High Performance Computer Architecture: Assignment 1

Group C

28-03-2021

1. The Top 10 Configurations

The benchmark program **qsort4.c** was compiled and its execution was simulated with the **gem5** simulator system across multiple varying parameters. These parameters and the possible values they could take up are:-

- **l1d_size** (Size of L1 data cache): 32kB, 64kB
- **l1i_size** (Size of L1 instrn. cache) : 32kB, 64kB
- **l2_size** (Size of L2 cache) : 128kB, 256kB, 512kB
- **l1_assoc** (Associativity of L1 cache) : 2, 4, 8
- **l2_assoc** (Associativity of L2 cache) : 4, 8
- **bp_type** (Type of branch predictor) : TournamentBP, BiModeBP, LocalBP
- **LQEntries** (No. of Load Queue Entries) : 16, 32, 64
- **SQEntries** (No. of Store Queue Entries) : 16, 32, 64
- **ROBEntries** (No. of RoB Entries) : 128, 192
- **IQEntries** (No. of Instrn. Queue Entries) : 16, 32, 64

The simulation was carried out with all possible configurations. By varying the above parameters over all their possible values, we simulated all **11664** configurations. From the statistics of the simulations, we extracted the **ten best** perfoming configuration with respect to **CPI**. These configurations are listed in Table 1.

No.	**CPI**	**l1d_size**	**l1i_size**	**l2_size**	**l1_assoc**	**l2_assoc**	**bp_type**	**LQEntries**	**SQEntries**	**RoBEntries**	**IQEntries**
1	0.53628	64kB	32kB	512kB	8	8	TournamentBP	64	16	192	64
2	0.53634	64kB	32kB	512kB	8	8	TournamentBP	64	64	192	64
3	0.53643	64kB	64kB	512kB	2	8	TournamentBP	32	16	192	64
4	0.53646	32kB	64kB	512kB	4	8	TournamentBP	64	64	192	64
5	0.53651	64kB	32kB	512kB	8	4	TournamentBP	64	16	192	64
6	0.53652	64kB	32kB	512kB	8	8	TournamentBP	32	32	192	64
7	0.53655	64kB	32kB	512kB	8	4	TournamentBP	64	64	192	64
8	0.53655	64kB	64kB	512kB	2	4	TournamentBP	32	16	192	64
9	0.53664	32kB	64kB	512kB	4	4	TournamentBP	64	64	192	64
10	0.53667	64kB	32kB	512kB	4	8	BiModeBP	64	64	128	64

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

1.1 Analysis

Out of all the 11, 664 configurations, these 10 perform the best for the execution of the **qsort4.c** program. We now analyse why these 10 configuration outperform the rest with respect to the characteristic of the benchmark program instructions. All the data used in the analysis comes from either the output **stats.txt** file of the simulations, or by determining the execution flow of the source program **qsort4.c**.

- **Size of L1 data cache**: This parameter 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, almost all of the top 10 configurations have l1d_size parameter as 64 kB. But it's 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 the overall miss rate difference between the caches of both size is very small (0.036 in 64kB and 0.038 in 32 kB) as seen from stats.txt, two 32 kB sizes are also observed in top 10 together with L1 associativity as 4, which again is faster to search than a L1 cache associativity of 8 in other cases. We believe that this faster search time compensates for the slightly greater miss rate in the 32 kb L1 data cache.

- **Size of L1 instrn. cache**: This parameter 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 equally take both the possible values and the size doesn't seem to have much effect on the performance of our benchmark program. This is a consequence of the fact that there aren't many different or wild branches in the source code as well as the source code size being quite small. The stats.txt tells us that the overall instruction cache miss rate is typically about 0.0006 which is practically negligible.

- **Size of L2 cache**: This parameter controls the size of the unified level 2 cache. The cache at level 2 is not split into instruction and data cache. All the top ten configurations have their L2 cache size as 512 kB. As the miss rate at level 1 cache is already very low, overall number of accesses to the L2 cache is also quite low and most of which are misses as apparent from stats.txt: overall miss rate is approx 0.6. Still, a 512 kB cache shall indisputably perform better than a lower sized cache in subsequent accesses due to the advantages gained from spatial and temporal locality, considering the fact that misses at level 2 are quite costly.

- **Associativity of L1 cache**: It represents the number of blocks in both the level 1 caches in which the data fetched from the memory can possibly be mapped into. Generally higher associativity gives better results when matched with a bigger cache size, as it decreases the conflict misses while also making sure spatially contiguous data doesn't wrap around that often. This is backed by the fact that half of the 10 best configurations have 64kB cache sizes with 8 way set associativity. Brief occurrences of lower associative caches are accompanied by large instruction cache sizes. Moreover, associativity of 2 is always accompanied with both the L1 caches being 64kB in size. These configurations more or less perform as well as the top configurations because the larger cache sizes compensate for the increase in conflict misses due to lower associativity. Hence they too occur in the top 10 configurations.

- **Associativity of L2 cache**: It represents the number of blocks in level 2 cache in which the data fetched from the memory can possibly be mapped into. From the table above, it is clear from comparing the 4th and 9th best configurations, that a higher associativity leads to better performance for the level 2 cache, as all the other parameters are the same and only l2_assoc changes. This is because the miss rate in level 2 cache can increase due to more number of conflict misses that arises due to the lower associativity of 4 as compared to the higher associativity of 8.

- **Type of Branch Predictor**: This parameter controls the type of branch predictor used to predict whether the branches fetched are taken or not. From the stats.txt we can see that, of the branches predicted during the simulation, approximately 50% are conditional, 17% are unconditional immediate, 17% are returns, and 16% are indirect branches. Among these branches, the branch predictor affects only the conditional branches. In our benchmark program, a major part of such branches executed compromises of just two particular branches - 1) the comparison in the *compar_double* function, and 2) the while loop in the *qsort* function. Moreover, *branch 2* is dependent on the decision of the *branch 1*. Such a prediction is pattern is best modelled by the Tournament Predictor (it can use the local predictor for *branch 1*, and the global predictor for the dependent *branch 2*), and hence it can be found in the top 9 configurations. But another fact is that, modelling this prediction scheme with a local 2 bit saturating counter (BiModal Predictor), also works nearly the same, but it suffers in a particular branch, where *Taken* and *Not Taken* occurs alternatively. The BiModal Predictor has a poor performance for such patterns (only 50% time correctly predicted), but as this branch occurs much less frequently, the difference between the misprediction rate of the Tournament and BiModal Predictor is less than 0.1%. Therefore, the BiModal Predictor managed to squeeze into the top 10 configuartions.

- **No. of Load Queue entries**: This parameter controls the maximum number of pending load operations either waiting for address dependency or resolving of the addresses of previous store operations. A bigger sized LQ will result in a better performance, as can be seen from the fact that a majority of the top 10 configurations having a 64 entry LQ. But, due to almost all the load requests being temporally or spatially local, the requests are resolved without much delay. Moreover, over 50% of the loads had data forwarded from stores. Thus, the size of LQ might not have as much as effect as the other parameters (except for SQ size) on the performance of our benchmark program.

- **No. of Store Queue entries**: This parameter dictates the maximum number of pending store instructions that can be buffered before being executed. In general, increasing the SQ size might lead to an increase in the performance as the SQ will help buffer the store instructions waiting for their dependencies to be ready. But in the case of our benchmark program, a majority (90+% of the total stores when the branches are predicted correctly and we do not care about the case when branches are mispredicted as these instructions are going to be squashed either way) of the stores happening are due to the loop increments, and the call to the *compar_double* function. In both of these cases, the address and the value to be stored are almost immediately ready. For the case of the function call to the *compar_double*, the stores happen at the address pointed by the stack register (which does not change unless a function is called) and the value to be stored can be easily retrieved due to spatial and temporal locality. In the case of the loop increments, the value is just an ADDI operation (one of the fastest executing) away and the address is in the register which doesn't change unless the loop terminates. Overall, both these store instructions execute almost immediately, and hence the SQ hardly ever gets full, thereby not affecting the performance of the simulation significantly irrespective of its size.

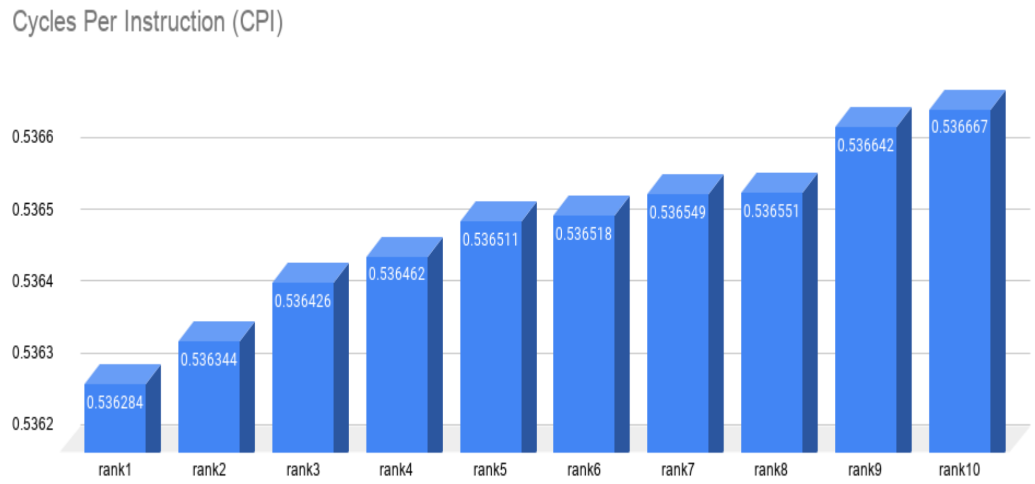
- **No. of RoB entries**: This parameter corresponds to the maximum number of entries the Reorder Buffer (RoB) can hold. As we know, the entries present in the RoB keeps track of the instructions currently in the instruction window and so, theoretically, bigger the RoB capacity, better the performance, as it can accommodate more instructions in case a long latency operation is stalling the commit. But, if we consider our benchmark program, the only long latency operations that could possibly occur is a cache miss at both L1 and L2. But from the stats.txt file, the number of such instructions is just around 0.01%. Hence, though a increase in the RoB size can improve the performance, it does not does so drastically. Moreover, the instruction window size can further be constrained by other factors such as the physical register file size, etc., and so, increasing RoB size after a point also results in diminishing returns.

- **No. of Instrn. Queue entries**: The number of Instruction Queue (IQ) entries specifies the size of the IQ. The IQ buffers the instructions which are decoded and are ready to be issued. As can be observed, all the top ten configurations have a IQ of size 64. This can be considered a consequence of the lesser 32 sized IQ unable to support the issue width of 8 at which the *gem5* system simulator operates, because of the large number of low latency operations in the benchmark program. By making full use of the multi issue system, a bigger IQ results in a better performance in the overall CPI.

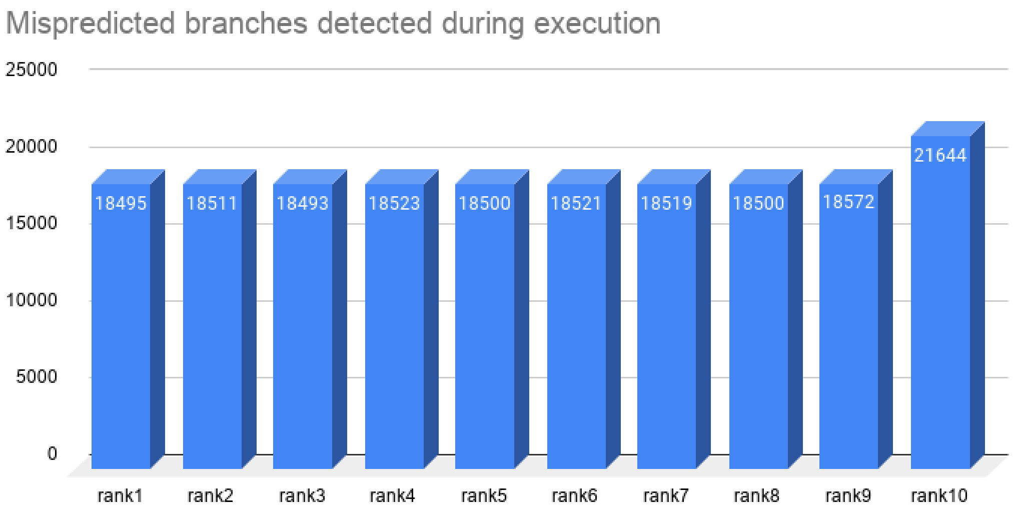
2. Plots

We will show plots/graphs on various statistics for the above mentioned top 10 configurations.

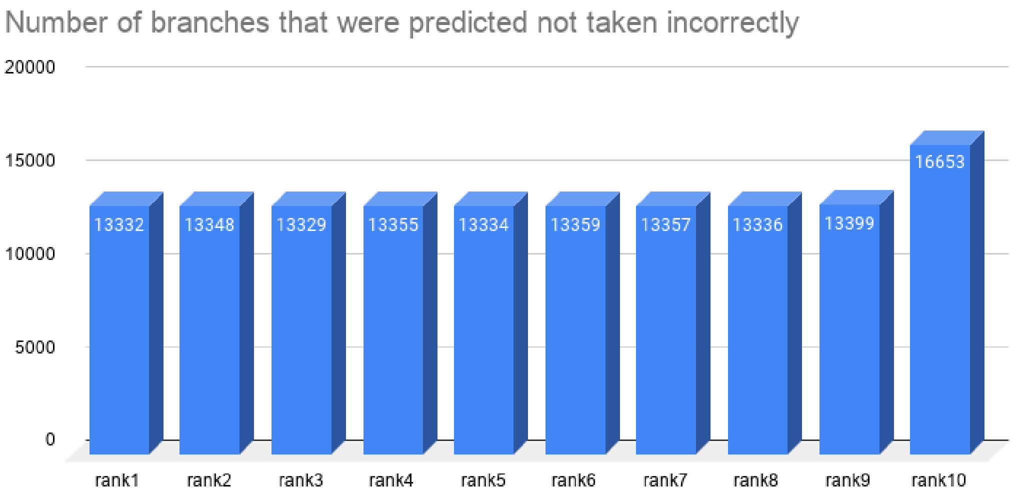
1. **Cycles Per Instruction (CPI)**



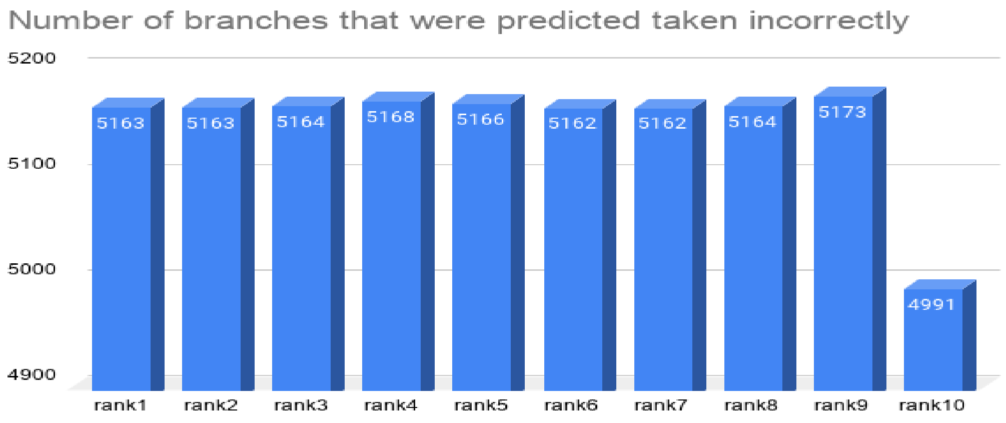
2. **No. of mispredicted branches detected during execution**


Mispredicted branches detected during execution

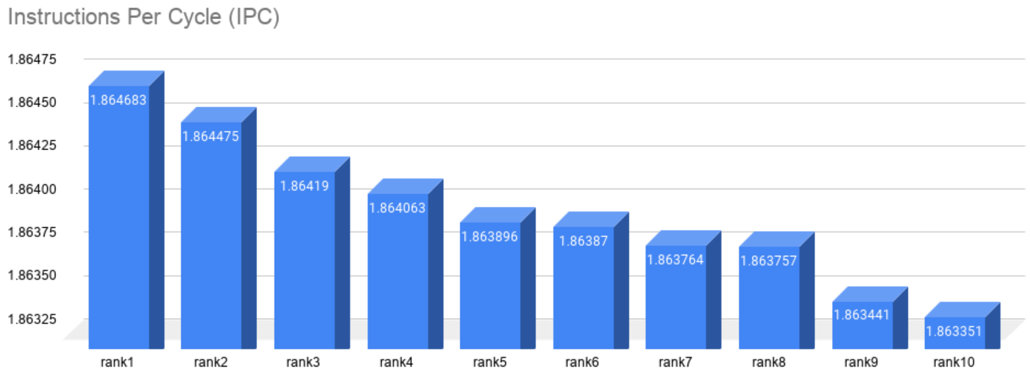
3. **No. of branches predicted not taken incorrectly**


Number of branches that were predicted not taken incorrectly

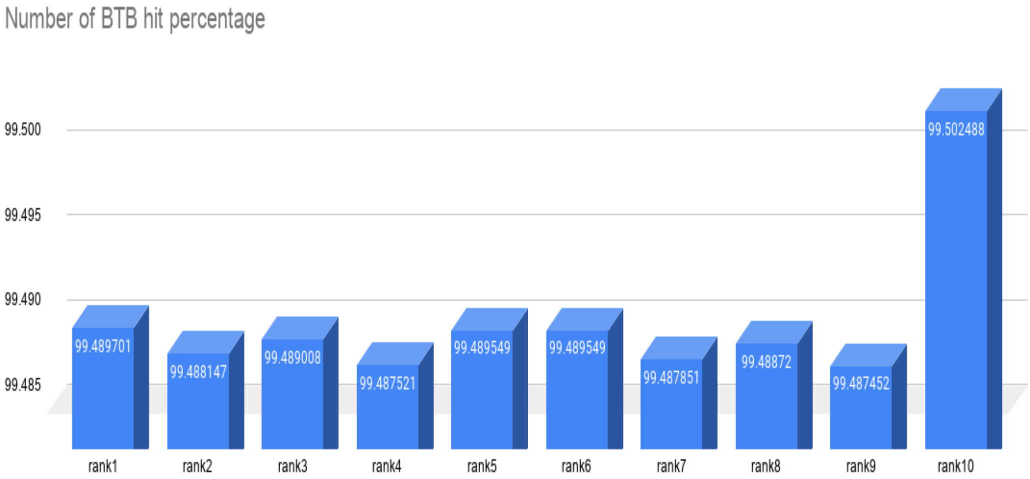
4. **No. of branches predicted taken incorrectly**


Number of branches that were predicted taken incorrectly

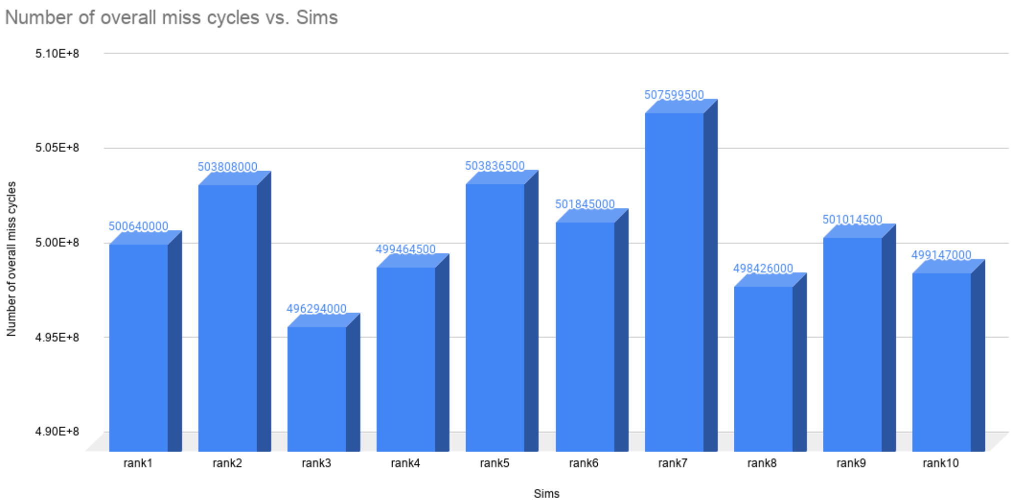
5. **Instructions Per Cycle (IPC)**

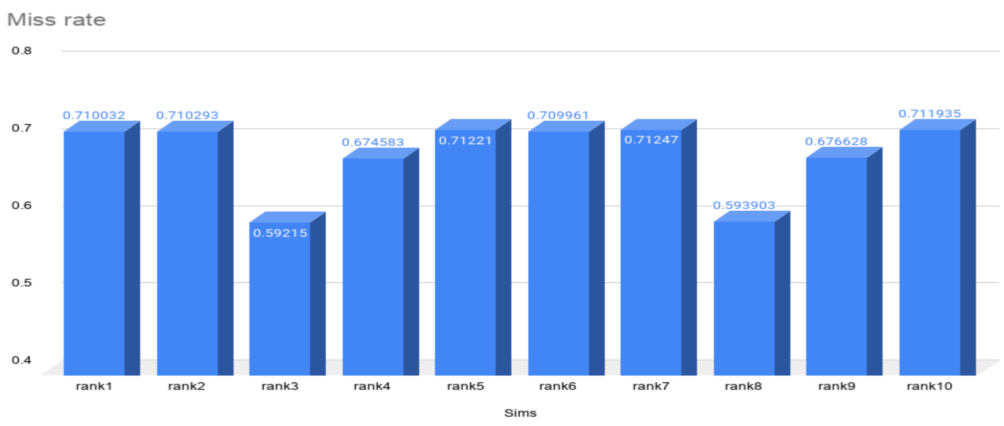


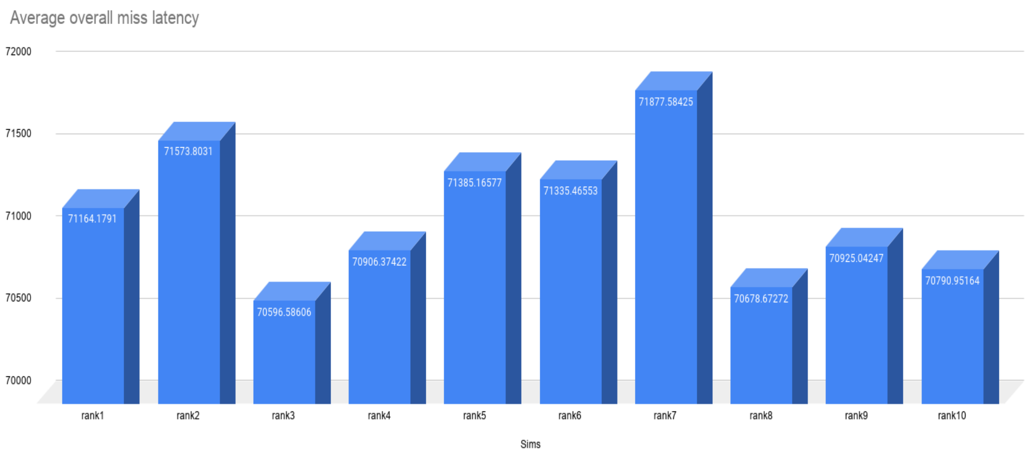
6. **BTB hit percentage**



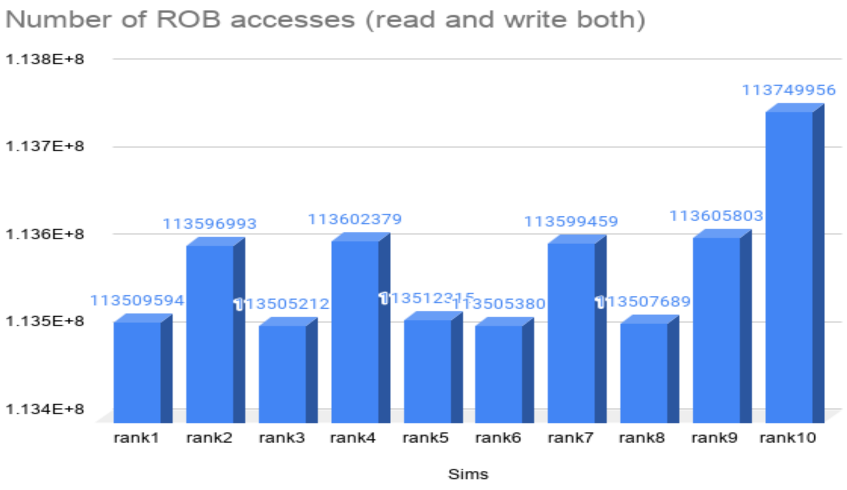
7. **No. of overall miss cycles, miss rate, avg. miss latency (for L2 cache)**



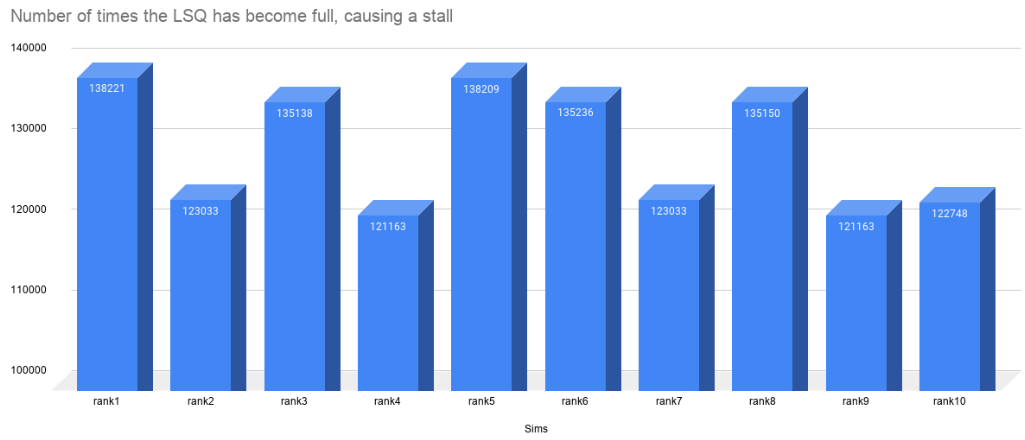




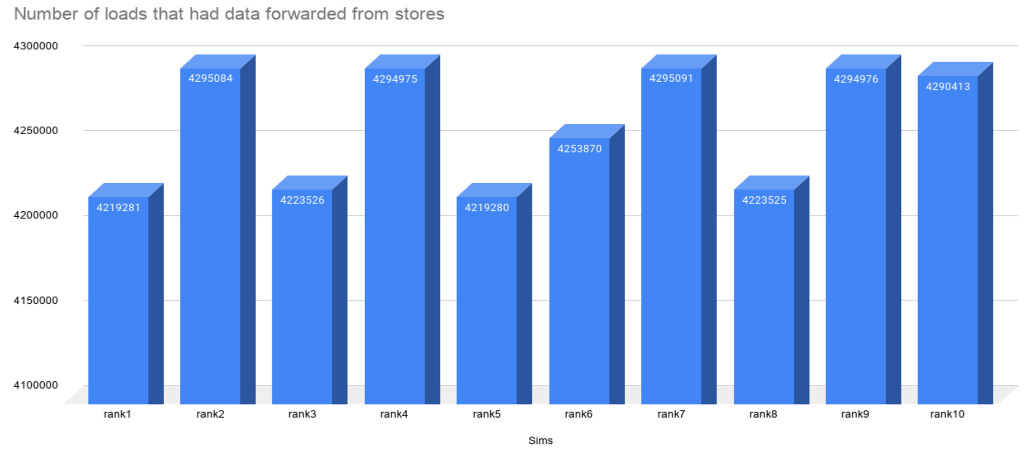
8. **No. of RoB accesses**



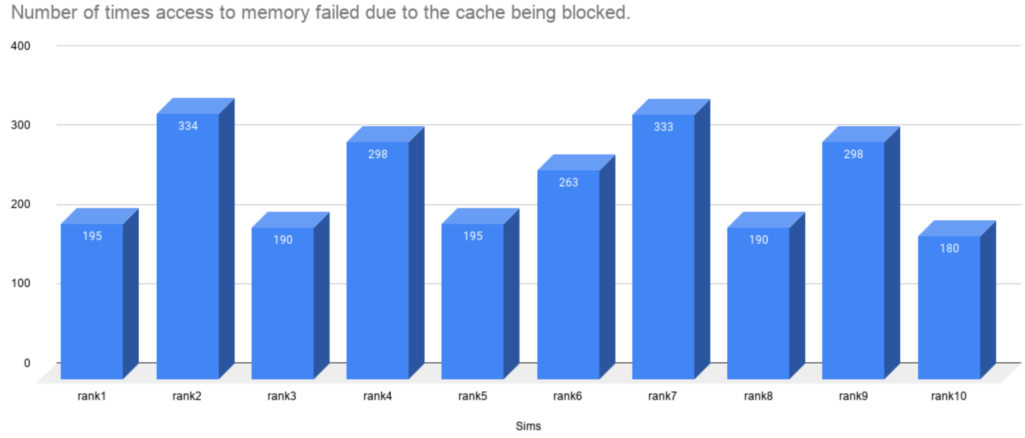
9. **No. of times stalled due to LSQ being full**



10. **No. of times load data forwarded from store**



11. **No. of times memory access failed due to cache block**



3. Group Members and Contribution

1. **Kshitiz Sharma (17CS30021)**
   - Running 1000 simulations
2. **Kousshik Raj (17CS30022)**
   - Creating the custom config script
   - Creating bash script for automating the simulation
   - Running 1600 simulations
   - Analysing the reasons for the performance
   - Creating the report
3. **Naimesh Pramanik (17CS30023)**
   - Plotting graphs for the statistics
4. **Nari Rohini (17CS30024)**
   - Running 1000 simulations
5. **Pericerla Amshumaan Verma (17CS30025)**
   - Running 1000 simulations
6. **Praagy Rastogi (17CS30026)**
   - Running 1000 simulations
   - Analysing the reasons for the performance
7. **Ritikesh Gupta (17CS30027)**
   - Running 1000 simulations
8. **Rohit (17CS30028)**
   - Running 1000 simulations
9. **Rohit Pathak (17CS30029)**
   - Extracting the data for plotting
   - Plotting graphs for the statistics
10. **Sanket Rajendra Meshram (17CS30030)**
    - Running 1000 simulations
11. **Sashank Bonda (17CS30031)**
    - Running 1000 simulations
12. **Shamin Aggarwal (17CS30032)**
    - Running 1000 simulations
13. **Shivam Kumar Jha (17CS30033)**
    - Running 1000 simulations
    - Synchronising the simulations ran across different devices
    - Extracting the 10 best configurations