Comparison of Cache Replacement Policies using Gem5 Simulator

Under the guidance of
Gita Alaghband

Bhagyashree Borate 107536335
Nivin Anton Alexis Lawrence 107078877
Sri Divya Anusha Basavaraju 106752423

University of Colorado Denver

Table of Contents

ABSTRACT

A cache replacement algorithm is the algorithm that decides what cache elements to evict when a cache reaches it's capacity. There are many replacement policies and cache configurations implemented over years and there are few policies and configurations that are proven to be efficient. In our project we mainly focus on understanding the existing norms by trying out different cache configurations and study its performance. This analysis will be done with the help of GEM5 simulator, using benchmark with varying workloads and report its performance metrics.

In the project we have taken four major components that affect the cache performance - cache associativity, memory hierarchy, cache replacement algorithm and cache size. We have gone one step ahead by trying out various configurations that are different from the norm.

KEY WORDS:

Cache memory and performance, cache replacement policies, performance evaluation, Gem5, inclusive and exclusive caches.

INTRODUCTION AND MOTIVATION

A cache replacement algorithm is the algorithm that decides what cache elements to evict when a cache reaches it's capacity. Latency and Hit rate are the major factors affecting the cache performance. Hit rate measurements are typically performed on benchmark applications. The actual hit ratio varies widely from one application to another. Understanding different cache replacement algorithms and their performance on various applications via spec benchmark helps us to come up with newer cache replacement policy.

As part of project, we are going to use GEM5 simulator in order to perform the cache replacement policy comparisons. GEM5 is an Open source, modular, object-oriented system architecture simulator platform. Simulation of full system with devices and operating system can be done efficiently and easily with GEM5. It provides support for Multiple Instruction Set Architectures (ISAs) and multiple processor cores. GEM5 allows us to modify the features and design as per our project.

As part of the project, the end deliverables will be comparing the existing cache replacement policies using benchmark and analyzing their performance. By this activity, we would gain strong understanding on cache policies and how it plays a vital role in overall CPU performance.

RESEARCH ANALYSIS ON CACHE

While a lot of research has been done in the field of cache memories, some important questions about cache replacement policies applied to the state-of-art workload still remain without complete answers. In this section we list the well-known observations about replacement policies, along with related questions to which our study offers answers.

Memory Hierarchy and Cache Levels

Memory hierarchy in general is designed in such a way that having lower and faster cache near the processor and bigger and slower cache at the memory side to yield better performance. In this project we have experimented by swapping the L2 and L3 cache and found interesting behavior; the new configuration being L1 cache, L3 and L2 respectively. With the above-mentioned configuration we also tested it by making L3 and L2 caches exclusive.

Replacement Algorithms

The cache replacement algorithms have a strong bias to the nature of the program and performance as a whole. To understand this better, we wanted to break the existing norm of having same replacement algorithms at all levels and try different cache replacement policies at different levels.

Associativity

Larger sets and higher associativity lead to fewer cache conflicts and lower miss rates, but they also increase the hardware cost. So, coming up with the right associative value for each cache level is a challenge and hence we wanted to study by varying the values and will provide the one that shows good performance.

COMMON CACHE REPLACEMENT POLICIES

Cache memories remain as one of the hot topics in the computer architecture. The ever-increasing speed gap between processor and memory emphasizes the need for efficient memory hierarchy and revisit the effectiveness of common cache replacement policies. When the cache memory is full and a new block of memory needs to be placed in the cache, the cache controller must discard an already existing cache memory line and replace it with the new line. Preferably, the cache block to be evicted will not be used in the future. Since it is generally impossible to predict how far in the future the information is needed, the existing replacement policies predict based on the behavior of the cache.

Least Recently Used (LRU)

Discards the least recently used items first. This algorithm requires keeping track of what was used when, which is expensive

if one wants to make sure the algorithm always discards the least recently used item. General implementations of this technique require keeping "age bits" for cache-lines and track the "Least Recently Used" cache-line based on age-bits. In such an implementation, every time a cache-line is used, the age of all other cache-lines changes.

In LRU replacement, a line, after its last use, remains in the cache for a long time until it becomes the LRU line. Such deadlines unnecessarily reduce the cache capacity available for other lines. In addition, in multilevel caches, temporal reuse patterns are often inverted, showing in the L1 cache but due to the filtering effect of the L1 cache, not showing in the L2 cache. At the L2, these lines appear to be brought in the cache but are never re-accessed until they are replaced. These lines unnecessarily pollute the L2 cache.

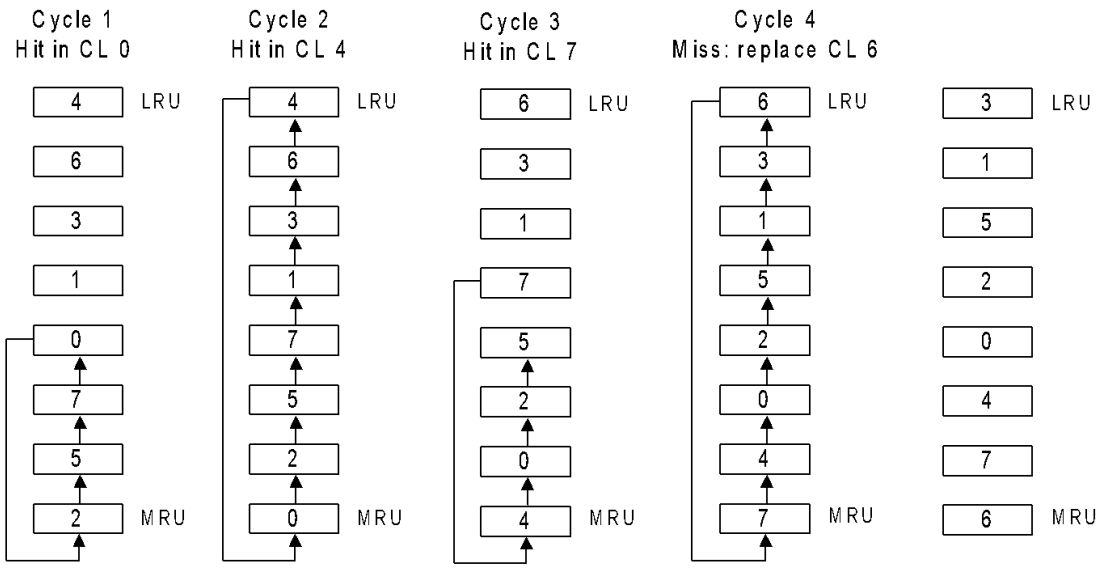


Figure 1. An example of Least Recently Used Policy.

First In First Out (FIFO)

Round Robin (or FIFO) replacement heuristic simply replaces the cache lines in a sequential order, replacing the oldest block in the set. Each cache memory set is accompanied with a circular counter which points to the next cache block to be replaced; the counter is updated on every cache miss. Using this algorithm the cache behaves in the same way as a FIFO queue. The cache evicts the first block accessed first without any regard to how often or how many times it was accessed before.

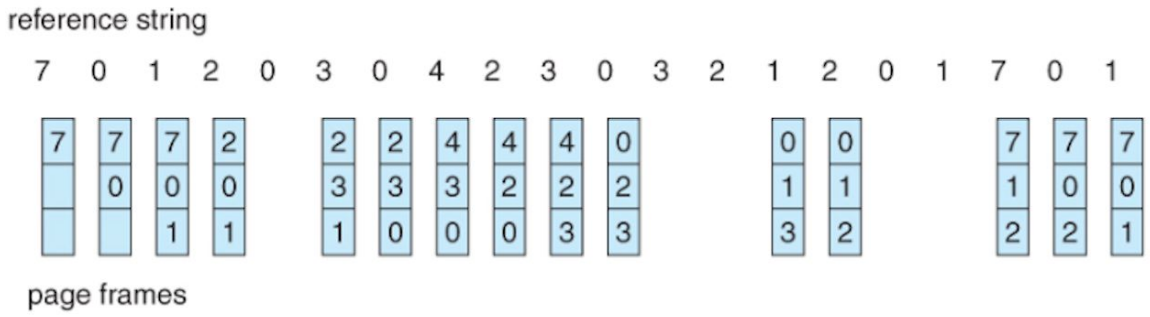


Figure 2. FIFO replacement policy with an example.

Random Replacement

Randomly selects a candidate item and discards it to make space when necessary. This algorithm does not require keeping any information about the access history. For its simplicity, it has been used in ARM processors.

INCLUSIVE AND EXCLUSIVE CACHES

Multi-level caches can be designed in various ways depending on whether the content of one cache is present in other level of caches. If all blocks in the higher-level cache are also present in the lower level cache, then the lower level cache is said to be inclusive of the higher-level cache. If the lower level cache contains blocks that are not present in the higher-level cache, then the lower level cache is said to be exclusive of the higher-level cache.

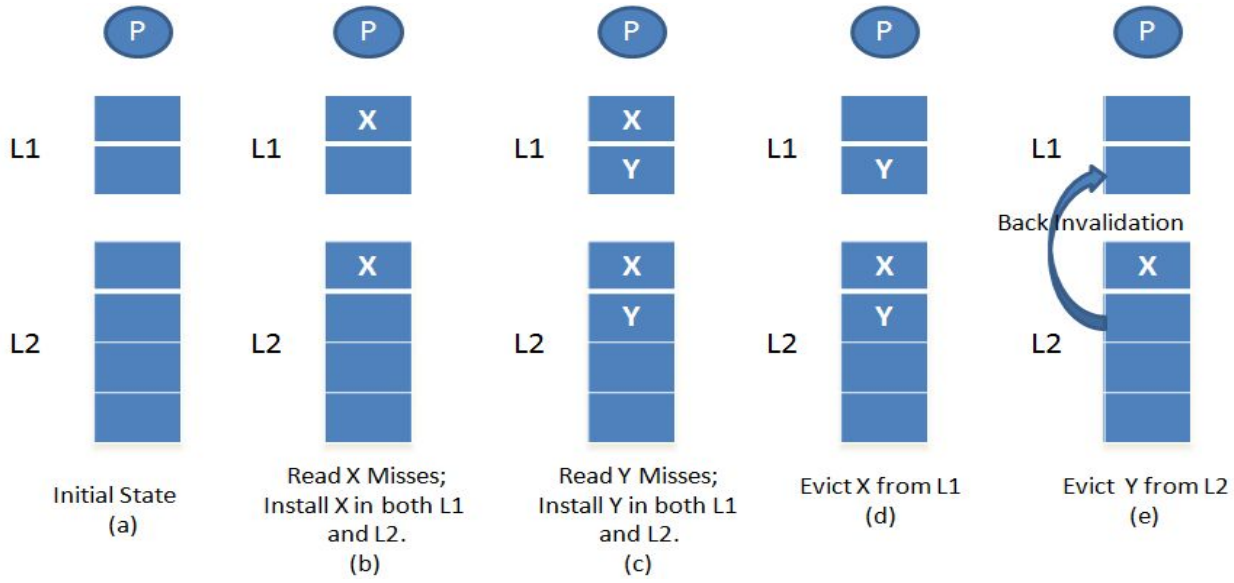


Figure 3: Illustration of Inclusive cache with an example

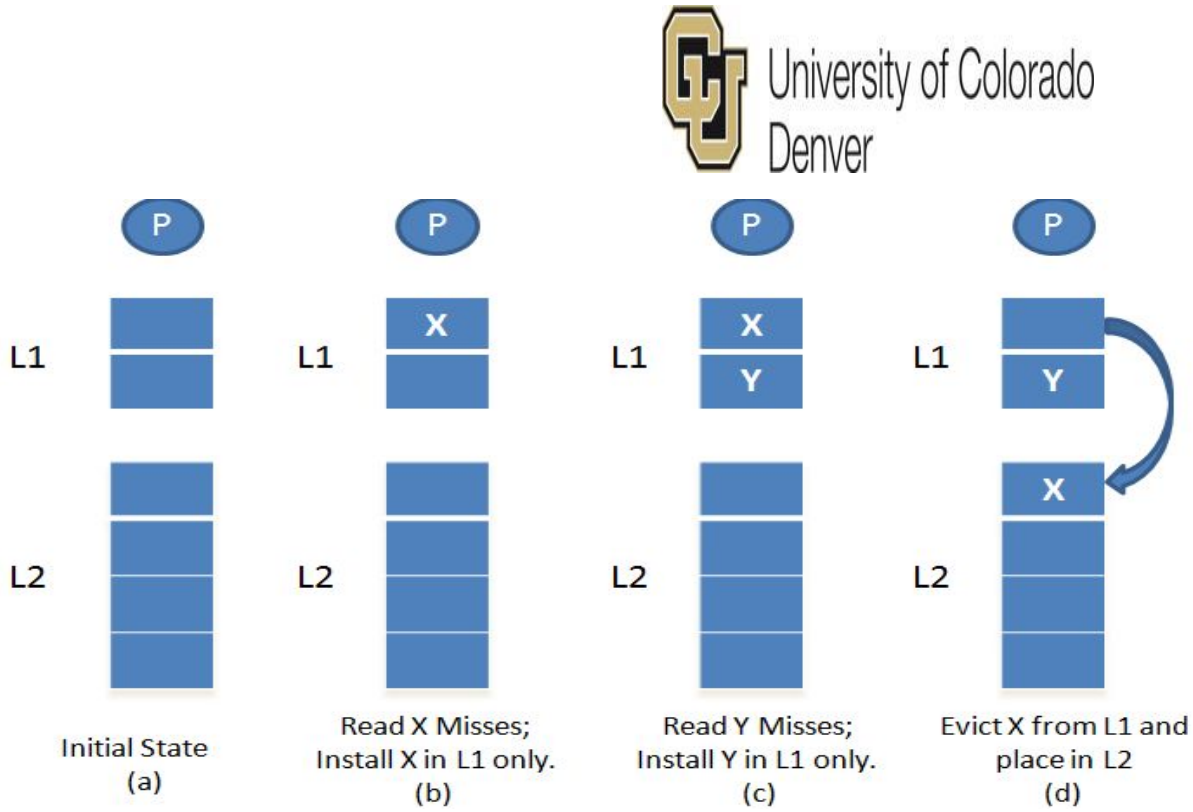


Figure 4: Illustration of Exclusive cache with an example

EXPERIMENTAL ENVIRONMENT

Performance evaluation of different cache replacement policies has been done using the GEM5 and the source code written in python. The benchmarks have been taken from University of Virginia that explores a range of demands on the simulated processor with varying workloads.

Below is the link to the benchmark programs:

https://www.cs.virginia.edu/~cr4bd/6354/F2016/homework2.html#part-c-effective-cache-miss-penalty-versus-miss-latency-required-for-checkpoint

GEM5 Simulator

The gem5 simulator is a modular platform for computer-system architecture research, encompassing system-level architecture as well as processor micro architecture. GEM 5 provides multiple interchangeable CPU models. The CPU models can be combined in arbitrary topologies, creating homogeneous and heterogeneous multi-core systems. GEM 5 decouples ISA semantics from its CPU models, enabling effective support of multiple ISAs. Currently GEM 5 supports Alpha, ARM, SPARC, MIPS, POWER, RISC-V and x86 architectures. We have chosen X86 architecture for our project.

Benchmarks

We have selected several benchmark programs that should explore a range of demands on the simulated processor.

BFS - computes a breadth-first-search problem. This program was selected because it should have poor data cache locality.

Sha - computes the SHA-1 cryptographic hash of its input. The program was selected because it should be integer operation intensive and very friendly for branch prediction and cache.

N-Queens - solves the N-Queen problem for an n specified as an argument. Queens program was selected because it should be very friendly to the cache, but very challenging for branch prediction.

Blocked-matmul - is a 2X2 register-blocked matrix multiplication of two 84X84 matrices. The matrices are pseudo randomly generated and all sizes are hard coded. This program was selected

because it should have a mix of cache accesses and floating-point operations.

Measurements

In this project we measure L1 Cache Overall Miss Rates and Program execution time to analysis the performance of the new configuration.

All stats are captured in stats.txt in m5src folder. In order calculate the overall_miss_rate we apply below formula.

$$\textbf{L1\_Overall\_Miss\_Rates} = \frac{\textbf{(total misses in d-cache + total misses in i-cache)}}{\textbf{(total accesses in d-cache + total accesses in i-cache)}}$$

Program_Execution = (time taken in seconds to run the Benchmark Programs)

Note: We observe that L2 & L3 showed high overall miss rate and reason being, L2 & L3 have cold misses and the total number of access during the execution of the program is very less compared to L1. So the ratio of (total_misses / total_access) is always close to one. And hence, we took L1 miss rate that being major contributor to the performance and also execution time.

Base Configuration

RAM Size	2048 MB
Memory Mode	Timing
CPU Type	Timing Simple CPU
L2 Memory Bus	Coherent Cross Bar

L3 Memory Bus	Coherent Cross Bar
System Domain Clock	3GHz

Table 1: Base configurations on GEM5

Configurable Configuration

Name	**L1_policy**	**L2_policy**	**L3_policy**	**L2_size**	**L3_size**	**L1_assoc**	**L2_assoc**	**L3_assoc**	**Cache_type**
fifo	fifo	fifo	fifo	256kb	512kb	2	8	8	inclusive
lru	lru	lru	lru	256kb	512kb	2	8	8	inclusive
rand	rand	rand	rand	256kb	512kb	2	8	8	inclusive
fifo_rand	fifo	rand	lru	256kb	512kb	2	8	8	inclusive
lru_rand	lru	rand	fifo	256kb	512kb	2	8	8	inclusive
fifo_lru	fifo	lru	rand	256kb	512kb	2	8	8	inclusive
rand_lru	rand	lru	fifo	256kb	512kb	2	8	8	inclusive
rand_fifo	rand	fifo	lru	256kb	512kb	2	8	8	inclusive
lru_fifo	lru	fifo	rand	256kb	512kb	2	8	8	inclusive
lru_inclu	lru	lru	lru	512kb	256kb	2	8	8	inclusive

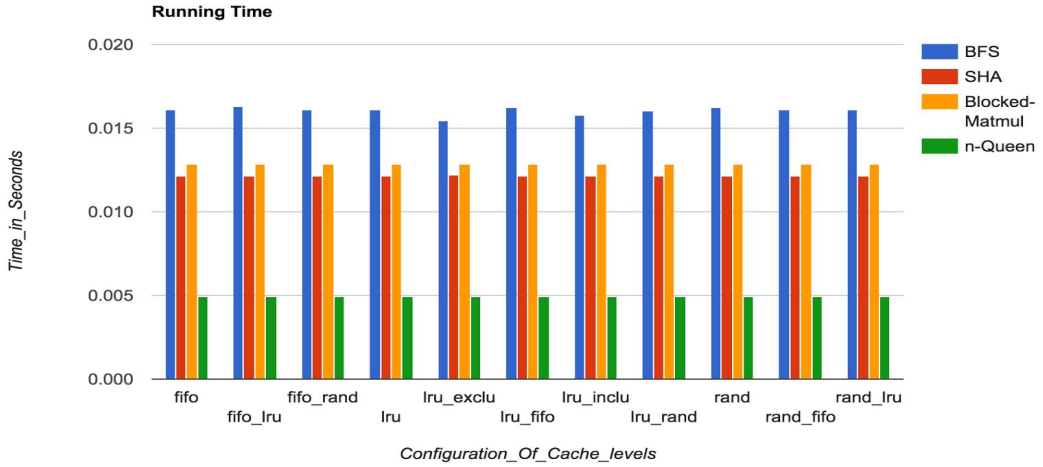
lru_exclu	lru	lru	lru	512kb	256kb	2	8	8	exclusive
2	lru	lru	lru	256kb	512kb	2	2	2	inclusive
4	lru	lru	lru	256kb	512kb	4	4	4	inclusive
8	lru	lru	lru	256kb	512kb	8	8	8	inclusive
248	lru	lru	lru	256kb	512kb	2	4	8	inclusive
842	lru	lru	lru	256kb	512kb	8	4	2	inclusive

Table 2: Configurable Configurations on GEM5

EXPERIMENTAL RESULTS

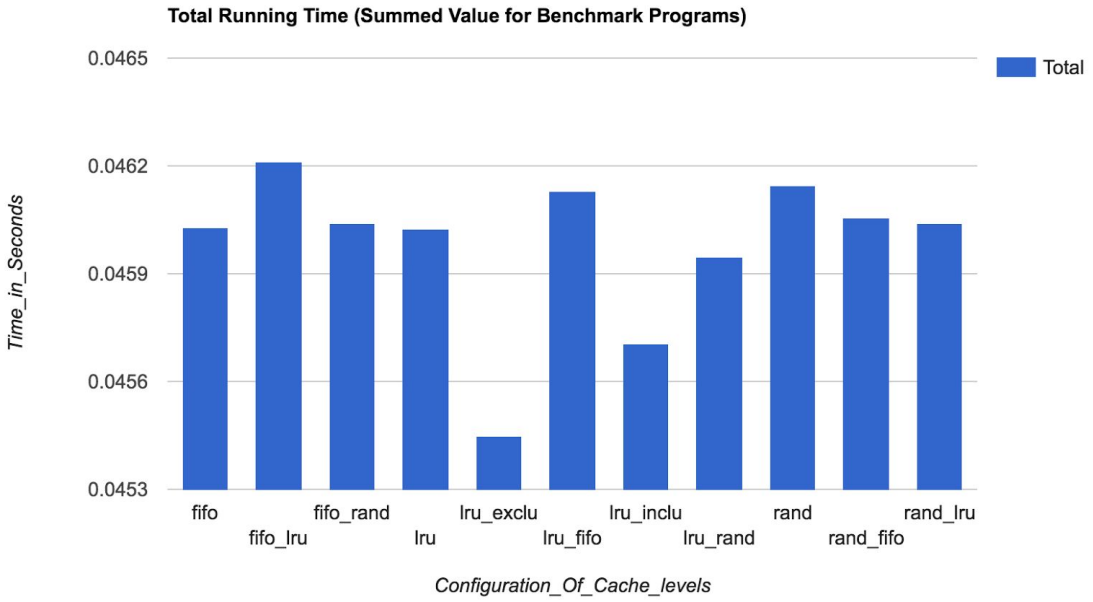
The x-axis parameters in the below graphs are the names indicated in the first column of Table 2. We have created a parser in python to generate the graphs to be plotted.

Graph 1: The below chart captures the total running time for different cache configuration with varying workload



The above grouped bar chart shows the execution of four benchmark programs for different configuration. It's clear that all configuration shown above have similar pattern for the execution time. The highest time taken is for BFS and lowest one is n-Queen (we have taken n=10 for the experiment). But it's not clear to conclude on which configuration performs better. Hence, we have consolidated the program execution time and generated a new bar chart below. **The detail about each configuration is present in the Table 2.**

Graph 2: The below chart capture the consolidated running time for different cache configuration.



The above consolidated bar chart shows total execution time for the benchmark programs for given configuration. It's clear that lru_exclu configuration performs the best. The fifo_lru configuration has the highest execution time.

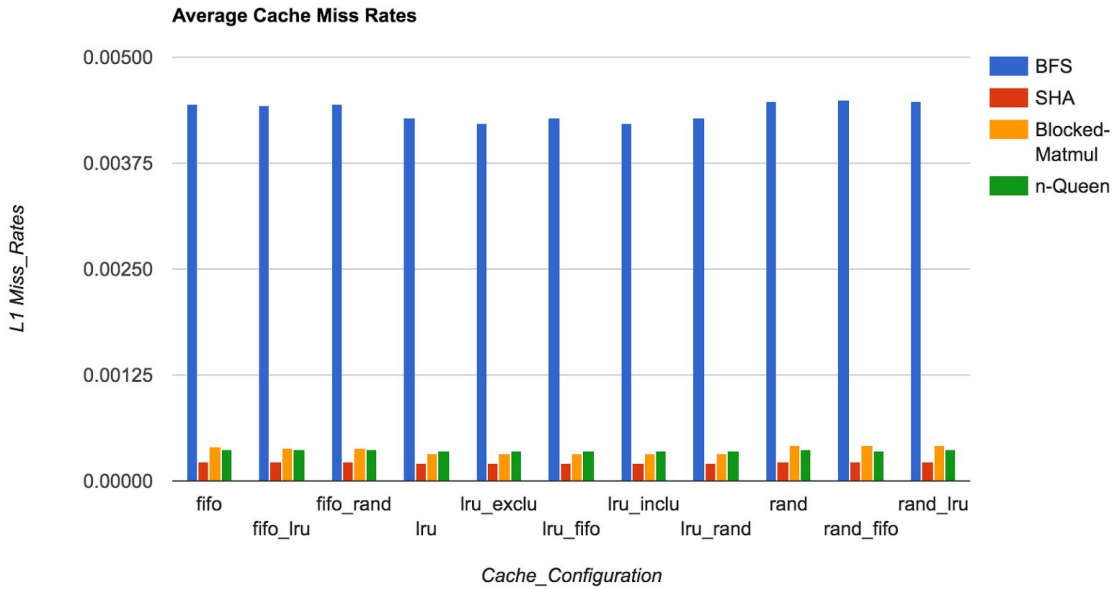
Same policy across levels - In comparison among (lru, fifo and rand) we observe that lru wins by slight margin in terms of execution time. To our surprise, we observed that rand is not the one with the highest execution time.

Different policy across levels - In comparison among (fifo_lru, fifo_rand, lru_fio, lru_rand, rand_fifo, rand_lru) we observe lru_rand has the lowest execution time.

Changing the Memory Hierarchy - In comparison among (lru_exclu, lru_inclu) we find that lru_exclu has the lowest execution time.

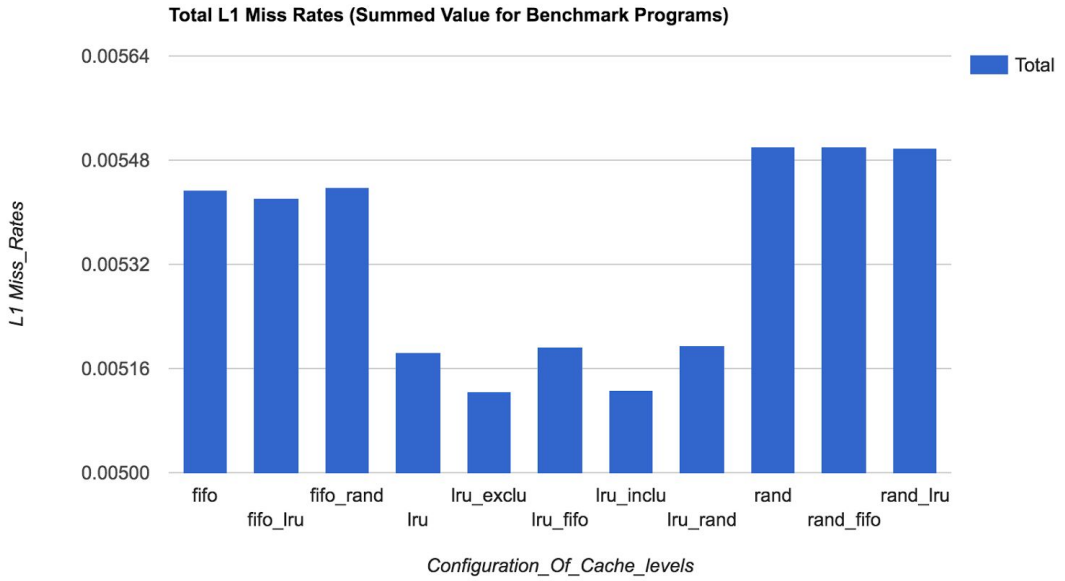
The detail about each configuration is present in the Table 2.

Graph 3: The below chart captures the cache miss rate at L1 for different cache configuration with varying workload.



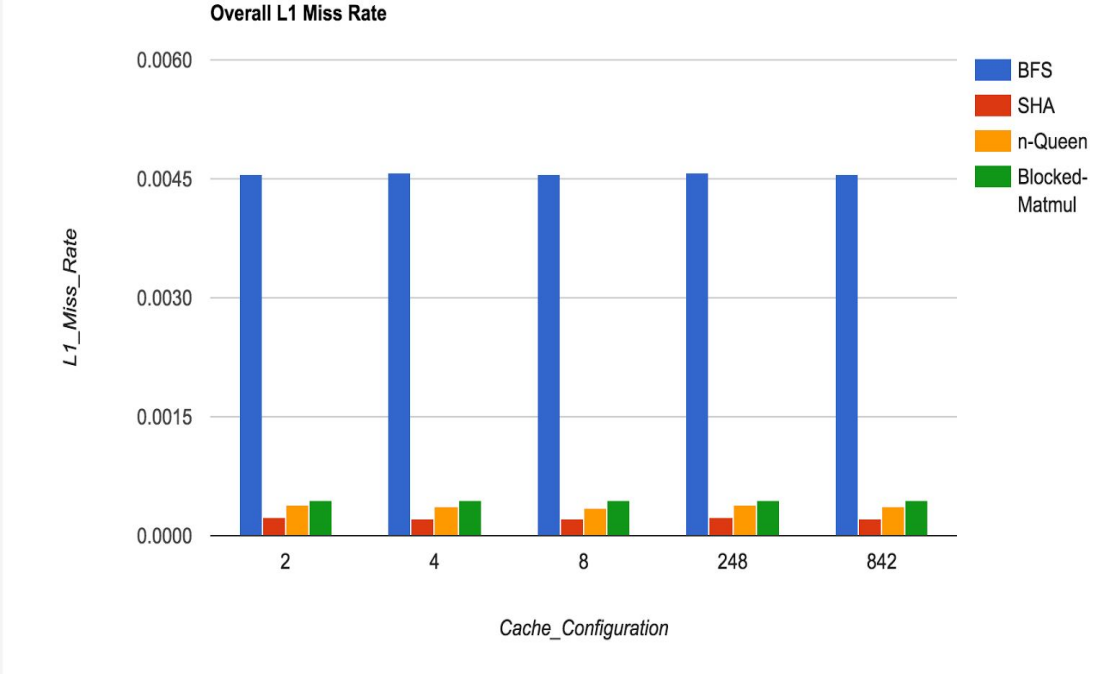
The above grouped bar chart shows the L1 miss rates of four benchmark programs for different configuration. It's clear that all configuration shown above have similar pattern for the miss rates in L1. The highest miss rates is for BFS and lowest one is Sha encryption. But it's not clear, to conclude on which configuration performs better. Hence we have consolidated the L1 miss rates and generated a new bar chart below. To our surprise, it's clear that having less miss rate at L1 doesn't guarantee faster execution. This is clear from Sha program which has less miss rate but its execution time is not the fastest. **The detail about each configuration is present in the Table 2.**

Graph 4: The below chart captures the consolidated cache miss rate at L1 for different cache configuration.



The above consolidated graph shows the L1 miss rates for different cache configuration. Among all configuration we have lru_exclu that performs the best. The second configuration with lesser miss rates being lru_incl and third being lru. In case of (rand, rand_fifo and rand_lru) show similar miss rates but from the Graph 2 we know that rand_lru has the least execution time. Most of configuration in the above graph adheres to execution time. Meaning, having lesser miss rates is directly proportional to faster execution of the program. But, we have other configuration that shows, in spite of having more miss rates the program executes faster. **The detail about each configuration is present in the Table 2.**

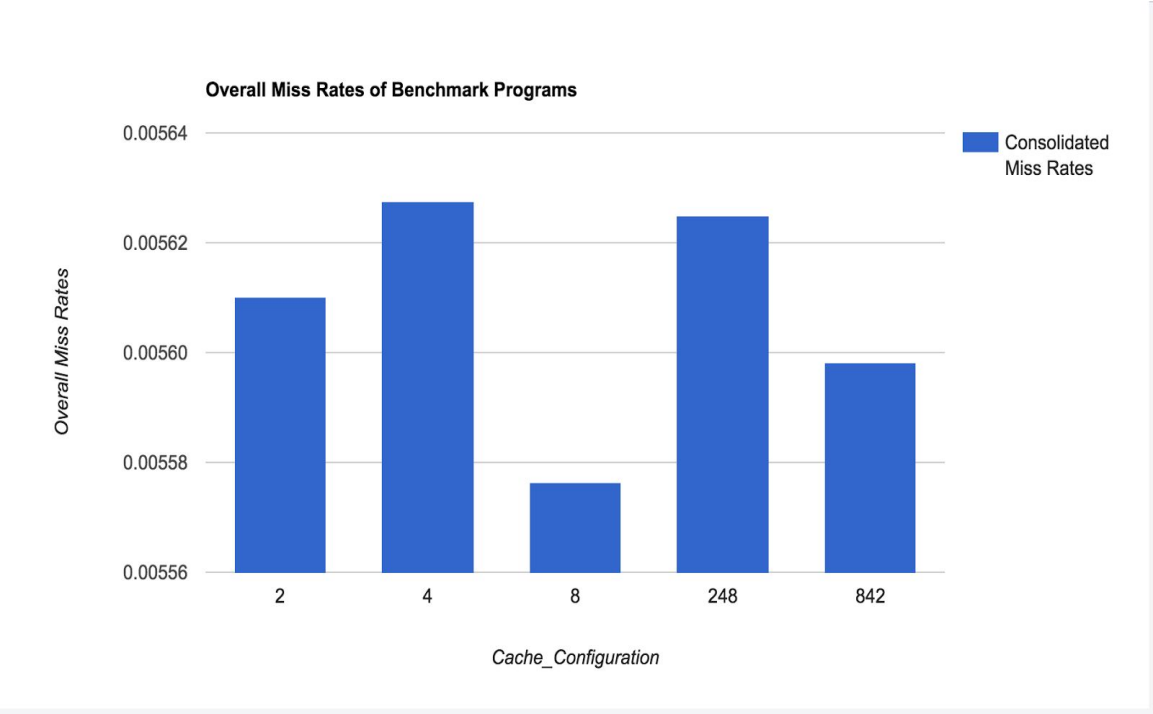
Graph 5: The below chart represents the overall cache miss rate at L1 for different set associativity



The above bar chart represents the overall miss rate at L1 cache for the four benchmarks. Y -axis represents the overall miss rate and X-axis denotes the associativity. 2 represents the set associativity at all the 3 levels of cache.Similarly 4 and 8 represent 4-way set associativity and 8-way set associativity at all the three level of cache respectively. 248 is an indication that 2-way, 4-way and 8-way set associativity are implemented at L1, L2 and L3 caches

respectively and vice versa for 842. It can be seen that all the programs perform similarly for all the different configurations. Much difference cannot be observed. As it is hard to conclude from the above bar graph we have considered the consolidated miss rates in the below graph. **The detail about each configuration is present in the Table 2.**

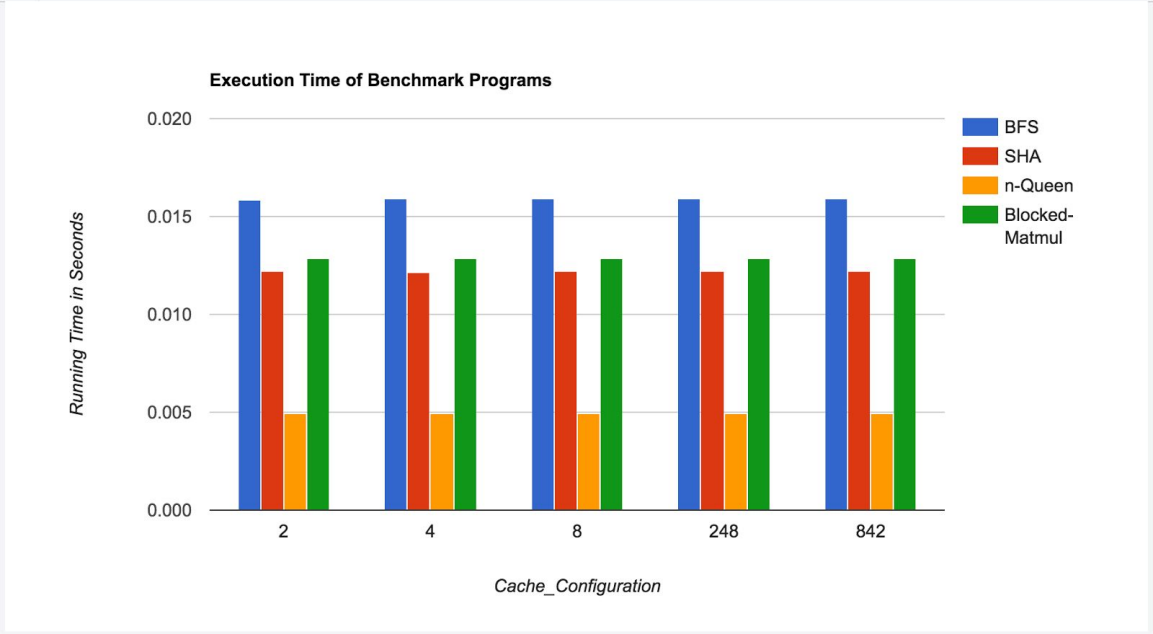
Graph 6: The below chart represents the consolidated Cache Miss Rate for L1 for different set associativity with varying workloads



The above graph represents the consolidated overall miss rates at L1 cache for different configurations. X-axis denotes the configurations we have tested and the details are listed in the above configurable configurations table. Here we can see that 8-way set associativity performs better as it is having the lowest miss rate.

The reason for this is well known as having higher set associativity at lower levels of cache always favours better performance. **The detail about each configuration is present in the Table 2.**

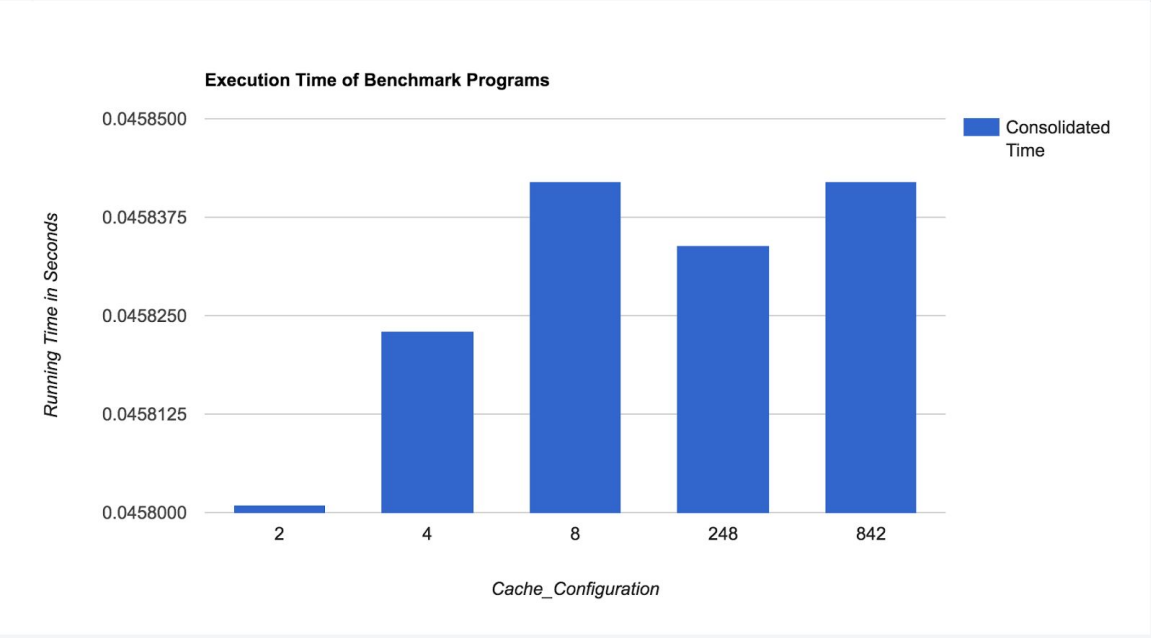
Graph 7: The below chart represents the CPU Time for different set associativity with varying workloads



The above graph represents the individual CPU Time for all the different benchmarks at L1 cache. The x-axis represents the different configurations of set associativity at L1, L2 and L3 levels of cache (details listed in the above configurable configurations table). The y-axis represents the CPU Time in seconds. We can see that all the programs perform at a same rate even when the associativity is changed. Since these parameters does not yield any conclusions to be made, we have taken a consolidated rate and

plotted the graph below. **The detail about each configuration is present in the Table 2.**

Graph 8: The below graph represents the overall consolidated CPU Time for different cache configurations of various workloads



The above graph is the consolidated CPU Times represented in seconds. Here we can see that the programs with set associativity 2 performs better as it is having the lowest CPU Time. But as we observed in Graph 6, even though the miss rates are very low at 8-way set associativity, the CPU does not perform better. This is because of the fact that the CPU Time is affected not only by the miss rates but there are many other factors like miss latency, tag latency etc., as well and hence it decreases the overall CPU

performance by giving a high CPU Time. **The detail about each configuration is present in the Table 2.**

CONCLUSION AND RECOMMENDATIONS

This project helped us to learn and understand the cache hierarchy well. Let us re-iterate the research topics and conclude with the results we observed while running the experiments.

Swapping Memory Hierarchy

Conclusion are made from above graph 2 & 4
The configuration of lru_inclu & lru_exclu is where cache levels 2 & 3 are swapped.
More about the configuration can be found in Table 1 & 2. From all the above configuration, we found out that changing the memory hierarchy provide better performance and lesser miss rates. The lru_exclu in which we have the hierarchy swapped as well as the level 2 & 3 are made exclusive has the lowest running time and lesser miss rate.

Conclusion 1: Swapping Cache Level 2 & 3 yields better performance.
Conclusion 2: Making Level 2 & 3 cache exclusive and level swapped shows the best performance among all configurations.

Cache Replacement Policy

Conclusion are made from above graph 2 & 4

The configuration of lru, rand, fifo, lru_fifo, fifo_lru, rand_fifo, rand_lru, fifo_rand and lru_rand is where the cache replacement policy are swapped. More about the configuration can be found in Table 1 & 2. To our surprise, rand did pretty decent and fifo_rand had the worst execution time. So to conclude, having different cache replacement policy doesn't provide any gain in performance.

Conclusion 3: Having different cache replacement policy doesn't provide any gain in performance.
Conclusion 4: Lesser Miss rates provides better performance but the inverse is not true.

Associativity

Conclusions made with the results based on Graph 6 & 8
As we observed in Graph 6, even though the miss rates are very low at 8-way set associativity, the CPU does not perform better. This is because of the fact that the CPU Time is affected not only by the miss rates but there are many other factors like miss latency, tag latency etc., as well and hence it decreases the overall CPU performance by giving a high CPU Time.

Conclusion 5: Having larger set associativity value yields lesser miss rate but that does not guarantee an increase CPU performance.

REFERENCES

- http://www.cs.ucf.edu/~neslisah/final.pdf
- http://www.cse.scu.edu/~mwang2/projects/Cache_replacement_10s.pdf
- https://ece752.ece.wisc.edu/lect11-cache-replacement.pdf
- https://en.wikipedia.org/wiki/Cache_replacement_policies
- http://www.ece.uah.edu/~milenka/docs/milenkovic_acmse04r.pdf
- http://people.csail.mit.edu/emer/papers/2010.06.isca.rrip.pdf
- http://www.cs.utexas.edu/users/mckinley/papers/evict-me-pact-2002.pdf
- http://snir.cs.illinois.edu/PDF/Temporal%20and%20Spatial%20Locality.pdf
- https://math.mit.edu/~stevenj/18.335/ideal-cache.pdf
- https://pdfs.semanticscholar.org/6ebe/c8701893a6770eb0e19a0d4a732852c86256.pdf
- https://fD3hhNnfL6kwww.youtube.com/watch?v=
- http://pages.cs.wisc.edu/~david/courses/cs752/Spring2015/gem5-tutorial/index.html
- http://blog.stuffedcow.net/2013/01/ivb-cache-replacement/
- https://github.com/dependablecomputinglab/csi3102-gem5-new-cache-policy