Topics:
- Coherence (3 requirements)
- Write-Update Coherence
- Write-Invalidate Coherence
- Snooping
- Directory-based Caches
- MSI
- MOSI
- MOESI
- 4C's of Cache Misses
- True/False Sharing

Multi-Processing → Cache Coherence → Synchronization

# Cache Coherence

This lesson discusses the problems and solutions for coherence. Different coherence protocols are discussed, including: MSI, MOSI, MOESI, and Directory. Each has advantages and disadvantages depending upon the program being executed and the number of cores in the system.

Coherence is needed to ensure that when one core writes to its cache, other cores get to see it when they read it out of their own caches.

## Cache Coherence Problem

The programmer expects to see shared memory (when Core A writes "15" to mem location "x", we expect Core B to get "15" when it reads "x"). Since each core has its own cache, cache coherence can become a problem because each cache can have its own copy of the same memory location.

Incoherent - each cache copy behaves as an individual copy, instead of as the same memory location. Incoherent when the same memory location as seen from different cores has different values. This should NOT happen with Shared Memory.

## Coherence Definition

There are 3 requirements for coherence

1. If a core reads a memory location, the data is receives was written by the last valid write. (When only 1 core is operating on a memory location, it must behave like a proper uniprocessor - its latest read must return the value it last wrote to that memory location).

2. If a core writes to a memory location, when another core reads that same memory location, it should see the same value. Any core should be able to read the last valid write to a memory location.

3. All cores should agree on the order of the writes to a memory location.
   This doesn't say anything about how we interpret the order of writes, all it says is that all cores must agree on that order.

## How to Get Coherence

- Don't do caches. The main memory will be coherent : This leads to poor performance
-All cores share the same L1 cache. This leads to poor performance
-Use private write-through caches. This is not coherent. This doesn't work for cores that are only reading from their cache. If Core A and B are both using Mem Location "X", but Core A is writing to it and Core B is only reading it, even if Core A writes-through the update to memory, Core B will keep reading its stale cached value and not fetch the update from memory. We need something else.

To maintain coherence property 2: (Force Reads in one cache to see writes made in another cache)

Pick 1

-Broadcast all writes so other cores can update their caches. This is write-update coherence. "Write-Through" only writes the update back to main memory. "Write-Update" = "Write-Through" + updates that memory location in other caches as well.
-A write will make any other copies of the data invalid, so other cores will not be able to use old data. This is write-invalidate coherence.

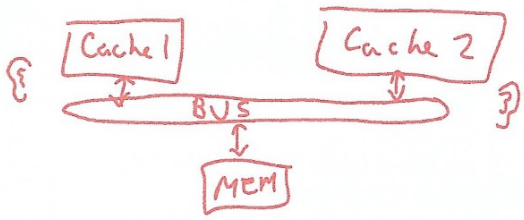
Pick 1 from each

To maintain coherence property 3: (Enforce that all the cores see the same order of writes on a memory location)

Pick 1

-Snooping: all writes are put on a shared bus and the cores snoop the bus to get the updated information for their caches. Pro: Serial access to the bus maintains ordering of Writes. Con: However, this can create a bottleneck if all the cores need to access this shared bus.
-Directory based: each block state is maintained by a directory. When a write occurs the directory reflects the state change.
For each cache block, all accesses are ordered by the same entity (a directory). Different blocks use different entities, so they don't have contention. This directory serves as an "ordering point" for a block, AND it also figures out which caches need to be updated OR invalidated when multiple caches have a version of the same block and an overwrite occurs.

With snooping, all caches are listening on the Bus for each other's "Write-Through" updates (committing a write from the cache to memory). So when Cache 1 writes an update to Mem on the Bus, Cache 2 hears it. The "Write-Update" part indicates the Cache's behavior AFTER it hears a Write on the Bus: it searches its own cache to see if it also has that data, and if so, then it updates itself with the value it heard on the bus.
The writes are also serialized and agreed upon because the bus only has serial access, so only one update/request at a time can be sent on the bus, so all caches will see the same Write order.




Property 2 Property 3

## Write-Update Snooping Coherence

Cores snoop to check for writes by other cores. When a write is detected, any other copies of the block are updated by the new data.

This is in the wrong spot

## Update Vs. Invalidate Coherence

If an application has a burst of write to one address -- invalidate is the better method.
If an application writes to different words in the same block -- invalidate is the better method.
If one core writes and another core reads often -- update is the better method

^ like a Producer-Consumer pattern where one Core produces (writes) the data while the other Core just consumes (reads) the data. If this was done with Write-Invalidate, then the Bus Access would be doubled: 1 access from the producer to invalidate, and another access from the consumer to fetch data.

All modern processors use the invalidate process, because it is better when a thread moves to another core.

Invalidate is better when data access stays local to one core. Even if access of that data transfers to another core (for example, with a thread), if only 1 core mostly accessing that data, then Write-Invalidate is better.

## Write Update Optimization

Avoiding memory writes:

So the bus isn't the bottleneck, it's how slow Main Memory is.

Only the last "Writer" to a memory location is responsible for keeping the Dirty Bit.

Writes need to be broadcast on the bus, so memory throughput becomes a bottleneck. To improve the bottleneck, the writes should be delayed to memory. To maintain coherence a dirty bit is added to the cache for each block. When the data is written by a core, all other caches are updated. Dirty data is updated in memory when the block is replaced in the cache.

Dirty Bit = 1 means that the Main Memory has not been updated with the latest cached value.
This means that the Caches need to Snoop READ requests on the Bus too! That way when one Cache (A) requests a value, the other Cache (B) that has that dirty block can respond because it knows that Main Memory doesn't have the latest value. Since Caches are faster than memory, the Cache will respond to the read request faster than Memory, and the other Cache (A) will take the first answer it hears.

Since another Cache that has that requested value will respond first (only if its Dirty Bit = 1, which means that it has the latest value for that MemLoc). The only time we read from Memory is if no other cache has that MemLoc where its Dirty Bit = 1.

Dirty Bit Benefits

-Writes to memory are greatly reduced.
-Reads from memory are also greatly reduced.

Write Update Optimization #2: Reducing Bus Writes (tracking which Cache Blocks are shared with a "Shared Bit" in each Cache Line).

## Write Invalidate Snooping Coherence

So the Shared Bit allows us to know if any other Cache ALSO has that block of data in their Cache. If so, then whenever that cache writes to that block, then it also needs to Write that update to the bus so that the other caches can pick up the latest values. However, if no other cache has that block, we have no need to write to the bus!

There is a shared bit in write-invalidate snooping.
A write causes other copies to be invalidated, this will cause a miss if a core wants to access the data. The newly written cache is the only valid copy and it will respond to any requests for the data. If a read is requested, the shared bit will be set to 1, showing there is more than copy of this data.
Disadvantage: there is a miss on all the readers when a core writes
Advantage: if a core needs to update the same block two or more times, the reads and writes can be done locally after the first write.

After a write is broadcasted on the bus, all the other Caches will hear it and invalidate their copies of that data (if they have them). That means that the original writer can correctly set it's "Shared Bit" for that Cache Block to 0 because it knows that all other Caches that had that block have invalidated it. It only gets set when another Cache has a cache miss (bc they previously invalidated that block) and requests for a copy.

Supposed to go here

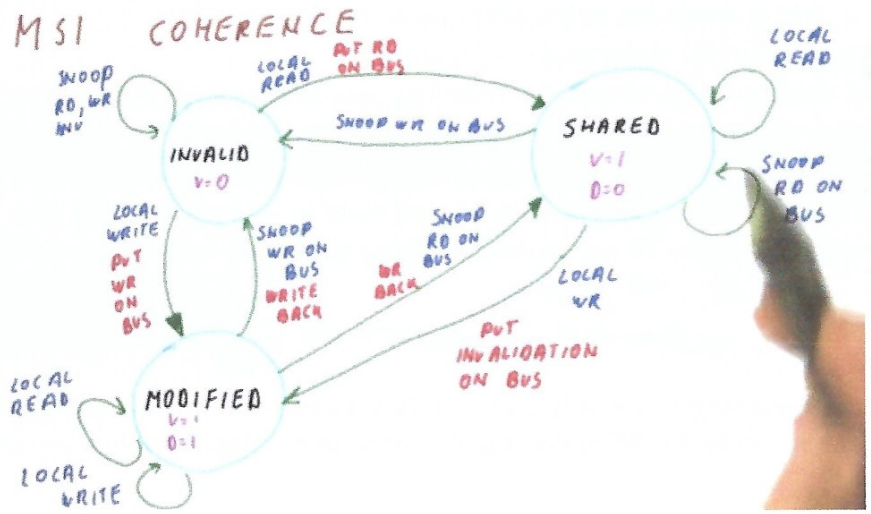
## MSI Coherence

This is an invalidation based protocol.

With MSI, a Cache Block can be in 1 of 3 states: (M)odified, (S)hared, & (I)nvalid.
Invalid - A block is in the invalid state either when it's in the cache but its Valid bit is not set (0), or if it's not in the cache at all. So a block that isn't in the Cache is simply treated like it is in the cache but with its Valid bit set to 0. So Valid = 0. Either way its a CACHE MISS.
Shared - A block that is in the Shared state can always be freely READ without any further ado, but if we WRITE we have to do something. A "Local Read" on a Shared block keeps it in the Shared state. Similar to Valid = 1 & Dirty = 0.
Modified - A block in the Modified state can also be freely READ without any further ado (like the Shared state, a "Local Read" will keep the block in the Modified state). However, a "Local WRITE" will also keep the block in the Modified State. With a block in the Modified state, we are sure that there are NO OTHER CACHED COPIES, which is why we're allowed to locally write without having to inform the other caches about the updates. This state is similar to having Valid = 1 & Dirty = 1. For a given block, only 1 Cache may have it in the Modified State. All the other Caches that have that block must have it in the Invalid State (cannot have it in the Shared state while 1 block has it in the modified state).

- From the Invalid State, a Local Write takes it to the Modified State by putting the Write Request on the Bus, which INVALIDATES the other caches' copies of this block, which ensures that nobody else has a valid copy of the data.
- That means that when a Cache has a block in the Modified state and it hears (or SNOOPs) another Cache's Write Request on the bus, then it will transition the state from Modified to Invalid AND it will WRITE BACK its data (since before we heard the Write Request, this Modified block had the latest copy of the data).

- If we're in the Modified State and SNOOP a Read Request on the bus, then we move to the Shared State AND we will also WRITE BACK its data (like the previous point).
- From the Invalid State, a Local Read puts a Read Request on the Bus, and then when it gets the data it transitions to the Shared State.
- That means that if we're in the Shared State and SNOOP a Write Request on the Bus, then we move the block to the Invalid state (no need to write the block back, since in the Shared State, Dirty = 0).
- From the Shared State, a Local Write will first put an Invalidation on the Bus (another Cache that has it as Modified will Write Back its copy and then transition to Invalid) and then transition itself to Modified.




If any Cache has a block in the Modified state, then all the other Caches MUST consider that block to be in the Invalid state.

If a Cache has a block in the Shared state, then all the other Caches can either have that same block in the Invalid of Shared states.

So we can have:
- 1 cache has a block as M, all others has it as I.
- 1 or more caches has a block as S or I, but none of them has it as M.

For the same block, we CANNOT have caches see it as Modified AND Shared at the same time. And only one Modified state can exist for a block (no other caches can see it as Modified as well).

**Cache to Cache Transfers**

Cache to cache transfers occur when a cache (C1) owns the data (the block is in the 'M' state) and a read request for the data (C2) is detected on the bus. C1 must supply the data because it has the only valid copy of the data.

Possible methods to do this:

1. Abort and retry

   C1 somehow cancels C2's request ("try again later"), C1 does a normal Write-Back of its data to Mem, C2 then retries Read from Mem

   Downside of this approach - there needs to be two memory latencies to get the data to the requester. 1 memory latency to write the data back to memory, another memory latency to then fetch that data from memory.

2. Intervention

   The core that owns the data intervenes and tells the memory it will respond. An intervention signal must be added to the bus.

   Disadvantage of this method: hardware is more complex

Modern processors use the Intervention method.

**Avoiding Memory Writes on Cache to Cache Transfers**

When using the Intervention method, the memory needs to be written when there is a read of modified data.

It would be better if the memory was only written when the block is kicked out the cache -- the Owner would be a new block state that is responsible for responding to read requests and updating memory.

So when a Cache has a block in the Modified state, its easy to tell that it is the owner and has the latest data. So that when other Caches request that block, it's clear that the Cache that has the block in the Modified state should respond with the latest up to date data.

**MOSI Coherence**

But what if multiple caches have the block in the Shared state? Who is the owner? The multiple caches shouldn't all respond when another read request is Snooped on the bus, so the request would have to depend on SLOW MEMORY to respond with the data. Instead, we should add another "Owner" bit.

The 'O' state is like the 'S' state except:

1. when a read is detected - the owner responds
2. write-back to memory when the block is replaced

With "MOSI", when a block is in the Modified state and it SNOOPs a Read Request, then it will move to the "O" (Owner) state instead of the "S" (Shared) state.
- When we're providing the data, memory does not get accessed! With MSI, moving from the M to S state because of a SNOOPed Read Request would require a Write-Back to Mem. However, with MOSI, we do not need to do that.

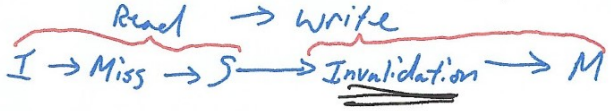
In summary, we used the "O" state to avoid inefficiency of the MSI protocol that had to do with the memory getting unnecessary accesses that could be satisfied by caches that already have the data.

M = a core has modified the data and has the only valid copy of the data
S = at least one core has the block in its cache and it is clean
O = a core has modified the data, and has shared the modified data with at least one other core

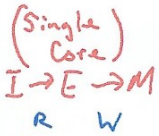
## M(O)SI Inefficiency

Both MOSI and MSI have this inefficiency. This occurs when dealing with "Thread-Private Data", or data this is only ever access by a single thread (so at any given time, only one core will be accessing that data). Ex: single-threaded programs, or Thread Stacks in multi-threaded programs. The inefficiency occurs when we do a Read and then a Write to Thread-Private Data.

There is still inefficiencies in MOSI. When going from a Shared state to a Modified state, the block must pass through the invalid state. To eliminate this step a new state is introduced, the Exclusive state.




For Thread-Private data (where the data should only ever used in 1 cache), this Invalidation step is useless work. We're taking up Bus time to send out the Invalidation that isn't actually needed by any of the Caches.




## The E State

The exclusive state is used when a core is the only core that has a clean copy of the data. When a block is in the 'E' state it can move to the 'M' state directly because no other core has a copy of the data.

M: Exclusive Access (Read & Writes Allowed), Dirty - One Cache has accessed this data through a write access. All other copies of the data are Invalid.
S: Shared Access (Only Reads Allowed), Clean - More than one cache has this data through Read Accesses.
O: Shared Access (Only Reads Allowed), Dirty - This cache has previously modified the data. It will respond to requests for this data from other caches.
E: Exclusive Access (Read & Writes Allowed), Clean - Only one cache has this data, it was Read Accessed.

Remember, "Dirty" means that the cache has the latest version of that data and is RESPONSIBLE for (1) responding to other Cache's read requests & (2) writing back the block to memory if that Cache line gets replaced.

## Directory Based Coherence

Snooping downside: every request must be broadcast, which means there must be one bus. This leads to a bottleneck and snooping ~~can be scalded to more than 16 processors~~.
does not work well with > 8 ~ 16 cores.

To eliminate the need to broadcast, while still observing the coherence requirements, a directory can be used.

Coherence requirements:
- Caches see the requests that they need to see
- Requests to the same block need to be ordered and unanimously agreed upon.

## Directory

A Directory allows us to use a non-broadcast network and go outside of the restrictive bottleneck of having only 1 Shared Bus.

A directory is:

Distributed across all cores, each core has its own 'slice'

Each "Slice" serves a set of Blocks. Within a Slice, there's one Entry for each Block it serves, where each Entry tracks which Caches have their respective Blocks (in a Non-Invalid state).
Order of accesses are determined by the "Home" Slice - the Slice of the Directory that has the entry for this block.

Each slice serves a set of blocks.

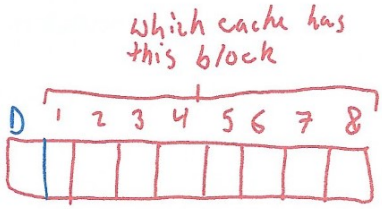
The directory keeps track of which caches have the block, for valid states only.

## The Directory Entry

The directory entry has:

With Directories, there's no SHARED BUS INVOLVED, or SNOOPING. Instead, it's direct Point-to-Point Communication.

1 Dirty bit (indicates that the Block is Dirty within SOME cache in the system)
1 bit/Cache Present/Not Present (there's 1 bit for EACH CACHE in the system)




0 = block is not present in a valid state in the cache

For example:

for an 8 core system there are 8 bits for signifying present/not present

The directory communication requires an acknowledgement from the cores after a request.

Unlike SNOOPing, instead of sending all requests to the single bus, with a Directory system, caches can send requests for a certain Block DIRECTLY to that Block's HOME SLICE (aka to the Cache that has the Slice that has the Entry that corresponds to that Block - we determine the Home Slice from the Address).

## Cache Misses with Coherence

The three 'C's' are now four:

Compulsory, Conflict, Capacity
Coherence Miss - a miss caused by coherence.

So the requests are sent to the Directory Slice, and then the Directory Slice is the one that makes the memory request (if necessary) - either to another Cache, or to Memory. When it returns the block to the requesting Cache, it also informs the Cache of the STATE that it should associate that Block to (for example, "Exclusive" if the Directory fetched that block from Memory bc no other Cache had it). The Directory Slice also updates the Entry's Present/Not Present bit for the Requesting Cache as necessary (ex: if cache 1 requested that block, then the Directory would change the Presence Bit for Cache 1 to "1").

With a Directory, when it gives a Cache "exclusive" access to a block of data, it'll go ahead and mark the Directory Entry for that block as "Dirty", since the Cache should be able to easily modify that block (local to itself) without notifying anybody else.

When a Write Message comes to a Directory, it'll find out all the other caches that have that data and tell them to invalidate (and write-back if necessary).

So Directory saves a lot of bandwidth because instead of all the Caches broadcasting everything, the Directory system only has point-to-point communication (only involve the Caches that need to be involved).

For example: A core (C1) reads a memory location, then another core (C2) writes to that location. When C1 attempts to read the memory location again, the data is invalid. This is a miss due to coherence.

Two types of coherence misses:

True Sharing- different cores access the same data

False Sharing- different cores are accessing different memory locations, but these memory locations are in the same block. From the standpoint of coherence, data in the same block are the same data.