Cache Simulation using Fully Associative Mapping

Problem Statement

The problem involves simulating a cache system with the following specifications:

- Main Memory size: 64K words
- Cache size: 2K words
- Block size: 16 words
- Word size: 32 bits
- Processor address: 16 bits
- Cache mapping: Fully associative
- Access pattern to simulate: Spatial access, Temporal access, and Mixed access (read and write)
- The objective is to estimate cache misses, number of cache searches, and hit/miss statistics.

The assumptions made for this simulation are as follows:

- The main memory always contains the required data, and there is never a need to access the hard disk.
- The access is performed with a fully associative cache mapping scheme.
- The cache uses the Least Recently Used (LRU) policy for cache replacement.
- The memory accesses are simulated for both read and write operations.

C++ Code

The following C++ code simulates the behavior of a fully associative cache system with spatial and temporal access patterns:

Listing 1: Cache Simulation C++ Code

```cpp
#include <iostream>
#include <vector>
#include <climits>

class CacheBlock {
public:
    bool valid;               // Valid bit, indicates if the block
        contains valid data
    bool dirty;               // Dirty bit, indicates if the block
        has been modified
    int tag;                  // Tag used for identifying the block
         in the cache
    int lastAccessTime;       // Time of last access for LRU
        replacement policy
    std::vector<int> data;    // Data stored in the cache block

    // Constructor to initialize a cache block
    CacheBlock(int blockSize) {
        valid = false;         // Initially, the block is invalid
        dirty = false;         // Initially, the block is not dirty
        tag = -1;              // Initially, no tag
        lastAccessTime = 0;    // Initially, no access time
        data.resize(blockSize); // Resize the data vector to hold
            the block's data
    }
};

class Cache {
private:
    std::vector<CacheBlock> cache; // Vector holding the cache
        blocks
    int numBlocks;                 // Number of blocks in the cache
    int blockSize;                 // Size of each block (number of
         words)
    int currentTime;               // Current time for LRU tracking
    int cacheMisses;               // Counter for cache misses
    int readMisses;                // Counter for read misses
    int writeMisses;               // Counter for write misses
    int cacheSearches;             // Counter for the number of
        cache searches

public:
    // Constructor to initialize the cache with the number of
        blocks and block size
    Cache(int numBlocks, int blockSize) : numBlocks(numBlocks),
        blockSize(blockSize), currentTime(0),
        cacheMisses(0), readMisses(0), writeMisses(0),
            cacheSearches(0) {
        cache.resize(numBlocks, CacheBlock(blockSize)); // Resize
            the cache to hold the specified number of blocks
    }
```

```
// Method to access the cache
void access(int memoryAddress, bool write) {
    currentTime++;            // Increment current time
    cacheSearches++;          // Increment the number of cache
        searches

    int blockOffsetBits = 4;  // Since the block size is 16
        words, 4 bits are used for the block offset
    int tag = memoryAddress >> blockOffsetBits; // Extract the
        tag from the memory address

    int blockIndex = -1;
    bool hit = false;

    // Search through the cache for a block with the same tag
    for (int i = 0; i < numBlocks; ++i) {
        if (cache[i].valid && cache[i].tag == tag) {
            blockIndex = i;
            hit = true;
            cache[i].lastAccessTime = currentTime; // Update
                the access time for the block
            break;
        }
    }

    // If there is no hit, perform a cache miss
    if (!hit) {
        cacheMisses++; // Increment the cache miss counter
        if (write) {
            writeMisses++;  // Increment write misses if it's a
                 write operation
        } else {
            readMisses++;   // Increment read misses if it's a
                read operation
        }

        // Find the Least Recently Used (LRU) block to replace
        int lruIndex = -1;
        int minTime = currentTime;

        // LRU replacement policy
        for (int i = 0; i < numBlocks; ++i) {
            if (!cache[i].valid || cache[i].lastAccessTime <
                minTime) {
                lruIndex = i;
                minTime = cache[i].lastAccessTime;
            }
        }

        blockIndex = lruIndex;

        // Replace the cache block with the new tag
        cache[blockIndex].valid = true;
        cache[blockIndex].tag = tag;
        if (write) {
                cache[blockIndex].dirty = true;  // Mark the block
                    as dirty if it's a write
            }

            cache[blockIndex].lastAccessTime = currentTime;  //
                Update the last access time
        } else {
            if (write) {
                cache[blockIndex].dirty = true;  // Mark the block
                    as dirty if it's a write hit
            }
        }
    }

    // Method to print cache statistics
    void printStats() {
        std::cout << "Cache␣Misses:␣" << cacheMisses << std::endl;
        std::cout << "Cache␣Searches:␣" << cacheSearches << std::
            endl;
        std::cout << "Cache␣Hit␣Rate:␣" << (1.0 - (double)
            cacheMisses / cacheSearches) * 100 << "%" << std::endl;
        std::cout << "Read␣Misses:␣" << readMisses << std::endl;
        std::cout << "Write␣Misses:␣" << writeMisses << std::endl;
    }
};

int main() {
    // Initializing cache parameters
    int numBlocks = 128;  // Cache has 128 blocks
    int blockSize = 16;   // Each block contains 16 words

    // Create a cache object
    Cache cache(numBlocks, blockSize);

    // Simulating spatial access - Read and Write
    std::cout << "Simulating␣Spatial␣Access␣-␣Read:" << std::endl;
    for (int i = 0; i < 1000; i++) {
        cache.access(i, false);  // Read access
    }
    cache.printStats();

    std::cout << "Simulating␣Spatial␣Access␣-␣Write:" << std::endl;
    for (int i = 0; i < 2000; i++) {
        cache.access(i, true);  // Write access
    }
    cache.printStats();

    // Simulating temporal access
    std::cout << "Simulating␣Temporal␣Access␣-␣Read:" << std::endl;
    for (int i = 0; i < 1000; i++) {
        cache.access(i, false);  // Read access to addresses 0 to
            999
    }
    cache.printStats();

    std::cout << "Simulating␣Temporal␣Access␣-␣Write:" << std::endl
        ;
    for (int i = 0; i < 4000; i++) {
        cache.access(i, true);  // Write access
    }
    cache.printStats();

    return 0;
}
```

Explanation of the Code

CacheBlock Class

The `CacheBlock` class represents a single block in the cache. It contains the following attributes:

- `valid`: A boolean that indicates whether the block contains valid data.
- `dirty`: A boolean that indicates whether the block has been modified.
- `tag`: The tag that identifies the block's memory location.
- `lastAccessTime`: The timestamp of the last access to the block used for LRU replacement.
- `data`: A vector that holds the actual data stored in the block.

The constructor initializes these attributes, with `valid` set to `false`, `dirty` set to `false`, and `tag` set to -1. The `data` vector is resized based on the block size.

Cache Class

The `Cache` class represents the cache itself and performs the caching operations. It contains:

- `cache`: A vector of `CacheBlock` objects representing the cache blocks.
- `numBlocks`: The number of blocks in the cache.
- `blockSize`: The number of words per block.
- `currentTime`: The current time used to track the least recently used block.
- `cacheMisses`, `readMisses`, `writeMisses`, `cacheSearches`: Counters for cache misses, read/write misses, and cache searches respectively.

The `access` method performs a memory access (either read or write) and updates the cache state accordingly. If a miss occurs, it updates the cache using the LRU policy.

The `printStats` method prints the cache statistics, including the number of misses, hits, and hit rate.