CS60003
HIGH PERFORMANCE IN COMPUTER ARCHITECTURE

SPRING 2023

Assignment 1

Group 7

19CS30030 - Monal Prasad
19CS30031 - Nisarg Upadhyaya
19CS30032 - P Anurag Reddy
19CS30033 - Parth Jindal
19CS30034 - Parth Tusham
19CS30035 - Piriya Sai Swapnika
19CS30036 - Pranav Rajput
19CS30037 - Rajas Bhatt

Submitted to

Prof. Debdeep Mukhopadhyay
Department of Computer Science and Engineering
IIT Kharagpur

Goals

1. Use Gem5 to customize the parameters of a CPU which supports out-of-order execution.
2. Run a Benchmark program on the CPU with 256 different parameter combinations, varying one component of the CPU each time and recording statistics of the benchmark run.
3. Identify the top 10 (out of 256) configurations based on CPI values for the Benchmark program.
4. Analyze the results and provide appropriate reasoning to justify the existence of these results.
5. Generate graphs to display how various other performance measures vary across the top 10 configurations. These performance measures include the branch mispredictions, incorrect NOT TAKEN predictions, incorrect TAKEN predictions, cache misses (for L1 and L2 caches) and the number of reads and writes in the reorder buffer (ROB).

Brief Introduction & Methodology

Gem-5 is a Linux-based software tool that can simulate different types of computer processors, such as MIPS and ARM. It also allows users to create and simulate custom hardware components.

In this experiment, no new hardware components were created. However, adjustment was done to the parameters of 8 existing hardware components. We kept some parameters fixed while varying others according to the assignment's instructions. We automated 256 different configurations (2 ^ 8) using a python script and selected the top 10 configurations with the best performance, based on Cycles Per Instruction (CPI). We then re-simulated these configurations and used a parser script to extract other performance metrics from the automatically generated stats file.

We plotted the results to observe how the performance varies with the different configurations. Finally, we analyzed the results and made relevant observations and justified their occurrence.

Analysis and Plots

Part A: Listing the top 10 configurations based on CPI values

The top 10 configurations have been listed in the table below. The explanation of the table columns is as follows:

`Conf #` The index of the configuration [0-255]
`LQ Entries` The number of entries in the load queue
`SQ Entries` The number of entries in the store queue
`L1d Size` The size (in kB) of the L1 Data cache
`L1i Size` The size (in kB) of the L1 Instruction cache
`L2 Size` The size (in kB) of the L2 cache
`BP Type` The type of the Branch Predictor used (Tournament or BiMode)
`ROB Entries` The number of entries in the reorder buffer
`NumIQ Entries` The number of entries in the instruction queue

Rank	Conf #	LQ Entries	SQ Entries	L1d Size (kB)	L1i Size (kB)	L2 Size (kB)	BP Type	ROB Entries	NumIQ Entries	CPI
1	59	32	32	64	16	512	TournamentBP	192	64	0.586712
2	243	64	64	64	16	256	TournamentBP	192	64	0.586713
3	51	32	32	64	16	256	TournamentBP	192	64	0.586713
4	219	64	64	32	16	512	TournamentBP	192	64	0.586713
5	251	64	64	64	16	512	TournamentBP	192	64	0.586714
6	211	64	64	32	16	256	TournamentBP	192	64	0.586714
7	27	32	32	32	16	512	TournamentBP	192	64	0.586715
8	19	32	32	32	16	256	TournamentBP	192	64	0.586717
9	241	64	64	64	16	256	TournamentBP	128	64	0.586751
10	57	32	32	64	16	512	TournamentBP	128	64	0.586751

Part B: Code and Configuration Analysis

First of all it is important to note that for the first 8 ranks there is negligible difference in the CPIs obtained. The following parameters are the ones which remain constant for the first 8 configurations:

1. BP Type - We observed that TournamentBP outperforms the BiModeBP with a 5% reduction in mispredictions. On assembly code analysis it was found that most of the branches are not taken branches. Then we compared the incorrect predictions when branch is taken and not taken respectively for configurations having TournamentBP and BiModeBP. It is observed that TournamentBP has lesser mispredictions for not taken branches and BiModeBP has lesser mispredictions for taken branches. Because the majority of the branches in the code are not taken, it is only fair that the TournamentBP performs better. Moreover, in the recursive section the major branches are whether something should be printed or not (which will always be taken or not depending on the flag set), for loop conditional check (mostly not taken) and checking the queens are placed properly or not (which will have some sort of global pattern). Overall, the BiModeBP which is meant to reduce destructive aliasing for branches which share the same global history pattern, is not of much use to us considering that most of the branches are not taken and do not clash with each other.

The next three points are related to the overall structure of our code - which is recursive. For the most part of the execution there is a single block (thus having good locality) of approx 90 instructions which keeps running in a recursive manner.

2. ROB Entries - A larger reorder buffer means more instructions can be issued.
3. IQ Entries - A larger instruction queue means quick access to instructions which can be issued.
4. L1 Instruction Cache - A larger cache ensures less cycles wasted to get the instructions from memory and all instructions can be fetched immediately from the cache.

Thus overall the above 3 combined ensure that instructions keep getting issued quickly and that latency is brought down.

From the generated assembly we could identify approximately 60 load and 24 store instructions. Of this in the recursive section we could identify 4 pairs which had a load immediately following a store which could benefit from store to load forwarding. Most of the read after write dependencies in the recursive sections are a load followed by a conditional branch testing the recently loaded value. With a good branch predictor these are not an issue.

Moreover, while the total loads are more than 60, the loads and stores in the recursive section are around 12 and 7 each. The recursive section does get called again and again but there are also several other instructions and thus sufficient cycles in between so that the load and store queues keep getting cleared. Also the memory footprint of the overall code is very less, most of the memory is allocated at the start as global variables and is accessed from there. It is a few thousand bytes of data for which a 32kB D-cache also suffices. Thus, the cycles for load and store are further reduced due to this ensuring the queues indeed get cleared very fast. Because of this, the D-cache and LSQ entries do not play such a significant role in determining the performance of the CPU.

Part C: Plots

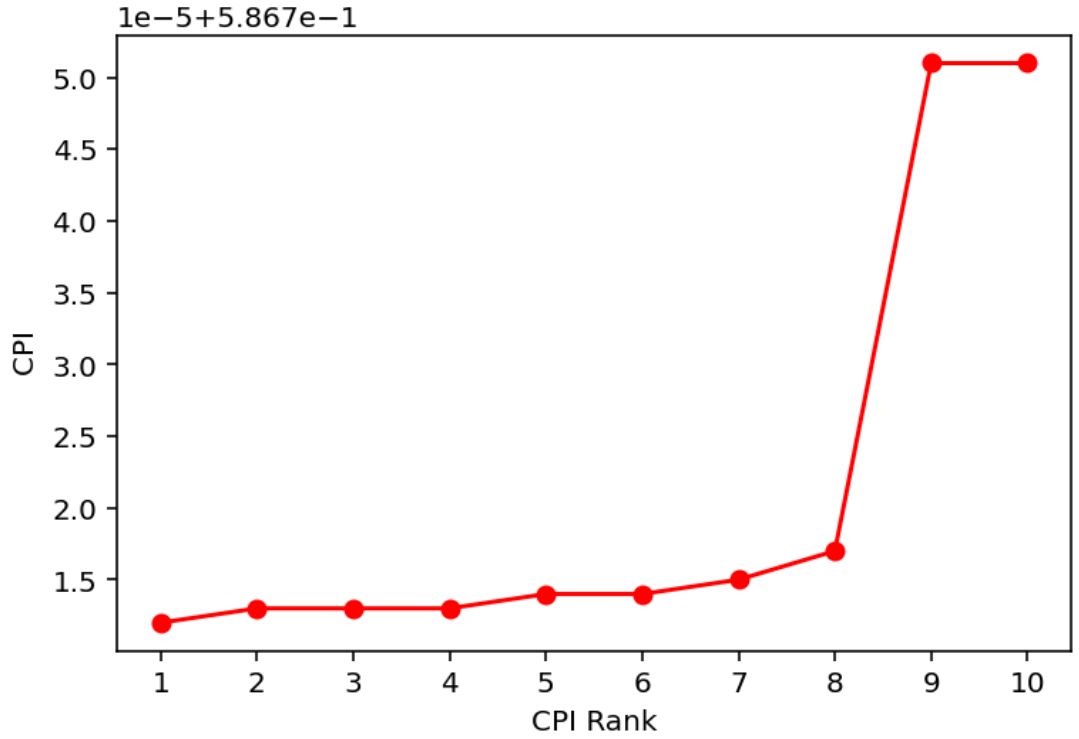

Fig 1: CPI for Top 10 configurations

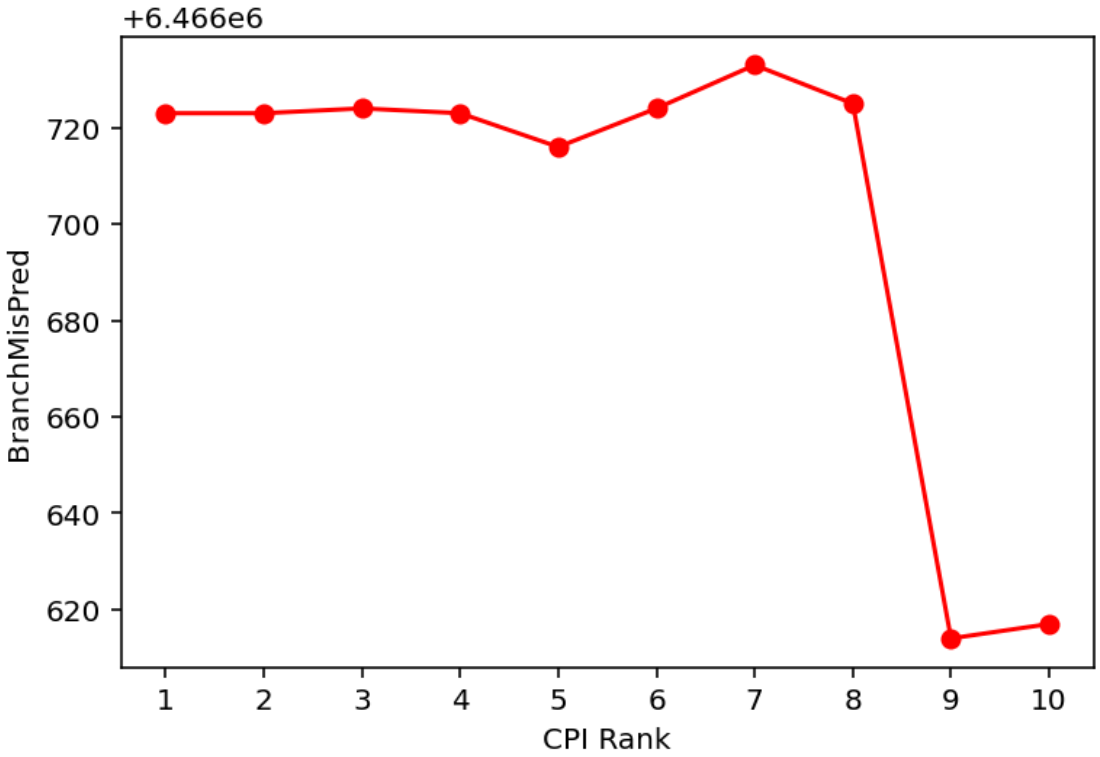

Fig 2: Mispredicted branches during execution

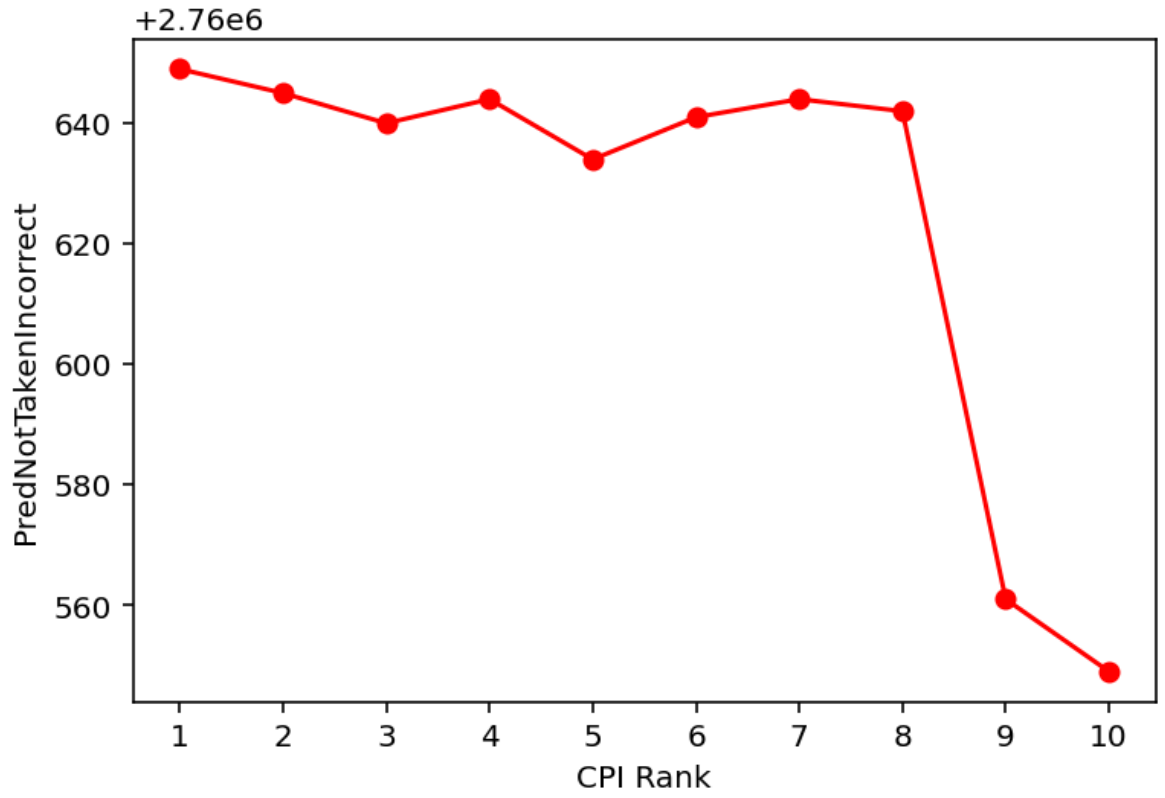

Fig 3: Number of branches that were predicted not taken incorrectly

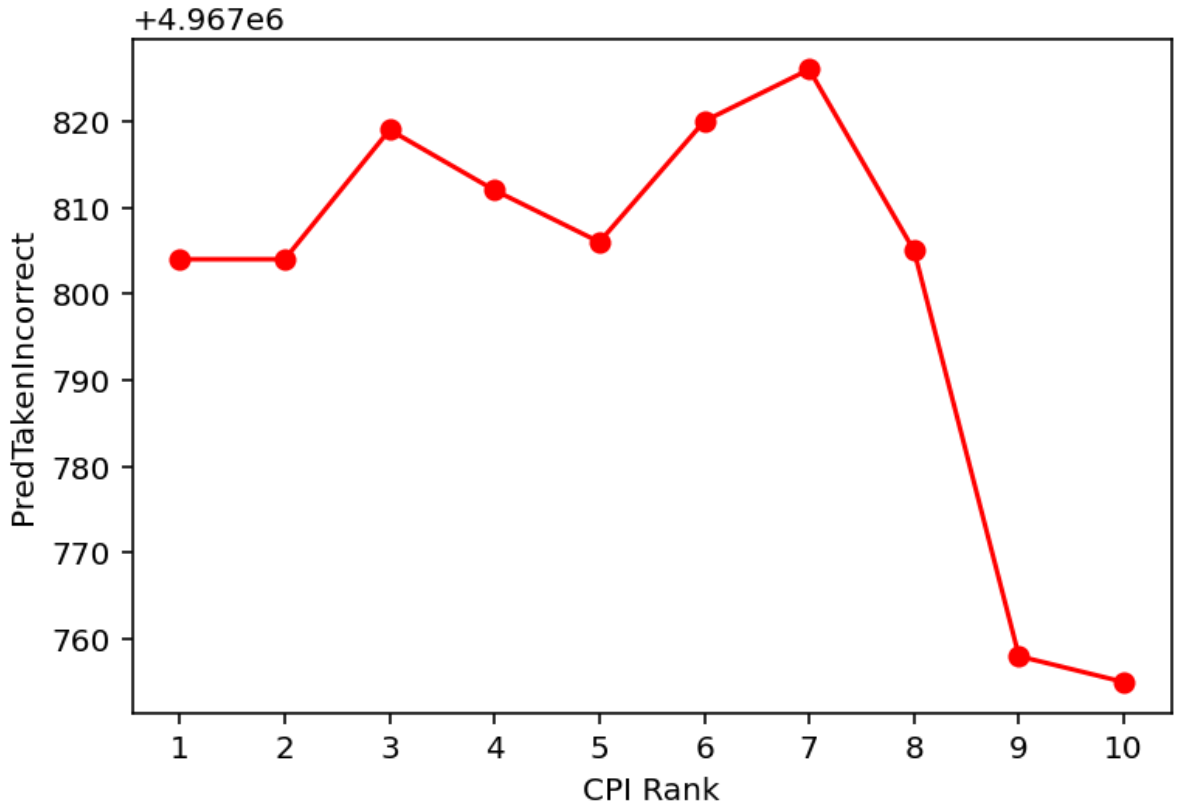

Fig 4: Number of branches that were predicted taken incorrectly

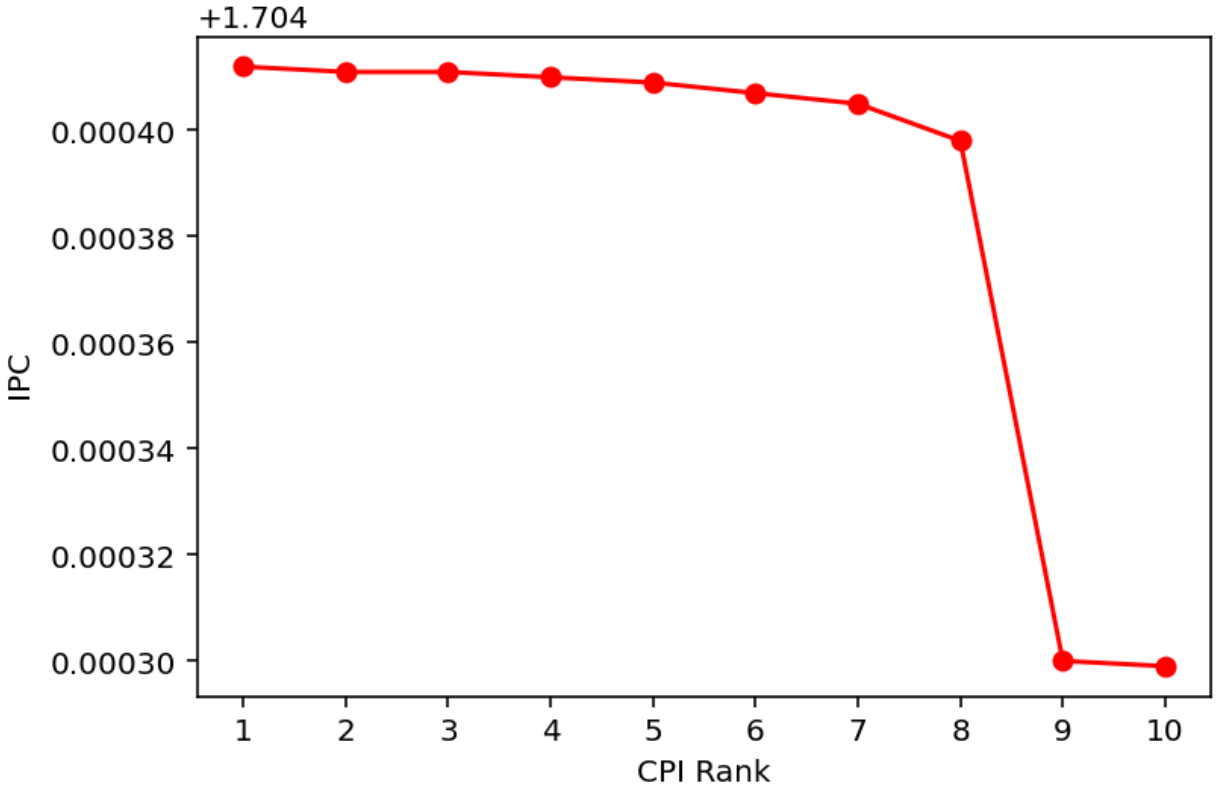

Fig 5: IPC for Top 10 configurations

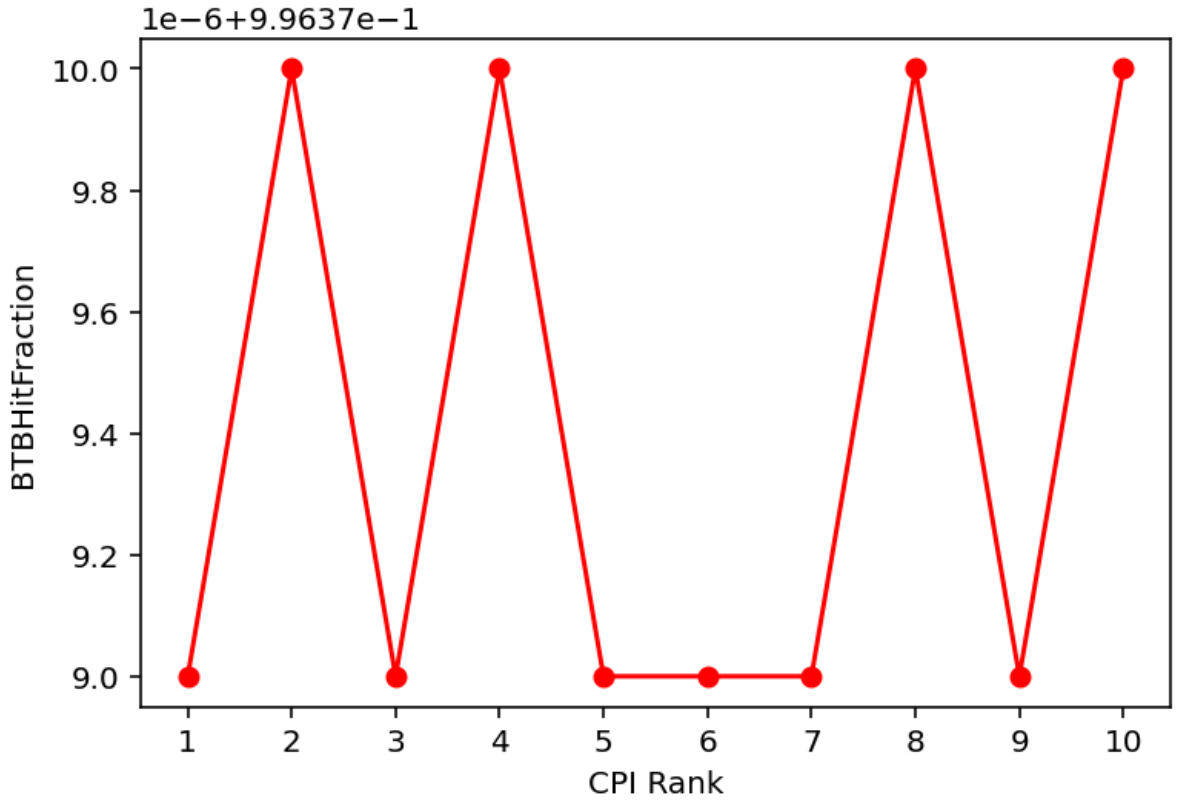

Fig 6: Number of BTB hit percentage for top 10 configurations

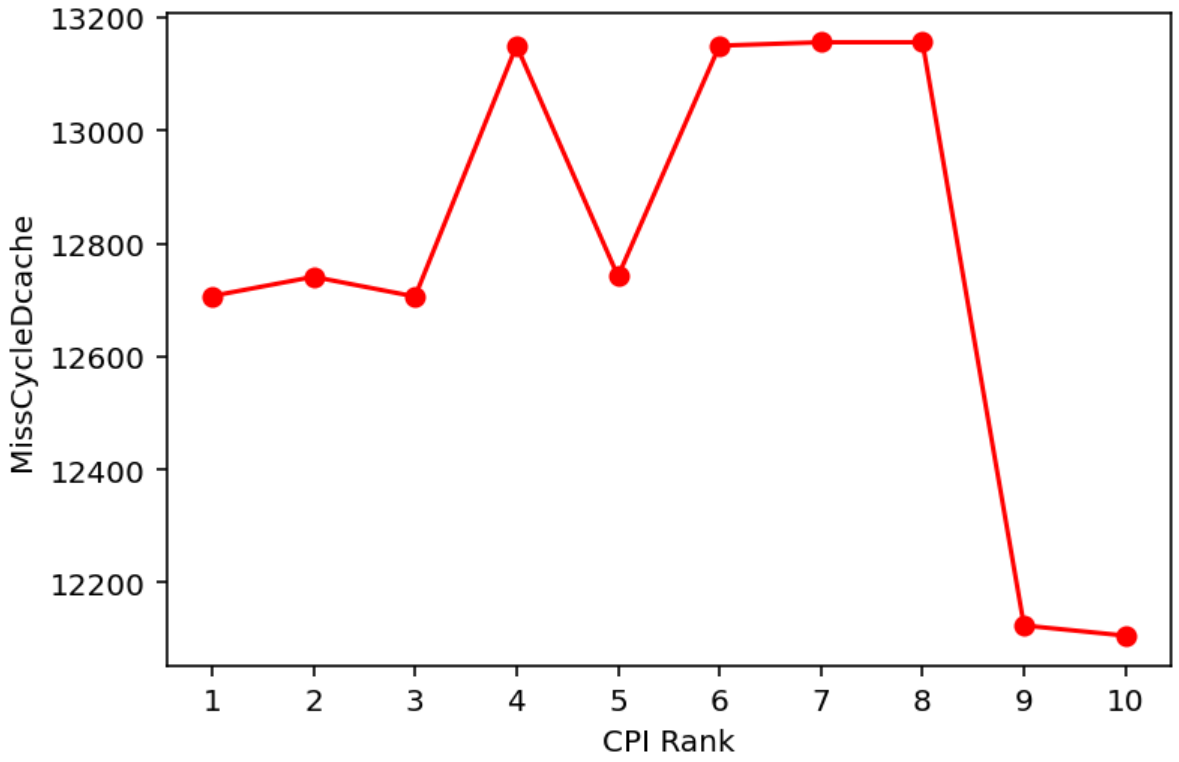

Fig 7: Number of Miss cycles for top 10 configurations - D cache

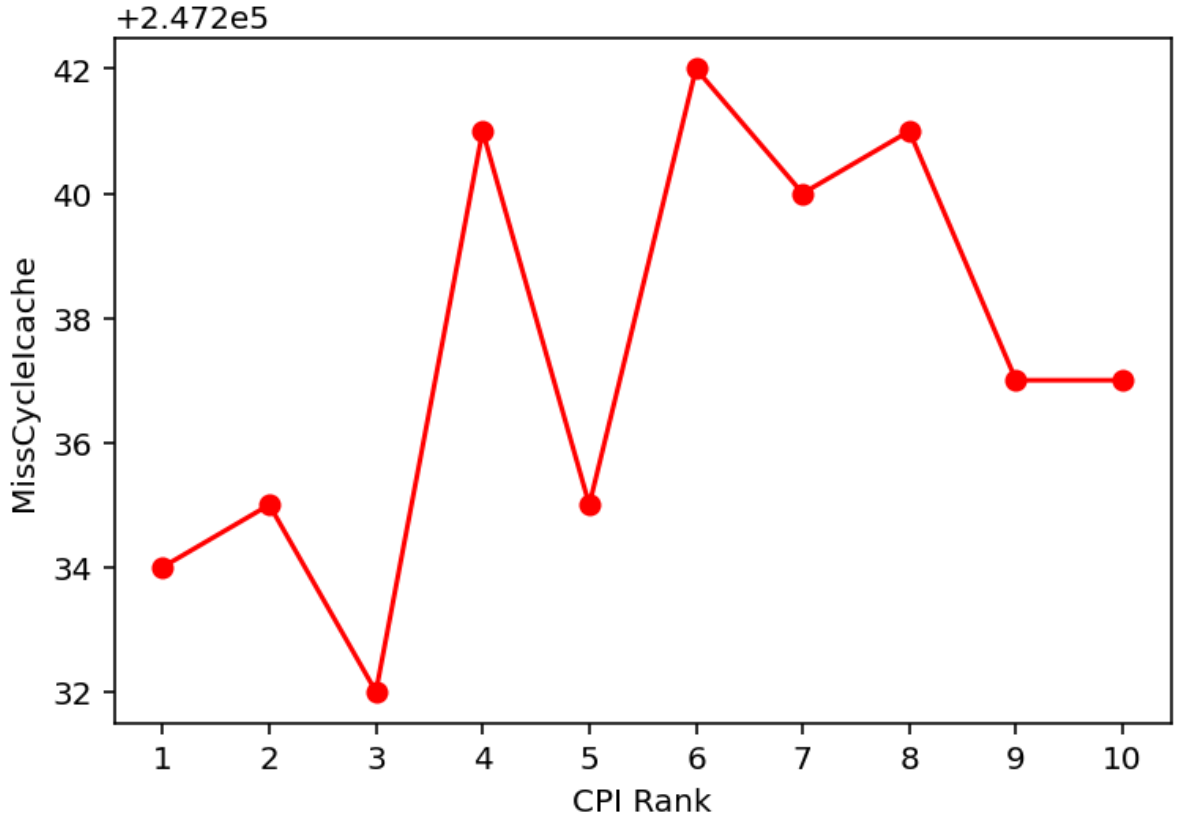

Fig 8: Number of Miss cycles for top 10 configurations - I cache

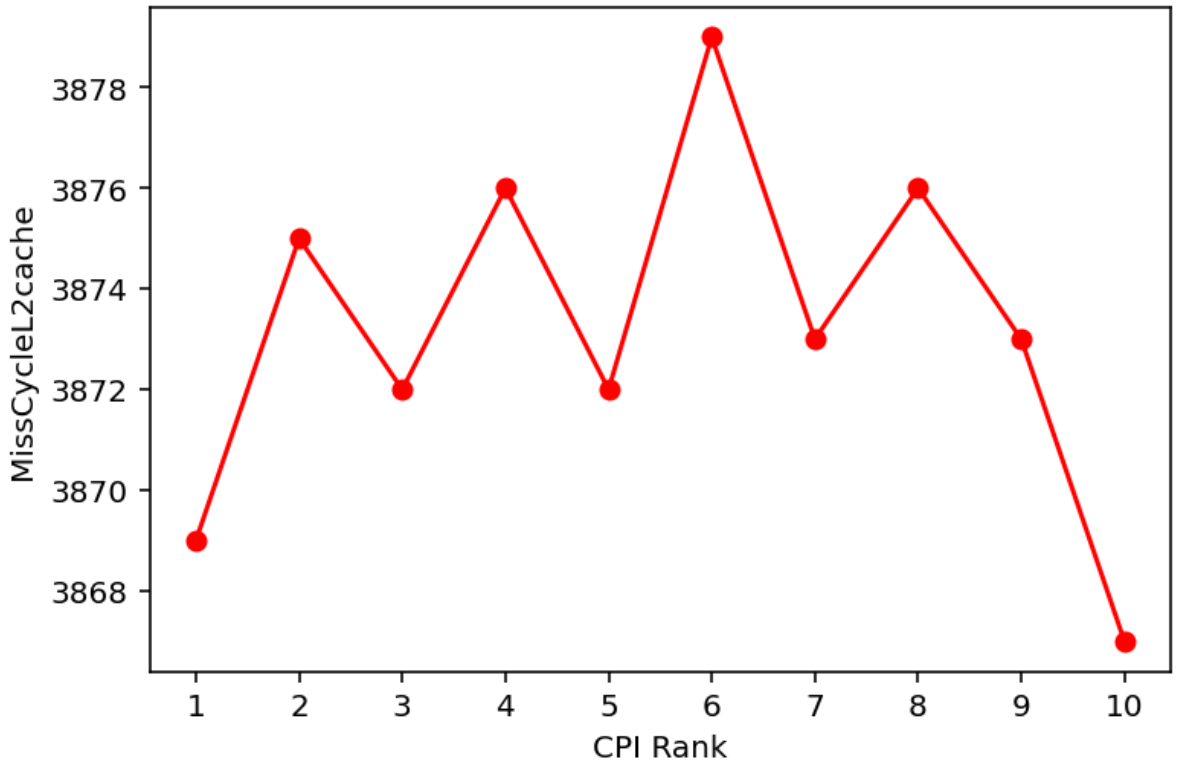

Fig 9: Number of Miss cycles for top 10 configurations - L2 cache

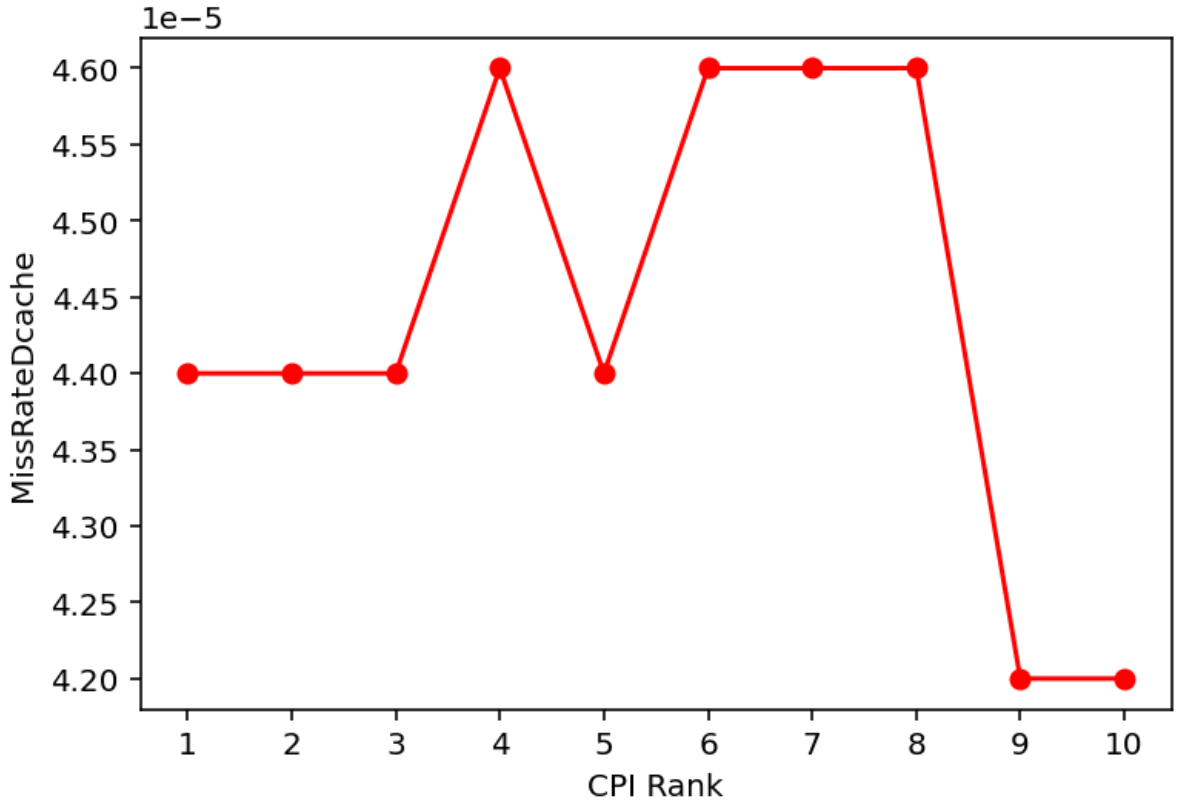

Fig 10: Miss rate for top 10 configurations - D cache

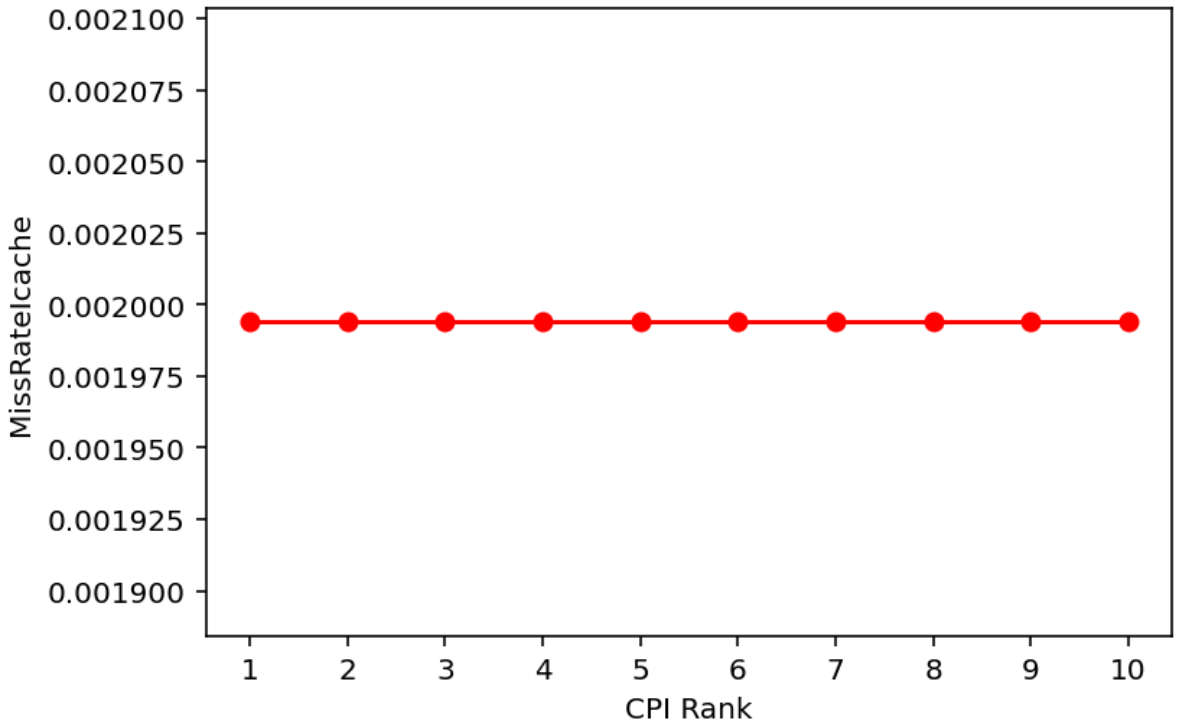

Fig 11: Miss rate for top 10 configurations - I cache

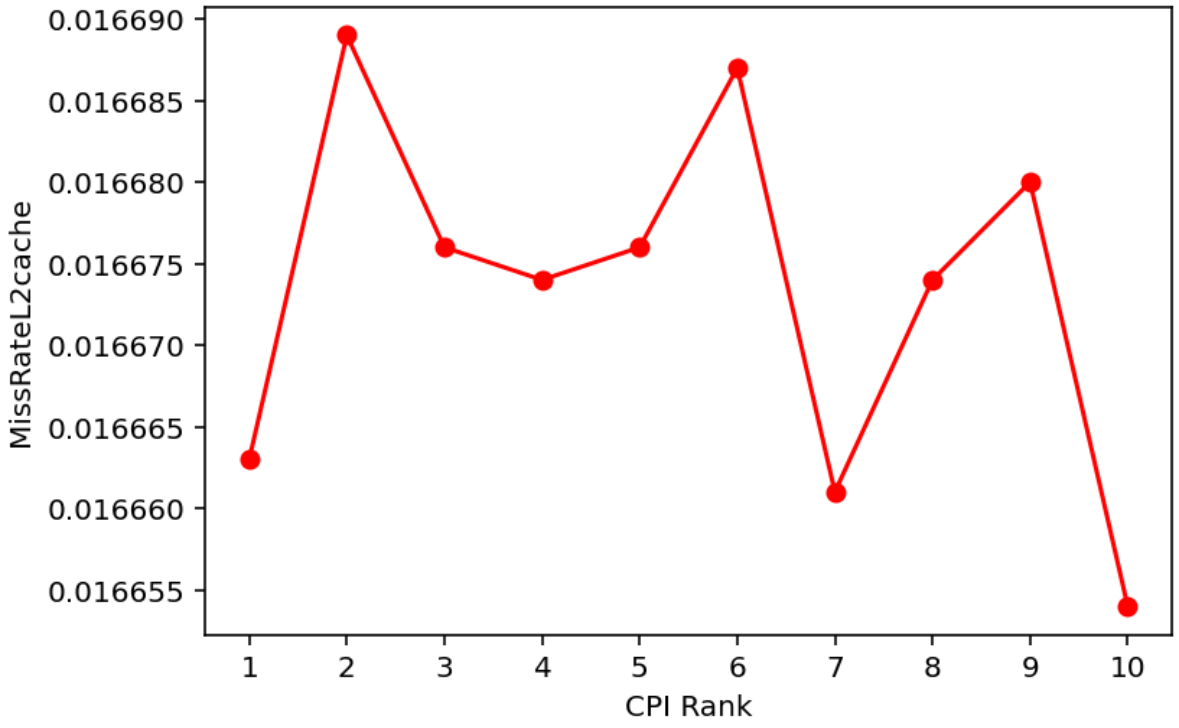

Fig 12: Miss rate for top 10 configurations - L2 cache

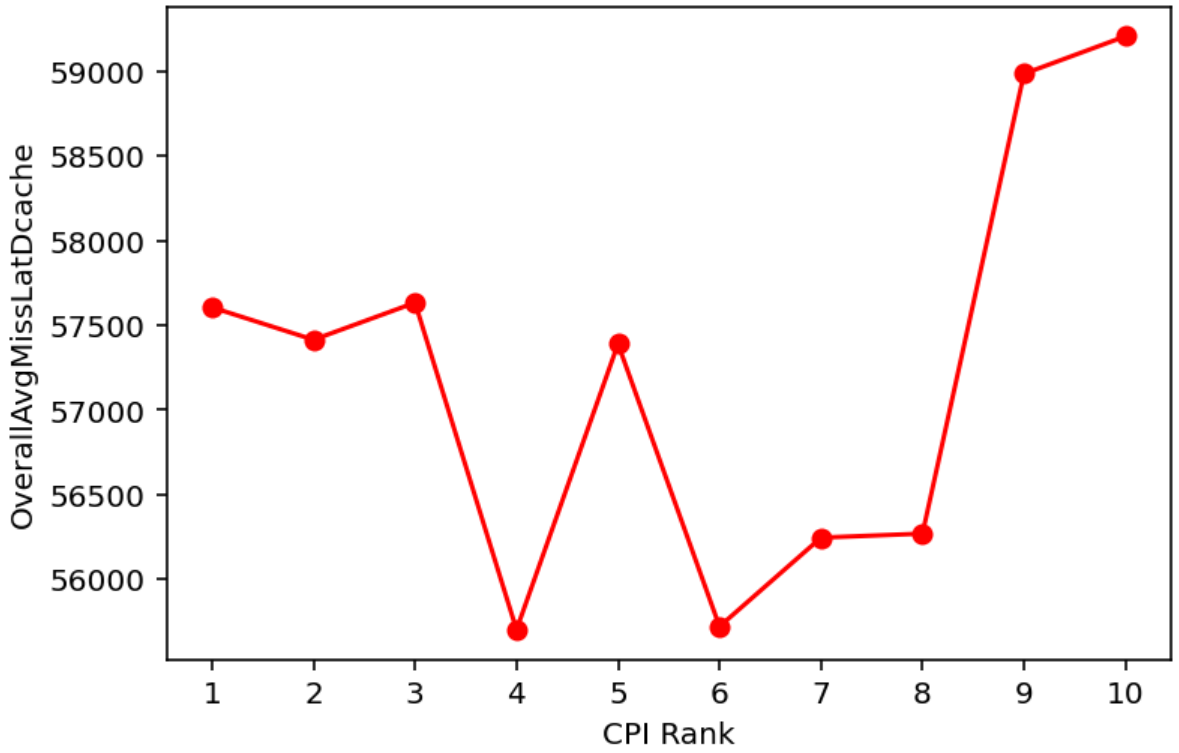

Fig 13: Overall average miss latency for top 10 configurations - D cache

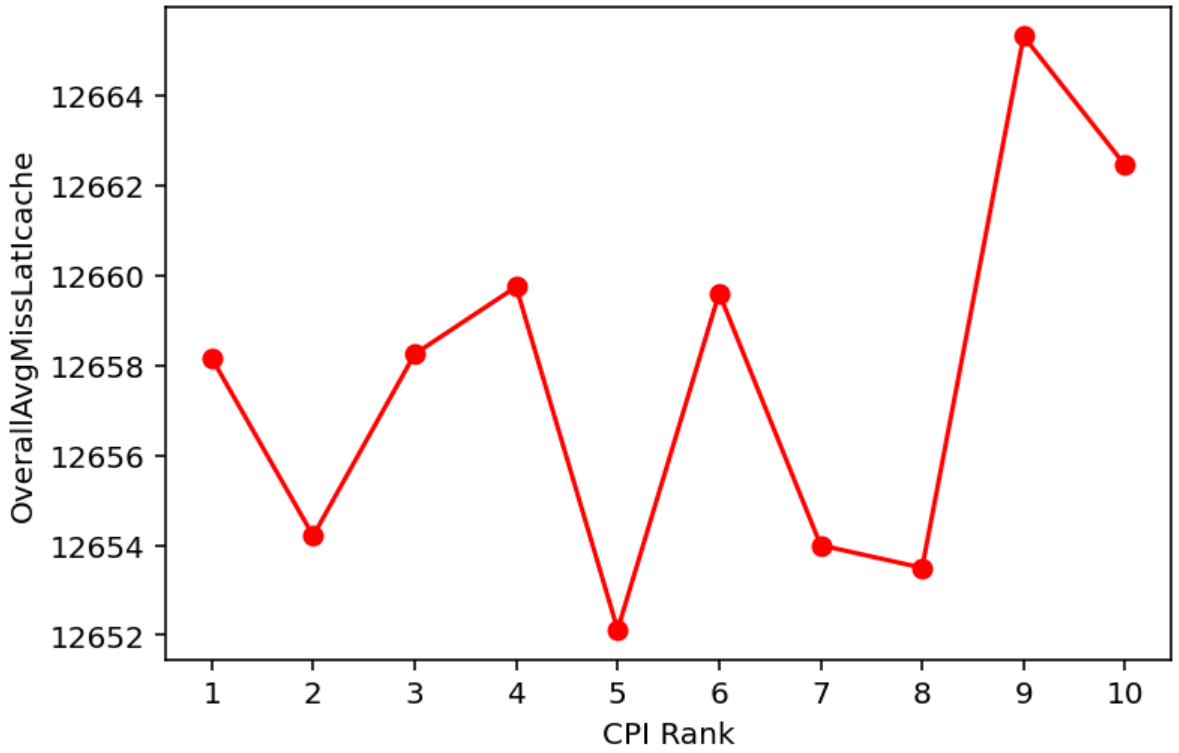

Fig 14: Overall average miss latency for top 10 configurations - I cache

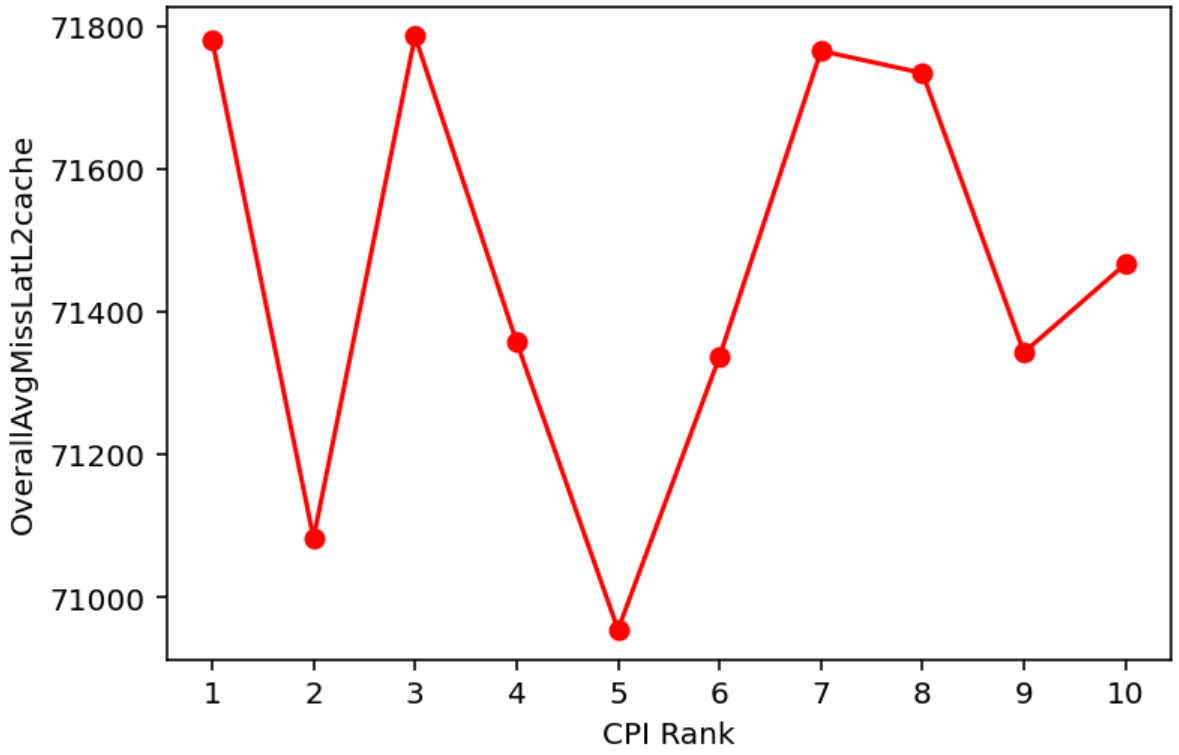

Fig 15: Overall average miss latency for top 10 configurations - L2 cache

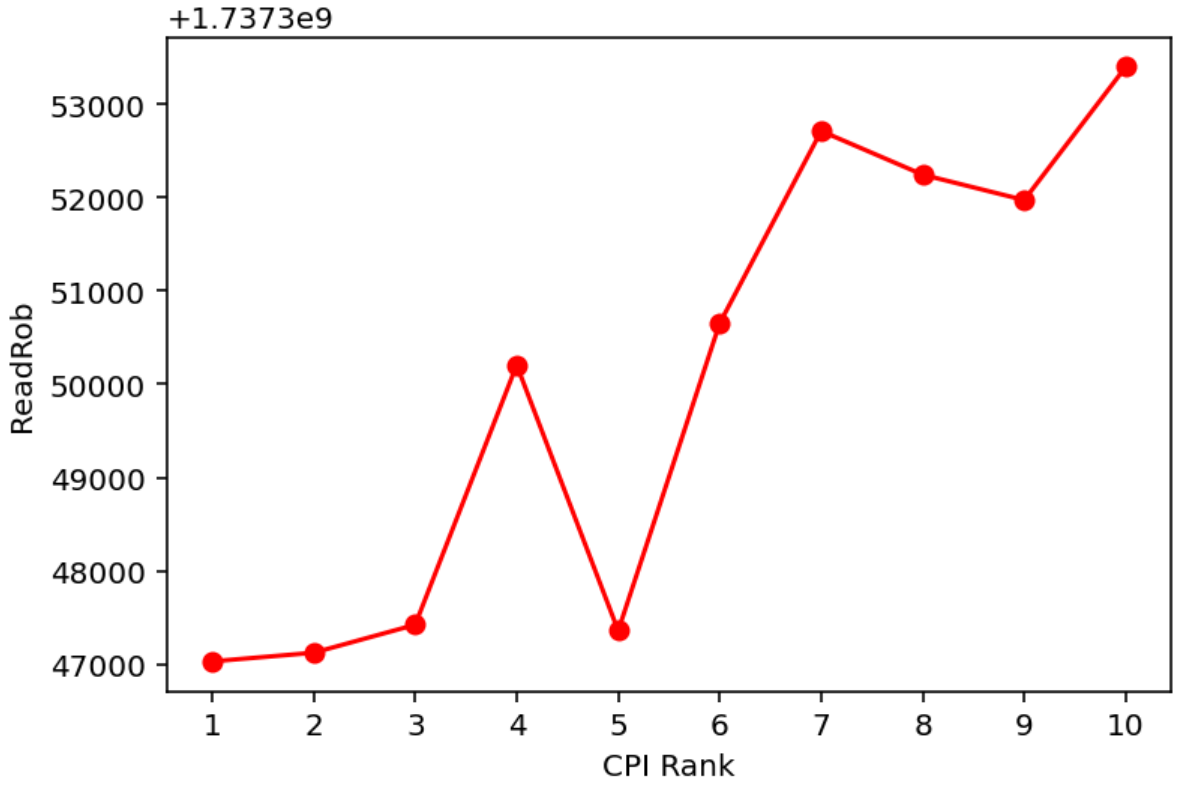

Fig 16: The number of ROB accesses (read)

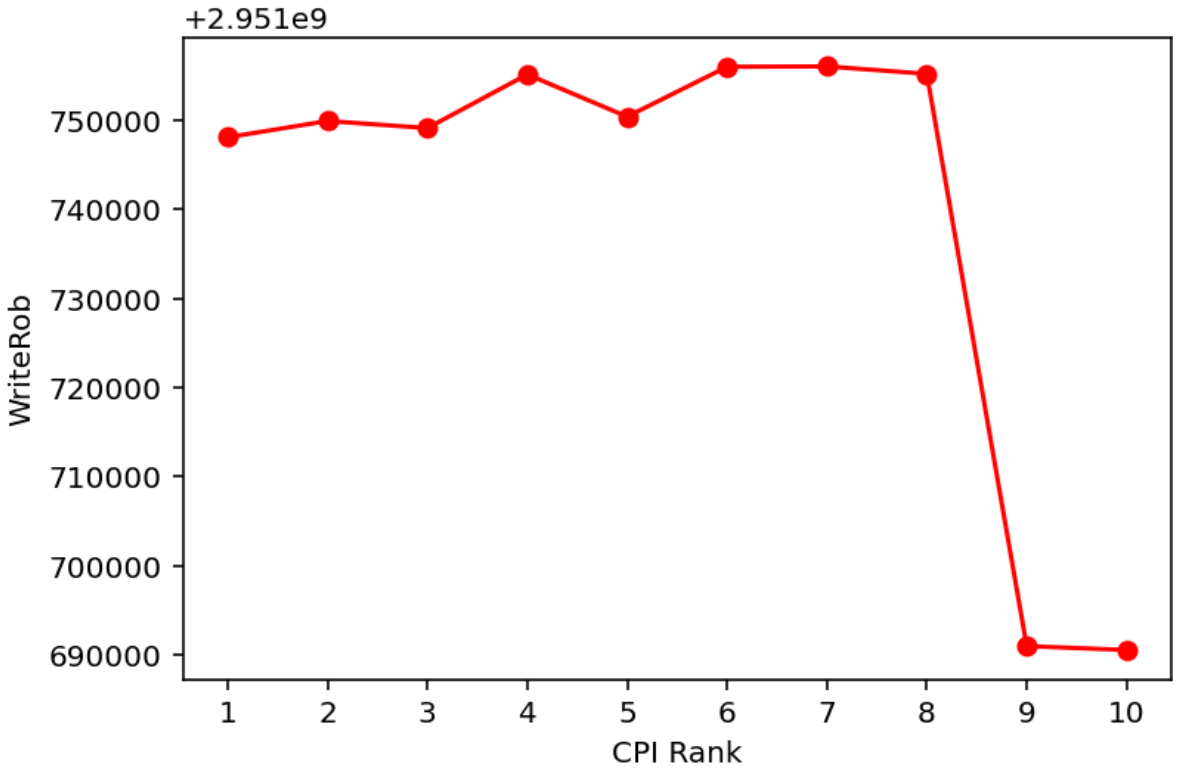

Fig 17: The number of ROB accesses (write)

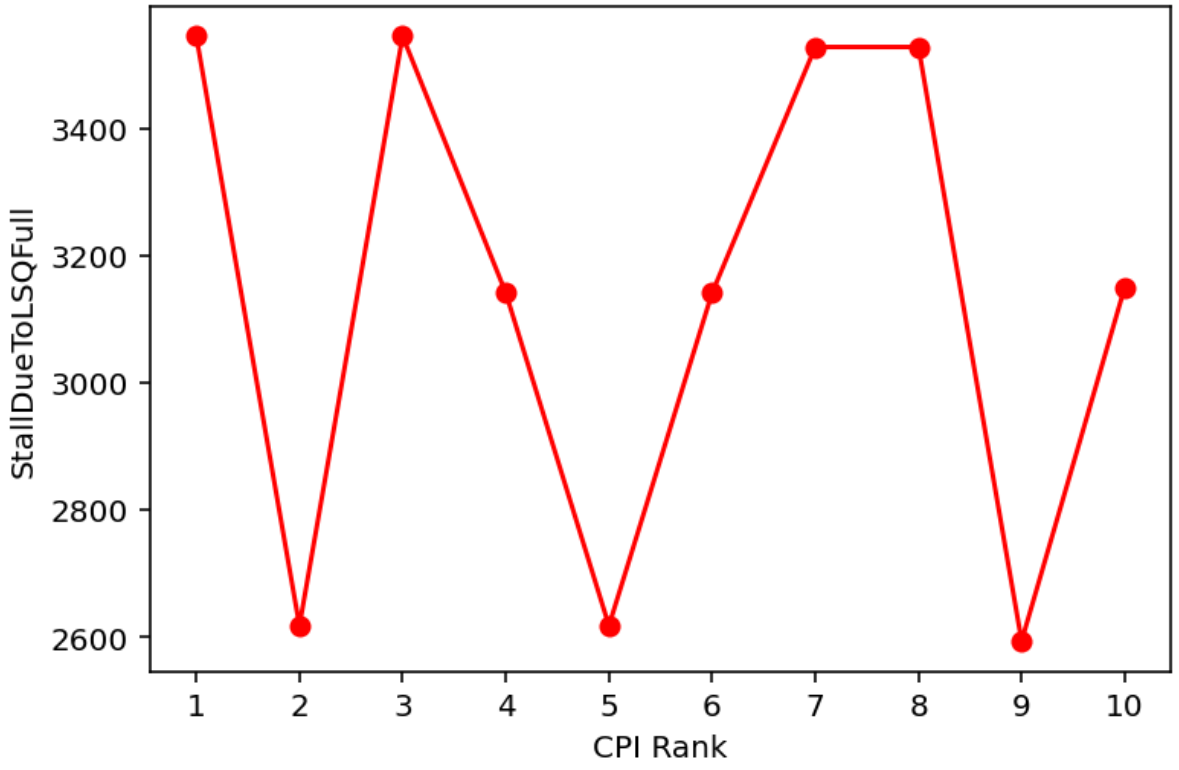

Fig 18: Number of times the LSQ has become full, causing a stall

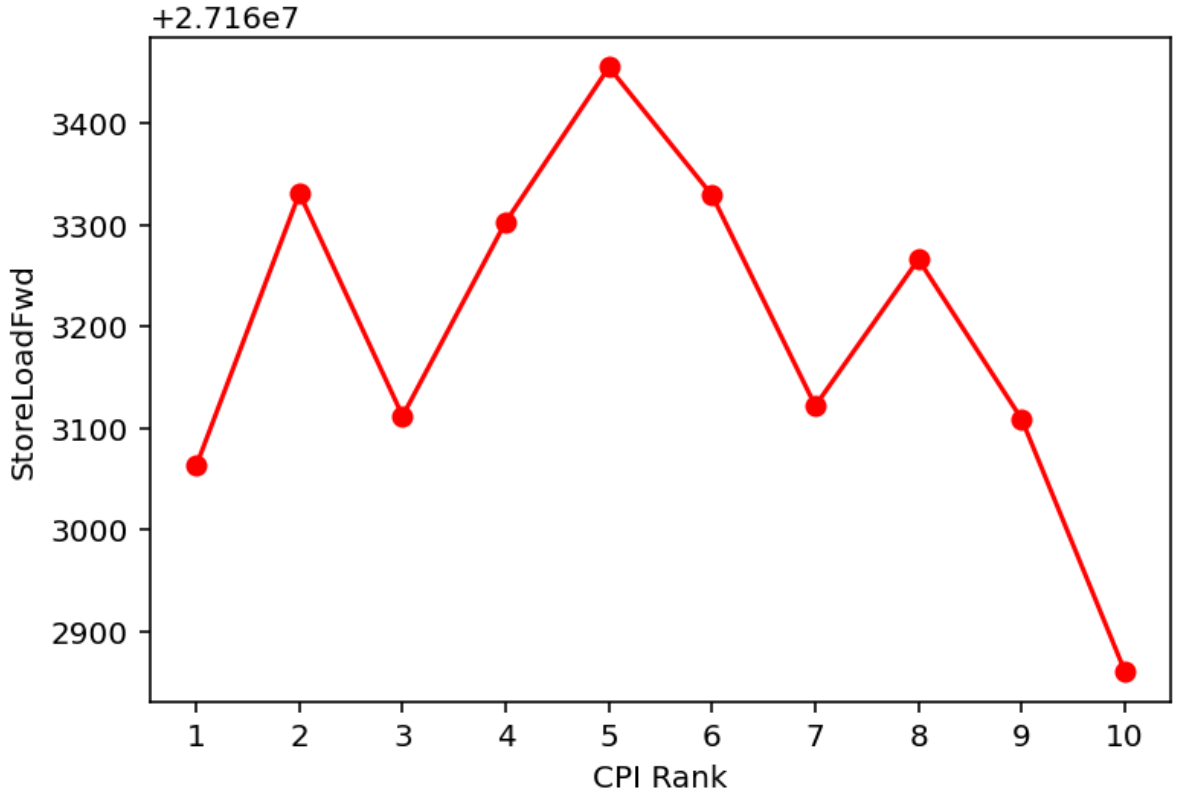

Fig 19: Number of loads that had data forwarded from stores

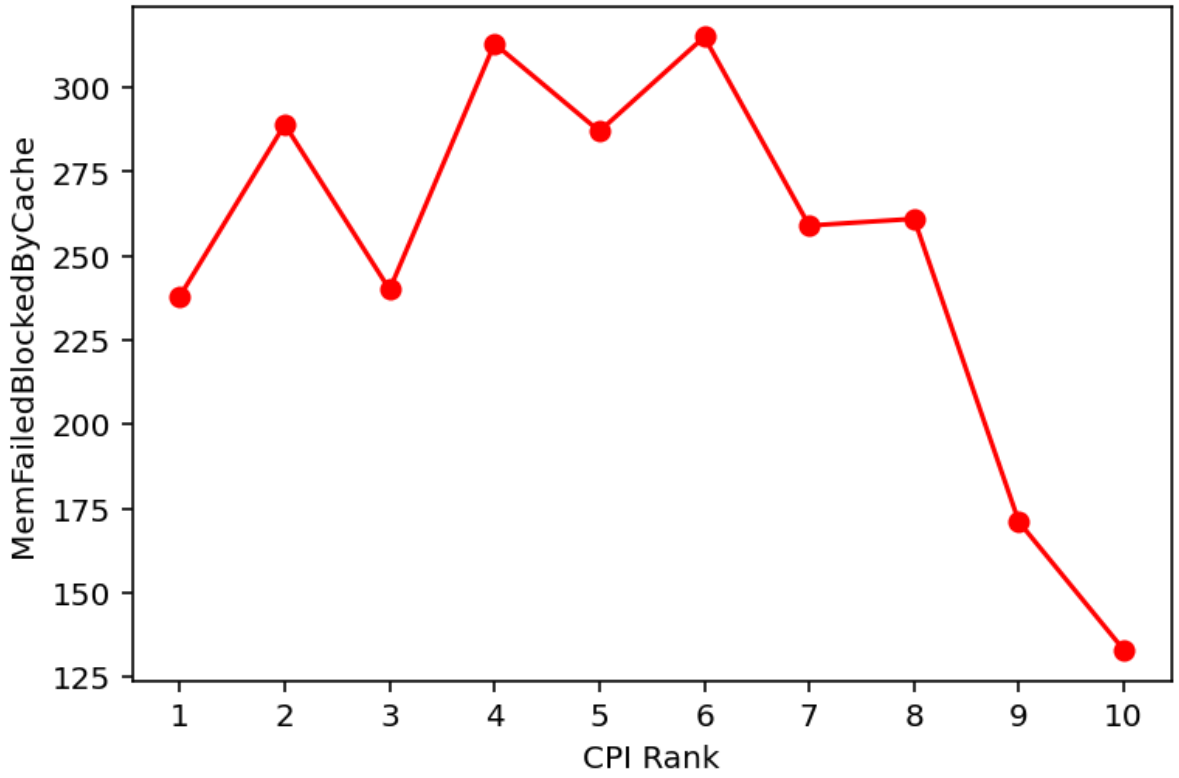

Fig 20: Number of times access to memory failed due to the cache being blocked

Contributions

S.No	Name	Contribution
1	Monal Prasad	Analyzing Statistics, Plotting Statistics, Preparing Report
2	Nisarg Upadhyaya	Programming and Simulating, Generating Results, Preparing Report
3	P Anurag Reddy	Analyzing Statistics, Preparing Report
4	Parth Jindal	Programming and Simulating, Generating Results, Plotting Statistics
5	Parth Tusham	Analyzing and Discussing Statistics
6	Piriya Sai Swapnika	Analyzing Statistics, Plotting Statistics, Preparing Report
7	Pranav Rajput	Programming and Simulating, Generating Results
8	Rajas Bhatt	Programming and Simulating, Generating Results, Plotting Statistics