High Performance Computer Architecture (CS60003)

Assignment 1 Report

Group 4

Contents

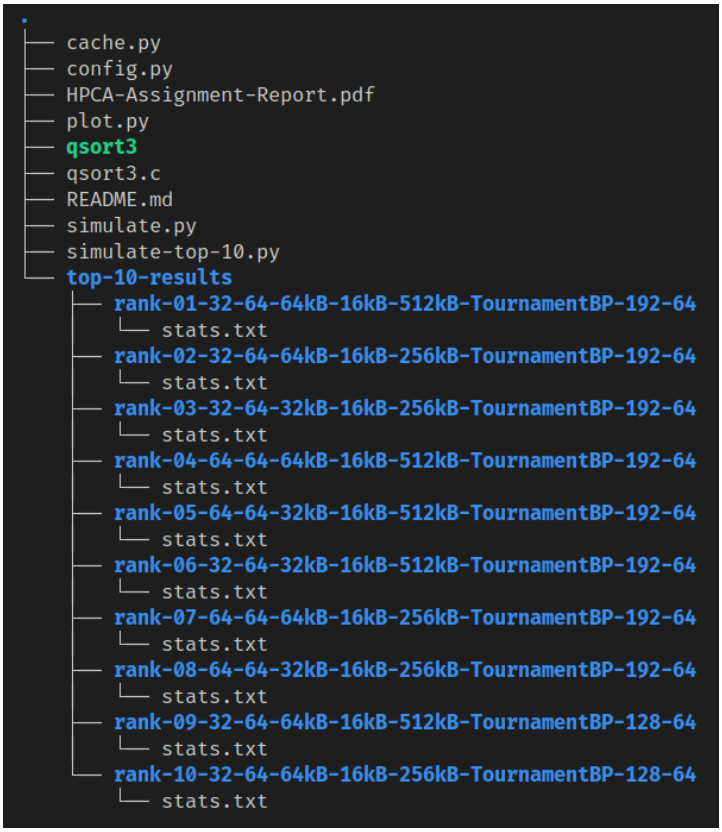
1 Objective

`gem5` is a system simulator that models CPUs at the micro-architecture level and all other associated structures such as caches, memory, interconnect buses, etc. In this assignment, we configure an out-of-order CPU with a list of various micro-architectural parameters on a given benchmark program. Some parameter values are kept fixed, and we vary rest of the parameters to get different configuration combinations. We then analyse the statistics generated using `gem5` using these various configuration combinations to identify the top 10 configurations. We also provide plots for various statistics for these top 10 configurations, and also try to reason out why these parameter configurations work best for our benchmark.

2 Procedure

- The first step is to download and build `gem5`.
- We then create a config file to set up our CPU. The fixed parameters are reflected directly in the config file and the the variable parameters are passed as command-line arguments.
- Through another python script, we automate the process of running the 256 simulations on the benchmark program, by going through all possible combinations of configuration parameters, and the statistics for all these simulations are stored separately.
- Using another python script, we extract the necessary information from the statistics files to obtain the top 10 configurations w.r.t CPI. We also extract all the other statistics asked for and plot them.

3 Submission Structure



4 Details of Files Submitted

- `cache.py` : Contains the definitions of the `L1Cache`, `L1ICache`, `L1DCache` and `L2Cache` classes.
- `config.py` : Configuration file containing the parameters and their values (for fixed parameters) for the simulations. It takes the values of the variable parameters as command-line arguments.
- `simulate.py` : Script to simulate all the 256 configuration combinations on the given benchmark.
- `simulate-top-10.py` : Script to simulate the top 10 configurations or any one of the top 10 configurations.
- `qsort3.c` : The benchmark program to be executed.
- `qsort3` : The binary of the benchmark program.
- `plot.py` : Script to extract the top 10 configurations w.r.t. CPI and plot various statistics.
- `top-10-results` : Directory containing the `stats.txt` files of the top 10 configurations.
- `HPCA-Assignment-Report.pdf` : The report for the assignment.

5 Execution Instructions

First, install and build `gem5`.
Navigate to the `gem5` directory in your system. All the further commands assume that you are inside the `gem5` directory.
Then, copy all files in the submission directory to `configs/assignment` inside the `gem5` directory

```
mkdir -p configs/assignment
cp -r Group_4_HPCA_Assignment_1/* configs/assignment/
```

To execute all the 256 configurations, execute the `simulate.py` file. It takes the path to the `gem5` directory as a command line argument (`-g`). This may take nearly an hour. This creates the statistics outputs in the `configs/assignment/results` directory. The name of each folder inside it is of the format `LQEntries-SQEntries-l1d_size-l1i_size-l2_size-bp_type-numROBEntries-numIQEntries`.

```
python configs/assignment/simulate.py -g <path-to-gem5>
```

To execute the top 10 configurations, execute the `simulate-top-10.py` file. It takes the path to the `gem5` directory as a command line argument (`-g`). To run a specific file among the top 10 (according to the rank), you can do so using the `--rank` command-line argument. So, for example, to run the 1st configuration, you can do the following. The results for this will be stored in `configs/assignment/top-10-results`.

```
python configs/assignment/simulate-top-10.py -g <path-to-gem5> --rank=1
```

If you do not specify the `--rank` argument, it will run all the top 10 configurations.

```
python configs/assignment/simulate-top-10.py -g <path-to-gem5>
```

To plot the various statistics asked for, execute the `plot.py` file. It takes the path to the `gem5` directory as a command line argument (`-g`), and the path to the directory containing the results as a command line argument (`-r`). This will create the plots in the `configs/assignment/plots` directory.

```
python configs/assignment/plot.py -g <path-to-gem5> -r <path-to-results>
```

6 Analysis and Discussion

6.1 Top 10 Configurations w.r.t. CPI

The benchmark program **qsort3.c** was compiled and its execution was simulated with the gem5 simulator system across multiple parameters (some fixed and some varying). The variable parameters and the possible values they could take up are:

- **LQEntries** (No. of load queue entries): 32, 64
- **SQEntries** (No. of store queue entries): 32, 64
- **l1d_size** (Size of L1 data cache): 32kB, 64kB
- **l1i_size** (Size of L1 instruction cache): 8kB, 16kB
- **l2_size** (Size of L2 cache): 256kB, 512kB
- **bp_type** (Type of branch predictor): TournamentBP, BiModeBP
- **numROBEntries** (No. of RoB entries): 128, 192
- **numIQEntries** (No. of instruction queue entries): 16, 64

Using all possible combinations, we ran **256** simulations. After extracting the CPI values for each configuration, the top 10 observed configurations and their corresponding CPI values are listed in Table 6.1.

Table 1: Top 10 configurations with the best performance w.r.t CPI

Rank	LQ Entries	SQ Entries	l1d_size	l1i_size	l2_size	bp_type	ROB Entries	IQ Entries	CPI
1	32	64	64kB	16kB	512kB	Tournament	192	64	0.812704
2	32	64	64kB	16kB	256kB	Tournament	192	64	0.812816
3	32	64	32kB	16kB	256kB	Tournament	192	64	0.812877
4	64	64	64kB	16kB	512kB	Tournament	192	64	0.813507
5	64	64	32kB	16kB	512kB	Tournament	192	64	0.813558
6	32	64	32kB	16kB	512kB	Tournament	192	64	0.813558
7	64	64	64kB	16kB	256kB	Tournament	192	64	0.81362
8	64	64	32kB	16kB	256kB	Tournament	192	64	0.81367
9	32	64	64kB	16kB	512kB	Tournament	128	64	0.814604
10	32	64	64kB	16kB	256kB	Tournament	128	64	0.814737

6.2 Analysis

- **No. of load queue entries (LQEntries)**: Both the 32-entry LQ and the 64-entry LQ perform equally. A higher number of load queue entries implies that we can have more load instructions waiting for data, or for address resolution, or for getting data from previous stores. For all cases, the number of stalls due to the LSQ being full is much less than the number of total loads (0.25%). Also, in our quicksort algorithm, all the load requests have high temporal and spatial locality. This, combined with the low cache miss rate leads to almost all load requests being resolved immediately, without much delay. Hence, both the 32-entry and 64-entry load queues perform equally.

- **No. of store queue entries (SQEntries)**: A store queue with more entries means that more store instructions can be issued and be on-the-fly before being committed. So, in general, a larger store queue implies that even if some store instructions take time to execute, they will not prevent other store instructions from getting issued. Also, with a larger store queue, there are more possibilities of store-to-load forwarding, as we have more store instructions we can search among for an address to match (nearly 20% loads get data forwarded from stores).

- **Size of L1 data cache (l1d_size)**: A 64kB L1 data cache tends to perform better than a 32kB cache. However, among the top 10 configurations, we can see a cache size of 32kB appearing too. A 64kB data cache has a lower miss rate than a 32kB cache (evident from the plots too). Hence, we see the 64kB data cache size dominating the top 10 CPI configurations. But interestingly, the miss latency is lower for the 32kB cache because the time to search through the cache entries increases as the cache size increases. Also, the difference in miss rate is not too much between the two cache sizes (0.0068 for 64kB and 0.0070 for 32kB), so multiple 32kB entries also appear in the top 10. So, we can conclude that the L1 data cache size is not a very big factor because the array size is nearly 8kB, and the access patterns have good spatial and temporal locality, hence the performance of both the 32kB data cache and the 64kB data cache are comparable.

- **Size of L1 instruction cache (l1i_size)**: From the table of top 10 configurations, it is visible that the 16kB instruction cache performs the best. This is also evident from the miss rate of the 16kB cache (0.06) compared to the miss rate of the 8kB cache (0.22). Although the miss latency for the 16kB cache is slightly higher than the miss latency for the 8kB cache, the high difference in the miss rate tilts the balance in the 16kB cache's favour. This can be explained by the presence of a few long branches in the code (e.g., the *if* on line 30 and the corresponding *else if* on line 38 in `qsort3.c`).

- **Size of L2 cache (l2_size)**: Both the 256kB and 512kB L2 cache perform almost equally in terms of performance (CPI). This can be attributed to the fact that the miss rate of the L1 data cache is extremely low (around 0.0068 or 0.68%), so the number of accesses to the L2 cache is anyways very less. The L1 data cache is almost always able to satisfy all requests because of its sufficient size (the array is only 8kB), and the L2 cache has diminishing returns with increasing size. So, the size of the L2 cache (256kB or 512kB) is not a big factor (as it is anyways much larger than 8kB).

- **Type of branch predictor (bp_type)**: In our simulation, we observe that nearly 79% of the branches are conditional branches. Moreover, since the array is already sorted, the *if* statements at line numbers 16, 18, 30 and 44 are always not taken, and the branches at line 38 and 47 are always taken. So, in general because of these easy to decipher patterns, the branch prediction accuracy is nearly 98%. But, the branch on line 38 (`else if (left_ptr == right_ptr)`) is dependent on the outcome of the branch on line 30 (`if (left_ptr < right_ptr)`) as they are complementary in our example. Similarly, the branch on line 47 is dependent and complementary to the branch on line 44 (`if (right_ptr - lo > hi - left_ptr)`). For such cases, a tournament predictor works best as it can incorporate both the local and global history while making predictions. Hence, we see that the tournament predictor seems to be a better choice by looking at the top 10 configurations.

- **No. of RoB entries (numROBEntries)**: The value of 192 seems to be a better choice but we also see the value 128 at the 9th and the 10th position. The entries in the Reorder Buffer (RoB) correspond to the instructions that are presently in the instruction window. So, more RoB entries should lead to better out-of-order execution, since even if some instructions take time to complete, other later non-dependent instructions can go ahead with their execution. So, when there are many such instructions that can take time to execute, having a larger RoB is better. But, in our benchmark `qsort3.c`, the operations that can correspond to such long latency instructions are possibly a L1 and L2 cache miss. Such cases happen only 0.8% of the time, hence we see that we

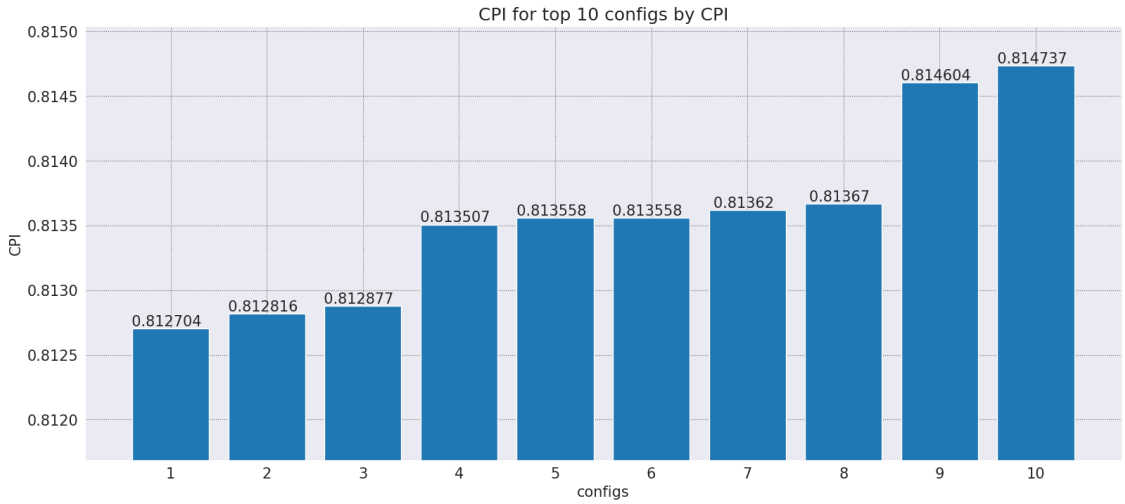
also have two entries corresponding to a 128-entry RoB at the last two positions among the top 10 configurations. So, with such cases when the percentage of low latency operations is low, after a certain point, increasing the number of entries in the RoB does not lead to much improvement in performance.

- **No. of instruction queue entries (numIQEntries)**: The results show that a 64-entry instruction queue is a better choice. The size of the instruction queue determines the number of instructions that have been decoded and are ready to be issued. If we look at any of the `config.ini` files, we will see that the `fetchWidth` and `issueWidth` for `gem5` is set as 8. So, a 64-entry IQ is able to fetch and issue more instructions, compared to its 32-entry counterpart, by utilising the `fetchWidth` and `issueWidth` of `gem5`, as the 32-entry instruction queue will get filled up pretty soon. Hence, a larger instruction queue results in a better CPI.

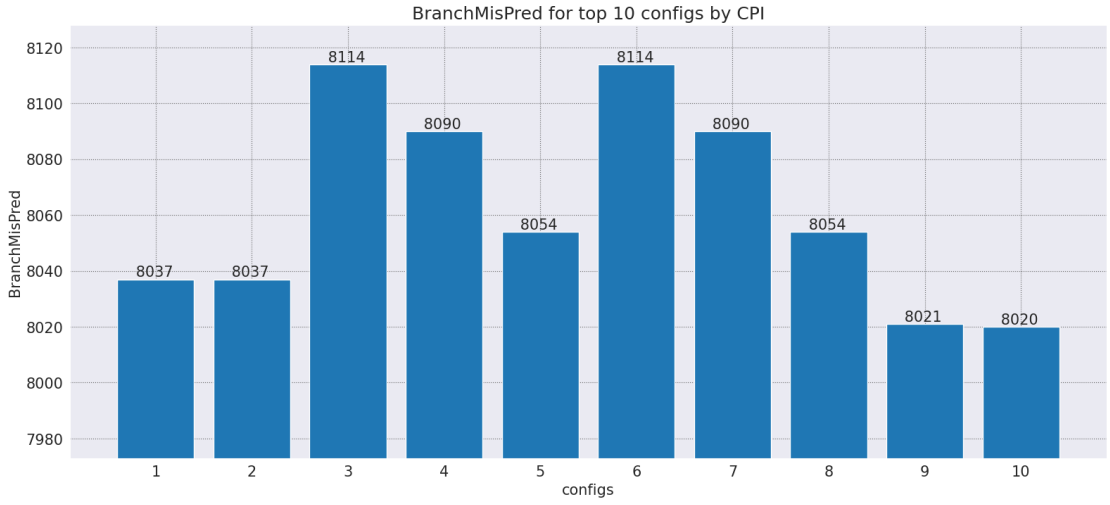
6.3 Plots for Various Statistics

We now plot various statistics for the above mentioned top 10 configurations.

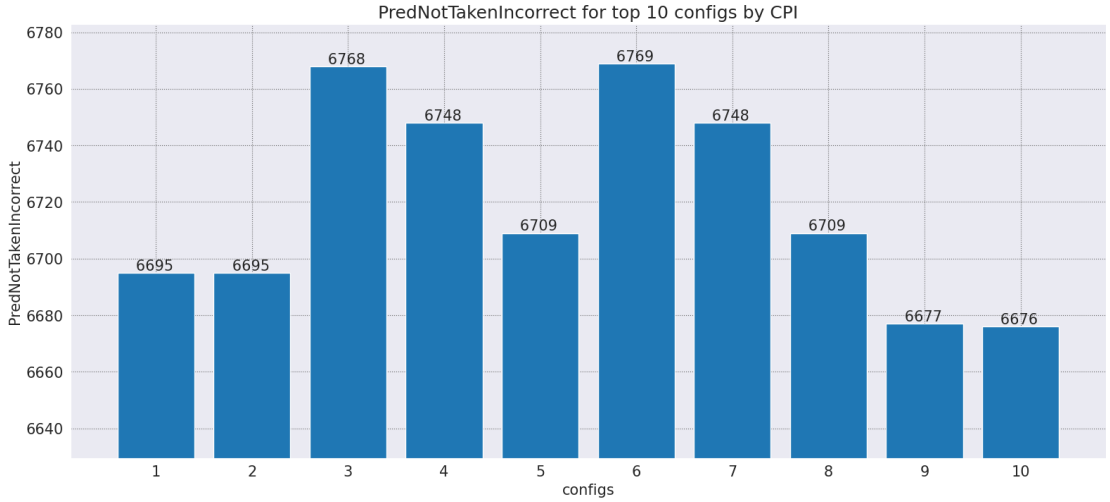
- **Cycles Per Instruction (CPI)**



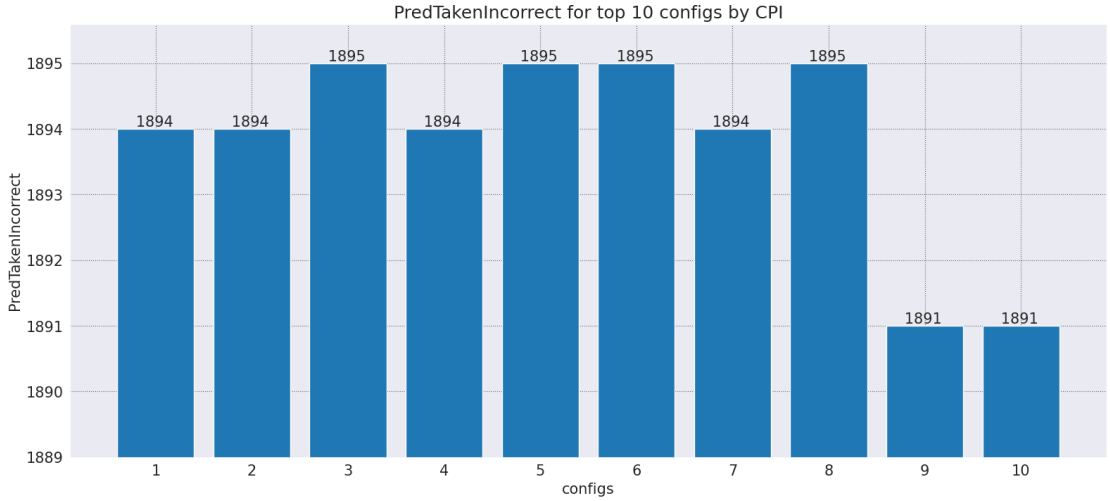
- **Mispredicted branches detected during execution**



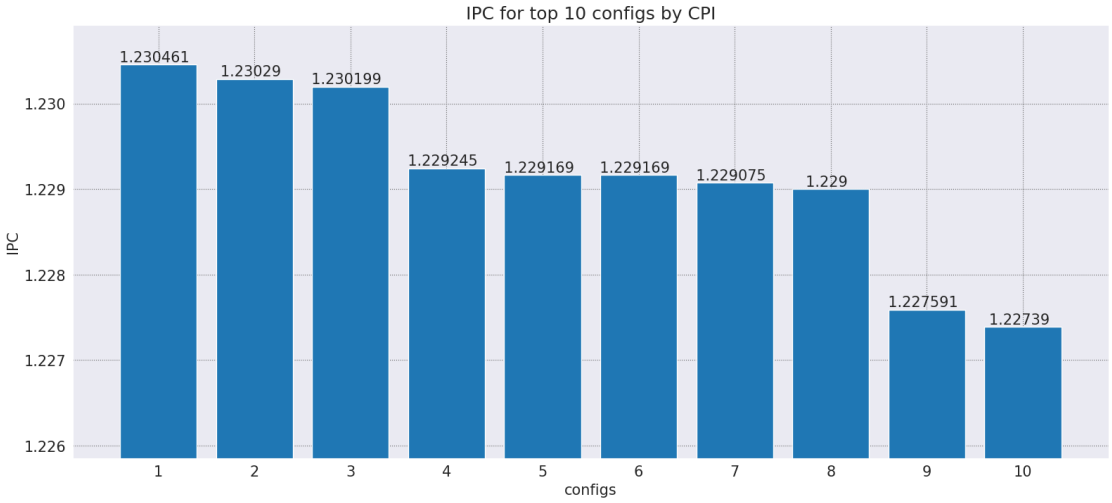
- **Number of branches that were predicted not taken incorrectly**



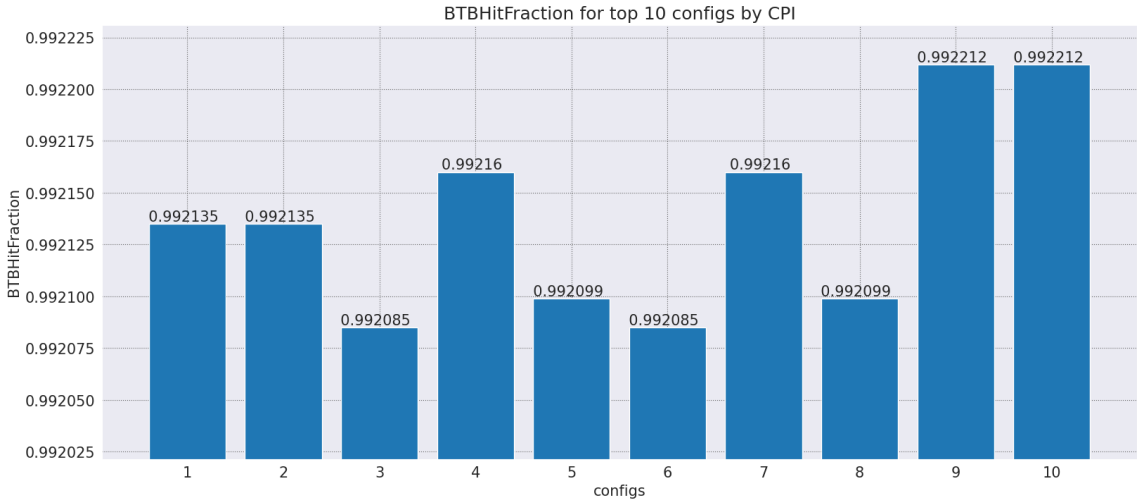
- **Number of branches that were predicted taken incorrectly**



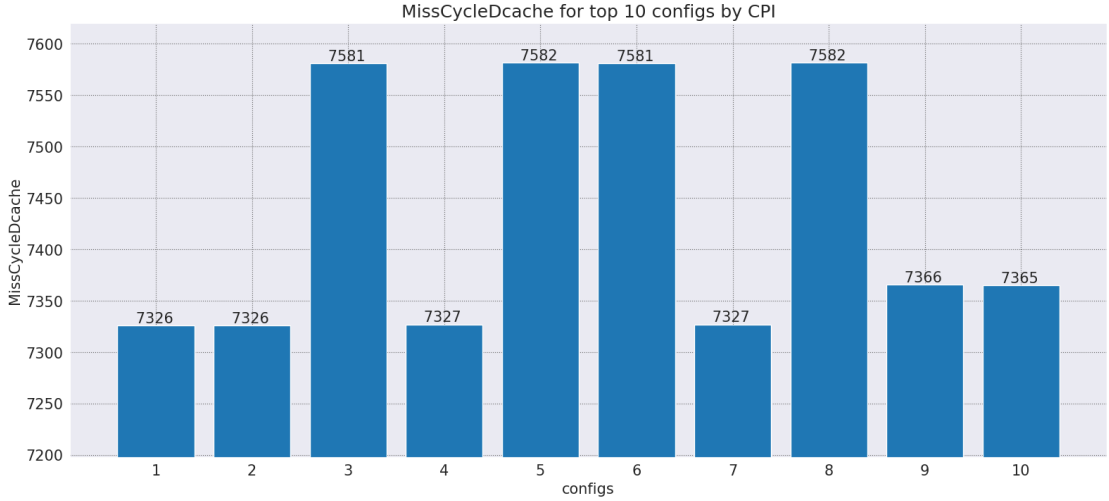
- **Instructions Per Cycle (IPC)**



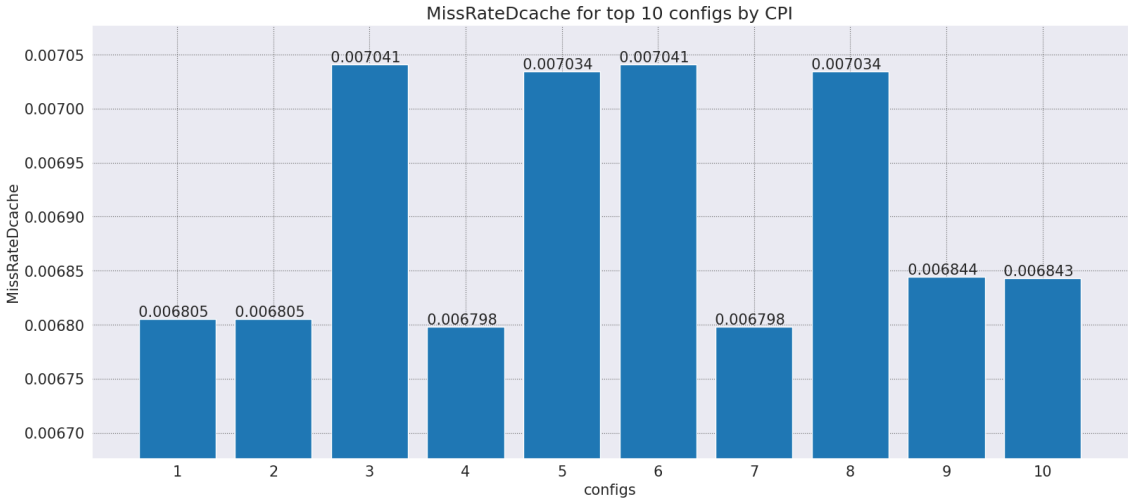
- **Number of BTB hit percentage (the plot shows the fraction)**



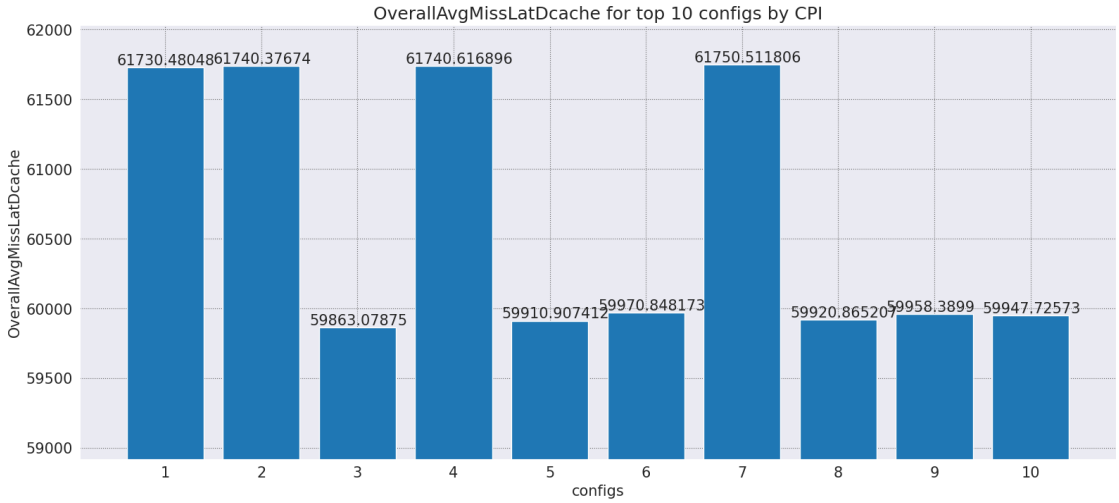
- **Number of overall miss cycles (for L1 data cache)**



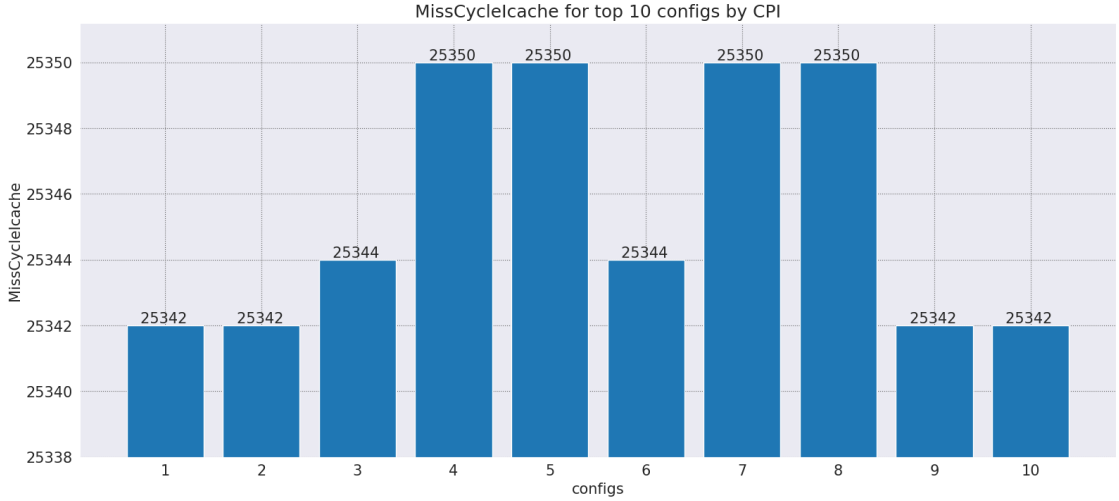
- **Miss rate (for L1 data cache)**



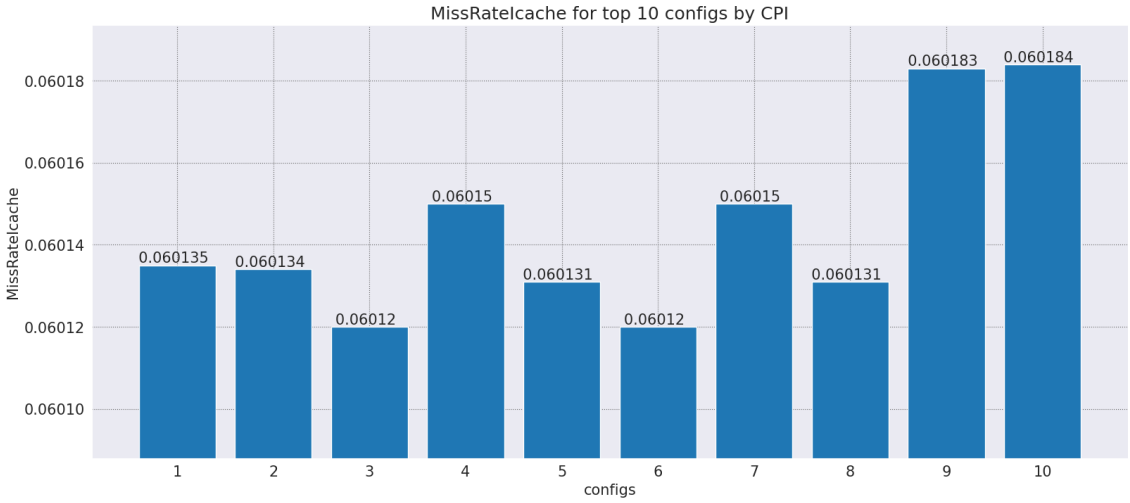
- **Average overall miss latency (for L1 data cache)**



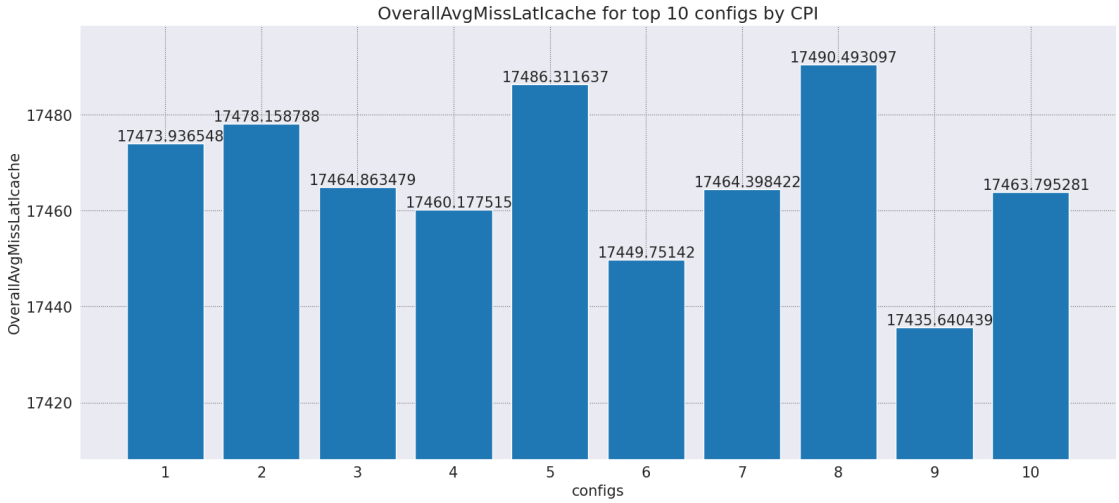
- **Number of overall miss cycles (for L1 instruction cache)**



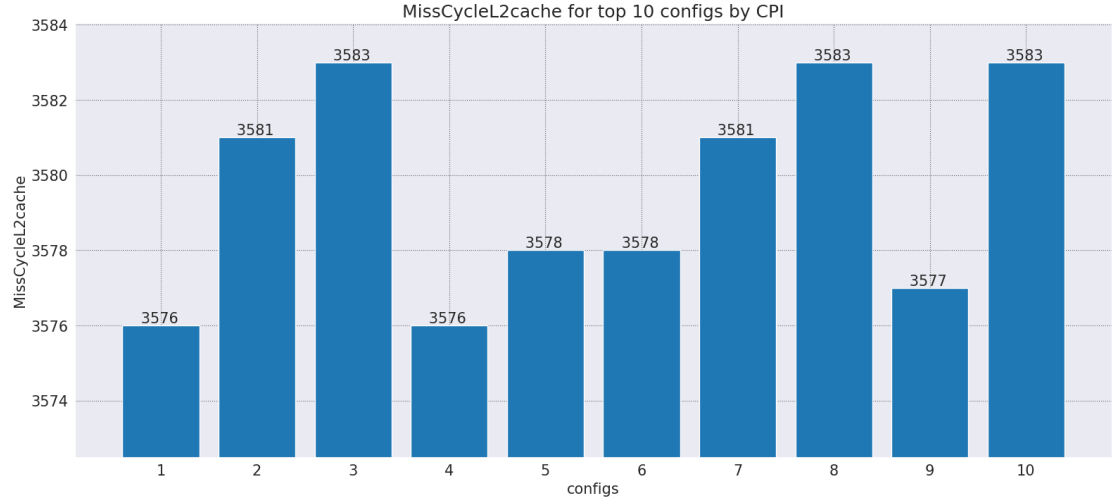
- **Miss rate (for L1 instruction cache)**



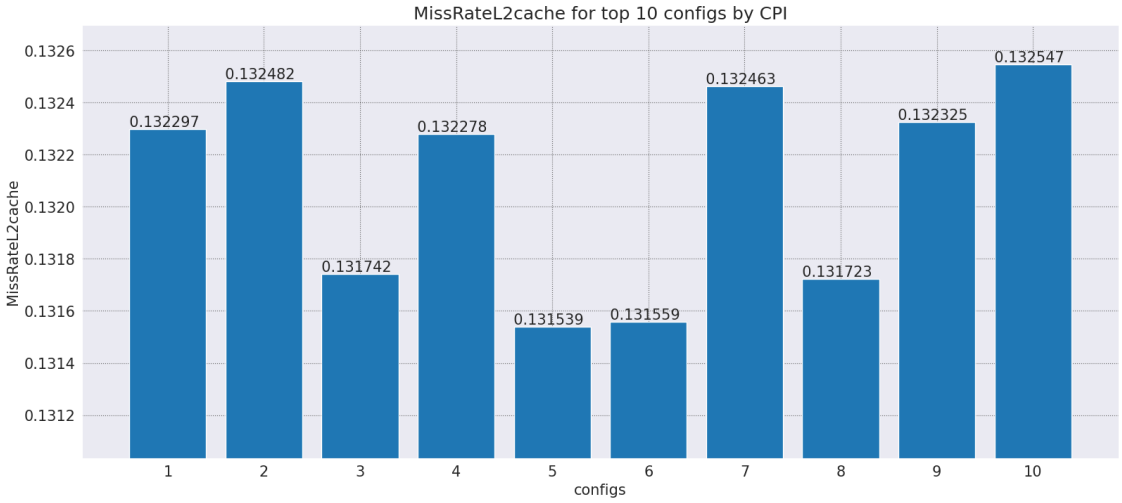
- **Average overall miss latency (for L1 instruction cache)**



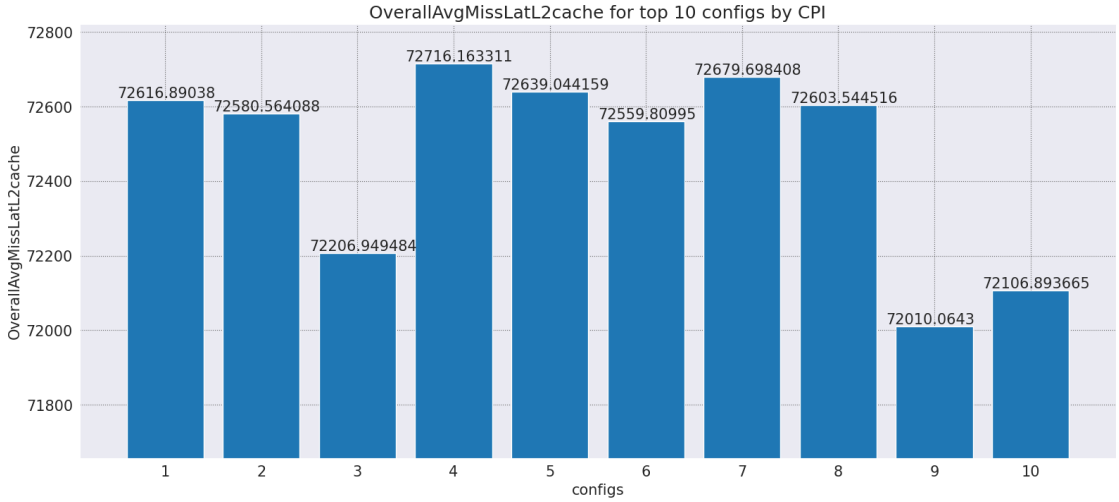
- **Number of overall miss cycles (for L2 cache)**



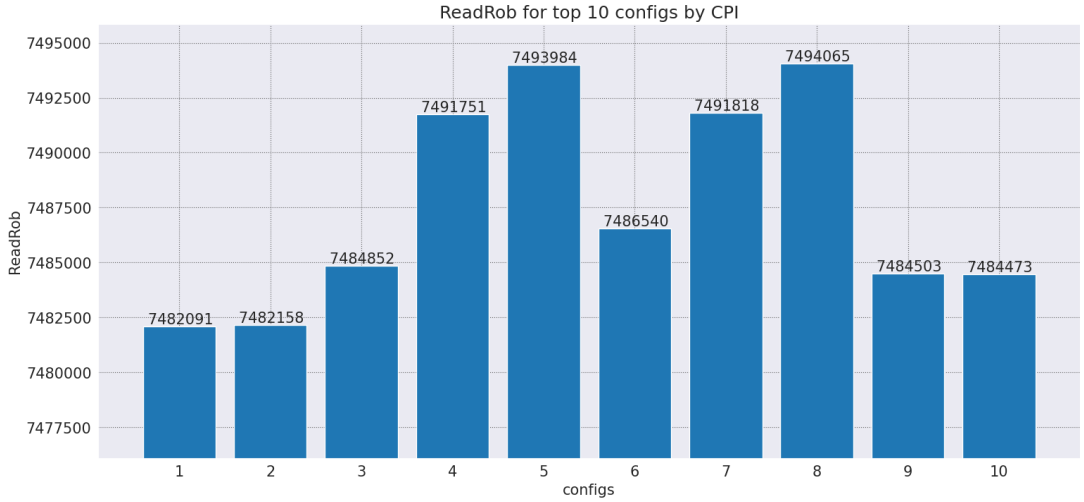
- **Miss rate (for L2 cache)**



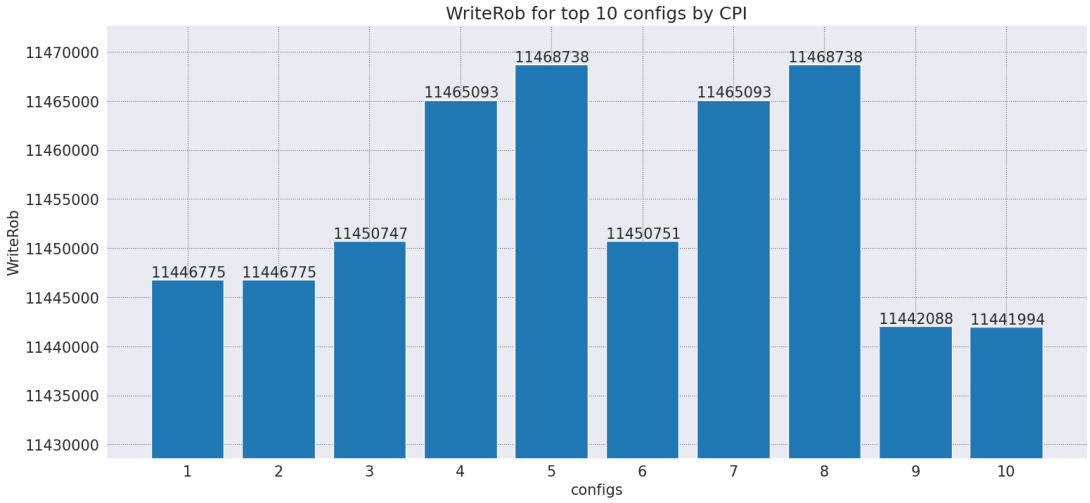
- **Average overall miss latency (for L2 cache)**



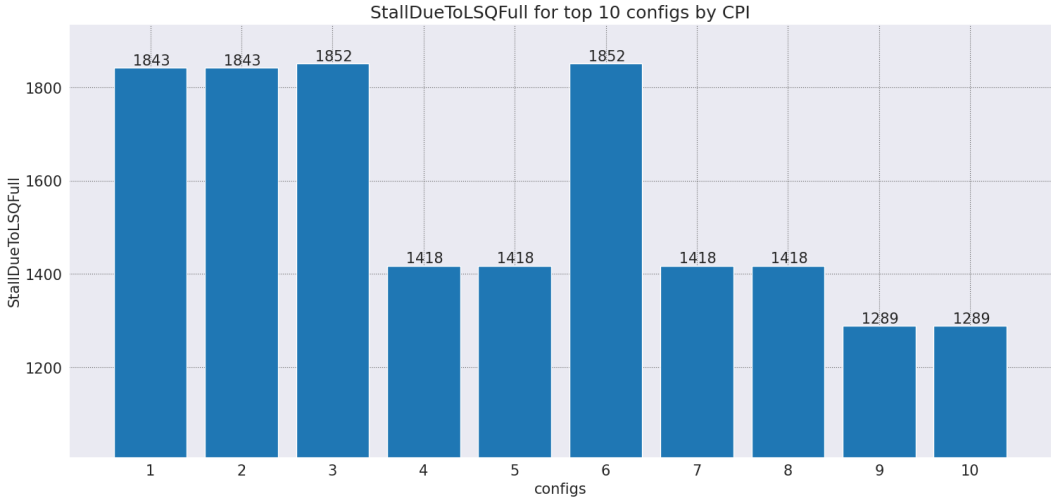
- **Number of RoB read accesses**



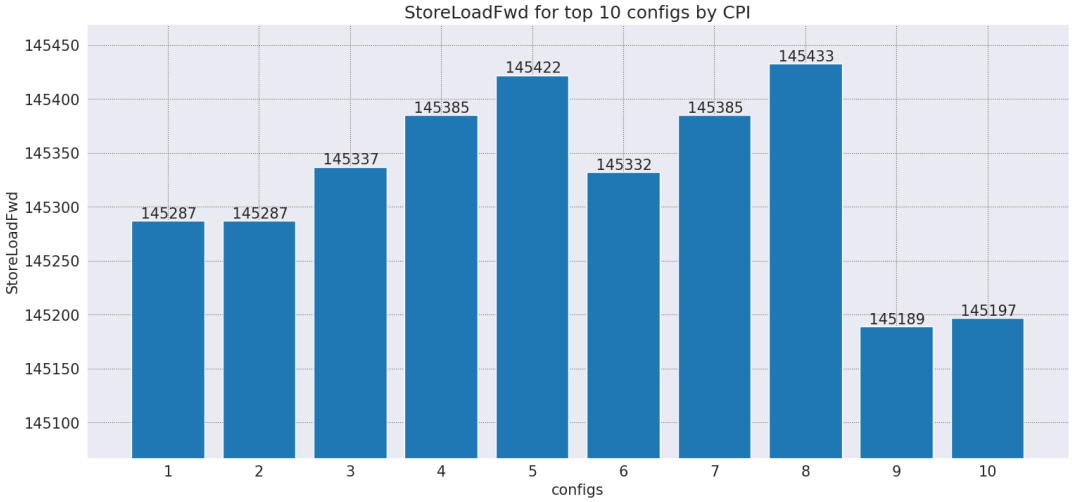
- **Number of RoB write accesses**



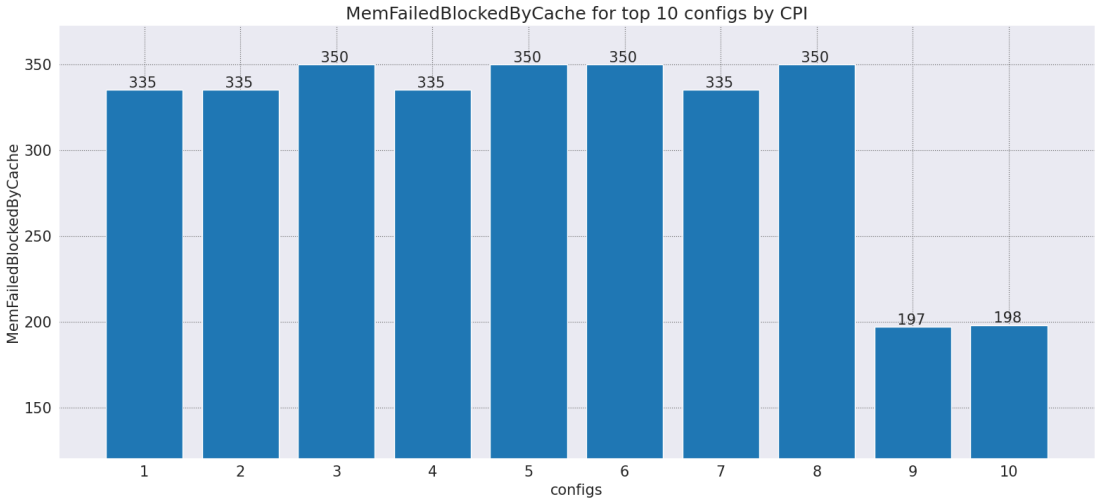
- **Number of times the LSQ has become full, causing a stall**



- **Number of loads that had data forwarded from stores**



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



7 Group Members and Their Contributions

Sl. No.	Roll No.	Name	Contributions
1	19CS30005	Aryan Agarwal	Writing the file to simulate the top 10 configurations (`simulate-top-10.py`)
2	19CS30006	Aryan Mehta	Writing the script to run all simulations (`simulate.py`)
3	19CS30007	Aryan Singh	Writing the script for plotting statistics (`plot.py`), and preparing the report
4	19CS30008	Ashutosh Kumar Singh	Writing the custom config script (`config.py`) and the script to run all simulations (`simulate.py`), running the simulations, and preparing the report
5	19CS30009	Ashwamegh Rathore	Analyzing the results and preparing the report
6	19CS30010	Athithya Prakash R	Analyzing stats.txt to figure out parameters and preparing the report.
7	19CS30011	Bokade Tushar Kishor	Analyzing statistics and preparing the README.md file
8	19CS30012	Bondalapati Venkata Abhiram	Analyzing the results and preparing the report