Cache Coherence and Cache Restrictions

Introduction

Cache coherence is a consistency mechanism in computer architecture that ensures that multiple caches in a shared memory system reflect a uniform view of data. In a multiprocessor system, processors may have individual caches, and maintaining consistency among these caches is critical for ensuring correctness in shared-memory operations.

Cache restrictions arise when designing and maintaining coherence, as specific rules must be enforced to avoid conflicts, inefficiencies, and inconsistencies.

1. Cache Coherence

1.1 Definition

Cache coherence ensures that:

1. **Read Coherence**: If a processor reads a memory location, it receives the most recent value written to that location.
2. **Write Propagation**: Changes made to a memory location by one processor are eventually reflected in the caches of other processors.

1.2 Problems in Cache Coherence

1. **Write Invalidation**: A processor writes to a memory location that another processor has cached, causing invalidation of stale copies.
2. **Write Propagation**: Delays in propagating writes can result in inconsistent views of data across processors.

1.3 Cache Coherence Protocols

Two major classes of coherence protocols:

1. **Directory-Based Protocols**:
   - A centralized directory tracks the state of each cache block.
   - Example states: Shared, Modified, Invalid.
2. **Snooping Protocols**:
   - All caches monitor a shared communication bus for coherence actions.

2. Coherence Mechanisms

2.1 MESI Protocol

The **MESI (Modified, Exclusive, Shared, Invalid)** protocol is a widely used snooping-based cache coherence protocol. Cache lines transition through four states:

1. **Modified (M)**: Line is modified in this cache and not in memory.
2. **Exclusive (E)**: Line is present in this cache only, but matches memory.
3. **Shared (S)**: Line is in multiple caches and matches memory.
4. **Invalid (I)**: Line is invalid.

State Transition Diagram: For example, in MESI:

- **Read miss** $\rightarrow$ Invalid to Shared or Exclusive.
- **Write miss** $\rightarrow$ Invalid to Modified.

2.2 Write Invalidation vs. Write Update

- **Write Invalidation**: The writer invalidates other caches and writes exclusively to its cache.
- **Write Update**: The writer updates other caches with the new value.

2.3 Coherence Overhead

The cost of maintaining coherence can be expressed as:

$$T_{\text{coherence}} = T_{\text{sync}} + T_{\text{comm}}$$

Where:

- T_{sync}: Time for synchronization.
- T_{comm}: Communication time to propagate updates or invalidations.

3. Cache Restrictions

3.1 Cache Consistency

Consistency defines how memory updates are observed by processors.

Sequential Consistency:

- Ensures that all processors observe memory operations in the same order.
- Requires synchronization and incurs overhead:

$$T_{\text{total}} = T_{\text{comp}} + T_{\text{sync}} + T_{\text{comm}}$$

Where:

- T_{comp}: Computation time.
- T_{sync}: Synchronization overhead.
- T_{comm}: Communication delay.

3.2 Restrictions on Cache Design

1. **Inclusion Property**:
   - Ensures that data present in the cache of any processor is also present in higher-level caches (e.g., shared cache).
   - Avoids stale data but increases memory overhead.
2. **Exclusion Property**:
   - Data is stored in only one cache level at a time.
   - Reduces memory usage but complicates coherence protocols.
3. **Cache Line Size**:
   - Large lines improve spatial locality but increase false sharing (when multiple processors update different parts of the same cache line).

3.3 False Sharing

False sharing occurs when multiple processors modify different variables within the same cache line, causing unnecessary coherence traffic.

Cost of False Sharing:

$$T_{\text{false\_sharing}} = N \cdot T_{\text{inv}}$$

Where:

- N: Number of invalidations.
- T_{inv}: Time per invalidation.

4. Mathematical Models for Cache Coherence

4.1 Coherence Misses

Coherence misses occur when a processor accesses a cache line that has been invalidated due to another processor's actions.

$$T_{\text{miss}} = T_{\text{access}} + T_{\text{coherence}}$$

Where:

- T_{access}: Access time for a local or shared cache.
- $T_{\text{coherence}}$: Time to retrieve or invalidate the cache line.

4.2 Bandwidth Usage

The bandwidth required for maintaining coherence:

$$B_{\text{coherence}} = N \cdot S_{\text{msg}} / T_{\text{cycle}}$$

Where:

- N: Number of coherence messages.
- S_{msg}: Size of each message.
- T_{cycle}: Time per communication cycle.

4.3 Scalability and Overhead

The total communication overhead in a multiprocessor system:

$$T_{\text{overhead}} = \frac{N_{\text{proc}}}{B} \cdot (T_{\text{latency}} + T_{\text{coherence}})$$

Where:

- N_{proc}: Number of processors.
- B: Bandwidth of the communication network.

5. Applications of Cache Coherence

1. **Multiprocessor Systems**:
   - Shared memory systems like NUMA and SMP.
2. **Distributed Systems**:
   - Maintaining consistency across distributed caches.
3. **Database Systems**:
   - Cache coherence for transaction consistency.

Conclusion

Cache coherence is critical for ensuring consistency in multiprocessor systems. While it introduces restrictions, such as inclusion and exclusion properties, and challenges like false sharing, coherence protocols like MESI effectively maintain a uniform memory view. Mathematical models help optimize coherence mechanisms, balancing performance and overhead for scalable, high-performance systems.