Cache Project 2 Report

Abdullah Elnahhas, Dalia Eissa, Omar Zainelabideen.
The American University in Cairo
CSCE 2303: Computer Organization & Assembly Language Programming
Dr. Mohamed Shalan
Summer 2024
22nd of July 2024

1. Introduction

In this project, we aim to explore the functioning of caches and examine how different cache parameters influence performance. We will develop a Direct-mapped simulator and Fully-associative simulator and validate its correctness. The simulator will then be used to conduct various experiments, and the results will be analyzed. We will study 6 different memory access patterns that we will be naming memGen1 to memGen6.

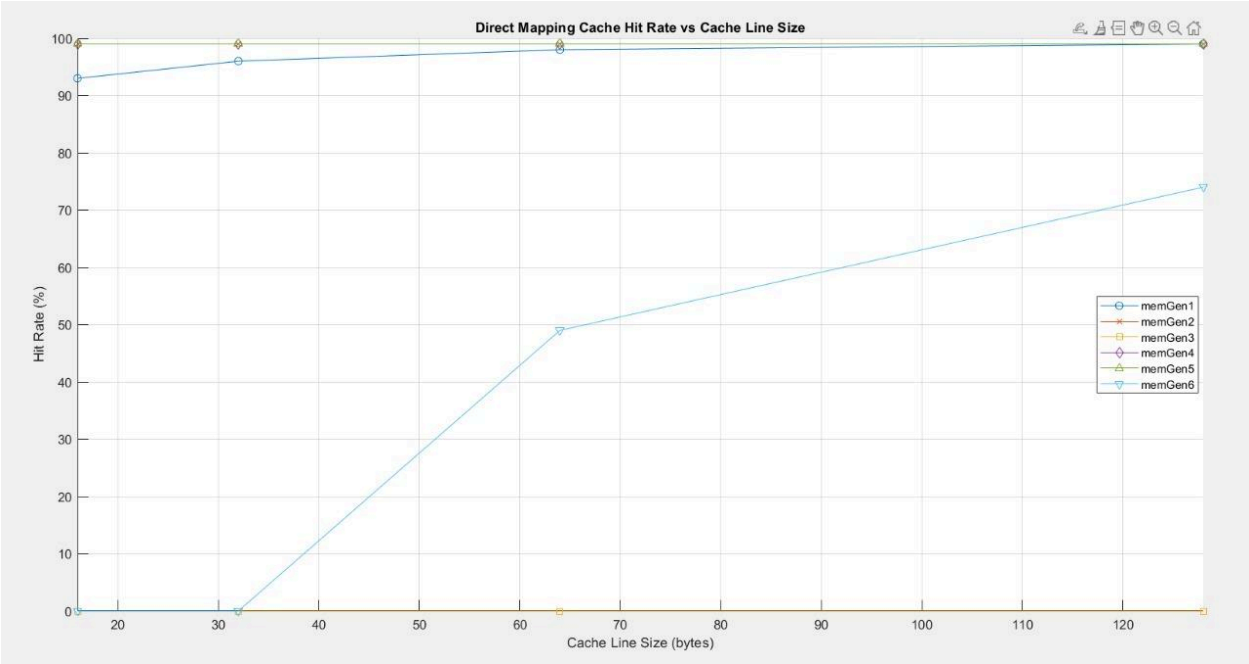
2. MemGen Functions

```
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);
```

}

3. Results

3.1 Direct Mapping Results



I. Memgen1:

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

Here in memgen1 the address is incremented by 1.
64Kbytes = 65536 bytes.
DRAM size is 64*1024*1024 =67108864 bytes.
The equation ((addr++)%(DRAM_SIZE)) generates the addresses sequentially until it reaches the DRAM size, then regenerates the addresses from the beginning.

In this generator a certain pattern occurs as:

- We fetch the first address, address 0, resulting in a miss since the cache is empty (cold start miss). The cache then fetches 16 bytes from memory, covering bytes 0-15.
- The next address, 1, is already in the cache because it is within the fetched range, so this is a hit.
- This continues for addresses 2 through 15, all resulting in hits.
- When we reach address 16, it results in a miss because it is outside the current range of bytes 0-15. The cache then fetches the next 16 bytes, covering bytes 16-31.
- This pattern continues with each new range of 16 bytes: 1 miss followed by 15 hits.

a) 16 bytes line size:
Number of lines = 65536/16 = 4096.
This gives us the hit ratio of 15/16 = 0.938.
Therefore, for the first 4096 memory accesses there will be 4096 cold start misses (compulsory misses) because the cache is initially empty. Then the next address accesses if the addresses (tag) don't match then it would be a conflict miss, however if they do match then that would be a hit. Hence, the 2nd 4096 memory accesses are conflict misses. The tags are the same up until 65536 bytes.

b) 32 bytes line size:
Number of lines = 65536/32 = 2048.
This gives us the hit ratio of 31/32 = 0.96875.
The first 2048 memory accesses will cause 2048 cold start misses (compulsory misses) because the cache is initially empty. Then the next address accesses if the addresses (tag) don't match then it would be a conflict miss, however if they do match then that would be a hit. Hence, the 2nd 2048 memory accesses are conflict misses. The tags are the same up until 65536 bytes.

c) 64 bytes line size:
Number of lines = 65536/64 = 1024.
This gives us the hit ratio of 63/64 = 0.984375.
The first 1024 memory accesses will cause 1024 cold start misses (compulsory misses) because the cache is initially empty. Then the next address accesses if the addresses (tag) don't match then it would be a conflict miss, however if they do match then that would be a hit. Hence, the 2nd 1024 memory accesses are conflict misses. The tags are the same up until 65536 bytes.

d) 128 bytes line size:
Number of lines = 65536/128 = 512.
This gives us the hit ratio of 127/128 = 0.9921875.
The first 512 memory accesses will cause 512 cold start misses (compulsory misses) because the cache is initially empty. Then the next address accesses if the addresses (tag) don't match then it would be a conflict miss, however if they do match then that would be

a hit. Hence, the 2nd 512 memory accesses are conflict misses. The tags are the same up until 65536 bytes.

As we can see as the line size increases, and the number of lines decreases, the number of hits increases. That is because larger cache line size means more data is covered and so it takes advantage of spatial locality, as data being accessed are in nearby locations.

II. Memgen2:

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

Memgen2 generates totally random addresses within a smaller range which is from 0 to 24 KB. 64Kbytes = 65536 bytes.

This application does not use the locality function of the cache as it is not sequential. But the reason the MemGen2 has a high hit rate is because of the size. The total cache size is 64KB while the generator is only 24 KB so the cache can hold the entire address space many times. Once the cache is all in, almost every subsequent access hits because the address space is fully cached. So in a 1000000 iterations only the first number of unique addresses miss then all of them hit for example:

Line Size: 16 Bytes

- Number of Cache Lines:

$$\frac{64\ KB}{16\ B} = 4096 \text{ lines}$$

- Number of Unique Lines in Address Space:

$$\frac{24\ KB}{16\ B} = 1536 \text{ lines}$$

So The cache can hold all 1536 unique lines easily within its 4096 lines capacity.

Line Size: 32 Bytes :

- Number of Cache Lines:

$$\frac{64\ KB}{32\ B} = 2048 \text{ lines}$$

- Number of Unique Lines in Address Space:

$$\frac{24\ KB}{32\ B} = 768 \text{ lines}$$

So The cache can hold all 768 unique lines easily within its 2048 lines capacity. And so on the same method in the rest of the sizes.

III. Memgen3:

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

Memgen3 generates totally random addresses within the range from 0 to the DRAM size. 64Kbytes = 65536 bytes.
The equation (rand_()%(DRAM_SIZE)) generates the addresses randomly within the size of the DRAM size.

Both memgen2 and memgen3 can be considered as applications that do not take advantage of locality, meaning they do not repeatedly access nearby memory locations. Given this context, the expected hit ratio in memGen3 will be significantly lower than in memGen2 due to the much larger range of addresses.

Line Size: 16 Bytes

- Number of Cache Lines:

$$\frac{64\ kB}{16\ B} = 4096 \text{ lines}$$

- Number of Unique Lines in Address Space:

$$\frac{64\ MB}{16\ B} = 4194304 \text{ lines}$$

So Given the vast number of unique lines (4,194,304) compared to the number of cache lines (4,096), the cache will not be able to store all unique addresses. Therefore, the hit ratio is expected to be lower compared to memGen2, with frequent cache misses due to the large number of possible addresses.

$$\text{Hit Ratio} = \frac{4096}{4194304} = 0.000976 \text{ percent}$$

Line Size: 32 Bytes

- Number of Cache Lines:

$$\frac{64\ KB}{32\ B} = 2048 \text{ lines}$$

- Number of Unique Lines in Address Space:

$$\frac{64\ MB}{16\ B} = 2097152 \text{ lines}$$

So Given the vast number of unique lines (2,097,152) compared to the number of cache lines (2,048), the cache will not be able to store all unique addresses. Therefore, the hit ratio is expected to be lower compared to memGen2, with frequent cache misses due to the large number of possible addresses.

$$\text{Hit Ratio} = \frac{2048}{2097152} = 0.000976 \text{ percent}$$

And so on in the rest of the sizes with the same percentage

IV. Memgen4:

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

Memgen4 generates sequential addresses within a range which is from 0 to 4 KB.
This means that the addresses being accessed are more likely to fall within the cache's capacity. It also has a high amount of temporal locality since it generates memory addresses sequentially in a cyclic manner.

For Each Line Size:

16 Bytes: Cache can hold 4096 lines; unique lines needed are 256.

32 Bytes: Cache can hold 2048 lines; unique lines needed are 128.

64 Bytes: Cache can hold 1024 lines; unique lines needed are 64.

128 Bytes: Cache can hold 512 lines; unique lines needed are 32.

So the hit rate will be high and close to 100 % .

V. Memgen5:

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

Memgen5 generates sequential addresses within a range which is from 0 to 64 KB.

VI. Memgen6 :

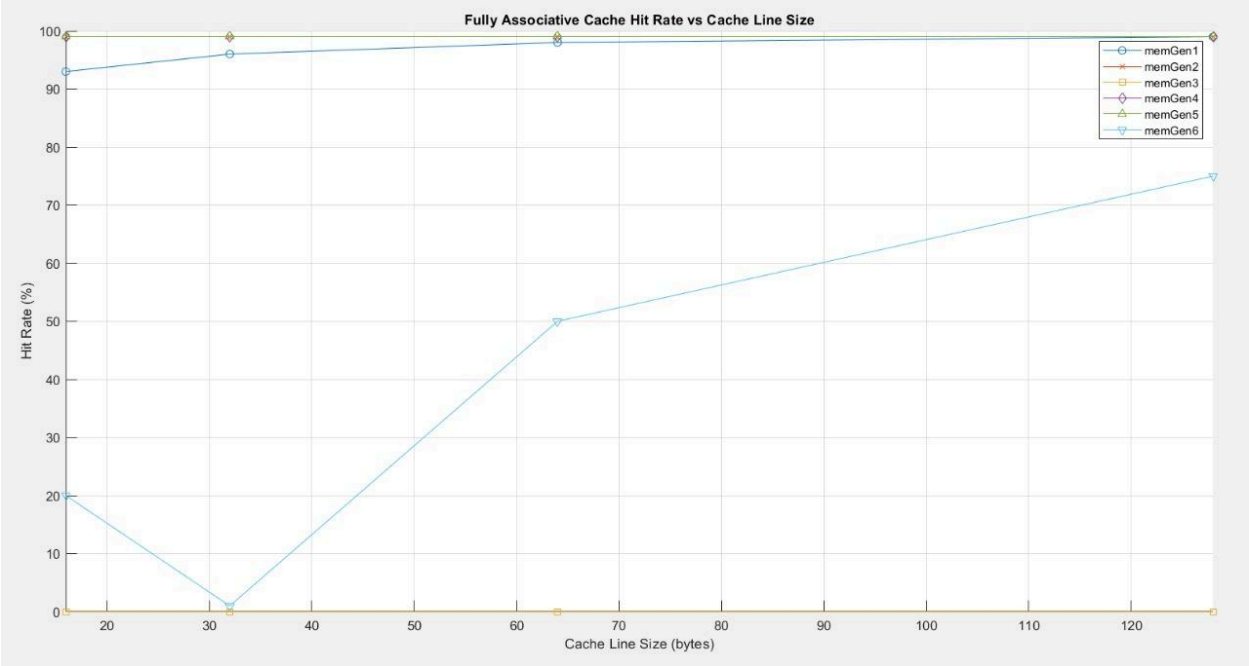
```
unsigned int memGen6()
{
        static unsigned int addr=0;
        return (addr+=32)%(64*4*1024);
```

}

Memgen6 generates addresses within a range which is from 0 to 256 KB with an increment of 32. Memgen6 is not sequential change of addresses so the hit rate varies because the addresses are not increasing in sequential order. The hit rate is 0 until 32 bytes line size because the address is being incremented by 32 instead of being sequentially incremented, which causes the cache to never make use of spacial locality. As the line size increases the cache makes more use of spacial locality and so it gradually increases but it is doesn't reach the efficiency of other memGen's.

As we can see as the line size increases, and the number of lines decreases, the number of hits increases. That is because larger cache line size means more data is covered and so it takes advantage of spatial locality, as data being accessed are in nearby locations. However, this depends on the memory access pattern as well. For example in memGen6 the hit rate is 0 at the start as the line size is very small and as the line size increases the cache makes more use of the spacial locality.

3.2 Fully-Associative Results



I. 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. In this generator a certain pattern occurs as:

- We fetch the first address, address 0, resulting in a miss since the cache is empty (cold start miss). The cache then fetches 16 bytes from memory, covering bytes 0-15.

- The next address, 1, is already in the cache because it is within the fetched range, so this is a hit.
- This continues for addresses 2 through 15, all resulting in hits.
- When we reach address 16, it results in a miss because it is outside the current range of bytes 0-15. The cache then fetches the next 16 bytes, covering bytes 16-31.
- This pattern continues with each new range of 16 bytes: 1 miss followed by 15 hits.

This gives us the hit ratio of 15/16 = 0.938 at 16 bites

This gives us the hit ratio of 31/32 = 0.96875 at 32bites

This gives us the hit ratio of 63/64 = 0.984375 at 64 bites

This gives us the hit ratio of 127/128 = 0.9921875 at 128 bites

This gives the equation of:
hit ratio = 1- line size/ line size

II. MemGen2

Memgen2 generates totally random addresses within a smaller range which is from 0 to 24 KB. This application does not use the locality function of the cache as it is not sequential. But the reason the MemGen2 has a hit rate is because of the size. The total cache size is 64KB while the generator is only 24 KB so the cache can hold the entire address space many times. Once the cache is all in, almost every subsequent access hits because the address space is fully cached. So in a 1000000 iterations only the first number of unique addresses miss then all of them hit for example:

Line Size: 16 Bytes

- Number of Cache Lines:

$$\frac{64\ KB}{16\ B} = 4096 \text{ lines}$$

- Number of Unique Lines in Address Space:

$$\frac{24\ KB}{16\ B} = 1536 \text{ lines}$$

So The cache can hold all 1536 unique lines easily within its 4096 lines capacity.

Line Size: 32 Bytes :

- Number of Cache Lines:

$$\frac{64\ KB}{32\ B} = 2048 \text{ lines}$$

- Number of Unique Lines in Address Space:

$$\frac{24\ KB}{32\ B} = 768 \text{ lines}$$

So The cache can hold all 768 unique lines easily within its 2048 lines capacity.

And so on the same method in the rest of the sizes

III. MemGen3

In MemGen3, we use the same rand() function as in MemGen2, but with a key difference: memGen3 generates addresses ranging from 0 to 64 MB, while MemGen2 generates addresses between 0 and 24 KB. Both setups can be considered as applications that do not take advantage of locality, meaning they do not repeatedly access nearby memory locations. Given this context, the expected hit ratio in memGen3 will be significantly lower than in memGen2 due to the much larger range of addresses.

Line Size: 16 Bytes

- Number of Cache Lines:

$$\frac{64\ kB}{16\ B} = 4096 \text{ lines}$$

- Number of Unique Lines in Address Space:

$$\frac{64\ MB}{16\ B} = 4194304 \text{ lines}$$

So Given the vast number of unique lines (4,194,304) compared to the number of cache lines (4,096), the cache will not be able to store all unique addresses. Therefore, the hit ratio is expected to be lower compared to memGen2, with frequent cache misses due to the large number of possible addresses.

$$\text{Hit Ratio} = \frac{4096}{4194304} = 0.000976 \text{ percent}$$

Line Size: 32 Bytes

- Number of Cache Lines:

$$\frac{64\ KB}{32\ B} = 2048 \text{ lines}$$

- Number of Unique Lines in Address Space:

$$\frac{64\ MB}{16\ B} = 2097152 \text{ lines}$$

So Given the vast number of unique lines (2,097,152) compared to the number of cache lines (2,048), the cache will not be able to store all unique addresses. Therefore, the hit ratio is expected to be lower compared to memGen2, with frequent cache misses due to the large number of possible addresses.

$$\text{Hit Ratio} = \frac{2048}{2097152} = 0.000976 \text{ percent}$$

And so on in the rest of the sizes with the same percentage

IV. MemGen4

MemGen4 is similar to MemGen1 but it operates within a much smaller address range of 4KB. This means that the addresses being accessed are more likely to fall within the cache's capacity. It also has a high amount of temporal locality since it generates memory addresses sequentially in a cyclic manner.

For Each Line Size:

- 16 Bytes: Cache can hold 4096 lines; unique lines needed are 256.
- 32 Bytes: Cache can hold 2048 lines; unique lines needed are 128.
- 64 Bytes: Cache can hold 1024 lines; unique lines needed are 64.
- 128 Bytes: Cache can hold 512 lines; unique lines needed are 32.

So the hit rate will be high and close to 100 % .

V. MemGen5

MemGen4 is similar to MemGen1 but it operates within a much smaller address range of 64KB. This means that the addresses being accessed are not as likely to fall within the cache's capacity as MemGen4. For memGen5, given the address range is 64 KB and assuming a fully-associative cache with a size of 64 KB, the cache can hold all unique addresses generated by memGen5. Thus, irrespective of the line size (16, 32, or 64 bytes), the cache can accommodate all possible addresses, leading to a hit ratio of nearly 100%.

VI. MemGen6

Memgen6 generates addresses within a range which is from 0 to 256 KB with an increment of 32. Memgen6 is not sequential change of addresses so the hit rate varies because the addresses are not increasing in sequential order. The hit rate is 0 until 32 bytes line size because the address is being incremented by 32 instead of being sequentially incremented, which causes the cache to never make use of spacial locality. As the line size increases the cache makes more use of spacial locality and so it gradually increases but it is doesn't reach the efficiency of other memGen's.

4. Conclusion

In this report, we investigated cache performance with fully-associative and direct-mapped simulators. We looked at how different cache structures and memory access patterns affected the cache performance. According to our findings, hit rates can be increased by using higher cache line sizes, which take advantage of spatial locality, or by using smaller address ranges, which fit within the cache capacity.