Georgia Institute of Technology
School of Computer Science
CS 4290/6290, ECE4100/6100: Spring, 2012 (Conte)

Project 3: Cache Coherence

Due: *April 27th 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.
3. You should do all your work in the C, C++ or Java 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.*

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 MI, MSI, MESI, MOSI, MOESI, 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: **MSI_protocol.h/.cpp**, **MESI_protocol.h/.cpp**, **MOSI_protocol.h/.cpp**, **MOESI_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.

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 type 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.

2) More will be added based on students' questions. Check back often.

Framework details:

The framework is downloadable as project3_framework.tar.gz. This framework uses C++ and should run the 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.

More details about the individual files and the provided functions will be given soon as a lecture and then released as a reference file.

Installing and Running:

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

1) Download project3_framework.tar.gz
2) Unzip using:
   tar zxvf project3_framework.tar.gz
3) cd project3_framework

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. (Note: You should run your simulator on one of the CoC Linux machines). 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/trace1 -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 are:

- MI
- MSI

- MESI
- MOSI
- MOESI
- MOESIF

Validation:

We will soon be providing more traces and validation runs. Inside of each trace directory you will find multiple text files for the validation runs of each protocol. We are only providing validation for some traces. This is to help with debugging without giving the answer to all of the traces. You should perform experiments on all of the provided traces.

Statistics (Output)

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

Experiments:

Compare each of the provided programs using the various protocol. Plot the execution time and using the statistics above (and any other information you deem necessary) explain why certain protocols perform better for certain traces.

Validation Requirement

Four sample simulation outputs will be provided on the website by the TAs. You must run your simulator and debug it until it matches 100% all the statistics in the validation outputs posted on the website, plus the final cache contents for each validation run. You are required to hand in this validated output with your project (see grading).

What to hand in via T-Square:

1. Output from your simulation showing it matches 100% all the statistics in the validation outputs posted on the website, plus the final cache contents for each validation run.
2. The design results of the experiments for each trace file, with a *persuasive* argument of the choices that were made.
3. The commented source code for the simulator program itself.
4. Note that late projects will not be accepted.

Grading:

0% You do not hand in anything by the deadline
+50% Your simulator doesn't run, does not work, but you hand in significant commented code
+10% Your simulator matches the validation outputs posted on the website
+30% You ran all experiments and found a cache for each (your simulator must be validated first!)
+10% The project hand in is award quality. For example, you justified each cache with graphs or tables and a persuasive argument.

Hints:

1. This is a difficult assignment. Start immediately!
2. Ask questions!
3. Check t-square and this document often as it will be updated as bugs are found and questions arise
4. A suggest starting point is to create a framework where you can read the traces, and look up data in the caches and use the MI protocol.
5. Look up information on cache coherence and state diagrams for assistance. A good resource is Chapters 5 and 6 in the book Parallel Computer Architecture: A Hardware/Software Approach by Culler and Singh