High Performance Computer Architecture: Assignment 1
Group No. 6

May 4, 2023

1 Introduction

Gem5 is a simulation tool that allows us to simulate CPUs with various configurations by using configuration scripts. These configurations can include the number of cores, pipeline complexity, cache size, etc. To conduct the simulations, we write a configuration script and use it to configure an Out-Of-Order CPU that mimics the micro-architectural parameters of a single-core x86 processor as specified in the assignment. Then, we run the benchmark program (qsort5.c) and identify the top 10 configuration combination according to their CPI. By observing the top 10 configuration combinations, we can gain insight into how different micro-architectural parameters affect CPU performance.

2 Implementation

For the overall implementation of the assignment, we modified the gem5 source files, and present three files - ***caches.py***, ***config.py*** and ***constants.py***. These files are used for adding arguments corresponding to LQ, SQ, IQ and ROB entries, inheriting new arguments corresponding to them, and assigning given values (fixed or variable) to the parameters.

Apart from that, we have added three scripts, whose functionalities are mentioned in brief below.

1. **script.sh:** The outputs for the 256 simulations are generated by executing *script.sh*, and are stored in *output/num-N_args-A* sub-directories, where N is the simulation number and A is the argument of the corresponding configuration.

2. **get_top_10.py:** To get the top 10 configurations as per the CPI, run the command *python get_top_10.py*. This displays the top 10 configurations along with their CPI. This also copies the *stats.txt* files of these configurations to *m5out/* directory as per thier rank.

3. **get_plots.py:** To get various plots for the top 10 configurations, run the command *python get_plots.py* after the previous command. This saves all the plots in the *plots/* directory. The limits in each of the plots has been specifically defined to highlight the variations with rank.

3 The Top 10 Configurations

The benchmark program **qsort5.c** was compiled and its execution was simulated with the gem5 simulator system across multiple varying parameters. The ten configurations with the best performance w.r.t **CPI** are listed in the table below.

R	CPI	LQEntries	SQEntries	l1d_size	l1i_size	l2_size	bp_type	ROBEntries	IQEntries
1	0.722338	64	64	64kB	16kB	512kB	TournamentBP	192	64
2	0.722589	32	64	64kB	16kB	512kB	TournamentBP	192	64
3	0.722605	64	64	32kB	16kB	512kB	TournamentBP	192	64
4	0.722605	32	64	32kB	16kB	512kB	TournamentBP	192	64
5	0.722825	64	64	32kB	16kB	256kB	TournamentBP	192	64
6	0.722825	32	64	32kB	16kB	256kB	TournamentBP	192	64
7	0.722984	32	64	64kB	16kB	256kB	TournamentBP	192	64
8	0.723025	64	64	64kB	16kB	256kB	TournamentBP	192	64
9	0.723113	64	32	64kB	16kB	512kB	TournamentBP	192	64
10	0.723113	32	32	64kB	16kB	512kB	TournamentBP	192	64

4 Analysing the combinations of parameters

To test our system on gem5, the qsort5.c benchmark is used, which implements the quicksort algorithm - an in-place sorting algorithm. The algorithm doesn't use additional memory space apart from the input array. Although it is a recursive routine, it updates the array in-place, which results in high spatial and temporal locality that can be leveraged by cache choices. Additionally, the quicksort algorithm involves a lot of store-to-load transfers, so the store-queue can also play an important role in performance.

The qsort5.c benchmark is a multi-threaded implementation of quicksort, where THREADS threads are available to execute recursive calls. If a recursive call receives an array piece with a size below THRESHOLD (which means it can fit in the cache of one of the cores), a separate thread is used for this if one is available. The count of free threads is kept in a variable, and a mutex is used to update it. However, we observe that threading is not used for the given value of N since N is less than the threshold.

The code of the benchmark uses function calls, standard library calls, other library calls, recursion, includes preprocessor, defines input/output threads, break/continue, and arrays. When testing a system using this benchmark, it is important to consider the specifics of the implementation, such as the use of threading and the size of the input data, as they can significantly affect the performance.

- **No. of Load Queue Entries (LQEntries):** It determines the maximum number of load operations waiting for address dependency or resolution of previous store operations. A larger LQEntries size usually improves performance. But store-to-load transfer will not really depend on the size of the load queue so size of store queue will be more important. Therefore, the no. of load queue entries will not have much impact on the performance.
- **No. of Store Queue Entries (SQEntries):** It determines the maximum number of store instructions that can be buffered before execution. A larger SQEntries size allows for more store

instructions to be issued, reducing the likelihood of instruction stalls due to insufficient available store instructions. Larger store queues also facilitate more store-to-load transfers, leading to lower memory overhead and improved performance. Thus, a larger store queue is likely to enhance overall performance.

- **Size of L1 Data Cache (l1d_cache):** It controls the size of the data-specific cache at level 1. A 32 kB data cache has a slightly higher miss rate than the 64 kB sized data cache and as a result, 6 of the top 10 configurations have l1d size parameter as 64 kB. But it is important to note that as a cache gets bigger, access to its data becomes slower as it needs to search among more numbers of entries, thereby increasing its hit time. Since Quick-sort manipulates the items in the same array space because it is in-place with respect to the array elements, the spatial locality is high in this case and therefore a large cache helps to increase the performance.

- **Size of L1 Instruction Cache (l1i_cache):** It controls the size of the instruction-specific cache at level 1. Instruction cache is used to fetch instructions faster. As observed from the top 10 configurations table, l1i_size is found to take 16 kB in all the cases. Since we know that spatial locality is high in quick sort algorithm, thus taking the highest size of l1i_cache i.e. 16 KB enhances performance.

- **Size of L2 Cache (l2_cache):** It controls the size of the unified level 2 cache. The cache at level 2 is not split into instruction and data cache. As the L1 (data) cache is at its optimal size, which is the largest available, increasing the size of the L2 cache will provide diminishing returns in terms of performance improvement. If the L2 cache size is increased beyond a certain point, the additional performance gains will become less significant for each unit of additional cache space, thus, not provide much additional benefit in terms of improving overall system performance.

- **Type of Branch Predictor (bp_type):** The Tournament branch predictor is superior to the BiModal branch predictor out of the two branch predictors. The Tournament predictor combines both Local and Global predictors and selects between them. By making multiple predictions and choosing the appropriate prediction based on the context of the specific branch, the Tournament predictors can improve the performance of the CPU. This is because they add global information, which further enhances their performance. Thus, it is evident that the Tournament predictors are better than the BiModal predictor as they enhance the CPU's overall performance.

- **No. of Reorder Buffer Entries (ROBEntries):** If the ROB (Reorder Buffer) table is larger, it will allow for more instructions to be issued before they are committed. This means that if certain instructions take longer to commit, it will not hinder the issuance of other instructions. The ROB table is used to maintain the original program order of instructions and can handle out-of-order instruction execution by reordering instructions as needed. Therefore, a larger ROB table allows for more out-of-order instruction execution, which can lead to better performance overall.

- **No. of Instruction Queue Entries (IQEntries):** With a larger instruction queue, the processor can fetch and issue more instructions in parallel, which can lead to better performance, allowing for more instructions to be stored and issued quickly. Therefore, the best performance is achieved when the instruction queue has 64 entries.

5 Analysing the outputs

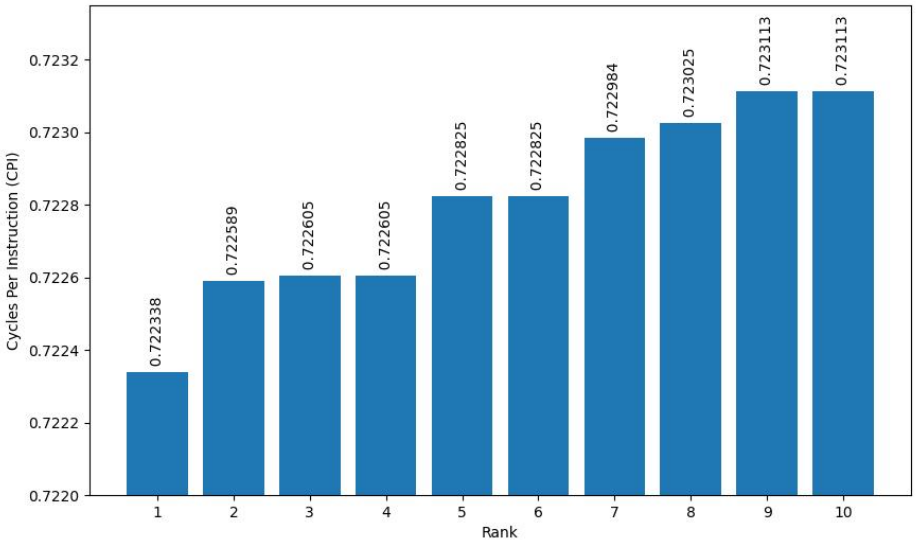
The results obtained from changing the number of Load Queue (LQ) entries showed several changes in the outputs. To explain these statistics and their dependence on input parameters, we can consider the following points:

- A larger LQ size will allow more instructions to be fetched from the instruction queue and executed, which can lead to more mispredictions, especially for branch instructions.
- A larger LQ size will also lead to predict not taken more and that will be incorrect predictions, because most branches will be predicted not taken based on the code history.
- With a larger LQ size, there may be more load events leading to more cache misses.
- Since most branches will be predicted as not taken based on the code history, increasing the LQ size can lead to more predict not taken incorrect predictions.
- More instructions executed and more mispredictions can result in more Read and Write operations for the ROB.

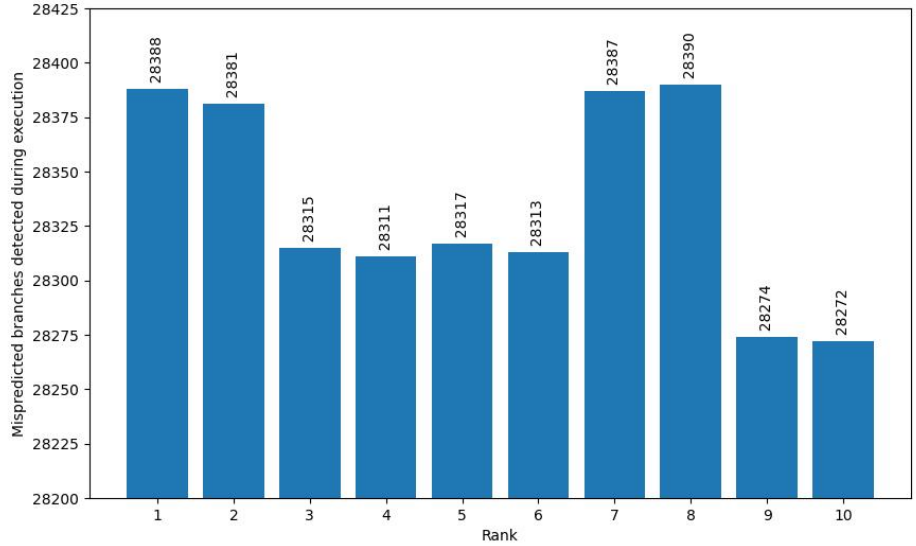
Furthermore, the top 10 configurations have lower miss rates and conflicts compared to other cases, which is supported by the fact that they offer higher hit rates for the same cache size. This indicates that optimizing the cache configuration can significantly improve performance.

6 Plots

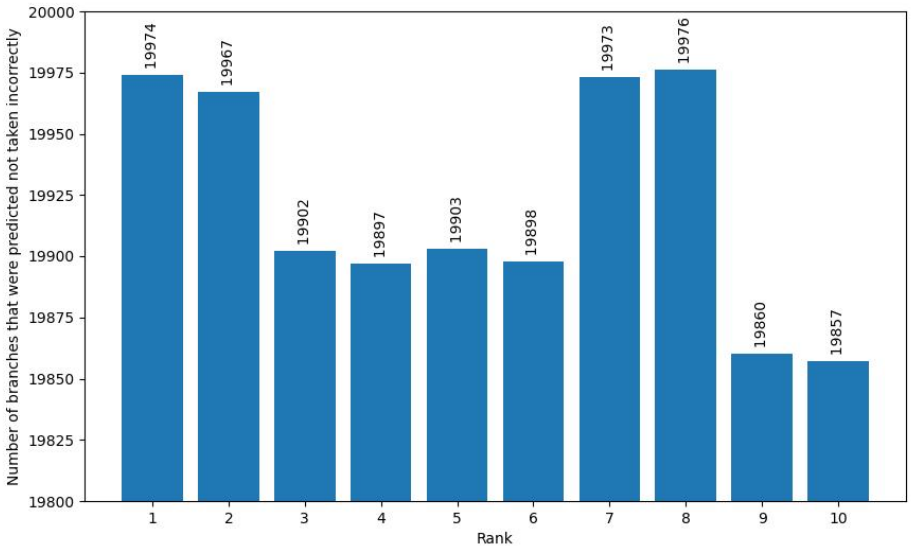
6.1 Cycles Per Instruction (CPI)



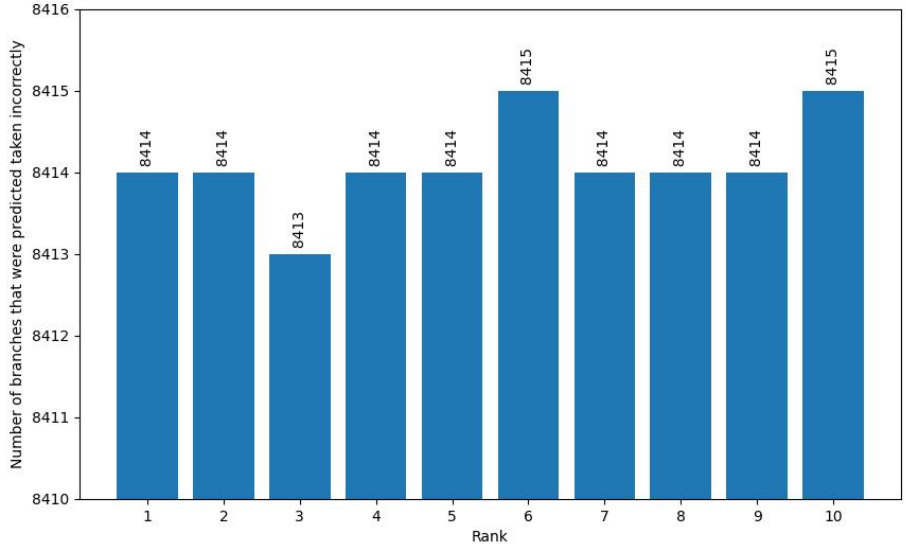
6.2 Mispredicted branches detected during execution



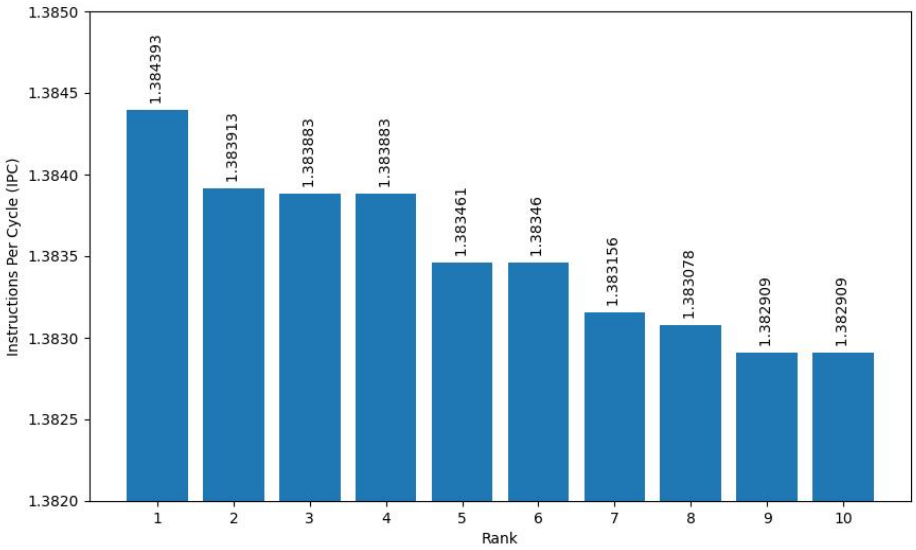
6.3 Number of branches that were predicted not taken incorrectly



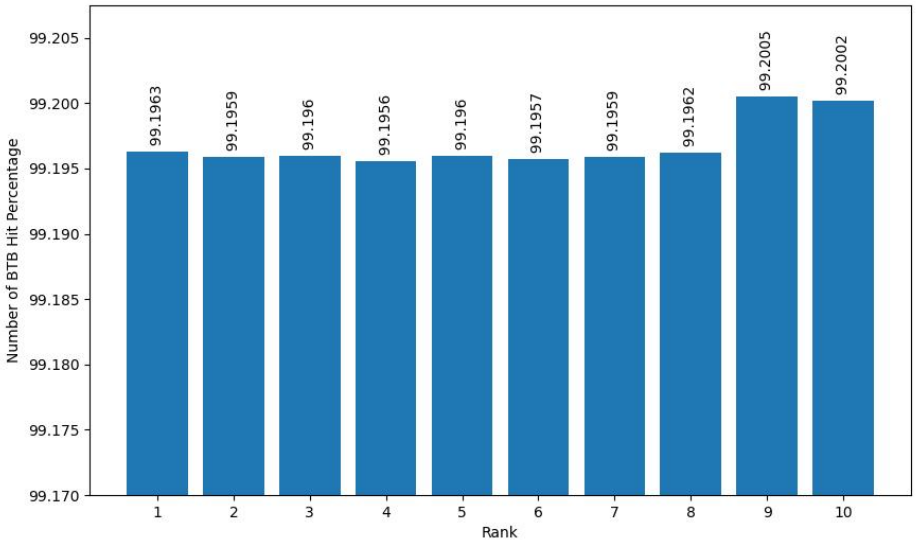
6.4 Number of branches that were predicted taken incorrectly



6.5 Instructions Per Cycle (IPC)



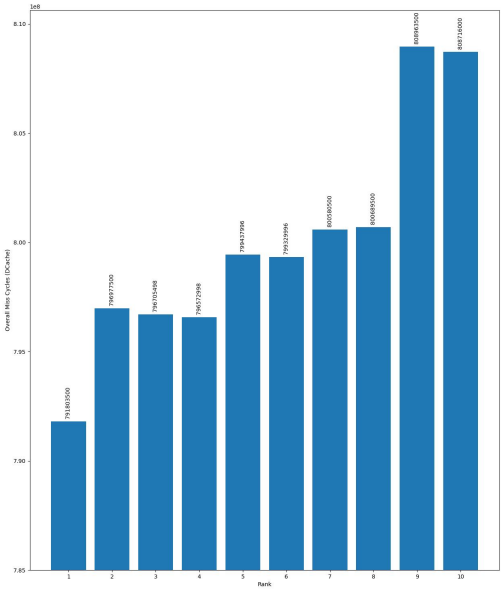
6.6 Number of BTB hit percentage



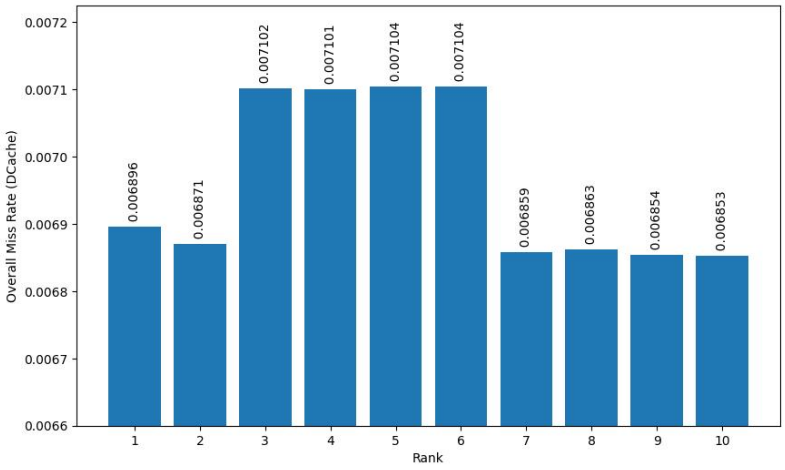
6.7 Number of overall miss cycles, miss rate, average overall miss latency

6.7.1 L1-DCache

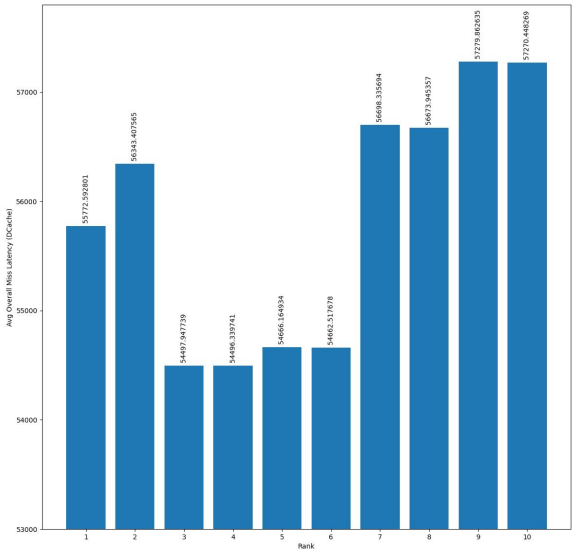
Overall Miss Cycles



Overall Miss Rate

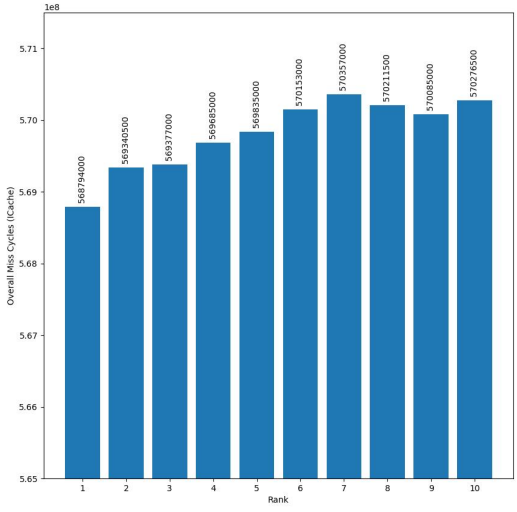


Average Overall Miss Latency

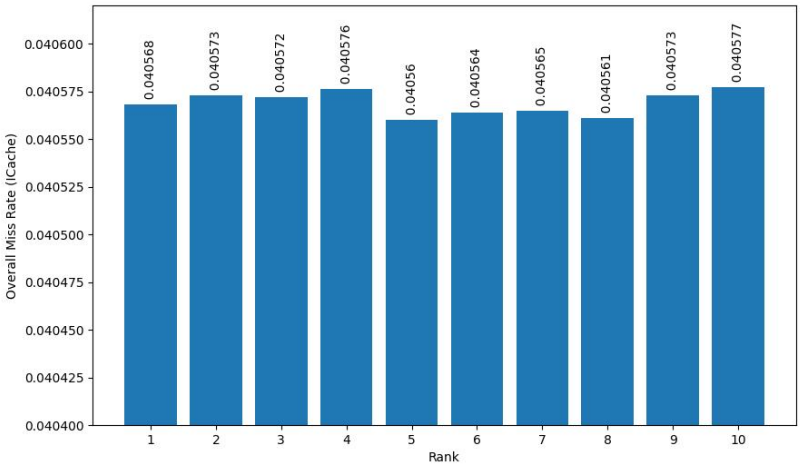


6.7.2 L1-ICache

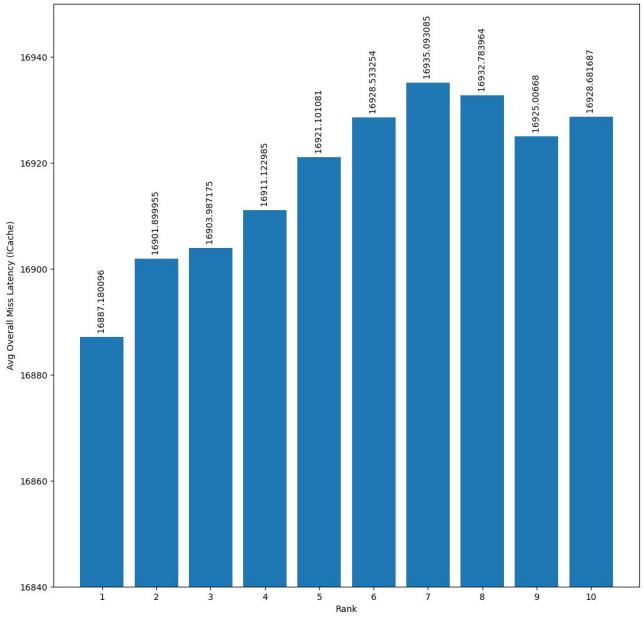
Overall Miss Cycles



Overall Miss Rate

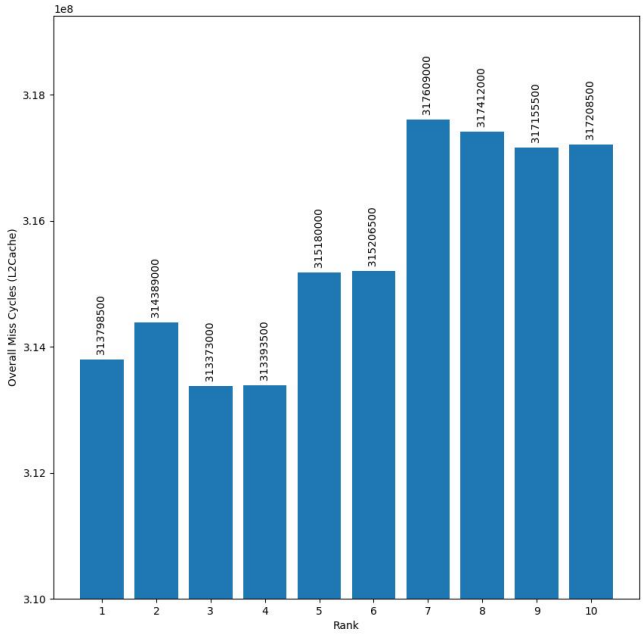


Average Overall Miss Latency

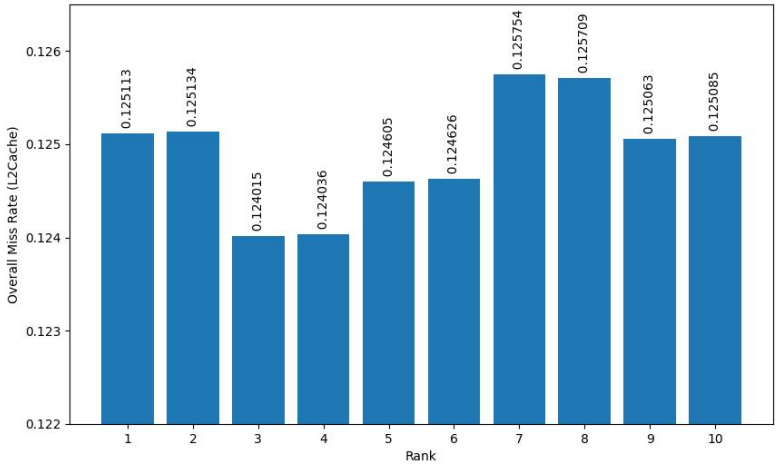


6.7.3 L2Cache

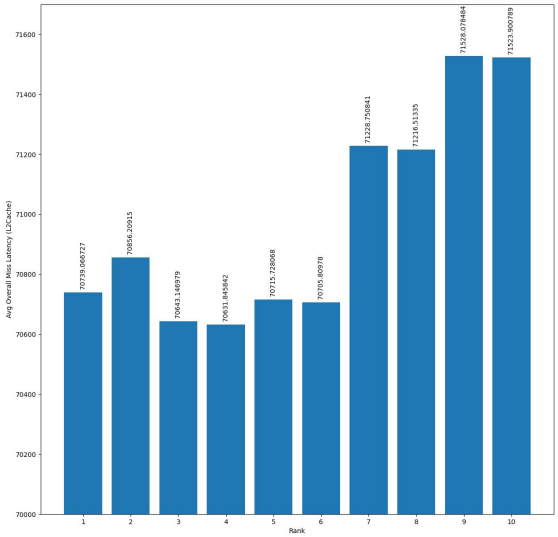
Overall Miss Cycles



Overall Miss Rate

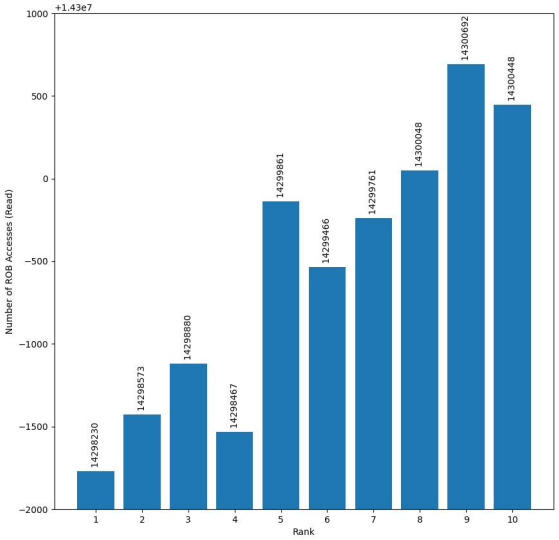


Average Overall Miss Latency

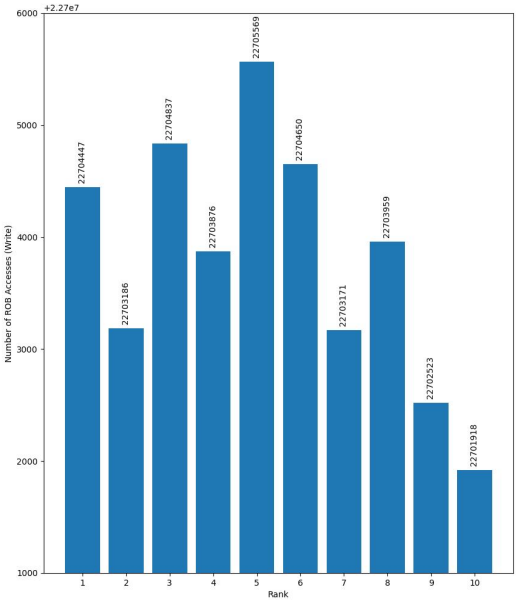


6.8 ROB accesses

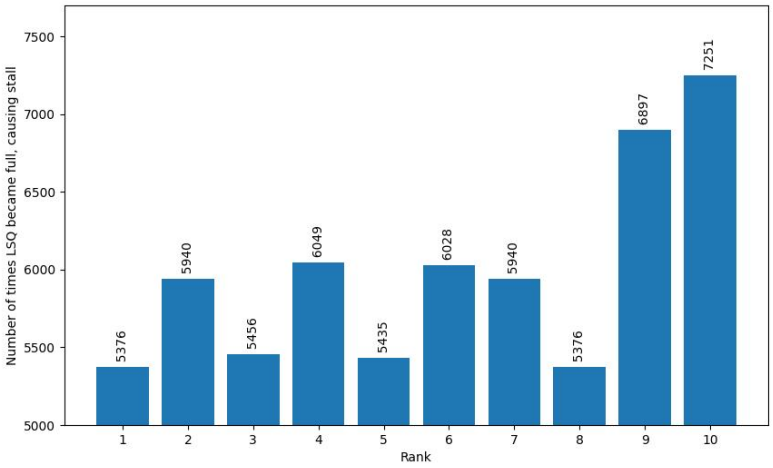
6.8.1 Number of ROB Accesses (Read)



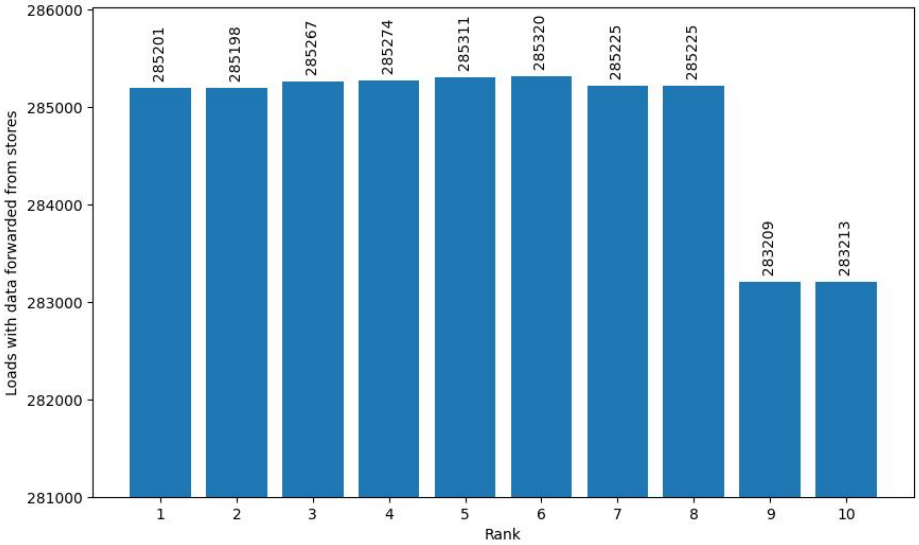
6.8.2 Number of ROB Accesses (Write)



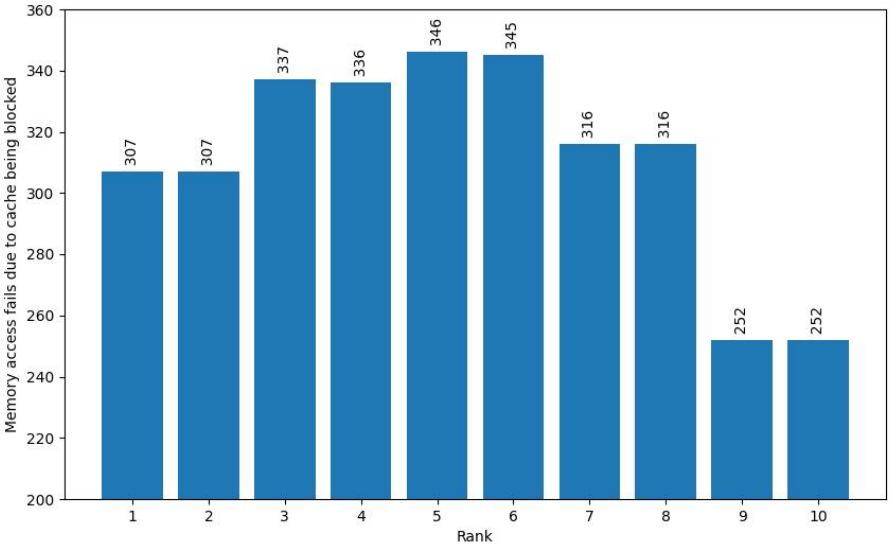
6.9 Number of times the LSQ has become full, causing a stall



6.10 Number of loads that had data forwarded from stores



6.11 Number of times access to memory failed due to the cache being blocked



7 Run Instructions

For the detailed instructions for executing the code and generating top 10 configurations and plots, refer to the README.md file.

8 Group Members and Contribution

1. Haasita Pinnepu (19CS30021)
   - Ran 32 simulations

2. Kamal Swami (19CS30022)
   - Ran 32 simulations

3. Konduru Rakesh Krishna (19CS30024)
   - Ran 32 simulations

4. Kunal Singh (19CS30025)
   - Created the custom caches (caches.py)
   - Ran 32 simulations
   - Created script to get top 10 configurations
   - Analysed the reasons for the performance
   - Prepared the report

5. Lavoori Vamshi (19CS30026)
   - Ran 32 simulations

6. Majji Deepika (19CS30027)
   - Ran 32 simulations

7. Matta Varun (19CS30028)
   - Ran 32 simulations

8. Mayank Kumar (19CS30029)
   - Created the custom config script (config.py) and constants.py
   - Created the bash script (script.sh) to automate the simulations
   - Ran 32 simulations
   - Created script to extract the data and get relevant plots
   - Analysed the reasons for the performance
   - Prepared the report