CS 61C — Summer 2022

Cache Coherency

Discussion 11

1 Precheck

1.1 Each hardware thread in the CPU uses a shared cache.

False, each thread has its own cache, which can lead to cache-incoherence.

1.2 The dirty bit signifies an address line in cache that has up-to-date memory, but main memory at this address is out-of-date.

True, the dirty bit is set to 1 when the cache has been written to with new data, commonly known for it's use in write-back caches. This tells the computer that main memory is out-of-date with respect to this specific address.

1.3 Given a multi-level cache setup that has n bytes worth of storage, we're always able to utilize all n bytes for distinct data.

False, this can only begin to be true in caches with an exclusive policy (between inclusive and exclusive caches) as inclusive multi-level caches *copy* lower level data that are hits into higher level caches. Exclusive multi-level caches *move* that data instead.

1.4 Cache state information, like cache line tags, are stored independently from the data cache.

False, usually, the valid, dirty, and shared bits are stored right with the data cache line itself because it makes comparison logically easier! This is acceptable because for the more complex cache coherency protocols, such as MOESI, typically we only need up to 3 additional bits outside of data to determine state, whereas for tag bits, there may be may more.

2 Writes

2.1 When it comes to writing data to cache memory, there are multiple write policies to consider that offer different options when building our system. Some of them you might encounter are:

1. **Write-through**: Write through: In this policy, when we have a write we write to both the cache and the memory. This is the case for every write, so the main memory always has the updated data. This is simple to implement, but writing to main memory every single time is slow.
2. **Write-back**: On a write, the data is only updated/written in the cache. The main memory only receives the data upon eviction. This means the cache has more up to date data most of the time. While this is faster as there is less accesses to main memory, it is harder to implement as we have to include more overhead, such as dirty bits and so on.

3. **Write-around**: Data is only written to main memory, and whenever we do so we invalidate the old data in the cache.

Another thing to consider is what we do when we have a write miss. For that, we have 2 possible policies:

1. **Write-allocate**: On a write miss, we pull the block you missed on into the cache
2. **No write-allocate**: On a write miss, you do not pull the block you missed on into the cache. Only memory is updated. On a read miss, we still pull the data into the cache.

Considering the above information, lets consider a direct mapped, no write-allocate write-through cache with a capacity of 8B and a block size of 4B. Lets also assume that the memory addresses are 8 bits each. Assuming the cache is completely empty in the beginning, we make memory accesses to the following locations:

- 0x6A, Write
- 0x85, Read
- 0x6B, Read
- 0x87, Read
- 0x68, Write

With the above memory access pattern and the given cache configuration, how many times do we access the main memory?

Short Answer: 4
Long Answer:

- The first cache access to the location 0x6A is a miss, as the cache is initially empty. As the cache is no write allocate, on this cache miss we just write to the main memory only, so this is our first main memory access, and the cache is still empty.
- Then on the second cache access, we have a read miss - for this one we go to main memory, and actually bring the line in the cache, which occupies the cache line with index 1. 2nd main memory access.
- Third one is again a read miss, so the same happens and the line with the index 0 gets filled out with the memory address this time. Third main memory access.
- This one is actually a cache hit - tag is 0b1 0000 (which was put in the cache in step 2) so no main memory access, read hit.
- This one is a write hit, but because our cache is write through we actually write in the main memory as well, so 4th main memory access.

2.2 Lets say for the same cache size, memory accesses but the only difference is that we have a no write-allocate write-back cache instead. How many memory accesses to

the main memory do we have in this case?

Short Answer: 3
Long Answer:

- The first cache access it the same as above, 1st main memory access.
- Again, same as above, 2nd main memory access.
- Same as above, 3rd main memory access.
- Same as above, read hit (no main memory access)
- This one is a write hit, but this time as we have a write back cache, we do not go to main memory - we write the new data on the cache, and turn the dirty bit on the relevant line to 1.

2.3 For one last optimization, we decide to use a write-allocate cache instead. So, now we have a write-allocate, write-back cache. How many times do we access the memory now?

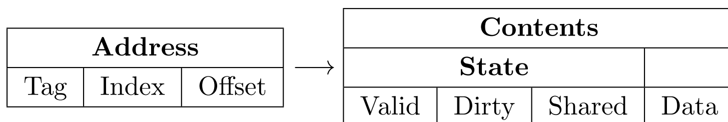
Short Answer: 2
Long Answer:

- This is again a write miss, but because our cache is write-allocate now, we actually bring the data in from the main memory into the cache, and the line with index 0 gets filled. 1st main memory access.
- This one is the same as the above 2 examples, so cache line with index 1 gets filled and we have a read miss. 2nd main memory access.
- Because the first write actually filled the 0th index line with the relevant data, this 3rd memory access actually becomes a read hit, as there is data on the cache now. No main memory access!
- Same as above, read hit (no main memory access).
- Same as 2.2, as the cache is write-back, this write hit is taken on the cache itself, and the line on the cache gets changed - no main memory access.

3 States

3.1 Parallel processing allows individual cores of a CPU to operate as independent units with their own caches. However, for this to be the case, the machine must be able to coordinate the information flow of all cores and all caches so that this information is reliable to some degree. Therefore, we impose **cache states**, composing of the **valid**, **dirty** and **shared** bits, to denote status of the cache data at a specific cache block. These cache states are used when there is a cache **miss** or **write** to a certain core's cache so that if the information is modified in one place, the other caches are informed. In summary, we don't want two caches with different data both saying that they have the most up-to-date data, because that simply can't be true. In other words, from the perspective of the **host processor**, their cache line states may be update due to actions taken by **proxy processor** execution.

Consider this visual representation of the addressing of a cache block and the updated construction of the block itself:



Each state describes a specific set of conditions, on a **single cache block**, in respect to the overall memory system(all caches and main memory). These conditions are listed below, your job is to pair all appropriate conditions with their corresponding state.

Note: these conditions can apply to multiple states, therefore pick all that apply.

(a) data in host cache up-to-date

(b) data in main memory out-of-date

(c) data in main memory up-to-date

(d) dirty bit = 1 in host cache's line copy

(e) no copies exist in other (proxy) caches

(f) copies may exist in other (proxy) caches

(g) write will not change cache line state in host processor

(h) access from processor will result in a miss

1. **Modified**(M)

   a, b, d, e, g

2. **Owned**(O)

   a, b, d, f, g

3. **Exclusive**(E)

   a, c, e

4. **Shared**(S)

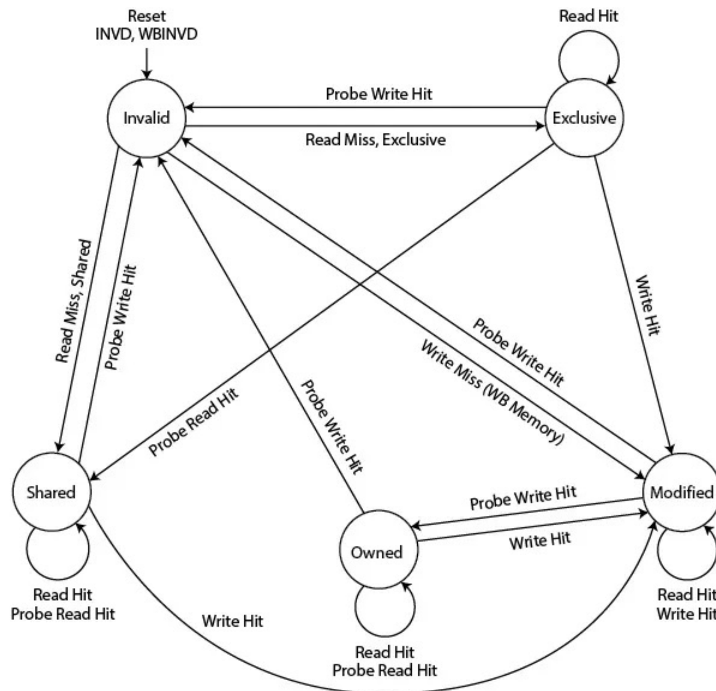
a, f

5. **Invalid**(I)

   h

4 Coherency

To be able to utilize multiple threads simultaneously, the hardware of your machine must be able to keep multiple caches in check so that there is no conflicting data. The diagram below represents the transitioning between states based on the reads and writes to that particular block of data.

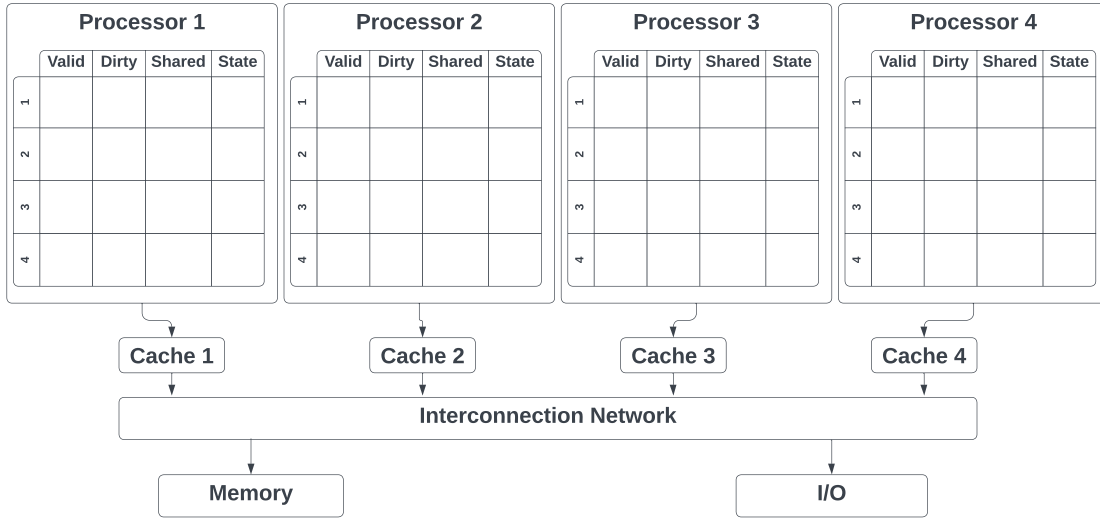


Note: A "probe" read or write indicates an action taken by a proxy processor; i.e. the action is taken by a processor other than the one we're examining.

In the following problems you will assume that the system is implemented with a **write-back** policy and you will be given a specific coherency protocol. Use the provided tables to fill out the processor states for each cache write or cache read.

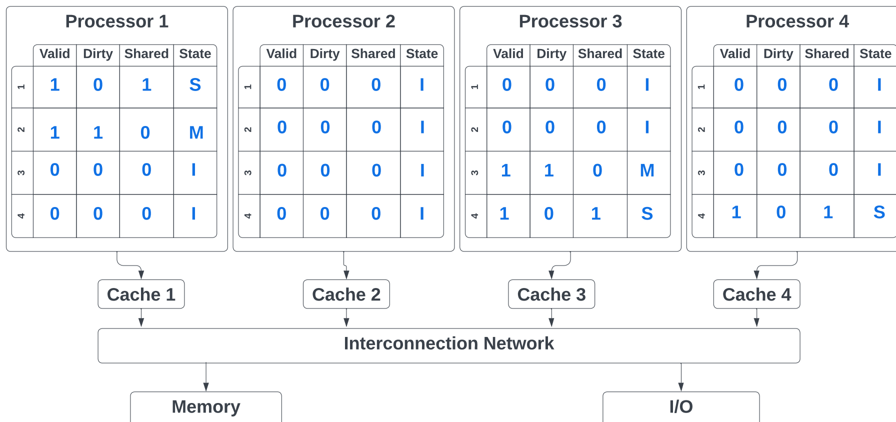
1. Memory Address 0xDEADBEEF in Processor 1 is read. Hit or miss?
2. Memory Address 0xDEADBEEF in Processor 1 is written to with integer 10.
3. Memory Address 0xDEADBEEF in Processor 3 is written to with integer 50.
4. Memory Address 0xDEADBEEF in Processor 4 is read. Hit or miss?

4.1 Fill in the following table with the corresponding processor states according to the **MSI** protocol for each step below.

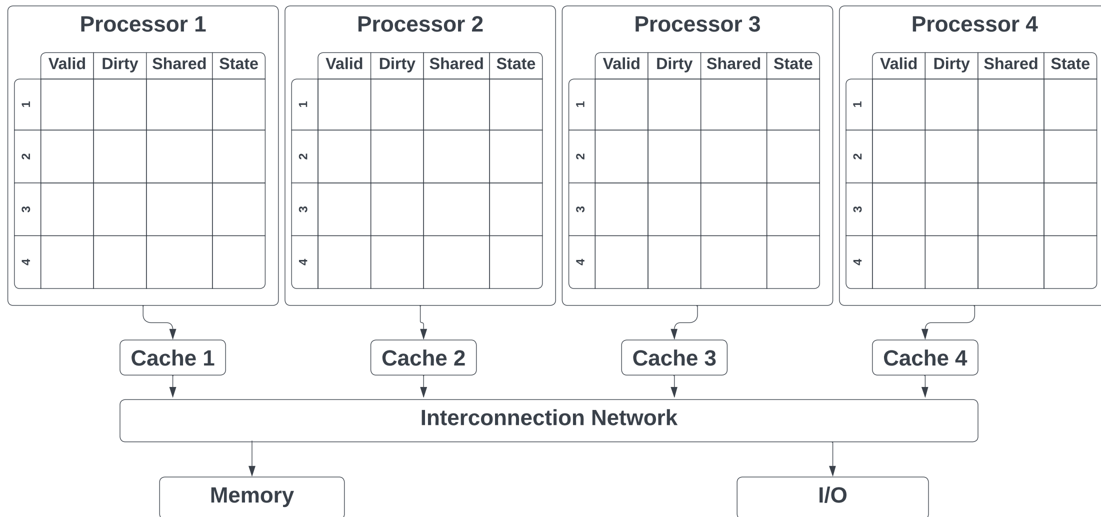


First, all cold caches begin **invalid**.

1. A read is attempted on Processor 1, which results in a **miss** and loads the data from main memory into the cache. Therefore, Processor 1 is now in a **shared** state.
2. A write is attempted on Processor 1, which results in a write **hit**. Therefore, Processor 1 has updated information and is now in a **modified** state. Note that since we have a write-back policy here, main memory is now out-dated(dirty bit=1).
3. A write is attempted on Processor 3, which results in a write **miss**. Processor 1 is then **invalidated**, and Processor 3 is written to with updated information. Therefore, Processor 3 is now in a **modified** state.
4. A read is attempted on Processor 4, which results in a read **miss**. Processor 3 is then directed to load dirty data to main memory(due to the write-back policy), which then allows Processor 4 to load this data into it's cache. Therefore, Processor 4 and Processor 3 are now in a **shared** state.

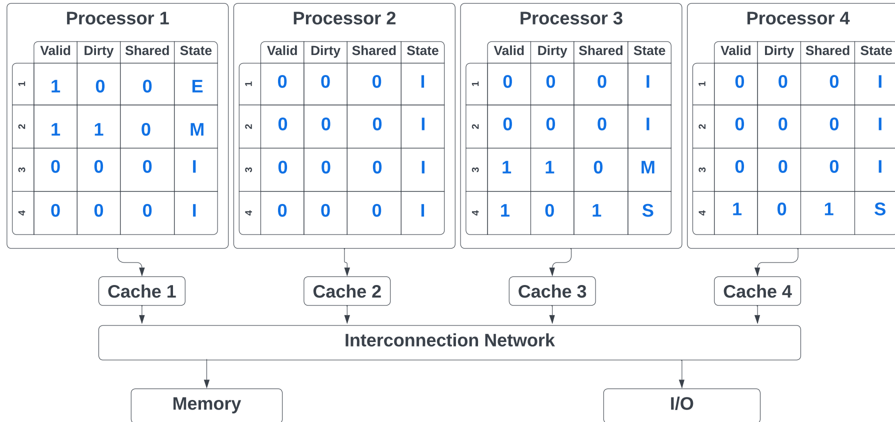


4.2 We now modify the MSI protocol by adding an **Exclusive(E)** state that represents the state where a cache is the **only** cache that has seen this data. Fill in the following table with the corresponding processor states according to the **MESI** protocol for each step below.

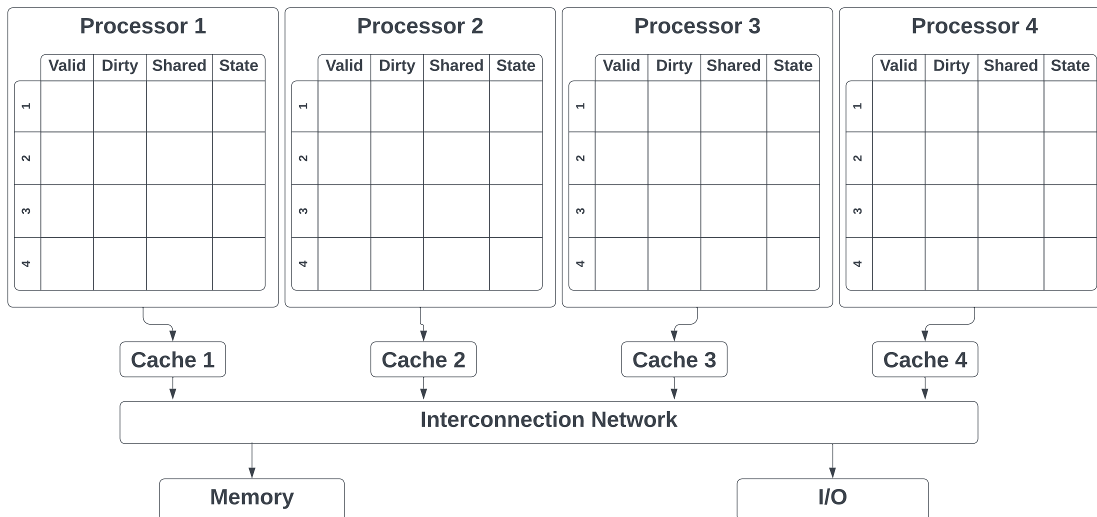


First, all cold caches begin **invalid**.

1. A read is attempted on Processor 1, which results in a **miss** and loads the data from main memory into the cache. Therefore, Processor 1 is now in an **exclusive** state because it's the only cache that has seen this data(exclusive is only possible for a read accessing a new address).
2. A write is attempted on Processor 1, which results in a write **hit**. Therefore, Processor 1 has updated information and is now in a **modified** state.
3. A write is attempted on Processor 3, which results in a write **miss**. Processor 1 is then **invalidated**, and Processor 3 is written to with updated information. Therefore, Processor 3 is now in a **modified** state.
4. A read is attempted on Processor 4, which results in a read **miss**. Processor 3 is then directed to load dirty data to main memory(due to the write-back policy), which then allows Processor 4 to load this data into it's cache. Therefore, Processor 4 and Processor 3 are now in a **shared** state.

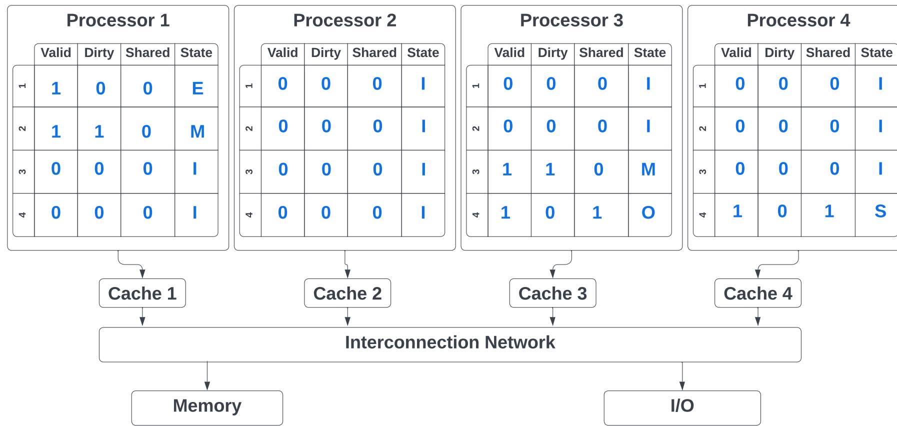


4.3 We will now introduce a fifth state, the **Owned(O)** state, that represents the instance where a cache has exclusive ownership of a cache line and other caches can read from it. Fill in the following table with the corresponding processor states according to the **MOESI** protocol for each step below.



First, all cold caches begin **invalid**.

1. A read is attempted on Processor 1, which results in a **miss** and loads the data from main memory into the cache. Therefore, Processor 1 is now in an **exclusive** state because it's the only cache that has seen this data(exclusive is only possible for a read accessing a new address).
2. A write is attempted on Processor 1, which results in a write **hit**. Therefore, Processor 1 has updated information and is now in a **modified** state.
3. A write is attempted on Processor 3, which results in a write **miss**. Processor 1 is then **invalidated**, and Processor 3 is written to with updated information. Therefore, Processor 3 is now in a **modified** state.
4. A read is attempted on Processor 4, which results in a read **miss**. Since Processor 3 has the line in a modified state, it flushes (sends) the updated data onto the interconnection network, which then allows Processor 4 to load this data into it's cache. Processor 3's copy is now in an **owned** state and Processor 4's copy is now in a **shared** state since the host processor's copy is modified but allows exclusive read-only access from proxy processors.


Processor 1
Valid Dirty Shared State
1 1 0 0 E
2 1 1 0 M
3 0 0 0 I
4 0 0 0 I
Processor 2
Valid Dirty Shared State
1 0 0 0 I
2 0 0 0 I
3 0 0 0 I
4 0 0 0 I
Processor 3
Valid Dirty Shared State
1 0 0 0 I
2 0 0 0 I
3 1 1 0 M
4 1 0 1 O
Processor 4
Valid Dirty Shared State
1 0 0 0 I
2 0 0 0 I
3 0 0 0 I
4 1 0 1 S
Cache 1
Cache 2
Cache 3
Cache 4
Interconnection Network
Memory
I/O