THE AMERICAN UNIVERSITY IN CAIRO

100 YEARS

CSCE 2303 - Computer Organization and Assembly Language Programming

Set-Associative Caches: A Performance Simulator

Ahmed Soliman, Saifeldin Abdulrahman

Dr. Mohamed Shalan

Spring 2025

https://github.com/saifsabry21/AssemblyProject2.git

Abstract

This report presents the design, validation, and performance analysis of a 64 KiB cache simulator that supports set-associative (SA). Implemented are six distinct memory generators—ranging from sequential full‑DRAM access to random patterns—and execute one million memory references per experiment. Two core experiments are tested: (1) varying cache line size (16, 32, 64, 128 bytes) with a fixed four‑set SA configuration, and (2) varying associativity (1, 2, 4, 8, 16 ways) at a constant 64 B line. Hit and miss ratios are recorded and visualized via plots. Results indicate that larger line sizes substantially improve spatial locality, thus improving hit ratio. Comparisons reveal that DM caches possess significant conflict misses due to their design, whereas SA caches maintain higher hit rates. These findings correlate to different trade-offs depending on the cache design implemented and the access patterns.

1. Introduction

In a world where processor speeds significantly outpace main memory latency, bridging the performance gap between registers and the DRAM is crucial for sustaining application throughput and energy efficiency. Caches provide this bridge by storing a small, fast subset of data—recently or frequently accessed—in on‑chip SRAM. The behavior of a cache depends on three main factors: cache organization (direct‑mapped, full-associative, set‑associative), cache line size (the scale of data transfer), and associativity (number of ways per set).

A direct‑mapped (DM) cache offers minimal hardware complexity by mapping each memory block to exactly one cache line. However, due to this minimal design, DM caches are highly susceptible to conflict misses when multiple blocks are destined to go to the same line. A set‑associative (SA) cache tackles this issue by dividing the cache into sets, each containing multiple lines (ways). In a SA cache, a block maps to a specific set, the same as the DM cache, but may occupy any way within that set, therefore reducing conflict misses significantly. While SA designs reduce conflict misses, they involve more complex replacement logic (random, least recently used, least frequently used) and slightly higher access latency.

This project investigates two fundamental questions:

1. How does varying cache line size (16, 32, 64, 128 bytes) affect hit and miss ratios for diverse memory access patterns in a four‑set SA cache?
2. How does changing associativity (1, 2, 4, 8, 16 ways) at a fixed 64 B line size influence cache performance across six distinct address generators?

To address these questions, we implemented both DM and SA cache simulators in C++, validated their correctness via a set of test cases, focusing on edge ones, then ran one million memory references per experiment, calculating the hit and miss ratios for each simulator while using the different generators. Statistics are recorded and visualized via plots, and analyzed to clearly explain the tradeoffs between different design approaches and conflict misses.

2. Methodology

2.1 Design

Our simulator models a 64 KiB cache sitting in front of a 64 MiB DRAM.

- Set-Associative (SA)**.** The cache is divided into `NumSets` sets. Each set contains `ways = (CACHE_SIZE / line_size) / NumSets` lines. A block maps to a single set but may occupy any way inside it.

Every cache line is modeled by the same structure:

```
struct CacheLineFA {

   unsigned int tag = 0;

   bool  valid = false;

   unsigned int leastrecentuseCounter = 0;

};
```

All lines are found in a dynamic array FaCache, with a counter FaAccessCount incremented with every memory reference.

To get memory Addresses, six built-in MemGen functions with different patterns – from Full-DRAM sequential to stride – generate a 32-bit byte address.

2.2 Test case setup

To ensure that the algorithms developed in each cache simulator is correct, a sequence of tests were developed with different cases, mainly focusing on edge ones, to ensure validity of results. The `SATestCases()` function feeds carefully chosen address sequences to the SA simulator and prints "HIT" or "MISS". Expected outcomes, written beside each call, check that:

- the first miss populates an empty line
- the LRU line is evicted when a set overflows

- independent sets do not interfere

Also, to ensure different test cases don't interfere with each other, a helper function resetSACache() changes line size and number of sets to reset the cache before being used in another test.

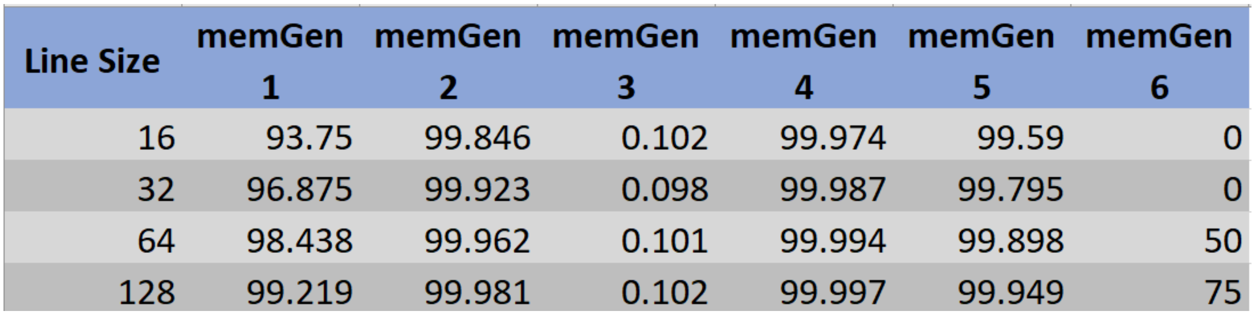
2.3 Data Collection in Experiments

For every generator and cache configuration, 1,000,000 memory references (NO_OF_Iterations) are issued and are put in two experiments:

- Experiment 1**:** Fix `NumSets = 4` and vary `line_size (16,32,64,128)`.
- Experiment 2: Fix `line_size= 64 B` and ary `NumSets(1, 2, 4, 8, 16, 32, 64)`, which produces ways = 64 all the way down to 1 (DM case is the 1-way endpoint).

During each experiment, the number of hits are calculated and displayed for further analysis via plots and charts.

3. Results



Line Size	memGen 1	memGen 2	memGen 3	memGen 4	memGen 5	memGen 6
16	93.75	99.846	0.102	99.974	99.59	0
32	96.875	99.923	0.098	99.987	99.795	0
64	98.438	99.962	0.101	99.994	99.898	50
128	99.219	99.981	0.102	99.997	99.949	75

Fig 1.1 `(NumSets = 4)`

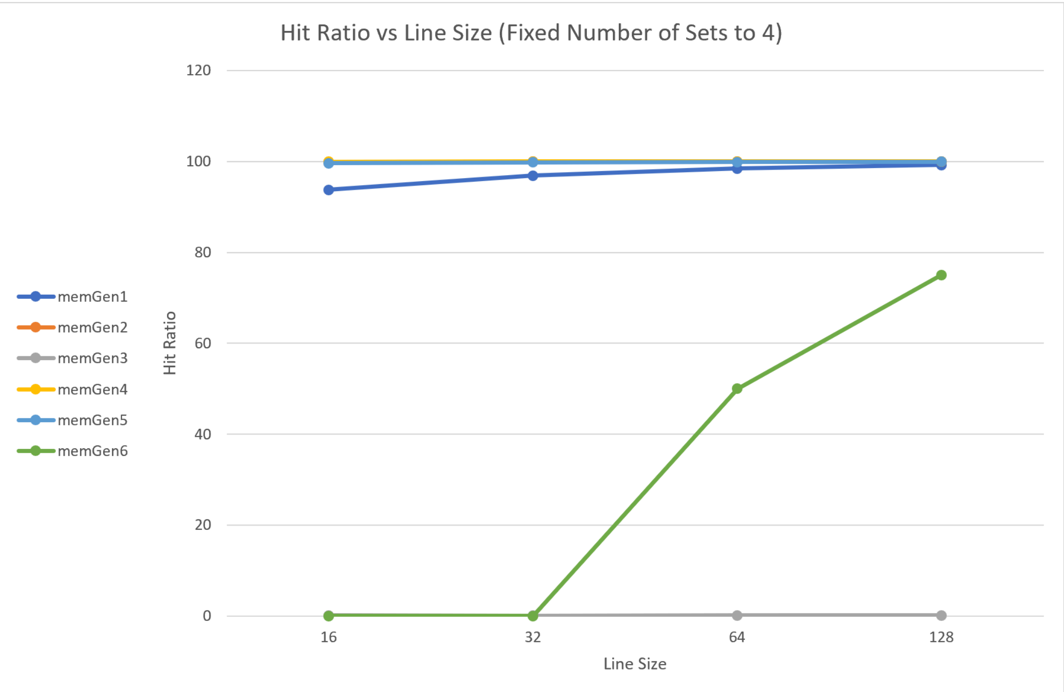
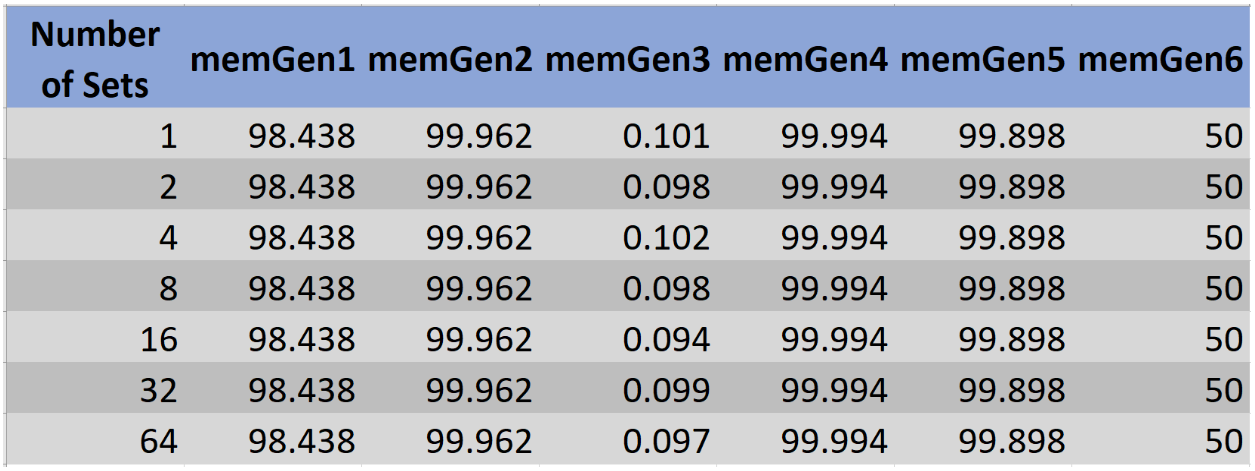


Fig 1.2 (Graphical Representation of Fig 1.1)



Number of Sets	memGen1	memGen2	memGen3	memGen4	memGen5	memGen6
1	98.438	99.962	0.101	99.994	99.898	50
2	98.438	99.962	0.098	99.994	99.898	50
4	98.438	99.962	0.102	99.994	99.898	50
8	98.438	99.962	0.098	99.994	99.898	50
16	98.438	99.962	0.094	99.994	99.898	50
32	98.438	99.962	0.099	99.994	99.898	50
64	98.438	99.962	0.097	99.994	99.898	50

Fig 2.1 (Line_size= 64)

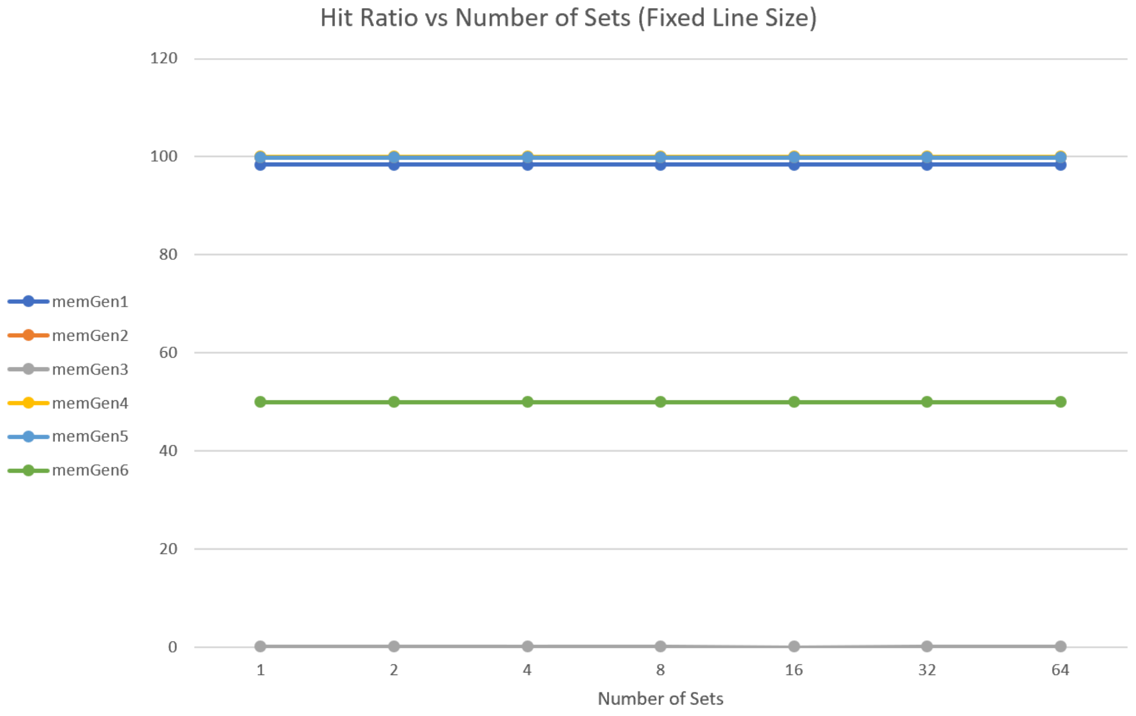


Fig 2.2 (Graphical Representation of Fig 2.1)

4. Discussion

The results gathered from both the functional test cases and large-scale data-collection simulations draw several key insights into how cache architecture impacts performance. This section analyses these outcomes from both functional correctness and performance aspects.

4.1 SA Cache implementation correctness

The five targeted test cases validated the accuracy of the Set-Associative (SA) cache implementation under varying patterns:

- **Test 1** confirmed correct Least Recently Used (LRU) replacement, showing how accessing a fourth unique address into a 3-line fully associative cache resulted in removal of the LRU line

- **Test 2** demonstrated that set isolation and LRU logic operated correctly across two sets with 2-way associativity.

- **Test 3** showed stable hit behavior upon repeated access to the same line, indicating correct caching and updating of LRU.

- **Test 4** verified independence between sets: accesses mapped to different sets showed no interference.

- **Test 5** revealed that the LRU algorithm correctly removed the oldest line when the set became full.

These results show that the SA cache simulator was implemented successfully

4.2 Experiment 1 discussion

Across all memory generators except MemGen3 and MemGen6, increasing the line size led to higher hit ratios. This behavior is expected as larger lines make use of spatial locality by fetching adjacent data on each miss. However, MemGen3 (random across 64 MiB) had negligible hit rates (about 0.10%) regardless of line size, confirming the idea that random access patterns prevent effective caching due to the absence of spatial and temporal locality. MemGen6 showed an abrupt jump—from 0% to 50% to 75%. This happens because MemGen6 uses a stride pattern tuned to 32 B. If the line is smaller than that, you'll never get hits (as indicated by the 16 and 32 B lines). On the other hand, if the line size is bigger such that it can take multiple 32 strides, then you can achieve hits as we saw in the cases where line sizes were 64 and 128B.

4.3 Experiment 2 discussion

Experiment 2 fixed the line size at 64 bytes and varied the number of sets.Interestingly, hit ratios remained largely **constant** for most memory generators. This suggests that for spatially or locally reused data, conflicts are minimal, and 4-way associativity eliminates most misses. Because of the randomness of selection in the memory, MemGen3 had a constant hit ratio of 0.10% , further supporting the nullification of the benefits of associativity. Because of its carefully selected step technique, there was a uniform 50.00% hit rate irrespective of associativity encountered. This suggests that every other access that is attempted registers a hit, most probably from the fact that data is reused precisely one time following the initial miss. Furthermore, associativity is not beneficial since the reuse interval is constant.

4.4 When does Associativity matter?

The results strongly suggest that beyond 4–8 ways of associativity, performance reaches a constant state. In all generators, all types of associativity eliminates almost all conflict misses. Only in certain cases – uniform randomness or weird stride patterns– does associativity fail to help as misses dominate.

4.5 Realistic Cache Behavior

From a practical standpoint, the cache simulator confirms that:

- Spatial locality works its magic with increased line size.
- Moderate associativity (4–8 ways) is sufficient in most scenarios.
- LRU performs well, but its benefits depend on the different patterns used.

5. Conclusion

The simulator develops and validates a set associative cache simulator that enables us to explore the cache's performance under varying configurations; whether number of lines or sets (ways). Using two experiments, we examined how the line size and associativity affects hit ratios for different memory addresses generated using random generators. The results of the experiments confirm that larger line sizes cause better hit rates in cases where there is high spatial locality(not high randomness). As a result when the randomness of the addresses were high like for memGen3, the hit rates weren't as boosted when the line size was increased as there is no locality. The results also show that 4 ways was enough to reach a low conflit miss rate. For a sequential pattern in memory access like memGen6 we were able to have a fixed hit ratio regardless of the associativity. Overall, our cache simulator provides an efficient tool to examine and evaluate cache behavior with different configurations that aids design and optimization efforts.