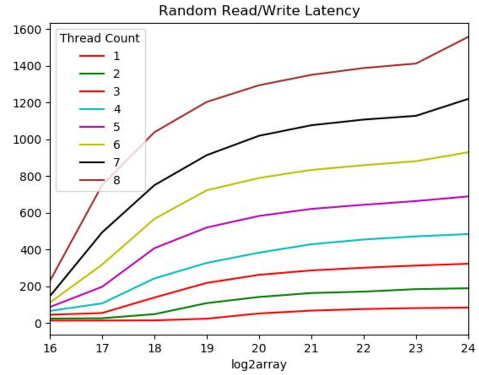
The read/write graphs seem a little different from read graphs as more memory is being stored in the read write graphs. According to the code, we read the linked list address, modify and write the uint32_t data. This adds to the memory hence we see that it takes lesser sized arrays to increase latency as we are storing the linked list and also modifying uint32_t data.

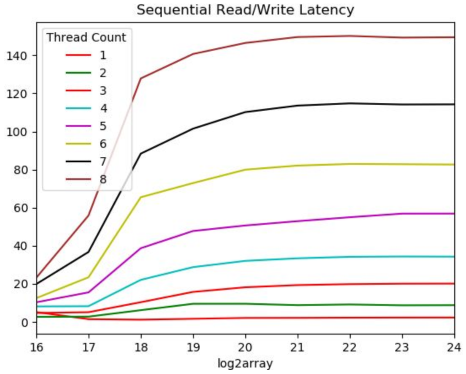
The random read graphs seem more stable due to their property of peaking on increase in memory. This makes it easier to spot the L1 - L2 cache usage easily.

To get a closer idea on higher size cache usage, we tested the cachetime on a size in the range 2^16 to 2^24 to check if it spots the L2-L3 cache. The results are not as explicit as L1-L2 difference but you do see a slight peak in a few of the graphs below.









As shown above, sequential access completely hides the L2-L3 hierarchy. We can see some evidence of the L2-L3 hierarchy in the Random Reads and Random R/W latency graphs in the 8 thread count but it could also be due to thrashing.

Extra Credit Work

When comparing sequential and non sequential accesses, we can observe that the time taken to get sequential accesses is way lower compared to random access as we have advanced prefetch mechanisms present in cache that hide the latency and make it easier to access more data in less time. We do observe the same kind of peaking after 18th power of 2 and a successive lowering of the rate of increase indicating that the L2 cache is still being used till the 2^20 array blocks.