Cache Simulation Code Analysis

MANOHAR MARIPE

November 23, 2023

1 Introduction

Cache memory plays a crucial role in enhancing the performance of computer systems by storing frequently accessed data closer to the processor. This document presents an implementation of a cache simulation in C programming language. The simulation aims to mimic the behavior of a cache memory system with different replacement policies.

The C code provided leverages various concepts such as memory allocation, hexadecimal to binary conversion, tag extraction, and different cache replacement policies like Least Recently Used (LRU), First-In-First-Out (FIFO), and Random.

The following report provides an analysis of a C program designed to simulate a cache memory system. The code evaluates cache hits/misses and implements various replacement policies based on specified configurations

2 Code Structure

The provided C code is structured into different sections, including:

- Header inclusion for necessary libraries (`stdio.h`, `stdlib.h`, etc.)
- Definition of the `Node` structure to store tag and replacement policy information.
- Several helper functions to convert hexadecimal to binary, perform binary operations, and extract tag/set index from addresses.
- The main function that reads configuration settings, processes memory addresses, and simulates cache accesses.

3 Functions & Data Structures

The code contains several essential functions and structures:

- `createnode`: Creates a node with a specified tag value.
- Address conversion functions (`hexDigitToBinary`, `hexTo32BitBinary`, `binaryToDecimal`) for handling hexadecimal and binary operations.
- Utility functions (`breakCharArray`) to manipulate character arrays.
- Functions (`extracttag`, `extractsetindex`) to extract tag and set index from binary strings based on specified bit information.

4 Cache Simulation Flow

The simulation follows these key steps:

1. Parsing configuration settings from the `cache.config` file to determine cache parameters (size, block size, associativity, replacement policy, etc.).
2. Processing memory addresses from the `trace0.txt` file to extract set index, tag, and evaluate cache hits/misses based on read/write operations and policies.
3. Implementing a 2D array (`cache`) to simulate the cache structure.
4. Evaluating cache hits/misses and implementing replacement policies (LRU, FIFO, RANDOM) based on the observed access patterns.

5 Code Implementation

The code reads the cache.config file and stores all the values that help in building cache and all the properties and policies of cache.

Now the code reads again the trace.txt file line by line and starts parsing and extracting the tag and set index and now it starts simulating the cache by checking if the tag is present in the 2D matrix called cache. if present then increase hits if it got missed then increase the miss counter and check the policy and update accordingly.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
... % (Paste the entire C code here)
```

Listing 1: Cache Simulation Code

6 First-In-First-Out (FIFO) Policy

The First-In-First-Out (FIFO) replacement policy evicts the oldest entry in the cache when a new entry is added.

In the provided C code, the FIFO policy is implemented as follows:

1. When a cache miss occurs in a set:
   - The policy selects the node that was added earliest (oldest) for replacement. In code the FIFO works as we will check the total associativity and check for the minimum flag counter, the minimum flag gives the earliest inserted tag as while we miss the tag we insert that tag value in the cache and we are updating the flag of that cache using a global pointer that updates for every instruction or operation in the input file. This means that the minimum flag value is the first inserted tag in that set index. so we replace that.

```
if (strcmp(policy, "FIFO") == 0) {
    // Cache miss - Find node added earliest for replacement (oldest node)
    int minvalue = cache[setindex][0].replacepolicy;
    int replaceindex = 0;
    for (int y = 1; y < associativity; y++) {
        if (minvalue > cache[setindex][y].replacepolicy) {
            minvalue = cache[setindex][y].replacepolicy;
            replaceindex = y;
        }
    }
```

```
    // Replace the oldest node
    cache[setindex][replaceindex].tag = tag;
    cache[setindex][replaceindex].replacepolicy = replacementcheck;
    replacementcheck++;
}
```

Listing 2: FIFO Replacement Policy (Code Snippet)

7 Least Recently Used (LRU) Policy

The Least Recently Used (LRU) replacement policy aims to evict the least recently accessed item from the cache when space is needed for a new entry.

In the provided C code, the LRU policy is implemented as follows:

1. When a cache miss occurs in a set:
   - In this LRU policy everything is the same as the FIFO policy except one thing even if got hit any tag in the cache we will update the flag pointer to that tag by updating the global pointer. As we need the least recently used tag to replace, we are updating even if we got hit as hit means that the tag is accessed recently so we need to update the flag.
   - The policy updates the "replace-policy" field for the accessed node to indicate its access order.
   - The node with the lowest "replacepolicy" value (indicating least recently used) is chosen for replacement.
2. If a cache hit occurs, the policy updates the "replace policy" for the accessed node.

```
if (strcmp(policy, "LRU") == 0) {
    // Cache hit - Update replacepolicy for accessed node
    // Cache miss - Find least recently used node for replacement
    int minvalue = cache[setindex][0].replacepolicy;
    int replaceindex = 0;
    for (int y = 1; y < associativity; y++) {
        if (minvalue > cache[setindex][y].replacepolicy) {
            minvalue = cache[setindex][y].replacepolicy;
            replaceindex = y;
        }
    }
    // Replace node with the lowest replacepolicy value
    cache[setindex][replaceindex].tag = tag;
    cache[setindex][replaceindex].replacepolicy = replacementcheck;
    replacementcheck++;
}
```

Listing 3: LRU Replacement Policy (Code Snippet)

8 Random Replacement Policy

The Random replacement policy selects a random entry from the cache for replacement.

In the provided C code, the Random policy is implemented as follows:

1. When a cache miss occurs in a set:
   - In the RANDOM replacement policy we will use srand() and rand() functions to generate some index between range (0 - ways/associativity) so that we can replace that indexed tag with the new one.
   - The policy randomly selects a node from the set for replacement.

```
if (strcmp(policy, "RANDOM") == 0) {
    // Cache miss - Randomly select a node from the set for replacement
    int range = associativity - 0 + 1;
    srand((unsigned int)time(NULL));
    int replaceindex = rand() % range + 0;
    // Replace the randomly selected node
```

```
    cache[setindex][replaceindex].tag = tag;
    cache[setindex][replaceindex].replacepolicy = replacementcheck;
    replacementcheck++;
}
```

Listing 4: Random Replacement Policy (Code Snippet)

9 Conclusion

The provided code showcases a comprehensive simulation of a cache memory system, demonstrating the practical implementation of various cache-related concepts. The analysis emphasizes the significance of replacement policies in optimizing cache performance based on different access patterns. In conclusion, the provided C code showcases a comprehensive simulation of a cache memory system. It effectively handles configuration parsing, memory address processing, and cache simulation. The code demonstrates the practical implementation of various cache-related concepts, including address conversion, cache organization, and replacement policies. The analysis underscores the significance of well-designed cache management for efficient memory access in computing systems.

10 Additional Observations

Potential areas for improvement or additional features include:

- Enhancing error handling mechanisms for memory allocation failures or invalid inputs.
- Incorporating more sophisticated replacement policies or cache management strategies for further optimization.
- Adding detailed logging or reporting mechanisms for a deeper understanding of cache behaviour during simulation.