CSCE230301-Computer Organization and Assembly Language Programming

Spring 2025

Submitted by:

Fatima N. Elsayed

Mohamed O. Slama

Data: May 21, 2025

Introduction

We know that there is a performance gap between modern processors and main memory. More specifically, modern processors operate at speeds much faster than main memory. This issue causes a performance bottleneck in memory access processes. To overcome this issue, smaller but faster memory units are introduced. These memory units are called caches. Caches are placed between the central processing unit (CPU) and main memory. They are significant because they make use of both spatial and temporal localities. The cache memory stores frequently accessed data and instructions, which in turn reduces the average access time.

Our project goal is to make a simulation for the set-associative cache. We want to observe how varying the parameters of the cache will affect cache performance. The parameters that act as independent variables are the cache memory line size and the number of ways. Cache performance will be measured as the hit ratio for each configuration. In our project, there are six memory address generator functions. These functions generate different memory addresses that vary in their pattern and configuration. We use the memory address generators for each experiment. After that, we analyze the hit and miss ratios across different configurations to understand their impact on cache performance.

Cache Simulator Design

We designed the simulator to model an n-way set-associative cache with the following attributes:

- **Total Cache Size**: 64 KB (64 × 1024 bytes)
- **Variable Line Sizes**: 16, 32, 64, 128 bytes
- **Variable Associativity (ways)**: 1, 2, 4, 8, 16 way
- **Main Memory Size**: 64 MB (64 × 1024 × 1024 bytes)

Cache structure: The cache is implemented as a vector of sets, where each set contains multiple cache lines (also known as blocks), in addition to some metadata such as the valid bit, the tag, and the replacement information.

Indexing: Each memory address is broken into some fields. These fields are the block offset, which identifies the specific byte in the cache line; the set index, which specifies the cache set the block maps to; and the tag, which distinguishes between different memory blocks that map to the same set. These fields vary depending on the cache line size and the number of sets.

Replacement policy: The simulator uses a round-robin replacement policy, meaning that it behaves as First-In, First-Out (FIFO) within each set. This simple approach allows the implementation to be easier. However, it may not exploit temporal locality as effectively as Least Recently Used (LRU).

Generator Used: In our simulator, there are six generator functions for the memory addresses, namely `memGen1()` to `memGen6()`. Each generator function provides a different address pattern, which simulates different workloads.

This simulator processes 1,000,000 memory addresses per configuration, tracking the hit and miss ratios accordingly.

Experiments and Results

In our simulation, we had two experiments. Each experiment focused on an independent variable (either line size or the number of sets) to see how changing this variable affects the cache memory performance.

Experiment 1: Effect of Line Size

For this experiment, we fixed the number of sets to 4 and varied the size of the cache line between 16, 32, 64, and 128 bytes. The number of ways was calculated accordingly to keep the total cache size constant (64 KB). If we put the line size to be L (measured in bytes), and the number of sts of be S (which is fixed in this experiment, and is equal to 4), and finally the number of ways to be W (which is what we need to compute), we can use the following equation:

$$CacheSize = ways * Sets * LineSize = W * S * L$$

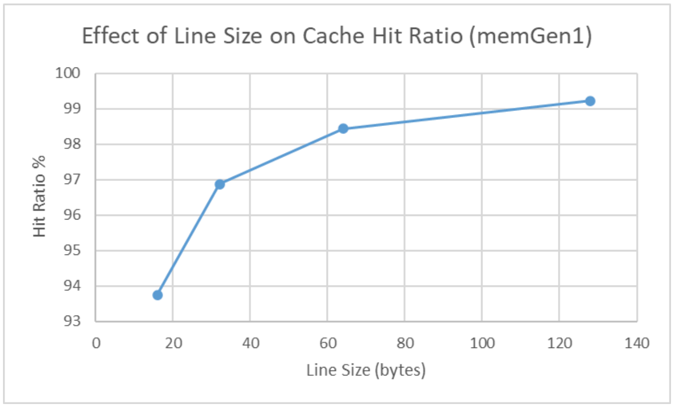
Solving for the ways we can see that:

$$Ways = CacheSize / LineSize * Number\ of\ Sets = CacheSize/L * 4$$

Also for this experiment, we used a total of 1000000 memory addresses per generator function.

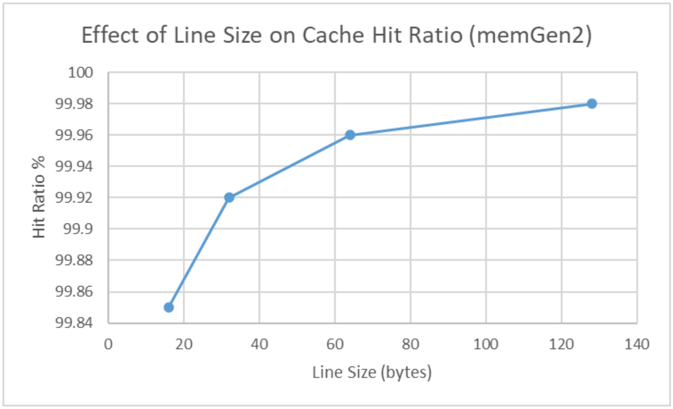
Results

For memGen1() which generates strictly sequential addresses through the entire 64 MB memory, we got the following results for the hit ratio when varying the line size:



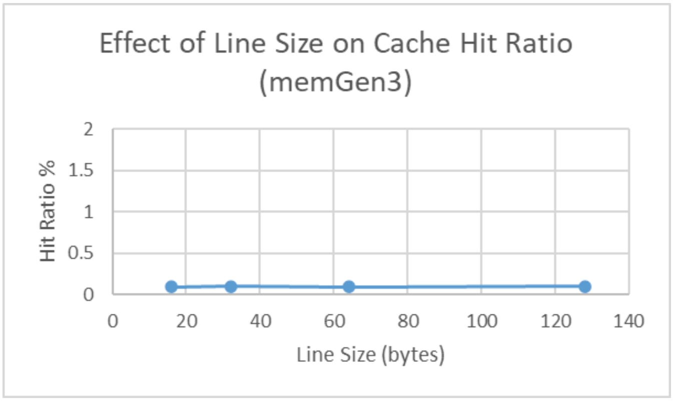
As shown in this graph, increasing the line size improved the hit ratio, because that makes better use of the spatial locality.

For memGen2() which generates uniformly random addresses within a 24 KB range, we got the following results for the hit ratio when varying the line size:



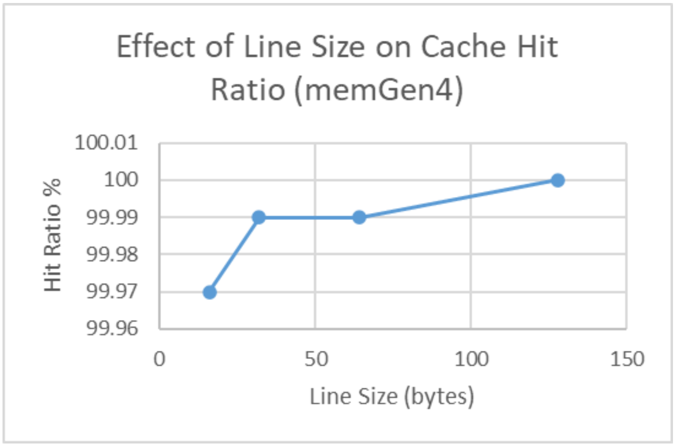
The above graph shows slight improvement in the hit ratio. That's because of the spatial locality and that the hit rates are already high.

For memGen3() that generates uniformly random addresses over the full 64 MB, we got the following results for the hit ratio when varying the line size:



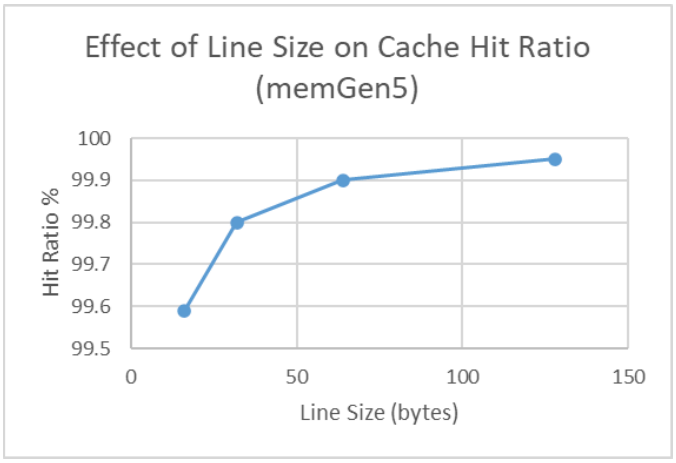
As illustrated in the above graph, there is no major impact from the line size which shows poor spatial and temporal locality.

For memGen4() which produces addresses that are strictly sequential in a small 4 KB range, we got the following results for the hit ratio when varying the line size:



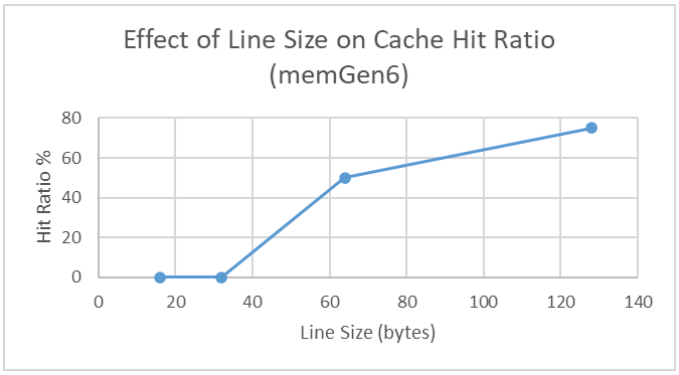
The above graph shows a stable, high hit ratios. This is because of the repeated access to a small working set.

For memGen5() that produces sequential addresses in the range of 64 KB, we got the following results for the hit ratio when varying the line size:



The above graph shows that increasing the line size improves the hit ratio for block reuse patterns.

For memGen6(), which generates memory addresses by incrementing by 32 bytes and wrapping around every 256 KB, we got the following results for the hit ratio when varying the line size:



The above graph shows that higher line size reduced conflict in some access patterns.

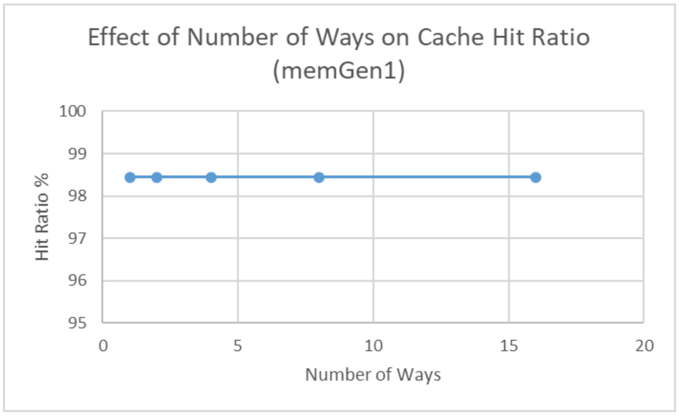
To sum up the results from experiment 1, we can see that varying the cache line size had a clear impact on patterns with strong spatial locality such as sequential or block-reues access patterns. Larger line sizes improved the cache performance in terms of the hit ratio because this way it captures more useful data per fetch. However, with random access patterns with little locality such as memGen3, the line size had a small effect.

Experiment 2: Effect of Associativity (number of ways)

For this experiment, the line size was fixed to be 64 bytes. The number of ways varied to see its effect on the cache performance, and the total number of memory addresses per configuration is 1000000.

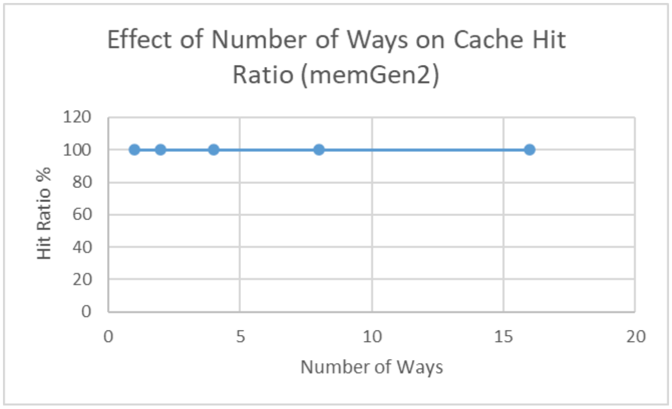
Results

For memGen1() which generates strictly sequential addresses through the entire 64 MB memory, we got the following results for the hit ratio when varying the number of ways:



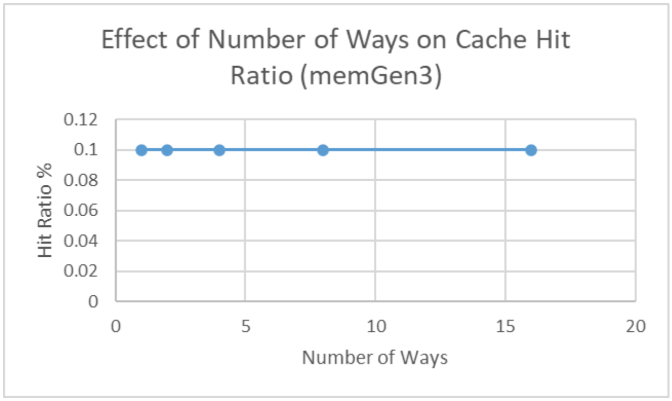
The graph shows no significant change meaning that there is an already high spatial locality.

For memGen2() which generates uniformly random addresses within a 24 KB range, we got the following results for the hit ratio when varying the number of ways:



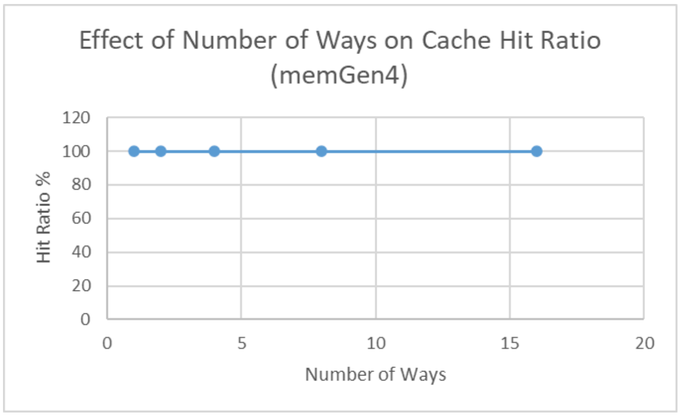
The graph shows a stable high hit ratio across all configurations.

For memGen3() that generates uniformly random addresses over the full 64 MB, we got the following results for the hit ratio when varying the number of ways:



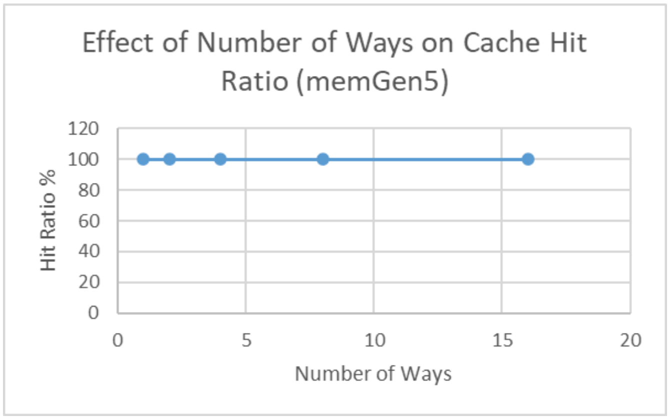
The above graph shows that random access patterns gain no benefit from increasing the number of ways (associativity)

For memGen4() which produces addresses that are strictly sequential in a small 4 KB range, we got the following results for the hit ratio when varying the number of ways:



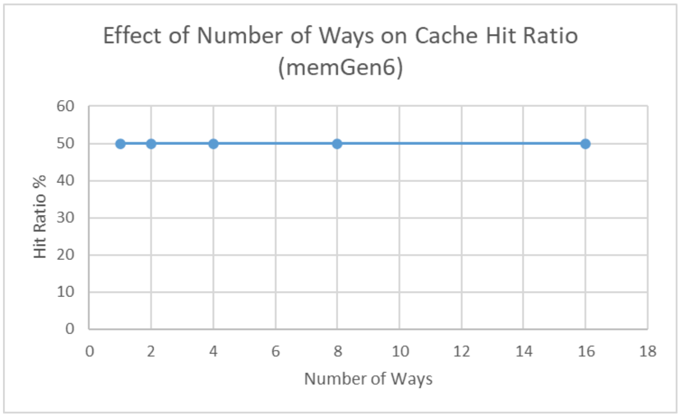
The graph shows a high hit ratio because of the repeated access to a small working set, and that increasing the number of ways has no effect.

For memGen5() that produces sequential addresses in the range of 64 KB, we got the following results for the hit ratio when varying the number of ways:



The above graph shows a very high hit ratio across all configurations, indicating high spatial and temporal locality.

For memGen6(), which generates memory addresses by incrementing by 32 bytes and wrapping around every 256 KB, we got the following results for the hit ratio when varying the number of ways:



The above graph shows that changing the number of ways didn't significantly reduce the number of conflict misses.

Overall, changing the number of ways didn't have a big impact on the cache performance on most memory access patterns. Patterns with strong spatial or temporal locality such as memGen1, memGen2, memGen4, and memGen5 already have high hit ratio, regardless of the associativity. Also in highly random patterns such as memGen3 showed no improvement. Even in conflict-prone scenarios like memGen6, higher associativity did not have a strong impact on the hit ratio.

Analysis and Discussion

Key Insights

It was shown from the experiments that increasing the cache line size significantly helps with sequential and block-access addresses accessing patterns. Which should be the case because of the spatial locality. In addition, while changing the number of ways didn't show significant improvement in the hit ratio, it can be useful for workloads that face conflict misses.

Trade-offs

While the line size can improve performance, larger lines reduce the total number of cache lines, which in turn can increase the conflict misses in some patterns. Moreover, while associativity can improve hit ratio for conflict prone patterns, it adds extra hardware complexity and might result in increased latency.

Testing the Cache Simulator

To make sure that our simulator is working properly, we implemented two things. The first one is functional correctness tests. The second thing is configurable experiments. In the configurable experiments, we varied our key parameters (line size and number of ways).

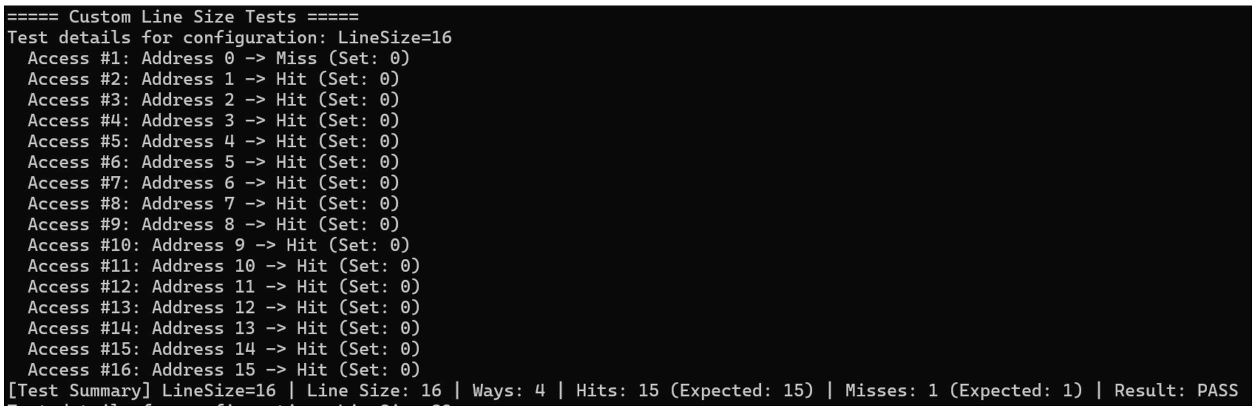
Correctness Testing

We wanted to validate the simulator against known memory access patterns and known hit and miss ratios, so we created the testCorrectness() function. It takes as parameters a set of addresses that are predefined to match a certain access pattern. After that, it compares the simulator's outcome with the known hit and miss ratios.

Custom Line Size Tests

To test the simulator, we had various line sizes (16 B, 32 B, 64 B, and 128 B), and a fixed associativity of 4. In the test, we tried to access consecutive memory locations within the same line size range. The known is that after trying to access the first address, which will result in a first cold miss, the cache will bring an entire block of memory that is of the size of the line. Consequently, the rest of the memory addresses accesses would be a hit due to the spatial locality.

Here is a snippet of the output for this test



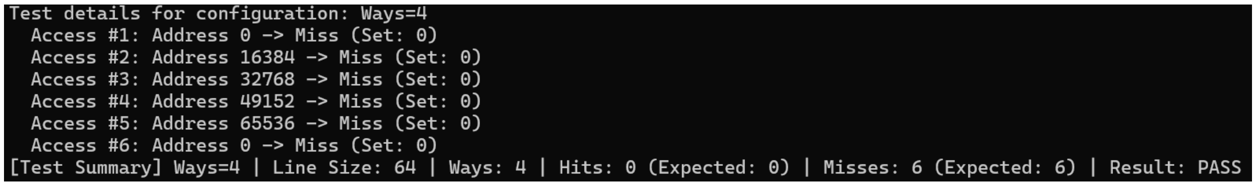
```
===== Custom Line Size Tests =====
Test details for configuration: LineSize=16
  Access #1: Address 0 -> Miss (Set: 0)
  Access #2: Address 1 -> Hit (Set: 0)
  Access #3: Address 2 -> Hit (Set: 0)
  Access #4: Address 3 -> Hit (Set: 0)
  Access #5: Address 4 -> Hit (Set: 0)
  Access #6: Address 5 -> Hit (Set: 0)
  Access #7: Address 6 -> Hit (Set: 0)
  Access #8: Address 7 -> Hit (Set: 0)
  Access #9: Address 8 -> Hit (Set: 0)
  Access #10: Address 9 -> Hit (Set: 0)
  Access #11: Address 10 -> Hit (Set: 0)
  Access #12: Address 11 -> Hit (Set: 0)
  Access #13: Address 12 -> Hit (Set: 0)
  Access #14: Address 13 -> Hit (Set: 0)
  Access #15: Address 14 -> Hit (Set: 0)
  Access #16: Address 15 -> Hit (Set: 0)
[Test Summary] LineSize=16 | Line Size: 16 | Ways: 4 | Hits: 15 (Expected: 15) | Misses: 1 (Expected: 1) | Result: PASS
```

As shown in the above snippet, the first access caused a miss, because the cache is trying to load a new block. However, all of the subsequent accesses to addresses that are in the same block resulted in a hit. This confirms utilization of the spatial locality and the correctness of the simulator for this parameter (line size).

Replacement Testing with Varying Associativity

In this test, we want to check if the replacement policy we implemented is working correctly. Therefore, we fixed the line size to 64 B, and varied the number of ways. In this test, we filled a single set with more blocks than it can hold, and observed whether the replacement was done as expected.

Here is a snippet of the output for this test



```
Test details for configuration: Ways=4
  Access #1: Address 0 -> Miss (Set: 0)
  Access #2: Address 16384 -> Miss (Set: 0)
  Access #3: Address 32768 -> Miss (Set: 0)
  Access #4: Address 49152 -> Miss (Set: 0)
  Access #5: Address 65536 -> Miss (Set: 0)
  Access #6: Address 0 -> Miss (Set: 0)
[Test Summary] Ways=4 | Line Size: 64 | Ways: 4 | Hits: 0 (Expected: 0) | Misses: 6 (Expected: 6) | Result: PASS
```

As shown in the above snippet, we had a test with the number of ways = 4. We customized the test so that the addresses used map to the same set (set 0 in this example), and since the associativity is 4, only four blocks can be stored in that set simultaneously. And after we filled all four ways of the set (which all resulted in a cold start miss), we tried to access a fifth address that map to the same set that resulted in a conflicting miss and replacing the first way of the set. After that, we tried to re-access the very first address (in this example address 0), and it resulted in a miss because it had been replaced. This confirms that the replacement strategy we implemented is working correctly.

Summary

The structure tests we had showed that the simulator works properly as it correctly maps the addresses to the correct cache blocks using the set index and tag logic. In addition, the tests proved that the replacement policy is working as expected. Finally, both the hit and miss ratios were proven to be correct. Overall, the testing functions and the custom tests made us more confident in the correctness of our cache simulator.

Conclusion

This project demonstrates how changing the cache memory parameters (line size and number of ways) can impact its performance (measured by the hit rate). We found that increasing the line size improves performance for patterns with strong spatial locality, while associativity is more beneficial for conflict-prone patterns. Random workloads do not benefit much from either. The project provided useful insights into the importance of tailoring the cache designs for specific applications.