CSE 240B PROJECT REPORT

Tanaya Kolankari – A53265700

Characterize the Cache Performance of a CPU Under Shared and Non-shared Conditions

1. Program Documentation

The key aim of the program is to characterize the performance of the memory hierarchy system that is present for the target processor that the program is being executed on. This basically implies that we want to determine the average memory access time for each entity that is part of this system namely, the L1, L2, L3 cache and the final main memory. We want the memory access time to be representative of the worst-case times to get a deeper understanding. For this, it is important to create the program flow such that the compiler or processor is unable to improve memory access times through intelligent code optimizations or prefetching. The program is aimed at minimizing the data reuse that is existent in the code.

The following section outlines the detailed implementation of the program that is used to characterize cache performance for this project.

This program mainly iterates through a linked list of length N, where in N can be varied based on the requirements of the experiment. In order to characterize the cache performance, the lower bound on N will be lesser than the L1 cache size and the upper bound will be greater than the L3 (highest cache) cache size or a desirable fraction of the main memory. Each node of the linked list consists of a 64-bit dummy data that can be utilized for read, modify and write calculations and a 64-bit pointer to another node in the linked list.

Measuring Latency –

A custom function 'time' has been defined in the program in order to measure the memory access latency. The logic implemented in the function is outlined below:

1. The size 'N' of the linked list is taken as an input in this function. Initially, we allocate memory required for N nodes of the list. Then, we create N pointers that are initialized to point to these nodes that we allocated memory for.
2. We do not want adjacent nodes in the linked list to be sequential so as to prevent prefetching. The current list of N pointers is sequential. Thus, prior to linking the nodes together, we randomly shuffle the list of pointers.
3. Now, the linked list is created by initializing the head of the linked list and setting the node->next pointer to the pointer that is next in the list of pointers created by random shuffling in step 2.
4. We move on to the actual time measurement. The chrono module's steady_clock is used for measurement of time in the program. The current time is stored in a start variable.

5. Each element in the linked list is accessed. In case of read only, we only use the pointers to navigate through the entire linked list, till we encounter a NULL pointer. For the read and write case, the data in the node is accessed, modified and written back and then we progress to the next node in the list.
6. Step 5 is repeated 'iters' number of times which is also an input to the function. Finally, the end time is noted using the steady_clock module. The difference in the start and end time is reported in nanoseconds which will correspond to the duration of the list accesses. This duration divided by the number of elements 'N' in the list and the iterations 'iters' corresponds to the average memory access time per element of the linked list. This value is returned by the function.

Program Options –

There are multiple options that are included in the program. However, these have not been introduced as compile time options in the current version. They need to be modified as required in the code prior to compiling and execution. The main options along with their significance have been outlined below:

1. _Number of threads_: This program also allows for multithreading. The number of threads is defined in the beginning of the program as - #define NUM_THREADS X. The value of X needs to be changed to the required value before compiling and running the program.
2. _Read modify write (RMW)_: The program can be run in two modes. In the read-only (default) case, only read operation will be performed on the linked list nodes. In the read-modify-write mode, the data in the node is accessed, modified and written back. In order to enable this mode, uncomment the #define RMW that is written in the beginning of the program.
3. _Linked List Size (N)_: The size of the linked list is varied over a range in the main function of the program. The range of variation is decided based on two variables in this function namely, minFactor and maxFactor. N will be varied from $2^{minFactor}$ upto $2^{maxFactor}$. Default value for minFactor is 6 and for maxFactor it is 30.
4. _Measurements per Size_: Another option that is introduced in the program is the number of measurements that are made per size. Instead of simply incrementing from minFactor to maxFactor for integer powers of two, a provision has been made to allow for fractional powers of two that will give a better idea of the overall trend. The number of divisions to be considered between factor1 and factor1 + 1 are determined by the variable steps in the main function. Default value for steps is 4.

The number of iterations for each size of the linked list are determined internally, requiring no user intervention.

Program Limitations –

The program does not have any significant limitations. The main shortcoming is that the options included in the program are not introduced as compile time or runtime options which makes the overall process of running different batches a little inconvenient.

2. Performance Reports

The information of the machine on which the code was executed, and performance was monitored has been outlined in Table 1.

Table 1: CPU Specifications

Architecture	x86-64
Number of CPUs	8
Threads per core	1
Model name	Intel (R) Xeon (R) CPU E5-2650 0 @ 2.00 GHz
L1d cache	32K
L1i cache	32K
L2 cache	256K
L3 cache	20480K

a. Performance reports for the read-only case

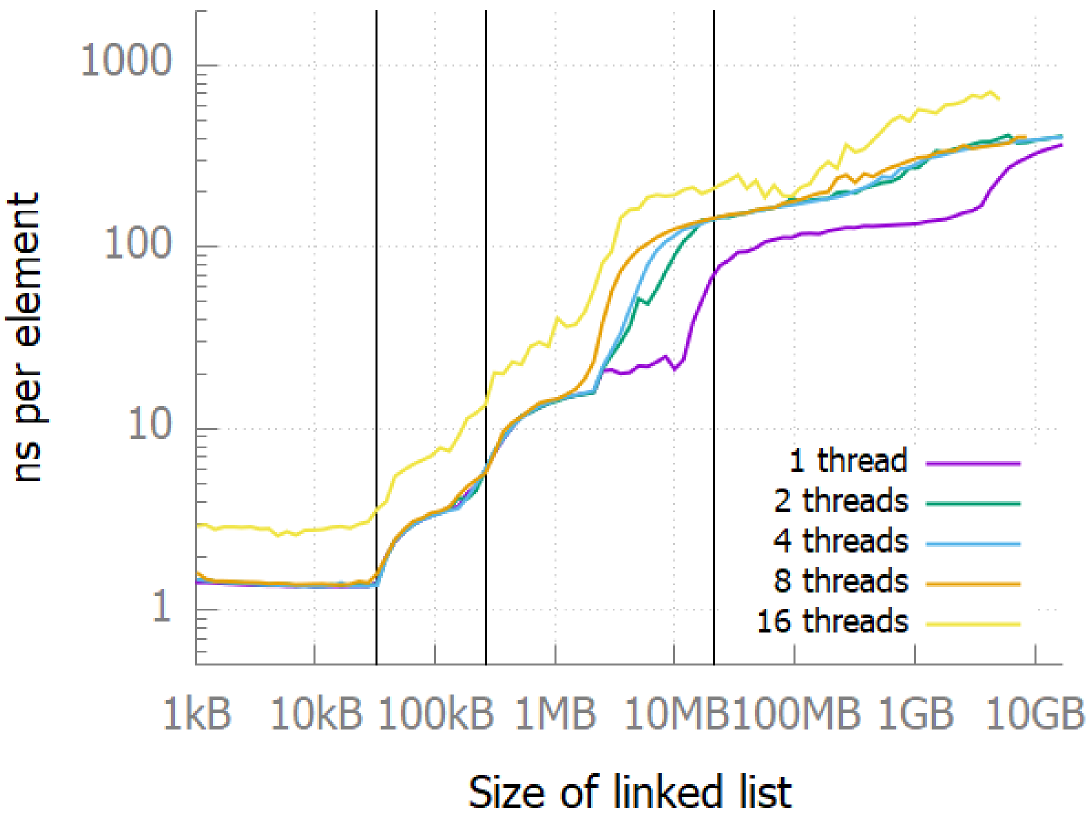

Figure 1: Variation of memory access latency with number of threads for read-only case

Please note that the three black standing lines denote the L1, L2 and L3 cache sizes respectively in all of the reported graphs.

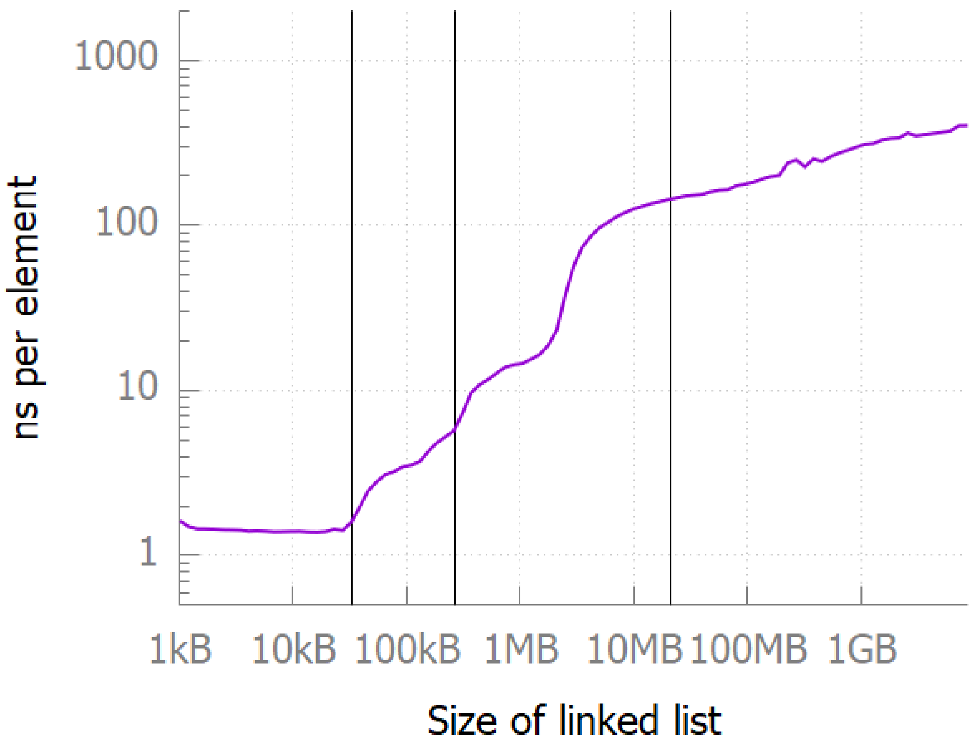

Figure 2: Variation of memory access latency with size of linked list for read-only case, threads=8

b. Performance reports for the read-modify write case

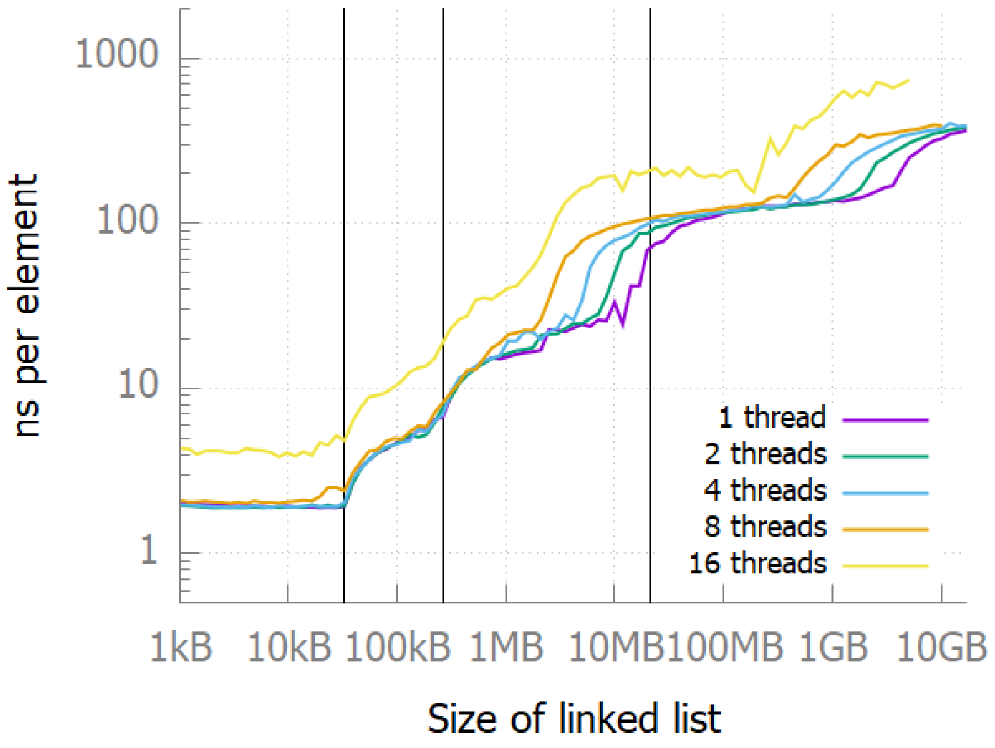

Figure 3: Variation of memory access latency with number of threads for read-modify-write case

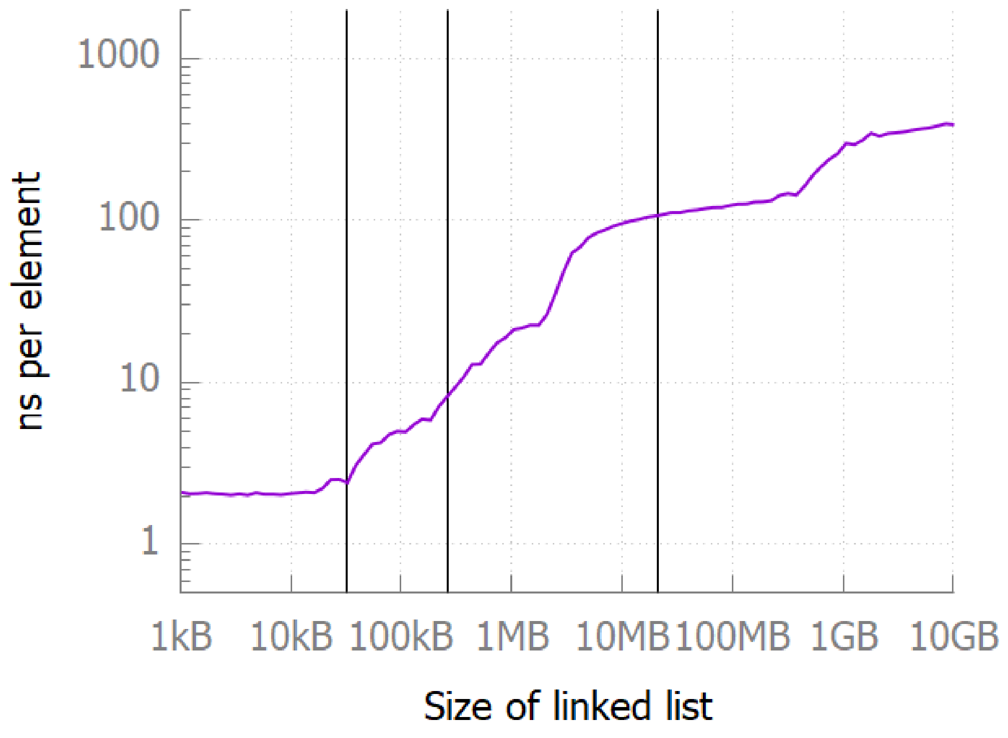

Figure 4: Variation of memory latency with size of linked list for read-modify-write case, threads=8

c. Performance comparison for read-only and read-modify-write case

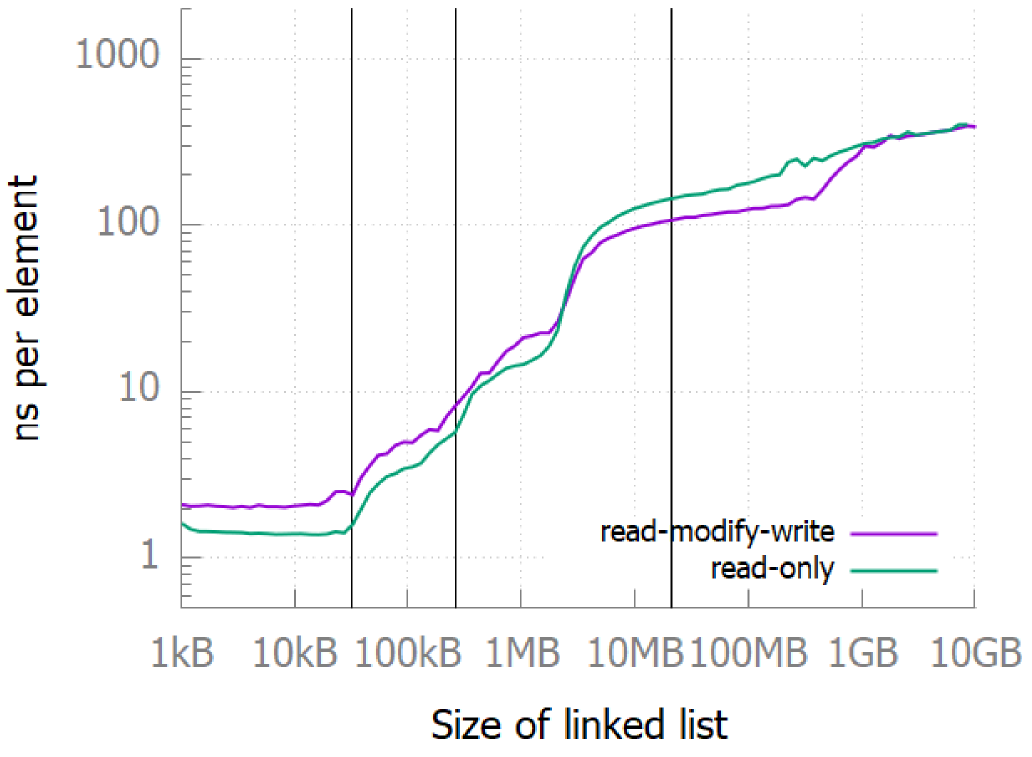

Figure 5: Comparison of latency for read-only and read-modify-write case for threads=8

3. Analysis

a. Analysis for read-only performance reports –

Analysis of the memory access latency with varying size of the linked list gives an idea of the performance of the cache hierarchy of the target processor. Initially, when the size of the linked list is lesser than that of the L1 cache (32KB) we can see that the access latency per element is extremely consistent and which can be considered to be equal to the L1 cache access latency. As the same linked list was accessed multiple times, it must have fit completely into the L1 cache post the first complete access. After that, every time the nodes in the list were accessed, it was a hit in the L1 cache. Now, as the size of the linked list increases beyond the capacity of the L1 cache, we see a steady increase in the access latency. There is initially a steeper increase in the latency that stabilizes as the size of the linked list is brought closer to the L2 cache size. A probable reasoning for this could be that initially only a small portion of the linked list must overflow into the L2 cache. As the size of the linked list increases, this portion also goes on increasing. This results in the expected increase in access latency. However, as the average access time approaches the L2 cache access time, with increasing linked list size, we see a stability in the access times. This exact trend is also observed as we increase the size of the list further from the L2 cache size towards the L3 cache size. As the size of the list is increased beyond the L3 cache size, the accesses will also be transferred to the main memory more frequently. Owing to this, there will be a relatively consistent increase in the memory access latency till the complete main memory is filled up by the list.

Further, we look into the variation in the latency with increasing number of threads. When the number of threads is less than or equal to 8, we see a consistent pattern in the memory latencies. Each thread can be expected to run on a separate core based on the CPU specifications that were outlined in Table 1. Additionally, each core has a separate L1 and L2 cache. Thus, as long as the size of the list is less than the L2 cache size for individual core, we observe that the latency is similar to the single thread case. However, as soon as the list size approaches the size of the L3 cache we observe an increase in latency with increasing number of threads. This is because multiple parallely running programs are utilizing the same cache for access. This will result in increased contention in case of higher number of threads. The latency values again become similar for all list sizes once the size exceeds the capacity of the L3 cache as well. In this case, the main memory is the bottleneck which will be similar irrespective of the number of threads running as long as they do not completely overfill the main memory as well. The underlying CPU mostly has provision only for maximum of 8 threads. Hence, once the number of threads is set to 16, we can observe a complete shift in the performance graph. This will imply that the threads introduce an additional overhead in the implementation. Also, if two threads need to be mapped to a single CPU, they will have to share the L1 and L2 caches as well, resulting in greater contention. This would result in an increase in the access time per element as the probability of a cache miss increases with increasing threads beyond 8.

In general, some extreme variations are observed at some points in the reported results. These sudden variations can be ignored as the program was being run on a server that had multiple jobs running on it as well. Thus, some sharp variations can be ignored while focusing on the general trend.

b. Analysis of read-modify-write performance reports –

The general trends in case of read-modify-write are found to be similar to the ones observed in case of the read-only results. Additionally, the distinction in performance with varying number of threads is more prominent in case of read-modify-write. One probable reasoning for this is that as threads modify entries in the list, it requires updates in the main memory as well. It will bring into action the consistency protocols that will further highlight the worst-case behavior of the memory hierarchy. Additionally, this should also increase the absolute access times for read-modify-write as compared to read-only case.

c. General Analysis –

In conclusion, the trend seemed to be more stable in case of the read-only case as compared to read-modify-write. The probable reason for this can be that the write-back is initiated under replacement scenarios only and hence the amount of write-back varies based on contention and the size of list. On the other hand, the behavior in case of read-only is more predictable resulting in a more stable performance trend.

We observe an interesting trend when comparing the read-only case with the read-modify-write case. Originally, we would expect the latency to be higher all throughout in case of the read-modify-write case. This trend is true as long as the size of the linked list is lesser than the L2 cache size. As the size increases further, we observe that the latency increases in case of the read-only case. The reasoning for this is slightly ambiguous but could be attributed to the fact the L3 cache is shared among all the cores. Thus, the initial disadvantage that was prevalent in case of the individual caches now turns into an advantage as all threads can access the changed values in the list. Additionally, a write could ensure longer retention of the value in the L3 cache as against the read-only case.