THE AMERICAN UNIVERSITY IN CAIRO

Computer Science and Engineering Department

N-Way Set Associative Cache Simulator

Kareem Sayed, ID:900212697
Mohamed Alashkar, ID:900214276
Andrew Aziz, ID:900213227
Fekry Mohamed, ID:900192372

CSCE230301 - Comp Org. and Assmbly Lang Prog
Summer 2023
Jul 18, 2023

Intoduction:

In modern computer systems, processors can perform operations on registers at an incredibly fast pace. However, accessing large capacity main memory, such as Dynamic Random-Access Memory (DRAM), takes significantly more time in comparison. To address this performance gap, a small, high-speed memory called a cache is introduced between the processor and the main memory. The cache serves as a type of memory, storing frequently accessed data and instructions, thus reducing the time spent waiting for main memory access. This project aims to explore the functionality of n-way set associative caches, understand their working principles, and analyze the impact of various cache parameters on system performance.

Experiments and Methodology:

The primary objective of this project is to build a cache simulator in C/C++ that models a set-associative cache. The simulator will offer flexibility in adjusting critical cache characteristics, such as cache size, cache line size, and the number of ways the cache is set-associative. The specific cache characteristics chosen for this simulation are as follows:

- Cache Size: A fixed 16 Kbytes (16,384 bytes) cache will be implemented.
- Cache Line Size: The cache line size will be variable and set to 16, 32, 64, and 128 bytes to evaluate its influence on cache performance.
- Number of Ways: The number of ways, representing the associativity of the cache, will be varied between 1, 2, 4, 8, and 16.
- Computer Memory Address Space: The simulation will consider a 64 Mbytes (67,108,864 bytes) computer memory address space.

To ensure the accuracy and reliability of the cache simulator, different address generator functions are used to produce the memory addresses for the simulation. Each generator will produce a large set of memory references (at least 1,000,000 references) to create realistic simulation scenarios.

Throughout the project, we will conduct two experiments to measure the hit and miss ratios of the cache under different configurations. Firstly, we will analyze the hit ratio for each memory reference generator and explore how varying the cache line size with the number of ways =4 impacts cache performance. Secondly, we will analyze the hit ratio for each memory reference generator and explore how varying the number of ways with the cache line size of 32 bytes impacts cache performance. The results of these experiments will be plotted and analyzed to gain valuable insights into the cache's behavior and its interaction with the system's memory hierarchy.

Results:

Hit Ratio	Miss Ratio	momGen()	ways	Cache Line Size	Hit Ratio	Miss Ratio	momGen()	ways	Cache Line Size
93.75	6.25	1	4	16 bytes	**99.9744**	0.0256	4	4	16 bytes
96.875	3.125	1	4	32 bytes	**99.9872**	0.0128	4	4	32 bytes
98.4375	1.5625	1	4	64 bytes	**99.9936**	0.0064	4	4	64 bytes
99.2187	0.7813	1	4	128 bytes	**99.9968**	0.0032	4	4	128 bytes
66.5758	33.4242	2	4	16 bytes	**93.75**	6.25	5	4	16 bytes
66.647	33.353	2	4	32 bytes	**96.875**	3.125	5	4	32 bytes
66.678	33.322	2	4	64 bytes	**98.4375**	1.5625	5	4	64 bytes
66.6629	33.3371	2	4	128 bytes	**99.2187**	0.7813	5	4	128 bytes
0.0271	99.9729	3	4	16 bytes	**0**	100	6	4	16 bytes
0.0256	99.9744	3	4	32 bytes	**0**	100	6	4	32 bytes
0.0258	99.9742	3	4	64 bytes	**49.9999**	50.0001	6	4	64 bytes
0.0229	99.9771	3	4	128 bytes	**74.9999**	25.0001	6	4	128 bytes

Figure 1. Experiment 1 data table

Hit Ratio	Miss Ratio	memGen()	ways	Cache Line Size	Hit Ratio	Miss Ratio	memGen()	ways	Cache Line Size
96.875	3.125	1	1	32 bytes	**99.9872**	0.0128	4	1	32 bytes
96.875	3.125	1	2	32 bytes	**99.9872**	0.0128	4	2	32 bytes
96.875	3.125	1	4	32 bytes	**99.9872**	0.0128	4	4	32 bytes
96.875	3.125	1	8	32 bytes	**99.9872**	0.0128	4	8	32 bytes
96.875	3.125	1	16	32 bytes	**99.9872**	0.0128	4	16	32 bytes
66.6512	33.3488	2	1	32 bytes	**96.875**	3.125	5	1	32 bytes
66.7131	33.2869	2	2	32 bytes	**96.875**	3.125	5	2	32 bytes
66.647	33.353	2	4	32 bytes	**96.875**	3.125	5	4	32 bytes
66.6425	33.3575	2	8	32 bytes	**96.875**	3.125	5	8	32 bytes
66.6466	33.3534	2	16	32 bytes	**96.875**	3.125	5	16	32 bytes
0.024	99.976	3	1	32 bytes	**0**	100	6	1	32 bytes
0.0245	99.9755	3	2	32 bytes	**0**	100	6	2	32 bytes
0.0249	99.9751	3	4	32 bytes	**0**	100	6	4	32 bytes
0.0249	99.9751	3	8	32 bytes	**0**	100	6	8	32 bytes
0.0258	99.9742	3	16	32 bytes	**0**	100	6	16	32 bytes

Figure 2. Experiment 2 data table

Observations:

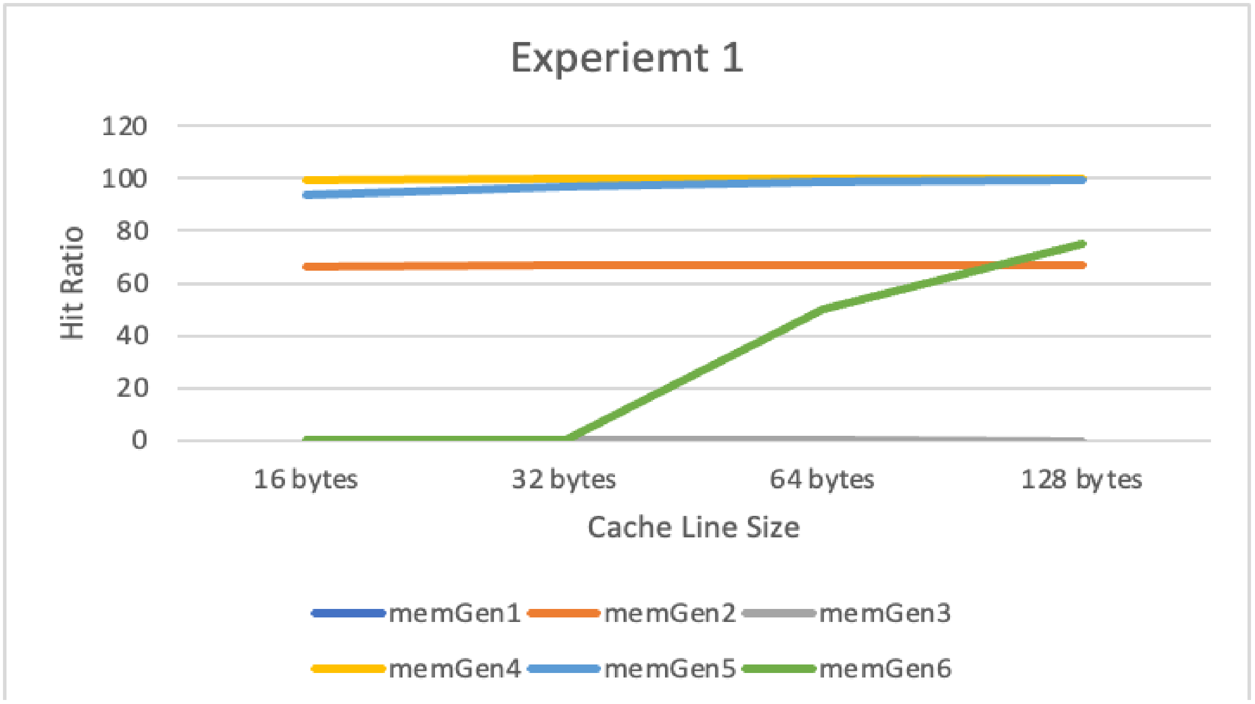

Figure 3. Experiment 1 data representation

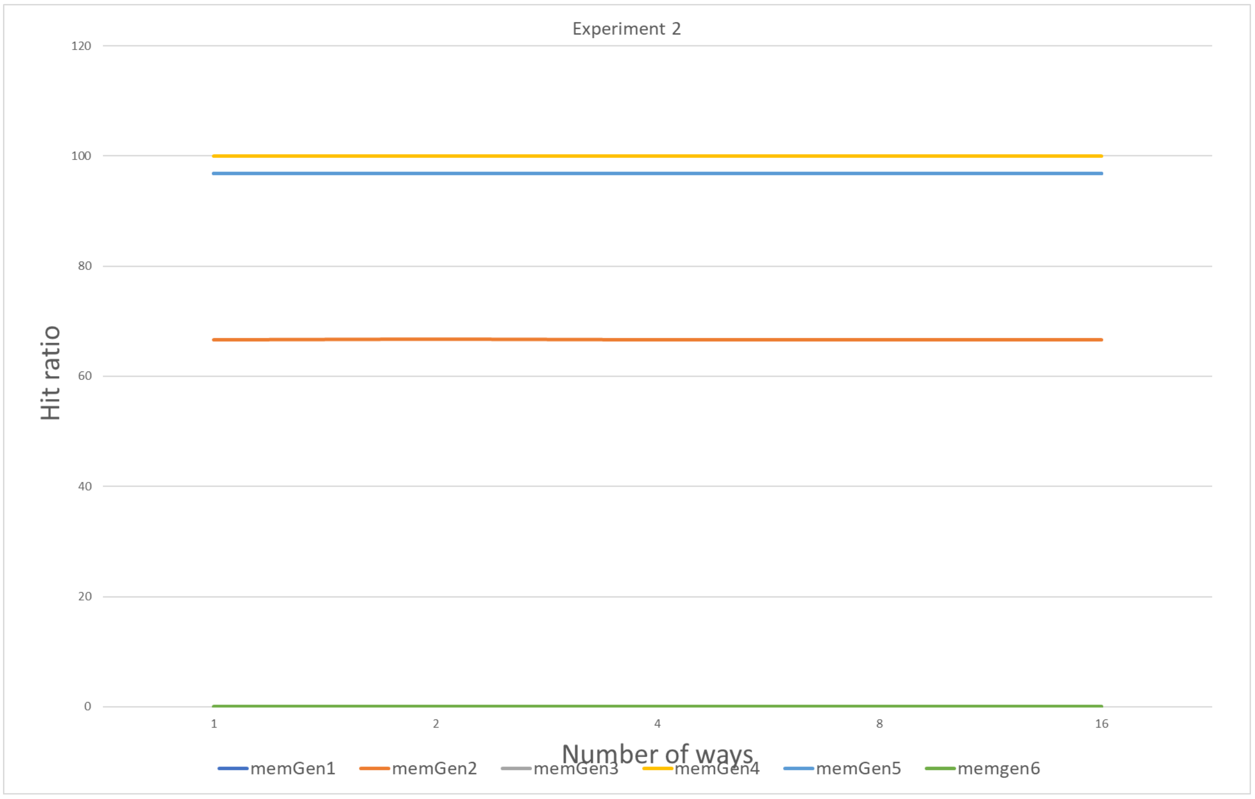

Figure 4. Experiment 2 data representation

Experiment 1:
As shown in Figure(3), the hit ratio remains approximately constant while increasing the cache line size except in memgen functions 1, 5,and 6. In memgen functions 1, 5 and 6, the Hit ratio increases as we increase the cache line size.
In Figure(3), we can also observe that there is a big difference in the hit ratios between the 6 functions given the same number of ways and cache line size. For example, at cache line size of 16 and 32 bytes, the hit ratio for memgen function 6 is approximately 0% and for memgen function 4 is 100%.

Experiment 2:

As shown in Figure(4), the hit ratio remains constant while increasing the number of ways for all the memgen functions. We can also observe that there is a big difference in the hit ratios between the 6 functions given the same number of ways and cache line size. For example,the hit ratio for memgen function 6 is approximately 0% and for memgen function 4 is 100%.

Data Analysis:

Experiment 1:

memGen1() Function: The address in this function increases by one, so by increasing the number of bytes in the cache line size the miss ratio will decrease. For instance, when the cache line size was 16 bytes, the first byte will be a miss and the other 15 bytes will be a hit. Therefore, the miss ratio will be 1/16. However, when the cache line size was 128, the first byte will be a miss but the other 127 bytes will be hit.Thus, the miss ratio will be 1/128. Considering that the number of instructions is 1,000,000 because if the number of instructions is greater than 64 MB, this will not be the case.

memGen2() Function: The memGen() function generates random addresses within a range of 0 to 24 KB. Since the addresses are random, there is no specific pattern or spatial locality in the memory accesses. As a result, the hit ratio tends to remain relatively constant across different cache configurations.

memGen3() Function: The memGen() function generates random addresses within a range of 0 to 64 MB. Since the addresses are random, and there is no specific pattern or spatial locality in the memory accesses, the probability that more than one address to be in the same cache line will be extremely low. As a result, the hit ratio tends to remain relatively constant and ranges from 0.024% to 0.027%.
Although memGen2() function and memGen3() function has the same random algorithm to generate the address, the hit ratio for memGen3() is very low compared to memGen2(). This is because the range of values of memGen3() function which is from 0 to 64 MB is larger than that of memGen2() which is from 0 to 24 KB. Hence, the probability that more than one address to be in the same cache line will be higher in memGen2() compared to memGne3().

memGen4() Function: The addresses in this function will be repeated every 4096 (4*1024) iteration and since the cache size is 16KB so all the address will be in a part of the cache. After this 4096 iterations, the hit ratio will be very high and there will be no misses for all the cache line sizes. However, the miss ratio will differ from cache line size to another in the first 4096 iterations due to the change in the cache line size.

memGen5() Function: memGen5 has the same behavior as memGne 4() but memGne5() generates memory addresses using the modulo operation (addr++) % (1024 * 64). This generates addresses within a larger range of values (0 to 1024 * 64 - 1). Thus, the chance of generating addresses that fall outside the cache's capacity increases, leading to a lower hit ratio.
The hit ratio in memGen4 is higher because the generated addresses fall within a smaller memory range, making them more likely to hit in the cache, while the hit ratio in memGen5 is lower because the generated addresses span a larger memory range, increasing the chances of cache misses.

memGen6() Function: The address in this function increases by 32 so for the cache line sizes of 16 bytes and 32 bytes, there will be a new index for a new set, so no two address will have the same index and if they have the same index there will be difference in their tags. However, for the cache line sizes of 64, the index and the tag of two consecutive address will be same. Thus, one of them will be a miss and the other will be hit. Thus, the hit ratio will be ½. Moreover, when cache line size is 128, the indexes and the tags of four consecutive addresses will be the same. Thus, the first address wil be miss, and the other three will be hit. Thus, the miss ratio will be ¼.

Experiment 2:

For all memgen() functions:
The hit ratio tends to remain relatively constant due to the fixed cache size. Even if there are some improvements, they are generally minimal. For instance, in memgen3, increasing the number of ways from 1 to two ways resulted in a slight increase in the hit ratio from 0.0276% to 0.249%. However, this improvement is negligible. The reason for this is that when we increase the number of ways, the occurrence of misses caused by different tags decreases. This is because we have multiple cache lines with the same index but different tags. However, this also leads to an increase in the number of addresses mapped to the same set, ultimately resulting in a higher number of misses.

memgen6():
The hit ratio is zero due to the lack of spatial locality. In the memgen6() function, the address is consistently incremented by 32 bytes. However, the cache line size remains constant at 32 bytes. Consequently, upon the initial memory access, a cache miss occurs, and the specific byte along with the subsequent 31 bytes are brought into the cache. Regrettably, these 31 bytes go unused since the subsequent address is accessed, which is located 32 bytes away from the previous address. This process repeats, rendering the cache line size ineffective as the

accessed data is non-consecutive. Furthermore, temporary locality is absent since we do not access the same address multiple times (each time accessing the previous address plus 32 bytes). Even if we were to access it, it would be replaced before subsequent access due to the small cache size of 16*1024 bytes.

memgen3():
The hit ratio is very low (between 0.024% and 0.025%) and this is because we do not have spatial locality because the data is accessed randomly and the range is very wild so the probability to access two neighbor addresses is very low. In addition to this, being random decreases the probability to access the same data many times.

memgen2():
The hit ratio is good (between 66.6% and 66.7%) which is better than memgen3(), and this because its range is much smaller than memgen3() (24*1024 instead of 64*1024*1024) which increases the probability of accessing the same address or neighbor addresses many times. Therefore, even if it is random, its range is much smaller and this what leads to increase the hit ratio)

memgen1():
The hit ratio for this function is higher than that for the aforementioned functions, and this is because we are accessing the addresses consecutively. Therefore, it is very similar to accessing a very large array or some consecutive arrays. In short, it will result into miss when we put data in a cache line then it will result into 31 consecutive hits ((31/32)*100=96.8%), and we will repeat that till we finish all needed accessing of the memory.

memgen5():
It is very similar to memgen1() but the addresses ranges is smaller, but here this will not affect the hit ratio because it have consecutive accessing similar to memgen1() (it will result into miss when we put data in a cache line then it will result into 31 consecutive hits ((31/32)*100=96.8%)), and also the range of the address is higher than the capacity of the cache so if we want to access the same group of data again we will repeat the same process(put then in a cache line which leads to a miss then 31 hits).

memgen4():
It is very similar to memgen1() and memgen4(), but its range is much smaller, and smaller than the cache size. Therefore, it will have similar behavior to the aforementioned examples, but after loading all data it will not need to reload the data if it will use them again because the range of addresses is less than the size of data, so it will lead to 32 consecutive hits if it accessed the same data again because the data is already in the cache. The scenario will be 1 miss 31 hits then 1 miss 31 hits till we upload all, the data in the this range (4 * 1024). After this stage, it will have only hits which will increase the hits ratio to be more than 96.875%, and this percentage will be 99.9872%.

Conclusion:

Through experiments, we analyzed different memory access patterns and observed how cache configurations influenced performance. The results highlighted the importance of spatial and temporal locality in improving cache hit ratios. In the case where the cache size is 16 kbytes, we concluded that the best performance comes from using a cache line size of 128 bytes, regardless of the number of ways used.