Cache Performance

Project 2

Ahmed Ali
Ahmed Emad Abdelbaset
Bavly Labieb
Omar Elfouly

A report for CSCE 2303
Computer Organization and Assembly Language

CSCE
AUC
July 2023

1 Introduction

The goal of this experiment is to study the effect of cache parameters and the way applications access memory on cache performance.
The parameters we will study are line size, number of ways while keeping cache size and DRAM size constant.
We will study 6 different memory access patterns that we will be naming memGen1 to memGen6.

1.1 Method

To properly isolate the effect each parameter has on cache performance we will be splitting the study into two experiments.

Experiment 1 will study the effect of line size on the hit ratio, while keeping the number of ways constant. Experiment 2 will study the effect of the number of ways on hit ratio, while keeping line size constant.

To simulate a cache we will be using a set-associative cache simulator to run each experiment.

1.2 MemGen functions

```
unsigned int memGen1()
{
        static unsigned int addr=0;
        return (addr++)%(DRAM_SIZE);
}

unsigned int memGen2()
{
        static unsigned int addr=0;
        return  rand_()%(24*1024);
}

unsigned int memGen3()
{
        return rand_()%(DRAM_SIZE);
}

unsigned int memGen4()

{
        static unsigned int addr=0;
        return (addr++)%(4*1024);
}
```

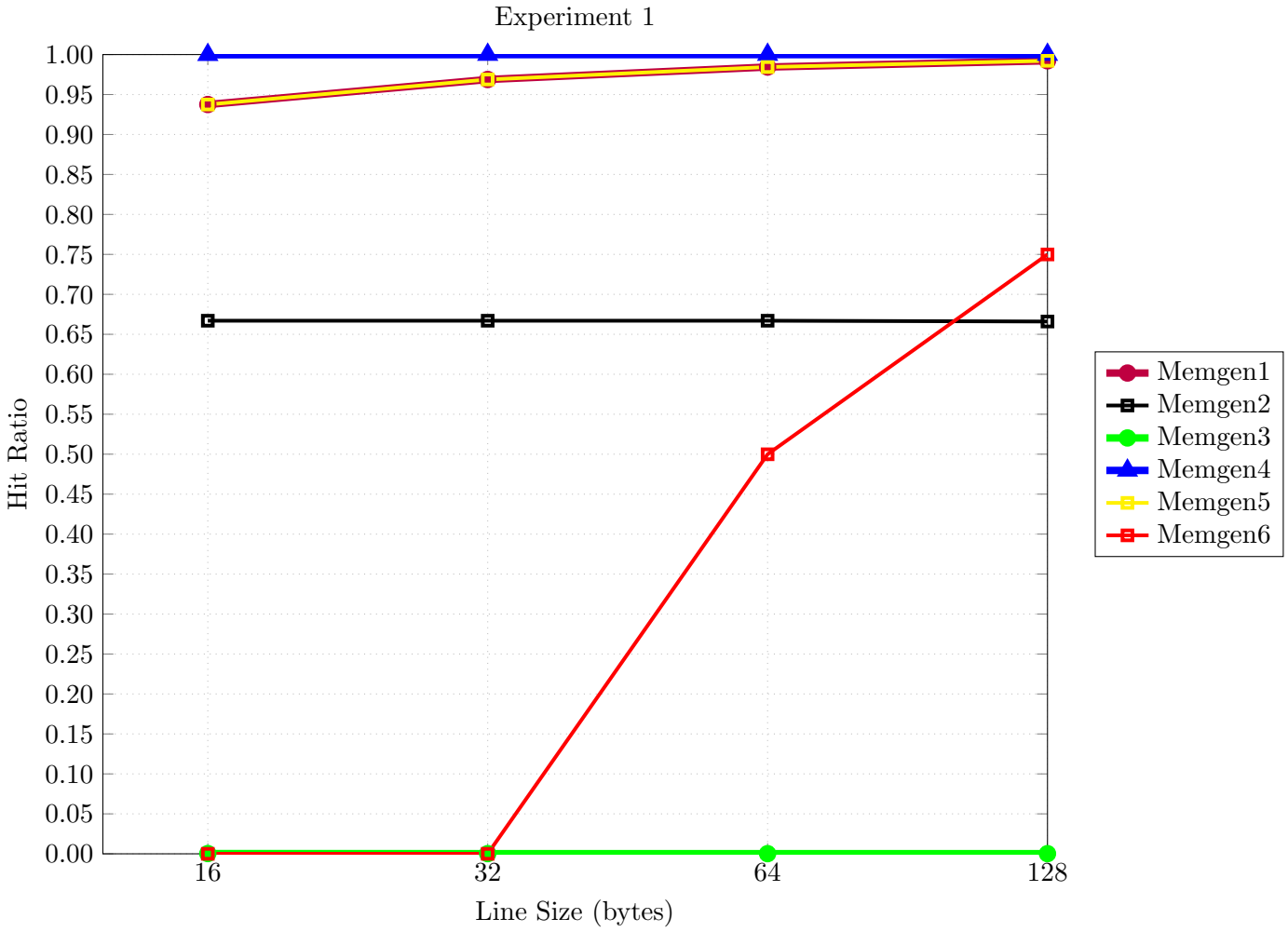
```
unsigned int memGen5()
{
        static unsigned int addr=0;
        return (addr++)%(1024*64);
}

unsigned int memGen6()
{
        static unsigned int addr=0;
        return (addr+=32)%(64*4*1024);
}
```

2 Experiment 1

2.1 Expectations

We expect that increasing line size will lead to an increase in hit rate due to cache being able to benefit more from spatial locality. However, it is also possible that the application will have very little spatial locality which would lead to line size having very little effect on the hit rate.
As line size increases beyond a certain length it will lead to the number of sets decreasing causing the cache to be unable to benefit as much from any temporal locality.

2.2 Results



2.3 Data Analysis

2.3.1 Memgen1

memGen1 simulates an application where the memory address is incremented sequentially, with each every consecutive address being incremented by 1, and we then perform the modulus operation on this address value such that the address returned is between 0 and the size of the DRAM, in this case, 64MB. Our cache size is 16KB, which is smaller than the DRAM size, so not all the DRAM memory can fit into the cache, however, since the cache fetches many bytes per every miss, and since we are incrementing sequentially, we would expect that the hit ratio to be high. This is verified by looking at the results we found and at the graph, where we can see that the hit ratio is in the 90s for every line size. More specifically however, let us take a look at every line size and try to dissect why the simulation arrived at each result. With a line size of 16 bytes, what happens is as follows: Initially, we try to fetch the first address, address 0. This is a cold start miss, as the cache is initially empty. This results in the cache fetching 16 bytes from the memory, namely, bytes 0-15. Now, the address is incremented by 1, so the cache looks for address 1, which has already been fetched since the cache now has bytes 0-15, so that is a hit. This continues and we get hits all the way until byte 15. At byte 16 however, we have another miss, as this is the 17th byte, and the line size only has 16, so the 17th is a miss. Then bytes 16-31 are fetched, and the same pattern continues,

where we get 15 hits after the initial miss. This continues for the entire memory generation, even when the address goes beyond the 16KB mark, as the same pattern will follow. What actually happens is that since our number of ways is 4 for experiment 1, each index can have up to 4 lines, but this does not actually affect the hit ratio, since what ends up happening is that the address is constantly getting larger, and we never end up making use of previously fetched memory, other than the aforementioned 1 miss and 15 hits per every 16 bytes. This gives us a hit ratio of

$$\frac{15}{16} = 0.9375 \tag{1}$$

which is precisely what we got. This pattern is the same throughout the varying line size, where for 32 bytes we end up with 31 hits and 1 miss, as per every initial miss we get, we fetch the next 31 bytes, which end up being the bytes we will be looking for due to memGen1's sequential incrementation, and we end up with a hit ratio of

$$\frac{31}{32} = 0.96875 \tag{2}$$

which is also what the simulation showed. In fact, we can create a formula to find the hit rate for any line size for Experiment 1, memGen1, which is

$$hit\ ratio = \frac{line\ size - 1}{line\ size} \tag{3}$$

and this gives us

$$\frac{63}{64} = 0.984375 \tag{4}$$

and

$$\frac{127}{128} = 0.9921875 \tag{5}$$

The conclusion reached for Experiment 1, memGen1, is that if we are accessing all memory elements sequentially, then the larger the line size, the larger the hit ratio. This is because we would be making use of the spatial locality present in this application, as we are accessing the memory location right after the previously accessed one, and with a large line size we would have more of these sequential memory elements already present in the cache after just a single initial miss.

2.3.2 Memgen2

In memGen2, the *rand*() function is used to generate a random address, which is then confined to be within 0 and 24KB, done using the modulus operator. This can be considered an application that does not make use of any locality whatsoever. Let us take a line size of 16 bytes as our base to try and analyze what happens. With a line size of 16 bytes, we get a total number of lines in the cache to be

$$\frac{16 * 1024}{16} = 1024 \tag{6}$$

and the total number of possible lines is

$$\frac{24 * 1024}{16} = 1536 \tag{7}$$

Initially, the probability of getting a miss is much larger than a hit, since the cache is empty. In fact, after the first cold start miss, the probability of getting a hit is 1/1023. As the cache grows, we cannot determine exactly how many hits and misses we are getting as it is the application is random, however, we know that eventually, with many iterations, the cache will be filled. At this point, we have 1024 lines in the cache, and a possibility of 1536 lines being generated in the application, which gives us

$$\frac{1024}{1536} = \frac{2}{3} = 0.667 \tag{8}$$

and this is what we saw from the results, with small deviations due to the random nature of the application. The same method can be applied to the rest of the block sizes, the only difference being that the number of lines would be smaller, however, since we are dividing both the cache size and the range by the same factor, the change is proportional, therefore leading to the same result, which is a hit ratio of about 0.667. There is no spatial locality in this application, hence the indifference in results in the varying line sizes, which did not affect the hits we would get due to the fact that the next address has nothing to do with the current address.

2.3.3 Memgen3

In memGen3, the same *rand*() function that was used in memGen2 is used here, with the only difference being the range of addresses produced by the different applications. We can also consider this an application that does not exploit localities at all. The application produces memory addresses in the range of 0 to 64MB, compared to 0 to 24KB in memGen2. Using the same approach, we can see that the hit ratio would be

$$\begin{aligned} \frac{16 * 1024}{16} &= 1024 \; lines \\ \frac{64 * 1024 * 1024}{16} &= 4194304 \; lines \\ \frac{1024}{4194304} &= 0.00024414 \end{aligned} \tag{9}$$

which is what we saw from the results, with deviations, again, present due to the randomness of the application. This also shows that, similar to application 2, there is no spatial locality present, which is why the varying line sizes did not affect the hit rate. Contrary to application 2 however is the much lower hit rate in this application, and this was due to the fact that the addresses spanned a much larger range, so the probability that the memory we looking for was already present in the cache was much lower, hence the lower hit rate.

2.3.4 Memgen4

We expect memGen4 to preform well since it has a high amount of temporal locality since it generates memory addresses sequentially in a cyclic manner, accessing the same memory locations repeatedly within a limited range.

The graph shows that as the line size increases, the hit ratio also increases. Which aligns with our expectation that larger line sizes would improve cache performance due to better utilization of the cache and improved temporal locality.
This happened because larger cache line sizes allow the cache to store more consecutive memory addresses from MemGen4. As MemGen4 accesses memory addresses sequentially and cyclically, larger line sizes can accommodate more of these sequential addresses within a single cache line. Consequently, the cache becomes more efficient in serving memory accesses from cached data, leading to a higher hit ratio.
Due to the address pattern MemGen4 address uses, the number of capacity misses is equal to zero for all cache line sizes. This is because MemGen4 accesses a 4-kilobyte memory region, which is fully contained within the 16-kilobyte cache for all line sizes tested. As a result, there are no capacity misses as the cache can hold the entire memory region accessed.
By analysing how the cache behaves with different line sizes and the MemGen4 memory address pattern we can prove that larger cache line sizes utilize the cache efficiency and lead to a higher hit ratio for MemGen4.

The characteristics of MemGen4:

- It generates memory addresses sequentially and in cyclic manner.
- It accesses a 4-kilobyte memory region (from address 0 to 4*1024-1).

Now, let's examine how the cache behaves with different line sizes (16, 32, 64, and 128 bytes) when serving the MemGen4 memory access pattern.
For each line size, we will calculate the number of cache hits and misses for MemGen4.
Assumptions from method:

- Cache Size: 16 KB (16 * 1024 bytes)
- Number of ways = 4

Line size: 16 bytes

The hit percentage: 99.9744 %

To find the number of cache hits:
Number of cache hits= $99.9744\% * 1,000,000 = 999,744$ Number of cache misses:

$$Number\ of\ cache\ misses = Total\ number\ of\ cycles - Number\ of\ cache\ hits \tag{10}$$

$$Number\ of\ cache\ misses = 1,000,000 - 999,744 = 256 \tag{11}$$

By calculating the number of misses that should theoretically occur as follows:

$$Number\ of\ Cache\ Lines = Cache\ Size\ (16KB)\ /block\ Size\ (16B) = 1024\ lines \tag{12}$$

The memory region can be fully accommodated in 256 cache lines; MemGen4 accesses 4 KB address space. Thus, it can accommodate the entire memory region of MemGen4 without any capacity misses.

$$4KB/16B = 256\ lines \tag{13}$$

$$Conflict\ misses = 0. \tag{14}$$

MemGen4 accesses 4 KB, which contains 256 lines.S

$$Cold\ start\ missies = number\ of\ lines\ memGen4\ needs\ to\ store\ all\ addresses = 256. \tag{15}$$

This aligns with the actual hit percentages we got from running the code.
Repeating the same steps for the (32, 64 and 128 bytes) line sizes we get the following:

Line Size: 32 bytes:

The hit percentage: 99.9872%

To find the number of cache hits:

$$\text{Number of cache hits} = 99.9872/ \tag{16}$$

Number of cache misses:

$$Number\ of\ cache\ misses = Total\ number\ of\ cycless - Number\ of\ cache\ hits \tag{17}$$

$$Number\ of\ cache\ misses = 1,000,000 - 999,872 = 128 \tag{18}$$

By calculating the number of misses that should theoretically occur as follows:

$$Number\ of\ Cache\ Lines = Cache\ Size/Line\ Size = 16KB/32B = 512lines \tag{19}$$

MemGen4 can be fully accommodated in 128 cache lines.
Conflict misses = 0.
MemGen4 accesses 4 KB, which contains 128 lines.

$$Cold\ start\ missies = number\ of\ lines\ memGen4\ needs\ to\ store\ all\ addresses = 128 \tag{20}$$

Again, this aligns with the actual hit percentages we got from running the code.

Line Size: 64 bytes

Hit rate: 99.9936%
Similarly, the number of misses = 64.

$$Number\ of\ Cache\ Lines = Cache\ Size/Line\ Size = 16KB/64B = 256lines \tag{21}$$

MemGen4 can be fully accommodated in 64 cache lines.
Conflict misses = 0.
MemGen4 accesses 4 KB, which contains 64 lines.

$$Cold\ start\ missies = number\ of\ lines\ memGen4\ needs\ to\ store\ all\ addresses = 64 \tag{22}$$

Line Size: 128 bytes

Hit rate: 99.9968%
Number of misses = 32.

$$Number\ of\ Cache\ Lines = Cache\ Size/Line\ Size = 16KB/128B = 128lines \tag{23}$$

MemGen4 can be fully accommodated in 32 cache lines.
Conflict misses = 0.
MemGen4 accesses 4 KB, which contains 32 lines.

$$Cold\ start\ missies = number\ of\ lines\ memGen4\ needs\ to\ store\ all\ addresses = 32. \tag{24}$$

General formula:

$$Total\ misses = Number\ of\ lines\ occupied\ by\ 4Kb\ address\ space = \frac{4 * 1024}{linesize}. \tag{25}$$

All of the misses are cold start; the number of lines needed to store the 4Kb address space is always less than the number of cache lines.

For all tested cache line sizes (16, 32, 64, and 128 bytes), the MemGen4 memory address pattern exhibits a very low miss ratio. This is because MemGen4 accesses a 4-kilobyte memory region, which can be fully accommodated within the 16-kilobyte cache for all line sizes tested. Additionally, the number of cache misses is equal to the number of lines the addresses will occupy and they are all considered cold start misses.
This leads us to conclude that increasing the line size in the cache leads to significant enhancements in cache performance for the 'memGen4' memory address generator. With larger line sizes, the cache becomes more efficient in handling sequential memory accesses, resulting in higher hit ratios and better overall cache efficiency. As a consequence, the principle of temporal locality is well-exploited, demonstrating the importance of line size selection in cache design for workloads that exhibit strong temporal locality.

2.3.5 Memgen5

We expect this memory address generator (memGen5) to show a significant improvement in cache hit ratio as the cache line size increases.
The graph shows that as the cache line size increases, the cache hit ratio improves, which aligns with our expectations.
This happened because larger cache line sizes allow the cache to store more consecutive memory addresses from MemGen5. As MemGen5 accesses memory addresses sequentially, larger line sizes can accommodate more of these sequential addresses within a single cache line. Consequently, the cache becomes more efficient in serving memory accesses from cached data, leading to a higher hit ratio.
Let's calculate the number of misses for each line size:

Line Size: 16 bytes

The hit percentage: 93.75%
By calculating the number of misses that should theoretically occur as follows:

$$\begin{aligned} \text{Number of misses for a single cycle}(64KB) &= \frac{64Kb}{16b} = 4096 \\ \text{Number of memory accesses} &= 1000000 \\ \text{Number of cycles} &= \frac{1000000}{64 * 1024} = 15.258 \\ \text{Total number of misses} &= 15.258 * 4096 = 62499\text{misses} \\ \text{The hit percentage} &= \frac{(1000000 - 62499)}{1000000} * 100 = 93.7501\% \end{aligned} \tag{26}$$

This aligns with the actual hit percentages we got from running the code.

Line size: 128 bytes

The hit percentage: 99.2187%
By calculating the number of misses that should theoretically occur as follows:

$$\begin{aligned} \text{Number of misses for a single cycle}(64KB) &= \frac{64Kb}{128b} = 512 \\ \text{Total number of misses} &= 15.2587 * 512 = 7812\text{misses} \\ \text{The first 512 are cold start misses and the others are conflict misses} & \\ \text{The hit percentage} &= \frac{(1000000 - 7812)}{1000000} * 100 = 99.2188\% \end{aligned} \tag{27}$$

This aligns with the actual hit percentage produced by the experiment.

The General formula:

$$\text{Number of misses for a single cycle (64KB)} = \frac{64Kb}{\text{block size}} \tag{28}$$

$$\text{Total number of misses} = 15.2587 * \text{total number of lines needed to store 64Kb address space} \tag{29}$$

This leads us to conclude that larger cache line sizes significantly improve cache hit ratio for MemGen5. The results demonstrate the importance of choosing an appropriate cache line size to optimize cache performance for specific memory access patterns. In this case, increasing the cache line size reduces cache misses and improves the overall efficiency of the cache when accessing memory addresses sequentially.

2.3.6 Memgen6

We expect this memory address generator (memGen6) to have a low cache hit ratio due to its sequential access pattern, where memory addresses are accessed at regular intervals of 32 bytes. Since the cache line size is relatively small at 16 bytes, it might not be able to accommodate enough consecutive memory addresses within a single cache line, leading to frequent cache misses. However, as the line size increase

the hit ratio should start to increase as well but still it will remain relatively small.

The graph shows that the cache hit ratio is 0% for both the 16-byte and the 32-byte cache line size and gradually increases as the cache line size increases to 64 bytes and 128 bytes. This trend aligns with our expectation that larger cache line sizes will result in a higher cache hit ratio for memGen6.
This happened because the sequential access pattern of MemGen6 causes cache misses in the smaller cache line sizes.
General equation:

$$\text{Number of hits per line} = \frac{line\ size}{32} - 1 = 0 \tag{30}$$

The number of iterations needed before moving to the next $line = \frac{\text{line size}}{32}$

$$\text{The total number of hits} = \frac{totalnumberofiterations(1000000) * Numberofhitsperline}{linesize/32} - 1 \tag{31}$$

Line size: 16 bytes

Number of hits per line $= \frac{16}{32} - 1 = -0.5 = 0$ since negative numbers aren't valid.
Thus, the total number of hits = 0.

Line size: 64 bytes

Number of hits per line= $\frac{64}{32} - 1 = 1$
The number of iterations needed before moving to the next line = $\frac{64}{32}$
The total number of hits= $\frac{1000000*1}{64/32} - 1 = 499999$
The hit percentage = 49.999%
This clearly aligns with the result we obtained initially from running the code as shown: 49.99999%

Line size: 128 bytes

Number of hits per line= $\frac{128}{32} - 1 = 3$
The number of iterations needed before moving to the next line= $\frac{128}{32} = 4$
The total number of hits = $\frac{1000000*3}{128/32} - 1 = 749999$
The hit percentage = 74.9999%
This clearly aligns with the result we obtained initially from running the code as shown: 74.9999%

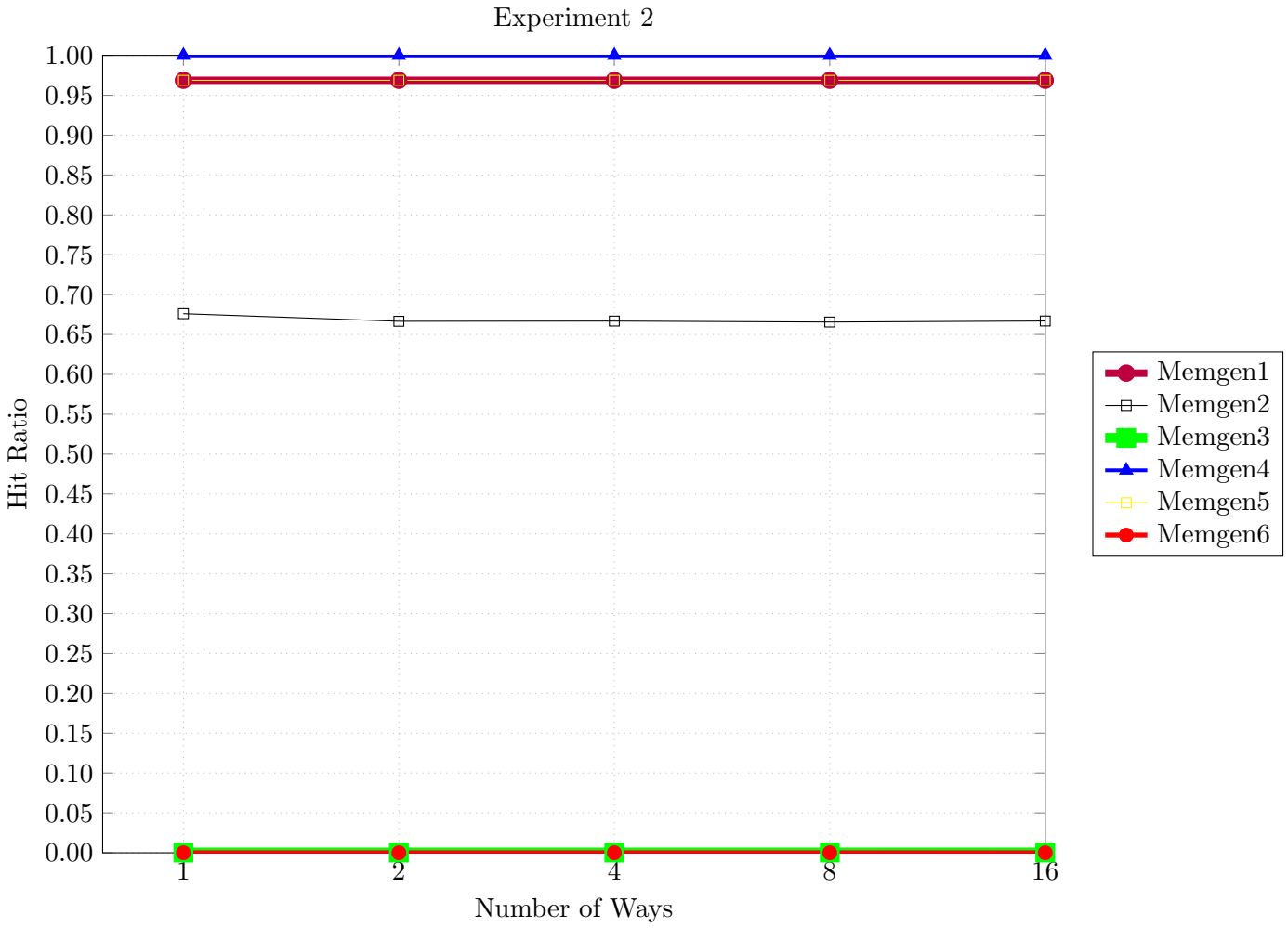
We conclude that MemGen6, with its sequential access pattern, results in zero cache hits for the (16 bytes and 32 bytes) line sizes. However, as the cache line size increases to 64 bytes and 128 bytes, the cache becomes more efficient in capturing consecutive memory addresses, resulting in a higher cache hit ratio. Therefore, a larger cache line size is more suitable for improving cache performance with MemGen6's specific access pattern.

3 Experiment 2

3.1 Expectations

We expect that an increase in the number of ways will reduce misses due to a decrease in the number of conflict misses which should mean an increase in performance assuming that addresses with the same index are repeatedly accessed.
If the number of ways ever exceeds a certain point, the number of indexes will decrease causing more comparisons since the cache will be forced to map more addresses to a single set, which will cause a large increase in cost due to the large amount of comparators required.

3.2 Results



3.3 Data Analysis

3.3.1 Memgen1

When analysing the memgen1 application, we can observe that it sequentially goes across all addresses that are < the Dram size (which is done by modding the Dram size). This means that for every miss, 31 hits follow, This is because for every byte accessed, all 32 following bytes would also be grabbed into the

cache, this is because the line size is 32 bytes. In this experiment however, we change the number of ways and it can be observed that this has no effect on the hit rate at all, because in this situation the hit rate would only depend on the line size. We can use ratios to easily calculate the number of hits.

$$\begin{aligned} miss : hit : \ total \\ 1 : 31 : \ 32 \\ misses : \ : 1,000,000 \\ \therefore misses = \frac{1,000,000}{32} = 31250 \end{aligned} \tag{32}$$

So, to get the hit rate that we got in our results we can do the following:

$$hit\,rate = \frac{1,000,000 - misses}{1,000,000} * 100 = 96.875\% \tag{33}$$

The results confirm that the number of ways would not affect the hit rate when going through the memory sequentially, only the line size would have an effect because as the line size increase we make use of special locality, however increasing the number of ways doesn't do so, hence the constant hit rate observed.

3.3.2 Memgen2

It can be observed that memgen 2 accesses memory locations randomly rather than sequentially, this means that we wont be able to make use of spacial locality as well as we did in the first memgen, however, the addresses used are confined within the first 24kb, this means that the programs locality can be utilized as the cache size isn't too small when compared to the number of addresses being called. This means that the chance to find a given address in cache is still high due to locality. Again in our case, increasing the number of ways doesn't have any major impact that affects the number of hits, however we can still see minor fluctuations in the hit rate of the different way numbers. This small difference can be attributed to the random manner that the addresses are called in. When we change the number of ways, we change the chance for a certain address access to cause a capacity miss and hence another address being removed, this will result in the removed memory location to be registered as a miss when called again. The probability of such an event occurring isn't very high, which is why the differences between each line sizes isn't significant. We can calculate the average hit rate for all ways by calculating the ratio between the cache size and the address pool size, this is of course the case after the initial cold start misses where the cache is full:

$$hit\,rate = \frac{16KB}{24KB} = \frac{16 * 1024}{24 * 1024} = \frac{2}{3} = 0.667 \tag{34}$$

This number can be considered to be the average hit rate for all the number of ways.

3.3.3 Memgen3

Memgen 3 is similar to memgen 2 in the sense that it also accesses memory locations randomly, however a vital difference exists between them. The difference being the amount of addresses being called by memgen 3 is significantly larger that memgen 2 which nullifies any locality that the previous memgen utilized. The

application randomly calls an address from the whole Dram memory locations, this ofcourse doesn't use any kind of locality, and given that the size of the Dram is significantly larger than the cache, the chance of finding a given address in the cache is very slim, this results in a very low hit rate. Again, the number of ways doesn't make any notable difference, the only reason for the minor differences in the hit rate can be attributed to something like our case in memgen 2 where the number of ways can cause/prevent a certain memory location to be removed when trying to access a new memory location of the same index. Similarly, we could also find an average hit rate by calculating the ratio of cache size to Dram size:

$$hit\,rate = \frac{16KB}{64MB} = \frac{16 * 1024}{64 * 1024 * 1024} = \frac{1}{4096} = 0.000244 \tag{35}$$

3.3.4 Memgen4

This memory generator function simulates an application that sequentially access memory in a small range of 4 kilobytes of addresses. This means this application is highly localised within a small set of addresses so we can logically expect a large hit rate. Furthermore, the cache is 4 times as large as the range of addresses used by memgen4 which means after some initial misses due to cold start we can expect continuous number of hits as there is plenty of space available for all bytes being used by this application.
The data we produced from the experiment shows a consistent hit rate of 0.999872. This number can be reached using the following calculations:

$$Number\ of\ Bytes\ being\ accessed = 4 * 1024 = 4096 \tag{36}$$

$$Number\ of\ lines\ being\ accessed = \frac{Number\ of\ Bytes\ being\ accessed}{Line\ size} = \frac{4096}{32} = 128 \tag{37}$$

Since the first access in every line is a miss then:

$$Number\ of\ misses = Number\ of\ lines\ being\ accessed = 128 \tag{38}$$

$$Hit\ rate = 1 - \frac{Number\ of\ misses}{1000000} = 1 - \frac{128}{1000000} = 0.999872 \tag{39}$$

Since number of ways was never a factor in our calculations it is logical that the hit rate remains constant.

3.3.5 Memgen5

MemGen5 is incredibly similar to memgen4 except for the fact that the range of addresses being accessed is 64 kilobytes which is 4 times larger than our cache size. Therefor our addresses are localised in 64 kilobytes which means while we still have a localised set of addresses it is a much larger range which means that implementation wise, memgen5 is identical to memgen1 since both applications have an address range too large for us to take advantage of. In other words, the range of addresses is too large which causes cache to be unable to benefit from their temporal locality.
We can reach the constant value of 0.96875 using the following calculations:

$$Number\ of\ misses = Number\ of\ lines\ accessed = \frac{Number\ of\ memory\ accesses}{line\ size} = \frac{1000000}{32} = 31250 \tag{40}$$

$$hit\ rate = 1 - \frac{Number\ of\ misses}{umber\ of\ memory\ accesses} = 0.96875 \tag{41}$$

3.3.6 Memgen6

Memgen6 simulates an application that accesses memory with a step size of 32 bytes. However, since we have a line size of 32 Bytes, we should expect every single access to be a miss since the line size is too small to take advantage of the dispersed spatial locality present in this application. Increasing the number of ways will have no effect on the line size so all accesses will still be misses.

This is clearly supported by our results, since memgen6 produces a hit rate of 0%.

4 Conclusion

The expectations we had for experiment 1 were valid. Which means that, under the assumptions we made for cache size and number of ways, an increase in line size will generally cause an increase in hit rate. However, we discovered some exceptions to this trend. Memgen2 didn't follow the trend due to it simulating an application that randomly accessing memory which is very difficult to optimise and leads to relatively bad performance. Memgen3 was similar to memgen2 except it was still random and even more delocalised which lead to a terrible hit rate. Lastly, due to having a large step size, memgen6 required a minimum of 64B per line to be optimised.
Experiment 2 managed to produce interesting results that differed from our expectations. This is because We discovered that increasing the number of ways did not have much of an effect on any of our results. For each memegen there is a unique reason as to why none of them benefit from the increase in ways. These reasons can be found in the sub sections dedicated to the data analysis of experiment 2.