perform_access_store

In the method *perform_action_store* the cache attempts to modify a value saved in the cache identifiable by its tag. The tag is determined by the address of the value in memory, the size of the block, and how many words fit inside that block. Given the tag, the program searches through the given cache using the *find_line* function to attempt to identify the exact line in the cache with that specific tag as ordered by LRU. Should the line be found and is not invalid, then the block is brought to the front of the LRU cache. If the state of the identified block is exclusive or modified then all other caches either don't have that block, or the block is already invalidated and thus doesn't need to be labeled as invalid to signify that they are "dirty". However, no matter what the state originally was, the state must become Modified.

Should there be instead, a write miss, the cache sends a request on the bus to all other caches searching for a clean version of that line using *write_search*. This method also sends out the signal to invalidate the block in other caches, so no separate invalidate signal needs to be sent on the bus. Then the new block is added to the cache with the state Modified using *add_line* method.

perform_access_load

In the method *perform_action_load*, the cache attempts to read a value saved in cache, identifiable by its tag. It searches for the block with the matching tag in its cacheto see if it is already saved in the cache using the method *find_line*. Given that it found the line and the line was not invalid, a Read Hit occurs which does not need to change the state of the block, but does bring it to the front of the cache, using *move_to_*front as it was the most recently used.

Should there be instead be a Read Miss, the cache broadcasts a request for the block along the bus to the other caches using *read_search*. If another cache has the requested block, then saving it in this cache would mean that there are multiple clean copies of this block, making its state Shared, otherwise it is Exclusive. The method then uses *add_line* to insert the new block with the appropriate state into the cache.

invalidate

Given a Write Hit, the corresponding block in the cache is Modified to be the most up-to-date version and thus the only "clean" version of the block, and all others must be identified as "dirty" or invalid. The cache which received the Write Hit must broadcast that to all other caches as a signal to invalidate their blocks. To simulate this, the *invalidate* method goes through each cache except the origin and uses *find_line* to both see if the block exists in the other caches, and to gain access to the block. Should the found block have the state of Modified or Owned then that block was previously the one with the most up-to-date value and thus must be copied back into memory to keep everything consistent. Then the line that is found has its state set to invalid.

add_line

Whether a Write Miss occurs or a Read Miss occurs, a new line must be added into the cache as a new block is created and saved. If the block already exists, but is invalid, this simply updates the block to the appropriate state and brings it to the front of the cache using *move_to_front*. Otherwise, it allocates space for the new block and saves the relevant data. Should the cache be full, the cache must *evict* the block at the tail to make room for the new block. With enough space in the cache, it inserts the block into the head of the cache.

find_line

This method searches to a line with a specific tag in a given cache. It loops through each member of the cache, starting at the head until it finds the line. Should the line be found, the function returns a reference to the line, otherwise it returns null.

evict

Evicting is typically done with there is no more room in the cache for new entries. It first identifies whether the value at the tail is the most up-to-date value of that block in all of the caches and if it is different than the line in main memory by seeing if the state is Modified or Owned. Should it belong in either state, then the main memory must be updated by copying back the value. After that it is save to delete that block from the tail of the cache.

move_to_front

This method takes a line which already exists in the cache and deletes it from where it was and reinserts it back into the front of the cache. This is done because the cache is organized by Least Recently Used, meaning that the more recently used blocks in the cache are found near the head, and the least recently used blocks are found at the tail.

read_search

Given a Read Miss, the originating cache broadcasts a request for any other caches which may have the desired block. This method goes through each cache and searches for the line with the needed tag. Should it find said tag, the other cache sends it back to its originating cache. If the block in the cache is Modified or Owned it has priority to sending it to the requesting cache as that is the most up to date value for the block. If it is in the Modified state then the state is changed to Own to indicate that it is not the only valid copy of the line in every cache. Should the state be Exclusive then it needs to change to Shared since once that block is sent back to the originating cache, it will not be the only copy.

write_search

Given a write miss, the originating cache broadcasts a request for any other caches which may have the block that needs to be modified. *Write_search* goes through each of the other caches and uses *find_line* to search for a block with the necessary tag. Should one be found, it is sent back through the bus to the originating cache for edits. At this point all other instances of the block become “dirty” and need to be invalidated, however should the state be Modified or Owned – meaning that the line in the cache differs from what is saved in memory, that instance of the block is copied back to the main memory.

flush

At the end of the program the system flushes the caches. To do this *flush* goes through each cache and starts deleting blocks from the head. Should the block that is being removed be more up to date than what is in memory, it is copied back to main memory. This process is repeated until the entire cache is empty.