CSCE 231 / 2303 – Computer Organization
Spring 2025 – Project 2

Cache Performance Evaluation Report

Group Members:

- Seba Ahmed Wahba – 900225470

Section: 10:00 am WU
Instructor: Dr Mohammed Shalan

1. Introduction

A cache is essentially a memory storage space that was introduced to tackle a significant problem that emerged in the world of computing in the late 1980s. That problem is basically the fact that before the 1980s the processor's speed and the main memory matched each other, but soon this ended with the computational advancements causing the processor to exceed the memory's speed. This caused the memory to become a bottleneck as it slowed down the processor and the processor advancements were no longer useful. So the cache solved this issue by acting as an intermediary between the processor and the memory such that the memory can be manufactured using the DRAM which is dense,slower, cheap and available in large sizes, while the cache is made out of the SRAM which is less dense, faster, expensive and available in small sizes. Now, the processor can get the data from the cache and if it finds it then it is a hit, if not then it is a miss and data is fetched from the memory with a miss penalty.

2. Cache Simulator Design

Data structures: CacheLine, Cache class

I made use of the LRU replacement policy which checks for the least recently used data and replaces it with the data fetched from the main memory when the cache's capacity is full (Capacity Cache). This is done by adding an additional bit to each line in the cache called the counter bit which keeps track of the order of accessing each line such that in the end we can tell which line has the minimum counter bit and so it will be the least recently accessed line of data.

Cache parameters:

- Fixed cache size = 64 KB
- Line size: 16, 32, 64, 128 bytes
- Associativity (ways): 1, 2, 4, 8, 16

How each component is calculated:

Offset: We calculate the log of the line size base 2.

Index bits: We calculate how many sets we have, and how many bits we need to pick a set.

As for the actual index, it makes use of the index bits that were calculated. They are shifted to the right to remove block offset bits (they're not needed to find the set). Then we use a bitmask to extract just the index bits.

Now moving on to how each memory generator functions:

memGen1: It Sequentially walks through memory, Type and has a high spatial locality, It is also great for testing how well the cache handles predictable access

memGen2: Random access in a very small region (24 KB), type and has a temporal locality, It is also good for testing small working sets that fit in cache.

memGen3: Random access across the entire 64 MB, type and has a No locality, It also Simulates worst-case for cache (everything is a miss).

memGen4: Sequential access in a very small range (4 KB), type and has a very high spatial + temporal locality, It should result in near 100% hit rate after warm-up.

memGen5: Sequential access in a 64 KB region, Medium spatial and temporal locality,Tests how cache handles a working set that matches cache size.

memGen6: Strided access, skips 32 bytes each time,Breaks spatial locality for small line sizes, Tests if line size is large enough to cover stride (e.g., stride = 32, so line size 64 or 128 helps).

---

3. Experiment Setup

I ran 1,000,000 memory accesses per test

Used 6 memory generators
Two experiments:
- Experiment 1: Fixed 4 sets, varied line size
- Experiment 2: Fixed line size = 64, varied ways (1–16)

C++ was used for cache simulation and Google Sheets was used for plotting, graphing and recording results.

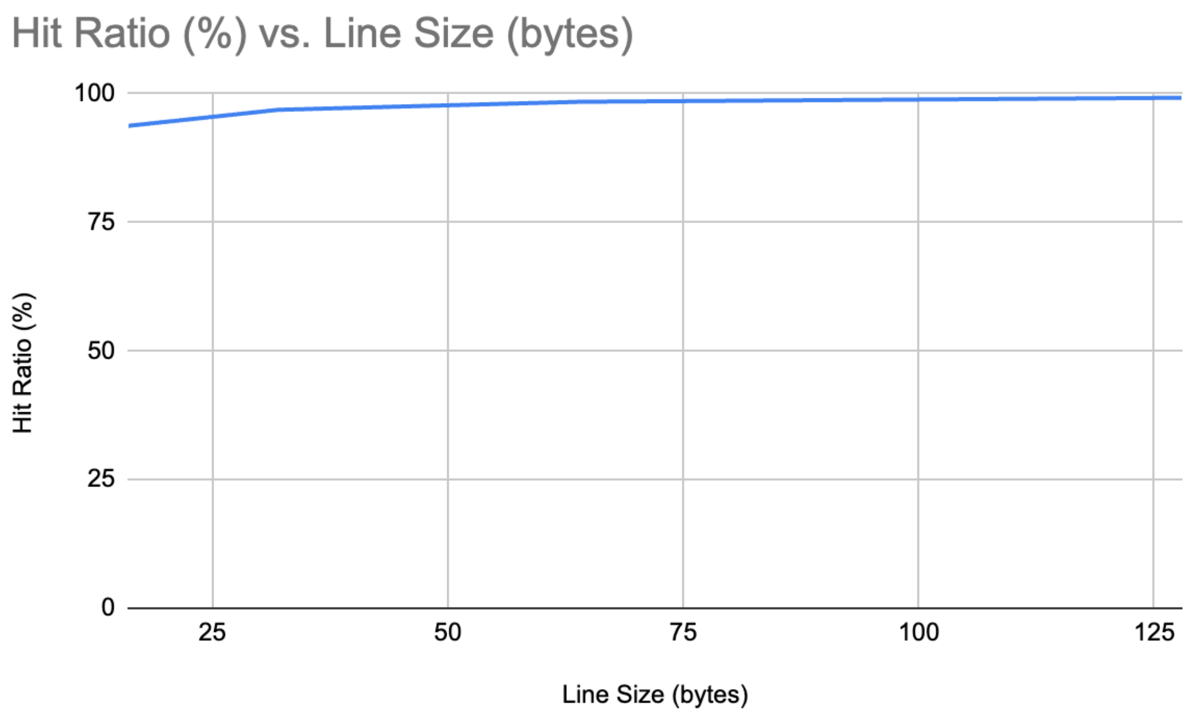
---

4. Results & Graphs

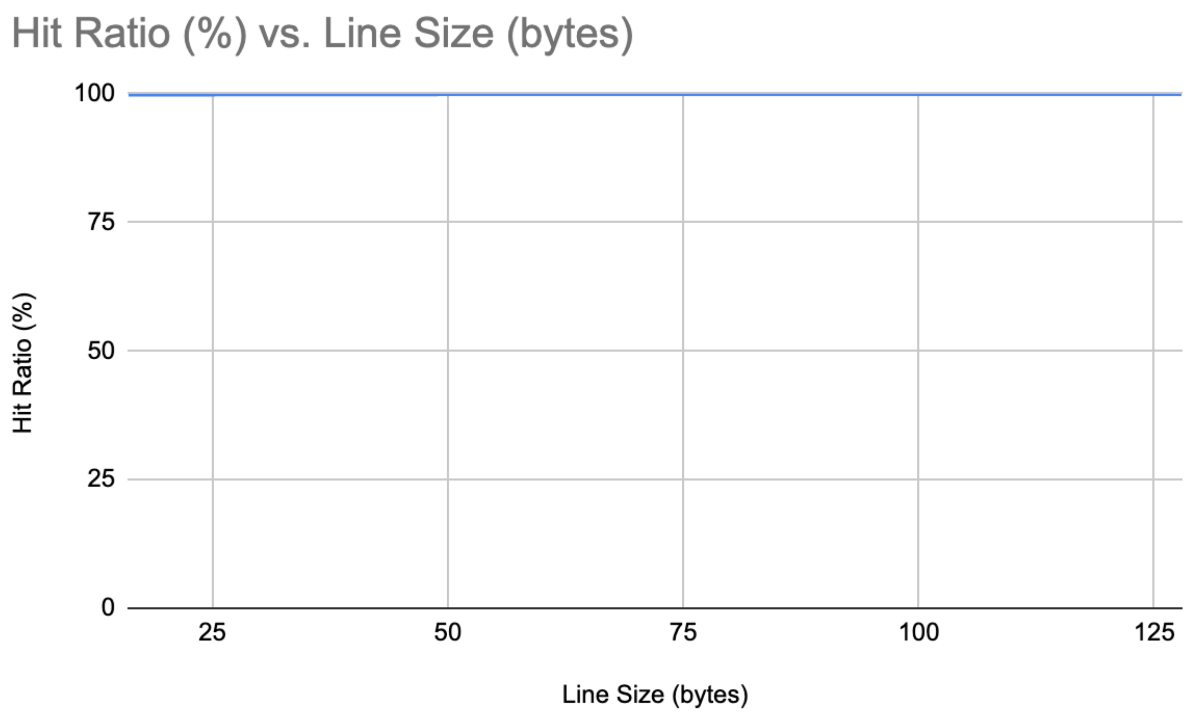
Experiment 1: Hit Ratio vs Line Size (Sets = 4)

[Insert tables and line charts here for each generator, e.g., memGen1 through memGen6.]

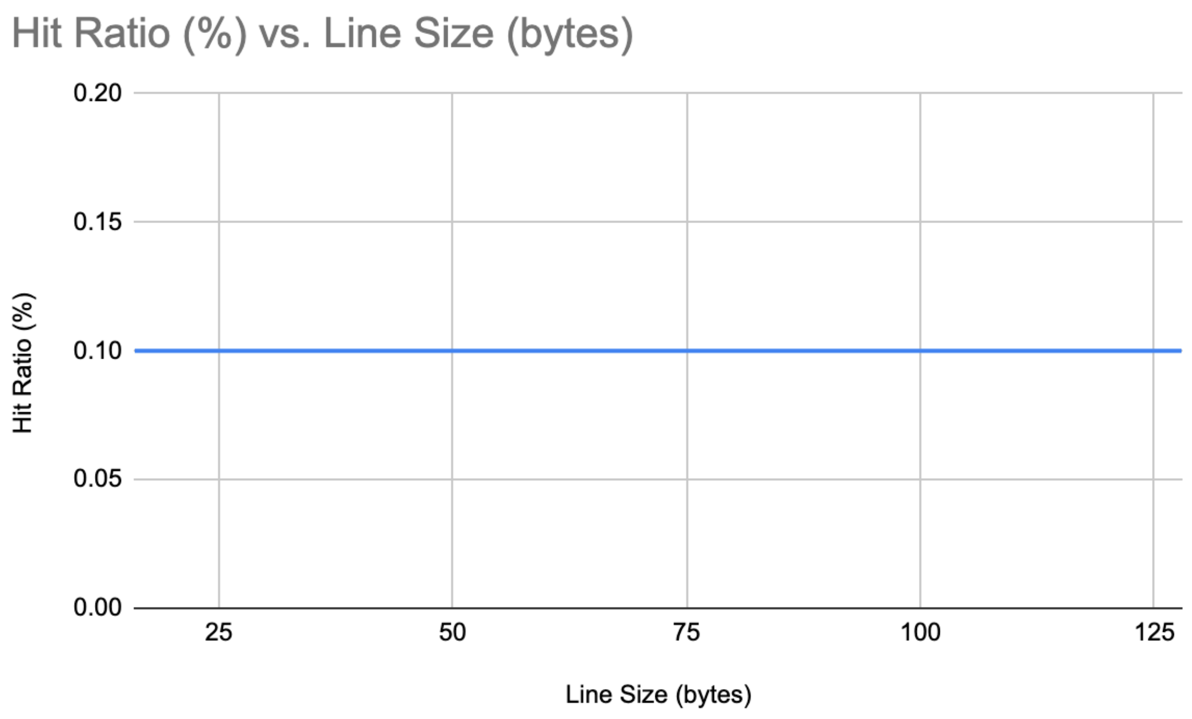
MemGen1:



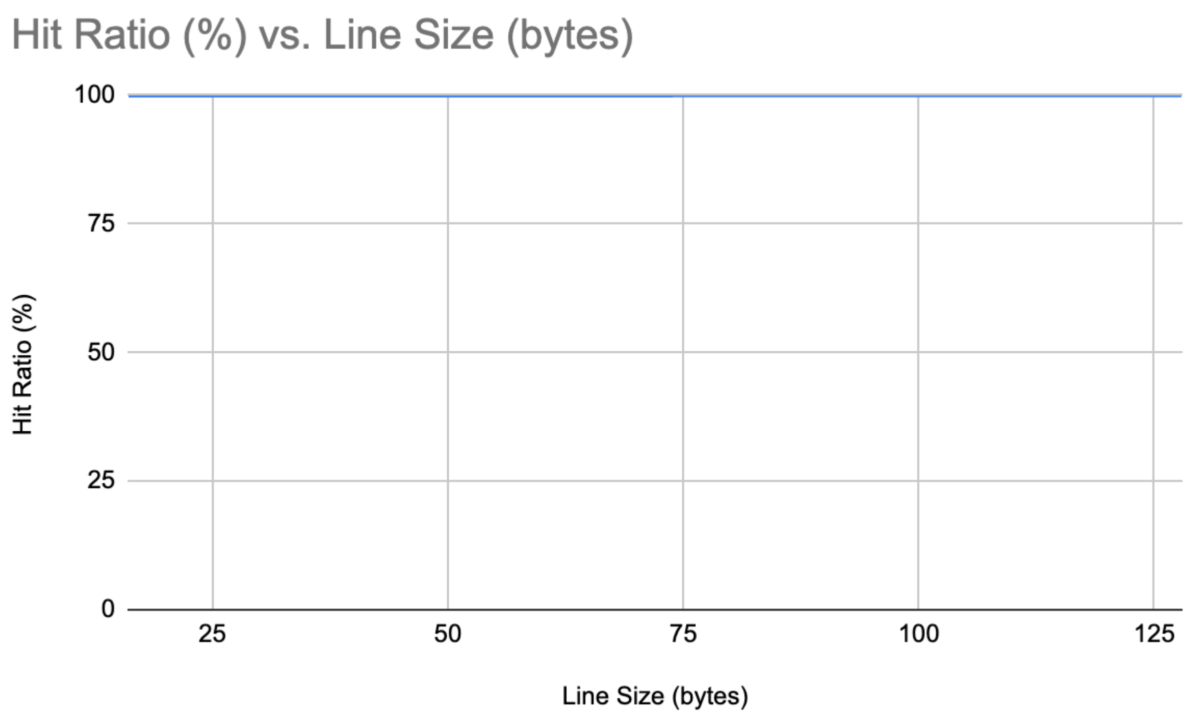
MemGen2:



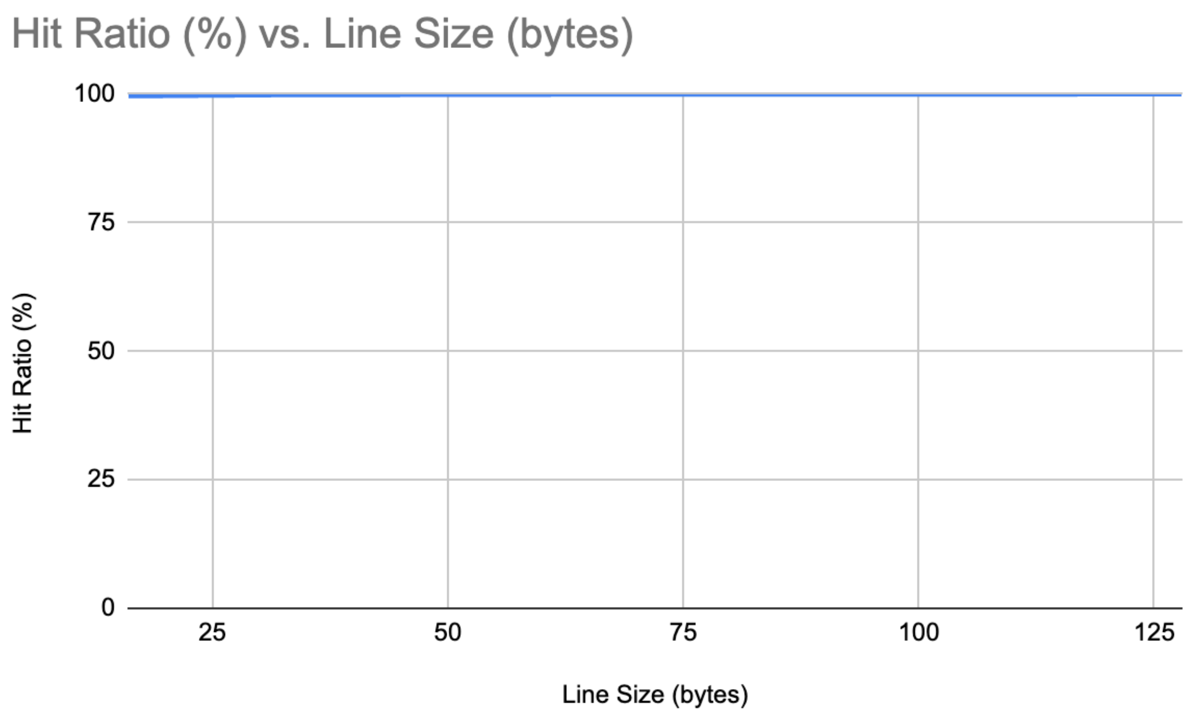
MemGen3:



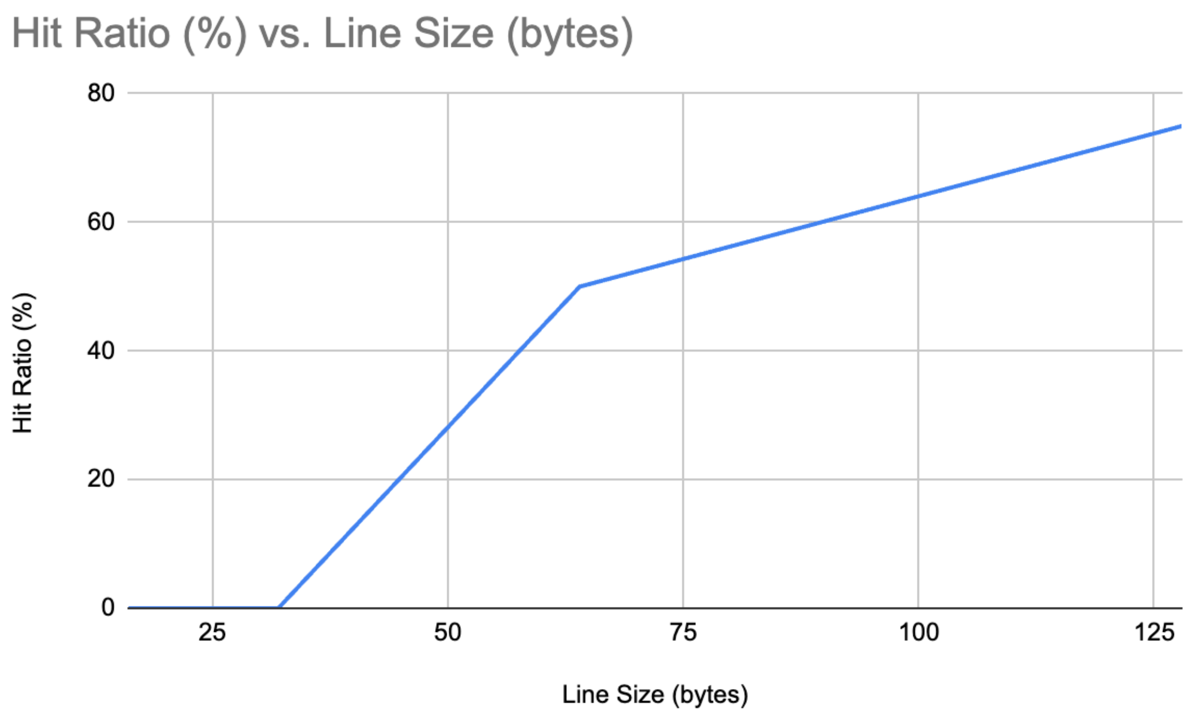
MemGen4:


Hit Ratio (%) vs. Line Size (bytes)
100
75
50
25
0
Hit Ratio (%)
25
50
75
100
125
Line Size (bytes)

MemGen5:


Hit Ratio (%) vs. Line Size (bytes)
100
75
50
25
0
Hit Ratio (%)
25
50
75
100
125
Line Size (bytes)

MemGen6:



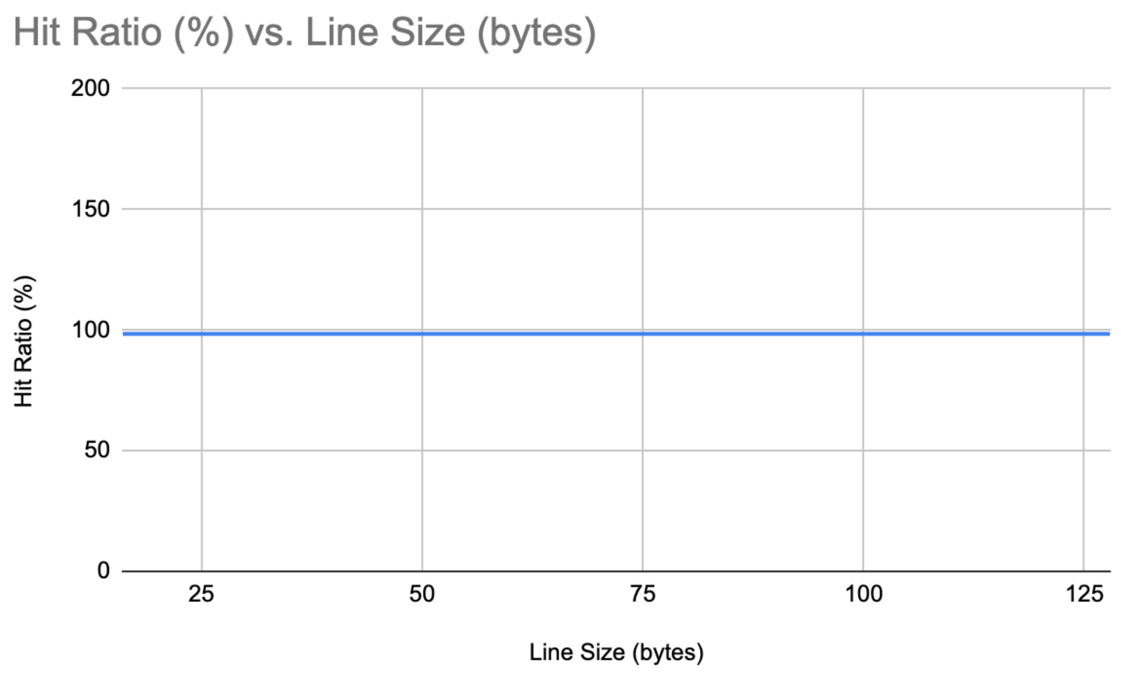
Generator	Line Size (bytes)	Hit Ratio (%)
memGen1	16	93.75
memGen1	32	96.88
memGen1	64	98.44
memGen1	128	99.22
memGen2	16	99.85
memGen2	32	99.92
memGen2	64	99.96
memGen2	128	99.98
memGen3	16	0.1
memGen3	32	0.1
memGen3	64	0.1

memGen3	128	0.1
memGen4	16	99.97
memGen4	32	99.99
memGen4	64	99.99
memGen4	128	100
memGen5	16	99.59
memGen5	32	99.8
memGen5	64	99.9
memGen5	128	99.95
memGen6	16	0
memGen6	32	0
memGen6	64	50
memGen6	128	75

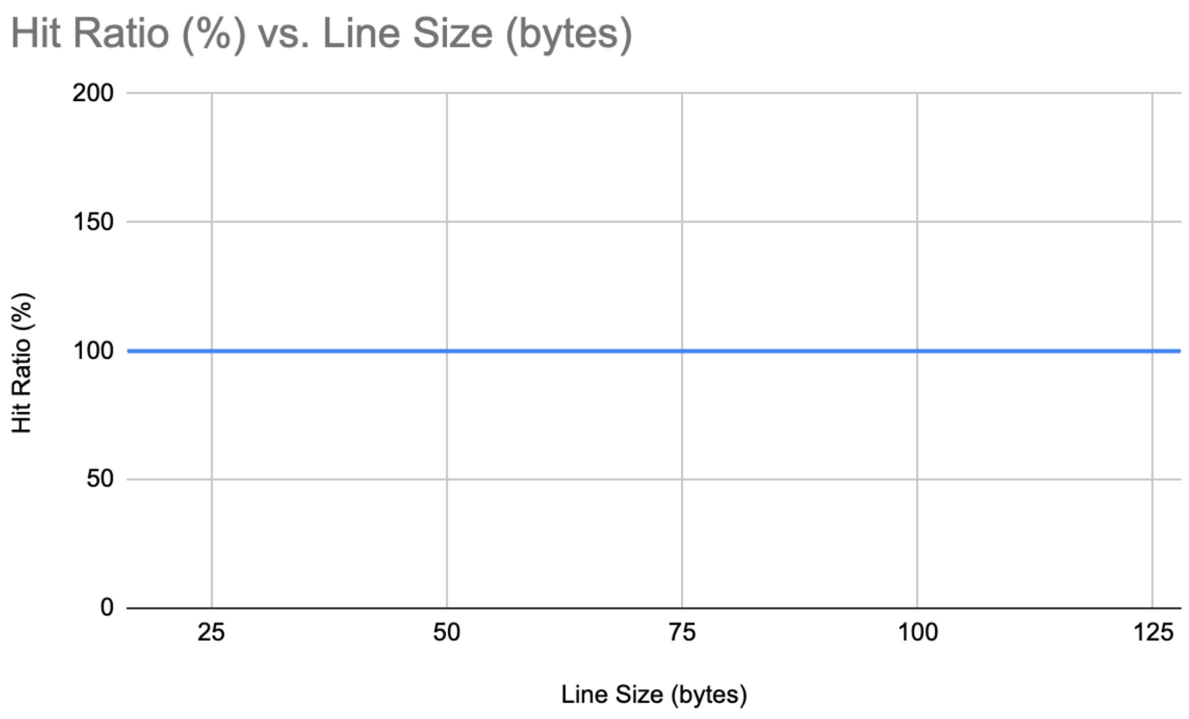
Experiment 2: Hit Ratio vs Associativity (Line Size = 64)

[Insert tables and line charts here for each generator, e.g., memGen1 through memGen6.]

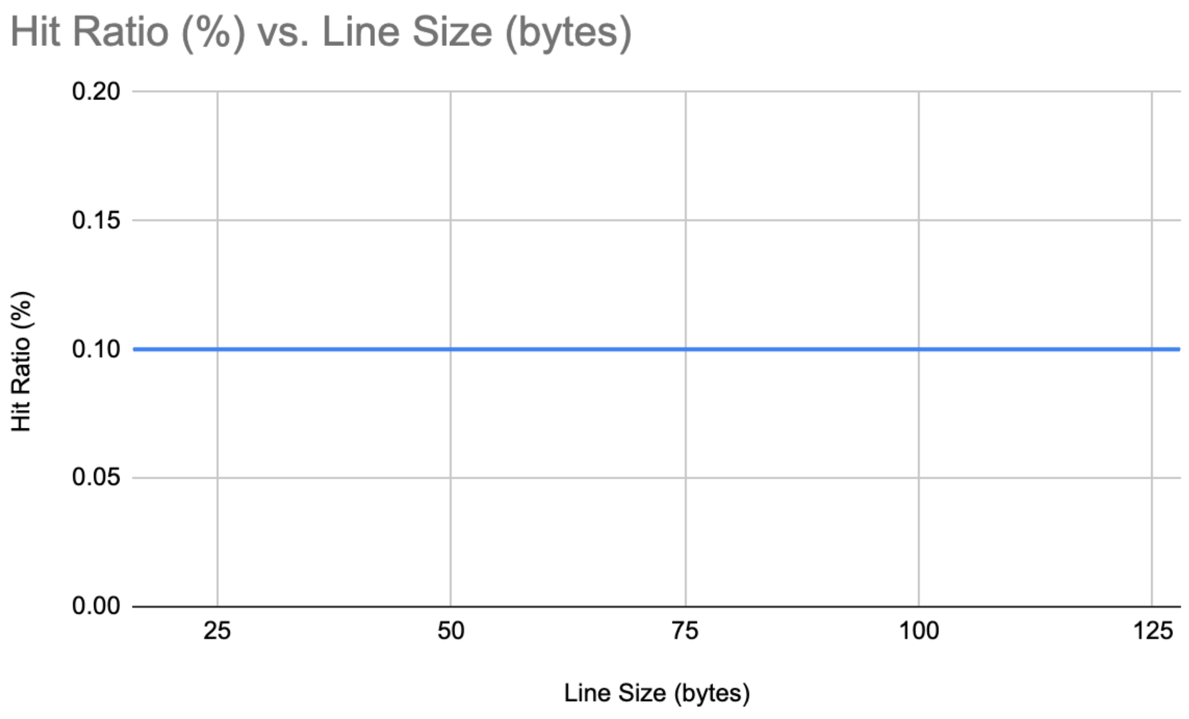
MemGen1:



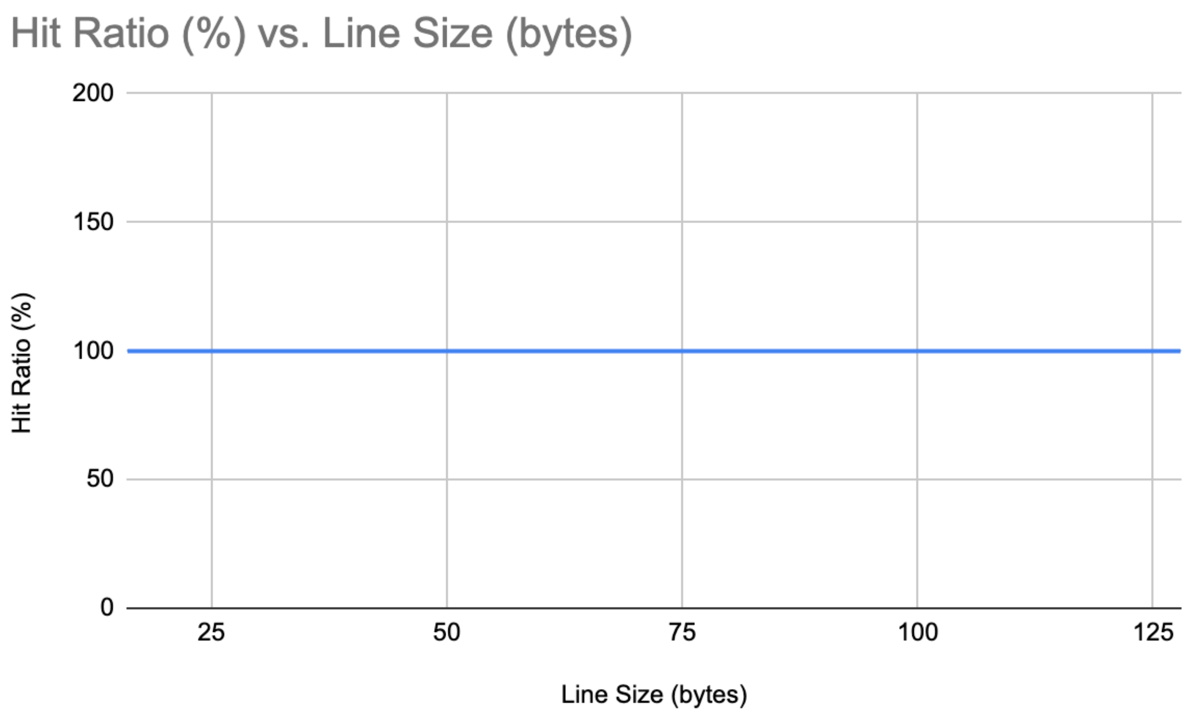
MemGen2:



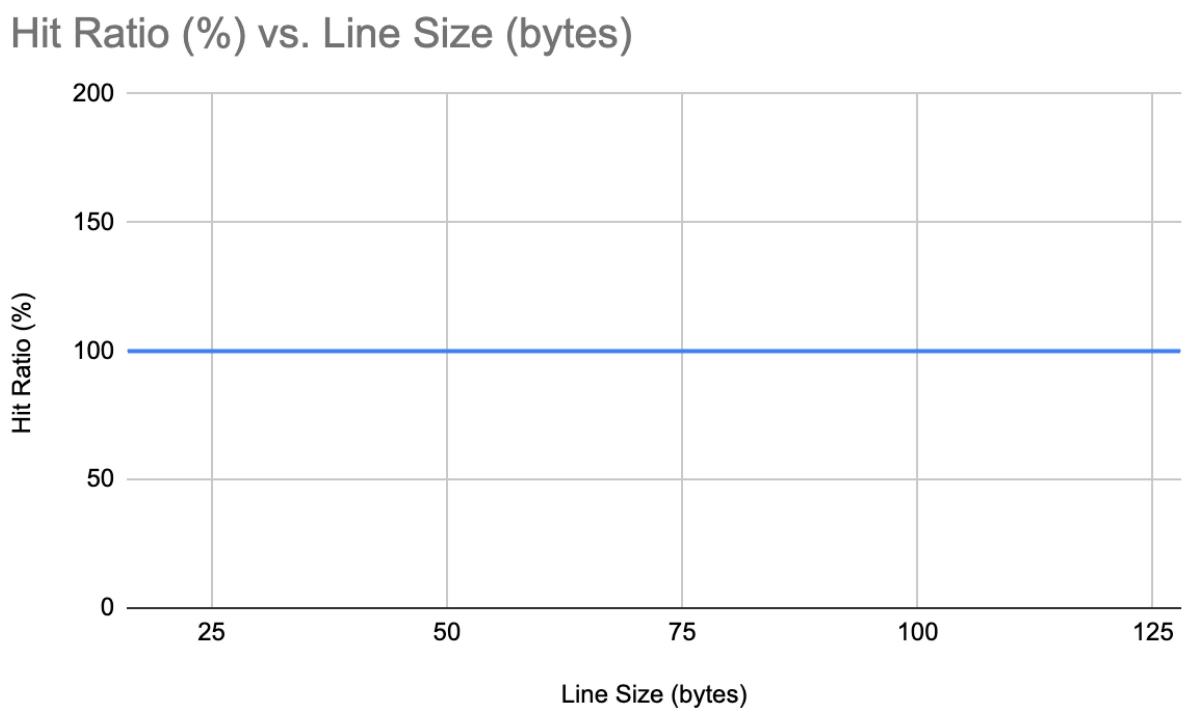
MemGen3:



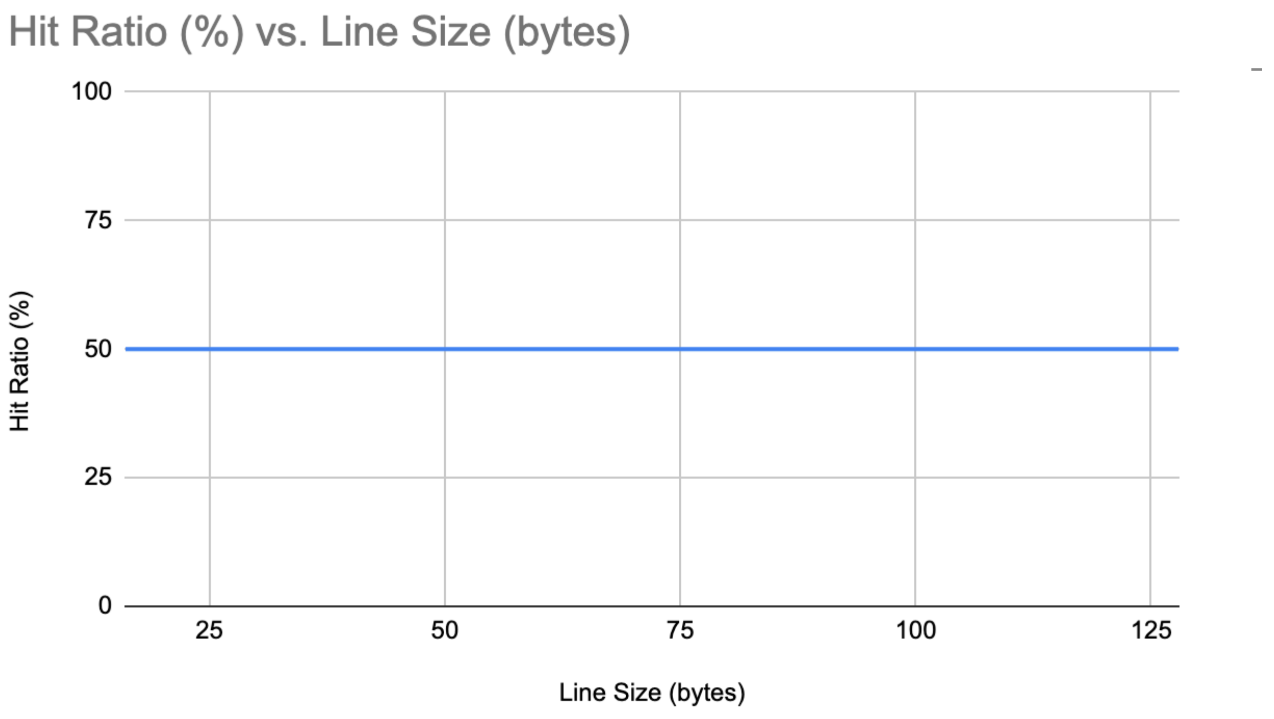
MemGen4:



MemGen5:



MemGen6:



Generator	Line Size (bytes)	Hit Ratio (%)
memGen1	16	98.44
memGen1	32	98.44
memGen1	64	98.44
memGen1	128	98.44
memGen2	16	99.96
memGen2	32	99.96
memGen2	64	99.96
memGen2	128	99.96
memGen3	16	0.1
memGen3	32	0.1
memGen3	64	0.1
memGen3	128	0.1
memGen4	16	99.99

memGen4	32	99.99
memGen4	64	99.99
memGen4	128	99.99
memGen5	16	99.9
memGen5	32	99.9
memGen5	64	99.9
memGen5	128	99.9
memGen6	16	50
memGen6	32	50
memGen6	64	50
memGen6	128	50

5. Analysis & Discussion

[Explain trends and observations here.]

- How did increasing line size affect the hit ratio?
  Larger line size improved hit ratio for generators with spatial locality (e.g., memGen1, memGen4), but had no effect for random access (e.g., memGen3).
- How did associativity impact performance?
  Associativity helped very little in most patterns; hit ratio stayed flat for almost all memory generators.
- Which generators benefited most from larger line sizes?
  memGen1, memGen5, memGen6 — because they access data sequentially or with predictable strides.
- Which access patterns saw no improvement from higher associativity?
  memGen3 and memGen6 — both had flat hit ratios regardless of associativity.
- Any unexpected results?
  Associativity had almost no impact even in strided or larger working set patterns.

---

6. Conclusion

This project demonstrated how two key configuration parameters: line size and associativity (ways) can impact cache performance. Through a set-associative cache simulator, and experimenting with six distinct memory access patterns, we deduced the following:

- Larger line sizes significantly improved hit ratios for access patterns with high spatial locality (e.g., memGen1, memGen4, memGen5). In strided access patterns (memGen6), performance improved dramatically once the line size exceeded the stride size.

- Increasing associativity (ways) had minimal impact for most access patterns. In nearly all cases, moving from 1-way (direct-mapped) to higher associativity produced little to no improvement, indicating that conflict misses were rare in the tested scenarios because the problem was due to the fact that the data was not in the set itself not that it was conflicting with other data in the same set.

- Access patterns with high temporal locality's (e.g., memGen2, memGen4) ratios rockettes regardless of cache structure, while purely random access patterns (memGen3) saw no performance gain from any configuration.

- Surprisingly, associativity had no impact on some patterns where it might be expected to help (e.g., memGen6), reinforcing the importance of line size alignment with memory access stride.

Overall, these findings highlight that cache performance depends more on access patterns and line size than on associativity alone. Understanding a program's memory behavior is crucial when tuning cache parameters for optimal performance.