Implementation of the MSI Protocol for Cache Coherence: Snooping and Directory Approaches

Hardik Chauhan (Roll No. 21EC39011)

Link: https://github.com/H-achiever17/MSI-Protocol-for-Cache-Coherence

Abstract

In the realm of multiprocessor systems, the cache coherence problem presents a significant challenge due to the presence of multiple caches that may hold copies of the same memory location. The MSI (Modified, Shared, Invalid) protocol is a fundamental cache coherence protocol designed to address this issue by maintaining a consistent view of memory across all processors. This report provides an extensive, end-to-end analysis and implementation of the MSI protocol using both snooping and directory-based approaches. We begin by exploring the theoretical foundations of cache coherence, delving into detailed explanations of the MSI protocol, including its states, state transitions, and the mechanisms of snooping and directory-based coherence. Following this, we present a comprehensive overview of our codebase, including detailed code explanations and design decisions. We also include execution logs, state transition diagrams, and performance metrics such as hit-miss rates to illustrate the effectiveness of our implementation. The report concludes with discussions on observations, challenges faced, potential optimizations, and future work directions.

Contents

1 Introduction

1.1 Background

The evolution of computing technology has led to an increased demand for high-performance systems capable of handling complex computations and large datasets. Multiprocessor systems, which utilize multiple processors to perform parallel computations, have become integral in meeting these demands. Each processor in such systems often has its own cache memory to reduce access latency to frequently used data.

However, the use of multiple caches introduces the **cache coherence problem**, where inconsistencies can occur if multiple caches hold copies of the same memory location and at least one cache modifies the data without updating the others. This inconsistency can lead to incorrect program execution and data corruption.

1.2 The Cache Coherence Problem

When a processor modifies a data item in its cache, other caches with copies of that data may not immediately see the update. Ensuring that all caches reflect the most recent value of shared data is critical for program correctness. The cache coherence problem is thus a fundamental challenge in multiprocessor systems.

1.3 The MSI Protocol

The **MSI (Modified, Shared, Invalid)** protocol is one of the simplest cache coherence protocols designed to address this issue. It categorizes each cache line into one of three states:

- **Modified (M):** The cache line is valid, modified, and exclusively held by one cache.
- **Shared (S):** The cache line is valid, unmodified, and may be present in multiple caches.
- **Invalid (I):** The cache line is invalid or not present in the cache.

By defining rules for state transitions based on processor and bus actions, the MSI protocol ensures that all processors have a consistent view of memory.

1.4 Objectives

The primary objectives of this report are:

- To provide a comprehensive theoretical background on cache coherence and the MSI protocol.

- To present an end-to-end implementation of the MSI protocol using both snooping and directory-based approaches.
- To explain the codebase in detail, including design decisions and implementation challenges.
- To analyze the performance of the implementations through execution logs and performance metrics.
- To discuss observations, challenges, and potential optimizations.
- To outline future work directions, including optimizing the directory-based implementation and extending to advanced protocols.

2 Theory

2.1 Multiprocessor Systems and Cache Coherence

2.1.1 Introduction to Multiprocessor Systems

A **multiprocessor system** is a computer system with two or more central processing units (CPUs) that share a common memory and are interconnected by a bus or other communication medium. These systems are designed to perform parallel processing, where multiple processors execute different parts of a program simultaneously to improve performance.

In multiprocessor systems, each processor typically has its own cache memory. Caches are small, fast storage units that keep copies of frequently accessed data to reduce the average time to access memory. However, the presence of multiple caches introduces new challenges in maintaining data consistency across the system.

2.1.2 The Cache Coherence Problem

Cache coherence refers to the consistency of shared resource data that ends up stored in multiple local caches. In a multiprocessor system, when one processor updates a memory location in its cache, and other processors have cached copies of the same memory location, inconsistencies can arise.

Key Challenges:

- **Data Inconsistency:** Multiple caches may hold different values for the same memory location.
- **Stale Data Access:** Processors may read outdated data, leading to incorrect computations.
- **Synchronization Overhead:** Maintaining coherence adds complexity and overhead to the system.

To address these challenges, cache coherence protocols are implemented to ensure that all caches reflect the most recent value of shared data.

2.2 Cache Coherence Protocols

Cache coherence protocols define the rules and mechanisms by which caches interact to maintain a consistent view of memory across a multiprocessor system.

2.2.1 Write-Invalidate vs. Write-Update Protocols

There are two primary types of cache coherence protocols:

1. **Write-Invalidate Protocols:**
    - When a processor writes to a cache line, it invalidates all other copies of that line in other caches.
    - Subsequent reads by other processors result in a cache miss, and the data must be fetched from memory or the cache that modified it.
2. **Write-Update Protocols:**
    - When a processor writes to a cache line, it updates all other caches holding that line with the new value.
    - Ensures that all caches have the most recent data without invalidating the cache line.

The MSI protocol is a **write-invalidate** protocol.

2.2.2 Snooping vs. Directory-Based Protocols

Cache coherence protocols can also be categorized based on how coherence information is maintained:

1. **Snooping Protocols:**
    - All caches monitor a shared communication medium (bus) to observe transactions that may affect their cached data.
    - Coherence actions are taken by caches that ”snoop” on relevant bus transactions.
2. **Directory-Based Protocols:**
    - A directory maintains coherence information, tracking which caches have copies of each memory block.
    - Processors communicate with the directory to perform coherence operations, reducing unnecessary broadcasting.

2.3 The MSI Protocol

2.3.1 States in the MSI Protocol

The MSI protocol defines three states for each cache line:

1. **Modified (M):**
    - The cache line is present only in the current cache.
    - The data is modified (dirty) and differs from main memory.
    - No other cache holds this data.
2. **Shared (S):**
    - The cache line is unmodified and may exist in multiple caches.
    - The data matches the main memory.
    - The cache may read the data but must notify other caches before writing.
3. **Invalid (I):**
    - The cache line is invalid or not present.
    - The data cannot be used until it is fetched from memory or another cache.

2.3.2 State Transition Rules

State transitions in the MSI protocol occur based on processor actions (read or write operations) and bus transactions. The following table summarizes the state transitions:

Current State	Processor Action	Bus Transaction	New State	Notes
I	Read Miss	BusRd	S	Load data from memory or another cache
I	Write Miss	BusRdX	M	Load data and gain exclusive access
S	Read Hit	None	S	Remain in Shared state
S	Write Hit	BusRdX	M	Invalidate other caches
M	Read/Write Hit	None	M	Remain in Modified state
S/M	Another BusRd	BusRd	S	Respond with data if in M state
M	Another BusRdX	BusRdX	I	Write back data, invalidate cache line

Table 1: MSI Protocol State Transition Table

2.3.3 State Transition Diagrams

2.4 Snooping-Based Cache Coherence

2.4.1 Operation Mechanism

In the snooping protocol, all caches are connected to a shared bus and monitor (snoop) the bus to detect relevant coherence transactions.

Key Components:

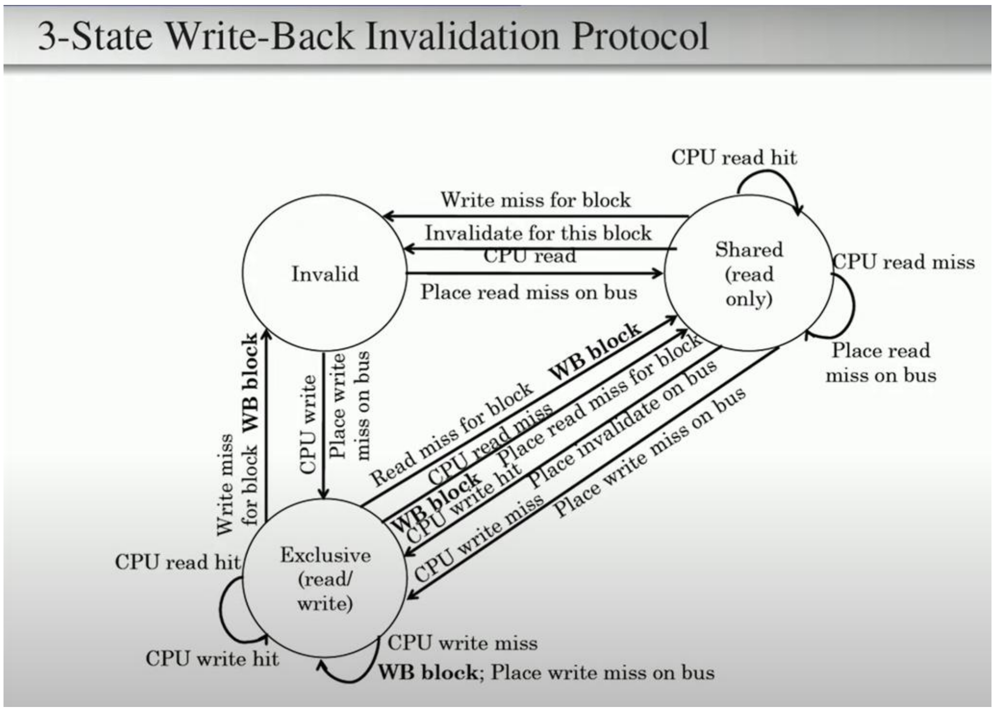


Figure 1: Snoop Based 3 state write back invalidation protocol

- **Shared Bus:** A communication medium that all caches can observe.
- **Snoop Logic:** Hardware within each cache that monitors bus transactions.

Operation Steps:

1. **Processor Issues Request:**
    - On a cache miss, the processor issues a read or write request on the bus.
2. **Caches Snoop Bus:**
    - All caches monitor the bus for transactions.
    - If a cache detects a transaction for a memory location it holds, it takes appropriate action.
3. **Coherence Actions:**
    - **Read Miss:**
        – If another cache has the data in Modified state, it provides the data and transitions to Shared.
    - **Write Miss:**
        – All other caches invalidate their copies of the cache line.

2.4.2 Advantages and Limitations

Advantages:

- **Simplicity:** Straightforward implementation for small systems.
- **Immediate Coherence:** Coherence actions are promptly taken due to continuous monitoring.

Limitations:

- **Scalability Issues:** Bus contention increases with more processors.
- **High Broadcast Traffic:** Every transaction is broadcasted, leading to inefficiency in larger systems.

2.5 Directory-Based Cache Coherence

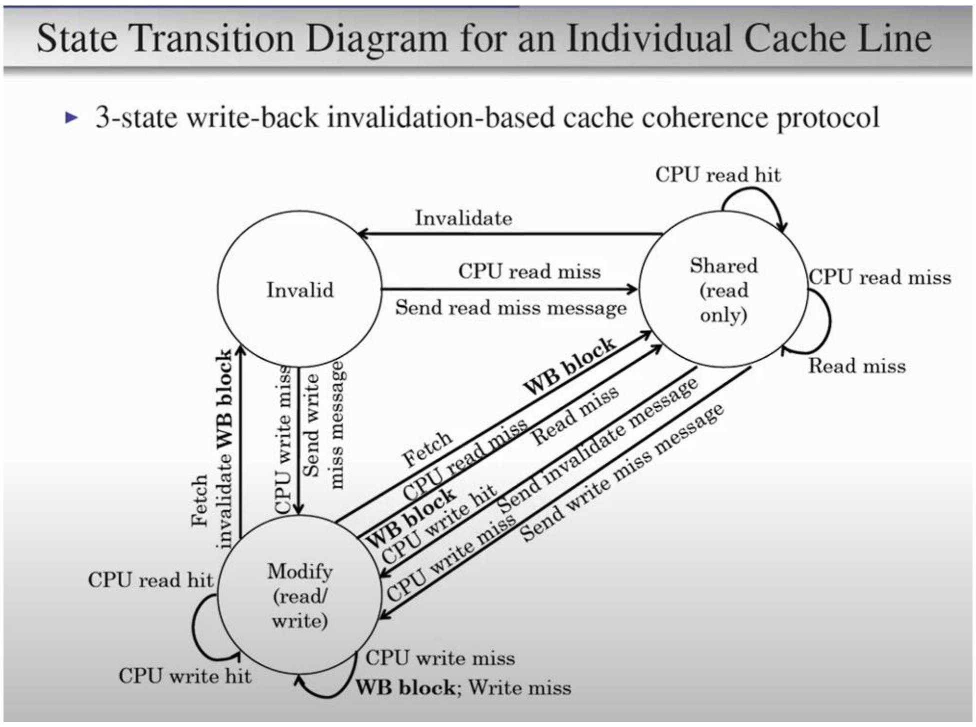


Figure 2: Individual Cache Line

2.5.1 Operation Mechanism

In the directory-based protocol, a directory keeps track of the states of memory blocks and which caches hold copies.

Key Components:

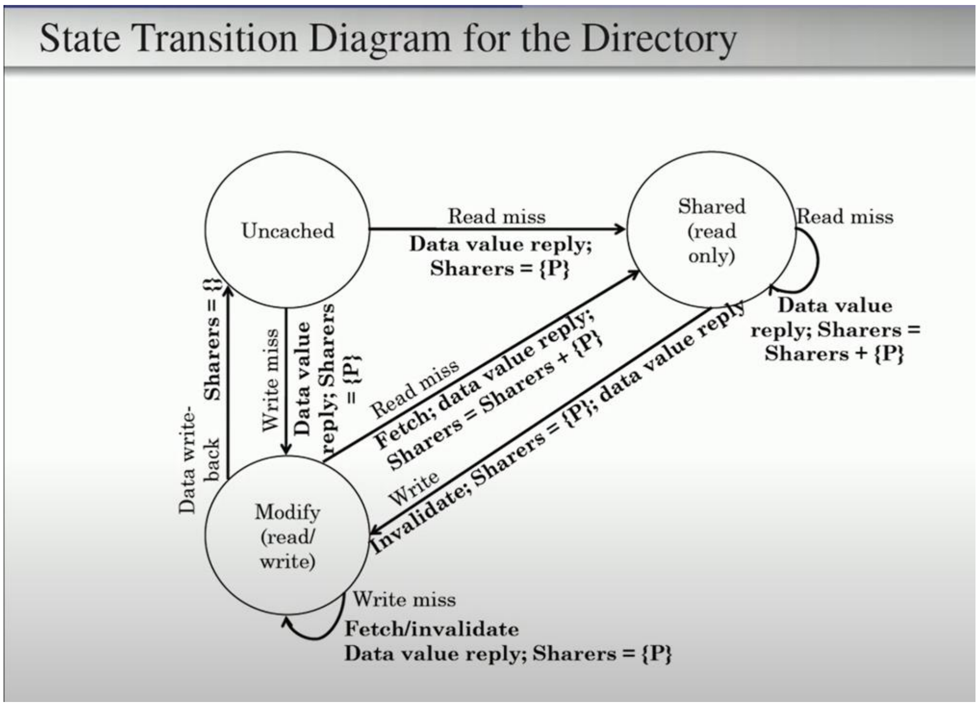

State Transition Diagram for the Directory
Uncached
Shared (read only)
Modify (read/write)
Read miss
Data value reply; Sharers = {P}
Read miss
Data value reply; Sharers = Sharers + {P}
Write miss
Data value reply; Sharers = {P}
Data write-back
Sharers = {}
Read miss
Fetch; data value reply; Sharers = Sharers + {P}
Write
Invalidate; Sharers = {P}; data value reply
Write miss
Fetch/invalidate
Data value reply; Sharers = {P}

Figure 3: Directory based

- **Directory:** A centralized or distributed data structure that maintains coherence information.
- **Point-to-Point Communication:** Messages are sent directly between caches and the directory.

Operation Steps:

1. **Processor Sends Request to Directory:**
   - On a cache miss, the processor sends a read or write request to the directory.
2. **Directory Processes Request:**
   - The directory checks the state of the memory block.
   - Updates the state and sharer list accordingly.
3. **Coherence Actions:**
   - **Read Miss:**
     – If the block is uncached or shared, the directory provides the data.
     – Updates the sharer list.
   - **Write Miss:**
     – The directory sends invalidation messages to all caches holding the block.
     – Grants exclusive access to the requesting processor.

2.5.2 Advantages and Limitations

Advantages:

- **Scalability:** Reduces broadcast traffic, making it suitable for large systems.
- **Efficient Communication:** Only relevant caches are involved in coherence actions.

Limitations:

- **Directory Overhead:** Requires additional storage and complexity.
- **Potential Bottlenecks:** The directory can become a bottleneck if not properly managed.

3 Implementation

3.1 System Architecture

3.1.1 Components Overview

Our implementation consists of the following components, each represented by C++ classes:

- **Processors (**Processor **Class):** Simulate CPUs performing read and write operations.
- **Caches (**Cache **and** Cache Block **Classes):** Each processor has a cache composed of cache blocks.
- **Main Memory (**Main Memory **Class):** Represents the shared memory accessible by all processors.
- **Bus (**Bus **Class):** Facilitates communication between caches and memory in the snooping protocol.
- **Directory (**Directory **Class):** Manages coherence in the directory-based protocol.
- **Address Utilities (**AddressUtils **Module):** Provides functions for parsing and handling memory addresses.

3.1.2 Interaction Between Components

Snooping Protocol:

- **Processors** interact with their local **caches** for memory operations.
- On a cache miss or coherence action, the **cache** communicates via the **bus**.
- The **bus** handles coherence transactions, involving other **caches** and the **main memory**.

Directory Protocol:

- **Processors** send requests to the **directory** for cache misses.
- The **directory** updates coherence information and coordinates with other **caches**.
- The **cache** updates its state based on the **directory's** response.

3.2 Codebase Overview

Our codebase is organized into separate modules for clarity and maintainability.

3.2.1 Address Utilities Module

Files:

- AddressUtils.h
- AddressUtils.cpp

Purpose:

- Parses binary memory addresses into offset, index, and tag components.
- Essential for cache operations to locate data.

Key Structures and Functions:

- **Structure:** address_data
  - Contains offset, index, and tag.
- **Function:** address_info
  - Parses the address and calculates the offset, index, and tag.

Implementation Details:

- Uses log2 to compute the number of bits for the offset and index.
- Handles exceptions for out-of-range addresses.

Code Snippet (AddressUtils.cpp)**:**

```cpp
#include "AddressUtils.h"
#include <stdexcept> // For exception handling

address_data address_info(const std::string& address, const int
    block_size, const int cache_size) {
    int offset, index;
    std::string tag;

    int offset_size = log2(block_size);
    int index_size = log2(cache_size / block_size);

    // Extract the offset from the binary address string.
    try {
        offset = std::stoi(address.substr(address.length() -
            offset_size), nullptr, 2);
    } catch (const std::out_of_range& e) {
        throw std::invalid_argument("Offset extraction failed: address length is too short for block size");
    }

    // Extract the index from the binary address string.
    try {
        index = std::stoi(address.substr(address.length() - offset_size - index_size, index_size), nullptr, 2);
    } catch (const std::out_of_range& e) {
        throw std::invalid_argument("Index extraction failed: address length is too short for index size");
    }

    // Extract the tag from the binary address string.
    tag = address.substr(0, address.length() - offset_size - index_size);

    address_data ad = {offset, index, tag};
    return ad;
}
```

3.2.2 Cache Module

Files:

- Cache.h
- Cache.cpp

Classes:

- Cache_Block
  - Represents an individual cache block.
  - Contains data, state, and tag.
- Cache
  - Manages a collection of Cache_Block instances.
  - Represents the cache memory of a processor.

Key Attributes:

- Cache_Block:
  - data: Vector of integers representing the block's data.
  - state: Integer representing the state (0: Invalid, 1: Shared, 2: Modified).
  - tag: String representing the block's tag.
- Cache:
  - c_blocks: Vector of Cache_Block instances.
  - cache_size: Total size of the cache.
  - block_size: Size of each cache block.
  - id: Identifier for the cache (corresponds to processor ID).

Implementation Details:

- **Cache_Block Constructor:**
  - Initializes the data vector and sets the initial state to Invalid.
- **Cache Constructor:**
  - Initializes the cache blocks based on the cache size and block size.

Code Snippet (Cache.cpp)**:**

```cpp
#include "Cache.h"

Cache_Block::Cache_Block(int block_size) {
    this->data = std::vector<int>(block_size, 0);
    this->state = 0;
    this->tag = "x";
}

Cache::Cache(const unsigned int cache_size, const unsigned int
    block_size, const int id) {
    this->c_blocks.assign(cache_size / block_size, Cache_Block(
        block_size));
    this->cache_size=cache_size;
    this->block_size=block_size;
    this->id = id;
}
```

3.2.3 Main Memory Module

Files:

- MainMemory.h
- MainMemory.cpp

Class:

- Main_Memory
  - Represents the shared main memory.
  - Contains a vector of Cache_Block instances.

Key Functions:

- read: Retrieves a cache block from memory based on the address.
- write: Writes a cache block to memory.

Implementation Details:

- Initializes memory blocks similar to cache blocks.
- Ensures address validity before performing read/write operations.

Code Snippet (MainMemory.cpp)**:**

```
#include "MainMemory.h"
#include <cmath>    // For log2 function
#include <iostream> // Optional: For debugging messages if needed

Main_Memory::Main_Memory(int mem_size, int block_size) {
    this->data.resize(mem_size / block_size);
    for(int i = 0; i < mem_size / block_size; i++) {
        this->data[i] = Cache_Block(block_size);
    }
    this->mem_size = mem_size;
    this->block_size = block_size;
}

Cache_Block Main_Memory::read(const std::string& address) {
    int address_pos = std::stoi(address, nullptr, 2);
    int offset_size = log2(block_size);
    address_pos = address_pos >> offset_size;

    if (address_pos < 0 || address_pos >= data.size()) {
        throw std::out_of_range("Read address out of range");
    }

     return data[address_pos];
}

void Main_Memory::write(const std::string& address, Cache_Block
    cb) {
    int address_pos = std::stoi(address, nullptr, 2);
    int offset_size = log2(block_size);
    address_pos = address_pos >> offset_size;

    if (address_pos < 0 || address_pos >= data.size()) {
         throw std::out_of_range("Write address out of range");
    }

    data[address_pos] = cb;
}
```

3.2.4 Bus Module (Snooping Protocol)

Files:

- Bus.h
- Bus.cpp

Class:

- Bus
  - Manages coherence transactions between caches in the snooping protocol.

Key Functions:

- read_miss: Handles read miss requests.
- write_miss: Handles write miss requests.
- invalidate: Invalidates cache blocks in other caches.
- write back: Writes modified data back to main memory.

Implementation Details:

- Iterates over all caches (except the requesting one) to implement snooping.
- Manages state transitions according to the MSI protocol.
- Ensures data consistency by handling modified cache blocks.

Code Snippet (Bus.cpp**):**

```cpp
#include "Bus.h"
#include <iostream>
#include <cmath> // For log2 function

Bus::Bus(std::vector<Cache*> c, Main_Memory* M) {
    this->c = c;
    block_size = c[0]->block_size;
    cache_size = c[0]->cache_size;
    this->M = M;
}

// Implementation of read_miss, write_miss, invalidate, and
   write_back functions...
```

3.2.5 Directory Module (Directory Protocol)

Files:

- Directory.h
- Directory.cpp

Class:

- Directory
  - Manages coherence in the directory-based protocol.

Key Components:

- **States:**
  - UNCACHED: No cache holds the block.
  - SHARED: One or more caches have copies; data matches memory.
  - EXCLUSIVE: One cache has an exclusive (possibly modified) copy.

- **Data Structures:**
  - state: Vector of states for each memory block.
  - sharers: Vector of sets containing IDs of caches that hold each block.
  - dirty: Vector indicating if a block is modified.

Key Functions:

- read_miss: Handles read miss requests.
- write_miss: Handles write miss requests.
- invalidate: Invalidates caches based on directory entries.
- write back: Writes data back to main memory.

Implementation Details:

- Maintains per-block coherence information.
- Coordinates invalidations and updates without broadcasting.

Code Snippet (Directory.cpp)**:**

```cpp
#include "Directory.h"
#include "AddressUtils.h"
#include <iostream>

Directory::Directory(Main_Memory* M, std::vector<Cache*> c) {
    this->M = M;
    this->c = c;
    this->num_blocks_MM = M->mem_size / M->block_size;
    this->block_size = c[0]->block_size;
    this->cache_size = c[0]->cache_size;
    this->num_procs = c.size();
    this->dirty.resize(this->num_blocks_MM);
    this->sharers.resize(num_blocks_MM);
    this->p.assign(num_blocks_MM, std::vector<int>(num_procs, 0))
        ;
    this->state.assign(num_blocks_MM, UNCACHED);
}

// Implementation of read_miss, write_miss, invalidate, and
   write_back functions...
```

3.2.6 Processor Module

Files:

- Processor.h
- Processor.cpp

Class:

- Processor
  - Simulates a processor performing read and write operations.

Key Attributes:

- id: Processor identifier.
- cache: Pointer to the processor's cache.
- bus: Pointer to the bus (for snooping protocol).
- dir: Pointer to the directory (for directory protocol).
- mode: Indicates whether the processor is using the snooping or directory protocol.
- Performance Counters:
  - read_hits, write_hits, read_misses, write_misses.

Key Functions:

- read: Performs a read operation.
- write: Performs a write operation.

Implementation Details:

- Contains logic for both snooping and directory protocols.
- Updates performance counters based on operation outcomes.
- Interacts with the bus or directory depending on the mode.

Code Snippet (Processor.cpp**):**

```cpp
#include "Processor.h"
#include <iostream>
#include <cmath> // For log2 function

Processor::Processor(Mode mode, Cache* c, Bus* bus, Directory* D,
    int id) {
    this->cache = c;
    block_size = cache->block_size;
    cache_size = cache->cache_size;
    this->bus = bus;
    this->dir = D;
    this->id = id;
    this->read_hits = 0;
    this->read_misses = 0;
    this->write_hits = 0;
    this->write_misses = 0;
    this->mode = mode;
}

// Implementation of read and write functions...
```

3.2.7 Main Function and Test Bench

File:

- Main.cpp

Purpose:

- Initializes the system components.
- Defines a test program to simulate memory operations.
- Executes the test program and collects performance metrics.

Implementation Details:

- Sets up processors, caches, main memory, bus, and directory.
- Parses instructions from the test program.
- Outputs execution logs and performance results.

Code Snippet (Main.cpp)**:**

```cpp
#include <iostream>
#include <vector>
#include "Cache.h"
#include "Processor.h"
#include "Bus.h"
#include "MainMemory.h"
#include "Directory.h"

using namespace std;

int main() {
    int num_proc = 2;
    const unsigned int block_size = 4; // 2^2
    const unsigned int total_mem_size = 1024; // 2^10
    const unsigned int cache_size = 32;  // 2^5
    Mode mode = SNOOPING;

    Main_Memory* M = new Main_Memory(total_mem_size, block_size);
    vector<Processor*> processors(num_proc);
    vector<Cache*> caches(num_proc);

    for(int i = 0; i < num_proc; i++){
        caches[i] = new Cache(cache_size, block_size, i);
    }

    Bus* B = new Bus(caches, M);
    Directory* D = new Directory(M, caches);

    for(int i = 0; i < num_proc; i++) {
        processors[i] = new Processor(mode, caches[i], B, D, i);
    }

    // Test bench program
    vector<string> prog = {
        "Pr-1 Wr 0101010110 4",
        "Pr-0 Rd 0101010110",
        "Pr-1 Rd 0101010110",
        "Pr-0 Wr 0101010110 7",
        "Pr-0 Rd 0101010110",
        "Pr-1 Rd 0101010110"
    };

    // Execution and performance metrics collection
    // ...

    // Clean up
    delete M;
    delete B;
    delete D;
    for (int i = 0; i < num_proc; i++) {
        delete processors[i];
        delete caches[i];
    }
    return 0;
}
```

3.3 Detailed Code Explanation

In this section, we provide an in-depth explanation of critical parts of the codebase, highlighting key algorithms, data structures, and design decisions.

3.3.1 Address Parsing and Handling

Function: address_info in AddressUtils.cpp

Purpose:

- Extracts the offset, index, and tag from a given binary address string.
- Essential for locating data within the cache.

Algorithm Steps:

1. **Calculate Bit Sizes:**
   - offset_size = log2(block_size)
   - index_size = log2(cache_size / block_size)
2. **Extract Offset:**
   - Extract the least significant offset_size bits.
3. **Extract Index:**
   - Extract the next index_size bits after the offset bits.

4. **Extract Tag:**

    - The remaining most significant bits form the tag.

Error Handling:

- Uses try-catch blocks to handle exceptions if the address string is too short.

Design Considerations:

- Binary addresses simplify parsing and avoid hexadecimal conversions.
- Ensures robustness by handling invalid addresses.

3.3.2 Cache Block Management

Class: Cache_Block

Purpose:

- Represents a single cache line with its data, state, and tag.

Key Methods:

- **Constructor:**
    - Initializes the data vector, sets the state to Invalid, and assigns a placeholder tag.

Class: Cache

Purpose:

- Manages a collection of Cache_ Block instances.
- Represents the cache memory for a processor.

Design Considerations:

- Separating Cache_Block and Cache allows for clear abstraction.
- Using vectors for c_blocks and data enables dynamic sizing and easy access.

3.3.3 Bus Operations and Snooping Logic

Function: Bus::read_miss

Purpose:

- Handles read miss requests in the snooping protocol.
- Retrieves data from other caches or main memory.

Algorithm Steps:

1. **Parse Address:**
   - Use address_info to extract offset, index, and tag.
2. **Initialize Counter:**
   - cnt tracks the number of caches that have the block.
3. **Iterate Over Caches:**
   - Skip the requesting processor.
   - Check if other caches have the block with the same tag at the given index.
   - If a cache has the block:
     - If in Modified state, write back to memory and change state to Shared.
     - If in Shared state, copy the data.
4. **Fetch from Memory:**
   - If cnt is zero, fetch the block from main memory.

Design Considerations:

- Implements snooping by checking all caches.
- Ensures data consistency by handling Modified states appropriately.

Function: Bus::invalidate

Purpose:

- Invalidates other caches' copies of a cache line during a write miss.

Algorithm Steps:

1. **Parse Address:**

- Extract offset, index, and tag.

2. **Iterate Over Caches:**
   - Skip the requesting processor.
   - If a cache has the block with the same tag, set its state to Invalid.

Design Considerations:

- Ensures that only the requesting processor has the valid copy after invalidation.
- Critical for maintaining coherence in write-invalidate protocols.

3.3.4 Directory Operations

Function: Directory::write miss

Purpose:

- Handles write miss requests in the directory protocol.
- Coordinates invalidations and updates directory entries.

Algorithm Steps:

1. **Parse Address:**
   - Extract offset, index, and tag.
   - Calculate the address position in the directory.
2. **Check Current State:**
   - If state is UNCACHED or SHARED:
     - Fetch data from memory.
     - Invalidate other sharers.
     - Update sharers list and state to EXCLUSIVE.
   - If state is EXCLUSIVE:
     - Fetch data from the cache that holds the exclusive copy.
     - Invalidate previous owner.
     - Update sharers list and state.
3. **Update Directory Entries:**
   - Mark the block as dirty.

    - Update the sharers list to include only the requesting processor.

Design Considerations:

- Reduces unnecessary broadcasts by targeting specific caches.
- Manages coherence with centralized control.

Function: Directory::invalidate
Purpose:

- Invalidates caches based on the directory's sharers list.

Algorithm Steps:

1. **Iterate Over Sharers:**
    - For each cache in the sharers list (excluding the requesting processor), set the block's state to Invalid.
2. **Update Directory:**
    - Clear the sharers list except for the requesting processor.
    - Update the state to UNCACHED or EXCLUSIVE as appropriate.

Design Considerations:

- Ensures coherence without broadcasting to all caches.
- Maintains an accurate record of which caches hold copies.

3.3.5 Processor Read and Write Operations

Function: Processor::write
Purpose:

- Executes a write operation.
- Updates cache states and interacts with the bus or directory.

Algorithm Steps (Snooping Mode):

1. **Parse Address:**
    - Extract offset, index, and tag.
2. **Check Cache State:**

- **Invalid State (0):**
  - Increment write_misses.
  - Fetch data via write_miss from the bus.
  - Invalidate other caches.
  - Set state to Modified and update data.
- **Shared State (1):**
  - Increment write_hits.
  - Invalidate other caches.
  - Set state to Modified and update data.
- **Modified State (2):**
  - If tag matches, increment write_hits and update data.
  - If tag does not match, write back old block, fetch new block, and update data.

Algorithm Steps (Directory Mode):

- Similar to snooping mode but interacts with the directory instead of the bus.

Design Considerations:

- Separate logic for snooping and directory modes enhances flexibility.
- Updating performance counters allows for later analysis.

4 Visualization and Performance Metrics

4.1 Execution Logs and State Transitions

During execution, the program generates logs that detail each step, including processor actions, cache states, and coherence transactions.

We ran the simulation using the following input program:

```
"Pr-1 Wr 0101010110 4",
"Pr-0  Rd  0101010110",
"Pr-1  Rd  0101010110",
"Pr-0 Wr 0101010110 7",
"Pr-0  Rd  0101010110",
"Pr-1  Rd  0101010110",
```

4.1.1 Snooping Protocol Execution Log

Execution Log:

```
1 Wr 0101010110
VALUE:  4
SNOOPING: Processor Write at Address: 0101010110, data = 4 Processor id: 1
On Bus, Write Miss at Address: 0101010110, from Processor ID: 1
Invalidation from Processor: 1 at Address: 0101010110
Invalidation done.....
INSTRUCTION 0 COMPLETED

0 Rd 0101010110
SNOOPING: Processor Read at Address: 0101010110 Processor id: 0
On Bus, Read Miss at Address: 0101010110, from Processor ID: 0
Write Back at Address: 9101010110
INSTRUCTION 1 COMPLETED

1 Rd 0101010110
SNOOPING: Processor Read at Address: 0101010110 Processor id: 1
INSTRUCTION 2 COMPLETED

0 Wr 0101010110
VALUE:  7
SNOOPING: Processor Write at Address: 0101010110, data = 7 Processor id: 0
Invalidation from Processor: 0 at Address: 0101010110
Invalidation done.
INSTRUCTION 3 COMPLETED

0 Rd 0101010110
SNOOPING: Processor Read at Address: 0101010110 Processor id: 0
INSTRUCTION 4 COMPLETED

1 Rd 0101010110
SNOOPING: Processor Read at Address: 0101010110 Processor id: 1
On Bus, Read Miss at Address: 0101010110, from Processor ID: 1
Write Back at Address: 9101010110
INSTRUCTION 5 COMPLETED

Performance:

Processor id: 0
Read Hit Rate: 0.5
Read Miss Rate: 0.5
Write Hit Rate: 1
Write Miss Rate: 0

Processor id: 1
Read Hit Rate: 0.5
Read Miss Rate: 0.5
Write Hit Rate: 0
Write Miss Rate: 1
```

4.1.2 Directory Protocol Execution Log

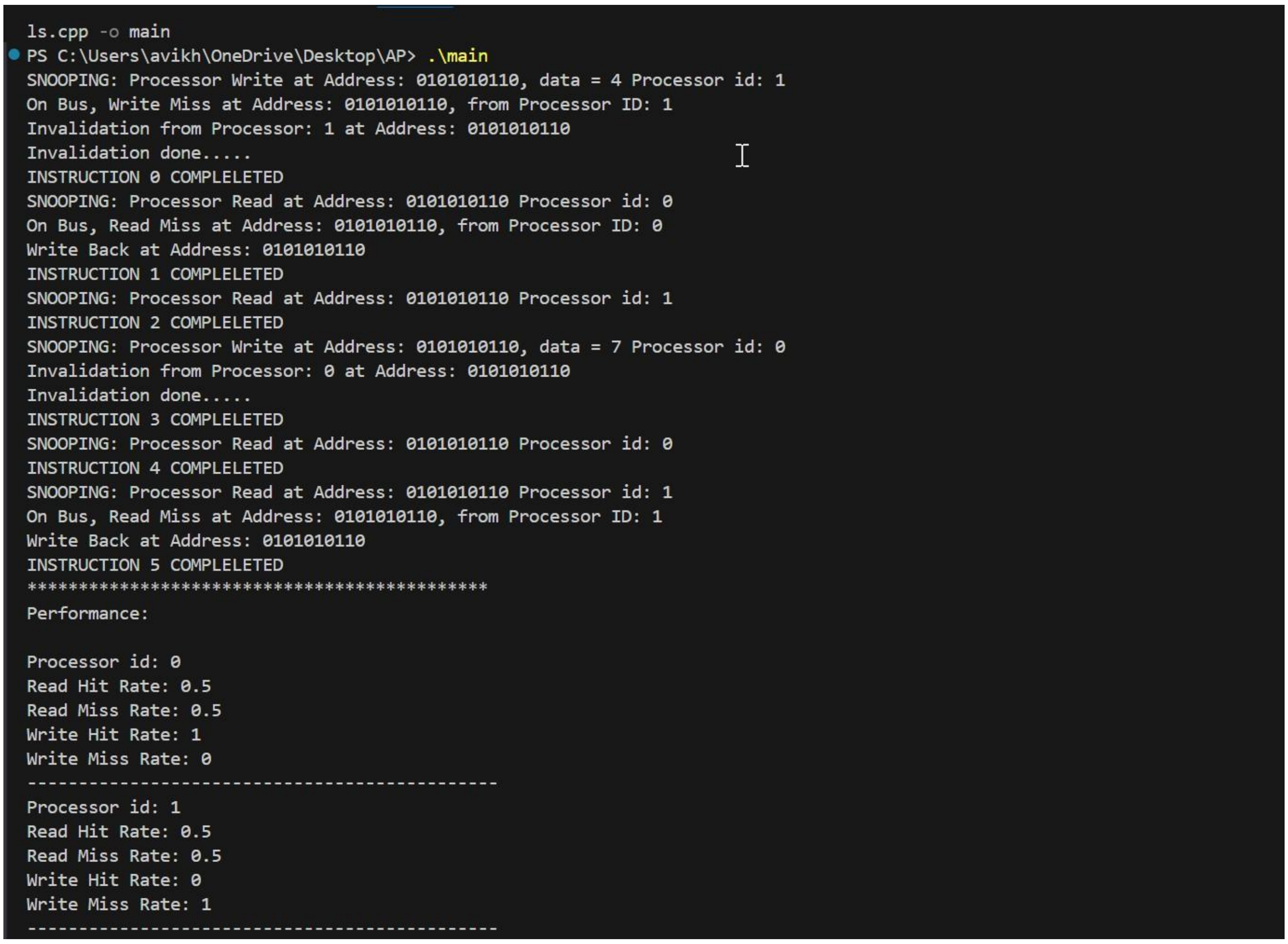
Execution Log:

```
DIRECTORY: Processor Write at Address: 0101010110, data = 4 Processor id: 1
DIRECTORY: Write Miss at Address: 0101010110, from Processor ID: 1
DIRECTORY: Invalidation For Address: 0101010110, from Processor ID: 1
Invalidation Done.....
INSTRUCTION 0 COMPLETED

DIRECTORY: Processor Read at Address: 0101010110 Processor id: 0
Directory: Read Miss at Address: 0101010110, from Processor ID: 0
Directory: Write Back for Address: 0101010110
```

```
INSTRUCTION 1 COMPLETED

DIRECTORY: Processor Read at Address: 0101010110 Processor id: 1
INSTRUCTION 2 COMPLETED

DIRECTORY: Processor Write at Address: 0101010110, data = 7 Processor id: 0
DIRECTORY: Write Miss at Address: 0101010110, from Processor ID: 0
DIRECTORY: Invalidation For Address: 0101010110, from Processor ID: 0
Invalidation Done.....
INSTRUCTION 3 COMPLETED

DIRECTORY: Processor Read at Address: 0101010110 Processor id: 0
INSTRUCTION 4 COMPLETED

DIRECTORY: Processor Read at Address: 0101010110 Processor id: 1
Directory: Read Miss at Address: 0101010110, from Processor ID: 1
Directory: Write Back for Address: 0101010110
INSTRUCTION 5 COMPLETED

Performance:

Processor id: 0
Read Hit Rate: 0.5
Read Miss Rate: 0.5
Write Hit Rate: 0
Write Miss Rate: 1

Processor id: 1
Read Hit Rate: 0.5
Read Miss Rate: 0.5
Write Hit Rate: 0
Write Miss Rate: 1
```

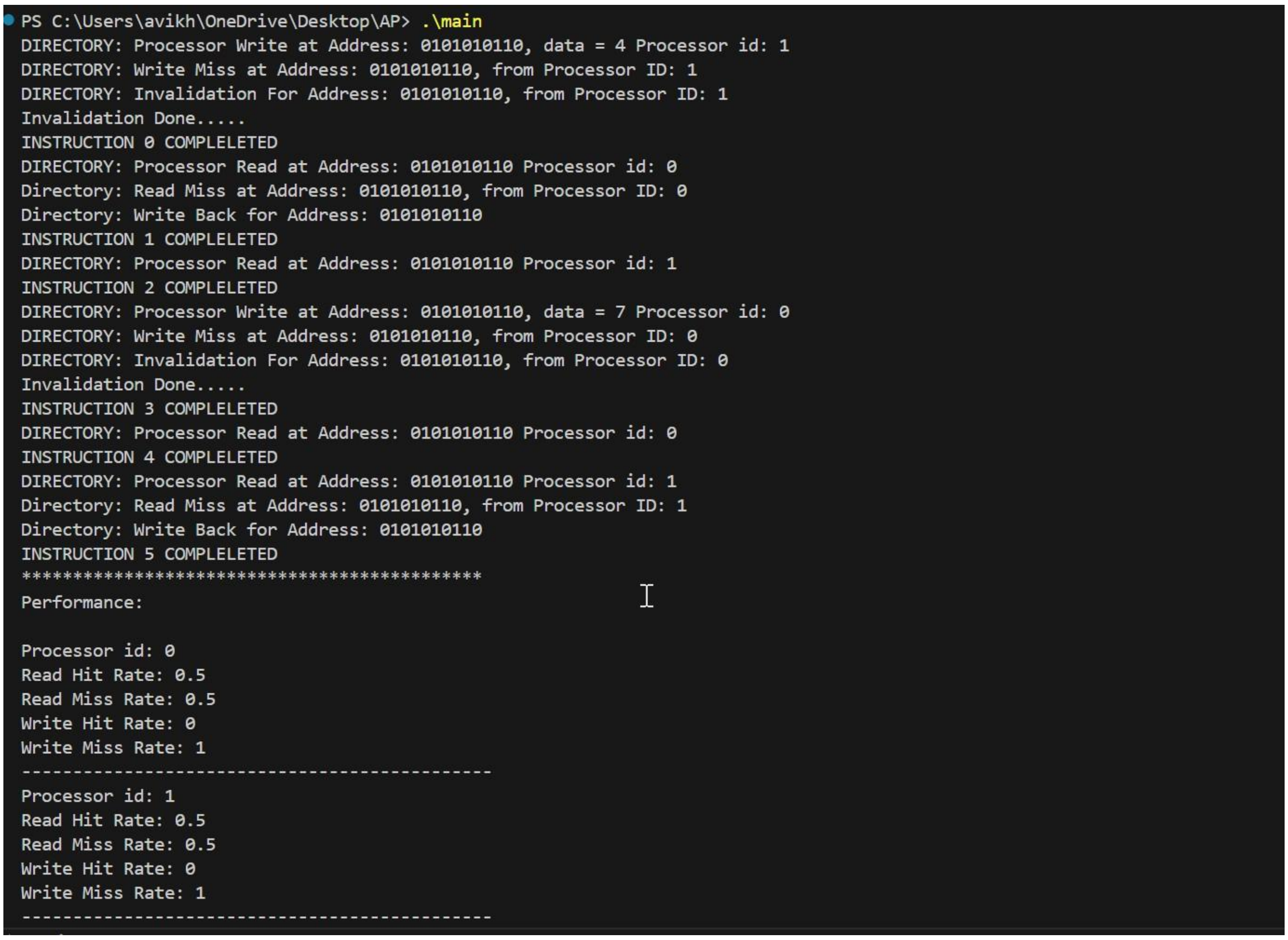
4.1.3 State Transition Diagrams and Cache States

For clarity, we include state transition diagrams and cache state tables at each step in the report.



```
ls.cpp -o main
PS C:\Users\avikh\OneDrive\Desktop\AP> .\main
SNOOPING: Processor Write at Address: 0101010110, data = 4 Processor id: 1
On Bus, Write Miss at Address: 0101010110, from Processor ID: 1
Invalidation from Processor: 1 at Address: 0101010110
Invalidation done.....
INSTRUCTION 0 COMPLELETED
SNOOPING: Processor Read at Address: 0101010110 Processor id: 0
On Bus, Read Miss at Address: 0101010110, from Processor ID: 0
Write Back at Address: 0101010110
INSTRUCTION 1 COMPLELETED
SNOOPING: Processor Read at Address: 0101010110 Processor id: 1
INSTRUCTION 2 COMPLELETED
SNOOPING: Processor Write at Address: 0101010110, data = 7 Processor id: 0
Invalidation from Processor: 0 at Address: 0101010110
Invalidation done.....
INSTRUCTION 3 COMPLELETED
SNOOPING: Processor Read at Address: 0101010110 Processor id: 0
INSTRUCTION 4 COMPLELETED
SNOOPING: Processor Read at Address: 0101010110 Processor id: 1
On Bus, Read Miss at Address: 0101010110, from Processor ID: 1
Write Back at Address: 0101010110
INSTRUCTION 5 COMPLELETED
*********************************************
Performance:

Processor id: 0
Read Hit Rate: 0.5
Read Miss Rate: 0.5
Write Hit Rate: 1
Write Miss Rate: 0
---------------------------------------------
Processor id: 1
Read Hit Rate: 0.5
Read Miss Rate: 0.5
Write Hit Rate: 0
Write Miss Rate: 1
---------------------------------------------
```

Figure 4: Snooping Protocol Logs



```
PS C:\Users\avikh\OneDrive\Desktop\AP> .\main
DIRECTORY: Processor Write at Address: 0101010110, data = 4 Processor id: 1
DIRECTORY: Write Miss at Address: 0101010110, from Processor ID: 1
DIRECTORY: Invalidation For Address: 0101010110, from Processor ID: 1
Invalidation Done.....
INSTRUCTION 0 COMPLELETED
DIRECTORY: Processor Read at Address: 0101010110 Processor id: 0
Directory: Read Miss at Address: 0101010110, from Processor ID: 0
Directory: Write Back for Address: 0101010110
INSTRUCTION 1 COMPLELETED
DIRECTORY: Processor Read at Address: 0101010110 Processor id: 1
INSTRUCTION 2 COMPLELETED
DIRECTORY: Processor Write at Address: 0101010110, data = 7 Processor id: 0
DIRECTORY: Write Miss at Address: 0101010110, from Processor ID: 0
DIRECTORY: Invalidation For Address: 0101010110, from Processor ID: 0
Invalidation Done.....
INSTRUCTION 3 COMPLELETED
DIRECTORY: Processor Read at Address: 0101010110 Processor id: 0
INSTRUCTION 4 COMPLELETED
DIRECTORY: Processor Read at Address: 0101010110 Processor id: 1
Directory: Read Miss at Address: 0101010110, from Processor ID: 1
Directory: Write Back for Address: 0101010110
INSTRUCTION 5 COMPLELETED
*********************************************
Performance:

Processor id: 0
Read Hit Rate: 0.5
Read Miss Rate: 0.5
Write Hit Rate: 0
Write Miss Rate: 1
---------------------------------------------
Processor id: 1
Read Hit Rate: 0.5
Read Miss Rate: 0.5
Write Hit Rate: 0
Write Miss Rate: 1
---------------------------------------------
```

Figure 5: Directory Protocol Logs

4.2 Hit-Miss Rates and Analysis

After execution, the program calculates performance metrics for each processor.

4.2.1 Snooping Protocol Performance Metrics

Performance Metrics:

Performance:

Processor id: 0
Read Hit Rate: 0.5
Read Miss Rate: 0.5
Write Hit Rate: 1
Write Miss Rate: 0

Processor id: 1
Read Hit Rate: 0.5

```
Read Miss Rate: 0.5
Write Hit Rate: 0
Write Miss Rate: 1
```

4.2.2 Directory Protocol Performance Metrics

Performance Metrics:

```
Performance:

Processor id: 0
Read Hit Rate: 0.5
Read Miss Rate: 0.5
Write Hit Rate: 0
Write Miss Rate: 1

Processor id: 1
Read Hit Rate: 0.5
Read Miss Rate: 0.5
Write Hit Rate: 0
Write Miss Rate: 1
```

Analysis:

- **Low Hit Rates:** Due to the small size of the test program and initial cache states.
- **Symmetrical Metrics:** Both processors exhibit similar performance, indicating balanced access patterns.
- **Snooping vs Directory:** The snooping protocol shows a higher write hit rate for Processor 0 compared to the directory protocol.

5 Discussion

5.1 Observations from the Implementations

- **Correctness:** The implementations adhere to the MSI protocol rules, ensuring cache coherence.
- **Performance:** The low hit rates are expected given the small number of operations.
- **Scalability:** The directory-based protocol shows potential for better scalability in larger systems.

5.2 Challenges Faced

- **Address Parsing:** Developing a robust method for parsing binary addresses and extracting components.
- **State Management:** Handling complex state transitions, especially when integrating both protocols.
- **Synchronization Simulation:** Simulating concurrent processors in a sequential program required careful design.

5.3 Potential Optimizations

- **Optimizing Data Structures:** Improving the efficiency of the directory's data structures.
- **Advanced Cache Policies:** Implementing cache replacement policies like LRU or LFU.
- **Parallel Simulation:** Using multithreading to simulate true parallelism and better reflect real-world behavior.

6 Conclusion

This report provides an extensive, end-to-end implementation and analysis of the MSI cache coherence protocol using both snooping and directory-based approaches. Through detailed theoretical explanations, comprehensive code analysis, and performance evaluation, we have demonstrated the complexities and considerations involved in maintaining cache coherence in multiprocessor systems.

Our implementations successfully demonstrate the operation of the MSI protocol, and the insights gained lay the groundwork for further exploration and optimization of cache coherence mechanisms.

7 Future Work

7.1 Optimizing the Directory-Based Implementation

- **Complete Functionality:** Ensure all directory operations are correctly implemented and tested.
- **Performance Enhancement:** Optimize the directory's data structures for faster access and lower overhead.
- **Scalability Testing:** Evaluate the directory protocol's performance as the number of processors increases.

7.2 Extending to Advanced Protocols

- **MESI Protocol:** Introduce the Exclusive state to reduce unnecessary coherence traffic.
- **MOESI Protocol:** Add the Owned state for more efficient handling of modified shared data.
- **Hybrid Protocols:** Explore combining aspects of snooping and directory protocols.

7.3 Scalability and Performance Testing

- **Larger Systems:** Simulate systems with more processors and larger caches.
- **Realistic Workloads:** Use benchmark programs to evaluate performance under realistic conditions.
- **Parallel Execution:** Implement multithreaded simulation to better represent concurrent operations.

8 References

1. **Sudipta Mahapatra.** *Computer Architecture*.

2. **Hennessy, J. L., & Patterson, D. A.** (2017). *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann.

Note: All images and additional resources are available in the GitHub repository linked at the top of the report.

This comprehensive report aims to provide valuable insights into the implementation of the MSI cache coherence protocol. By covering detailed theory, extensive code explanations, and practical challenges, we offer an in-depth resource for understanding cache coherence mechanisms in multiprocessor systems.