Cache Simulator Report

Introduction

The cache simulator project aims to replicate the behavior of a cache memory system, allowing customization for various cache parameters such as size, block size, associativity, replacement policy, and write policy. The program reads the cache configuration from the "cache.config" file and processes the access sequence from the "cache.access" file. This report provides insights into the implementation of each sub-part of the assignment, detailing the coding approach, design decisions, and testing strategies for each phase.

Part-1: Read Access Modeling with FIFO Replacement Policy

Coding Approach

In this phase, the program focuses on simulating read access for a direct-mapped cache with FIFO replacement policy. The cache is organized into sets, and each set contains only one block. The simulator tracks cache hits and misses, the corresponding index, and the stored TAG for each access.

As for FIFO implementation, it was done by storing the line number of access while storing it in a matrix name cache_rep and removing the least values for a given row while replacing thereby letting FIFO happen at an overall level.

Design Decisions

The implementation involves a straightforward approach, where the cache is modeled as an array with each element representing a set. A matrix is used for managing the replacement policy. The index is calculated based on the cache size and block size, and the TAG is extracted from the memory address.

Testing Strategies

Unit testing is conducted to verify the correctness of the cache simulator for read accesses in a direct-mapped cache with FIFO replacement policy. Specific test cases include scenarios with various cache sizes, block sizes, and access sequences.

Part 2: Addition of LRU and RANDOM Replacement Policies

Coding Approach

This phase extends the simulator to support LRU and RANDOM replacement policies in addition to FIFO. The program now evaluates cache hits and misses based on the chosen replacement policy and outputs the corresponding index and stored TAG for each access.

As for LRU policy, it uses almost the same technique as FIFO with the only difference being that every time a cache hits the corresponding cache_rep data is updated thereby removing the last used data when cache is full.

For random it is different ,random number is generated and we remove element at that place.

Design Decisions

The implementation involves introducing data structures to manage LRU information or select a block randomly for replacement. LRU information is updated on each cache access to prioritize the least recently used block.

Testing Strategies

Extensive testing is performed to ensure the correct functioning of the simulator with the addition of LRU and RANDOM replacement policies. Test cases cover combinations of cache sizes, block sizes, and access patterns to validate the simulator's adaptability.

Part-3: Inclusion of Write Access Modeling

Coding Approach

This final phase extends the simulator to support write access modeling. The program now handles both read and write accesses, implementing the WriteThrough and WriteBack policies as specified in the assignment.

For write-back and write through there is almost no visible change in the cache tag part and index part so our goal is to verify if that newly rewritten value is present in the cache or not for the case of write-through with no allocation so even if there is a miss the new data will not be loaded into the cache so we have to delete this mapping to protect the program from taking wrong data.

Design Decisions

The implementation involves distinguishing between read and write accesses and incorporating the specified write policies. For WriteBack, dirty bit information is maintained to track modified blocks.

Testing Strategies

Comprehensive testing is performed to validate the simulator's correct handling of both read and write accesses. Test cases cover various combinations of cache parameters, write policies, and access sequences.

Conclusion

The cache simulator is designed to be flexible, accommodating various cache configurations and replacement policies. Through the progressive implementation of each sub-part, the simulator evolves to handle read and write accesses in direct-mapped, set-associative, and fully associative cache designs. Rigorous testing procedures ensure the correctness and reliability of the simulator across different scenarios.