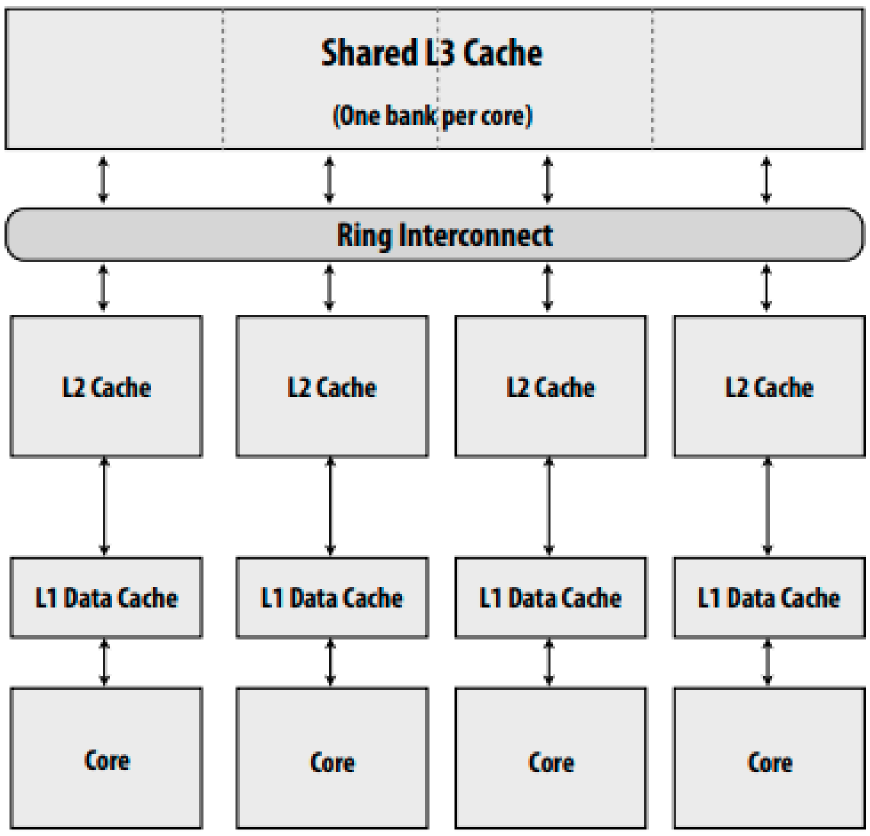
CACHE COHERENCE AND ITS PROTOCOLS

INDEX

1. Cache Coherence

A cache coherence issue results from the concurrent operation of several processors and the possibility that various caches may hold different versions of the identical memory block. The practice of cache coherence makes sure that alterations in the contents of associated operands are quickly transmitted across the system.The cache coherence problem is the issue that arises when several copies of the same data are kept at various levels of memory.

Cache coherence is the discipline which ensures that the changes in the values of shared operands (data) are propagated throughout the system in a timely fashion.



2. Coherence Mechanism

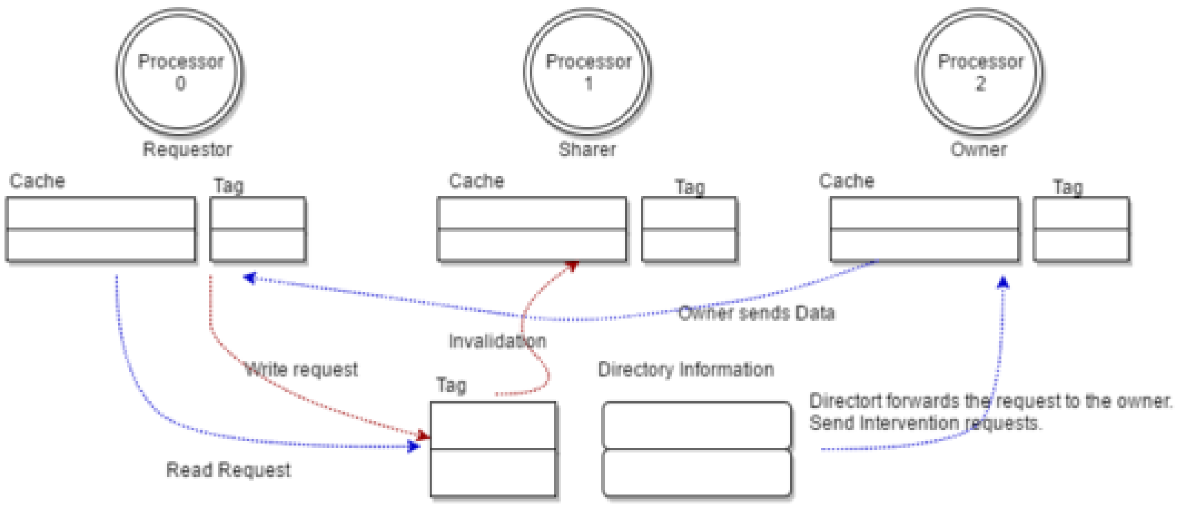
The two most common mechanisms of ensuring coherence are *snooping* and *directory-based*, each having their own benefits and drawbacks .

> Snooping based protocols tend to be faster, if enough bandwidth is available, since all transactions are a request/response seen by all processors. The drawback is that snooping isn't scalable. Every request must be broadcast to all nodes in a system, meaning that as the system gets larger, the size of the (logical or physical) bus and the bandwidth it provides must grow.
>
> Directories, on the other hand, tend to have longer latencies but use much less bandwidth since messages are point to point and not broadcast. For this reason, many of the larger systems (>64 processors) use this type of cache coherence.

2.1 Directory-based

In a directory-based system, the data being shared is placed in a common directory that maintains the coherence between caches. The directory acts as a filter through which the processor must ask permission to load an entry from the primary memory to its cache. When an entry is changed, the directory either updates or invalidates the other caches with that entry.

Unlike snoopy coherence protocols, in a directory based coherence approach, the information about which caches have a copy of a block is maintained in a structure called *directory*. In a directory based scheme, participating caches do not broadcast requests to all other sharing caches of the block in order to locate cached copies, instead it queries the directory to retrieve the information about which block have cached copies and sends only to those particular processors and hence traffic saving compared to a snoopy protocol is large.



As shown in the data flow diagram, the actors involved in a distributed shared memory system implementing directory based coherence protocol are:

- **Requestor Node**: This node is the processor who is requesting for a read/write of a memory block.
- **Directory Node**: This node maintains the information of the state of each cache block in the system and the requestor directs its requests to the directory node.
- **Owner Node**: An owner node owns the most recent state of the cache block, note that directory might not always be up to date with latest data.
- **Sharer Node**: One or many nodes which are sharing a copy of the cache block.

Requestor and Owner nodes maintain their state transition similar to snoopy coherence protocols like MESI protocol. However, unlike a bus based implementation where nodes communicate using a common bus, directory based implementation uses message passing model to exchange information required for maintaining cache coherence.

Directory node acts as a serializing point and all communications are directed through this node to maintain correctness.

2.1.1 Directory node and states

A directory node keeps track of the overall state of a cache block in the entire cache system for all processors. It can be in three states :

- **Uncached (U)**

  No processor has data cached, memory up-to-date .
- **Shared (S)**

  one or more processors have data cached, memory up-to-date. In this state directory and sharers have a clean copy of the cached block.
- **Exclusive/Modified (EM)**

  one processor (owner) has data cached; memory out-of-date. Note that the directory cannot distinguish a block cached in an exclusive or modified state at the processor, as processors can transition from an exclusive state to modified state without any bus transaction.

2.2 Snooping

First introduced in 1983, snooping is a process where the individual caches monitor address lines for accesses to memory locations that they have cached.The *write-invalidate protocols* and *write-update protocols* make use of this mechanism.

The snooping mechanism is a key technique in maintaining cache coherence in multi-processor systems. It involves caches "snooping" or monitoring the address lines to ensure that they maintain a consistent view of memory. Here’s a detailed explanation of how snooping works, including the role of snoop filters:

1. **Basic Concept:**

   Snooping: Each cache in a system listens to all memory transactions (reads and writes) on the address bus. This allows caches to observe when other caches access or modify memory locations that they also hold.

   Write-Invalidate Protocols: When a cache writes to a memory location, it broadcasts an invalidate message to all other caches holding a copy of that location, ensuring that they invalidate their copies.

   Write-Update Protocols: When a cache writes to a memory location, it sends an update message to all other caches holding a copy, allowing them to update their copies with the new data.

2.2.1 Snoop Filter

A snoop filter is a mechanism designed to optimize the performance of snooping by reducing unnecessary snooping traffic. Here's how it works:

1. **Purpose of the Snoop Filter:**

   Reduce Traffic: By maintaining a set of entries representing cache lines that may be owned by one or more caches, the snoop filter helps to minimize the amount of traffic on the address bus. It prevents all caches from having to monitor all memory transactions, reducing overhead and contention.

2. **How the Snoop Filter Works:**

   Entries: The snoop filter maintains a table (or filter) with entries that correspond to memory cache lines. Each entry represents a cache line that may be present in one or more caches.

   Presence Vector: Each entry includes a presence vector that indicates which caches have a copy of the cache line. The presence vector helps the snoop filter determine how many caches hold the line and whether it needs to be invalidated or updated.

   Replacement Policy: When a cache line in the snoop filter needs to be replaced (e.g., due to space constraints), the filter selects the entry representing the cache line that is owned by the fewest nodes. This is determined using the presence vector.

3. **Replacement Algorithm:**

Selection Criteria: The snoop filter selects the cache line entry with the fewest owners for replacement. This minimizes the impact of the replacement on cache coherence, as fewer caches are affected.

Temporal or Other Algorithms: If there are multiple cache lines with the same minimal number of owners, the filter uses additional algorithms (such as temporal or least recently used) to refine the selection. This ensures that the replacement decision takes into account additional factors beyond just the number of owners.

Temporal or Least Recently Used (LRU) Algorithm:

Temporal: This approach considers how long an entry has been in the filter. Older entries might be replaced in favor of newer ones, assuming that recently accessed lines are more likely to be needed again soon.

Least Recently Used (LRU): This algorithm tracks the order of accesses to the entries. The entry that has not been used for the longest time is chosen for replacement. This helps in keeping entries that are frequently accessed and likely to be reused

3. Cache line

A cache line is a fixed-size block of memory that is the smallest unit of data transfer between the main memory and the cache. Typical cache line sizes range from 32 to 128 bytes.

Example of Cache Line Operation:

Memory Request: The CPU needs to read a value from memory address 0x1000.

Cache Check: The CPU checks if the cache contains the data for address 0x1000.

Steps:

1. **Cache Miss:**

If the data is not in the cache, a cache miss occurs. The CPU fetches the entire cache line containing address 0x1000 from the main memory. Suppose the cache line size is 64 bytes; the fetched cache line might include addresses 0x1000 to 0x103F.

2. **Loading Cache Line:**

   The fetched cache line is loaded into the cache. The tag for this cache line is set to identify the memory range it covers (e.g., 0x1000).
   The **tag** is part of the metadata stored alongside each cache line. It helps uniquely identify which block of main memory the data in the cache line corresponds to. The tag is crucial for determining whether the data requested by the CPU is present in the cache (cache hit) or if it needs to be fetched from the main memory (cache miss).

3. **Subsequent Accesses:**

   If the CPU needs data from address 0x1001 or 0x1010, it will find it in the cache, resulting in a cache hit, thus avoiding the need to access slower main memory.

3.1 Performance Considerations

3.1.1 Cache Line Size

Larger cache lines can reduce the number of cache misses by fetching more data per miss, leveraging spatial locality. However, they may increase cache pollution if unnecessary data is fetched.

3.1.2 Cache Line Replacement

When a new cache line needs to be loaded, an existing line might need to be evicted based on cache replacement policies (e.g., LRU, FIFO).

3.1.3 Write Policies

When the CPU writes data to the cache, there are two primary policies that determine how and when these writes are propagated to the main memory: Write-Through and Write-Back. Each policy has its own implications for performance and consistency.

3.1.3.1 Write-Through

In the **Write-Through** policy, every write operation to the cache is immediately written to the main memory as well. This means that any change made to a cache line is simultaneously made to the corresponding location in the main memory.

3.1.3.2 Write-Back

If the cache is using the **Write-Back** policy, the evicted cache line might be "dirty." A dirty cache line means it has been modified in the cache but not yet written to the main memory.

Before eviction, the data in the dirty cache line is written back to the main memory to ensure that the main memory has the most up-to-date data.
Once the dirty cache line is written back, it can be safely evicted from the cache to make space for the new data

3.2 Evicted Cache Line in Write-Back Policy

3.2.1. What is an Evicted Cache Line?

An evicted cache line is a line of data in the cache memory that is being removed or replaced to make room for new data. This typically occurs when the cache is full, and new data needs to be loaded into the cache.

3.2.2. Write-Back Policy and Dirty Cache Lines

In a **Write-Back** cache policy:

Dirty Cache Line: A cache line is marked as "dirty" if it has been modified in the cache but not yet written back to the main memory. This means the data in the cache is more up-to-date than the corresponding data in the main memory.
Clean Cache Line: A cache line is "clean" if it has not been modified or if any modifications have already been written back to the main memory.

3.2.3. Eviction Process

When a dirty cache line is evicted under the Write-Back policy, the following steps occur:

1. **Identify the Dirty Cache Line:**

   The cache controller identifies the cache line that needs to be evicted based on a replacement policy (e.g., Least Recently Used (LRU), First-In-First-Out (FIFO)).
2. **Write Back to Main Memory:**

   If the cache line to be evicted is dirty, the cache controller writes the modified data from the cache line back to the main memory. This ensures that the main memory is updated with the latest data.
3. **Evict the Cache Line:**

   After writing back to the main memory, the cache line can be safely evicted. The space in the cache is now available for new data to be loaded.
4. **Load New Data:**

   The new data is loaded into the cache, occupying the space of the evicted cache line.

4. Bus Transactions

Bus transactions in a multiprocessor system are crucial for ensuring that multiple caches maintain a consistent view of memory and for coordinating memory operations. Here's a detailed look at each transaction type and their roles:

4.1. Write-Through

Definition: A write-through cache policy ensures that when data is written to the cache, it is simultaneously written to the main memory.
Details:

When: Used to ensure data consistency between the cache and main memory.

Advantages: Simplifies consistency models because main memory is always up-to-date.

Disadvantages: Increased traffic to main memory due to every write operation.

Example:

Processor A writes 0xABCD to address 0x2000.

Data 0xABCD is written to both the cache and main memory.

4.2. Write-Back

Definition: A write-back cache policy only writes data to the cache. The data is written to main memory only when the cache line is evicted.

Details:

When: Used to reduce memory traffic by deferring writes to main memory.

Advantages: Reduces the number of write operations to main memory.

Disadvantages: Main memory may be outdated, leading to potential consistency issues.

Example:

Processor A writes 0xABCD to address 0x2000.

Data 0xABCD is written to the cache. The write to main memory occurs only when the cache line is evicted.

4.3. Write Allocate

Definition: Write-allocate (fetch-on-write) means that if a write operation is performed on an address not currently in the cache, the cache line is first fetched from main memory and then updated.

Details:

When: Useful when the address is likely to be read again soon.

Advantages: Subsequent accesses to the address can benefit from cache speeds.

Disadvantages: May lead to unnecessary cache line allocations.

Example:

Processor A writes to address 0x3000, which is not in the cache.

The cache line for 0x3000 is fetched from main memory, and then the data is written to the cache.

4.4. Write-No-Allocate

Definition: Write-no-allocate (no-fetch-on-write) means that if a write operation is performed on an address not currently in the cache, the data is written directly to main memory without fetching the line into the cache.

Details:

When: Used for write-heavy workloads or when future reads are unlikely.

Advantages: Reduces unnecessary cache line allocations.

Disadvantages: Subsequent reads to the address will miss the cache, potentially increasing latency.

Example:

Processor A writes to address 0x4000, which is not in the cache.

Data 0xABCD is written directly to main memory.

4.5. Cache Intervention

Definition: Cache intervention involves a cache operation where one cache may need to notify other caches to update or invalidate their copies of a cache line to maintain coherence.

Intervention occurs when a cache that holds a valid copy of a cache line supplies the data to another cache that requests it, often bypassing the need to fetch the data from the main memory. This typically happens in the following scenarios:

- **Read Requests:** When a processor wants to read a cache line that it does not have, it sends a read request (e.g., BusRd). If another cache holds this line in a valid state (e.g., Modified, Exclusive, or Shared), it can intervene and provide the data directly.
- **Efficiency:** This process improves efficiency by reducing the latency of memory accesses and decreasing the load on the main memory.

Types:

4.5.1 Shared Intervention

Updates or invalidates copies in caches that have the line in the Shared state.

4.5.2 Dirty Intervention

Updates or invalidates copies in caches that have the line in the Modified state.

Example:

Processor A writes to address 0x5000.

Processor B, which has address 0x5000 cached, receives an intervention to invalidate or update its copy.

4.6. Invalidation

Invalidation occurs when a cache line is marked as invalid in one or more caches to maintain coherence when the line is being modified. This ensures that no stale copies of the data are read by other processors. Invalidation is a crucial mechanism in write-invalidate protocols, where write operations trigger the invalidation of other copies of the cache line.

Types of Invalidation:

4.6.1 Write Invalidate

When a processor wants to write to a cache line, it sends an invalidation request (e.g., BusRdX or BusUpgr). All other caches holding this line must invalidate their copies before the write can proceed.

4.6.2 Invalidate Acknowledgement

Caches that invalidate their copies might send acknowledgements back to the requesting cache, ensuring the write operation can proceed safely.

Details:

When: Typically used in response to write operations to maintain cache coherence.

Advantages: Prevents the use of outdated or incorrect data.

Disadvantages: Can generate additional traffic on the system bus due to invalidation messages.

Example:

Processor A writes to address 0x6000.

All other caches that have 0x6000 are sent invalidation signals, marking their lines as Invalid.

Interaction Example: Consider a multiprocessor system with three processors, P1, P2, and P3, each with their own cache.

Scenario:

1. **P1 Reads X:** P1 sends a BusRd for cache line X.
    - **C2 Intervenes:** If C2 holds X in Modified state, it intervenes and provides X to P1, transitioning X to Shared state in both caches.
2. **P3 Writes to X:** P3 wants to write to X.
    - **P3 Sends BusRdX:** P3 sends a BusRdX for X.
    - **C1 and C2 Invalidate:** C1 and C2, holding X in Shared state, invalidate their copies.
    - **P3 Writes to X:** After invalidation, P3 writes to X and transitions X to the Modified state.

4.6.3 Cache Coherence Protocols Using Intervention and Invalidation

1. **MESI Protocol (Modified, Exclusive, Shared, Invalid):**
    - **Intervention:** Occurs when a cache line in the Modified or Exclusive state is read by another processor, transitioning to Shared state.
    - **Invalidation:** Occurs when a processor writes to a line, invalidating other copies.
2. **MOESI Protocol (Modified, Owner, Exclusive, Shared, Invalid):**
    - **Intervention:** The Owner state allows a cache to supply data directly to other caches.

    - **Invalidation:** Similar to MESI, ensuring other copies are invalidated before a write.
3. **MSI Protocol (Modified, Shared, Invalid):**
    - **Intervention:** Less efficient than MESI or MOESI as it doesn't have an Exclusive state, leading to more frequent invalidations.
    - **Invalidation:** Essential for maintaining coherence when writes occur.

4.7. Write-Broadcast

Definition: Write-broadcast involves broadcasting a write operation to all caches to ensure they are aware of the update and can perform necessary coherence actions.

Details:

When: Ensures that all caches are updated or invalidated to maintain consistency.

Advantages: Ensures all caches receive the update.

Disadvantages: Increases traffic on the system bus.

Example:

Processor A writes to address 0x7000.

A broadcast message is sent to all caches to update or invalidate their copies of 0x7000.

4.8. Intervention-Broadcasting

Definition: Intervention-broadcasting refers to broadcasting an intervention action (such as an invalidation or update) to all caches to ensure they perform the necessary actions to maintain coherence.

Details:

When: Used to inform all caches about changes to a particular cache line.

Advantages: Facilitates global coherence by notifying all relevant caches.

Disadvantages: Can lead to increased system bus traffic.

Example:

Processor A writes to address 0x8000.

An intervention-broadcast message is sent to other processors, requesting them to update or invalidate their caches for address 0x8000.

4.9. Flush Operation

- **Flush** is used to ensure that any modifications made to a cache line (which is in the Modified state) are propagated to the main memory. This operation is crucial for maintaining consistency between the cache and main memory and for ensuring that other caches or processors have the most up-to-date data.

When It Occurs:

1. **Cache Eviction:**
   - When a cache line needs to be replaced or evicted from the cache (e.g., due to a cache miss or replacement policy), and if the line is dirty (Modified state), it must be flushed to memory before it is removed. This ensures that the main memory has the latest version of the data.
2. **Coherence Protocols:**
   - The flush operation is often integrated into coherence protocols like MESI (Modified, Exclusive, Shared, Invalid). When a cache line in the Modified state is invalidated or replaced, it is flushed to memory.

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.

6. Cache States

6.1. Modified (M)

Definition

Modified: The cache line is present only in the current cache, has been modified (i.e., written to), and is not consistent with the main memory. This means that the data in this cache line is different from the data in the main memory.

Responsibilities

Write-back: The cache that holds a line in the Modified state is responsible for writing the data back to main memory before the line can be read by another cache or evicted from the cache.

Example Scenario

Processor A writes to a memory location x. This causes the cache line containing x to enter the Modified state in Processor A's cache. No other cache has a copy of this line.

If **Processor B** wants to read x, Processor A must write back the modified data to the main memory first. Processor B then reads the updated data from the main memory.

6.2. Exclusive (E)

Definition

Exclusive: The cache line is present only in the current cache, has not been modified (i.e., it has only been read), and is consistent with the main memory. This means that the data in this cache line matches the data in the main memory.

Responsibilities

No write-back needed: Since the data is consistent with the main memory, there is no need to write back the data to the main memory when this cache line is invalidated or shared.

Example Scenario

Processor A reads a memory location y that is not in any other cache. The cache line containing y enters the Exclusive state in Processor A's cache.

If **Processor A** writes to y, the state transitions to Modified.

6.3. Shared (S)

Definition

Shared: The cache line may be present in multiple caches and is consistent with the main memory. All copies of the cache line are the same, and all are consistent with the main memory.

Responsibilities

Read-only: Caches can read the cache line, but if a cache wants to write to it, it must first obtain exclusive ownership of the line, causing other caches to invalidate their copies.

Example Scenario

Processor A and **Processor B** both read a memory location z. The cache line containing z enters the Shared state in both Processor A's and Processor B's caches.

If **Processor A** wants to write to z, it sends an invalidation request. Processor B's copy of z transitions to the Invalid state, and Processor A's copy transitions to the Modified state.

6.4. Invalid (I)

Definition

Invalid: The cache line is not valid in the current cache and cannot be used. Any attempt to read or write to this line will result in a cache miss.

Responsibilities

No data access: The cache must fetch the data from the main memory or another cache that holds a valid copy of the line when an attempt is made to access the invalid line.

Example Scenario

Processor A writes to a memory location w that is also present in **Processor B's** cache. The cache line containing w transitions to the Invalid state in Processor B's cache after Processor A's write operation.

6.5. Owner (O)

- **Definition**: The cache line has been modified and is consistent with the main memory. It is present in multiple caches, but **only the cache in the Owner state can write the data back to the main memory.**
- **Responsibilities**: The cache that holds a line in the Owner state can supply the data to other caches, reducing the need for multiple write-backs to the main memory.
- **Owner State** (O) allows one cache to be responsible for updating the main memory, reducing the number of write-backs.
- The **Owner** cache can supply the data to other caches, maintaining coherence while optimizing memory traffic
- **Example Scenario**

  Consider a multiprocessor system with two processors, P1 and P2, each with its own cache. We have a memory location X that both processors will access.

  - **Initial State**

    Memory: X = 0

    P1 Cache: X is Invalid (I)

    P2 Cache: X is Invalid (I)

  Step-by-Step Operations

  1. P1 Reads X

  P1 issues a read miss for X.

  Since X is not present in any cache, P1 fetches X from the main memory.

  P1 Cache: X is now Exclusive (E).

  Memory: X = 0

  Summary: X is loaded into P1's cache with the value 0 and is marked as Exclusive (E) because P1 is the only cache with X.

  2**. P2 Reads X**

  P2 issues a read miss for X.

  P2 fetches the value of X from P1’s cache.

P1 changes the state of X from Exclusive (E) to Owner (O), and P2 sets X to Shared (S).

P1 Cache: X is now Owner (O).

P2 Cache: X is now Shared (S).

Summary: X is now shared between P1 and P2, with P1 having the Owner (O) state and P2 having the Shared (S) state.

3. P1 Writes to X

P1 wants to write to X.

P1 invalidates X in P2’s cache by issuing an invalidation signal.

P2 sets X to Invalid (I).

P1 writes the new value to X and changes its state to Modified (M).

P1 Cache: X = new value, state is Modified (M).

P2 Cache: X is Invalid (I).

Summary: After the write, P1 has the only valid copy of X, and it is marked as Modified (M).

4. P2 Reads X Again

P2 issues a read miss for X.

P1, having X in the Modified (M) state, must write back X to the main memory before supplying it to P2.

P1 writes X to memory and transitions X to the Owner (O) state.

P2 fetches the value of X and sets it to Shared (S).

P1 Cache: X is Owner (O).

P2 Cache: X is Shared (S).

Memory: X = new value.

Summary: X is shared again between P1 and P2, with P1 in the Owner (O) state and P2 in the Shared (S) state.

6.6. Forward (F)

Definition: The cache line is present in multiple caches and is consistent with the main memory, similar to the Shared state. However, the cache in the **Forward state is designated to supply the data to other caches requesting it.**
Responsibilities: The cache in the Forward state is responsible for responding to read requests from other caches, reducing the number of requests to the main memory.

Example Scenario:

Processor A reads a memory location x that is not in its cache (Invalid state).
Processor A fetches the data from the main memory, and the state of x transitions to Exclusive in Processor A's cache.
If **Processor B** also reads x, the state transitions to Shared (S) in both Processor A's and Processor B's caches, with one of them designated as Forward (F).

Example Scenario in Detail

Consider a multiprocessor system with two processors, P1 and P2, each with its own cache. We have a memory location X that both processors will access.

- **Initial State**

  Memory: X = 0
  P1 Cache: X is Invalid (I)
  P2 Cache: X is Invalid (I)

Step-by-Step Operations

1. P1 Reads X

P1 issues a read miss for X.
Since X is not present in any cache, P1 fetches X from the main memory.
P1 Cache: X is now Exclusive (E).
Memory: X = 0

Summary: X is loaded into P1's cache with the value 0 and is marked as Exclusive (E) because P1 is the only cache with X.

2. P2 Reads X

P2 issues a read miss for X.

P2 fetches the value of X from P1’s cache.

P1 changes the state of X from Exclusive (E) to Shared (S), and P2 sets X to Forward (F).

P1 Cache: X is now Shared (S).

P2 Cache: X is now Forward (F).

Summary: X is now shared between P1 and P2, with P2 having the Forward (F) state and P1 having the Shared (S) state.

3. P1 Writes to X

P1 wants to write to X.

P1 invalidates X in P

7. Cache operations and State Transitions

1. Read Miss

A **read miss** occurs in a cache when the processor requests a read operation for a data item that is not currently present in the cache. This triggers a series of actions to retrieve the data from a higher level in the memory hierarchy (e.g., another cache, main memory) and bring it into the cache

- **Invalid to Exclusive (if no other cache holds the line)**
  - **Condition:** If the cache line is in the Invalid state and no other cache holds a copy of the line.
  - **Transition:** The cache sends a read request to the memory or the bus. If it’s the only cache interested in the line, it transitions to the Exclusive state.
  - **Details:** In the Exclusive state, the cache has the only copy of the data and can modify it without needing to communicate with other caches.
- **Invalid to Shared (if another cache holds the line)**
  - **Condition:** If the cache line is in the Invalid state but another cache holds the line in Shared or Modified state.

    - **Transition:** The cache sends a read request to the bus. The bus will broadcast this request, and if other caches have the line, they provide the data. The cache transitions to the Shared state.
    - **Details:** In the Shared state, the cache has a copy of the data, but the data might also be in other caches.

2. Write Miss

A **write miss** occurs in a cache when the processor attempts to write to a data item that is not currently present in the cache. This situation requires the cache coherence protocol to take steps to ensure that the write operation is correctly handled and that all other caches maintain coherence

- **Invalid to Modified (with invalidation request)**
    - **Condition:** If the cache line is in the Invalid state.
    - **Transition:** The cache sends a write request and an invalidation signal to other caches. Once the invalidation is acknowledged, the cache transitions to the Modified state.
    - **Details:** In the Modified state, the cache has the exclusive right to modify the data and must eventually write it back to the main memory.
- **Shared to Modified (with invalidation request)**
    - **Condition:** If the cache line is in the Shared state.
    - **Transition:** The cache sends a write request and invalidation signals to other caches holding the line. Once the invalidation is acknowledged, the cache transitions to the Modified state.
    - **Details:** In the Modified state, the cache has updated the data and now has the sole copy.

3. Read Hit

A **read hit** occurs in a cache when the processor requests a read operation for a data item that is already present in the cache. This is a straightforward scenario in which the requested data can be accessed directly from the cache without needing to interact with other caches or main memory.

- **Shared or Exclusive (no state change)**
    - **Condition:** If the cache line is in the Shared or Exclusive state.
    - **Transition:** The cache can read the data without changing its state.
    - **Details:**
        - In the Shared state, multiple caches can hold a copy of the data.
        - In the Exclusive state, the cache has the only copy and can access the data without changing the state.

4. Write Hit

A **write hit** occurs in a cache when the processor attempts to write to a data item that is already present in the cache

- **Exclusive to Modified:**
    - **Condition:** If the cache line is in the Exclusive state.
    - **Transition:** The cache writes to the data, which causes the state to transition to Modified.
    - **Details:** After the write, the cache has the sole copy of the data and must eventually update the main memory. Other caches are invalidated.

5. Invalidate

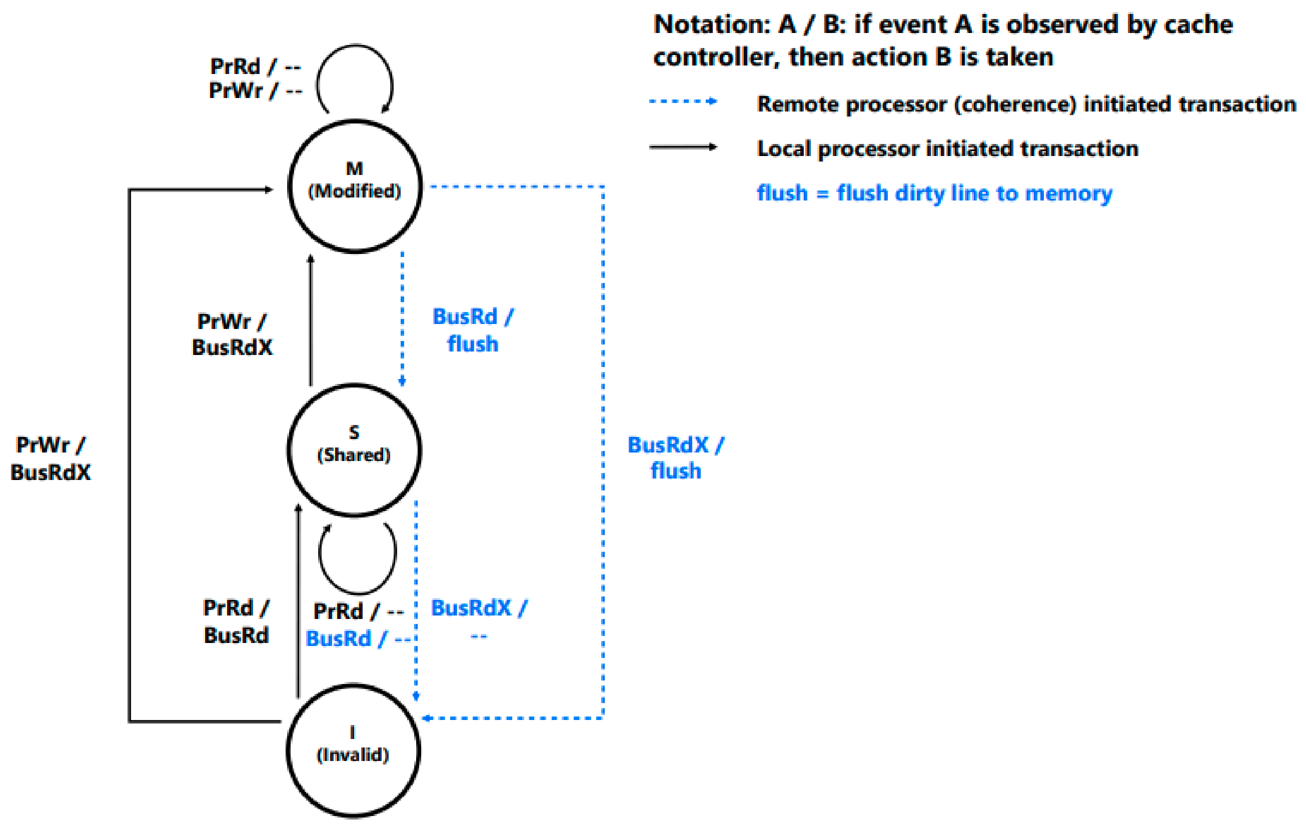
Invalidation in the context of cache coherence protocols is a mechanism used to ensure that a cache line is marked as invalid in other caches. This process helps maintain data consistency across multiple caches in a multiprocessor system. When a cache line is invalidated, it means that the data in that cache line is no longer considered valid, and if a processor needs to access that data again, it must fetch the latest version from main memory or another cache.

- **Shared/Modified to Invalid:**
    - **Condition:** If another cache wants to write to a line currently in Shared or Modified state.
    - **Transition:** The cache receiving the invalidate signal must transition the line to the Invalid state.

- **Details:** This ensures that no stale data is used. In the Shared state, it invalidates the line as other caches are updated. In the Modified state, it invalidates the line and must eventually write back the updated data to memory.

8. State transition diagrams

8.1. MSI state transition diagram


Notation: A / B: if event A is observed by cache controller, then action B is taken
Remote processor (coherence) initiated transaction
Local processor initiated transaction
flush = flush dirty line to memory
PrRd / --
PrWr / --
M (Modified)
PrWr / BusRdX
BusRd / flush
S (Shared)
PrWr / BusRdX
BusRdX / flush
PrRd / BusRd
PrRd / --
BusRd / --
BusRdX / --
I (Invalid)

Remote Processor Initiated Transactions

Remote processor-initiated transactions involve actions initiated by processors other than the one where the cache line is currently being accessed. These transactions are crucial for maintaining cache coherence across multiple processors.

Local Processor Initiated Transactions

Local processor-initiated transactions involve actions triggered by the processor whose cache is being accessed. These transactions are initiated in response to read or write operations performed by the local processor.

1. BusRd (Read Request):

- **Description:** A remote processor sends a BusRd request when it needs to read a cache line. This is a request for data from a memory location or cache line without modifying it.
- **Actions:**
  - The cache that holds the line in a Modified state must send the data back to the requester and write the data to memory (flush).
  - The line transitions to the Shared state in the cache that responds.
  - If the line is already in the Shared state in other caches, they provide the data to the requester.

2. BusRdX (Read-Modify Request):

- **Description:** A remote processor sends a BusRdX request when it needs to read a cache line and also intends to modify it.
- **Actions:**
  - The cache that has the line in the Shared state must invalidate its copy and provide the data.
  - If the line is in the Modified state, the cache writes the data back to memory and then sends it to the requester.
  - The cache with the line transitions to Invalid, and the requesting processor transitions the line to the Modified state.

3. Flush (Write-Back Request):

- **Description:** This transaction is used to write a dirty cache line (Modified state) back to the main memory. It is typically initiated when a cache line is being evicted or updated.
- **Actions:**
    - The cache with the Modified line writes the data to memory.
    - The line is marked as Invalid in the cache after the write-back.

4. PrRd (Processor Read):

- **Description:** This is a read operation initiated by the local processor. The processor requests to read data from a cache line.
- **Actions:**
    - If the cache line is in the Invalid state, the cache will initiate a BusRd request to fetch the data from memory or another cache.
    - If the line is in the Shared or Modified state, the cache can provide the data directly to the processor.

5. PrWr (Processor Write):

- **Description:** This is a write operation initiated by the local processor. The processor requests to modify data in a cache line.
- **Actions:**
    - If the cache line is in the Invalid state, the cache will initiate a BusRdX request to obtain the data and invalidate other caches.
    - If the line is in the Shared state, the cache will send a BusRdX request to invalidate other caches and transition to the Modified state.
    - If the line is already in the Modified state, the cache can update the data directly.

Example Scenario 1 : PrWr/BusRdX

Let's use an example with two processors, P1 and P2, and a memory location X.

Initial State:

- P1 has cache line X in the Shared state.
- P2 has cache line X in the Shared state.

P1 Performs PrWr:

1. **P1 Initiates PrWr:**
    - P1 wants to write to cache line X.
    - Since X is in the Shared state, P1 needs to get exclusive access.
2. **P1 Sends BusRdX:**
    - P1 sends a BusRdX request on the bus to obtain exclusive access to X.
    - All other caches with X in the Shared state (e.g., P2) must invalidate their copies.
3. **P2 Responds to BusRdX:**
    - P2 detects the BusRdX request and invalidates its copy of X.
    - If P2 had the line in the Modified state, it would flush the data to memory before invalidating.
4. **Memory or Cache Response:**
    - If X was not in the Modified state in any cache, the data is fetched from memory.
    - P1 receives the data for X and transitions the line to the Modified state.
5. **P1 Writes Data:**
    - P1 now has exclusive access to X and writes the new data to the cache line.

Example Scenario 2 : BusRdx/flush

Let's use an example with two processors, P1 and P2, and a memory location X.

Initial State:

- P1 has cache line X in the Shared state.
- P2 has cache line X in the Modified state (dirty).

P1 Initiates BusRdX:

1. **P1 Wants to Write:**

    - P1 decides to write to cache line X.
    - Since P1 does not have the line in the Modified state, it sends a BusRdX request on the bus.
2. **P2 Responds with Flush:**
    - P2 detects the BusRdX request and realizes it has line X in the Modified state.
    - P2 performs a flush operation, writing the dirty data of line X back to the main memory to ensure the data is up-to-date.
    - After the flush, P2 transitions line X to the Invalid state.
3. **Memory or Cache Response:**
    - P1 receives the data for line X from the main memory (or directly from P2 if the system is optimized for this).
    - P1 transitions its cache line X to the Modified state.
4. **P1 Writes Data:**
    - P1 now has exclusive access to line X and writes the new data to the cache line.

Example Scenario 3 : PdRd/BusRd

Let's use an example with two processors, P1 and P2, and a memory location X.

Initial State:

- P1 does not have cache line X (Invalid state).
- P2 has cache line X in the Shared state.

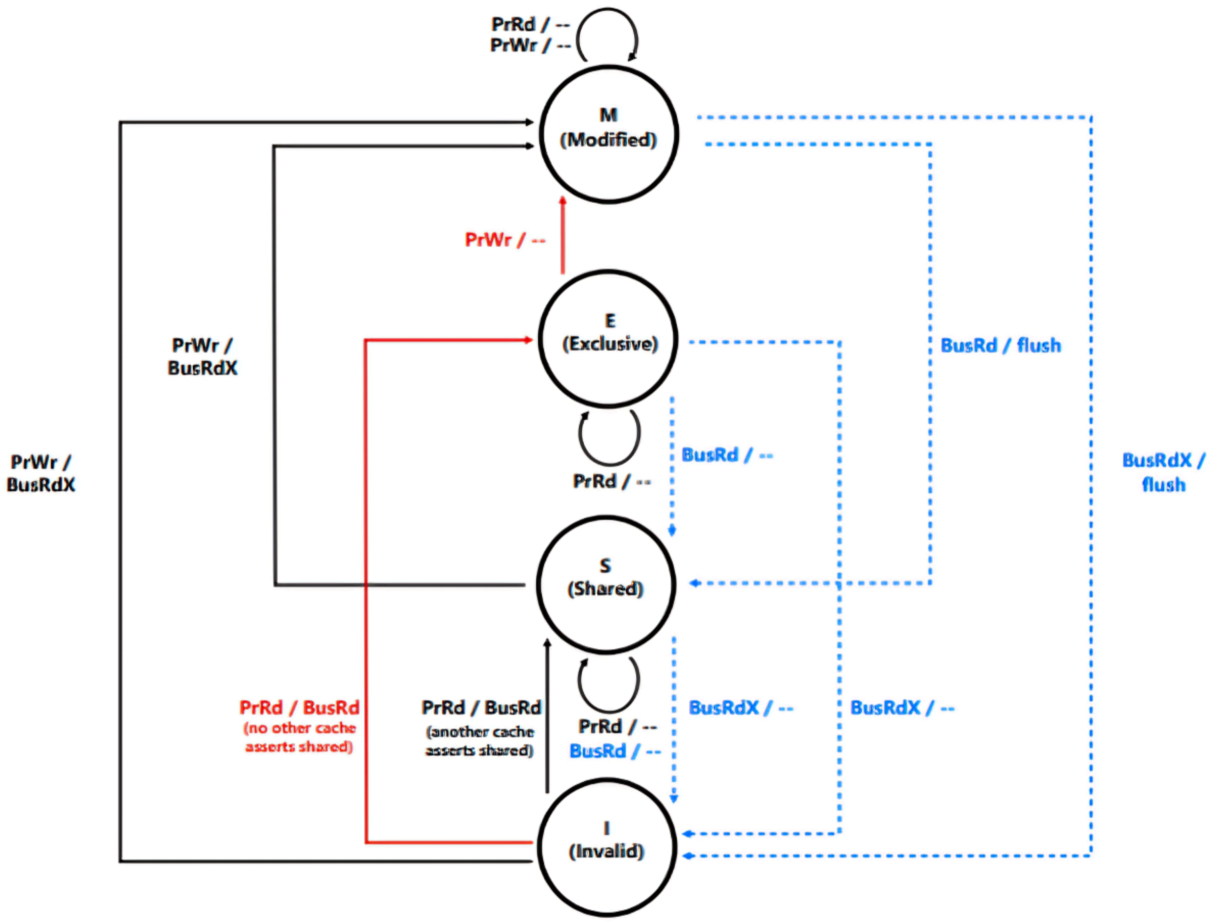
P1 Performs PrRd:

1. **P1 Initiates PrRd:**
    - P1 wants to read cache line X but finds it in the Invalid state.
2. **P1 Sends BusRd:**
    - P1 sends a BusRd request on the bus to obtain the data for line X.
3. **P2 Responds to BusRd:**
    - P2 detects the BusRd request and notices it has line X in the Shared state.
    - P2 responds to P1 with the data for line X.
4. **Memory Response (if applicable):**

- If P2 did not have the line in the Shared state, the main memory would respond with the data for line X.

5. **P1 Receives Data:**
    - P1 receives the data for line X and transitions its cache line to the Shared state.

8.2. MESI state transition diagram


PrRd / --
PrWr / --
M
(Modified)
PrWr / --
E
(Exclusive)
PrRd / --
S
(Shared)
PrRd / --
BusRd / --
I
(Invalid)
PrWr / BusRdX
PrWr / BusRdX
PrRd / BusRd (no other cache asserts shared)
PrRd / BusRd (another cache asserts shared)
BusRd / --
BusRd / flush
BusRdX / flush
BusRdX / --
BusRdX / --