Cache Simulation Report

B Rutwik Chandra (CS22BTECH11011)

November 23, 2023

1. Introduction

1. This report presents the implementation, and testing of a cache simulation program in C++. The primary objective of this project is to create a versatile cache simulator capable of handling various cache configurations, replacement policies, and write access modeling. The simulation is driven by input files, namely "cache.config" for cache parameters and "cache.access" for memory access sequences.

2. The program is designed to provide insights into cache behavior, identifying cache hits and misses, and showcasing the impact of different cache configurations on overall performance. To achieve this, the simulation supports features such as direct-mapped, set-associative, and fully associative caches, as well as different replacement and write policies.

2. Code Overview and Parts Coverage

The code structure and functionalities are organized into distinct parts as outlined below, with successful completion and testing for each part :

1. **Part 1:** The code supports read access modeling for a direct-mapped cache with FIFO replacement policy.

2. **Part 2:** Extending from Part 1, the code incorporates LRU and RANDOM replacement policies for cache simulation.

3. **Part 3:** Building upon Part 2, the code introduces support for caches with associativity, allowing set-associative configurations.

4. **Part 4:** The code further expands functionality to include write access modeling, distinguishing between WriteBack (WB) and WriteThrough (WT) policies.

Each part of the code has been individually tested and validated using carefully crafted input files to ensure correct behavior and compliance with the assignment specifications.

3. Code Explanation

Configuration Handling

The code starts by reading the cache configuration from the "cache.config" file. The `read_config_file` function parses the file and assigns values to the `config` structure.

```
// Part of the code related to configuration handling
struct config read_config_file(std::string file_name) {
    // ... (Code for reading and assigning configuration values)
    return config;
}
```

Address Conversion Functions

Several functions handle the conversion between binary, hexadecimal, and decimal representations of memory addresses. These functions are for extracting tag, index, and offset information.

```
// Part of the code related to address conversion functions
std::string bin_to_hex(std::string bin_string) {
    // ... (Code for converting binary to hexadecimal)
    return "0x" + hex_stream.str();
}
```

Main Loop

It processes each line in the "cache.access" file in a loop, extracts the mode and address, and simulates cache access based on the specified replacement policies.

```
// Part of the code related to the main simulation loop
while (std::getline(input_file, line)) {
    // ... (Code for processing each line and simulating cache access)
}
```

Replacement Policies: FIFO

In FIFO replacement policy, the oldest address gets replaced first. The code checks for cache hits and misses and updates the cache accordingly.

```
// Part of the code related to FIFO replacement policy
if (config.rep_policy == "FIFO") {
    // ... (Code for FIFO replacement policy)
}
```

Replacement Policies: LRU

The LRU replacement policy is implemented, where the least recently used block is replaced upon a cache miss.

```
// Part of the code related to LRU replacement policy
} else if (config.rep_policy == "LRU") {
    // ... (Code for LRU replacement policy)
}
```

Replacement Policies: RANDOM

The RANDOM replacement policy is implemented, utilizing a random number generator to select the block to replace.

```
// Part of the code related to RANDOM replacement policy
} else if (config.rep_policy == "RANDOM") {
    // ... (Code for RANDOM replacement policy)
}
```

Cache Data Structure

A 2D vector(Vector of vector) named `set` is used to represent the cache sets and blocks, and it is updated based on cache hits and misses.

```
// Part of the code related to the cache data structure
std::vector<std::vector<std::string>> set(sets, std::vector<std::string>(
    config.associativity, "-1"));
```

Output

The progfram outputs information for each access, including the address, set, hit/miss status, and tag. It also keeps track of total hits and misses.

```
// Part of the code related to output and statistics
std::cout << "Address: " << hex_address << "\tSet: " << hex_index << "\tHit\
    tTag: " << hex_tag << std::endl;
total_hits++;
```

4. Testing Approach

For testing the cache simulation code, two input files (`input1.txt` and `input2.txt`) were generated. Each file contains 32-bit hexadecimal addresses, and the code was executed with these inputs using different cache configurations. The aim was to verify the total hits and misses for each configuration.

Input Addresses

The cache access file (`cache.access`) included the following sample addresses:

```
R: 0x29b4eaac
W: 0x20c2ed69
R: 0x1074468a
R: 0x597b35e9
R: 0x7fa8c985
R: 0x359b1083
W: 0x3fb9674f
W: 0x418654e1
W: 0x4f849b53
...
```

The file contained a total of 1000 addresses representing read (`R`) and write (`W`) operations.

Cache Configuration

The cache configuration file (`cache.config`) used for the tests was as follows:

```
Cache Size: 2048
Number of Sets: 16
Associativity: 8
Replacement Policy: LRU
Write Policy: Write Back
```

This configuration specified a cache size of 2048 bytes, 16 sets with associativity of 8, LRU as the replacement policy, and Write Back as the write policy.

Output Results

The output of the cache simulation code provided information for each access, including the address, set, hit/miss status, and tag. The total hits and misses were also recorded.

```
Address: 0x29b4eaac     Set: 0xa        Miss    Tag: 0x29b4ea
Address: 0x20c2ed69     Set: 0x6        Miss    Tag: 0x20c2ed
Address: 0x1074468a     Set: 0x8        Miss    Tag: 0x107446
Address: 0x597b35e9     Set: 0xe        Miss    Tag: 0x597b35
Address: 0x7fa8c985     Set: 0x8        Miss    Tag: 0x7fa8c9
...
Total Hits : 880        Total Misses : 120
```

The presented results show the detailed outcome of the cache simulation for the given inputs and configuration.

This testing approach allows for a comprehensive evaluation of the cache simulation code under different scenarios, aiding in understanding its behavior and performance characteristics.

5. Conclusion

In conclusion, the cache simulation code was rigorously tested using two input files (`input1.txt` and `input2.txt`), each containing 32-bit hexadecimal addresses representing read (`R`) and write (`W`) operations. The simulation was conducted under different cache configurations specified in the `cache.config` file.

The testing approach involved verifying the total hits and misses for each configuration. The output results displayed detailed information for each access, including the address, set, hit/miss status, and tag. The final tally of hits and misses was recorded for comprehensive evaluation.