NC State University

Department of Electrical and Computer Engineering

ECE/CSC 506: Fall 2015

Machine Problem 3: Multilevel Cache Coherence Protocol

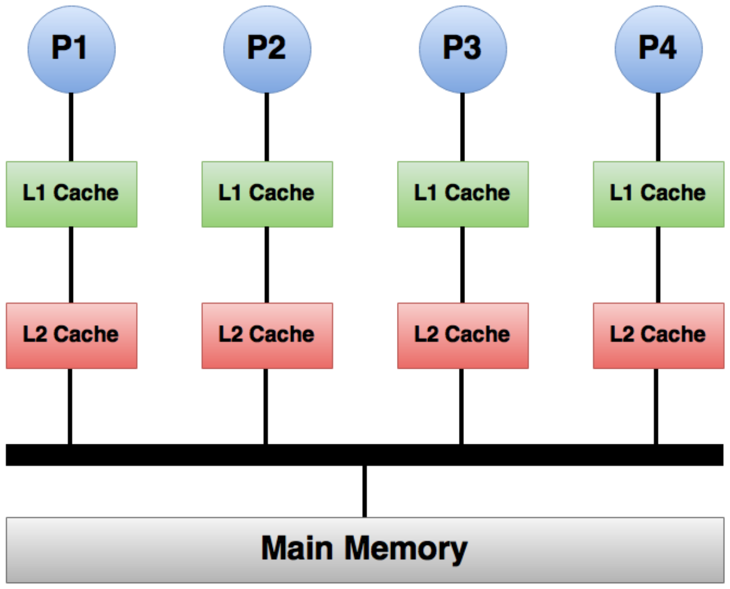
Due: Tuesday, December 1 at 11:55 PM

1. General Guidelines

- All students must work alone.
- Sharing of code between students is considered cheating and will receive appropriate action in accordance with University policy.
- A Moodle forum will be provided for posting questions and discussions. Students should never post actual code on the the forums.
- You must do all your work in the C++ languages.
- Use of the eos Linux environment is required. This is the platform where the TAs will compile and test your simulator.

2. Overall Description

In this machine problem, you will extend the memory hierarchy you worked on in Machine Problem 2. You will add one level of cache right above the cache that you worked on in Machine Problem 2. We will call the new level of cache the "L1 cache." We will call the cache for which you implemented the coherence protocols in Machine Problem 2 "the L2 cache."



3. Specifications

This section will explain the specifications of the SMP simulator we are implementing.

Memory Hierarchy

Each processor will have its own private L1 and L2 caches. The memory hierarchy is the same as in the figure on the side.

Coherence Protocol

MESI cache coherence will be maintained across the L2 caches only (No MSI nor Dragon protocols are needed). Between the L2 cache and L1 cache, coherence is maintained using a multi-level cache coherence protocol. To simplify the problem, assume the L1 cache is write through. Thus, the state of any L1 cache block can only be one of the following: 1) present and invalid; and 2) present and valid. Again, you do not need to provide functionality for MSI or Dragon protocols for L2 caches.

Cache Requests Management

- Read or write requests to any address will be scanned from the trace file (just as MP2), and they will be delivered to the L1 cache of the corresponding processor.
- Read or write hits in either L1 or L2 happen when the requested block is **Present** and **Valid** in the corresponding cache.
- Read or write misses in either L1 or L2 happen when the requested block is either **Not Present** in the cache or present but **Invalid**.
- **Handling a read request**:
    - L1 read hit:
      Update L1's LRU and increment the relevant statistics (Do not propagate the request to L2 and do not update L2's LRU).
    - L1 read miss:
        - First, allocate a space in L1 cache for the block you will receive from L2 (Do not wait for potential back invalidations from L2).
        - Update L1's LRU.
        - After that, propagate the request to the L2 cache of the same processor.
        - If the block in L2 is in **Modified, Exclusive, or Shared** states, L2 updates its LRU and sends the block to L1.
          If the block in L2 is in **Invalid** state or is **Not Present**, then L2 has to request the block from another cache or the memory (by issuing a BusRd request). Then, L2 updates its LRU and sends the block to L1.
- **Handling a write request**:
    - L1 write hit:
        - Update L1's LRU.
        - After that, propagate the request to the L2 cache of the same processor to verify that it has the block in Modified state.
        - If the block in L2 is in **Modified** state, proceed with the L1 write (i.e. update L2's LRU and increment the relevant statistics, since we are not simulating the cache data array).
          If the block in L2 is in **Exclusive** state, it transitions to Modified state, and then L1 can proceed with the write (i.e. update L2's LRU and increment the statistics).
          If the block in L2 is in **Shared** state, L2 cache has to issue a BusUpgr before it can upgrade the state to Modified and let L1 proceed with the write (update L2's LRU and increment the statistics).

- L1 write miss:
  - Propagate the request to the L2 cache to verify that the block is valid and in Modified state (Do not update L1's LRU since it is an L1 write miss).
  - If the block in L2 is in **Modified** state, L1 can proceed with the write (i.e. update L2's LRU and increment the statistics).
    If the block in L2 is in **Exclusive** state, it transitions to Modified state. Then L1 proceeds with the write (i.e. update L2's LRU and increment the statistics).
    If the block in L2 is in **Shared** state, L2 cache has to issue a BusUpgr before it can upgrade the state to Modified. Then L1 can proceed with the write (i.e. update L2's LRU and increment the statistics).
    If the block in L2 is in **Invalid** state or is **Not Present**, L2 has to request the block from another cache or memory (by issuing BusRdX request). Then L1 can proceed with the write (i.e. update L2's LRU and increment the statistics).

Inclusion Policy

- You will implement 1 inclusion policy: inclusive.
- According to the inclusive property, an outer cache should be a superset of all inner caches it surrounds. i.e. any reference in L1 cache must also hit in the L2 cache.
- In inclusive policy, when there is an L2 eviction (happens when read or write request misses at the L2 cache and a victim block in the L2 cache needs to be evicted), the L2 cache must **back invalidate** the corresponding block in L1 cache as well (assuming it exists there). When the L2 cache snoops a BusRdX or BusUpgr, it also has to **back invalidate** the L1 cache, in addition to invalidating its own cache block.
- Note that in the case of L1 read miss, we do not wait for potential back invalidations from L2 before allocating a space for the block we requested from L2. L1 cache has to allocate a space for the block it will receive from L2 before propagating the read request to L2. If there is a back invalidation from L2, then the back invalidation would arrive at L1 later. These are two separate events: back invalidation is serviced, if any, after block allocation (block allocation is in case of read miss only).

Write Policy

- The write policy for the L1 cache is write-through, write-no-allocate.
- The write policy for the L2 cache is write-back, write-allocate, which is what you already implemented in MP2. No changes to write policy for L2 caches.
- Write-through means: a write in L1 cache has to update the L2 cache.
- Write-no-allocate means:
  - A write to a block that is not present in L1 cache will not cause the block to be allocated in L1 cache. In this case, the write will update the L2 cache directly bypassing the L1 cache.
  - If the block is also not present in L2 cache, the L2 cache has to allocate the block first by doing the following: 1) issuing a BusRdX request; 2) finding an empty spot for the newly fetched block; 3) if there is no empty spot, it evicts an existing block. After that, L2 cache will write to the block it just fetched and allocated.
    Note that because L2 cache is inclusive, if L2 is evicting an existing block, it needs to **back invalidate** the evicted block in L1 cache, if L1 cache has a copy of it.

- A write to a block that is present in L1 cache but is invalid will not update the value of the block in L1 cache. It will only update the value in L2 cache bypassing the L1 cache copy. In other words, if the block exists in L1 cache but is invalid, it will remain invalid upon write (refer to page 205 in the textbook for more clarification).
- If L1 cache has a valid copy of the block, it will update its local valid copy as well as the L2 copy (refer to page 205 in the textbook for more clarification).

4. Getting Started:

- Download and unzip the folder named MP2.zip. After you download and unzip the folder, you will find the following files and directories:
  1. cache.cc
  2. cache.h
  3. main.cc
  4. Makefile
  5. trace/
- "Makefile" generates the executable binary. You might need to extend this Makefile if you added some source files.
- The validation runs will be provided to you later on.
- You have to literally match the output files. We are going to use "diff" in order to spot out any differences. If your output does not exactly match the given validations in terms of results and formats, you will be deducted points.
- You can use the following command to check if your output matches the given validation runs:

```
diff -iw <your_file> <validation_run>
```

- In order to "make" your simulator, go to the MP2 directory and execute the following command (You may need to make changes to the make file)

```
make clean; make
```

- After making successfully, it should print out the following:

```
--------------------------------------------------------
-------------FALL15-506 SMP SIMULATOR (SMP_CACHE) ------
--------------------------------------------------------
Compilation Done ---> nothing else to make :)
```

- An executable called "smp_cache" should be created when you make. In order to run your simulator, you need to execute the following command:

```
./smp_cache <L1_cache_size> <L2_cache_size> <L1_assoc>
<L2_assoc> <block_size> <trace_file>
```

  Where
  - `smp_cache` is an executable of the SMP simulator generated after making.
  - `L1_cache_size` is the size of all L1 caches (all L1 caches are of the same size).
  - `L2_cache_size` is the size of all L2 caches (all L2 caches are of the same size).

- `L1_Assoc` is associativity of all L1 caches.
- `L2_Assoc` is associativity of all L2 caches.
- `block_size` is block size of all cache lines (all caches have the same block size).
- `trace_file`: is the input file that has the multi threaded workload trace.

5. Inputs to the Simulator

You will scan the processor requests from trace files just the same as MP2. The directory "trace" in the MP2 directory contains the trace files. Each trace file follows the following format:

```
<processor#> <r|w> <hex address>
<processor#> <r|w> <hex address>
. . .
```

- `<processor#>` specifies which processor is requesting the block. It takes values 0, 1, 2, or 3 (There are only 4 processors).
- `<r|w>` specifies the operation requested. It can be either read “r” or write “w”.
- `<hex address>` marks the address requested.
- Example:
  ```
  1  r  a1663dc4
  3  r  e41e82f0
  ...
  ```
  You need to propagate this request to the L1 cache, and if needed to the L2 cache (maintaining coherence across L2 caches at the same time).

6. Report:

- For this problem, you will experiment with various cache configurations and measure the cache performance with fixed number of processors (4 processors). The cache configurations that you should try are:
  - Varying cache sizes:

Simulation #	L1 cache size	L2 cache size	L1 associativity	L2 associativity	L1 & L2 block size
1	256 KB	512 KB	4-way	8-way	64 bytes
2	512 KB	1 MB			
3	1 MB	2 MB			
4	2 MB	4 MB			

  - Varying block size:

Simulation #	L1 cache size	L2 cache size	L1 associativity	L2 associativity	L1 & L2 block size
5	2 MB	4 MB	4-way	8-way	32 bytes
6					64 bytes
7					128 bytes

    - Inclusion policy: keep it fixed at inclusive.
    - Write policies: keep it fixed at writeback, write allocate for L2 caches and write-through, write no allocate for L1 caches (writeback is already implemented in the code you used for MP2).
    - Replacement policy: keep it fixed as LRU for all cases (LRU is already implemented in the code you used for MP2).
    - Protocol: keep it fixed at MESI protocol for L2 caches. No protocol for L1 caches.
- For each simulation, run and collect the following statistics:
    - Number of L1 read misses.
    - Number of L1 write misses.
    - Number of coherence misses in L2 cache.
    - Number of back invalidations from L2 to L1 cache in case of inclusive policy.
    - Number of L1 cache fills. L1 cache fills are the cases when you request a line and the line is delivered to you by L2. Note that in case of write misses, when you request the line from the L2 cache, you do not fill the line in L1 cache. This is because it is write no allocate policy for L1 cache.
    - Total miss rate for L1 cache (rounded to 2 decimals, should use percentage format).
- Present all the statistics above in tabular format.
- Present the number of L1 back invalidations and L1 cache fills in two separate figures as well as in tables. For both of them, use the average values of the 4 processors.
- Finally, by comparing the behavior of various statistics, compare the performance and explain your observations. You should provide a detailed description with an insightful discussion to earn full credit.

7. Submission

- What you need to submit: all source files – all header files – the Makefile – the report
- You need to **zip all of the files listed above in one zipped folder named "*uintyID*.zip"** (For example, if your unity ID is "*mjone7*," your filename should be "*mjone7*.zip").
- No folders are allowed in your zip file.
- To zip the files, put the *report.pdf* file in the directory where all the .cc, .h, and Makefile are. Then use the following command from the main directory where all the files are:

```
zip unityID.zip *.cc *.h Makefile report.pdf
```

- Note that **we will test your project only on *grendel*** (grendel.ece.ncsu.edu) machine, so make sure your project can be properly compiled (using make command) and run on *grendel* before submission.

8. Grading Policy:

- Grade will be distributed as follows:
  - 60%: Inclusive policy
  - 20%: Two secret runs
  - 20%: Report
- If your output does not **exactly** match any validation run, you will be given only 24% for that part. (Instead of 60%).
- If your output does not match the secret runs, you will get 0% for that part.
- In order to get full credit for the report, you need to provide detailed discussion and performance evaluations results.
- Automatic 2-day extension beyond the deadline is given (Dec. 3rd at 11:55 PM). However, extra 3 points will be given if the assignment is submitted by the deadline.

9. Implementation Suggestions

- You are free to implement the Machine Problem whichever way you want. Though, you have to make sure of two things: 1) it runs on eos Linux environment; and 2) it compiles correctly and produces an executable using “make” command.
- It is up to you to implement a new class for L1 caches or to use the class you built in Machine Problem 2 as a starting point. You can make other instances of the cache class you modified in MP2 to be the L1 caches. Then you can manage the communication between L1 and L2 caches from the “main” function. For example, when you call the “Access” function of the L1 cache instance, the function can return the request it wants to forward to the L2 cache, if any. Then you send the request to the L2 cache by calling the “Access” function of the L2 cache instance. After that you send the bus request, if any, to other caches.
- Some of the functionality you implemented in MP2 will not be used in L1 cache. For example, there is no cache coherence protocol for L1 cache. You only need to mark the block valid or invalid. If it is not present, you treat it as invalid.