Cache Simulation Report

DOKKU HEMANADH

November 23, 2023

1 Introduction

This report presents an analysis of a cache simulation program written in C, outlining its key functionalities, coding approach, and the testing strategy used.

2 Code Overview

The provided C code is a cache simulation program that emulates various cache replacement policies (FIFO, RANDOM, LRU). It includes the following key functions:

- **find_set_index**: Computes the set index bits for a given memory address.
- **find_tag**: Calculates the tag bits for a memory address.
- **search_tag_cache**: Searches for a specific tag within a cache set.
- **cache_set_full**: Checks if a cache set is full.

2.1 Code Explanation

Now, let's delve into the main function and its functionalities:

1. **Initialization**: - The program starts by reading configuration parameters from a file and initializes various variables based on these inputs, such as cache size, block size, ways, and replacement policies. - It computes essential bits required for set indexing and tagging, then initializes the cache array and necessary flag arrays.
2. **Memory Access Processing**: - The code proceeds to read memory access instructions from another file. For each memory access instruction: - It parses the mode (Read/Write) and memory address. - It calculates the set index and tag bits for the memory address using the `find_set_index` and `find_tag` functions, respectively.

3. **Cache Update Based on Policies**: - Depending on the specified policy (FIFO, RANDOM, LRU) and the read/write mode: - The program checks for hits or misses using the `search_tag_cache` function. - If a miss occurs, it updates the cache based on the replacement policy, either FIFO, RANDOM, or LRU, and increments hit or miss counters accordingly.

3 Coding Approach

The program begins by reading configuration parameters (cache size, block size, ways, policies) from a file. It initializes the cache structure using arrays, computes the required bits for set indexing and tagging, and sets up necessary data structures.

The main function processes memory access instructions (read/write) from another file. For each memory access, it calculates set index and tag bits, checks for hits or misses based on the specified replacement policies, and updates the cache accordingly.

4 Testing Methodology

To ensure the correctness and robustness of the code, the following testing methodologies were employed:

- **Manual Verification**: Multiple configuration files were created to cover different scenarios. The code was inspected for logical errors and edge cases.
- **Unit Testing**: Test cases were designed for individual functions to verify their correctness.
- **Random Test Cases**: Generated random test cases to evaluate the program's behavior under diverse scenarios.
- **Comparison with Expected Results**: Executed the program with known input/output data to validate the correctness of results.

5 Input and Output Formats

5.1 Input Format

The program expects input in two main formats:

1. **Configuration File**: - The first input file contains configuration parameters such as cache size, block size, ways, replacement policies, and write policies. - Each parameter is specified in a separate line within the file.

2. **Memory Access Instructions**: - The second input file contains memory access instructions in the form of read (R) or write (W) operations along with memory addresses. - Each instruction is listed in a separate line, following the format: `<Mode: Address>` (e.g., `R: 0x12345678`).

5.2 Output Format

Upon execution, the program produces output that displays cache simulation results:

- **Hits and Misses**: - The output shows the total number of hits and misses encountered during the simulation.
- **Cache Access Information**: - For each memory access instruction, the program displays information about the cache access, including set index, hit/miss status, and the corresponding tag.

The output format follows a structured representation where each cache access information is displayed in a formatted manner, providing insights into cache behavior for different memory accesses.

6 Conclusion

The cache simulation program demonstrates accurate behavior across various replacement policies. The employed testing methodologies confirm its correctness and reliability under different scenarios.