Mahidol University
International College

ICCS315 Applied Algorithms

False Sharing

Written by
Archer N. Phillips, Krittin Nisunarat 6280782

9 April 2023

Abstract

We can not deny that cache protocols are essential for improving the performance of computer systems in several ways. One essential way is by reducing the time it takes to access data from memory. However, the restriction is the size of it with a tread off. Obviously, any kinds of volatile memories have a small size, but fast storage area that stores frequently used data and instructions so that they can be accessed quickly by the processor. Nevertheless, the performance is uncertain as some data can be stored on the same address on share stage cache line. This situation is well-known in computer science that is called **false sharing**.

In general, false sharing is a performance-degrading usage pattern that can arise in systems with distributed, coherent caches at the size of the smallest resource block managed by the caching mechanism When a system participant attempts to periodically access data that is not being altered by another party, but that data shares a cache block with data that is being altered, the caching protocol may force the first participant to reload the whole cache block despite a lack of logical necessity. The caching system is unaware of activity within this block and forces the first participant to bear the caching system overhead required by true shared access of a resource. This can happen several ways which will be discussed in this paper after the experiment.

In this paper, MESIF cache coherence protocol will be discussed in detail and stage management of the cache protocol in multiprocessor. In addition, existing solution will be illustrated as many of architectures and programming languages are integrated. Finally, our false sharing solution will be determined at the end of the paper.

Contents

Chapter 1

Cache Coherence and its performance

1.1 Understanding of cache coherence in single core

Before we determine and analyze MESIF protocol, there are a few important concept about read-write memory transaction in both single core or multiple cores. To understand coherence, suppose that we are executing a single core algorithm S on x86 machine and the task that is sent to processor is to either read or write value to cache at address A where A is an arbitrary hexadecimal number. For reading memory by S algorithm, the task is scheduled by CPU with read instruction on address A at local cache. Then, the cache on the core is being searched according to the read instruction and return the value od address A that is currently stored in the cache line as shown in the figure 1.1.

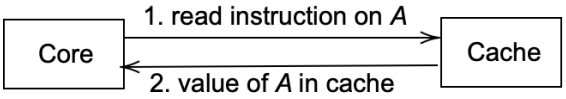

Figure 1.1: Read access on cache line

Same thing goes the same way with write access on cache. In order to write data into a cache line in a particular core, the task of S is being scheduled on core and CPU performs write instruction to address A specify by algorithm S and finish with callback. But, the value on algorithm S will be not be automatically update which the algorithm require another read instruction to the same address to retrieve a value as shown in figure 1.2. This is also the reason why writing value to cache takes more CPU cycle than read in general on both single core and multiple cores. Overall, the timeline of accessing will be write and read repeatedly. In single score, everything seems perfectly fine with the design. Once we run same algorithm in more than 1 processor, there are a drop on performance on accessing cache and memory which will be determined in the next section.

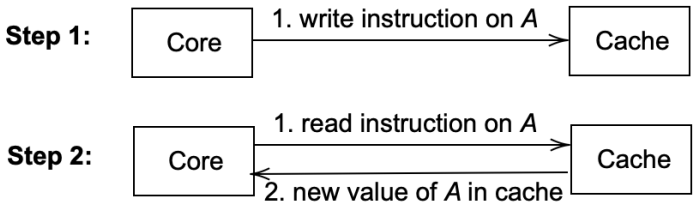

Figure 1.2: Write access on cache line

1.2 Understanding of cache coherence in multiple cores

To understanding coherence when we execute the same algorithm in parallel, suppose that we have a program S running on 3 cores C_1, C_2, C_3 and accessing the shared memory address L.

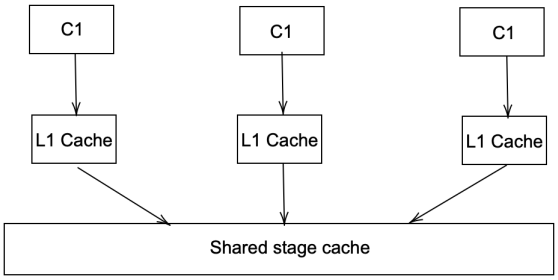

Figure 1.3: Parallel access which shared memory

Therefore, each core have to read data from address L if their local cache is missing or dirty and some cores update the value in shared stage memory. The process is similar to single core, but each core needs to communicate to each other by transferring a message in interconnection network in order to keep the 'last' write value to memory. That means every time core C_i where $i = 1, 2, 3$ updates the value in memory address L the local cache of the rest will become dirty immediately which also means that the core need to inform to others that the value is being updated and fetch the value by read instruction. Now the problem begins that decrease the performance. Due to the message delay and latency between interconnection network, the ordering of write and read disoriented as we need to guarantee that every time we read value it must be the latest update across all cores. This is also called false sharing. One way to achieve coherency for accesses is to have the writing core wait for all caches to receive the previous write value before sending a new write task which is called **an acknowledgement message (ACK)**.

1.3 Coherence for Shared Memory in Parallel Execution

As we explained in the previous section, the concept of coherence in single core applies to all cores when we run tasks simultaneously on shared memory. A single core executes accesses to a single memory location with an order. When we add caches and multiple cores accessing the same memory locations, everything does not seem to work out nicely. Caches allow cores to read values written by previous writer and all caches must be updated. There are two invariants which define what is required for a parallel execution on memory accesses to be consistent which are

1. Single-Write-Multiple-Readers (SWMR): At any time, every memory location L has either one core that may write and read L (writer-reader period), or any number of cores that can only read L (readers period) stated by Hay [2012].
2. Data-Value: The value of a memory location is the previous value written by this core if this core is writing. Or it is the last value written by the previous writing core, if there is no writer

To clarify more, SWMR invariant guarantees that for each memory location at any time, there is either one writer who can also read location L or many readers who can not write to memory. The invariant is similar to how rust compiler handle concurrency with ownership. Next, Data-Value invariant states that value of read access is either the last value written by the previous writing core (the previous writer-reader).

1.4 Cache Coherence Protocols

To understand cache coherence protocol, it is a set of rules that ensures that the data in different memories that share a common memory is kept consistent. There are a few important components that we need to understand for cache coherence protocol in a multicore system.

1.4.1 System Components

A core is a processing unit with a single thread of execution that executes some multi-threaded program consisting of multiple sequence of instructions, each core runs one sequence of instructions (Hay, 2012). Some instructions are memory operations which either read from memory locations or write to it. Each core handles memory operations using associated finite state machine (FSM) called cache controller which encodes the rules of cache coherence protocol designed to achieve coherence as shown in figure 1.4.

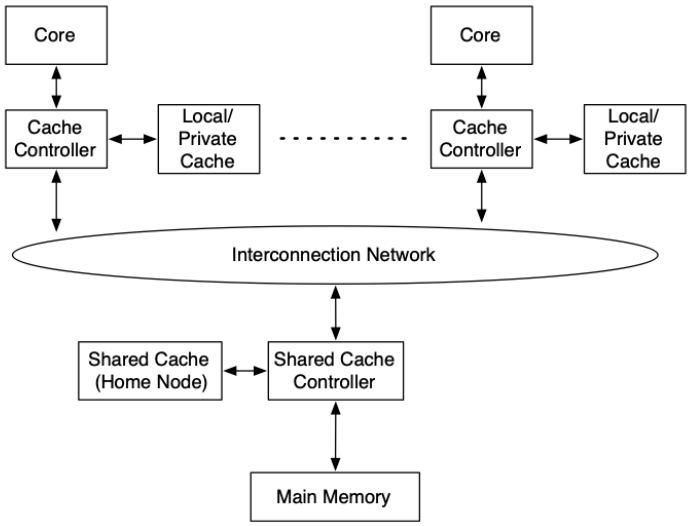

Figure 1.4: System component with cache controller and protocol provided by Hay, 2012

Memory operation enables the local cache controller to check if the local cache has the correct permissions to service the memory operation from the cache. The permissions are encoded in cache states which helps to satisfy the mentioned 2 invariants (SWMR and Data-Value). That means that if the location has the right permission, the memory operation hits in the cache and can be serviced by the local cache without communicating with other caches. Otherwise, the cache is missed adn the controller issue a memory request for the cache block requesting to write or read.

1.4.2 Interconnection Network

This component is the part that joins all local caches together in multiple cores adn the controller of each local cache. Network nodes can address other network nodes with virtual address to exchange messages and data inside a machine. Also, the cache coherence protocol uses messages and send data through the interconnection network to service memory while ensuring that the system is consistent and write-read memory across all cores. There are 3 different types of interconnection network with a different benefits and drawbacks.

1. Totally-ordered: all nodes receive all messages in the same order.
2. Point-to-Point: message sent between two nodes are delivered in order
3. Unordered: no guarantee on the ordering of messages sent through them.

In the real world, two main types are considered in most and current architecture which are

1. Bus based interconnection: Nodes communicate over a shared medium which is a set of wires.
2. Point-to-point interconnection: nodes are linked together directly into topology. A node can send message only yo nodes that they are linked together.

With bus interconnection, it guarantees that the message and data are broadcasted with order with limit bandwidth which can cause bottleneck. Point-to-point is more scalable as it has lower latency and higher bandwidth than bus interconnection.

Chapter 2

MESIF Cache Coherence Protocol

Chapter 3

Existing Solution

Chapter 4

Theoretical Solution

Reference

Andrew Hay. *MESIF Cache Coherence Protocol*. PhD thesis, ResearchSpace@ Auckland, 2012.