CS810: Advanced Computer Architecture

Assignment 3

25 September 2023

Shashank P (200010048), Tabish Khalid Halim (200020049)

The objective of this assignment was to use the **Tejas** tool to simulate the running of a program on an actual CPU and obtain the corresponding performance. The tool allows us to modify the CPU specifications run various trace files, and find the bottleneck in a CPU.

1 Introduction

Our experimental setup consisted of a set of a fixed number of CPU specifications to change and a fixed number of output metrics to record. The CPU specifications that we changed from the basic **Tiger Lake** configuration are shown below:

Branch Predictor Mode, Memory Latency, Core Frequency, BHR Size, Main Memory Frequency, ITLB size, DTLB size, Integer Vector Multiplication Latency, Float Multiplication Latency, Float ALU Latency, FMA Latency

The Tejas tool gave lots of performance metrics as an output; among those, we chose the following performance metrics for analysis:

IPC, Branch Prediction Accuracy, Time Taken, Core Energy, Main Memory Controller Energy, Core Power

2 Experimental Setup

In our experiment, we kept every specification constant except one. We varied on specifications and ran the tool. The values used for each specification are shown below.

- **Branch Predictor Mode:** All of the given predictors
- **Memory Latency:** 10 to 500 cycles
- **Core Frequency:** 100 to 5000 MHz
- **BHR Size:** 2 to 64 bits
- **Main Memory Frequency:** 100 to 5000 bits

- **ITLB size:** 10 to 500
- **DTLB size:** 10 to 500
- **Integer Vector Multiplication Latency:** 1 to 20
- **Float Multiplication Latency:** 1 to 10
- **Float ALU Latency:** 1 to 10
- **FMA Latency:** 1 to 10
- **Cache Size:** Depending on the type of cache
- **IW Size:** 50 to 500

The files contained **Intel Pin** traces of **SPEC CPU 2017** benchmarks from the previous assignments. For each benchmark, 10 $million$ instructions were run on Tejas.

3 Approach

We have made a loop that runs all the possible combinations of inputs and outputs to generate the stat file. One of those commands is shown below

```
bash copy.sh
```

```
java -jar ~/cs810_resources/Tejas/jars/tejas.jar \
         ./config/gcc_ICache_32K_8-ReadLatency_20.xml \
         ./stats/gcc_ICache_32K_8-ReadLatency_20.stat \
         ~/cs810_resources/CPU2017_benchmarks/tejas_traces/gcc
```

This will generate the stat file for the **Read Latency** variation of **ICache** with **20** cycles on **gcc** benchmark. The statistics file is then parsed using **Regular Expressions** as shown in the submitted code to get the required metrics. **parsed using python** and graphs are plotted with various **x** and **y** axis metrics.

4 Plots

Among the combinations of Specifications, Metrics and Benchmarks, we plotted around $70 - 80$ The plots are shown below with appropriate headings. The x and y axes and legends are mentioned in the plots.

4.1 Branch Predictor Accuracy vs Branch Predictors

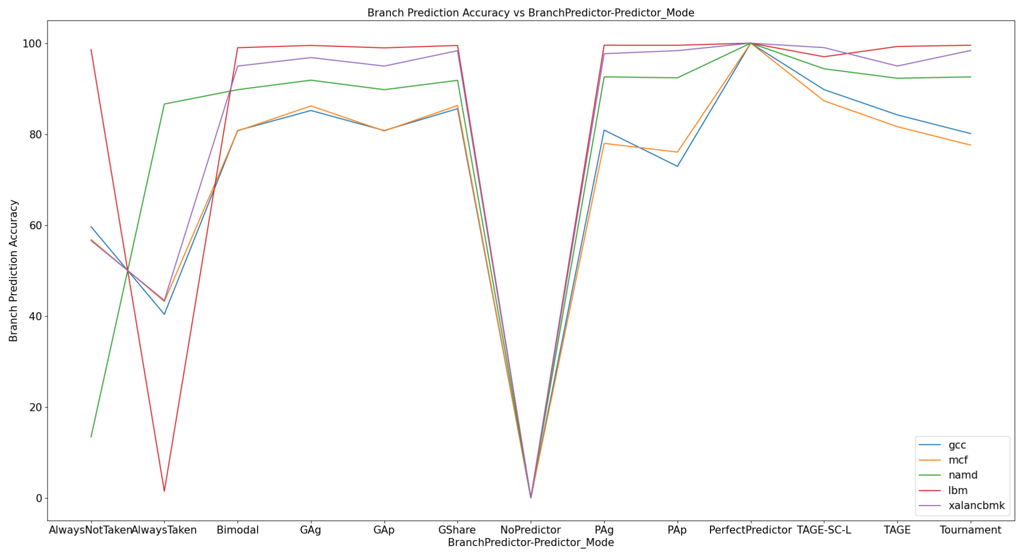

Figure 1: Graph of Branch Accuracy vs Branch Predictor for Various Benchmarks

- From the un-scaled data, we have found that the Perfect branch predictor is as expected, giving 100% branch prediction accuracy for all the benchmarks followed by its contenders TAGE-SC-L, TAGE, G-Share, GAg, GAp, Bimodal, Tournament, PAg, PAp branch predictors respectively.
- The lbm benchmark shows high branch prediction accuracy for the Always Not taken predictor; this can be because the branch instructions in the lbm benchmark maybe in a loop that executes many times without actually taking the branch.

4.2 Core Energy vs Branch Predictors

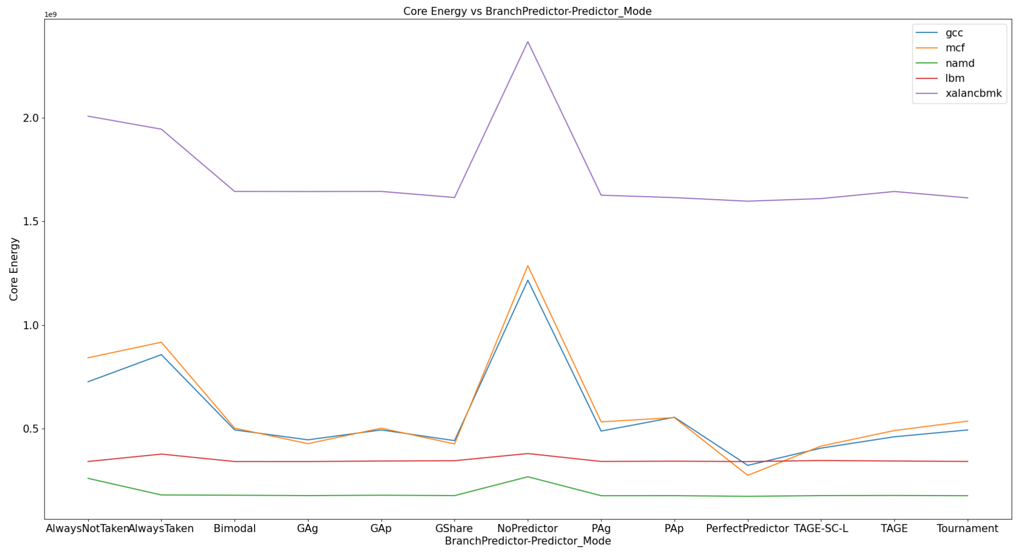

Figure 2: Graph of Core Energy vs Branch Predictor for Various Benchmarks

- From the plotted data, we can see that the No predictor consumes the most amount of energy throughout all 5 benchmarks. This can be because the no predictor may stall inflow of instructions into the pipeline as branch is unresolved. Until the branch instruction has completed it's execution in EX stage then the newer instructions come into the pipeline.
- The namd benchmark consumes the least amount of energy and the xalancbmk consumes the most amount of energy across all predictors. This can be because namd may consume more CPU cycles but potentially less energy per computation due to the high computational throughput or the degree of parallelization of namd is higher than xalancbmk.
- GCC and mcf follow a similar trend in the graph, this can be because of the identical nature of ILP exploits or identical nature of instructions.
- The Perfect predictor consumes the least energy across all the benchmarks.

4.3 Time Taken vs Branch Predictors

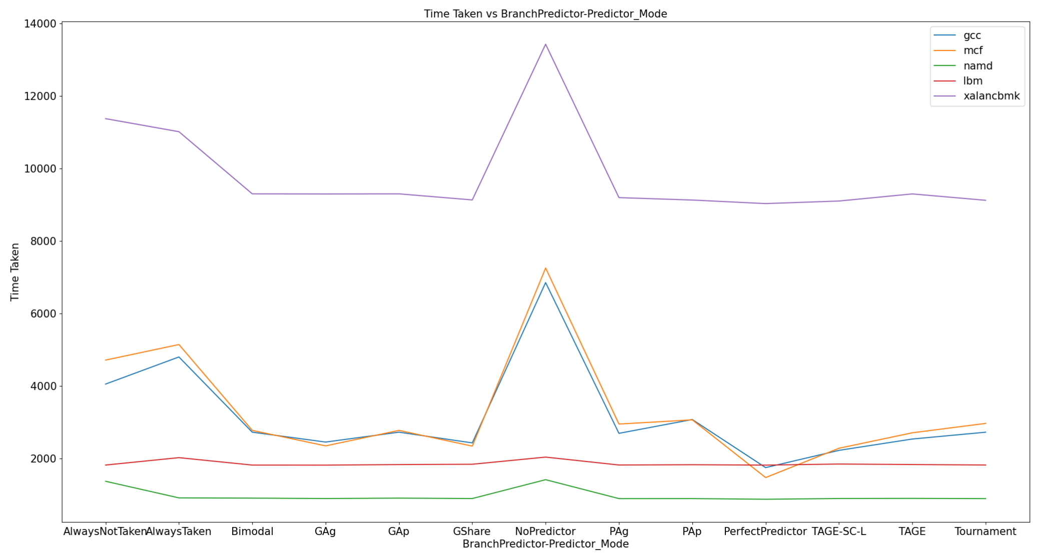

Figure 3: Graph of Time Taken vs Branch Predictor for Various Benchmarks

- From the plotted data, we can infer that the graph follows the same nature as discussed in Core Energy vs Branch Predictors section. This can be because because the power consumption is consistent throughout so energy consumption and time taken converge to get consistent power output.

4.4 Core Energy vs Core Frequency

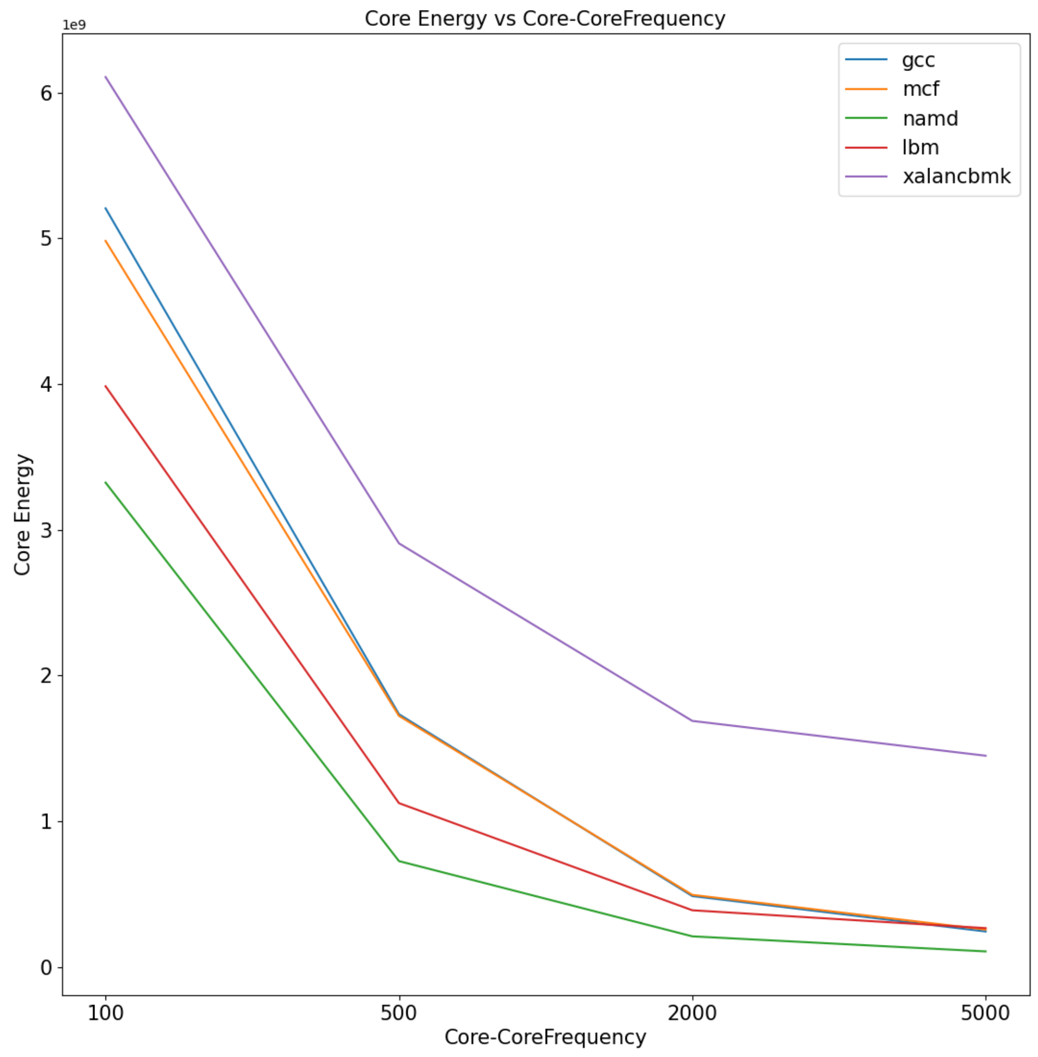

Figure 4: Graph of Core Energy vs Core Frequency for Various Benchmarks

- From the graphs depicted above, we can infer that as the core frequency increases the core energy consumption decreases. This can be because the power consumption of the core is proportional to the clock frequency but as more time is taken to fulfill that particular task the energy consumption turns out to be very high thus, we are left with this down slope result.

4.5 IPC vs Branch Predictors

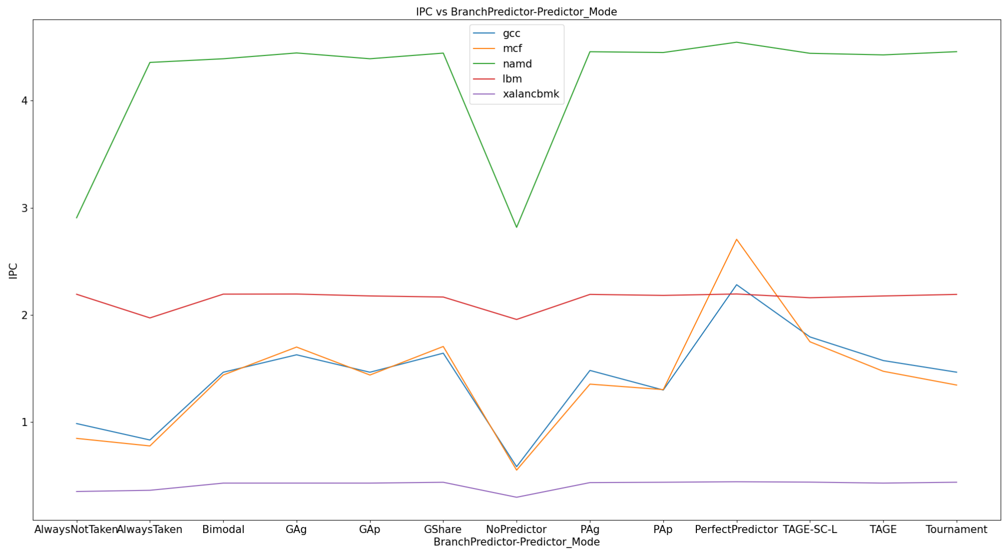

Figure 5: Graph of IPC vs Branch Predictor for Various Benchmarks

- The IPC of the No predictor is the least across all benchmarks as expected, as there is no instruction coming into pipeline until and unless we resolve whether the branch instruction is taken or not.
- The IPC of the perfect predictor is highest which is to be expected.
- The IPC of lbm benchmark using the Always not taken predictor is unusually high, this can be because the branch prediction accuracy may be high.
- The IPC of GCC and MCF benchmarks of Always not taken vs Always taken predictors is higher, this can be because maybe the loops that executes many times without actually taking the branch.

4.6 IPC vs Main Memory Frequency

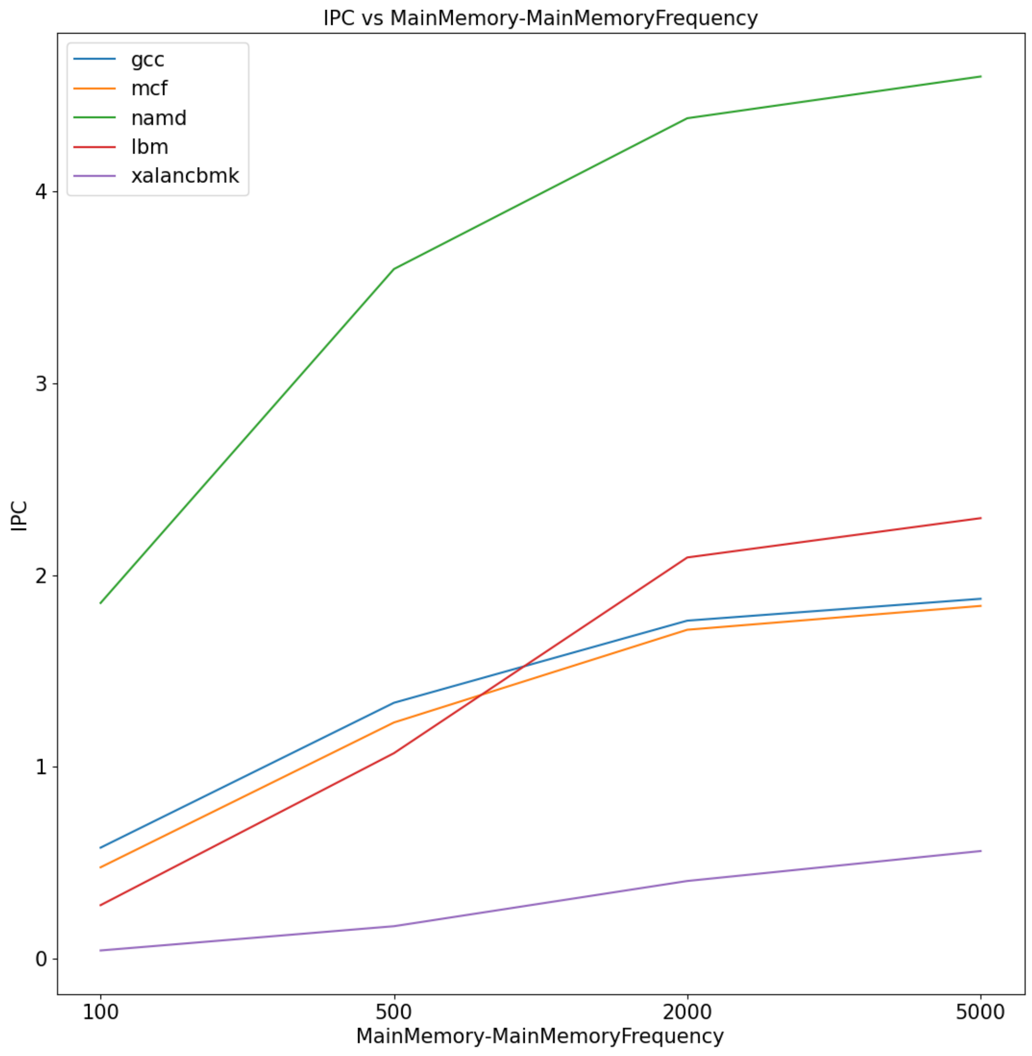

Figure 6: Graph of IPC vs Main Memory Frequency for Various Benchmarks

- We observe an upward slope trend for IPC vs the main memory frequency suggesting that higher the memory frequency, the more higher the IPC.

- The performance of the lbm benchmark greatly exceeds the casual trends of the other benchmarks. This can be because of the large amount of memory access it may perform internally in it's algorithm.
- namd benchmark has the best IPC value whereas xalancbmk has the least IPC suggesting that time taken to execute by namd may be lesser than that of xalancbmk.

4.7 Time Taken vs Float ALU Latency

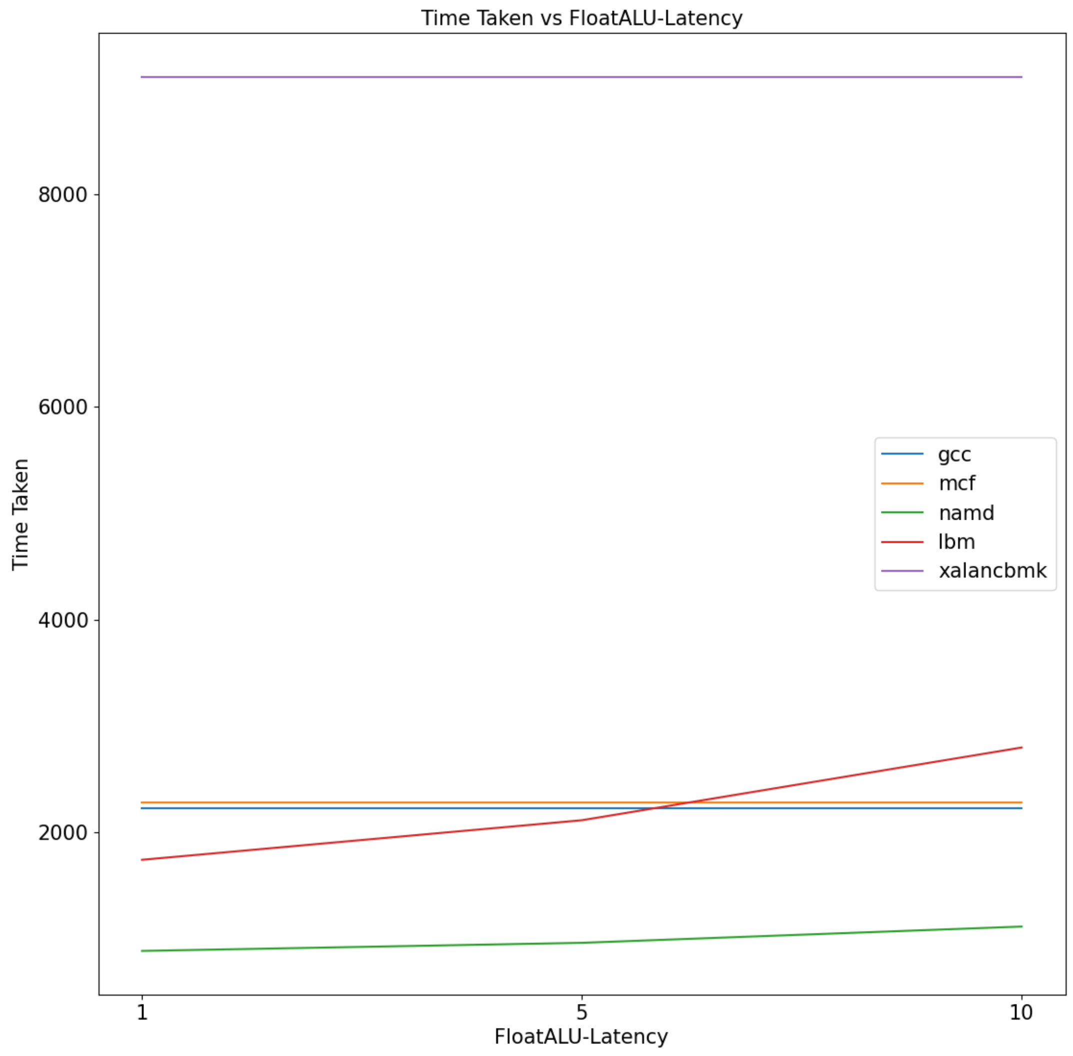

Figure 7: Graph of Time Taken vs Float ALU Latency for Various Benchmarks

- We observe a constant relationship of the form (y = constant) across most of the benchmarks.
- However, the lbm benchmark does not follow this trend, as we increase the latency on x axis,

the time taken increases. This may suggest us that the lbm may have a lot of floating point based arithmetic instructions that get executed. Increasing latency would mean lowering the IPC and increasing time taken to execute our program.

- namd benchmark has the least execution time whereas xalancbmk has the most time taken.

4.8 Time Taken vs Main Memory Latency

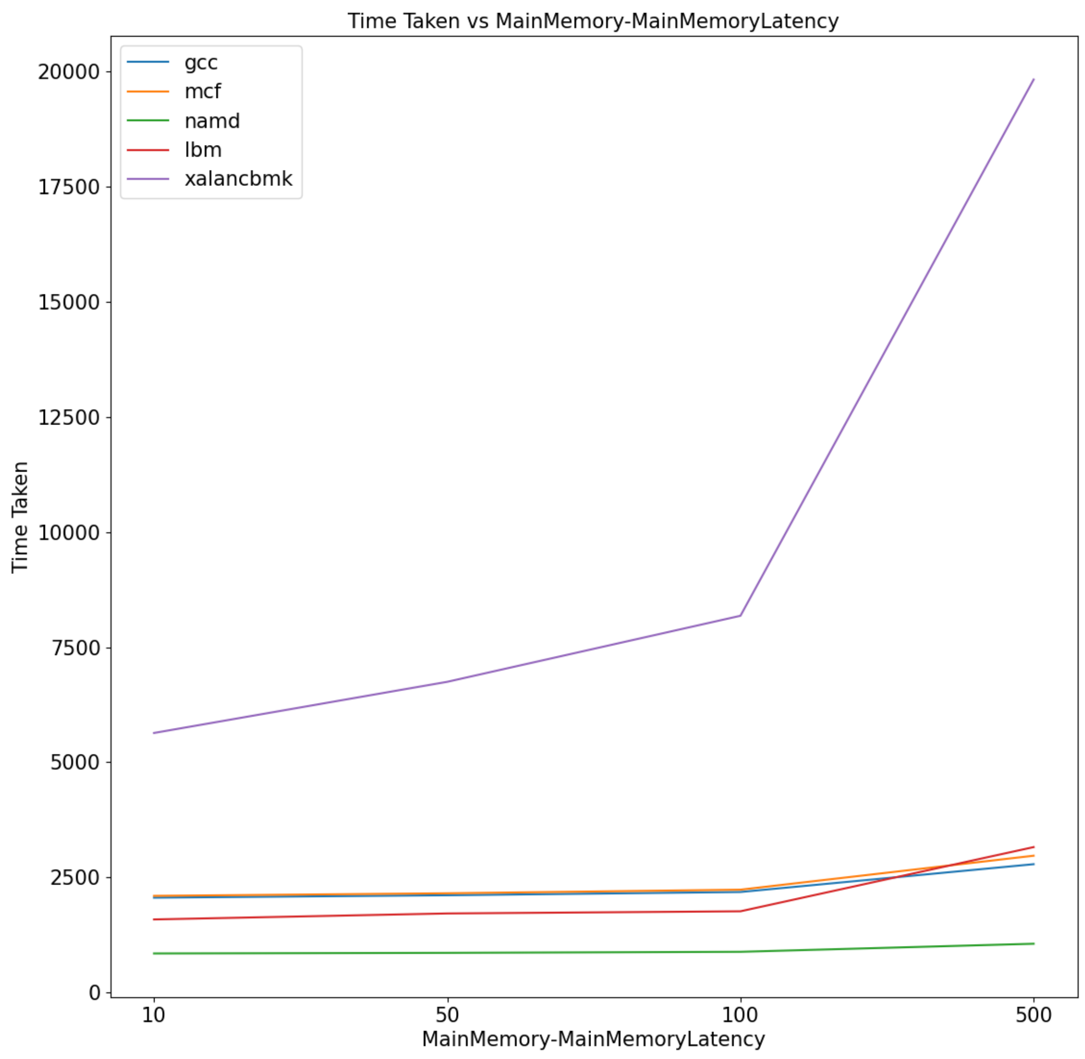

Figure 8: Graph of Time Taken vs Main Memory Latency for Various Benchmarks

- We observe a general trend between time taken and main memory latency arising due to the request for access of data at memory addresses. The general trend shows that the time taken increases as memory latency increases

- The xalancbmk benchmark is most affected by this cause as we see an increasing spike beyond main memory latency (x) = 100 cycles. The namd benchmark stays near constant suggesting not much presence of memory based instructions.

4.9 Time Taken vs Main Memory Frequency

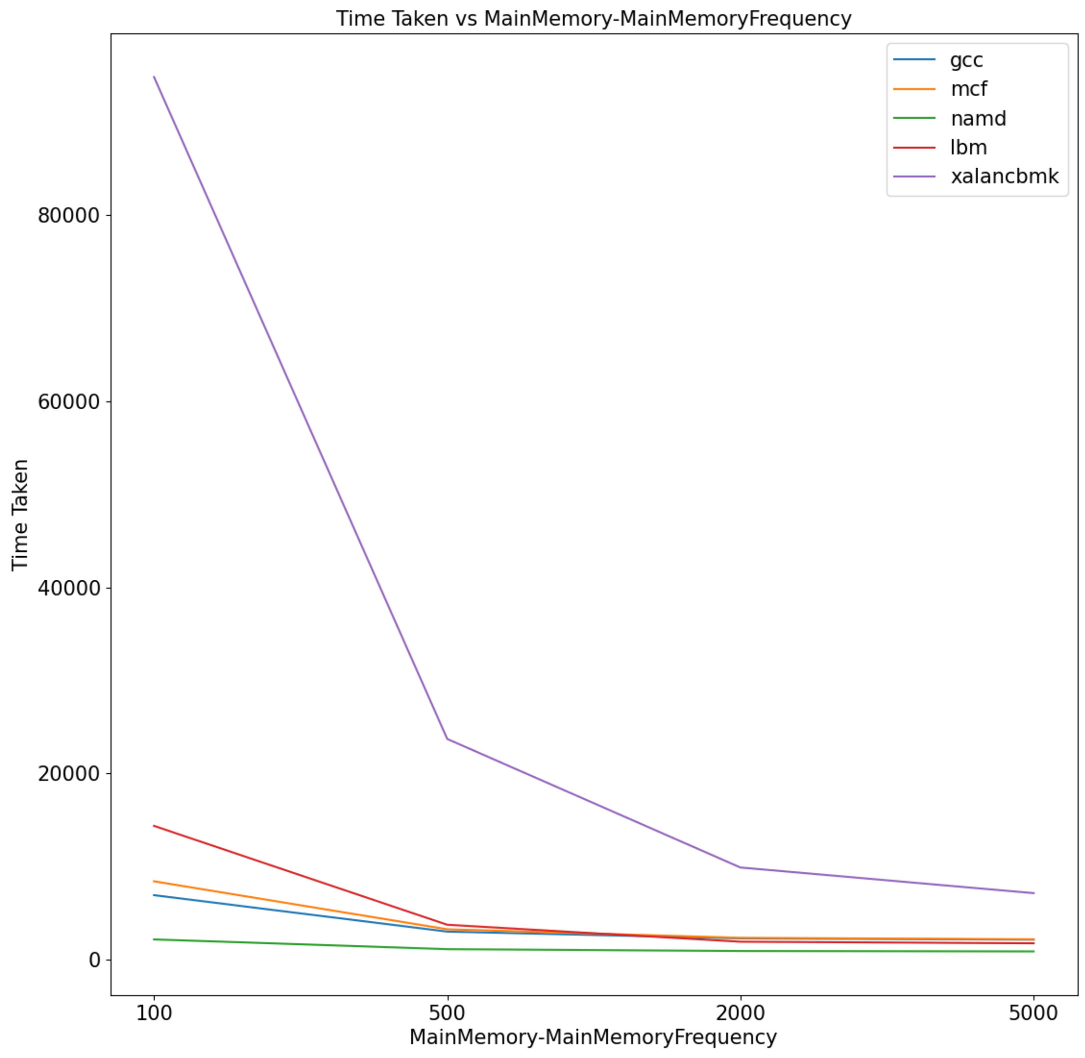

Figure 9: Graph of Time Taken vs Main Memory Frequency for Various Benchmarks

- We observe a general trend between time taken and main memory latency arising due to the request for access of data at memory addresses. The general trend shows that the time taken decreases as memory latency increases.
- The xalancbmk benchmark is most affected by this cause. The namd benchmark stays near constant, suggesting not much presence of memory-based instructions. This follows our previous inferences for Time taken vs Main Memory Latency.

4.10 Core Energy vs L3 Cache 3M 12

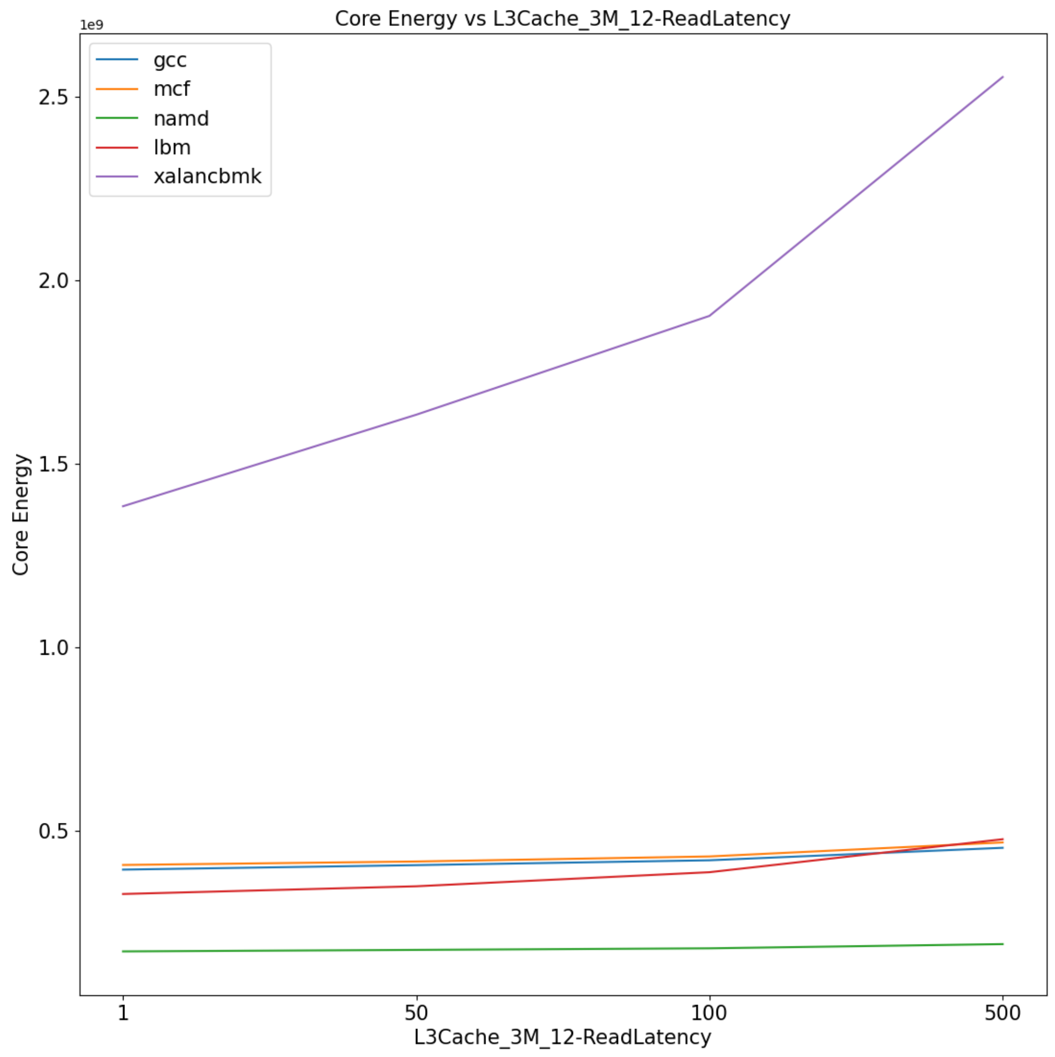

Figure 10: Graph of Core Energy vs L3 Cache 3M 12 for Various Benchmarks

- We observe a general trend between core energy and L3 Cache 3M 12 arising due to the request for access of data at memory addresses. The general trend shows that the time taken increases as memory latency increases. This is identical to our observations for time taken vs main memory latency graph.

- The xalancbmk benchmark is most affected by this cause as we see an increasing spike beyond main memory latency (x) = 100 cycles. The name benchmark stays near constant, suggesting not much presence of memory-based instructions. All L_i caches follow this same trend with core energy. The smaller the size of the cache, the lesser is the amount of corresponding energy consumed.

4.11 Core Power vs Core frequency

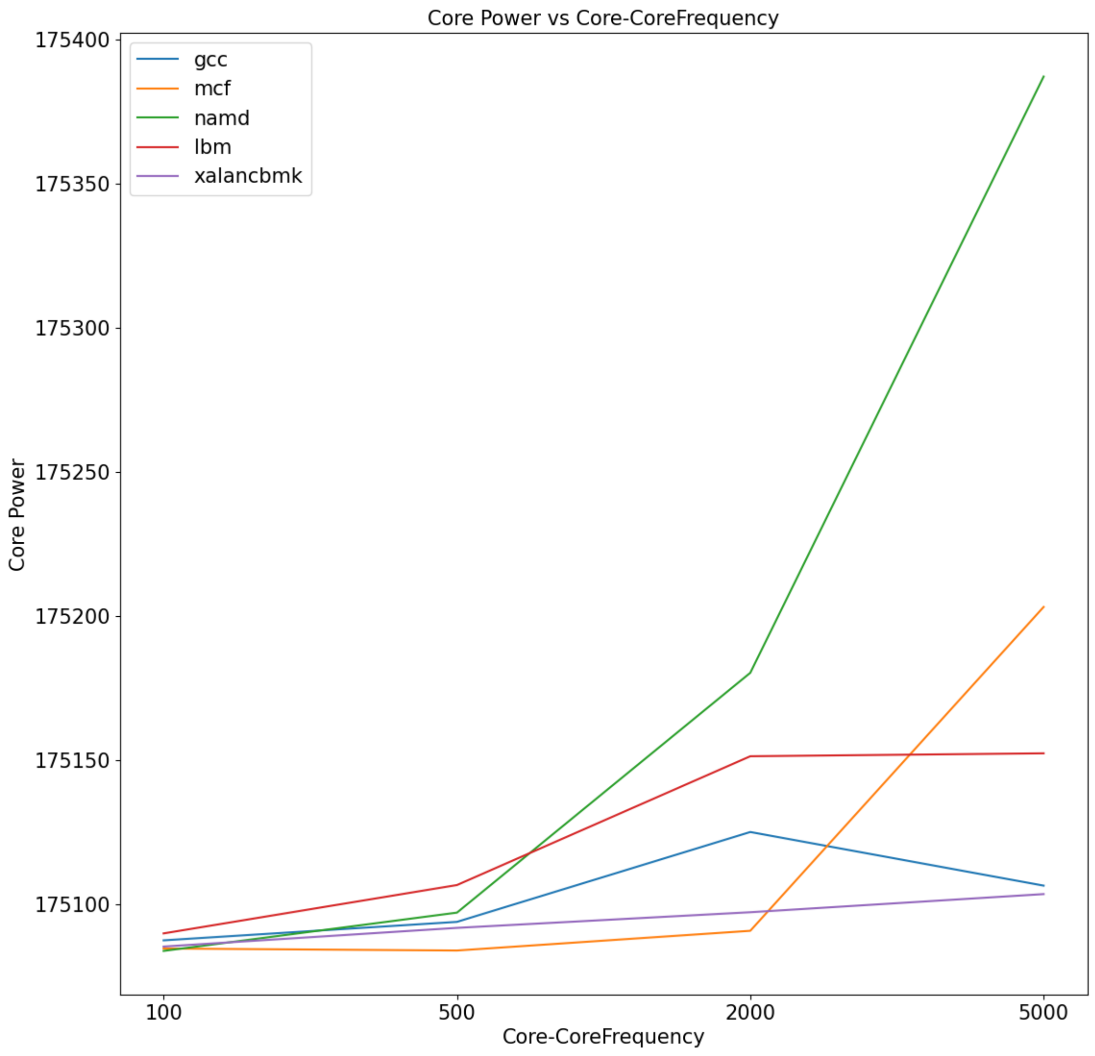

Figure 11: Graph of Core Power vs Core frequency for Various Benchmarks

- We observe a general trend between core power and core frequency. The nature of the graph appears to be polynomial. The core power increases as we vary the core frequency.
- The power consumption of the named benchmark drastically increases as we increase the

core frequency; this can be because more arithmetic & floating point instructions get executed faster, leading to an increase in overall IPC and more power consumption. xalancbmk consumes less power due to it being memory hungry and not CPU hungry.

5 Conclusion

From the data and plots, we observed that most of the performance is getting affected by **Memory Latency** and **Memory Frequency**. This is evident as, among other operations, memory access has the highest latency, and **xalancbmk** benchmark is affected the most. **lbm** benchmark is affected by **Floating ALU Latency**, but is minimal.
The next most important feature affecting the performance is the **Branch Prediction**. Perfect Predictors save a lot of time as well as energy, while no branch predictor causes loss. Even always not taken branches sometimes perform well
Apart from this, **CPU Core Frequency** also seems to improve performance compared to other specifications. Although it might consume more power, for a fixed amount of tasks, it will complete it faster, thereby reducing energy consumed.
Finally, we also modified IW size, which shows a steep increase in energy consumed as we increased its size from 50 - 500 Intstructions