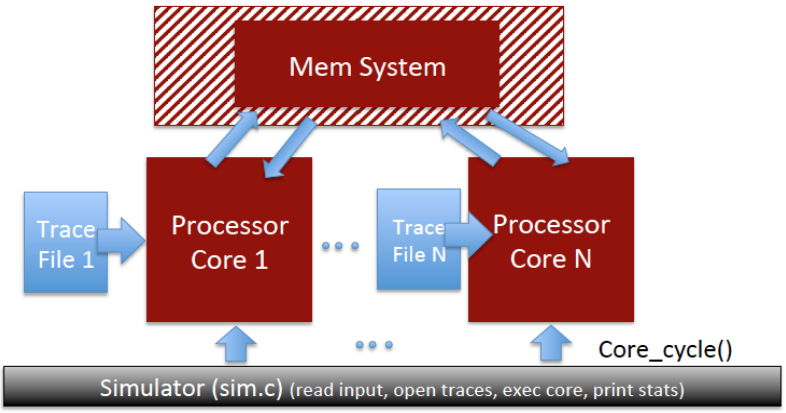
CS 4290 / CS 6290 / ECE 4100 / ECE 6100
Advanced Computer Architecture

Lab 4: CMP Memory System Design - Coherence (3 Pts Extra Credit)
Part F Due: Sunday, December 3, 2023 (11:55 pm)



This is an individual assignment. You can discuss this assignment with other classmates but you should code your assignment individually. *You are NOT allowed to see the code of (or show your code to) other students or of past submissions that may be available online.* We will be using software that detects code similarity. If you think you collaborated closely enough with another student that could lead to more code similarities than would normally be expected, please declare it upfront, at time of submission (as a comment in your Canvas submission). *Please note that such declaration is not granting you an excuse for copying code.*

Part F extends the previous parts of previously released Lab 4. **This part is for extra credit. The TAs will only help with any clarification questions, and should not be expected to offer assistance in debugging.**

Part F: Design MSI coherence mechanism for the multicore system (3 points Extra Credits)

Reference execution time: 30-40 sec per trace mix

In Part D, we assessed the effectiveness of your memory system for a multicore processor for three different mixes of workloads. We used an address translation function that produced unique addresses for each core, removing the requirement of implementing coherence, as the different cores were guaranteed to never access overlapping addresses. In Part F, we remove that guarantee, so you must design an MSI-based coherence mechanism for the multicore processor system. Your task is to account for all the coherence state changes and messages requires during a workload's execution. As discussed in the lecture, snoopy MSI coherence protocols use the following messages: GetX, GetS, PutX and Invalidation. When there is a read/write access to a cache, the coherence mechanism generates the appropriate message and places it on the bus.

Based on the message, the cache line is fetched either from another cache or from the next level of memory.

For simplicity, we will not be modeling the bus. Instead, we want to simply account for the messages that would be exchanged over a bus to maintain coherence. On every L1 data cache access, you must check the access cache line's coherence state. If the access cannot be locally completed given the cache line's current coherence state, you must check the other core's L1 data cache. If the cache line is found there, you must change its coherence state as needed, change the local cache's cache line state as needed, account for the latency of the required coherence transaction, and update coherence statistics that keep track of all types of coherence-related events.

Notes:

1. The cores execute the instructions atomically and you should assume that coherence transitions complete atomically. So, we do not have the issue of race condition among the cores. This also removes the necessity for intermediate states while transitioning from one coherence state to another.
2. You only need to model the coherence mechanism for the Data Caches. For data accesses, use the function memsys_vpn_to_pfn_modeF() which takes in the entire virtual address and returns a physical address that can be used directly. You do not need to understand what this function does, as it does not have any semantic meaning – it is constructed with the sole purpose of creating address overlaps that will result in coherence requirement. **For instruction caches, we will continue to use the function memsys_vpn_to_pfn() to translate the VPN to PFN.**

Your objective is to write the following functions in memsys.cpp and cache.cpp:
1. **memsys_access_modeF**, which returns the delay required to service a given memory access.
2. **cache_access_coherence,** which is called by memsys_access_modeF. If the accessed cache line is present in the cache, return HIT. If HIT, set the required coherence state of the cache line.
3. **cache_install_coherence**, which installs the line in the cache and track the evicted line. This function also sets the coherence state of the newly installed cache line.
4. **set_Cacheline_state**, which sets the Coherence State of a cache line based on the snooped read/write of the cache line in the other cache.
5. **get_Cacheline_state**, which returns the Coherence State of the cache line.

How to run the simulator:

1. ./sim -h (to see the simulator options)
2. ./sim mode 1 ../traces/*.mtr.gz (to test the default configuration)
3. ../scripts/runall.sh runs all configurations for all traces
4. ../scripts/runall.sh takes nearly 30 minutes to execute.

WHAT TO SUBMIT FOR Part F:

For part F you will submit a compressed folder **src_F.tar.gz**, containing the src folder with all source code files and the makefile:

src_F.tar.gz:
- src/*.cpp
- src/*.h
- src/makefile

REFERENCE MACHINE:

Gradescope autograder:
A simple autograder on Gradescope generates a score based on the difference of the reference outputs and your code's outputs for a subset of all traces. Please note that the autograder is not comprehensive and your code will be tested against additional traces.

You can use the virtual machine **oortcloud.cc.gatech.edu** to develop and test your code. Your submission must compile, run, and produce the correct results on Gradescope. Please ensure that your submission on Gradescope produces the desired output (without any extra printf statements).

Notes:

1. We are dealing with the coherence of only two cores. So, we are taking a simplistic approach to modeling the coherence protocol by directly checking the state of the other cache, unlike an actual coherence protocol implementation, which exchanges messages between snoopy coherence agents over a bus. We still want to account for all the messages that would have been exchanged over a bus (GetX, GetS, Invalidate, PutX), which should be reflected in the collected statistics.
2. According to the MSI protocol, we will write back a cache line using PutX message if there is an Invalidation or GetX message seen on the bus. We will model the above behavior using writebacks and state transitions.
3. The decision to invalidate a Modified cache line or degrade it to the Shared state depends on whether there is a write or read request from the other core for the same cache line.
4. The replacement policy for the L1 caches will be LRU. L2 cache will use LFU+MRU replacement policy.
5. Upon a cache access, the set_Cacheline_state function is used to change the coherence state of a cache line of both the local cache and another core's cache. The same function is used again to change the coherence state of the remote cache as well, if necessary.
6. Since the L2 Cache does not deal with the coherence, we can reuse the same memsys_L2_access().

Appendix A: Cache Model implementation

In the "src" directory, you need to update two files:

- cache.h
- cache.cpp

The following data structures may be needed for completing the Lab. Their basic structure is the same as the one in the previous parts of Lab 4. Some extensions are required to implement the coherence behavior of part F. You are free to use any name, any function (with any number of arguments) to implement a cache. You are free to deviate from these structural definitions as well.

1. "**Cache Line**" structure (Cache_Line), will have the following fields:
    - **Valid**: denotes if the cache line is indeed present in the Cache
    - **Dirty**: denotes if the latest data value is present only in the local Cache
    - **Tag**: denotes the conventional higher-order address bits beyond Index
    - **Core ID**: needed to identify the core to which a cache line (way) is assigned to in a multicore scenario (required for Part D, E)
    - **Last_access_time**: to keep track of when each line was inserted, which helps with the LRU/MRU replacement policy
    - **Frequency:** to track the number of accesses to the specific cache line, which helps with the LFU replacement policy
    - **Coherence:** to track the coherence state of the specific cache line (Modified, Shared, Invalid). If a line is unavailable, then the coherence state is INVALID.
2. "**Cache Set**" structure (Cache_Set), will have:
    - **Cache_Line Struct** (replicated "# of Ways" times, as in an array/list)
3. The overarching "**Cache**" structure should have:
    - Cache_Set Struct (replicated "#Sets" times, as in a list/array)
    - # of Ways
    - Replacement Policy
    - # of Sets
    - Last evicted Line (Cache_Line type) to be passed on to next cache hierarchy level for an install if necessary

Status Variables (Mandatory variables required for collecting statistics and generating the desired final reports. Aimed at complementing your understanding of the underlying coherence-related concepts):

- **stat_read_access**: Number of read (lookup accesses do not count as READ accesses) accesses made to the cache
- **stat_write_access**: Number of write accesses made to the cache
- **stat_read_miss**: Number of READ requests that lead to a MISS at the respective cache
- **stat_write_miss**: Number of WRITE requests that lead to a MISS at the respective cache
- **stat_dirty_evicts**: Count of requests to evict DIRTY lines
- **stat_num_invalidations**: Number of invalidation messages sent by the cache.
- **stat_GetX_msg**: Number of GetX messages sent by the cache
- **stat_GetS_msg**: Number of GetS messages sent by the cache
- **stat_PutX_msg**: Number of PutX messages sent by the cache
- **num_s_to_i_transitions**: Number of transitions from SHARED to INVALID state
- **num_s_to_m_transitions**: Number of transitions from SHARED to MODIFIED state

- **num_m_to_i_transitions**: Number of transitions from MODIFIED to INVALID state
- **num_m_to_s_transitions**: Number of transitions from MODIFIED to SHARED state
- **num_i_to_m_transitions**: Number of transitions from INVALID to MODIFIED state
- **num_i_to_s_transitions**: Number of transitions from INVALID to SHARED state

Take a look at "types.h" to choose appropriate datatypes.

NOTE: Don't change the print order/format/names of the cache_print_stats_coherence function.

Appendix B: MSI Coherence Implementation

In our simplified implementation of the MSI protocol, we do not actually model the bus traffic, or the coherence messages sent between the cores. We instead account for all coherence actions by updating the coherence states of the local and remote L1 data cache, triggered by the read/write operations of each core, and by keeping statistics about the bus actions that would have been required to complete each coherence action.

The MSI Coherence protocol has the following states:

- **MODIFIED**: This state indicates that the cache line is present only in the current cache and is dirty. When evicted, the cache must writeback to the memory. No other cache can have a copy of a cache line in MODIFIED state.
- **SHARED**: This state indicates that the cache line may be stored in multiple caches simultaneously and have the same value. In short, this line is clean. If there is a write to a line in SHARED state, it must send an invalidation message to all the other caches to evict their copies of the same line.
- **INVALID**: This state indicates that the cache line is not present or has invalid data. A cache line that becomes invalidated results in the same behavior as if it had been selected as a victim by the cache replacement policy. INVALID lines are always the first candidates to be replaced in a cache set.

To maintain the states across different cores, the protocol also dictates the following bus actions:

- GetX – Triggered when there is a write to a line in INVALID state
- GetS – Triggered when there is a read to a line in INVALID state
- Invalidation – Triggered when there is a write to line in SHARED state
- PutX – Triggered when there is an eviction of a MODIFIED line or when a line in MODIFIED state receives GetX/GetS message from another cache.

Your implementation will not model the exchange of such messages, but must keep statistics of how many of them would have occurred if you were actually modeling a bus that exchanged those messages.

Take the following into consideration while developing your coherence implementation:

1. Each load/store operation looks up the coherence state of the corresponding cache line in the remote cache, and changes its coherence stated as needed.
2. Coherence misses can occur. A coherence miss occurs when the accessed cache line is not in the required coherence state. An example could be a line about to be written that is found in SHARED state. Though the line is in the cache, this constitutes as a miss,

because we do not have the line in MODIFIED state. To simplify things, we treat coherence misses as cache hits with a longer latency, equal to STATE_UPGRADE_LATENCY.

3. A miss due to the line being found in INVALID state is equivalent to missing due to the target block being entirely absent from the cache.
4. When there is a GetS/GetX message received by a cache line in MODIFIED state, it responds back with a PutX message containing the dirty cache line. This will be written to both the L2 cache as well as to the core requesting it (cache-to-cache transfer). So, we model the latency for this scenario using CACHE_TO_CACHE_LATENCY
5. A PutX should be treated as a writeback to the L2 cache. However, it differs from eviction. Eviction occurs when there is an invalidation or a line is chosen as a victim based on replacement policy. In short, PutX is a superset of dirty evictions. To write back a line to L2, you can make use of the last_evicted_line data structure in the Cache data_structure.

The new latencies introduced are as below and are provided in the memsys.cpp:

STATE_UPGRADE_LATENCY 3
CACHE_TO_CACHE_LATENCY 5