Multicore Cache Coherence

Gabriel Chacon Alfaro
Escuela de Ingeniería en Computadores
Tecnológico de Costa Rica
Cartago, Costa Rica
gchacon8@estudiantec.cr

Daniel Castro Elizondo
Escuela de Ingeniería en Computadores
Tecnológico de Costa Rica
Cartago, Costa Rica
dcastroeli@estudiantec.cr

Jose Eduardo Cruz Vargas
Escuela de Ingeniería en Computadores
Tecnológico de Costa Rica
Cartago, Costa Rica
jecruz@estudiantec.cr

Marco Rivera Serrano
Escuela de Ingeniería en Computadores
Tecnológico de Costa Rica
Cartago, Costa Rica
marco99@estudiantec.cr

***Abstract*—This paper presents a simulation of a multicore system with cache coherence implemented through the MESI protocol. Each core is represented as a Processing Element (PE) with an individual L1 cache and a shared RAM. A Bus Interconnect is used to enforce cache coherence across cores, with each PE implemented as a separate thread using `pthreads`. The project was designed in C++ due to its memory management capabilities and low-level functionalities, providing fine control over threading and memory access. Step-by-step execution (stepping) is enabled to observe and debug the coherence process in real-time.**

I. Introduction

Multicore architectures are increasingly popular in modern computing, with shared memory and cache coherence being critical to performance and consistency in data access [1]. This project aims to simulate a multicore system where each core has a private L1 cache and accesses a shared RAM. Cache coherence is maintained using the MESI (Modified, Exclusive, Shared, Invalid) protocol, commonly used in multiprocessor systems [2].

The implementation uses C++ to leverage its low-level control over memory and threading, with each PE acting as a separate thread. This design allows for parallel execution and realistic simulation of cache coherence. Additionally, stepping functionality provides step-by-step execution, making the coherence process visible and facilitating debugging.

II. The Role of C++ in the Project Design

C++ was selected for this simulation due to its extensive capabilities for memory manipulation and thread management, essential for developing a realistic multicore system. The project benefits from C++'s support for low-level memory operations and object-oriented design, which facilitates encapsulating each element (e.g., `PE`, `Cache`, `Bus Interconnect`) into distinct classes [3].

The use of `pthreads` in C++ allows each PE to function as an independent thread, simulating parallel execution effectively [6]. Threaded execution is crucial in testing coherence protocols like MESI, as it requires simultaneous interactions among caches and shared memory to maintain consistency [5]. By leveraging C++'s control over these mechanisms, this project models cache coherence with precision, offering a practical framework for understanding the complexities involved.

III. Class Design

A. Processing Element (PE)

The `PE` class simulates a core in the multicore system, handling its unique cache, instruction set, and coherence states. Each PE:

- Contains an instance of `Cache` to store data locally, minimizing access to shared RAM.
- Uses a `ROM` to store a predefined instruction set, loaded from a text file, that specifies each PE's execution path.
- Interacts with `MESI` to maintain cache coherence across the multicore environment.

Each PE runs as an independent thread via `pthreads`, allowing for concurrent execution and inter-core interaction. This concurrent approach provides insights into cache performance and coherence management across multiple cores, similar to real-world multicore systems [4].

B. Cache

The `Cache` class simulates an L1 cache for each PE, storing data values and tracking access state using the MESI protocol. The cache contains:

- Data and address fields for local storage and quick data access.
- A miss counter to evaluate cache efficiency and hit/miss rates during execution.
- A state variable managed by `MESI` to track cache line states (Modified, Exclusive, Shared, Invalid).

By reducing RAM access, caches minimize latency and improve overall system efficiency. MESI protocol

state transitions, triggered by the Bus Interconnect, help maintain data consistency across caches, as explained by Hill and Agarwal's foundational work on cache coherence protocols [5].

C. ROM

The `ROM` class provides each PE with a unique set of instructions, loaded from external text files. This modular design enables flexibility by allowing different instruction sets per PE without recompiling the code. ROM enables each PE to act independently, supporting diverse workflows and enhancing the overall system's versatility [2].

D. MESI

The `MESI` class implements the MESI coherence protocol, which defines four states—Modified, Exclusive, Shared, and Invalid—for each cache line. Each PE's cache can be in one of these states, which are updated based on specific events, such as data reads or writes. The state transitions are managed by `Bus Interconnect`, which signals changes to ensure data consistency [1]. This protocol is effective for preventing stale data issues while reducing unnecessary data duplication across caches.

E. RAM

The `RAM` class represents shared memory accessible to all PEs. RAM serves as a centralized data repository, from which each PE can load data into its cache. Consistency is maintained by enforcing coherence through the MESI protocol, ensuring that any data modifications by one PE are visible to all others, preventing inconsistencies and maintaining synchronization across threads [5].

F. Bus Interconnect

The `Bus Interconnect` class is responsible for coordinating cache coherence by managing requests and responses among PEs. The bus utilizes `MESI` to manage cache state transitions based on the actions of each PE, ensuring that data remains consistent. This centralized bus model, while simple, is effective for simulating coherence in a multicore environment, as described in prior works on coherence protocols [4].

IV. Implementation Details

A. Concurrency with pthreads

Each `PE` functions as an independent thread using the `pthreads` library, enabling concurrent execution. This concurrency models real multicore processing, where multiple cores operate in parallel. The `Bus Interconnect` orchestrates these threads by handling requests and enforcing coherence via MESI, aligning with real-world requirements for efficient cache synchronization [6].

B. Instruction Management with Text Files

Each PE loads its instruction set from a text file into its `ROM`, allowing for dynamic updates to the PEs' execution paths. Using external files for instruction loading adds flexibility, as it allows developers to modify each core's behavior without code recompilation, a technique that enhances modularity [3].

C. Stepping

The project includes a stepping feature, allowing for the simulation to execute one instruction at a time across all PEs. Stepping enables close examination of each stage in the MESI protocol, allowing developers to observe coherence in action and debug any issues. This method aligns with the debugging techniques recommended in systems that require high accuracy and strict synchronization [2].

V. Results

Since we're limited to just five instructions, we can use them like this in the different processing elements:

```
INC 0
INC 0
STORE 0 10
LOAD 1 10
```

Fig. 1. PE0 Code

```
INC 0
INC 0
LOAD 0 10
```

Fig. 2. PE1 Code

```
INC 1
INC 1
STORE 1 10
LOAD 0 10
```

Fig. 3. PE2 Code

```
INC 2
INC 2
STORE 2 10
LOAD 3 10
```

Fig. 4. PE3 Code

Thanks to this order of instructions, the MESI protocol will have the next behavior:

VI. Conclusion

This project successfully models a multicore system with cache coherence maintained through the MESI protocol. By employing C++ for low-level memory control and concurrency, the system efficiently simulates parallel execution and coherent data sharing. The use of `pthreads` enables real-time thread management for each core, and the stepping feature offers

	Cache	Address	Old State	New State
1	Cache0	13	Unknown	Unknown
2	Cache1	10	Invalid	Exclusive
3	Cache2	13	Unknown	Unknown
4	Cache3	13	Unknown	Unknown

Fig. 5. First Step

	Cache	Address	Old State	New State
1	Cache0	10	Modified	Invalid
2	Cache1	10	Invalid	Invalid
3	Cache2	10	Modified	Invalid
4	Cache3	10	Invalid	Modified

Fig. 6. Second Step

	Cache	Address	Old State	New State
1	Cache0	10	Shared	Shared
2	Cache1	10	Invalid	Invalid
3	Cache2	10	Invalid	Shared
4	Cache3	10	Shared	Shared

Fig. 7. Third Step

	Core	Cache Misses	Cache Hits	Data Transmitted
1	Core0	1	1	8KB
2	Core1	1	0	8KB
3	Core2	1	1	0KB
4	Core3	1	1	0KB

		Invalidations	Read Requests	Read Responses	Write Requests	Write Responses
1	General Stats	4	1	1	1	1

	Cache	Address	Old State	New State
1	Cache0	10	Invalid	Shared
2	Cache1	10	Shared	Invalid
3	Cache2	10	Shared	Shared
4	Cache3	10	Shared	Shared

Fig. 8. Final Result

a clear view of coherence operations, making this a practical tool for understanding multicore cache coherence.

VII. Code

This is the link to access the github repository: https://github.com/GChacon8/Multi-Core-Cache-Coherence

References

[1] K. Lee, "Cache coherence in multiprocessor systems," *IEEE Computer*, vol. 41, no. 6, pp. 75–82, 2008.

[2] M. Henry, "The MESI protocol: Theory and practice," *Journal of Computer Architecture*, vol. 20, no. 3, pp. 55–63, 2002.

[3] B. Stroustrup, *The C++ Programming Language*. Boston: Addison-Wesley, 2013.

[4] S. Thakkar and M. Sweazey, "The impact of cache coherence protocols on multiprocessing performance," *IEEE Transactions on Computers*, vol. 41, no. 9, pp. 1143–1154, 1992.

[5] M. Hill and A. Agarwal, "Cache performance of multiprocessors," *ACM Transactions on Computer Systems*, vol. 7, no. 4, pp. 370–396, 1989.

[6] B. Kernighan and D. Ritchie, *The C Programming Language*. Englewood Cliffs, NJ: Prentice Hall, 1988.