ASSIGNMENT : CACHE

Course Name: Computer Architecture

Siddharth Anil IMT2023503

Krish Patel IMT2023134

Siddhanth Deore IMT2023539

INDEX

1 CACHE IMPLEMENTATION

1.1 Purpose:

This Python script simulates a set-associative cache that can be configured dynamically with varying cache sizes, block sizes, and associativity (ways). It reads memory addresses from a trace file (gcc.trace) and simulates cache hits and misses using the Least Recently Used (LRU) replacement policy.

1.2 Key Components:

1.2.1 Inputs:

- The trace file is read, and the memory addresses are extracted from it.

```
with open('gcc.trace', 'r') as file1:
    # ins is a list that stores each line of input file on corresponding indices
    ins = file1.read().splitlines()

    # creating a list that will contain (only) the addresses from the ins
    addresses = []

    for line in ins:
        parts = line.split()
        addresses.append(parts[1])
```

Figure 1: Input

1.2.2 Cache Configuration:

- cache_size, `block_size`, and `ways_in_cache` are defined
- From these, the number of cache sets (`index_size`) and the number of tag bits are calculated.

```
##################
# we need to set these values inside the code itself
cache_size = 1024*1024  #bytes
block_size = 4          #bytes
ways_in_cache = 32       #ways
index_size = cache_size // (block_size * ways_in_cache)  #decimal number

cache_size_log = int(math.log2(cache_size))       #power of 2
block_size_log = int(math.log2(block_size))       #power of 2 #offset bits
ways_in_cache_log = int(math.log2(ways_in_cache)) #power of 2
index_size_log = int(math.log2(index_size))       #power of 2

tag_bits = 32 - index_size_log - block_size_log
##################
```

Figure 2: Cache Configuration

1.2.3 Cache Structure:

- The cache is a 3D list:
    - The first dimension represents the cache sets (or indices).

o The second dimension represents the blocks (or ways) in each set.

o Each block has three components:

1. Valid bit (indicating if the block is valid),
2. Tag bits (used to match the requested address),
3. Timestamp (for tracking LRU).

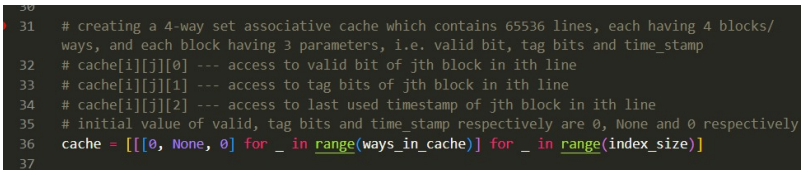

Figure 3: Cache Structure

1.2.4 Cache Access Logic:

• For each memory address, the address is broken down into tag, index, and offset.

• The cache is then checked for a hit (if a valid block with matching tag is found).

• If it's a miss, an empty block (if available) or the least recently used block is replaced with the new data.

1.2.5 Hit/Miss Counting:

• The script tracks the number of cache hits and misses, and updates the LRU timestamp to ensure that the least recently used block is replaced on future misses.

1.3 Flexibility:

• The code can handle any configuration of:

o Cache size

o Block size

o Associativity (number of ways)

• This makes it a dynamic and versatile simulation for different cache designs.

1.4 Output:

• At the end of the simulation, the code prints the total hits and misses, giving insights into the cache's performance for the given memory trace.

2 QUESTION (a)

The task involves designing a 4-way set associative cache with 1024 KB size and 4-byte blocks, using a 32-bit address. You need to calculate the number of cache lines and sets and implement cache logic to track hits and misses. The goal is to report the hit/miss rates for 5 provided memory trace files, without implementing the main memory.

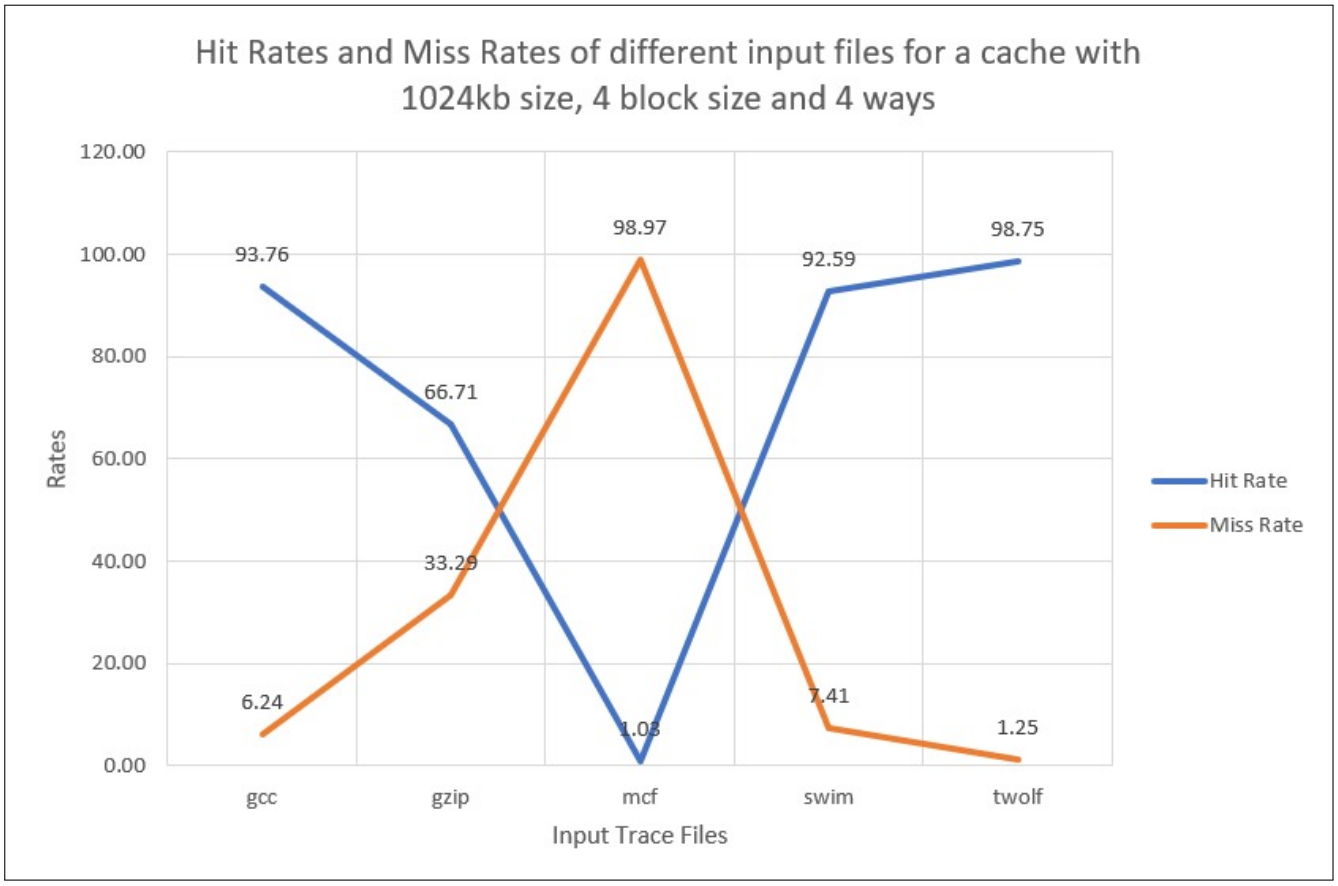

Figure 4: Results for all trace files

Observation

For the given configuration—block size of 4 bytes, cache size of 1024 kilobytes, and 4-way set associativity—the `twolf.trace` file shows the highest hit rate. This suggests that the program is effectively utilizing the cache with fewer tag collisions, benefiting from the 4-way associativity. In contrast, the `mcf.trace` file exhibits the lowest hit rate, indicating it may be accessing large data sets that exceed the cache capacity or would benefit from increased associativity to improve cache performance.

3 QUESTION (b)

The task involves varying the cache size from 128 KB to 4096 KB while keeping the block size and associativity constant. For each cache size, you will measure the hit/miss rates across the 5 provided trace files. After gathering the data, you need to plot a miss rate vs. cache size graph for all trace files on the same plot.

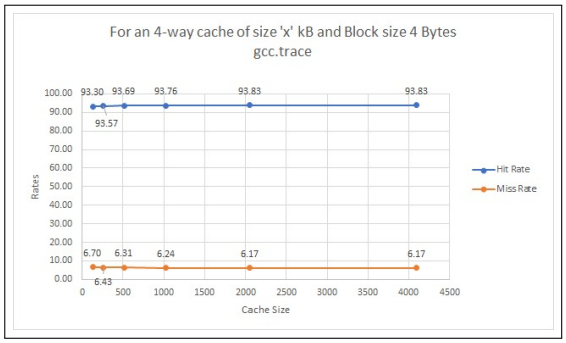

Figure 5: Results for gcc.trace file

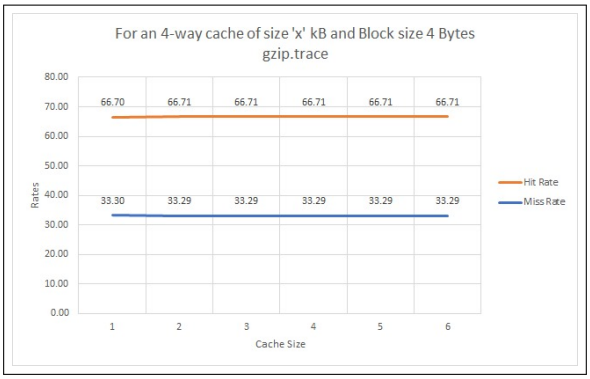

Figure 6: Results for gzip.trace file

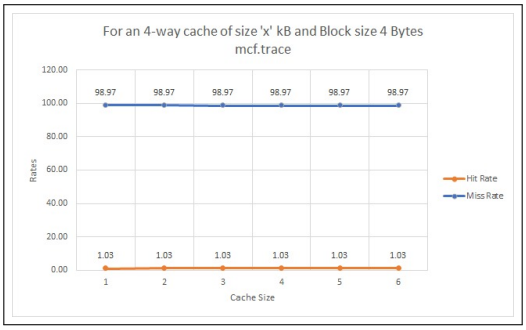

Figure 7: Results for mcf.trace file

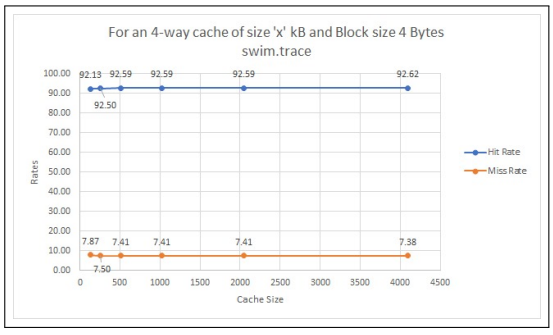

Figure 8: Results for swim.trace file

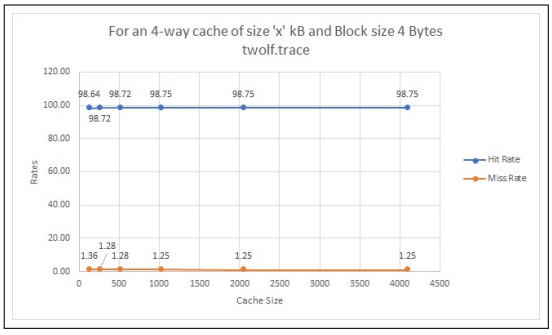

Figure 9: Results for twolf.trace file

Observation

By varying the cache size, we observed that the programs did not exhibit significant improvements in hit rates. This behavior is atypical, as larger cache sizes generally enhance temporal locality. The lack of improvement may indicate that these programs do not effectively leverage temporal locality. Instead, they might be making greater use of spatial locality, potentially due to extensive use of arrays or sequential memory accesses.

4 QUESTION (c)

For a fixed cache size of 1024 KB, vary the block size from 1 byte to 128 bytes and analyze the impact on cache performance. As the block size increases, the number of cache lines decreases. Repeat this experiment across all trace files and plot the miss rate against block size. Observe and compare how the miss rates change with different block sizes and determine if all trace files exhibit similar behavior.

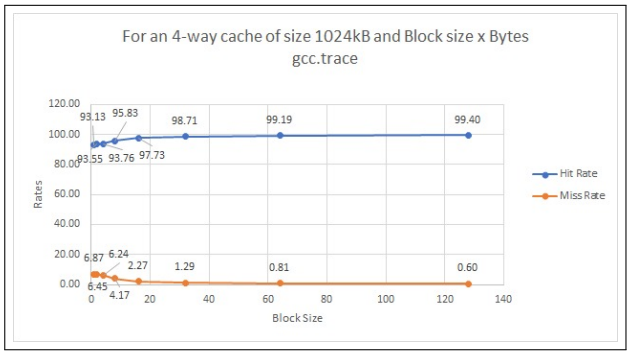

Figure 10: Results for gcc.trace file

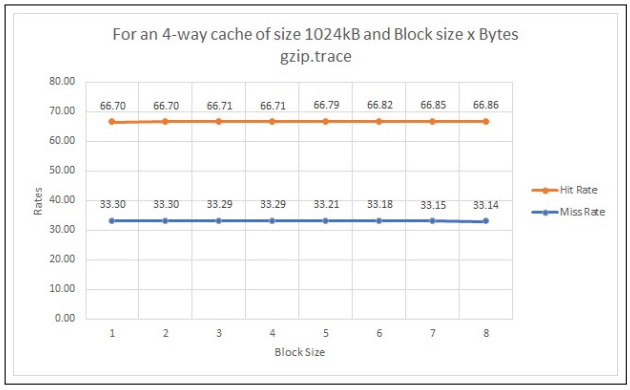

Figure 11: Results for gzip.trace file

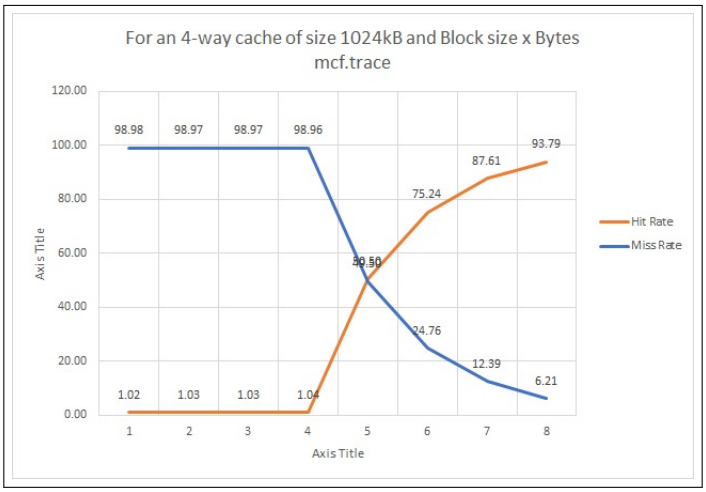

Figure 12: Results for mcf.trace file

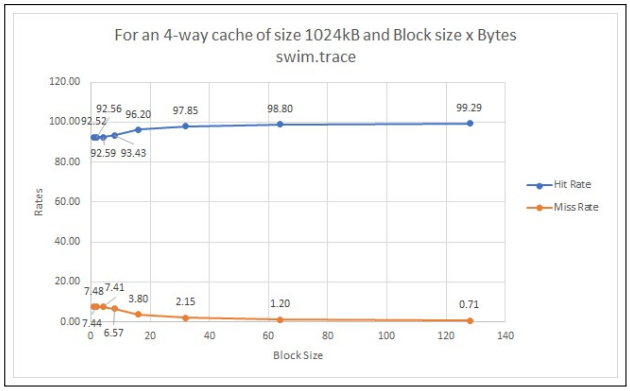

Figure 13: Results for swim.trace file

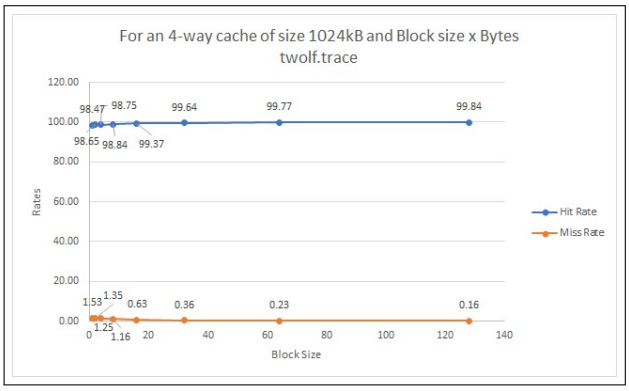

Figure 14: Results for twolf.trace file

Observation

With the exception of `gzip.trace`, all other programs experienced better hit rates as the block size increased. This suggests that these programs likely access consecutive memory locations more frequently than random addresses, indicating a preference for spatial locality over temporal locality.

5 QUESTION (d)

Vary the cache associativity from 1-way to 64-way while keeping the cache size fixed at 1024 KB. Plot the hit rates against associativity. Observe how hit rates change with varying levels of associativity and analyze why some trace files might exhibit different behaviors.

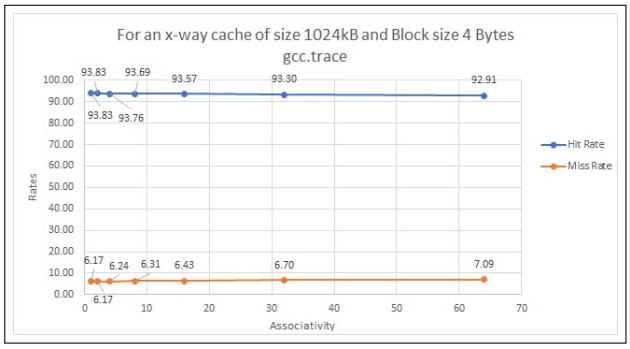

Figure 15: Results for gcc.trace file

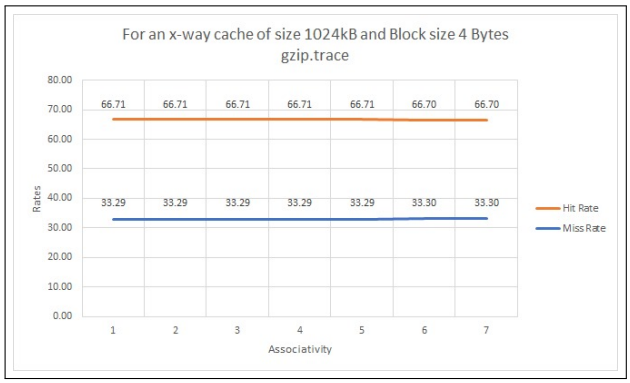

Figure 16: Results for gzip.trace file

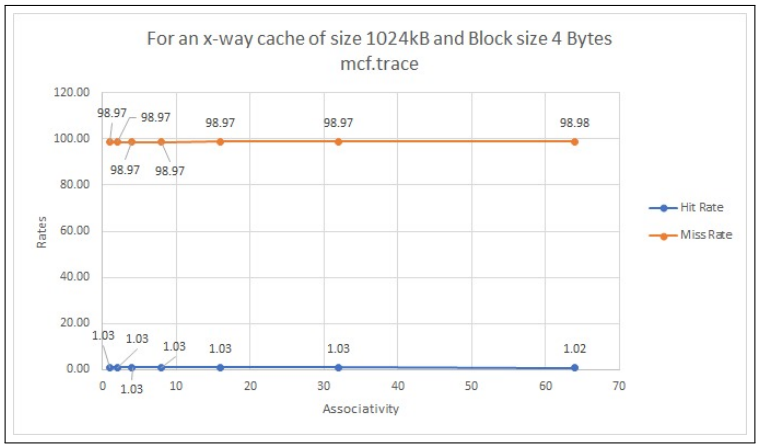

Figure 17: Results for mcf.trace file

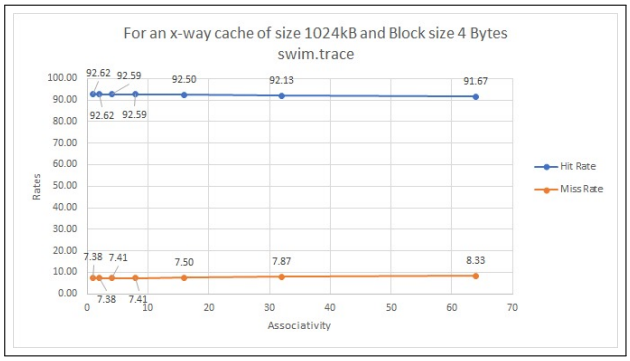

Figure 18: Results for swim.trace file

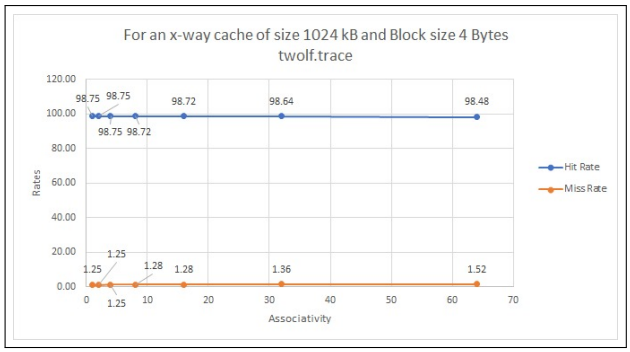

Figure 19: Results for twolf.trace file

Observation

The hit/miss rates for all files remained relatively stable with varying associativity, indicating that increasing associativity did not significantly improve performance. Notably, `twolf.trace` and `gcc.trace` showed only a slight increase of 0.0001 in hit rate, suggesting that these programs did not gain substantial benefits from higher associativity.

6 CONCLUSION

This cache simulation code successfully implements a dynamic set-associative cache with configurable parameters such as cache size, block size, and associativity (number of ways). It models realistic cache behavior using the Least Recently Used (LRU) replacement policy to determine which cache block should be replaced when the cache is full.

The flexibility of the code allows for testing different cache configurations, enabling users to explore how varying cache parameters affect hit and miss rates for memory traces. By adjusting the cache size, block size, and associativity, one can observe changes in cache performance, which can be invaluable for optimizing cache designs in real-world computer systems.