Cache Coherence using MSI protocol

Abhinav Sharma
Stony Brook University
Stony Brook, NY, USA
abhinsharma@cs.stonybrook.edu

Rohan Bansal
Stony Brook University
Stony Brook, NY, USA
rbansal@cs.stonybrook.edu

Vivek Golani
Stony Brook University
Stony Brook, NY, USA
vgolani@cs.stonybrook.edu

Abstract

In this work, we present an implementation of the MSI page coherence protocol for multi-processing. We developed a simulator that takes memory read/write traces from Intel's Pin tool as input, collected while running the PARSEC Benchmark suite. It then uses the MSI page coherence protocol to maintain coherence among multiple process caches. We evaluated the performance of our implementation on a variety of benchmark programs and our results suggest that the MSI page coherence protocol can be an effective solution for maintaining coherence in multi-process programs. Our code can be found here: /github.com/vivekg12/MSI_protocol

1 Introduction

In modern computer systems, multiple processors often share the same memory, which can lead to cache coherence problems. When multiple processors have copies of the same memory block in their caches, it is essential to ensure that they are all consistent with the most recent version of the data. One way to maintain cache coherence is by using coherence protocols, which are mechanisms that ensure that all caches have the same view of memory.

In this work, we focus on the MSI page coherence protocol, which is a widely used coherence protocol. The MSI protocol has three states: Modified, Shared, and Invalid. The MSI protocol maintains cache coherence by using these three states to keep track of the status of each cache line.

In the Modified state, the cache has exclusive access to the memory block and can read and write to it. In the Shared state, the cache can only read the memory block, and other caches may also have copies of it. In the Invalid state, the cache has no valid copy of the memory block. The MSI protocol ensures that there is always only one cache with the Modified state for a given memory block and that all other caches have the Shared or Invalid state.

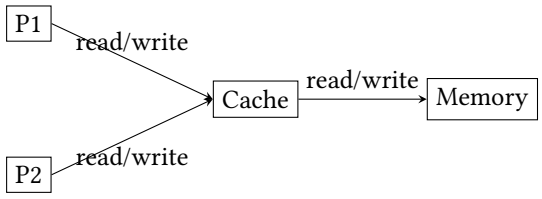

Figure 1. Two processors accessing shared memory through a cache.

In Figure. 1, there are two processors, P1 and P2, each with its own cache. The memory is shared between the two processors. When a processor reads or writes a memory location, it checks the coherence state of the corresponding cache line in its cache. If the cache line is in the Shared state, the processor can access the data directly from the cache. If the cache line is in the Modified state, the processor must write back the modified data to the main memory before other processors can access it. If the cache line is in the Invalid state, the processor must fetch the data from the main memory and update its cache accordingly.

To evaluate the effectiveness of the MSI page coherence protocol, we developed a simulator that takes memory read/write addresses from the PARSEC Benchmark suite using Intel's Pin tool as input. The PARSEC Benchmark suite is a widely used benchmark suite for multi-core processing. The suite consists of various applications, including scientific applications, multimedia applications, and web server applications.

Intel's Pin tool is a dynamic binary instrumentation tool that enables the creation of custom instrumentation tools. Pin can be used to intercept and instrument the execution of binary applications at the instruction level. Using Pin, we can obtain detailed information about the memory accesses made by the benchmark programs. This information can be used as input to our MSI page coherence simulator.

2 Related Work

Cache coherence is a well-known problem in computer architecture, and various coherence protocols have been proposed to address it. The most common coherence protocols are the MSI, MESI, and MOESI protocols. The MSI protocol is the simplest of these protocols and has been widely used in many systems.

In recent years, several studies have evaluated the performance of coherence protocols. For example, [3] proposed a new coherence protocol called the ReJIT protocol, which improves performance by reducing the number of coherence messages. Similarly, [7] proposed a new coherence protocol called the COMA protocol, which uses a directory-based approach to improve performance.

Several studies have also evaluated the performance of coherence protocols in multi-process programs. For example, [5] evaluated the performance of coherence protocols on a multi-threaded database system. They observed that the MOESI protocol performs better than the MSI protocol in this

context. Similarly, [4] evaluated the performance of coherence protocols on a multi-core processor and observed that the MESI protocol performs better than the MSI protocol.

More recently, several studies have explored the use of dynamic instrumentation tools such as Intel's Pin tool to obtain detailed information about memory accesses and evaluate the performance of coherence protocols. For example, [8] used Pin to analyze the memory access patterns of parallel applications and proposed a new coherence protocol that performs better than the MSI protocol. Similarly, [1] used Pin to obtain memory access traces and evaluated the performance of the MSI protocol on multi-core systems.

In this work, we evaluate the performance of the MSI page coherence protocol on a variety of benchmark programs using a simulator that takes memory read/write addresses from the PARSEC Benchmark suite [6]. We use Intel's Pin tool [2] to obtain detailed information about memory accesses, which enables us to accurately evaluate the performance of our implementation. Our study complements the existing literature by evaluating the MSI protocol in the context of multi-process programs using real-world benchmarks.

3 Methodology

The aim of the project is to implement a crude-level simulation of the MSI protocol. We implement two programs - MSI protocol for n processes without inter-process communication and MSI protocol for 2 processes using IPC.

3.1 MSI protocol for multi-processing

We simulate MSI protocol for n processes using n arrays that represent the caches of the processes. An i^{th} index of this array represents the MSI state (modified/shared/invalid) of that address. Ideally, when a process accesses an address for the first time, the MMU should throw a page fault before establishing a mapping from the accesses virtual address to a physical address in memory. We simulate this scenario by initializing the state of that address in the process' cache by 'N'.

When a process accesses an address, the state of that address is updated based on the operation(read or write). If the previous state was shared/modified and the operation is read, the state would not change. If the previous state was invalid and the operation is read, the process fetches the correct copy from the modified cache. An Invalid state on CPU read places a read miss on the bus and transitions to a Shared state. If the previous state was shared and the operation is write, the state in the executing process is changed to modified and the state in the rest of the processes' cache is deemed invalid.

To map the virtual address to the cache index, we implement a hash function. In case of a collision, the previous address is evicted and replaced by the new address. Ideally, there should be a better eviction policy, however, we implement this simple policy for project purposes. The virtual addresses are fetched from the output of the PIN tool which gives the read and write traces of a program.

3.2 MSI protocol using IPC

We implement another program that simulates cache coherence between two processes. The two processes communicate their page information using sockets. The program takes port numbers (local and remote) as arguments to establish a connection between two processes. After the connection is established, the processes send and receive messages through these ports.

We call one process as the server side and the other as client side. To distinguish between them, we set a boolean flag that determines if the code is running on server side or client side. To implement the IPC, we use APIs provided by the socket programming. The *socket* API creates the socket and *bind* API binds the server to the port number input by the user. *Connect* API is used to establish the connection between two sides. Finally, *send* API is used to send the messages to either side.

The first process scans user input for the number of pages to allocate. Then it creates a mapping from the virtual address to the physical address for the given number of pages. This process sends a message to the second process conveying the mapped address and the size. The second process creates a mapping too in its virtual address space.

Each process now scans user input for an operation (read/write) and the page on which this operation is to be performed. The MSI protocol is followed after every operation as explained in Section 3.1. The MSI states for each process are also printed after each operation. We present sample executions for both implementations in Section 4.

4 Experiments

This section presents sample executions for both of our implementations. We display code outputs to demonstrate the working of MSI protocol.

4.1 MSI protocol for multi-processing

PARSEC benchmark contains 13 programs from different areas including computer vision, video encoding, financial analytics, animation physics, and image processing. We use the read/write traces fetched from the PIN tool for the 'Blackscholes' program of the PARSEC benchmark. This program involves lots of computations, so the traces have a good amount of both read and write traces. Figure 2 displays a sample trace output of the PIN tool for Blackscholes program.

We have simulated a cache network with 5 processes and every process has a cache with 5 pages. Initially, all pages in all caches are marked N which indicates that no address

```
R 0x7f207a96fe80
W 0x7f207a970b00
W 0x7f207a970ba0
R 0x7f207a96fe90
W 0x7f207a970b50
R 0x7f207a96fea0
W 0x7f207a970d90
R 0x7f207a96feb0
W 0x7f207a970b58
R 0x7f207a96fec0
W 0x7f207a970b60
R 0x7f207a96fed0
W 0x7f207a970b80
```

Figure 2. sample trace output of the PIN tool for blackscholes program

is mapped to any page. Each incoming address in the PIN tool trace is mapped to a page if not already mapped. After the address is mapped to a page number, all caches have an invalid copy of the page as they haven't read on that address yet. We then modify corresponding pages in caches as follows.

Write Operation: When a write is performed on a particular address, the address is mapped to a page for all caches. The process performing the write marks that page to Modified in its cache and invalidates the same page in caches of all other processes. As seen in Figure 3, the address 0x7ffd44da56b8 is mapped to Page number: 4 which is marked I initially in caches of all processes. Since Process C has performed the write, it's cache is marked as M for page 4 whereas all other caches are invalidated.

Read Operation: When a read is performed on a particular address by a particular process, the page corresponding to that address is marked Shared in its cache. If the same page is marked Modified in any other cache, we mark it as shared too. As seen in Figure 4, a read is performed by Process A on the address 0x7fa0320cbe80, The address is mapped to Page number: 1 which is marked I initially in caches of all processes. Since Process A has performed the read, it's cache is marked as S for page 1 whereas all other caches remain in the same state.

4.2 MSI protocol using IPC

A connection is established between two processes to send and receive messages using sockets. Each process has its own cache (implemented as an array) which is printed after every operation. Each index denotes the address state of that page number.

Figures 5 and 6 show the output for a state where the caches have the correct copies for all the shared addresses. The user input is scanned for the page number and operation to perform. We have a write on page 1 on process 1 with

```
Please give Process id (A/B/C/D/E): C
Address: 0x7ffd44da56b8 mapped to Page number: 4

Process A:

|       Operation       | 0     | 1     | 2     | 3     | 4     |
|W on 0x7ffd44da56b8 by C| N    | N     | N     | N     | I     |
Process B:

|       Operation       | 0     | 1     | 2     | 3     | 4     |
|W on 0x7ffd44da56b8 by C| N    | N     | N     | N     | I     |
Process C:

|       Operation       | 0     | 1     | 2     | 3     | 4     |
|W on 0x7ffd44da56b8 by C| N    | N     | N     | N     | M     |
Process D:

|       Operation       | 0     | 1     | 2     | 3     | 4     |
|W on 0x7ffd44da56b8 by C| N    | N     | N     | N     | I     |
Process E:

|       Operation       | 0     | 1     | 2     | 3     | 4     |
|W on 0x7ffd44da56b8 by C| N    | N     | N     | N     | I     |
```

Figure 3. Cache states after write for multiprocess environment

```
Please give Process id (A/B/C/D/E): A
Address: 0x7fa0320cbe80 mapped to Page number: 1

Process A:

|       Operation       | 0     | 1     | 2     | 3     | 4     |
|R on 0x7fa0320cbe80 by A| N    | S     | N     | N     | I     |
Process B:

|       Operation       | 0     | 1     | 2     | 3     | 4     |
|R on 0x7fa0320cbe80 by A| N    | I     | N     | N     | I     |
Process C:

|       Operation       | 0     | 1     | 2     | 3     | 4     |
|R on 0x7fa0320cbe80 by A| N    | I     | N     | N     | M     |
Process D:

|       Operation       | 0     | 1     | 2     | 3     | 4     |
|R on 0x7fa0320cbe80 by A| N    | I     | N     | N     | I     |
Process E:

|       Operation       | 0     | 1     | 2     | 3     | 4     |
|R on 0x7fa0320cbe80 by A| N    | I     | N     | N     | I     |

Please give Process id (A/B/C/D/E):
```

Figure 4. Cache states after read for multiprocess environment

the message as "hello". A write operation updates the state to modified and sends an invalidation signal to process 2. Hence, page 1 on process 2 cache's is updated to invalid state as shown in Figure 6. A read operation is performed on page 0 (in shared state) which does not change the state of the cache but just prints the cache content.

Figures 7 and 8 show the output for a state where the process 1 cache has a stale copy of page 0 and process 2 cache has the latest copy. The user input is scanned for the page number and operation to perform. We have a read on page 0 on process 1. Since page 0 on process 1 cache's is invalid, process 1 will request the clean copy from process 2 (page 0 is in a modified state). Following this, process 1 fetches the clean copy from process 2 and updates the state to shared as shown in Figure 8. Process 2 now no longer has

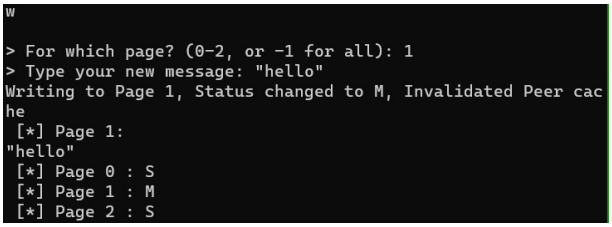

Figure 5. Write on page 1 on process 1 updates the state to modified

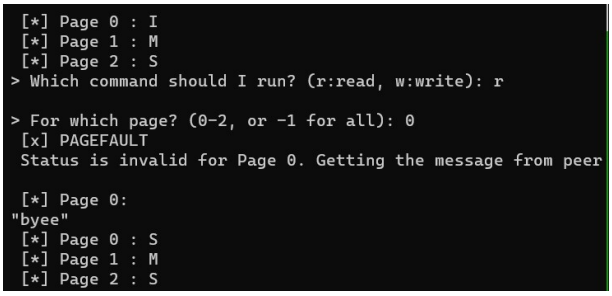

Figure 6. Read on page 0 by process 1 fetches the clean copy from process 2 and updates the state to shared.

exclusive ownership of page 0 and thus is updated to shared as well.

5 Future Work

As a future work, we would like to extend the second program which involves IPC to more processes. Such an implementation would require running n processes with each of them connected to every process via sockets. Our idea for such an implementation is that we would have to maintain a list of sockets corresponding to the connections of each process with every other process. When a process would perform a write, it would mark its cache as M and would invalidate caches of all other processes by broadcasting an invalidation message over it's list of socket connections. When the process performs a read, it would mark its cache S and the cache of the process in the M state if any to S.

We would also like to implement the MOESI protocol which is an extension of the MSI protocol and is used in modern multiprocessor systems. The MOESI protocol uses a series of state transitions to maintain cache coherence. It ensures that multiple processors or cores can access the same shared memory without producing inconsistent results. The acronym MOESI stands for Modified, Owned, Exclusive, Shared, and Invalid, which are the five states that a cache line can be in. The two new states: Owned and Exclusive are as follows:

Owned (O): When a processor has a cache line marked as "Owned," it has a copy of the data that is consistent with the

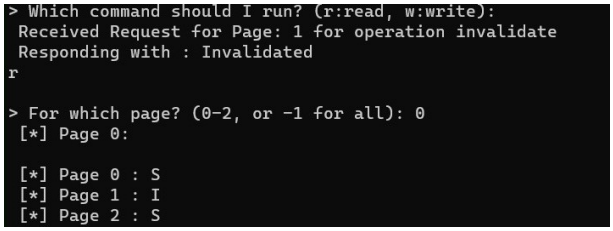

Figure 7. Write on page 1 on process 1 invalidates the page 1 index in the cache of process 2

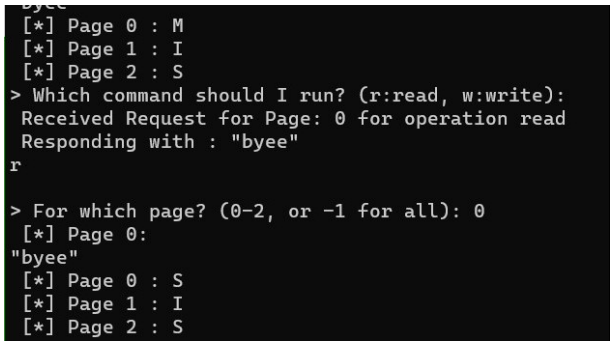

Figure 8. Process 2 now no longer has exclusive ownership of page 0 and thus is updated to shared as well.

shared memory, but it has not modified it. Other processors may read the data, but any writes must be coordinated with the owner.

Exclusive (E): When a processor has a cache line marked as "Exclusive," it has a copy of the data that is consistent with the shared memory, and it is the only processor with a copy of the data. It can read and write the data without coordinating with other processors.

6 Conclusion

This work presents an implementation of the MSI page coherence protocol for multi-processing. We implement two programs - with and without inter-process communication. The first implementation is a basic level implementation in which we emulate the caches of n processes using n arrays. The second implementation is restricted to two processes which involves the two processes sending and receiving data sharing and invalidation messages using sockets. In the experiments section, we show sample executions for both of our implementations to demonstrate the working of our implementation. For the first program, we take read-write memory traces of a PARSEC benchmark program using the PIN tool. In the second program, we show that both processes are able to effectively communicate so as to invalidate each other's cache and share data.

References

[1] Hongzhang Chen et al. “Evaluating MSI Cache Coherence Protocol Performance Using Pin”. In: *ICPP*. IEEE. 2020, pp. 1–10.

[2] *Intel’s Pin Tool*. https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html.

[3] Hyunsu Jung et al. “ReJIT: A Coherent and Optimized JIT Compiler for Scale-Out Database Systems”. In: *HPCA*. IEEE. 2021, pp. 503–516.

[4] Kyungtae Kim et al. “Cache coherence protocols in multi-threaded database systems”. In: *ASPLOS*. ACM. 2020, pp. 945–958.

[5] Soohyun Kim, Jaehyuk Jeong, and Jooyoung Lee. “Analyzing the performance of cache coherence protocols in multicore processors”. In: *HPCA*. IEEE. 2017, pp. 253–265.

[6] *PARSEC Benchmark Suite*. https://parsec.cs.princeton.edu/.

[7] Seyed Vahid Saeedi, Zhenyu Li, and Xiaolin Li. “COMA: a scalable and efficient approach to cache coherence in distributed shared memory systems”. In: *ISCA*. ACM. 2018, pp. 501–513.

[8] Huiyu Wang et al. “Towards efficient cache coherence: Analyzing memory access patterns with dynamic instrumentation”. In: *EuroSys*. ACM. 2019, pp. 1–15.