AI22BTECH11020
Pranay jain

REPORT

Introduction:

This cache simulator program is designed to model the behavior of a cache system based on various parameters specified in the 'cache.config' file. The simulation supports different cache configurations, including total cache size, block size, associativity, replacement policy, and write policy. The program reads access sequences from the 'cache.access' file and outputs information about each access, such as the index/set, whether it is a hit or miss, and the stored tag in the cache.

hexCharToDecimal(char hex):

- This function takes a single hexadecimal character as input and converts it into its decimal equivalent. Hexadecimal characters represent numbers in base-16, and this conversion is crucial for further calculations and comparisons.

hexToBinary(const char hexString):

- Given a hexadecimal string as input, this function converts the entire string into its binary representation. This binary conversion is essential for addressing and bitwise operations involved in cache simulation.

extractAddress(char address, int setIndex, char* tag, char* offsetString, char* indexString, char* tagString):

- This function parses the memory address into its constituent parts, including the set index, tag, and offset. It separates these components and stores them in corresponding strings, allowing for easier manipulation and analysis during cache simulation.

binaryToHex(char binaryString):

- Performs the reverse operation of hexToBinary. It takes a binary string as input and converts it into its hexadecimal representation. This conversion is useful for displaying addresses and data in a more human-readable format.

binaryToDecimal(const char binaryString):

- Converts a binary string into its decimal equivalent. This function is valuable for interpreting binary values, particularly when dealing with numerical calculations and comparisons.

simulateReadAccessAssociative(char mode, char address, int Indexbit, char arr2[Indexbit][ASSOCIATIVITY], int valid[Indexbit][ASSOCIATIVITY], int dirty[Indexbit][ASSOCIATIVITY]):

- It uses RANDOM replacement policy. It Simulates a read access in a set-associative cache, considering the specified replacement policy and associativity. It determines whether the requested data is present in the cache (a hit) or needs to be fetched from the main memory (a miss). This function also updates the cache state based on the read access.

simulateWriteAccessWB(char mode, char address, int Indexbit, char arr2[Indexbit][ASSOCIATIVITY], int valid[Indexbit][ASSOCIATIVITY], int dirty[Indexbit][ASSOCIATIVITY]):

- Simulates a write access with a write-back policy in a set-associative cache. It involves updating the cache state based on the write operation and handling the potential need to write data back to the main memory. This is done by Random Replacement policy

simulateWriteAccessWT(char mode, char address, int Indexbit, char arr2[Indexbit][ASSOCIATIVITY], int valid[Indexbit][ASSOCIATIVITY], int dirty[Indexbit][ASSOCIATIVITY]):

- Simulates a write access with a write-through policy in a set-associative cache. Similar to simulateWriteAccessWB, it updates the cache state but immediately writes the

modified data back to the main memory. This is done by Random Replacement policy.

simulateReadAccessAssociativeFIFO(char mode, char address, int Indexbit, char arr2[Indexbit][ASSOCIATIVITY], int FIFO_Index[Indexbit], int valid[Indexbit][ASSOCIATIVITY], int dirty[Indexbit][ASSOCIATIVITY]):

- Simulates a read access in a set-associative cache using a FIFO (First In, First Out) replacement policy. It determines cache hits or misses and updates the cache state accordingly, considering the order in which entries were placed in the cache.

simulateReadAccessAssociativeLRU(char mode, char address, int Indexbit, char arr2[Indexbit][ASSOCIATIVITY], int LRU_Index[Indexbit][ASSOCIATIVITY], int valid[Indexbit][ASSOCIATIVITY], int dirty[Indexbit][ASSOCIATIVITY]):

- Simulates a read access in a set-associative cache using an LRU (Least Recently Used) replacement policy. It identifies cache hits or misses and updates the cache state while keeping track of the usage history of cache lines.

simulateWriteAccessLRUWB(char mode, char address, int Indexbit, char arr2[Indexbit][ASSOCIATIVITY], int LRU_Index[Indexbit][ASSOCIATIVITY], int valid[Indexbit][ASSOCIATIVITY], int dirty[Indexbit][ASSOCIATIVITY]):

- Simulates a write access with a write-back policy in a set-associative cache using the LRU replacement policy. It updates the cache state, marks the appropriate cache line as dirty, and handles the potential need to write data back to the main memory.

simulateWriteAccessLRUWT(char mode, char address, int Indexbit, char arr2[Indexbit][ASSOCIATIVITY], int LRU_Index[Indexbit][ASSOCIATIVITY], int valid[Indexbit][ASSOCIATIVITY], int dirty[Indexbit][ASSOCIATIVITY]):

- Simulates a write access with a write-through policy in a set-associative cache using the LRU replacement policy. It updates the cache state and immediately writes the modified data back to the main memory while considering the usage history of cache lines.

simulateWriteAccessFIFOWB(char mode, char address, int Indexbit, char arr2[Indexbit][ASSOCIATIVITY], int FIFO_Index[Indexbit], int valid[Indexbit][ASSOCIATIVITY], int dirty[Indexbit][ASSOCIATIVITY]):

- Simulates a write access with a write-back policy in a set-associative cache using the FIFO replacement policy. It updates the cache state, marks the appropriate cache line as dirty, and handles the potential need to write data back to the main memory, considering the order of entry placement.

simulateWriteAccessFIFOWT(char mode, char address, int Indexbit, char arr2[Indexbit][ASSOCIATIVITY], int FIFO_Index[Indexbit], int valid[Indexbit][ASSOCIATIVITY], int dirty[Indexbit][ASSOCIATIVITY]):

- Simulates a write access with a write-through policy in a set-associative cache using the FIFO replacement policy. It updates the cache state and immediately writes the modified data back to the main memory, considering the order of entry placement in the cache.

main():

The main function of the program plays a pivotal role in orchestrating the cache simulation. It begins by reading the cache configuration parameters, such as cache size, block size, associativity, and replacement policy, from the 'cache.config' file. Subsequently, it processes access sequences from the 'cache.access' file, matching each access against the specified cache policies, associativity, and write-back/write-through policies. Depending on these configurations, it then invokes the corresponding functions to simulate read and write accesses in the set-associative cache. This function serves as the central control point, ensuring that the cache simulation accurately

reflects the behavior of the cache system under diverse conditions and policies.

Testing:

The cache simulator has been done testing to ensure correctness across various scenarios. It was tested with all six possible combinations of read and write accesses, covering different cache policies, associativity settings, and write-back/write-through policies. The tests included:

Read-Only Access:

- Checked accuracy in detecting cache hits and misses under various configurations.

Write-Only Access:

- Assessed proper handling of write operations with different cache settings.

Read and Write Access (FIFO Replacement, Write-Back):

- Validated the simulator's performance with read and write accesses using FIFO replacement and write-back policies.

Read and Write Access (LRU Replacement, Write-Back):

- Tested the simulator's ability to manage data with LRU replacement and write-back policies.

Read and Write Access (Random Replacement, Write-Back):

- Ensured correct handling of random cache replacements and write-back operations.

Read and Write Access (FIFO Replacement, Write-Through):

- Checked the simulator's behavior with FIFO replacement and write-through policies.

Read and Write Access (LRU Replacement, Write-Through):

- Validated proper data management with LRU replacement and write-through policies.

Read and Write Access (Random Replacement, Write-Through):

- Assessed the simulator's capability in handling random replacements and immediate write-through updates.

The combination of all of them are submitted in the txt file.