CACHE COHERENCE VS MEMORY CONSISTENCY

Comparison Between Cache Coherence and Memory Consistency

1. Introduction

In modern multiprocessor systems, both cache coherence and memory consistency are essential concepts to ensure the proper functioning of memory operations. Although these terms are sometimes used interchangeably, they refer to different aspects of how memory behaves in multicore systems.

2. Cache coherence

- Cache coherence ensures that all caches in a system maintain a consistent view of shared memory.
- When multiple processors have cached copies of a shared memory location, cache coherence protocols guarantee that any changes made by one processor are visible to others.
- Cache coherence is particularly crucial in multiprocessor systems where each processor has its own local cache.
- For example, if Processor 1 modifies a value in its cache, that modification must either be propagated to the caches of other processors or their copies of the value must be invalidated. Without cache coherence, different processors may operate on stale or incorrect data.

3. Memory consistency

- Memory consistency, on the other hand, deals with the ordering of memory operations across multiple processors
- It defines how operations on different memory locations should appear with respect to one another.
- In simpler terms, memory consistency determines the order in which reads and writes from different processors become visible to all other processors.

- For example, if Processor 1 writes to location A and then to location B, a memory consistency model ensures that other processors observe these writes in the correct order. Memory consistency is essential for writing correct concurrent programs and applies to systems with or without caches.

4. Key differences Between Cache Coherence and Memory Consistency

Cache Coherence	**Memory Consistency**
Ensures consistency of reads and writes to the same memory location across caches.	Defines the order of memory operations (reads/writes) across multiple memory locations.
Required in cache-equipped systems to prevent stale data.	Applies to all systems, with or without caches.
Guarantees that caches will maintain up-to-date values for a specific memory location.	Guarantees that memory operations are observed in a specific order across all processors.
Ordering of writes to the same memory location.	Ordering of reads and writes to all memory locations.
If Processor 1 writes to variable X, cache coherence ensures Processor 2 sees the updated value when reading X.	If Processor 1 writes to X and then to Y, consistency ensures all processors see these operations in the correct order.

5. Cache Coherence Protocols

Coherence protocols are essential for maintaining consistency in shared memory systems across multiple processors or cores. These protocols ensure that all processors have a coherent view of memory, even when they each have their own local caches. Let's break down the key aspects of coherence protocols and how they manage memory consistency.

Coherence protocols are designed to handle the following issues in a multi-core system:

- **Cache Coherence**: Ensuring that all copies of a memory location in different caches are consistent.
- **Consistency**: Ensuring that memory operations are observed in a consistent order by all processors.

5.1 Types of Coherence Protocols

There are several key coherence protocols, each with its own method for ensuring cache coherence:

a. MESI Protocol (Modified, Exclusive, Shared, Invalid)

The MESI protocol is one of the most widely used cache coherence protocols. It categorizes the state of each cache line into one of four states:

- **Modified (M)**: The cache line is present only in the current cache and has been modified. This means it is the only copy and is inconsistent with the main memory.
- **Exclusive (E)**: The cache line is present only in the current cache and is consistent with main memory. It has not been modified.
- **Shared (S)**: The cache line may be present in multiple caches and is consistent with main memory. All copies are the same.
- **Invalid (I)**: The cache line is not valid and should not be used.

b. MSI Protocol (Modified, Shared, Invalid)

The MSI protocol is a simpler version of MESI with only three states:

- **Modified (M)**: The cache line is present only in the current cache and has been modified.
- **Shared (S)**: The cache line may be present in multiple caches and is consistent with main memory.
- **Invalid (I)**: The cache line is not valid.

c. MOESI Protocol (Modified, Owner, Exclusive, Shared, Invalid)

The MOESI protocol extends MESI by adding an **Owner** state:

- **Modified (M)**: The same as MESI.
- **Owner (O)**: The cache line is held by one processor, and it will provide data to other processors. It is not necessarily modified.
- **Exclusive (E)**: The cache line is present only in the current cache and is consistent with main memory.
- **Shared (S)**: The cache line may be present in multiple caches and is consistent with main memory.
- **Invalid (I)**: The cache line is not valid.

d. MESIF Protocol (Modified, Exclusive, Shared, Invalid, Forward)

The MESIF protocol is an extension of the MESI protocol, introducing the Forward (F) state to further optimize cache coherence in multi-core systems. This protocol is commonly used in Intel processors. Each state in the MESIF protocol has specific rules for how cache lines are handled, and the Forward state helps reduce the amount of data traffic on the interconnect.

- **Modified (M):** The cache line is modified and different from the main memory. This cache is the only one with this data.

- **Exclusive (E):** The cache line is the same as the main memory, and this cache is the only one that has it.
- **Shared (S):** The cache line is the same as the main memory and may be present in multiple caches.
- **Invalid (I):** The cache line is not valid and must be fetched from the main memory or another cache.
- **Forward (F):** This state is similar to Shared, but this cache is designated to respond to requests for this line from other caches, reducing the need to go to the main memory

e. MOSI Protocol (Modified, Owner, Shared, Invalid)

It has one extra state than the MSI protocol, which is discussed below:

- **Owned** : It is used to signify the ownership of the current processor to this block and will respond to inquiries if another processor wants this block.

For further more information about the Cache coherence refer the below documentation:

Cache_Coherence_protocols.docx

6. Memory consistency models

A memory consistency model is a formal set of rules that specifies the order in which memory operations (loads and stores) appear to execute across all processors in a multiprocessor system. It determines the behavior of shared memory and how changes in one processor's memory are perceived by other processors.

Examples of memory models include:

Sequential Consistency (SC): All memory operations appear to execute in a single global order, where all processors see the same sequence of operations.

Relaxed Memory Models: Such as **Total Store Order (TSO), Release Consistency (RC), or Weak Consistency,** which allow certain memory operations to be reordered to improve performance while still providing consistency under specific conditions.

6.1. Sequential Consistency (SC)

Definition: Sequential Consistency requires that the results of execution appear as if all operations were executed in some sequential order, and each process sees the operations in the same order.

We have two processes, P1 and P2, and two shared variables x and y:

P1:

```
x = 1;
y = 2;
```

P2:

```
a = x;
b = y;
```

Here, P1 writes 1 to x and then 2 to y. P2 reads the values of x and y and stores them in a and b, respectively

Sequential Consistency requires that the result of execution is as if all operations were executed in some sequential order, and each process sees the operations in that order.

1. **Execution Order:**

   Suppose P1 executes x = 1 followed by y = 2.
   Then P2 reads x and y.
2. **Possible Observations under SC:**

   If P1 finishes executing both operations before P2 starts, P2 will see a = 1 and b = 2 because SC ensures that operations appear in a global, sequential order.
   The order of operations across processes is strictly preserved.
3. **Result:**

   P2 will definitely observe a = 1 and b = 2 if the reads occur after P1 has completed its writes.

6.2. Total Store Order (TSO)

Total Store Order allows some reordering of writes to different locations but maintains the order of writes to the same location. Reads might see stale values due to the reordering.

1. **Execution Order:**

   Suppose P1 writes x = 1 and then writes y = 2.
   P2 reads x and y.
2. **Possible Observations under TSO:**

   Write Reordering: TSO allows writes to different locations (x and y) to be observed in a different order than they were issued. For example, P2 might observe y = 2 before x = 1 if there's reordering.
   Write Order Preservation: Writes to the same location are observed in the order they were issued. If P1 writes x = 1 and then x = 2, P2 will see x = 2 if it reads after the write.
3. **Possible Results:**

Scenario A: If P1 writes x = 1 and then y = 2, but the writes are reordered, P2 might see a = 1 and b = 2 if it reads x after x = 1 and y after y = 2.

Scenario B: Due to potential reordering, P2 might see a = 1 and b = 0 if P1's write to y was delayed or not observed yet when P2 reads y.

6.3. Release Consistency (RC)

Definition: Release Consistency is an optimization of sequential consistency that differentiates between acquire and release operations. It requires that all operations between an acquire and a release operation be seen as consistent, but does not require a global ordering of all operations.

Example:

Consider:

P1:

```
acquire(lock);
x = 1;
release(lock);
```

P2:

```
acquire(lock);
y = x;
release(lock);
```

Under release consistency, P2 will read x = 1 only if P1 has released the lock after writing x = 1, ensuring proper synchronization.

6.4. Weak Consistency (WC)

It is a memory consistency model that provides a more relaxed guarantee compared to Sequential Consistency (SC) and Total Store Order (TSO). It is designed to improve performance by allowing more flexibility in the ordering of memory operations. However, this relaxation comes at the cost of weaker guarantees about the order of operations observed by different processes.

Key Characteristics of Weak Consistency:

1. **Relaxed Ordering: WC** allows for significant reordering of reads and writes across different memory locations. This flexibility helps in optimizing performance but can lead to less predictable behavior in terms of memory visibility.
2. **Consistency Constraints: WC** ensures consistency only when certain conditions are met. It introduces additional synchronization mechanisms (such as locks or barriers) to enforce consistency.
3. **No Global Order Guarantee:** Unlike SC, which provides a global order of operations, WC does not guarantee a single, global order of operations. The order in which operations are observed can vary based on the synchronization mechanisms used.

Example of Weak Consistency:

Let's consider two processes, P1 and P2, and two shared variables, x and y:

P1:

```
x = 1;
sync();  // Synchronization point
y = 2;
```

P2:

```
sync();  // Synchronization point
a = x;
b = y;
```

The sync() operation acts as a point where memory operations before the sync() are guaranteed to be visible to all processes or threads after the sync(). It ensures that all preceding writes are observed by other processes before any operations that follow the synchronization point.

Before Synchronization: The model allows reordering of operations before synchronization points, which means P2 can see y = 2 before x = 1 if the sync() is not yet encountered.

After Synchronization: Once synchronization points are reached, the model ensures that the operations before the synchronization in P1 are visible to P2 after the synchronization point.

7. Conclusion

1. **Cache Coherence:**
    - Ensures all processors have the same, up-to-date value for a shared memory location.
    - Prevents different processors from having different versions of the same data in their caches.
    - Example: If one processor changes a variable, all others will eventually see that new value.
2. **Memory Consistency:**
    - Defines the order in which operations (reads and writes) happen across different processors.
    - Ensures that processors see memory updates in a particular order.
    - Example: If a processor writes two values in order, other processors should see them in that same order (depending on the memory consistency model).

In short:

- Cache Coherence focuses on keeping data consistent across caches.
- Memory Consistency focuses on keeping the order of operations consistent across processors.