Georgia Institute of Technology
School of Computer Science
CS 4290/6290, ECE4100/6100: Spring, 2018 (Conte)

Project 3: Cache Coherence

Due: *April 25th at 11:55pm*

Version 1.0

Rules

1. Sharing of code between students is viewed as cheating and will receive appropriate action in accordance with University policy.
2. It is acceptable for you to compare your results, and **only** your results, with other students to help debug your program. It is not acceptable to collaborate either on the code development or on the final experiments, either in person or on piazza.
3. You should do all your work in the C++ programming language, no exception.
4. Unfortunately experience has shown that there is a very high chance that there are errors in this project description. The online version will be updated as errors are discovered. *It is your responsibility to check the website often and download new versions of this project description as they become available.*
5. Late projects will not be accepted.

Project Description:

On the surface cache coherence seems straightforward; all caches simply must see **all** operations on a piece of data in the same order. Implementation of coherence, however, is not so simple. In this project, you will be creating a simulator that maintains coherent caches for a 4,8, and 16 core CMP. **You will be implementing the MESI, MOSI and MOESIF protocols for a bus-based broadcast system.**

Specification of the Simulator:

A simulator will be provided that is capable of simulating a 4, 8, and 16 core CMP system. Each core will consist of a memory trace reader. This trace reader will read in trace files provided to you. The trace reader code will be provided to you.

Each core in the CMP has one level of cache. The cache is fully associative, has infinite size, and has a single cycle lookup time. The base cache code is provided for you. You will only need to implement the protocol files needed to process requests at the cache (described later). The CMP has a single memory controller which can access the off chip memory. This memory controller is provided for you and will respond to any query (GETS or GETM) placed on the bus with data after a 100 cycle delay.

The bus modeled is an atomic bus. This means that once a query is placed on the bus, the bus will not allow any other requests onto the bus until a DATA response is seen. Caches request the bus using the bus_request() function. If the bus is not available, it will place the request on an arbitration queue to be scheduled in the future. This is done on a first come first served basis, with node 0 having the highest priority and node N having the lowest priority.

Each processor (trace reader) will have up to one outstanding memory request at a time. The processor will send a request to the cache and will wait until the cache responds with a DATA msg. (This will be done using the send_DATA_to_proc() command)

We have provided a full simulator framework in C++. This framework creates the simulator, reads in the traces, creates a basic cache structure, and creates the memory controller. All of this code is in the sim/ directory of the downloadable code. **You should not need to change ANY code in the sim/ directory.**

The protocols/ directory contains files you may need to modify. Most notably, you need to implement the protocol files. When a request comes from a processor to the cache, the cache finds the entry and then calls **process_cache_request()** in the protocol. It is in this function that you should look at the cache entry's state and decide what messages (if any) should be sent, and what state the cache should transition to. When a request is snooped on the bus, the cache finds the entry and then calls **process_snoop_request()** in the protocol. It is in this function that you should look at the cache entry's state and decided what messages (if any) should be sent, and what state the cache should transition to.

To help you in understanding the framework, the MI protocol is already completed and given to you. You MUST fill in the following files: **MESI_protocol.h/.cpp**, **MOSI_protocol.h/.cpp**, **MOESIF_protocol.h/.cpp**.

In order to interface properly with the Simulator, do not change any of the class names or delete any functions. ***You may however need to add additional functions, states, and/or messages*** in order to complete the assignment. If you add additional states, they need to be placed in both the enum in the header file, as well as the array in the protocol's dump() function.

Important Notes About Simulator Assumptions:
1) All requests that are not DATA (GETS and GETM) always expect to have someone reply with DATA. To ensure this, the memory will always respond 100 cycles after the request with DATA **unless** another cache places DATA on the bus first.

There are cases in the traditional protocol where there were certain messages that did not expect replies (e.g. Bus_Upgrade). These types of messages are not supported by the bus and memory, so you cannot use them. Instead you should always send a query that expects a data response (e.g. GETS and GETM). This creates situations where the cache sending the GETS or GETM may be the one that should supply the data. In these cases, the cache should simply send DATA to itself on the bus. Given the difficulties in identifying your own GETS or GETM request, the validations assume that you only reply when you are already responsible for supplying the data.

2) In general, there is more than one way to implement each protocol. For this project, the reference implementations were made with simple logic and emphasizing cache to cache transfers, while running in minimal time. For example, contrary to the MESIF article in Wikipedia, the reference implementation will keep the F state at the same processor and not move it.

3) More will be added based on students' questions, if needed. Check back often.

Framework details:

The framework is downloadable as a tar archive. This framework uses C++ and should run on university Linux machines.

You should fill in the empty functions (and add any needed functions or states) to the protocol files in the protocols/ directory.

Installing and Running:

Installing:
To begin this assignment first download and extract the provided framework.

1) Download project3.tar
2) Unzip using:
   tar xf project3.tar
3) cd project3

In this directory you will find a makefile, sim/ directory (**provided code**), protocols/ directory (**where you should edit**), and various traces (in the traces/ directory).

How to run:
In the root directory type *make* and hit enter. This should build an executable in the root directory called ***sim_trace***.

Run the simulator using

./sim_trace -t *trace_directory* -p *protocol*

As an example, right after download you can test your install by running:

./sim_trace -t traces/4proc_validation/ -p MI

Trace_directory is the directory with the trace you want to run. A trace directory consists of a trace for each core in the machine and a config file that contains the number of cores for this trace. Each line in the trace directory denotes one memory access and includes the action (read or write) and the address.

Protocol is the protocol you want to run. The options supported by the framework are:

- MI (already implemented)
- MESI (need to implement)
- MOSI (need to implement)
- MOESIF (need to implement)
- MSI (**no** need to implement)
- MOESI (**no** need to implement)

Validation:
Inside of each trace directory you will find multiple text files for the validation runs of each protocol. You should perform experiments on all of the provided experiment traces. To help you with debugging, validation outputs are also provided for MSI and MOESI (in addition to the required protocols) – you do **not** have to match these.

Statistics (Output)

- Final cache coherence state (Already output by framework)
- Number of cycles to complete execution (Already output by framework)
- Number of cache misses (This can be due to a cold miss or coherence)
- Number of cache accesses (Already output by framework)
- Number of "silent upgrades" for the MESI (and extensions) protocol
- Number of Cache-to-cache transfers (This refers to the number of times data is not supplied by Memory)

Analysis:

1. For each program (trace) individually, which protocol (MESI, MOSI, MOESIF) would you recommend, and why?
    a. Summarize key results/take-aways in an intuitive manner using appropriate data visualization techniques (such as plots and tables) to explain your reasoning.
    b. Hint: Compare each of the provided programs using the various protocols you implemented (MESI, MOSI, MOESIF).
    c. Hint: Using the statistics above (and any other information you deem necessary), reason out why certain protocols perform better for certain traces.
2. If you were to architect a system where all the provided programs were equally important, which protocol (MESI, MOSI, MOESIF) would you use, and why?
    a. Summarize key results/take-aways in an intuitive manner using appropriate data visualization techniques (such as plots and tables) to explain your reasoning.
    b. Hint: Recall lessons learnt on aggregation and performance evaluation in the early part of the course.

   c. Hint: Note that the traces have a mix of 4-core, 8-core and 16-core configurations. You may choose to propose one protocol for all three configurations, or one for each. Justify your reasoning.

3. What are the limitations of the simulator? What are some of the enhancements needed to make it more realistic? Limit your answer to a couple of paragraphs.

Validation Requirement

Three sample simulation outputs will be provided on the website. You must run your simulator and debug it until it matches 100% all the statistics in the validation outputs (for MESI, MOSI, MOESIF), plus the final cache contents for each validation run.

What to hand in via Canvas:

- Submit a single **.tar.gz** file that contains exactly one folder and one file in its root:
  - The commented source code for the **protocols** added to the simulator program. This is the **protocols/** folder from your completed implementation.
  - A report (**.pdf**) that contains the design results as required by the Analysis section above.

Grading:

0% You do not hand in anything by the deadline .
25% Your simulator doesn't run, does not work, but you hand in significant commented code.
+25% Your simulator matches the validation outputs posted on the website.
+40% If your simulator is successfully validated, 15% each for analysis parts 1 and 2, 10% for part 3.
+10% The analysis report is award quality, with efficient data visualization and persuasive arguments.

Hints:

1. This is a difficult assignment. Start immediately!
2. Review the code for the MI protocol. Additional hints for implementation can be found in the comments.
3. Ask questions if you are stuck!
4. Check canvas and this document often as it will be updated as bugs are found and questions arise.
5. Look up information on cache coherence and state diagrams for assistance. In the past, some students have used Chapters 5 and 6 in the book Parallel Computer Architecture: A Hardware/Software Approach by Culler and Singh.

6. Re-read the RULES at the beginning of the document, and respect them in accordance with the Georgia Tech Honor Code.