# Cache Coherence

Programmer want shared memory behavior; however, for hardware, each core has its own cache. For different cores, the data in memory is not the same -> this system is incoherent

Coherence Requirement

* Read R on core C1 will return the value written by the most recent written W by C1, if no other core has written to that value between R and W.
* If C1 write to X and C2 read after a sufficient time with no other writes in-between, C2’s read returns the value from C1’s writes.
* If more than one writes, writes are serialized -> any two writes to same value must be seen to occur in the same order on all cores. If C3 sees C1 comes before C2, all other core must also see C1 comes before C2.

How to get coherence?

* All cores share same L1 cache (bad performance)
* Force read in one cache to see write in another
  + Broadcast writes to update other updates (write-update coherence)
  + Writes prevent hits to other copies (write-invalidate coherence) – for requirement 2
  + Writes broadcast on shared bus, same order seen by everyone (snooping)
  + Each block assigned an ordering point (directory) - for requirement 3

## Write-Update Snooping

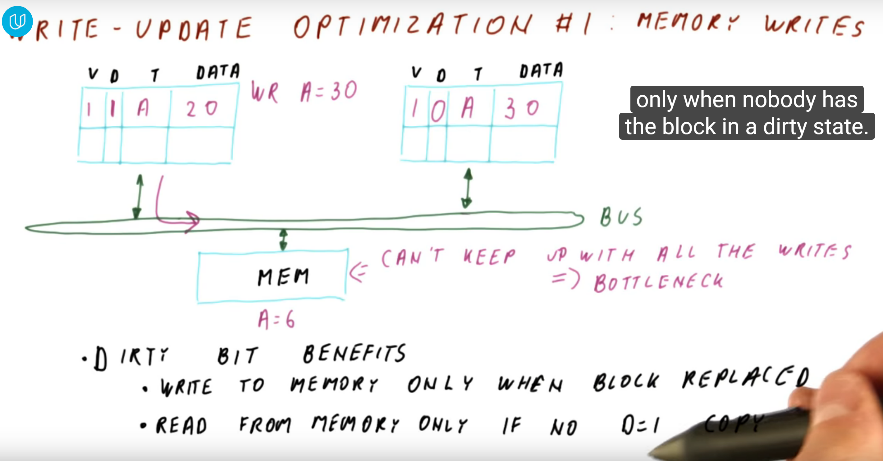
Connect to bus, caches are monitoring the bus. When a processor write value to X, sending write to bus, memory write value, and other cache see the update in X and update value of X. Bus is single broadcast bus, only one write happening at one time.

Optimization:

1. Avoid update memory when write -> add dirty bits to delay memory write (inconsistent with memory), write issue will be responded by newly written cache. If new writer updates the value, clear the dirty bit and change the new-write block dirty bit to 1. Only write to memory when the block is replaced.

图片包含 文字

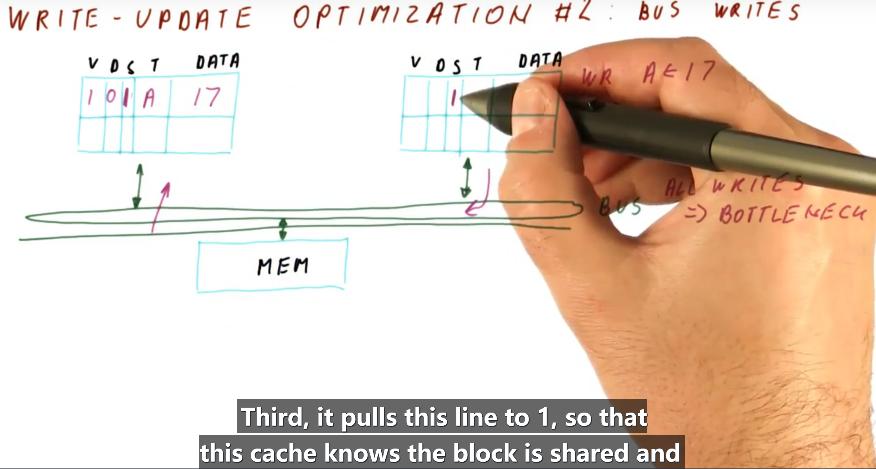
描述已自动生成



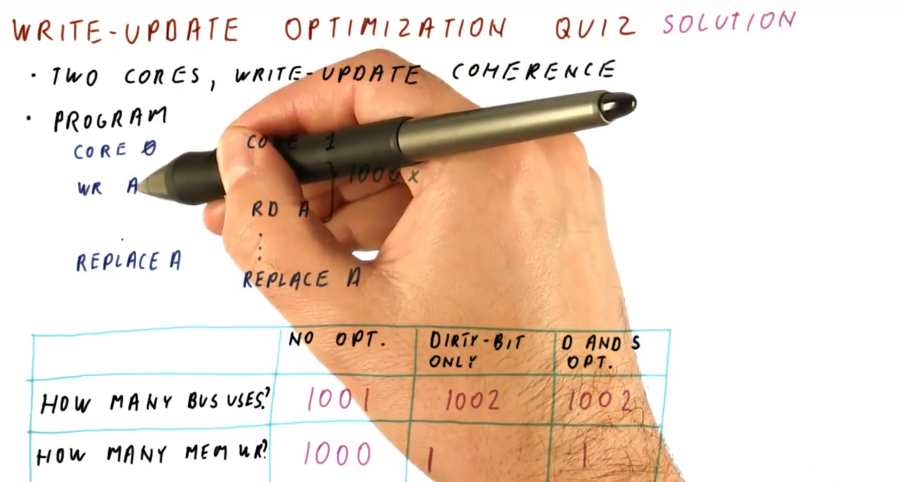
1. Avoid all writes going through bus -> add share bit, if block not sharing, no need to use bus to update others -> add a shared line in bus, if shared, pull the shared line to 1 and update the shared bit in blocks

图片包含 人员

描述已自动生成



With first write, need to use bus to bring data from memory and thus use the bus, at the same time, behavior as broadcasting. Each operation max using 1 bus. With no optimization, need to write to memory when write and read only use 1 bus to access data and every write will broadcast its result -> 1000 write to memory, 1000 + 1 bus used; With dirty bit, only need to write to memory when replace, write still need to be broadcast and when replace, result will be written to memory through bus -> 1 write to memory, 1000 + 1 (first read) + 1 (write to memory) bus used; With dirty bit and share bit, for first write no need to broadcast.

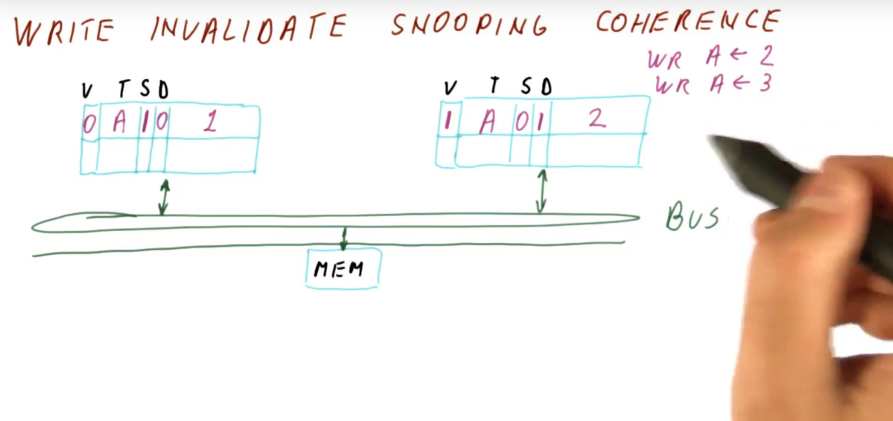


## Write Invalidate Snooping

When write, change the valid bit. Every time do write change the share bit, if other block will become invalid and no sharing. Advantage: if need to update same block twice, no need to broadcast to global. Disadvantage: write will result cache miss. (local performance is good)

图片包含 人员, 掌握, 文字

描述已自动生成



For write update: more hit but will require more broadcast to update value (more efficient when data is sharing between different cores in a producer-consumer behavior)

For write invalidate: less hit and enforce to get new version from recently write block (more efficient when data is only accessed locally)

图片包含 人员, 掌握

描述已自动生成 write invalidate only as 3

Update v.s. Invalidate

图片包含 文字

描述已自动生成

Modern operating system tends to move thread from one core to other. Update will keep updating thread data in original core, leading bad traffic.

## MSI coherence (invalidation-based)

* I -> Invalid state (valid bit not set / no in cache, v = 0)
* S -> Shared state (read without op, if write, need to do something)
* M -> Modified state (read block as S, write block by local core), ensure no other sharing in modified state (v = 1, d = 1)

图片包含 文字

描述已自动生成

Also, we assume: if the data is in modified state, if other core write to it, we should write back the data to memory and then other core read it from memory (simpler), instead of directly feeding to the other core.

The reason that you need to write to memory when transitioning from M->S is because if you don't, that means a cache would need to do a write-back to memory in S state. However, note that multiple cores can be in S state (in this example, C1 and C2), thus that would result in whenever a core leaves S state (the cache line is evicted), it will need to write to memory, and that means you end up writing to memory multiple times. (Also, block in share state with dirty = 0, thus memory must catch the recently dirty data to keep consistency with cache state)

Cache to Cache Transfer

1. Abort and retry (write back and retry read)
2. Intervention (other provide data and memory need to pick up data: once in share state, both blocks are clean, thus memory need to pick up the dirty data to keep consistency)

图片包含 文字

描述已自动生成

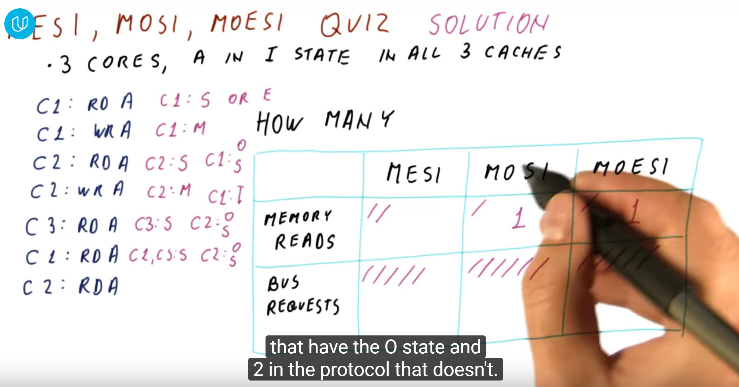
Memory bandwidth is much smaller than cache, when block in share state, memory will provide data -> introduce another state O to provide sharing data and O write back to memory

State M -> when detect Read -> State O, the block providing data for other read blocks

图片包含 人员

描述已自动生成

Assume use intervention: when in modified state, data is supplied by modified state block.

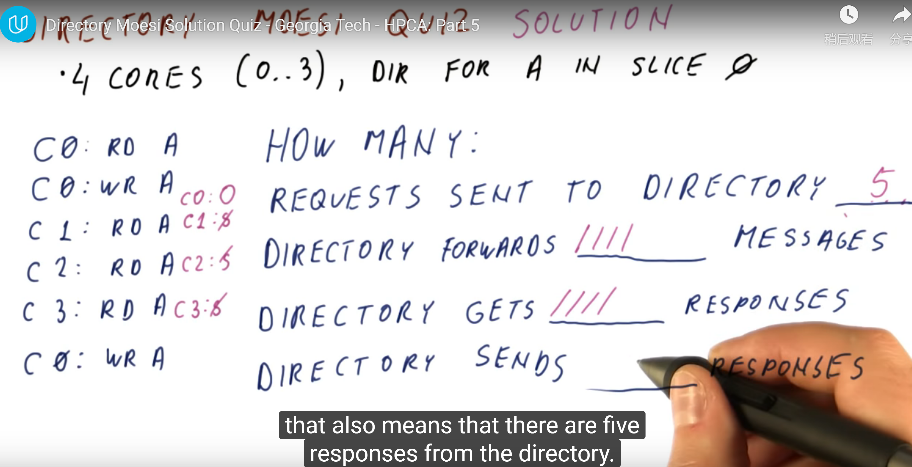


## Directory-based Coherence

Bus will become bottleneck when having lots of cores. -> directory is a distributed structure across cores. Directory has one entry for each block it serves and track which caches have block (not in I state). For different cores, if access different blocks, they can get information from different directory, no longer in one single bus but a network. Directory will forward data from modified block or e state block to the read request block. Since E state can easily change to M states without notifying others, the directory will set the dirty bit as 1 for block in E state. Also, no matter the block is in O state, directory have no idea about who to provide the data, due to the dirty bit as 0, thus, new read request will require memory.

图片包含 文字

描述已自动生成

 will need to forward multiple

In a bus-based MOESI protocol, now C0 would intervene and supply data to all others who ask for it. But with the directory that was described in the previous video, we don't know that the block is in the O state - the directory sees the O state as equivalent to the S state.

Coherence Miss: read after write, will not be miss -> if not consider coherence, it would not be missed.

1. True sharing: different core access same data (need to perform coherence)
2. False sharing: different core access different data (but in same block) (not need to perform coherence, but we still label the block as dirty, leading coherence miss)

图片包含 文字

描述已自动生成

For second choice, all is compulsory miss, since block is not in cache. Even there is no coherence concern, these blocks will still miss.