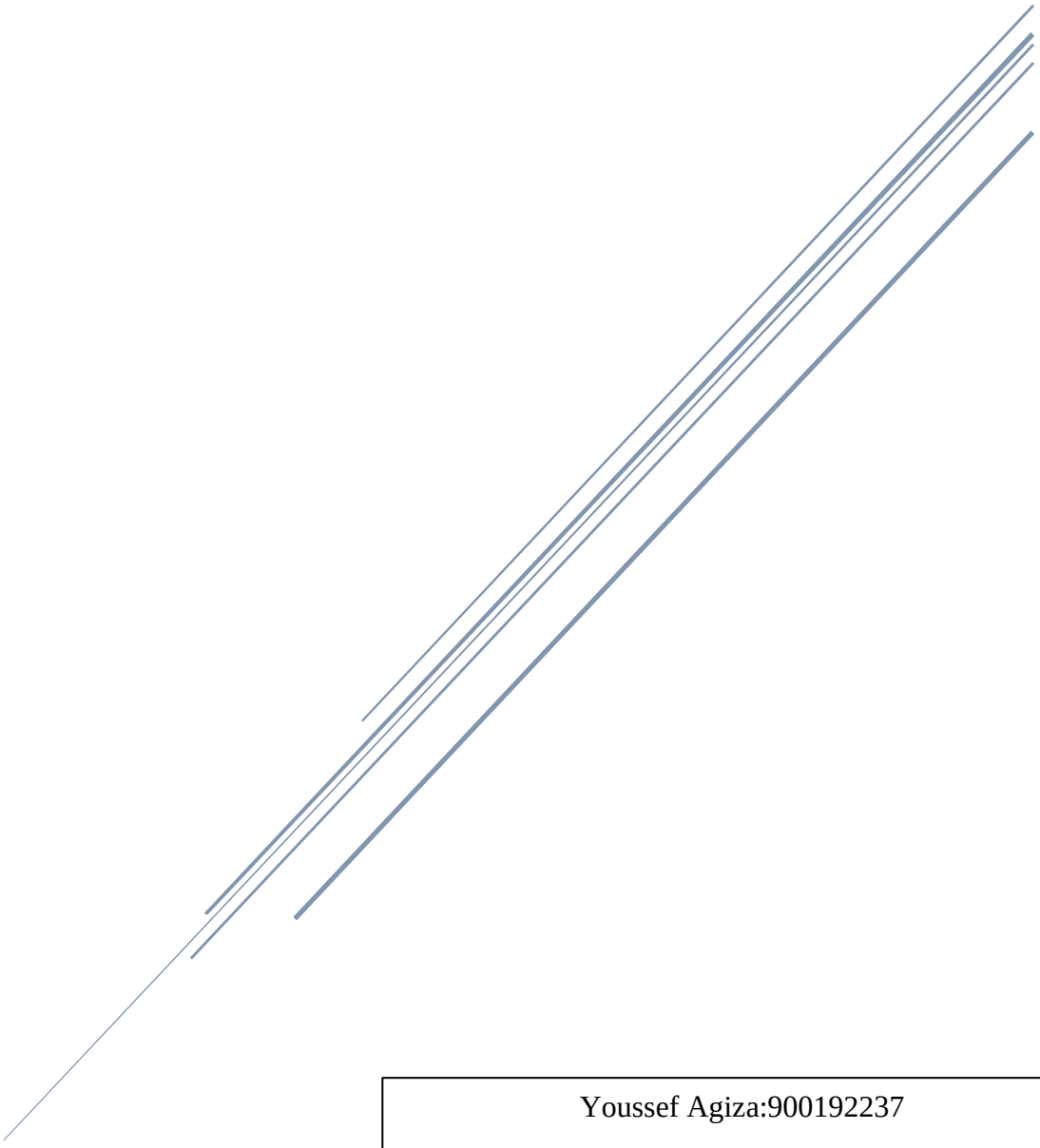
S.A. CACHE ANALYSIS

Cache Performance

GitHub Repo:
https://github.com/Youssef-Agiza/Cache-Simulator



Youssef Agiza:900192237

Seif Y. Sallam: 900193668

Kareem A. Mohammed Talaat: 900192903

Brief Description

A cache simulator that uses different replacement policies(Random, LRU, LFU). The project is made with a main class that represents a Set Associative Cache and a set of several functions that assists in testing the cache and doing the simulations (Experiments).

Tests:

We designed three functions that produce three distinct patterns to test different features of the cache.

Test 1:

Objective: Setting the n-set associative cache to 1 way, and check if it behaves like a direct cache as it should be by referencing two memory addresses of the same index and different tag in order to get a 100 % miss rate.

Parameters:

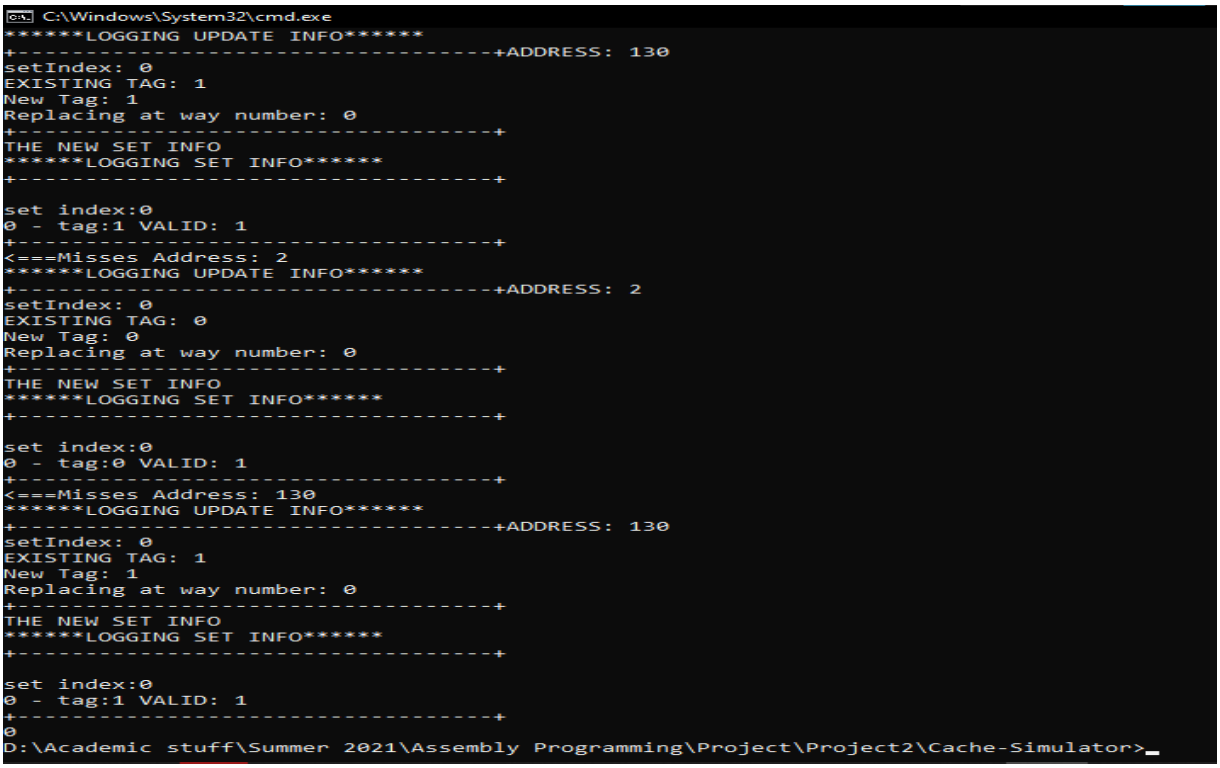
- Number of ways: 1 way.
- Line size: 4 Bytes.
- Replacement Policy: LRU (Least Recently Used).
- Cache Size: 128 Bytes.
- Number of iterations of the test: 100

Expected output: 0% hit rate.

Output: 0% hit rate.

Behavior: addresses two values that have the same set and byte offset but different tags (The cache should replace the current line with the new line).

Photo:



Test 2:

Objective: Tests whether the cache responds to adding different addresses of the same set but with different tags to the same set (way-storing test).

Method: use a cache with one set, loop 100 times and increment the tag of address given to the cache by 1 in every iteration while fixing the index.

Cache Parameters:

- Number of ways: 4 ways.
- Line size: 8 Bytes.
- Replacement Policy: LRU (Least Recently Used).
- Cache Size: 128 Bytes.
- Number of iterations of the test: 100

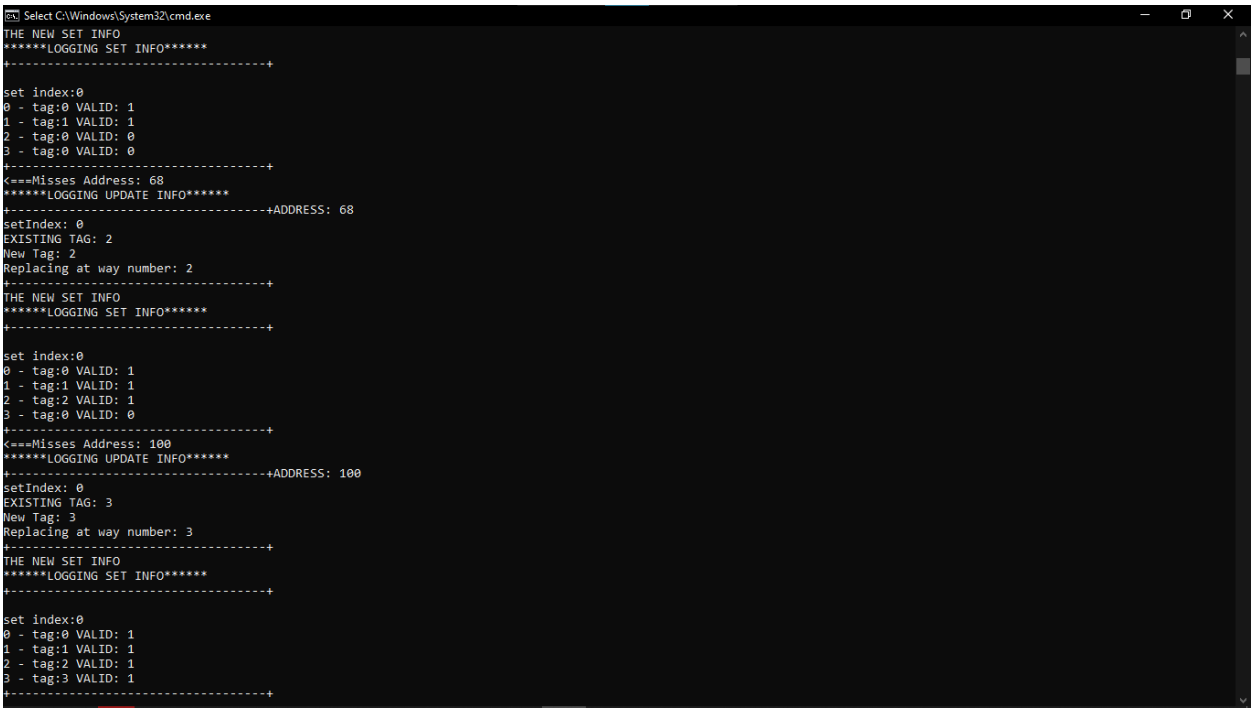
Expected output: 0% hit rate.

Output: 0% hit rate.

Behavior: Addresses several values that have the same set number but different tags, to make sure that the cache fills up the other ways (lines) when there is a **MISS** and it chooses the lines according to the least recently used way.

Photo:

You can notice in the photo how the same set keeps adding the different addresses with different tags and changing the valid bits once we end up with a MISS.



Test 3:

Objective: The first practical test that assesses the whole cache functionality. It is extracted from an online example found at this link: https://www.youtube.com/watch?v=quZe1ehz-EQ&t=201s.

Cache Parameters:

- Number of ways: 2 ways.
- Line size: 32 Bytes.
- Replacement Policy: LRU (Least Recently Used).
- Cache Size: 128 Bytes.
- Number of iterations of the test: 18

Expected output: 44% in the 9 Iterations, and 77% hit rate in the second 9 Iterations.

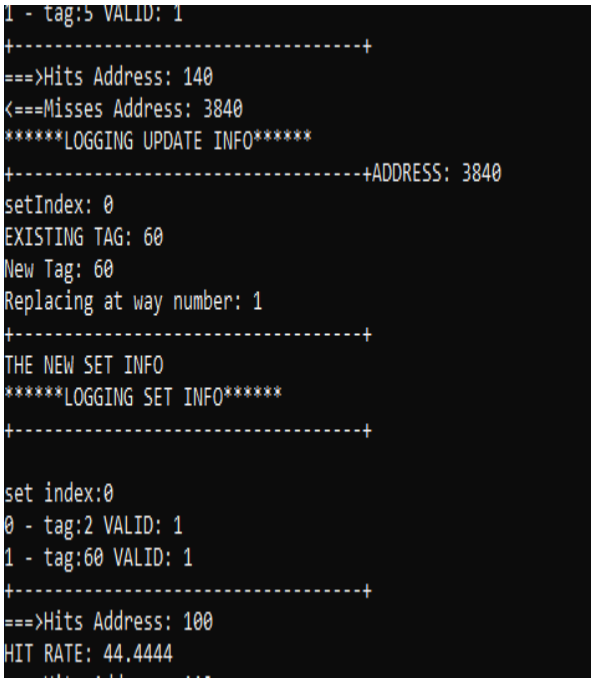
Output: 44% in the 9 Iterations, and 77% hit rate in the second 9 Iterations.

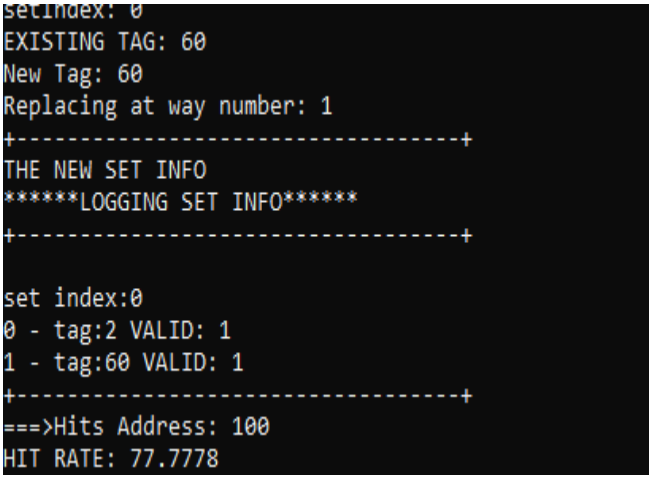
Behavior: Different addresses with different tags and sets. It is considered a mini practical example of how the cache might be addressed.

Photo:

On the left: The first 9 iterations.

On the right: The second 9 iterations.





Address Generators:

Memory reference generators are used in the simulation, and they simulate the memory access behavior of different applications. The generators are implemented by the functions: memGenA(), memGenB(), memGenC(), memGenD(), memGenE() and memGenF(). They are used during the simulation to generate 1 million memory references to be used in each experiment, and measure the hit and miss ratios.

Behaviors of each memGen:

memGenA:

It generates addresses in a sequential manner (0, 1 ,2 ,3 ,4, ….) not exceeding the memory size.

memGenB:

It generates random addresses between 0 and (64 * 1024).

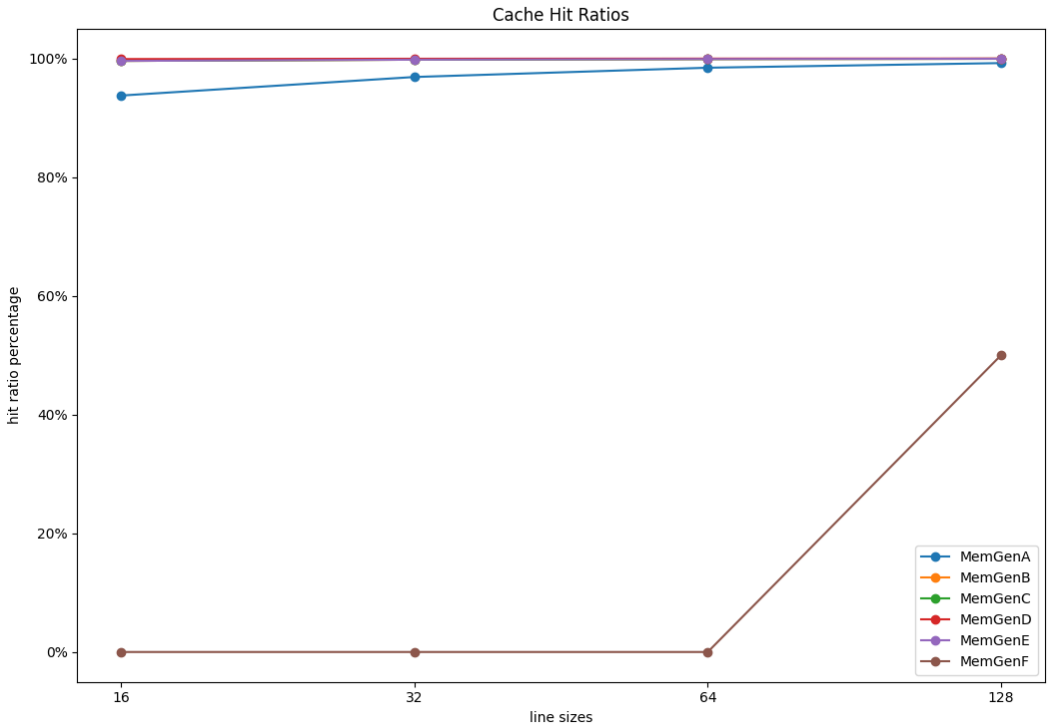
memGenC:

It uses two variables one that sequentially increases to 512 (a0) then resets, and the other sequentially increases to 128 (a1) every 512 addresses, the generated address is by the linear equation: a1 + a0 * 128.
This translates to addresses: (128, 256, ….., 65536, 129, 257, ...).
memGenD:
It generates random addresses between 0 and (16 * 1024).
memGenE:
It generates addresses in a sequential manner (0, 1 ,2 ,3 ,4, ….) not exceeding (64 * 1024).
memGenF:
It generates addresses in intervals of 64 up until the value (4 * 64 * 1024).
Sample addresses: (64, 128, 192, 256, ….)

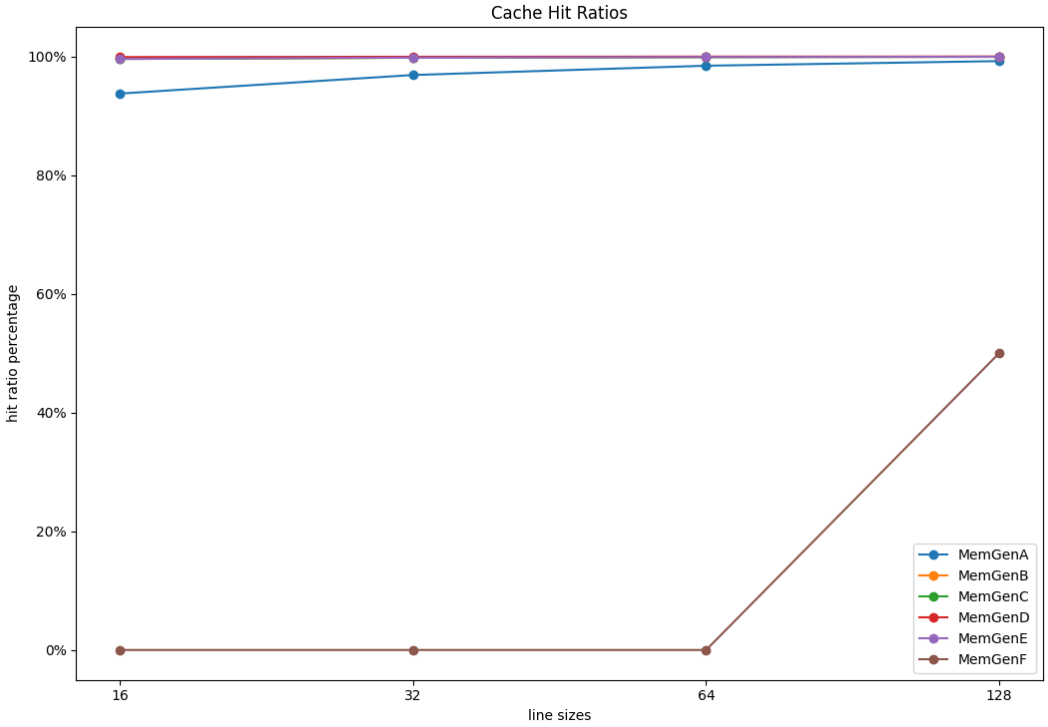
Experiment Analysis

Experiment #1

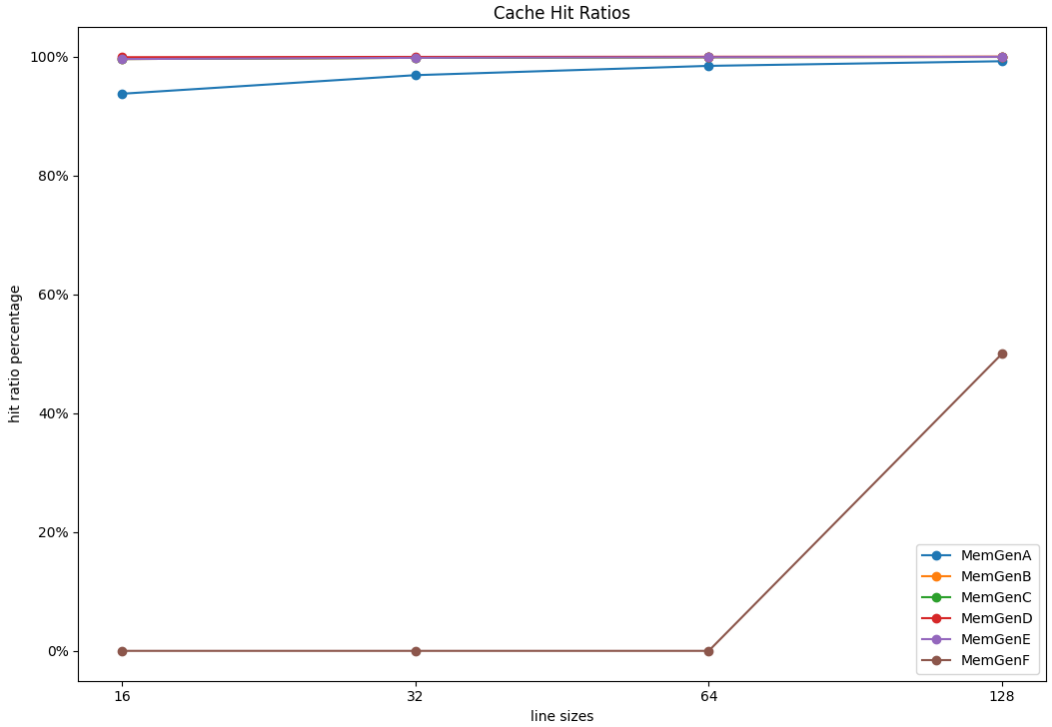
We tested the different memory generators on the 1 way set associative cache that is simulating a direct map design.



1 Way - LFU



1 Way LRU



1 Way Random

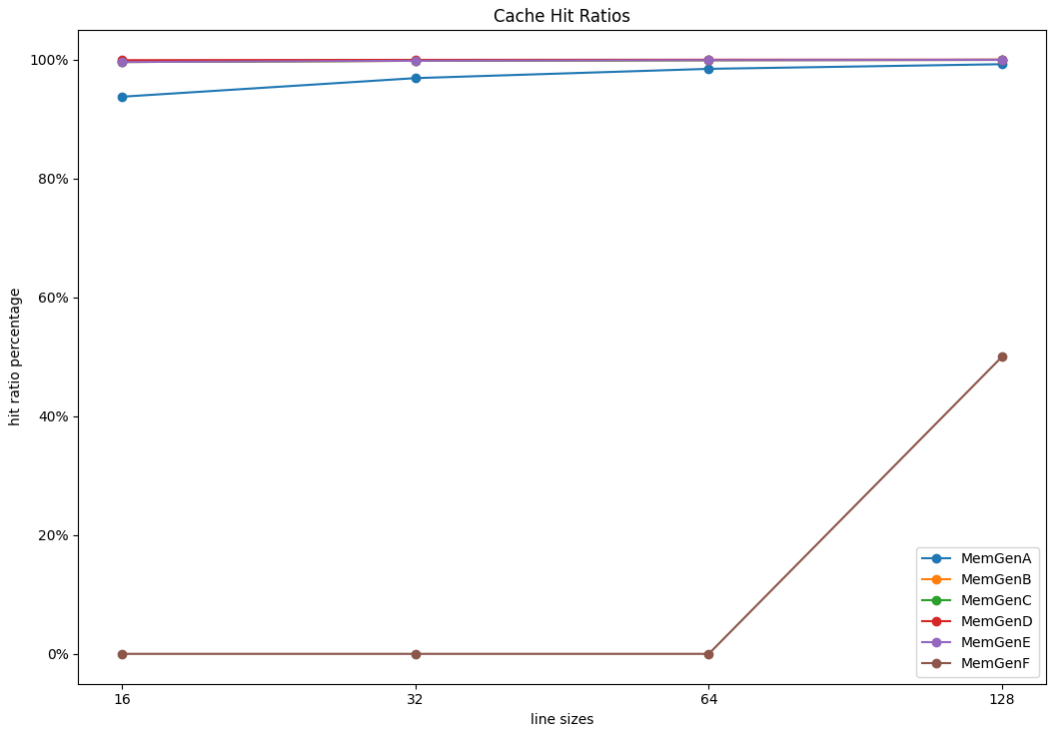
We have tested the same generators on different replacement policies (Random, LFU, and LRU). They have all produced very similar results in terms of Hit Ratios.

According to the graphs With increasing the line size, the hit ratio is increasing, and this is almost independent of the program using it, since we have 5 different functions (excluding F) that simulate 5 different programs and they all have the same hit ratio.

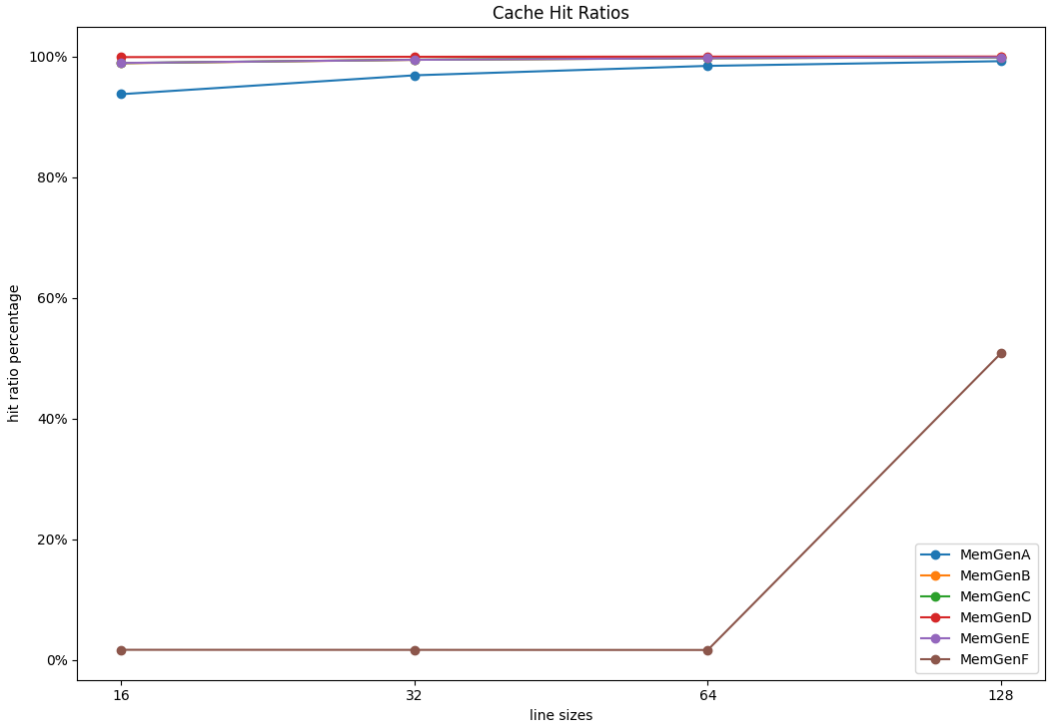
The exceptional behaviour of the F memory generator:

For the 1 way set associative, because the generator F generates the addresses in intervals of 64, the miss rate will be always 100% unless the line size is 128.
Since the cache is simulating a Direct map (it has only 1 way), the one set line will contain only one tag and 128 bytes (line size), therefore, hits only generate because of this pattern:
First Address: 64 (Tag = 0, Index: 0)
Second Address: 128 (Tag = 0, Index: 1)
Third Address: 192 (Tag = 0, Index: 1) // HIT
and it keeps repeating every 2 iterations that we encounter ONE miss and ONE hit.That explains the 50% hit ratio.

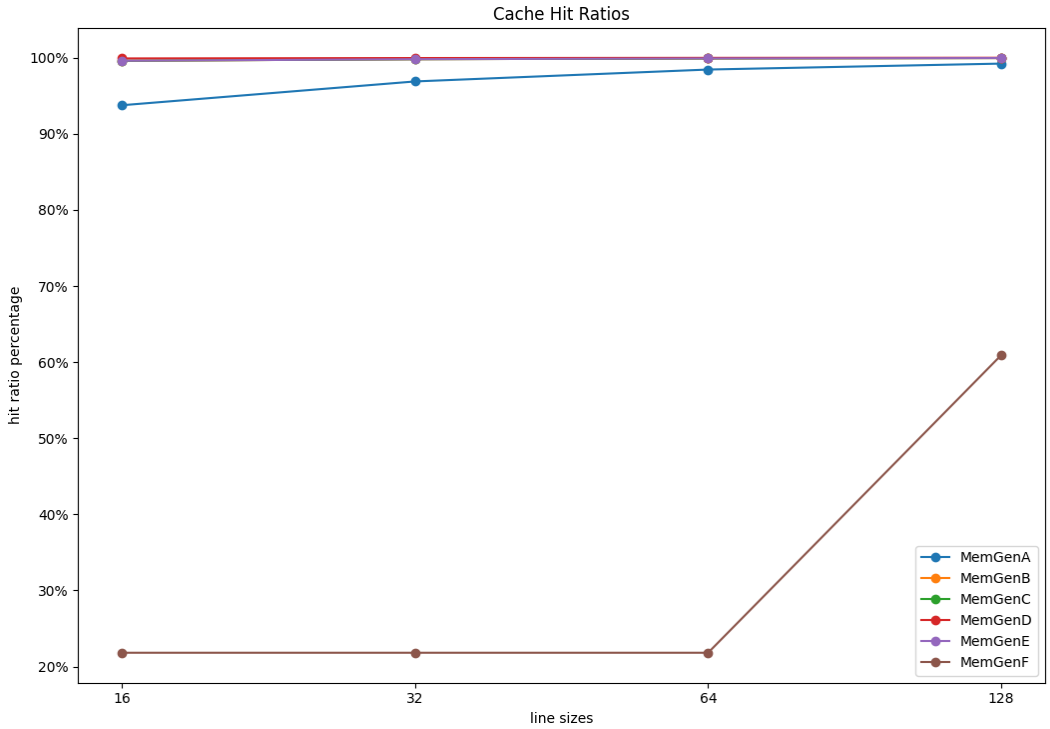
We tested the different memory generators on the 8 way set associative cache that is simulating a direct map design.



8-ways LRU



8 way Random



8-ways LFU

We have tested the same generators on different replacement policies (Random, LFU, and LRU). They have all produced very similar results in terms of Hit Ratios.

According to the graphs, with increasing the line size, the hit ratio is increasing, and this is almost independent of the program using it, since we have 5 different functions (excluding F) that simulate 5 different programs and they all have the same hit ratio.
This behavior is very similar to the 1-way set associative cache.

The exceptional behavior of the F memory generator:

Replacement policy **LRU:**

This policy gives 0% hit ratio because of how the memory generation works, and the least recently used way changes. For example: If we have a cache line size of 16 Bytes, and we have 8 way set associative cache, the addresses will start to repeat going back to the same set index every 2048 iterations, but because of the behavior from the replacement algorithm, every way line is replaced at a time which produces all misses, and once we loop back to the beginning, we then still get more misses since the tags are different. It gives 50% in 128 Bytes, because the one line has two sequential addresses given by the memory generator, so it alternates between a Miss and a Hit.

Replacement policy **Random:**

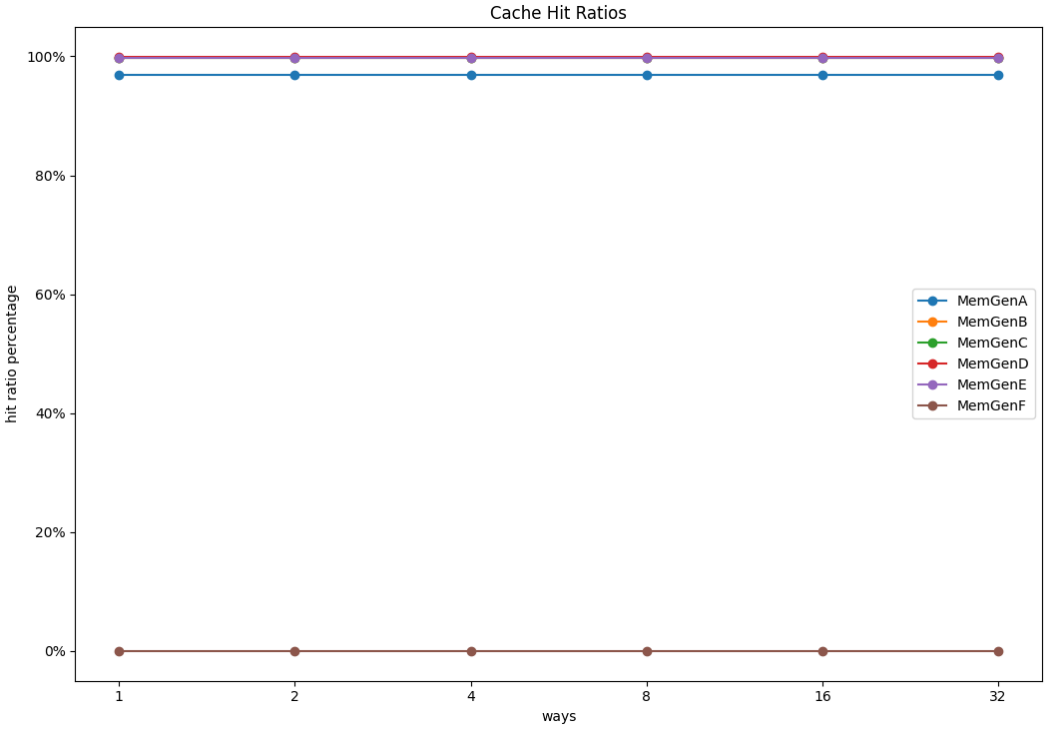
This policy gives random hit ratios that are less than 2% constantly for all the line sizes of less than 128. This is due to the fact that some bytes (and therefore some ways in the set) are not replaced throughout the memory generator cycles. The reason for the spike in 128 Bytes line size, is still because of the two sequential addresses given by the memory generator that alternates between a Miss and a Hit.

Replacement policy **LFU**:

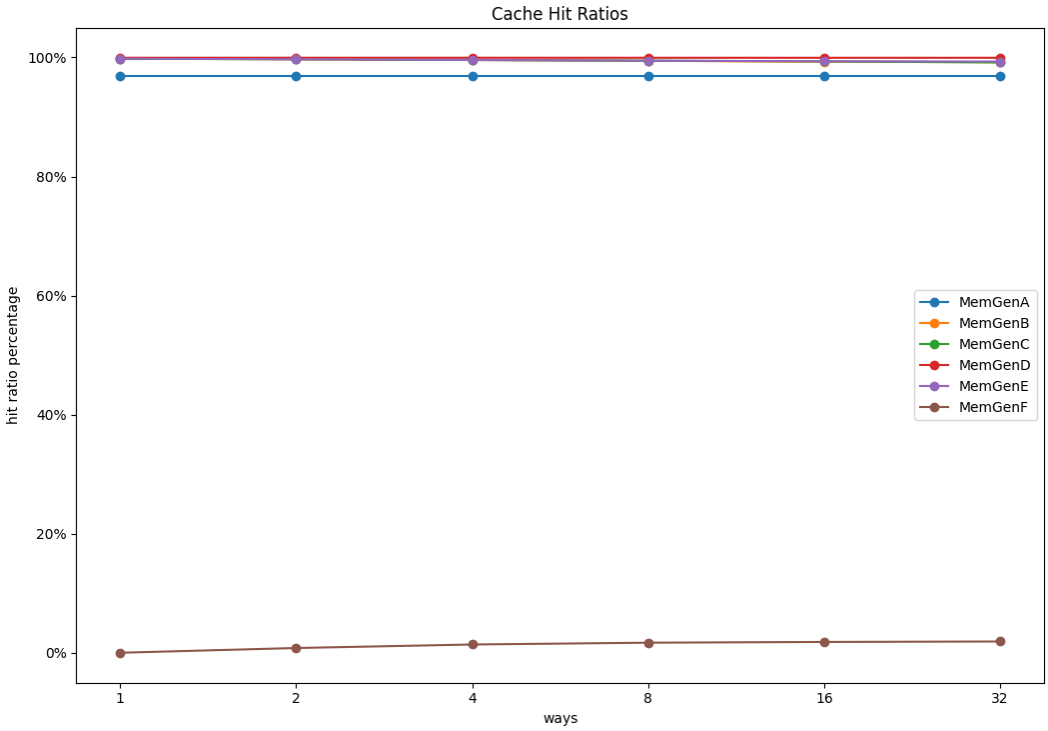
The policy gives more than 20% and the biggest hit ratio because for the same reason as LRU, except the behavior of LFU does the replacement only in ONE way (the 0th way), and then in the next cycle, we start getting hits. The miss rate is still very big because of how big the number of sets is with the low number of line sizes. It increases dramatically with the 128 line size because of the overhead of the two sequential addresses and the former reason. All of these result in a 60+% hit ratio, which is higher than the other replacement policies.

Experiment #2:

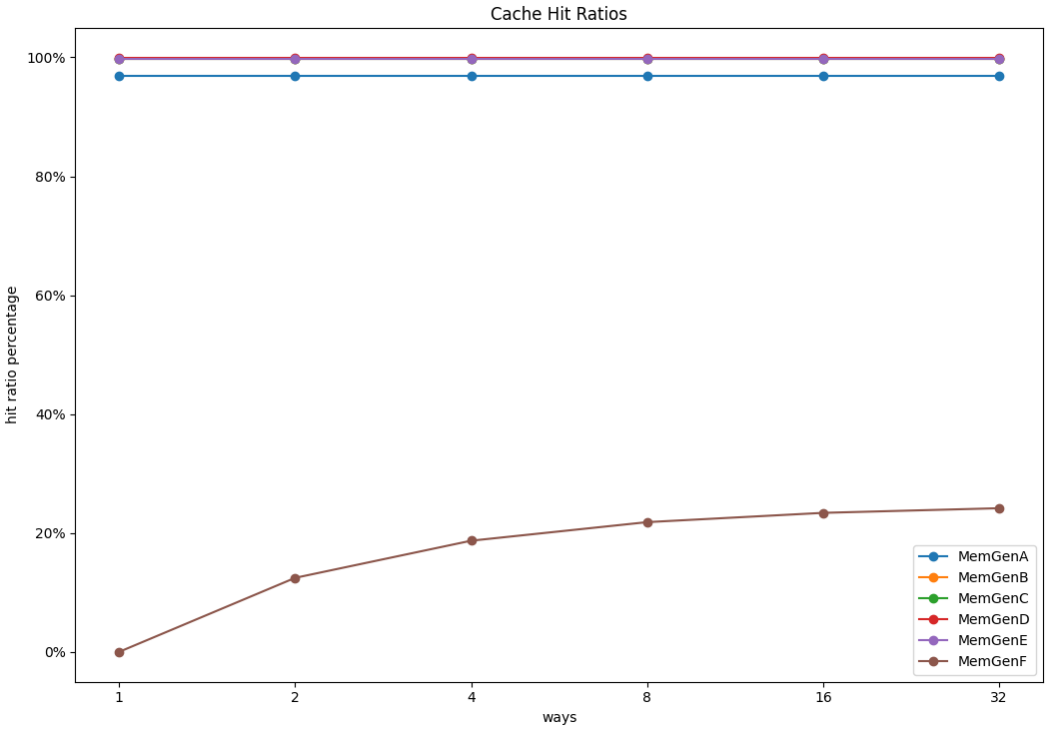
In this experiment we fixed the cache line size to 32 bytes and changed the number of ways the cache operates on (1, 2, 4, 8, 16, and 32). We used all the 6 memory generators again for the experiment from A to F.



Exp2. LRU



Exp2. Random



Exp2. LFU

Replacement policy **LRU:**

With all the memory generators, the hit ratio did not change with the change of the number of ways the cache had, and given the policy of replacement LRU all the available ways get populated equally, that is why in a memory generator like F, we find it always 0 from start to finish because even when we cycle through the addresses, no old addresses stay there since they all change rapidly.

Replacement policy **Random**:

The hit ratio did not change from the Least Recently Used replacement policy, however, we can find some little change in the last memory generator (F) as the random policy can result in choosing specific lines to change and leave others without change until it gets back to it again after going through the whole cycle (addresses > 4*64*1024).

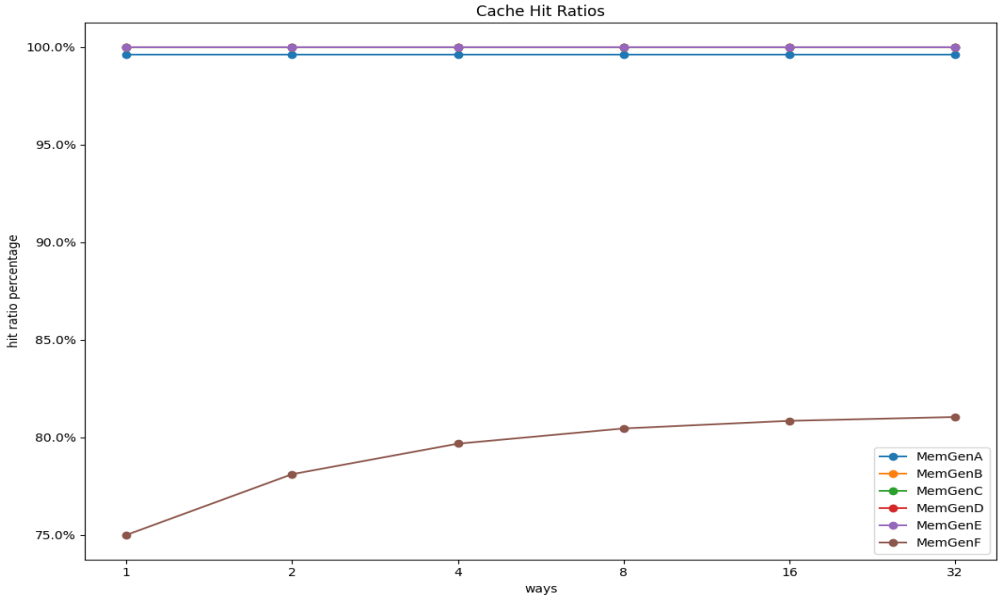
Replacement policy **LFU**:

The hit ratio with most generators was consistent with the change of the number of ways, however for generator F there was a clear increase in the performance of the cache the more ways the cache had. This is mostly because of the policy itself. It stores the old referenced addresses in different ways during the first portion of the cycle, and by then all the ways have the same frequency, and with the next miss, the first way (0th way), is replaced. Once replaced, its frequency gets back to 1, and therefore with any future replacements, it still replaces the 0th way. This results in hits with the next cycles. And the more ways we have in the set, the more hits we will get from the one cycle.

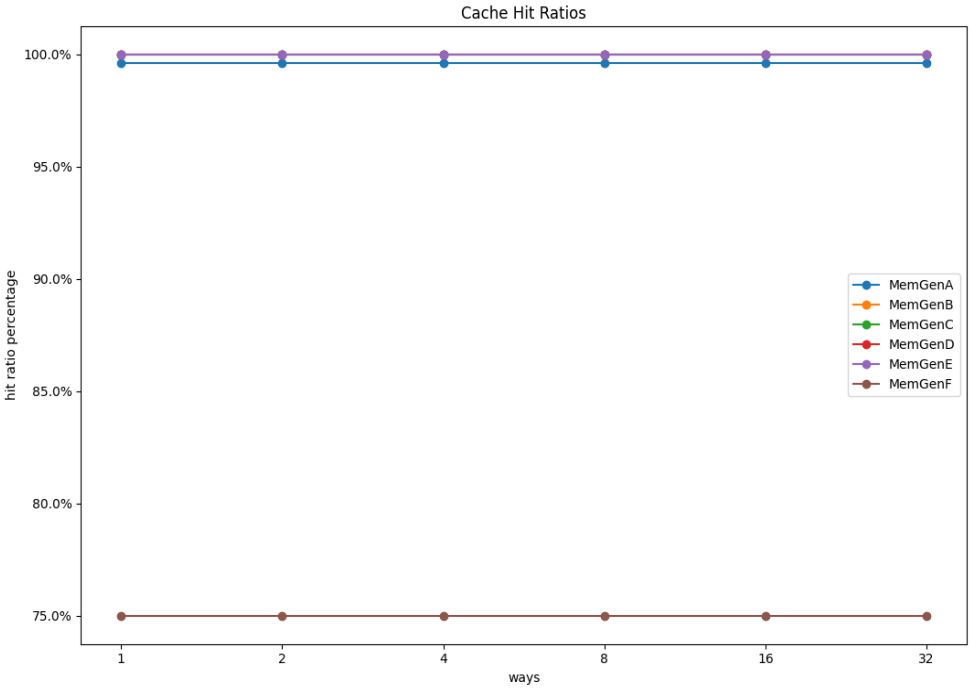
This explains the increase of the hit ratio with the increase of the number of ways.

Experiment #3(EXTRA):

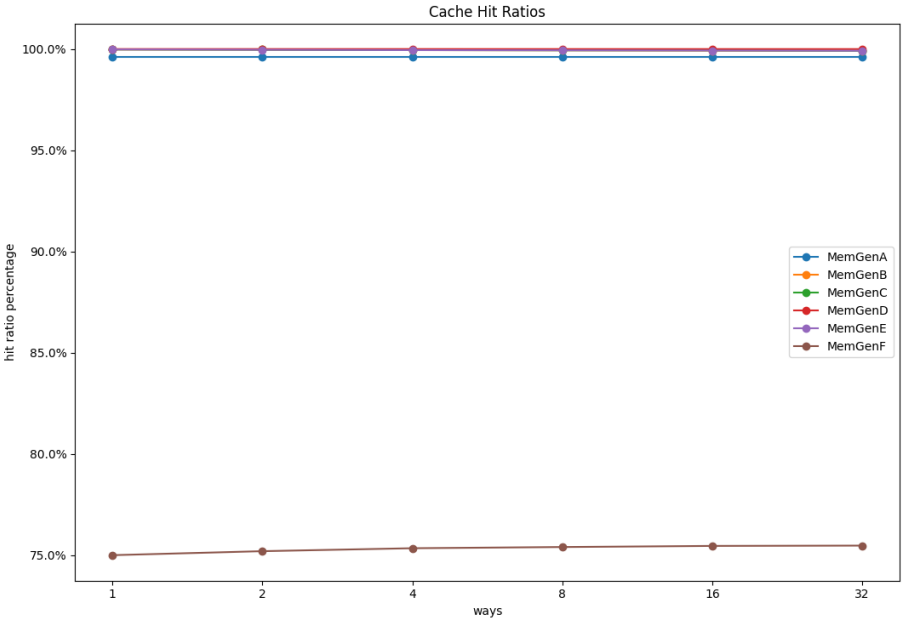
In this experiment, we decided to go a little bit further and test the different replacement policies on the 256 and 512 line sizes caches, and we also experimented with changing the number of ways for each cache (1, 2, 4, 8, 16, and 32 ways). We used all the 6 memory generators again for the experiment from A to F. We wanted to check if the performance will drop after we increased the cache line size.



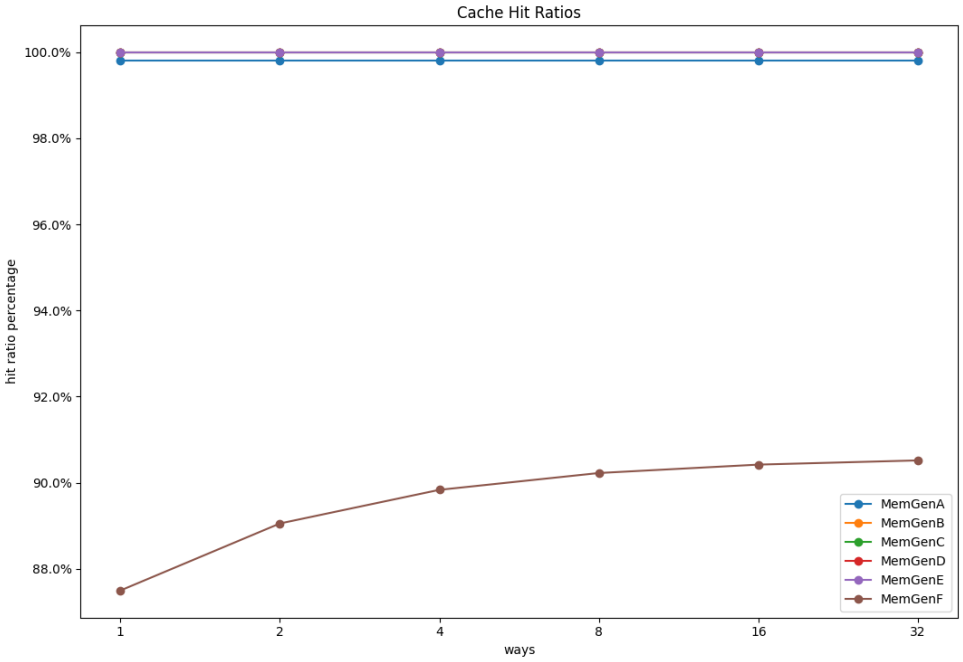
Exp3_256_LFU



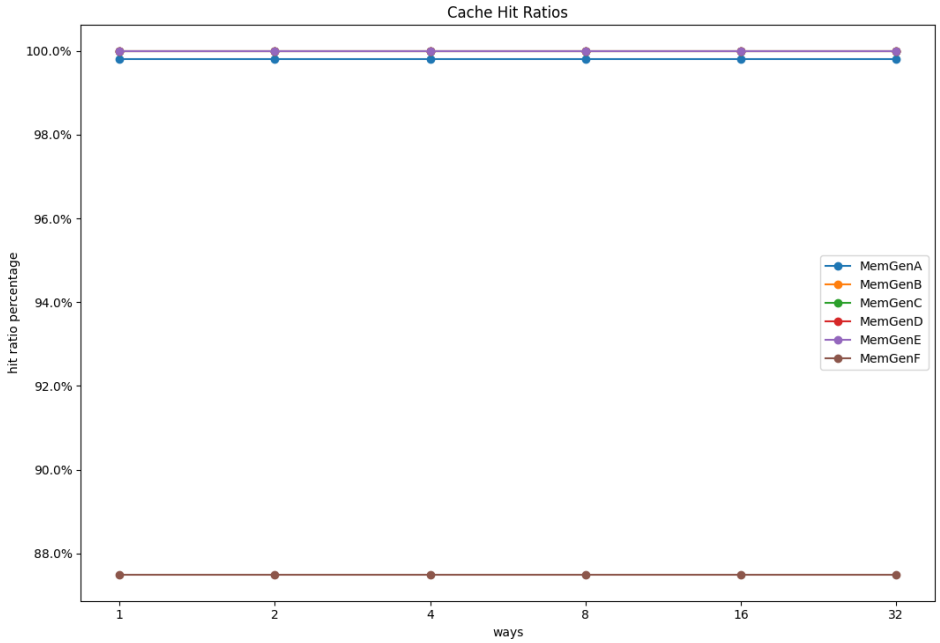
Exp3_256_LRU



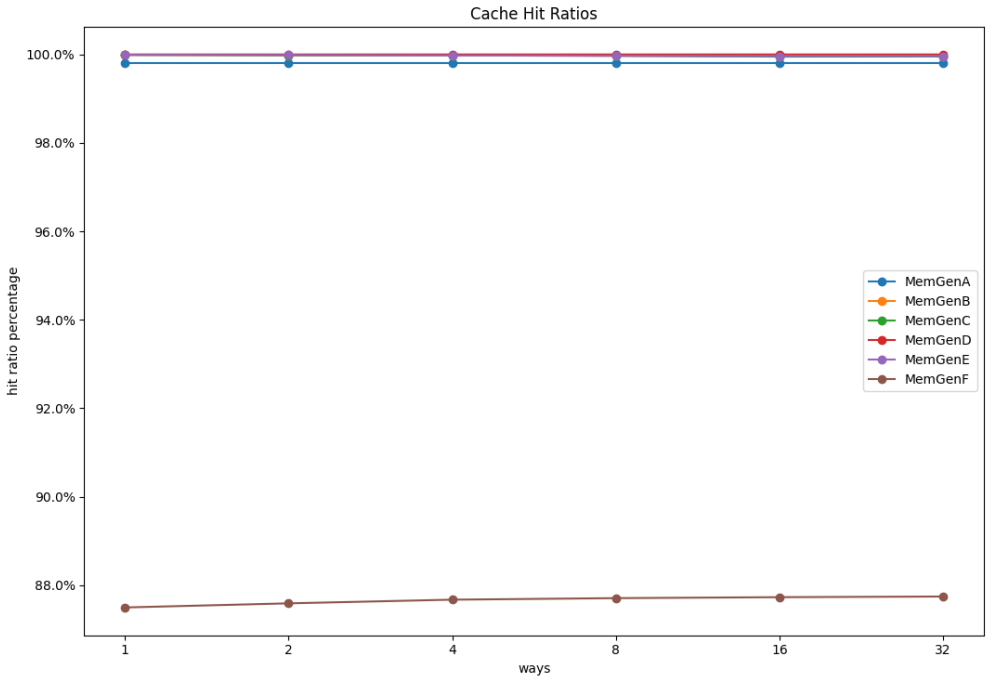
Exp3_256_Random



Exp3_512_LFU



Exp3_512_LRU



Exp3_256_Random

Replacement policy **LRU:**

With all the memory generators, the hit ratio did not change with the change of the number of ways the cache had, and given the policy of replacement LRU all the available ways get populated equally, that is why in a memory generator like F, we find it always 0 from start to finish in the 32 line size cache because even when we cycle through the addresses, no old addresses stay there since they all change rapidly, however we can see that when we increased the line size from 32 to 256 to 512, the hit ratio increased significantly for generator F (from 0% to 75% to 80% in the same order), while remaining constant for the other generators.

Replacement policy **Random**:

The hit ratio did not change from the Least Recently Used replacement policy, however, we can find some little change in the last memory generator (F) as the random policy can result in choosing specific lines to change and leave others without change until it gets back to it again after going through the whole cycle (addresses > 4*64*1024). If we look at the line sizes comparisons, we can see that when we increased the line size from 32 to 256 to 512, the hit ratio increased significantly for generator F (from 0% to 75% to 80% in the same order), while remaining constant for the other generators.

Replacement policy **LFU**:

The hit ratio with most generators was consistent with the change of the number of ways, however for generator F there was a clear increase in the performance of the cache the more ways the cache had. Also, if we compare the results between the 32, 256, and 512 line sizes, we are going to notice that the hit ratio increases as we increase the line size for generator F, but other generators' hit ratios were almost the same for all line sizes.

Conclusions:

- From the previous experiment results, changing the replacement policy (least frequently used (LFU), least recently used (LRU), or random), the performance did not significantly change (only significant performance change was noticed in experiment #2 regarding memory generator F which we can observe that the performance increased as we changed the replacement policy from LRU to Random to LFU in that order.)
- From the previous results, we can see that we can establish a general rule of thumb that the performance increases (hit ratio increases) as we increase the number of cache lines, the line sizes, and the number of ways.
- In experiment #2, the performance is almost the same when we increase the number of ways and fix the cache line size (with the exception of random generator F).
- The previous results, however, do *not* test all the possible conditions of a cache, and the performance is still highly affected by the program we run.
- In Experiment #3, even though we increased the cache line size to 4 times the last experiments, no performance drops were found. In fact, generally speaking, all the memory generators got a better result than the previous experiments regardless of the replacement policy. We are not entirely sure yet if the performance may drop if we increase the cache line size further.

Contributions:

- Seif Sallam: base skeleton of our design, a lot of code refactoring, and wrote a test case.
    - Functions & Methods:
        - Random Class (entirely) - encapsulation of existing code
        - SetAssociativeCache class:
            - Constructor
            - TestCache
            - IsInSet
            - UpdateSet
            - FindReplacmentIndex
            - FindLeastFrequent
            - Find RecentlyUsed
        - Utilities:
            - GetPatternB
            - TestB
            - TestC
- Youssef Agiza: Implementation of SetAssocitave cache class, code refactoring, collaborated on implementing and debugging the test cases and the replacement policies. Also implemented the plotter in python.
    - Set Associative Cache Class methods:
        - InitalizeSets
        - GetTag
        - GetSetIndex
        - InitalizeBitNumbers
        - IsInSet
        - UpdateSet
        - Debugging Loggers: LogSetInfo, LogCacheInfo, LogUpdateInfo
    - Utility functions
        - TestC
    - Makefile
    - plotter.py
- Kareem Amr: Handling the user input, code refactoring, implementation of the experiment and the test cases.
    - Functions & methods:
        - GetPatternA
        - TestA
        - Experiment #1, #2, and #3
        - Handling user input
        - Saving the output into CSV files
        - Utility functions such as:
            - GetAddress
            - ExecuteExp
            - GetHitRatio
            - InitalizeVariables

- freePointers
- SaveFiles