CSCE 231/2303 - Computer Organization and Assembly Language Programming

Fully Associative and Direct Mapped Cache Simulator

Summer 2024

Dr. Mohamed Shalan

Aly Elaswad (900225517)

Islam Abdeen (900225835)

Seif Elansary (900221511)

Project introduction:

In Random access memory (RAM), memory is organized so that data can be transferred to or from any cell in a time independent of the selected cell. There are two types of Random access memories which are DRAM and SRAM. DRAM is cheaper and slower and can be used to build large memory. SRAM is more expensive and cannot be used to build large memory, but it is significantly faster than DRAM. Therefore, the concept of caches Processors execute operations much faster with registers than with large main memory (DRAM), which is slower despite SRAM being faster. Fully replacing main memory with SRAM is expensive. Introducing a small SRAM buffer, known as a cache, between the main memory (DRAM) and the processor helps solve this issue. Due to the small size of the cache, only part of the data in the Main memory can be stored in cache; therefore, a memory hierarchy is created so that the memory is large and fast most of the time. To achieve this, the concept of locality is utilized where the cache contains a copy of a subset of the main memory. There are two types of localities: Spatial locality and temporal locality. Temporal locality refers to the situation in which items accessed recently are likely to be accessed again soon. Spatial locality refers to the situation in which items near those accessed recently are likely to be accessed soon. To increase the probability of finding the data in the cache (which is called the hit rate), each time data is not found in the cache (which is called a miss), a whole line is extracted from the main memory. This line typically contains more than one element. This helps make use of the spatial locality, as the next time the following address is accessed, some hits are probably achieved. To implement this, there are 3 types of caches, which are direct-mapped, fully associative and set associative. The direct mapped cache maps multiple addresses to a specific line in the cache depending on the index. This causes the direct mapped cache to have conflict misses, and this happens when several pieces of data map to the same cache line. Therefore, some of them are evicted even though there is space in other parts of the cache. The advantages of the direct mapped cache include its simplicity and its low cost. Another type of cache is the fully associative cache which searches for data by comparing each bit of the tag excluding the offset with the tag in the cache. This allows flexibility where data can be stored in any cache line. As a result, there are no conflict misses. However, there are capacity misses which occur when the cache is full and the data searched for is not present in the cache. Despite that the hit ratio is generally higher than the direct mapped cache, fully associative caches are expensive, as they require more hardware. The third type is the set associative cache, which is generally used. Its advantage lies in the fact that it requires minimal hardware but achieves a hit rate that is close to the fully associative cache, as several data is mapped into sets which include n lines. This project's aim is to simulate the fully associative cache and the direct mapped cache. The programming language used was C++.

Testing the simulator:

To test our simulators, we ran some test cases that did not have a random factor in them, to be able to trace the correctness of our simulator. Below are some test cases and brief breakdown of their results:

Our first test case was the ***memgenT1()*** function, which keeps the only referenced address at 0x000000. Such sequences of memory addresses are expected to have a single cold start cache miss, but then the rest are hits. Our simulator passed this test case and gave a hit rate of 99.9999%. This test case tackles one of the cache's main goals in temporal locality where the same memory index was accessed over and over again.

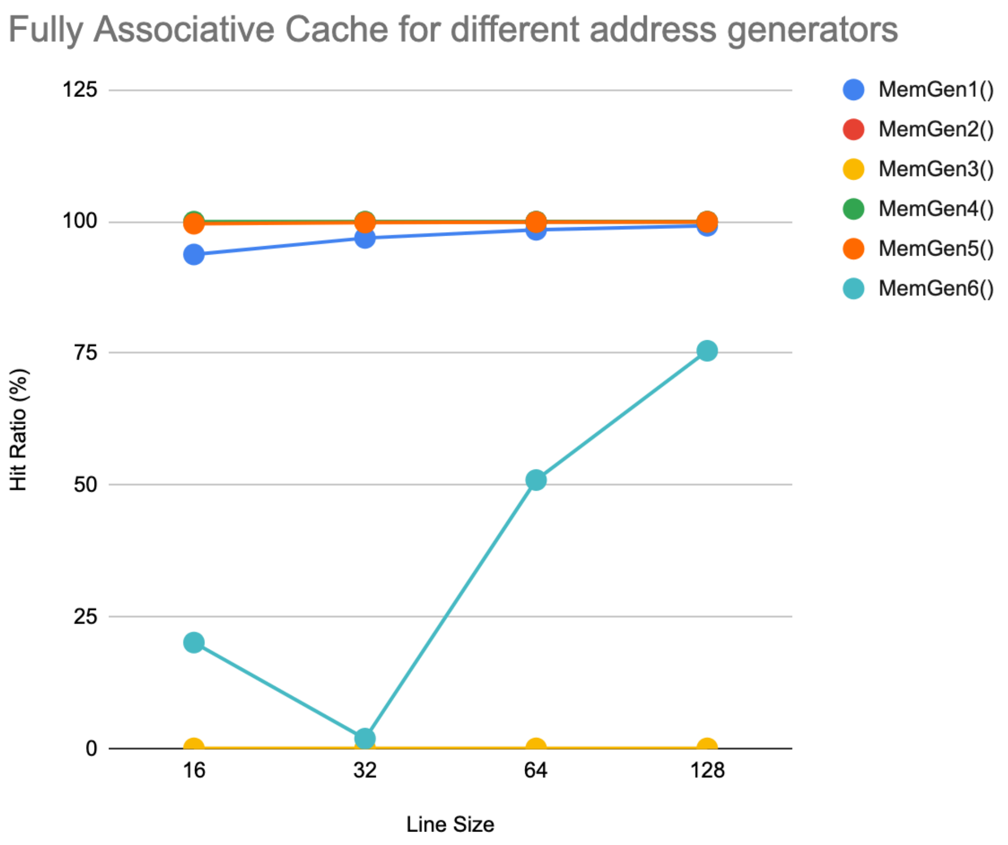
Our second test case was the ***memgenExp()*** function, to apply the tests of this function, we reduced the number of iterations to 25 (to avoid overflow since we only have a 32-bit address), the goal of this function was to test the different line sizes on a reasonable niche of the addresses that we are able to trace. For example, when given a line size of 16 bytes in FA/DM caches the hit rate was (8%), but given a line size of 32 bytes, the cache hit rate was higher (12%), this is correct since we are able to prefetch more bytes to be addressed in the future giving us less cold start misses and increasing the hit rate.

Our third test case was the ***memgenAlt()*** function, this function uses addresses that are far away from each other, which ignores the concept of spatial locality. Consequently, this was confirmed in our results, since the hit rate was 0% in both the FA and the DM caches. The reason this happens is because each iteration we either add 1024 or subtract 512 alternatingly to the previous address, so we will never have nearby addresses indexed in sequence.

Our fourth test case was the ***memgenCyclic()*** function, which keeps cycling addresses in a small portion of the DRAM. As for the results, the trend was that bigger line sizes resulted in higher hit rates, since we access nearby addresses frequently, so when a line size is bigger, then more bytes are to be prefetched resulting in higher hits rates. Since the addresses are cycled in a small portion of the dram, higher line sizes did not necessarily cause any capacity misses.
Our fifth test case used the ***memGenInterleaved()*** function, which alternates between two addresses, incrementing each by 1024 bytes. For the fully associative cache, there were higher hit rates with larger line sizes. The pattern's spatial locality benefits from larger lines as nearby addresses are cached together. As for the direct-mapped cache, there was a 0% hit ratio due to conflict misses, this is because each address maps to a specific cache line. If addrA and addrB map to the same cache line, every access to one address will evict the other from the cache.

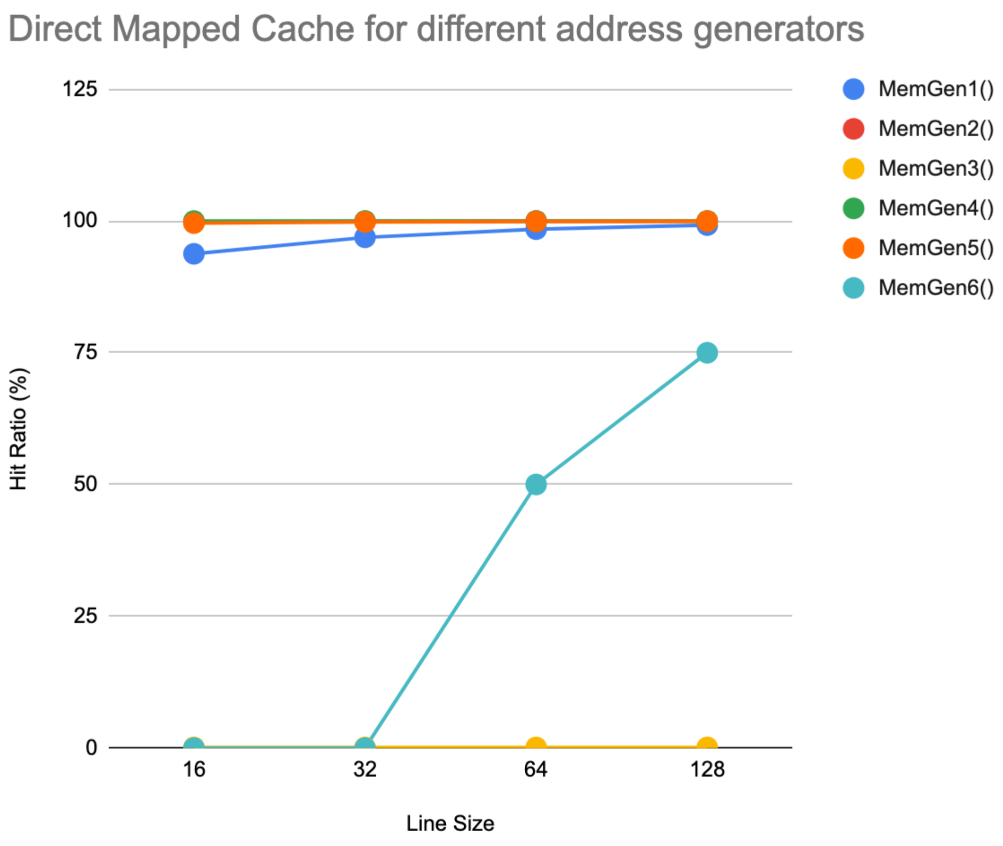
<u>Fully associative cache experiment results:</u>

Line size				
MemGen	16	32	64	128
MemGen1()	93.75	96.875	98.4375	99.2187
MemGen2()	99.8464	99.9232	99.9616	99.9808
MemGen3()	0.1001	0.097	0.0968	0.0969
MemGen4()	99.9744	99.9872	99.9936	99.9968
MemGen5()	99.588	99.7928	99.895	99.9458
MemGen6()	20.1558	1.9535	50.9871	75.4861



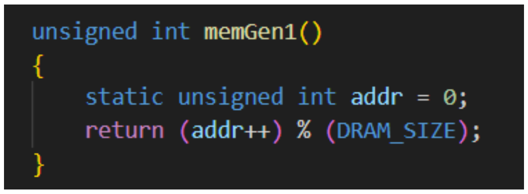
Direct-mapped cache experiment results:

	Line size			
MemGen	16	32	64	128
MemGen1()	93.75	96.875	98.4375	99.2187
MemGen2()	99.8464	99.9232	99.9616	99.9808
MemGen3()	0.1021	0.1021	0.1023	0.0978
MemGen4()	99.9744	99.9872	99.9936	99.9968
MemGen5()	99.5904	99.7952	99.8976	99.9488
MemGen6()	0	0	49.9999	74.9999



Analysis of results on different memory generators:

MemGen1:



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

As shown in the image, the address starts from 0 and is incremented with respect to the DRAM size (Main Memory). Subsequently, this will take full advantage of the spatial locality phenomenon, which will result in a large amount of hits, depending on the block size. For example, when the block size is 16 bytes. The first search will result in a miss (cold start); however, this will lead to the next 15 searches being hits, giving us a hit ratio of 15/16, which is 93.75% as shown by the data. Intuitively, using a larger block size will increase the number of hits compared to misses, with the ratio depending on the size.

16 Bytes: 15/16 * 100 = 93.75%

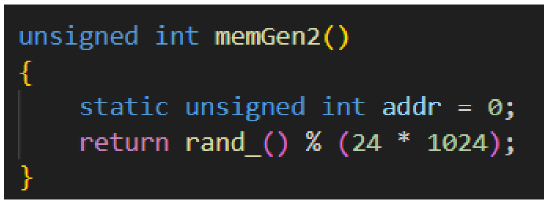
32 Bytes: 31/32 * 100 =96.875%

64 Bytes: 63/64 * 100 =98.4375%

128 Bytes: 127/128 * 100 =99.2187%

Corroborated by the data gathered.

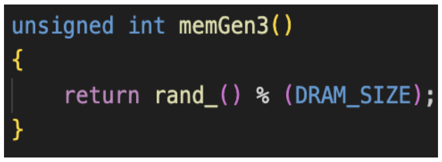
MemGen2:



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

As shown in the image, the address is randomly generated in a small subset of the DRAM (24 Kbytes). Fortunately, the DRAM subset is smaller than the size of the cache, which means that the entirety of the data can fit into the cache. Initially, there will be some cold start misses, but then, most of the searches will yield to a hit giving us a pretty high hit rate as shown in the table. This is also a strong example of spatial locality and temporal locality at place. Increasing the block size will decrease the number of block lines, leading to a lower amount of cold start misses and thus a higher hit rate. It is safe to assume that given that the program size is smaller than the cache size and the program is repeated numerous times, it will lead to very high hit rates.

MemGen3():

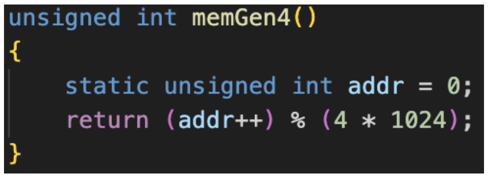


```
unsigned int memGen3()
{
    return rand_() % (DRAM_SIZE);
}
```

The memgen3() function just generates a random number within the size of the DRAM, which is 64 Mbytes. Such sequence of memory addresses does not make use of the temporal locality since by generating random addresses, there is a relatively slim chance that the same address will be generated again. Similarly, it does not make use of the spatial locality, since there is no particular sequence in which the memory addresses are accessed, it is randomly generated so elements in proximity are not necessarily going to be called frequently. Spatial and temporal locality are one of the main motives of implementing cache designs, so by not making use of them, the cache would not be as useful. This can be seen in the experiments' results where the hit rate did not exceed 0.2% for both the direct-mapped cache and the fully associative cache. The hit rate is not exactly 0% because we might randomly get elements that are mapped to the same block (occasional hits).

MemGen4():

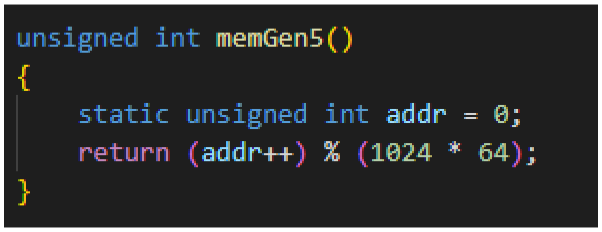
This function generates addresses in a sequential manner and wraps around after reaching 4KB. This sequential addressing means that once an address is accessed, the next few addresses are to be accessed after, making spatial locality very much in use. The cache will prefetch and hold adjacent addresses as they are to be accessed soon after the current address. As for Spatial locality, we have that since the cache size is way bigger (64 KB) than the 4KB in which this function wraps around. The data is not to be overwritten (reducing capacity or conflict misses), so we will access the same memory address more than once, making good use of temporal locality. Since spatial locality and temporal locality are highly used in this benchmark, the hit rates were very high (>99%), for both the direct-mapped cache as well as the fully associative cache.



```
unsigned int memGen4()
{
    static unsigned int addr = 0;
    return (addr++) % (4 * 1024);
}
```

MemGen5():

The memGen5 function generates sequential memory addresses from 0 to 65535, cycling through this range. Compared to memgen4, this function has a little more cold start misses, as this function wraps around a number greater than the number memgen4 wraps around.This sequential access pattern benefits from high temporal and spatial locality, as recently accessed data is likely to be accessed again soon, and consecutive addresses are accessed in order. Therefore, both the hit rate of both fully associative cache and direct mapped cache will be high. The few misses that arise are cold start misses. The hit rate also increases upon increasing the line size, as the spatial locality is improved further, and the cold start misses decrease. This is apparent in the following hit rates:

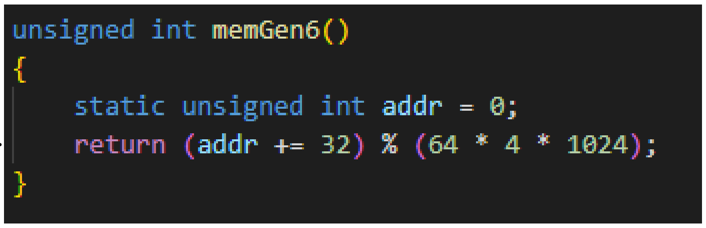


```
unsigned int memGen5()
{
    static unsigned int addr = 0;
    return (addr++) % (1024 * 64);
}
```

99.588 for 16-byte line
99.7928 for 32-byte line
99.895 for 64-byte line
99.9458 for 128-byte line

MemGen6():

This function generates addresses that are 32 bytes apart. The modulo operation (addr += 32) % (64 * 4 * 1024) ensures that the generated addresses wrap around after reaching (64 * 4 * 1024) bytes, which equals 256 Kbytes. In the direct mapped cache, each block of main memory maps to exactly one cache line. Because each address is 32 bytes apart, if the line size is 32 bytes or less, spatial locality will not be utilized and there will be no hits. Using 64-byte line size and 128-byte line size would utilize the spatial locality better, with the 128-byte-line size being better, as the hits in each line increase. The following data clearly shows that:



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

0 for 16-byte line
0 for 32-byte line
49.9999 for 64-byte line
74.9999 for 128-byte line

For the fully associative cache, the flexibility to place any block in any line reduces misses; however, as explained above, the nature of the function makes the hit ratio low if the line size is smaller or equal to 32 bytes. When using 64-byte lines, we end up with a pattern of a miss and then a hit, so the hit rate is close to 50%. When using 128-byte lines, we end up with a pattern of a miss and then 3 hits, so the hit rate is close to 75%.

Below is a breakdown of the results and the reasoning behind them:

16 Bytes: Each address increment (32 bytes) spans 2 cache lines (0 and 1, 2 and 3, etc.). This means the cache lines are effectively used, resulting in a reasonable hit rate (20.1558%).

32 Bytes: Each address increment (32 bytes) lands exactly on the next cache line. This causes the previous line to be evicted before it is reused, leading to a very low hit rate (1.9535%).

64 Bytes: Each address increment (32 bytes) remains within the same cache line (each line covers 64 bytes), leading to better utilization of each cache line and higher hit rate (50.9871%).

128 Bytes: Each address increment (32 bytes) remains within the same cache line, and the lines cover larger blocks of memory, leading to an even higher hit rate (75.4861%).

In a fully associative cache with random replacement, even though new addresses may cause evictions before the wrap-around, some cache lines that have been filled and not yet evicted will result in hits when accessed again in the other wraps. This randomness means that occasionally, addresses that were recently accessed may still be in the cache.

Conclusion

- Caches are ideal when the given program uses the principles of spatial locality and temporal locality and produces high hit rates for these programs, decreasing the average memory access time (AMAT).
- One limitation of the cache designs is that it is inefficient in handling access patterns with minimal spatial and temporal locality.
- The fully associative cache produces higher hit rates than direct-mapped cache in specific programs in which multiple memory addresses are to be mapped to the same index.
- If there are low conflict misses, then the direct-mapped cache is a better option since it makes the hardware simpler and does not require traversing the entire cache to determine whether a certain address is a hit or a miss
- For most memory access patterns, larger cache line sizes result in higher hit rates as long as it does not lead to more frequent evictions, hence why it is a trade-off between number of lines and bytes per line.