CSCE 231/2303: Computer Organization and Assembly Language — Project 2: Cache Performance

Salma Khalil
ID: 900213357
Islam Hassan
ID: 900213579
Ziad Hassan
ID: 900213728
Ahmed El-Barbary
ID: 900213964

The American University in Cairo

Summer 2023

Contents

1 Program Overview

This program aims to study the effect of varying the line size and number of ways on the performance, here measured using hit ratio, of a n-way-set-associative cache. We implemented a cache simulator in C++ to run two experiments to gain insights about the two independent variables in our study, Line Size and Number of Ways. The two experiments were done assuming a constant-sized cache of 16KB and memory of 64MB. As for the first experiment, the line size was varied from 16 to 128, while each time doubling the size, keeping the number of ways constant at 4. The second experiment had the line sized held constant at 32B, while the number of ways was varied from 1 to 16. For each combination of line size and number of ways in both experiments, 6 memory references generators were used, the second and third of which generate random references, while the rest generate sequential references, but has different upper values.The experiments have shown that the hit ratio not only depends on the line size and number of ways, but also on the references it receives. Indeed, this simulator validates our understanding of the cache, that it works optimally when the references make use of spatial and temporal locality, which is the case in non-random memory generators.

2 Program Implementation

We followed the object-oriented programming paradigm in our implementation of the simulator. Although the implementation is simple and a functional programming paradigm would have incurred less overhead, object-oriented programming is more maintainable in the sense that we can follow on this project in the future to allow more line sizes, number of ways, non-constant cache, and even other cache placement policies. The program has one class *CacheSim* that has an array of *Line* Structs and one main method *search*. *Search* receives a memory address and returns true if this memory address is a hit in the cache. The following snippet shows how this is implemented

```
bool CacheSim::Search(uint32_t address) {
    // Removed the offset bits
    address = address / line_size;
    // Extracted the index
    uint16_t index = (address % num_of_sets) * num_of_ways;
    // The only remaining bits are tag bits now
    address = address / num_of_sets;

    // check all blocks of the set
    bool hit = false;
    int i = 0;
    for(; i < num_of_ways; ++i) {
        if(cache[index + i].valid) {
            if (cache[index + i].tag == (uint16_t) address) {
                hit = true;    // hit
                break;
            }
        }
        else{
            // invalid block
            cache[index + i].valid = true;
            cache[index + i].tag = (uint16_t) address;
            break;     // cold-start
        }
    }
}
```

Listing 1: Extracting the address fields and searching in the cache

The part of the search function above extracts the index to get the set index in the cache, and the tag to check the matches. Using the tag and the valid bit of the Line Struct, the for loops goes over the lines of the set, to check if the line is valid so that it can compare the tags and declares a hit if there is a matching tag. If the line is not valid, then the tag of the address we are searching for is placed in this line and a miss is returned. If the set has all its blocks valid and yet no matching tags, FIFO replacement policy is implemented and again a miss is declared, as shown below:

```
bool CacheSim::Search(uint32_t address) {
    ...
    // all blocks have data ==> replace according to FIFO.
    if(i == num_of_ways) {
        // effectively popping the first block
```

```
6           for(int j = index + 1; j < num_of_ways + index; ++j)
7               cache[j - 1] = cache[j];
8           cache[index + num_of_ways - 1].tag = address;   //
    effectively push back the new line
9       }
10  }
```

Listing 2: Replacing a line according to FIFO

In the main, the above function is called 1M for each memory address generator, all of which are shown in *Listing 3*, for each combination of line size and number of ways in both experiments. The results are then written to a CSV file for analysis.

```
1  unsigned int memGen1()
2  {
3      static unsigned int addr=0;
4      return (addr++)%(DRAM_SIZE);
5  }
6  unsigned int memGen2()
7  {
8      static unsigned int addr=0;
9      return rand_()%(24*1024);
10 }
11 unsigned int memGen3()
12 {
13     return rand_()%(DRAM_SIZE);
14 }
15 unsigned int memGen4()
16 {
17     static unsigned int addr=0;
18     return (addr++)%(4*1024);
19 }
20 unsigned int memGen5()
21 {
22     static unsigned int addr=0;
23     return (addr++)%(1024*64);
24 }
25 unsigned int memGen6()
26 {
27     static unsigned int addr=0;
28     return (addr+=32)%(64*4*1024);
29 }
```

Listing 3: Implementation of the memory generators

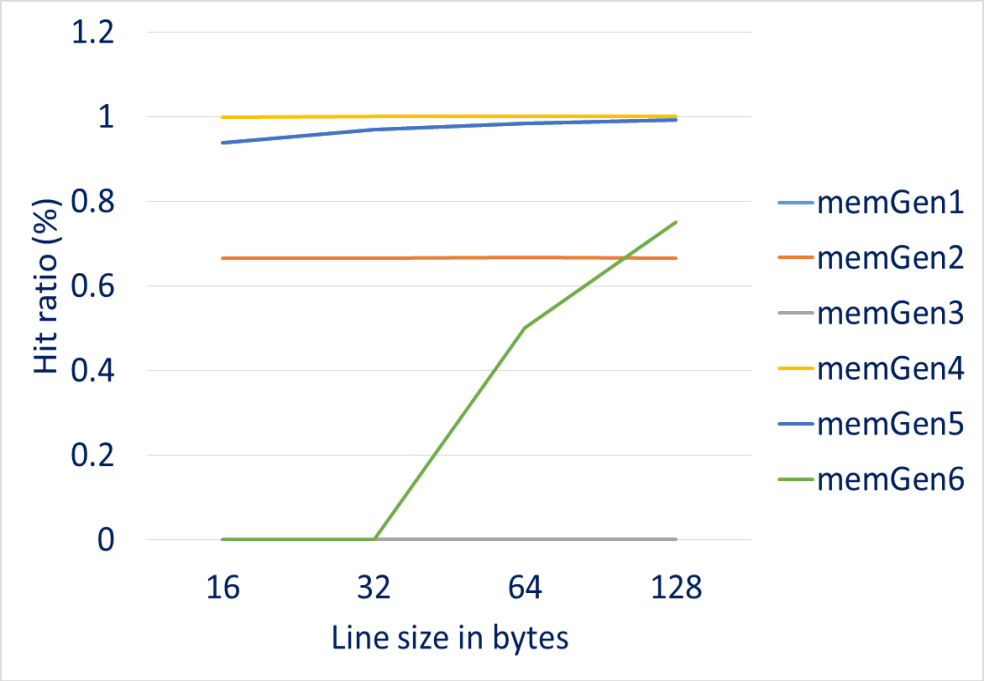
3 Data Representation And Analysis

Two experiments have been conducted to analyze the hit ratio of different memory generators. The main objective of such experiments is to investigate the impact of changing line size and the number of ways (associativity) on the hit ratio. To do so, the program considers multiple memory generators [memGen1(), memGen2(), memGen3(), memGen4(), memGen5() and memGen6()] to provide random addresses for each experiment to utilize, and it then collects data on their hit ratios. It should be noted that for each test case in each experiment 1000000 random addresses were generated using the memory generators. Then, the resulted data was written and graphed in a CSV for analysis.

Given the implementation of the memory generators shown above and the cache size to be 16 * 1024 bytes, we have considered several calculations to validate the results.

3.1 First Experiment: Impact of Line Size on Hit Ratio

This experiment evaluated the hit ratio for each memory generator using various line sizes while keeping the number of ways fixed at 4. The hit ratio was measured for the given line sizes, ranging from the smallest to the largest, and plotted against the line size as shown in graph (1).



Graph(1)

Graph(1) concludes that increasing the line size generally improves the hit ratio for memory generators, except for memGen3 and memGen6. The following sections try to provide a clear analysis of the behaviour of each line in the graph.

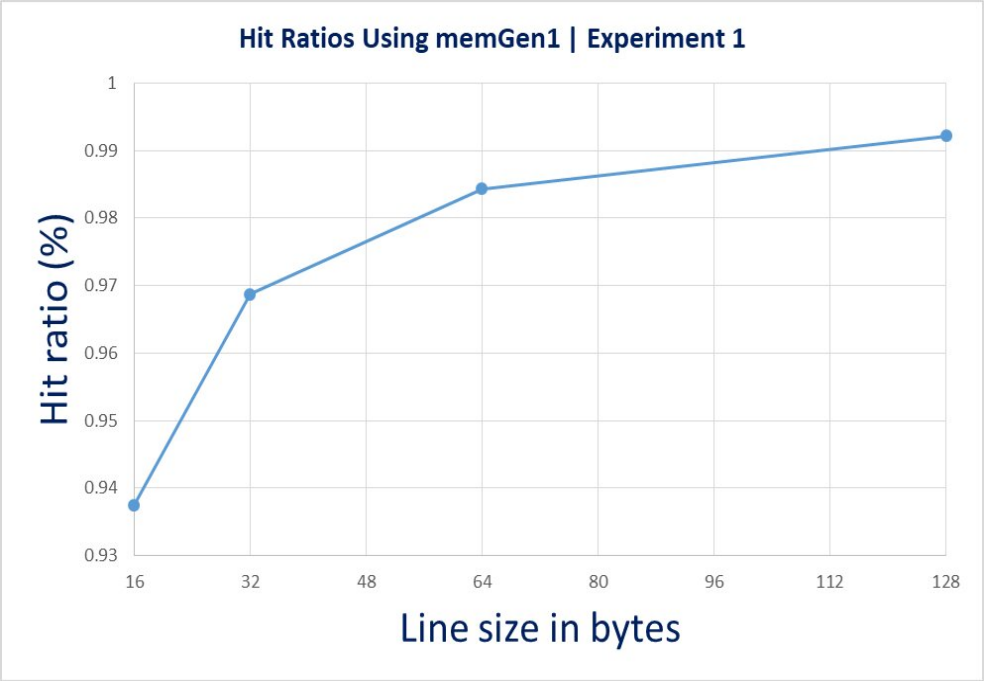
3.1.1 Hit Ratios Using memGen1:

Since it is sequential accessing of memory we will have one miss per each block used.

```
1. Line Size 16 :
    Number of cache blocks = (16*1024)/16 = 1024
    Number of used cache blocks = 10000000/16 = 62500
    Number of misses = 62500
    Number of hits = 1000000-62500 = 937500
    Hit ratio = 937500/1000000 = 0.9375
2. Line Size 32 :
    Number of cache blocks = (16*1024)/32 = 512
    Number of used cache blocks = 1000000/32 = 31250
    Number of misses = 31250
    Number of hits = 968750
    Hit ratio = 968750/1000000 =0.96875
3. Line Size 64 :
    Number of cache blocks = (16*1024)/64 = 256
    Number of used cache blocks = 10000000/64 = 15625
    Number of misses = 15625
    Number of hits = 1000000-15625= 984375
    Hit ratio = 984375/1000000 = 0.984375
4. Line Size 128 :
    Number of cache blocks = (16*1024)/128 = 128
    Number of used cache blocks = 128 /10000000 = 7812.5
    Number of misses = 7813
    Number of hits = 1000000-7813 = 992187
    Hit ratio = 1000000 / 992187 = 0.992187
```

Listing 4: Calculating hit ratios at different line sizes

Which matches the experimental results shown in graph(2).

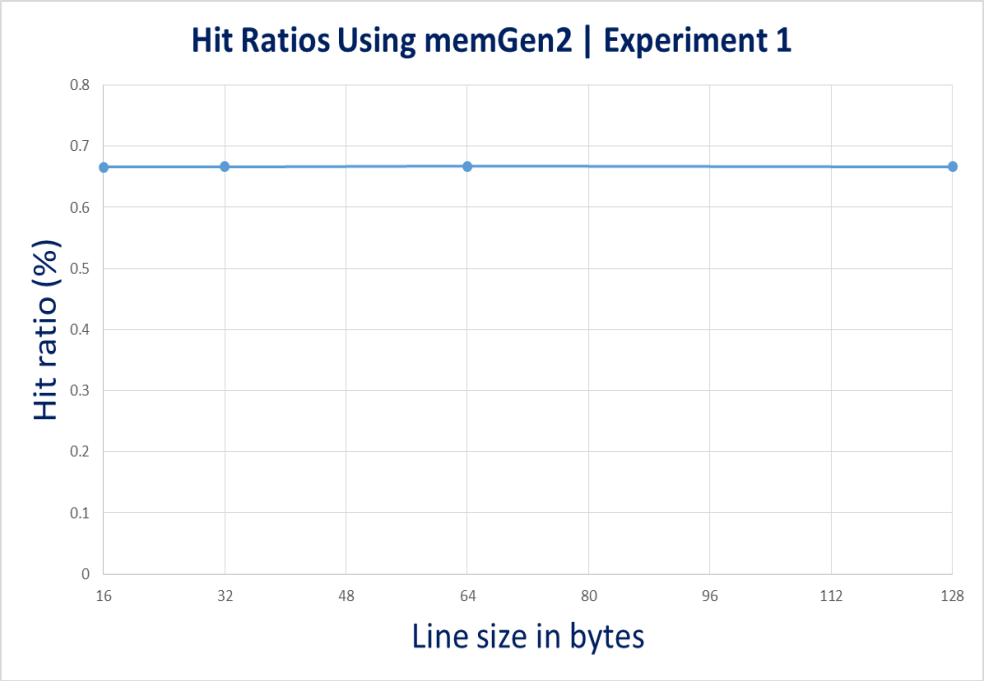


Graph(2)

It is justified that the hit ratio becomes higher and higher as we increase the line size more each time because this increase in size allows to use the spatial locality more efficiently as illustrated in the graph(2).

3.1.2 Hit Ratios Using memGen2:

Although the numbers shown in graph(3) are completely random but we know from the implementation of memGen2() function that they are mod 24576. So, we will always have memory references ranging from zero to 24576. We also know that cache size is 16384 bytes. We know that all of the possible references to cache blocks are subset of the range of values we have. We can use probability to get an approximate hit value The probability that one of indices of the cache lines generated by the random function = 24576/16384 = 0.66667

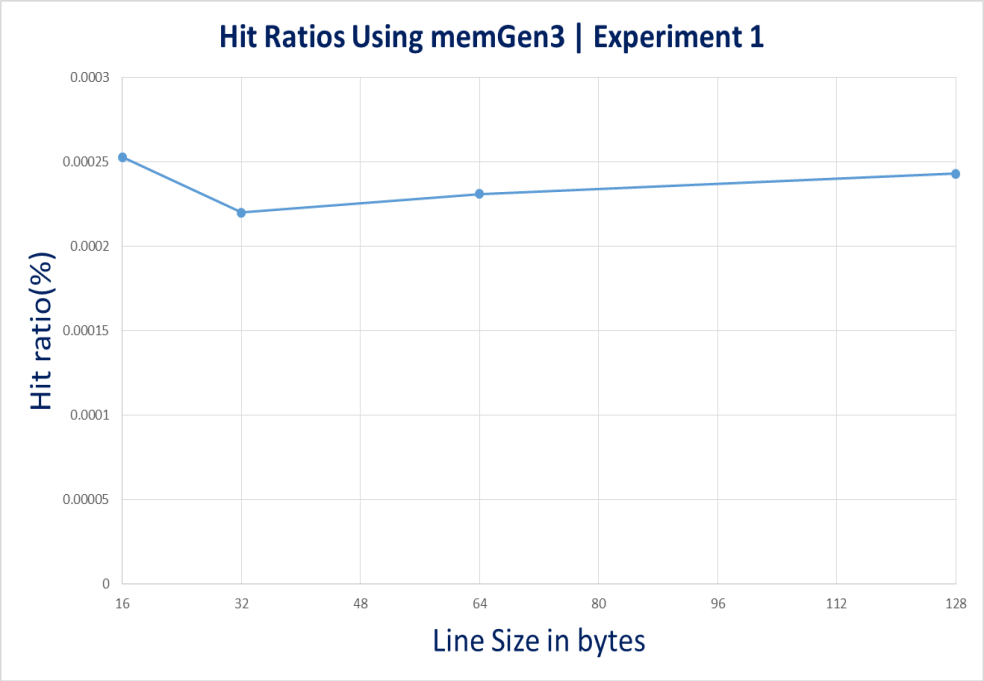


Graph(3)

The same analysis won't be different for any line size since the same analogy can be used for all line sizes and the fact the line size isn't a factor in our calculations.

3.1.3 Hit Ratios Using memGen3:

Although the numbers shown in graph(4) are completely random but we know from the implementation of memGen3() function that they are mod 67108864. So, we will always have memory references ranging from zero to 67108863.We also know that cache size is 16384 bytes. We know that all of the possible references to cache blocks are subset of the range of values we have. We can use probability to get an approximate hit value The probability that one of indices of the cache lines generated by the random function $\approx 67108864/16384 = 0.00024414$



Graph(4)

The same analysis won't be different for any line size since the same analogy can be used for all line sizes and the fact the line size isn't a factor in our calculations

3.1.4 Hit Ratios Using memGen4:

We are sequentially accessing. However, from the implementation of memGen4(), there is mod 4096. So, the references to the memory we are going to receive are from 0 to 4095. Then, we repeat this again until we reference 1000000 times. The cache size is 16 Kilobytes which is 16384 bytes so we won't use all of them and, instead of using them all, we would keep hitting older values as we loop back through 0 to 4096 again . The previous reasoning justifies why are the hitting ratios are higher than memGen1.

```
1. Line Size 16 :
```

Since line size is 16 , in 4096 accesses the number of misses = 4096/16 = 256 and the rest is going to be hits since we will reference the previous values again with no replacement.

```
Hit ratio = (1000000-256)/1000000 = 999744/1000000 =0.999744
```

```
2. Line Size 32 :
```

Since line size is 32 , in 4096 accesses the number of misses = 4096/32 = 128 and the rest is going to be hits since we will reference the previous values again with no replacement.

```
Hit ratio = (1000000-128)/1000000 = 999872/1000000 =0.999872
```

```
3. Line Size 64 :
```

Since line size is 64 , in 4096 accesses the number of misses = 4096/64 = 64 and the rest is going to be hits since we will reference the previous values again with no replacement.

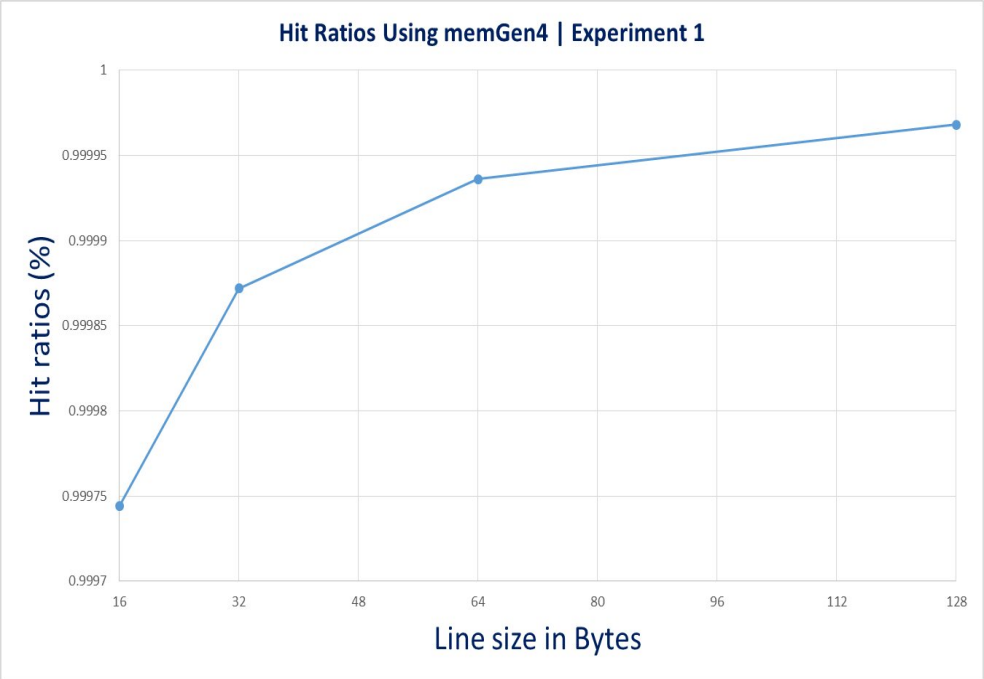
```
Hit ratio = (1000000-64)/1000000 = 999936/1000000 =0.999936
```

```
4. Line Size 128 :
```

Since line size is 128 , in 4096 accesses the number of misses = 4096/128 = 32 and the rest is going to be hits since we will reference the previous values again with no replacement.

```
Hit ratio = (1000000-32)/1000000 = 999968/1000000 =0.999968
```

Which matches the experimental results plotted in graph(5).

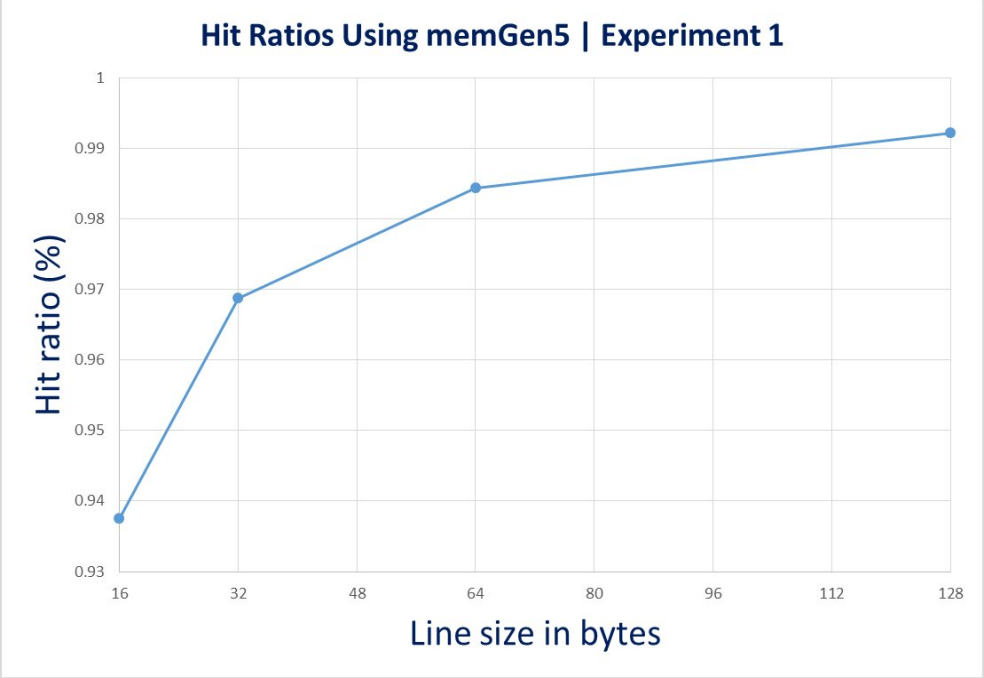


Graph(5)

It is justified that the hit ratio becomes higher and higher as we increase the line size more each time because this increase in size allows to use the spatial locality more efficiently as illustrated in graph(5).

3.1.5 Hit Ratios Using memGen5:

We have sequential accessing but it is mod 65536. So, the references to the memory we are going to receive memory references from 0 to 65535. Then, we repeat this again until we reference 1000000 times. The cache size is 16 Kilobytes which is 16384 bytes so we will keep sequentially accessing until it passes the 16384 and then it won't hit any of the old values again, but it would continue on the same pattern as memGen1. The previous reasoning justifies why are the hitting ratios are equal to memGen1. We can use same analysis used in memGen1 to justify the results.



Graph(6)

3.1.6 Hit Ratios Using memGen6:

The case of memGen6 is similar to memGen1 and memGen5. However, the increments are not sequential ; it increments 32 per time.

```
1. Line Size 16 :
```

Since line size is 16 bytes , and the increments are in units of 32 bytes it's impossible to use spatial locality because when the cache fetches the next 16 bytes from memory and we try to access after these it will be considered as miss each time.

```
Hit ratio = 0
```

```
2. Line Size 32 :
```

Since line size is 32 bytes , and the incremenets are in units of 32 bytes it's impossible to use spatial locality becuase when the cahce fetches the next 32 bytes from memory and we try to access after these it will be considered as miss each time.

```
Hit ratio = 0
```

```
3. Line Size 64 :
```

Since line size is 64 bytes , and the incremenets are in units of 32 bytes, our first access will be a miss but after this access the cache will fetch the next 64 bytes in the line so the next refernce will be a hit and this pattern keeps repeating

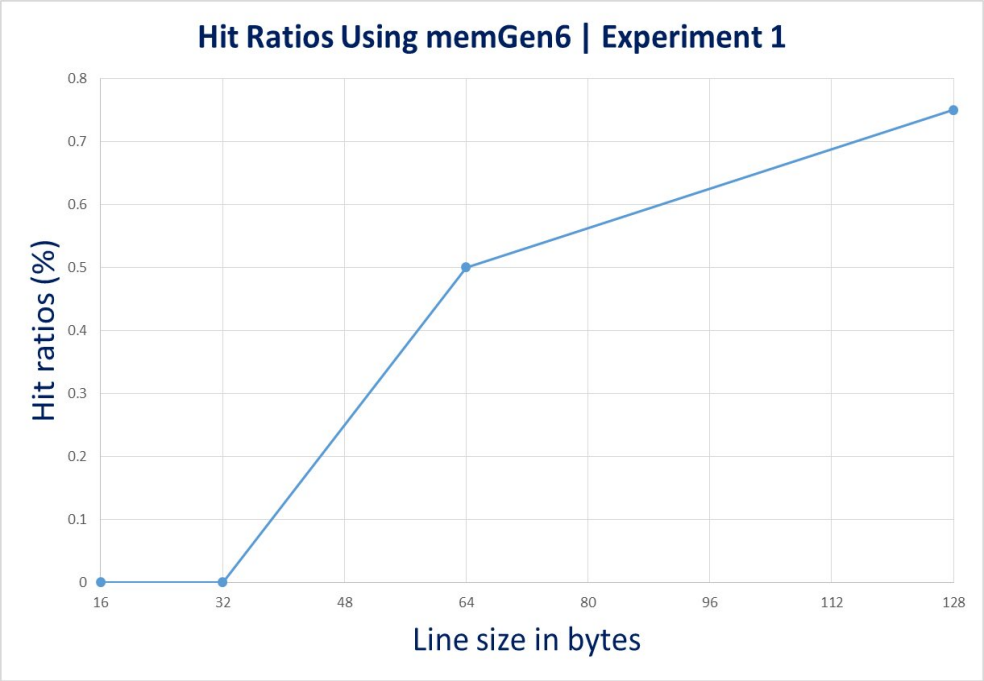
```
Hit ratio ~ 0.50
```

```
4. Line Size 128 :
```

Since line size is 128 bytes , and the incremenets are in units of 32 bytes, our first access will be a miss but after this access the cache will fetch the next 128 bytes in the line so the second, third and fourth refernce will be a hit and this pattern keeps repeating

```
Hit ratio ~ 0.75
```

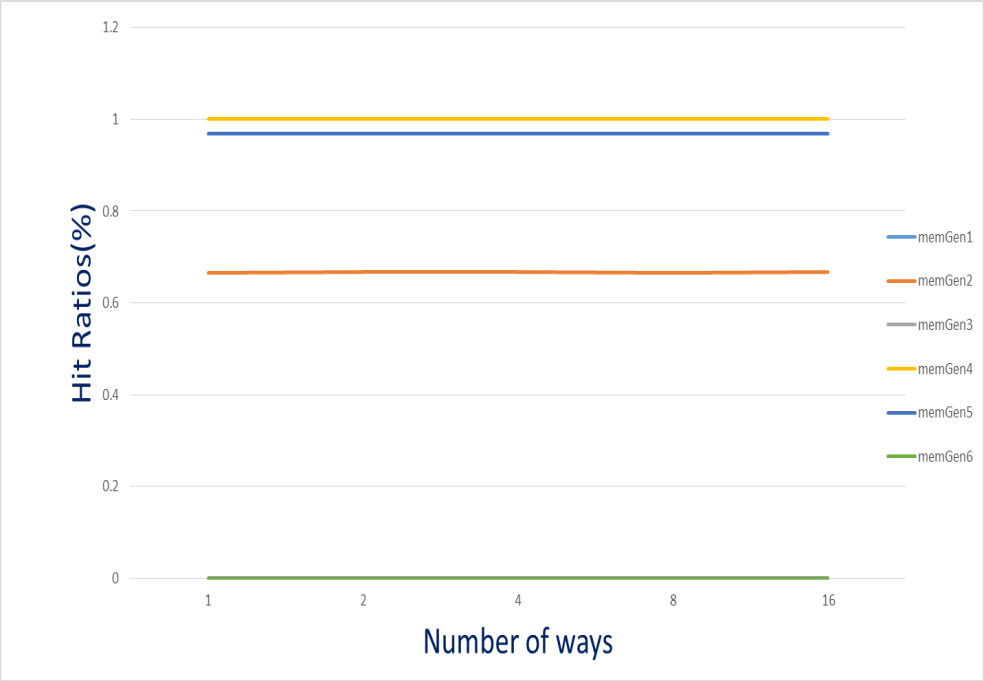
Which matches the experimental results plotted in graph(7).



Graph(7)

3.2 Second Experiment : Impact of Number of Ways on Hit-Ratio

For this experiment, we computed the hit ratio by varying the number of ways while keeping the line size fixed at 32 bytes. All possible numbers of ways and their corresponding hit ratios were considered. These data points were then plotted against number of ways as shown in graph(8).



Graph(8)

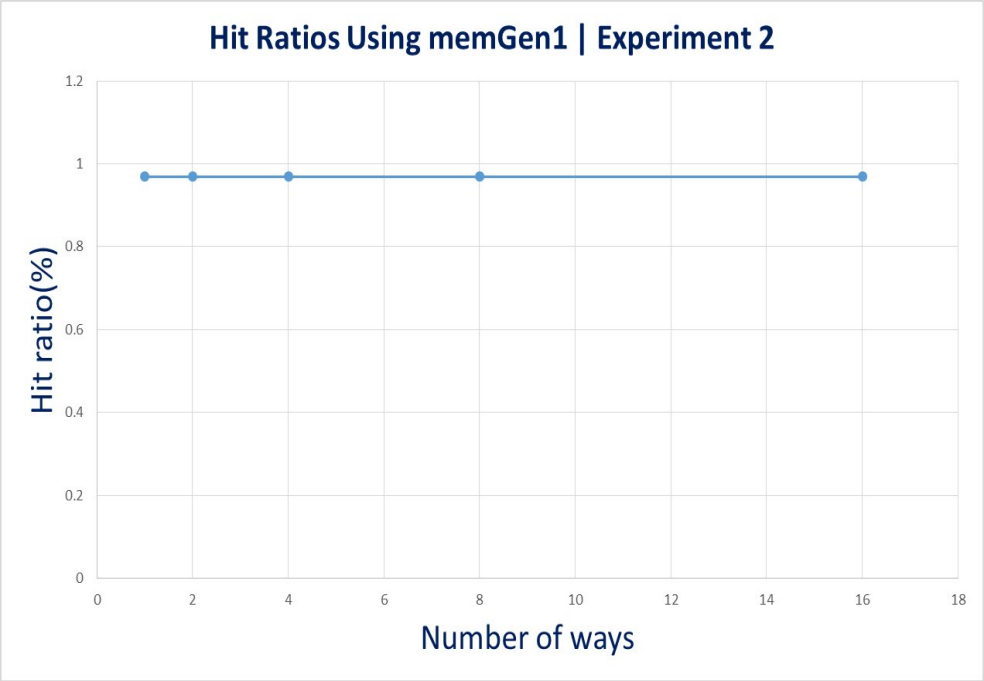
Graph(8) concludes that varying the number of ways in n-way-set associative cache has no impact on the cache performance. The following sections try to provide a clear analysis of the behaviour of each line in the graph.

3.2.1 Hit Ratios Using memGen1:

In memGen 1, the references are called sequentially mod the line size, which means there is no temporal locality in this memGen, so increasing the number of ways has no effect on the hits rate; instead, the hit rate is decided by the following equation, which depends on the number of lines that is constant across all the ways, as illustrated in *Graph 9*. For Line Size 32, these results will be the same:

```
1. Line Size 32 :
    Number of cache blocks = (16*1024)/32 = 512
    Number of used cache blocks = 1000000/32 = 31250
    Number of misses = 31250
    Number of hits = 968750
    Hit ratio = 968750/1000000 =0.96875
```

Listing 5: Calculating hit ratio for memGen1



Graph (9)

3.2.2 Hit Ratios Using memGen2:

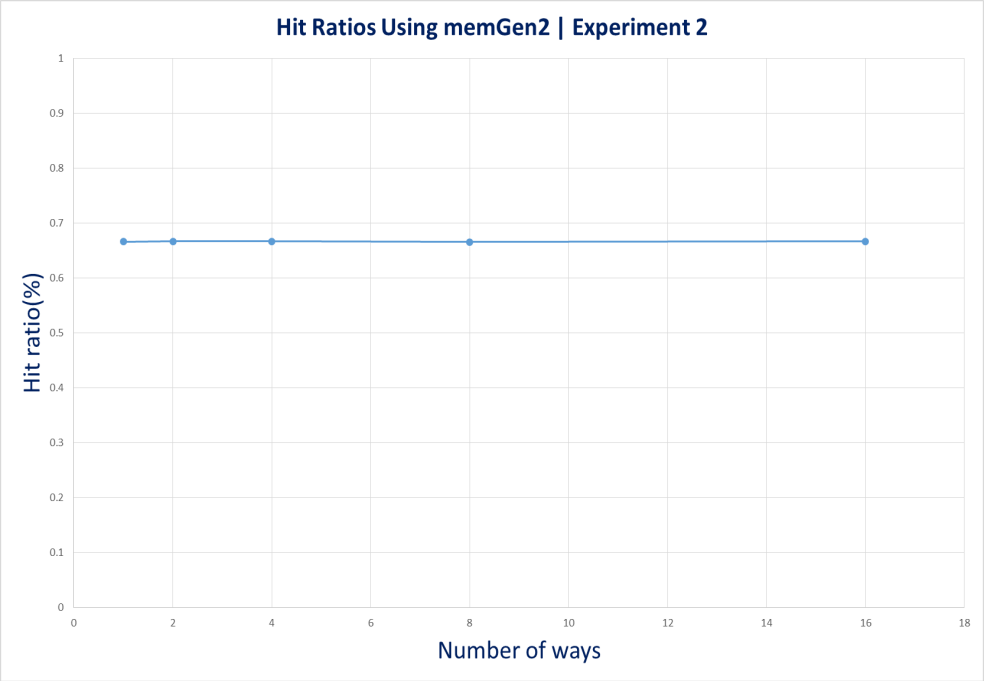
In memGen 2, the hit ratio is not dependent on the number of ways, but it is dependent on the mapping of the number of byte references in the memory to the number of available bytes in the cache. Although the numbers shown are completely random but we know from the implementation of memGen2() function that they are mod 24576. So, we will always have memory references ranging from zero to 24576. We also know that cache size is 16384 bytes. We know that all of the possible references to cache blocks are subset of the range of values we have. We can use probability to get an approximate hit value The probability that one of the indices of the cache lines generated by the random function =

```
For Line Size 32, these results will be the same:
 the random function = 24576/16384 = 0.66667

```

Listing 6: Calculating hit ratios for memGen2

Since the above calculation holds for all number of ways, the hit ratio does not change, *Graph 10.*



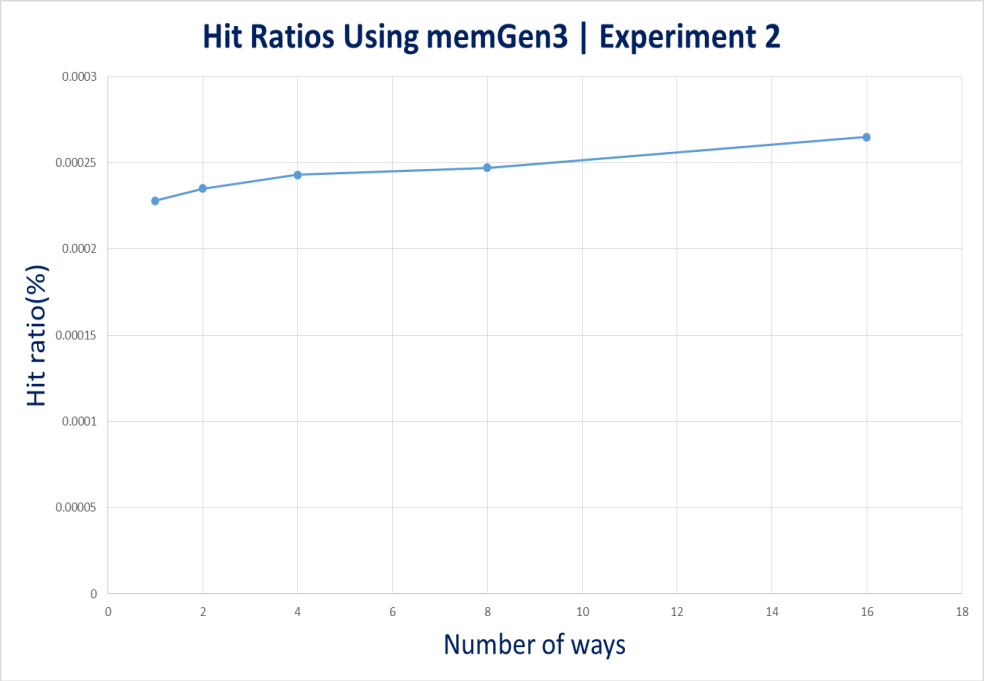
Graph (10)

3.2.3 Hit Ratios Using memGen3:

In memGen 3, the hit ratio is not dependent on the number of ways, but it is dependent on the mapping of the number of byte references in the memory to the number of available bytes in the cache. Although the numbers shown in *Graph 11* are completely random but we know from the implementation of memGen3() function that they are mod 67108864. So, we will always have memory references ranging from zero to 67108863. We also know that the cache size is 16384 bytes. We know that all of the possible references to cache blocks are subset of the range of values we have. We can use probability to get an approximate hit value The probability that one of the indices of the cache lines generated by the random function

```
67108864/16384 = 0.00024414
```

Listing 7: Calculating hit ratios for memGen3



Graph (11)

3.2.4 Hit Ratios Using memGen4:

In memGen4, we are sequentially accessing. However, from the implementation of memGen4(), there is mod 4096. So, the references to the memory we are going to receive are from 0 to 4095. Then, we repeat this again until we reference 1000000 times. The cache size is 16 Kilobytes which is 16384 bytes so we won't use all of them and, instead of using them all, we would keep hitting older values as we loop back through 0 to 4096 again.
The main difference between this experiment and experiment 1 is that we have varying number of ways and as a result the number of sets will change as we keep the cache size constant at 16Kib and the line size constant at 32 bits.
Calculating number of sets for each number of ways

1 Way- Number of sets $= \frac{16\times1024}{32\times1} = 512$ sets

2 Ways- Number of sets $= \frac{16\times1024}{32\times2} = 256$ sets

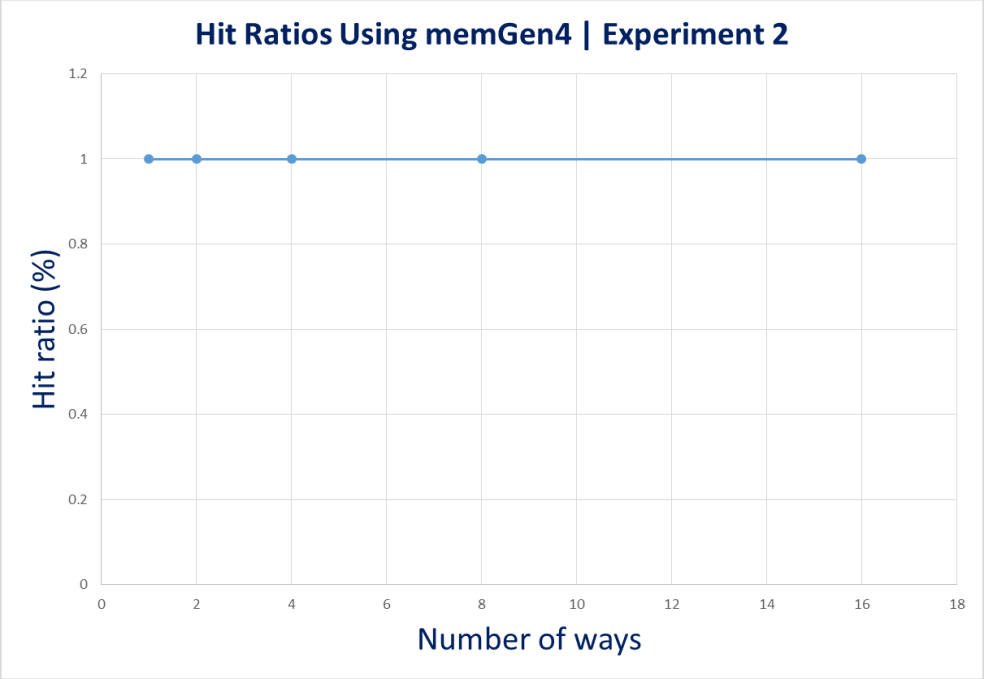
4 Ways- Number of sets $= \frac{16\times1024}{32\times4} = 128$ sets

8 Ways- Number of sets $= \frac{16\times1024}{32\times8} = 64$ sets

16 Ways- Number of sets $= \frac{16\times1024}{32\times16} = 32$ sets

In all of the previous cases of experiment two the number of sets will still be less than the number of indices we are trying to reference so we will end up hitting the same old results over and over again. The only misses we will have are going to be the misses due to cold starts until we fill the cache completely and start accessing without misses. In our case Since line size is 32 , in 4096 accesses the number of misses = 4096/32 = 128 and the rest is going to be hits since we will reference the previous values again with no replacement. The previous reasoning explains why are we having a higher hit ratio than memGen1 and also explains why are we having same results for all the different number of ways we tested.

```
Hit ratio = (1000000-128)/1000000 = 999872/1000000 =0.999872
```



Graph (12)

3.2.5 Hit Ratios Using memGen5:

We have sequential accessing but it is mod 65536. So, the references to the memory we are going to receive memory references from 0 to 65535. Then, we repeat this again until we reference 1000000 times. The cache size is 16 Kilobytes which is 16384 bytes so we will keep sequentially accessing until it passes the 16384 and then it won't hit any of the old values again, but it would continue on the same pattern as memGen1. The previous reasoning justifies why are the hitting ratios are equal to memGen1.
The main difference between this experiment and experiment 1 is that we have varying number of ways and as a result the number of sets will change as we keep the cache size constant at 16Kib and the line size constant at 32 bits.
Calculating number of sets for each number of ways

1 Way- Number of sets $= \frac{16\times1024}{32\times1} = 512$ sets

2 Ways- Number of sets $= \frac{16\times1024}{32\times2} = 256$ sets

4 Ways- Number of sets $= \frac{16\times1024}{32\times4} = 128$ sets

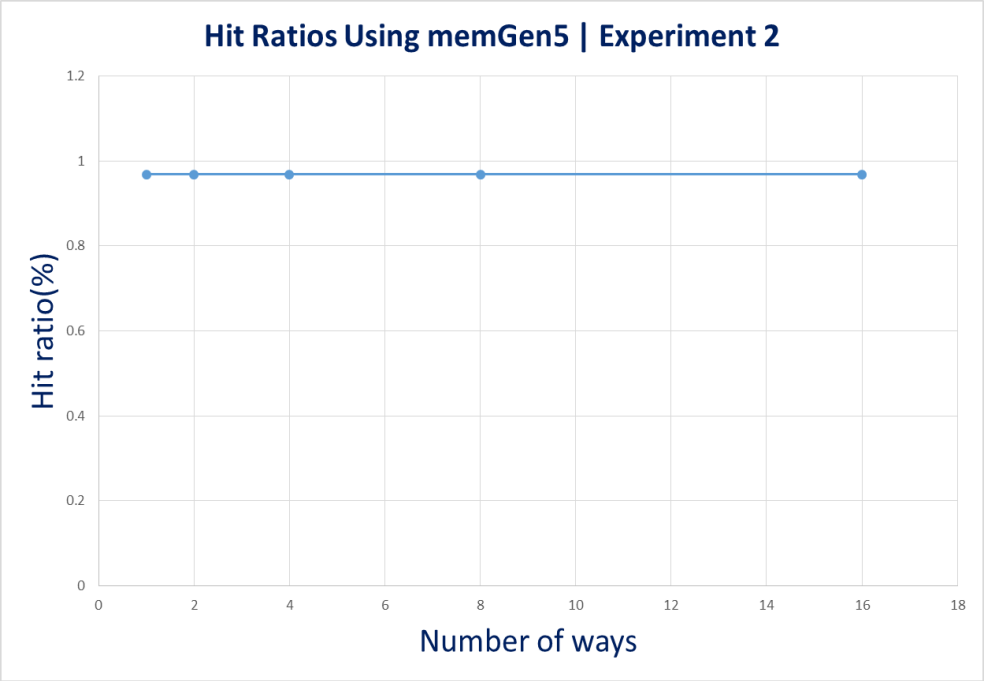
8 Ways- Number of sets $= \frac{16\times1024}{32\times8} = 64$ sets

16 Ways- Number of sets $= \frac{16\times1024}{32\times16} = 32$ sets

Although the number of sets are still less than the memory refernces we have, we will actually use the full capacity of the cache until it's completely full and we won't be receiving old values that exist again. At some point we will receive old values again, but at this point these values won't be there because they would be replaced by earlier refernces. So , we can colnclude that the result would just like the sequential accessing from memGen1 and we can use same analysis to get the hit ratio.

```
1. Line Size 32 :
    Number of cache blocks = (16*1024)/32 = 512
    Number of used cache blocks = 1000000/32 = 31250
    Number of misses = 31250
    Number of hits = 968750
    Hit ratio = 968750/1000000 =0.96875
```

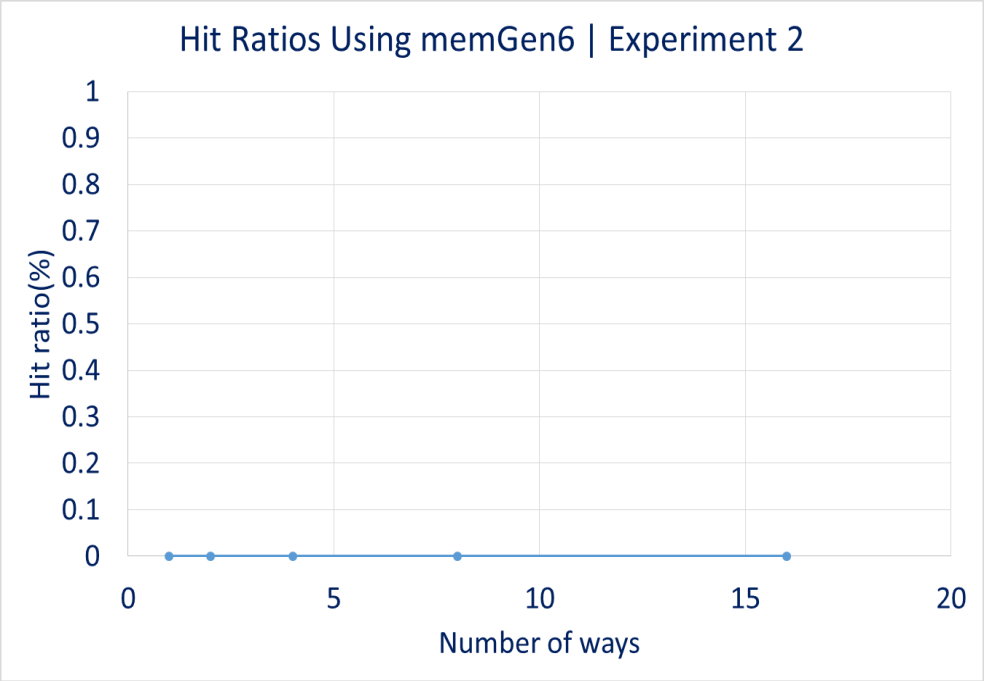
Listing 8: Calculating hit ratio for memGen5



Graph (13)

3.2.6 Hit Ratios Using memGen6:

In this memGen , the references are incremented by 32 bytes each time and our line size is 32 bytes. So, when there's a miss and the cache fetches bytes into the line and we try to access the next memory reference we won't get a hit because the step size is bigger than what we can fetch per line. In this analysis we don't care about the number of ways because it is only dependent on the line size and the step size itself. For the previous reasoning we can conclude that we won't get any hit no matter the number of ways we used which matches the result retireved by the simulator.


Hit Ratios Using memGen6 | Experiment 2
Hit ratio(%)
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
5
10
15
20
Number of ways

Graph (14)

4 Validation of Results

As mentioned in the data representation and analysis section, results have been theoretically validated through some calculations and conceptual analysis. Nonetheless, practical test cases are necessary for additional processing and validation; four test cases have been created from scratch for further testing. The idea behind these test cases is to investigate the results obtained by memGen6() function in experiment 2, and ask what if the step size is different and whether this would impact the spatial locality and, thus, the hit ratio. When looking at the implementation of memGen6(), we find that the address gets incremented by the line size or by 32 bytes. This clearly results in hit ratio of zero (refer to 3.2.6). On that basis, we created the following test cases of step sizes 4, 8, 16. Then, we investigated the obtained hit ratios from these test cases.

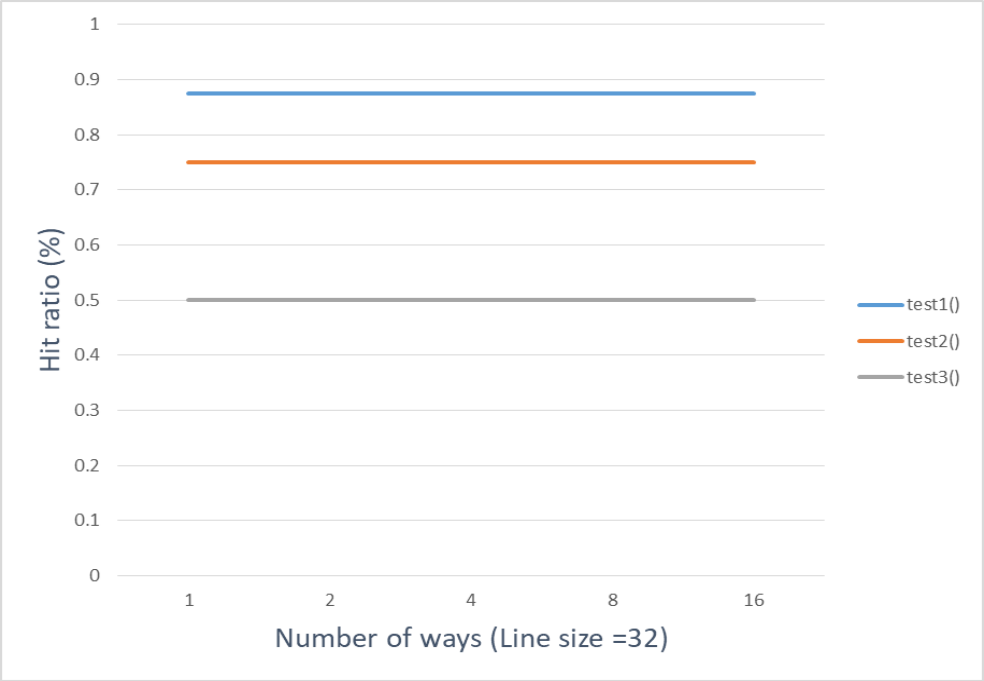
```
unsigned int test1(){
    static unsigned int addr=0;
    return (addr+=4) % DRAM_SIZE;
}

unsigned int test2(){
    static unsigned int addr = 0;
    return (addr+=8) % DRAM_SIZE;
}

unsigned int test3(){
    static unsigned int addr = 0;
    return (addr+=16) % DRAM_SIZE;
}
```

Listing 9: Implementation of the test cases

Varying the step sizes within a range less than the cache line size highlighted the concept of locality even more. The test cases' results concludes that as long as the step size increases to become closer to the line size, the hit ratio will decrease due to the misuse of locality. As shown in listing 9, test1() has a step size of 4, test2() has a step size of 8, and test3() has a step size of 16. Each function gave a bunch of hit ratios plotted in graph(15).



Graph (15)

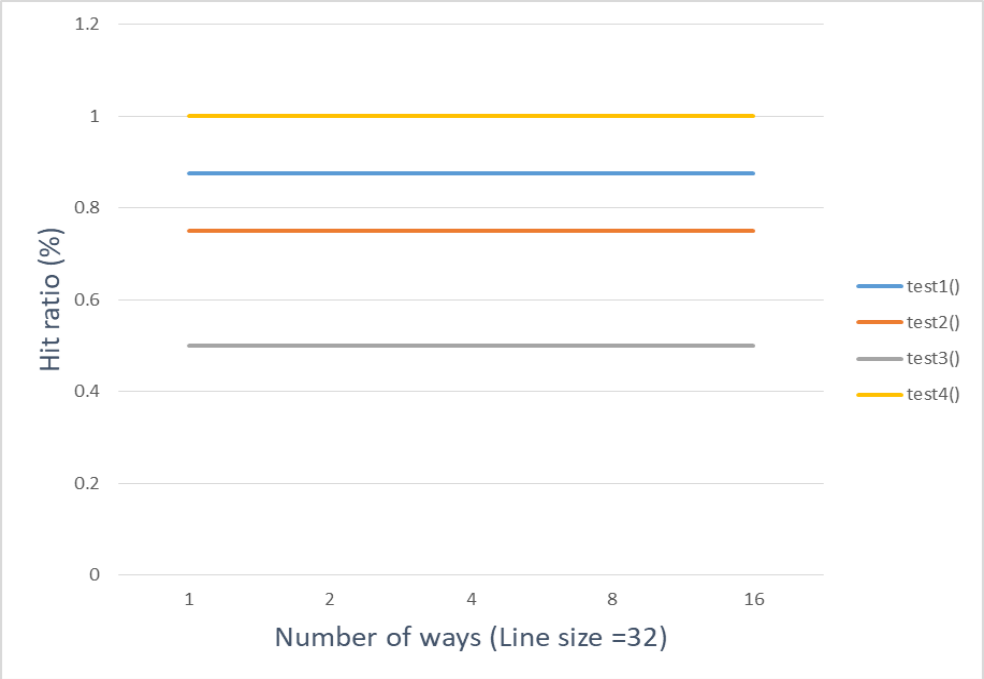
It is obvious that test1() has the highest hit ratio, followed by test2() ,and test3() comes after them. These test cases shows that the closer the step size to the line size the lower the locality and the hit ratio. The graph also confirms the observation mentioned before that varying the number of ways does not contribute to any improvement or change. The main contributors to the hit ratio and cache performance are the line size and the step size if any. In addition, these test cases practically confirm the results obtained by memGem6() in experiment 2. As its step size is the line size and, thus, it gets straight zeros against the different numbers of ways.

The first three test cases are done to investigate the change in the hit ratios while the address is incremented by different sizes. The fourth and the final test case is mainly done to keep the address fixed at a particular value. No increments happen, and, thus, the same address is accessed regardless of the number of ways. The following is the implementation of test4() function.

```
unsigned int test4(){
    unsigned int addr = 14;
    return (addr);
}
```

Listing 10: Implementation of the fourth test case

The implementation shows that the address has a fixed value of 14. Theoretically, this should result in only one miss over all trails regardless of the number of ways. This exactly happened and the results are as follows:



Graph (16)

Graph(16) copies Graph(15) with an additional test4() function. It shows that the hit ratio of test4() is almost 1. It also highlights that this is the highest value obtained of all the test as it gets no step size.

5 Conclusion

The cache performance experiments have shed light on the significant influence of line size and reference patterns on the hit ratio of a n-way-set-associative cache. Larger the line sizes generally lead to improved cache performance, while sequential memory references outperform random ones. It also should be noted that varying the number of ways did not show any contributions to the cache overall performance. These findings offer valuable insights into cache design decisions and reaffirm the importance of tailoring cache configurations to match the access patterns of the target applications. As cache hierarchies play a crucial role in modern computing systems, understanding these performance characteristics is essential for maximizing overall system efficiency and responsiveness.