CSC/ECE 506: Architecture of Parallel Computers
Program 2: Bus-Based Cache Coherence Protocols
Due: Wednesday, October 26, 2016

1. Overall Problem Description

In this project, you will add new features to a trace-driven cache-coherence simulator. The purpose of this project is to give you an idea of how parallel architectures handle coherence, and how to interpret performance data. You will be given a C++ cache simulator implementing the MSI protocol, and you need to extend that simulator to implement the MOSI and MOESI protocols. Your project should build on a Linux machine. The most challenging part of this machine problem is to understand how caches and coherence protocols are implemented. Once you understand this, the rest of the assignment should be straightforward.

2. Simulator

How to build the simulator

You are provided with a working C++ program for a cache implementing the MSI protocol. There is an abstract base class, `cache.cc`, and a derived `MSI.cc`, which actually implements the cache-coherence protocol. The Cache class implements what is common to each class, and the MSI class implements functionality that is unique to the MSI protocol. You should derive other classes to implement each new protocol. The following methods differ from protocol to protocol:

- `void PrRd(ulong addr, int processor_number)`
- `void PrWr(ulong addr, int processor_number)`
- `void BusRd(ulong addr)`
- `void BusRdX(ulong addr)`
- `void BusUpgr(ulong addr)`
- `boolean writeback_needed(cache_state state)`

The `Bus*` methods take care of snooping a bus operation in *a single* cache; for methods that apply these bus operation to all caches, see the methods `sendBusRd` and `sendBusRdX`, described below.

The `PrRd`, `PrWr`, `BusRd`, and `BusRdX` methods are different for each protocol, because each protocol handles processor and bus actions differently. Of course, for some protocols, you may also need to implement additional bus operations, such as `BusUpgr`. The `writeback_needed` method is different for each protocol, because when a line is ejected from the cache, it has to be written back if it has been modified, and the states that represent modified lines differ from protocol to protocol.

You are provided with a basic main function (in `main.cc`) that reads an input trace and passes the memory transactions down through the memory hierarchy (in our case, there is only one level of cache in the hierarchy). You need to make a couple of changes to this file.

- It needs to read in a parameter representing the protocol, and instantiate the appropriate kind of cache.
- It also handles bus operations, applying them to each of the caches.
- `main.cc` also has methods `sendBusRd` and `sendBusRdX` that take care of applying `BusRd` and BusRdX to each of the caches. Thus, when `PrRd` or `PrWr` needs to send a BusRd or BusRdX out on the bus, it just invokes `sendBusRd` or `sendBusRdX`.
- There is a function `c2c_suppliers()`, which will be used to determine if the requested cache block comes from main memory or cache of other processors. The function body provides comments to required changes and its application.

You may choose not to use the given basic cache and to start from scratch (i.e., you can implement everything required for this project on your own), provided your simulator also uses inheritance to implement the different cache protocols. However, your results should match the posted validation runs exactly.

In this project, you need to maintain coherence across the one-level cache. For simplicity, assume that each processor has only one private L1 cache connected to the main memory directly through a shared bus, as shown in Figure 1.

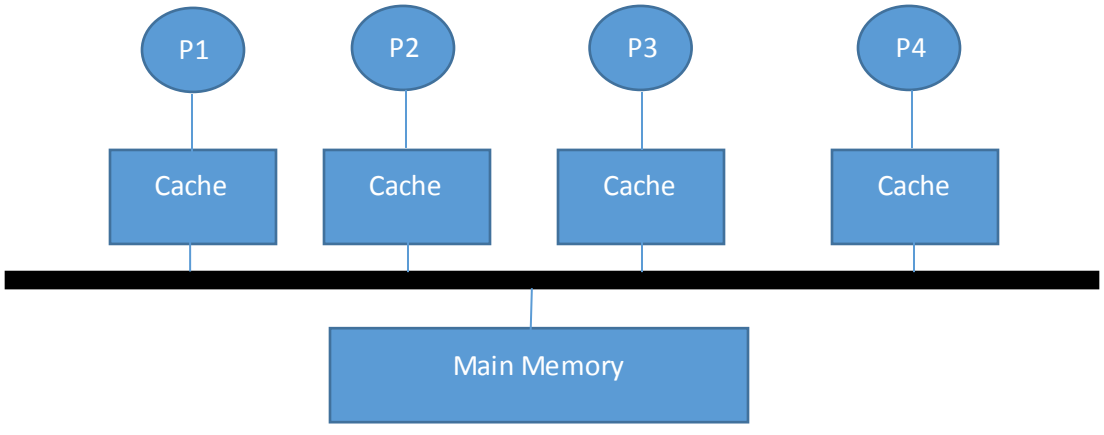

Figure 1. Homogenous SMP system consists of four processors, each connected to a private L1 cache. All Caches are connected to the Main memory through a shared bus.

Note: The given simulator implements write-back, write-allocate (WBWA) and the LRU replacement policy. So in case you are planning to create or use your own simulator please keep these policies in mind.

Requirements

For this programming assignment, you should implement the MOSI protocol and the MOESI protocol and match the validation runs given to you exactly. The output will consist of statistics mentioned below.

Your simulator should accept multiple arguments that specify different attributes of the multiprocessor system. One of these attributes is the coherence protocol that is being used. **In other words, your simulator should be able to generate one binary that works with all coherence protocols.** More description about the input arguments is provided in following section.

3. Getting Started

You have been provided with a `makefile`. If you need to modify it, please feel free to do so but don't change the targets and their intent. Your simulator should compile using a single make command.
After making successfully, it should print out the following:

```
------------------------------------------------------------
-----------FA2016-506 BUS-BASED CACHE SIMULATOR -----------
------------------------------------------------------------ -3-

Compilation Done ---> nothing else to make :)
```

An executable called simulate_cache will be created. In order to run your simulator, you need to execute the following command:
`./simulate_cache ⟨cache_size⟩ ⟨assoc⟩ ⟨block_size⟩ ⟨num_processors⟩ ⟨protocol⟩ ⟨trace_file⟩`
where—
`simulate_cache`: Executable of the SMP simulator generated after making
`cache_size`: Size of each cache in the system (all caches are of the same size)
`assoc`: Associativity of each cache (all caches are of the same associativity)
`block_size`: Block size of each cache line (all caches are of the same block size)
`num_processors`: Number of processors in the system (represents how many caches should be instantiated)
`protocol`: Coherence protocol to be used. The different types are listed below.

0: MSI
1: MOSI
2: MOESI

trace_file : The input file that has the multithreaded workload trace. The trace file to use are

canneal.04t.Longtrace and canneal.04t.debug

Your output should match the given validation runs in terms of given results and format. You will need to literally match the results using the diff command. You can use the following command

`diff –iw ⟨given output file⟩ ⟨your output file⟩`

You can dump the output from your simulator to stdout and redirect it to a file using the ">" operator. You will be provided with outputs of 9 validation runs including MSI protocol.

Each trace file has a sequence of cache transactions; each transaction consists of three elements:

`⟨processor(0-3)⟩ ⟨operation (r,w) ⟩ ⟨address (8 hex chars) ⟩`

For example, if you read the line `3 w 0xabcd` from the trace file, that means processor 3 is writing to the address `0xabcd` to its local cache. You need to propagate this request down to cache 3, and cache 3 should take care of that request (maintaining coherence at the same time). Please assure that your program runs on `login.hpc.ncsu.edu`. That is the environment the TA will be using to test it.

4. Report

For this problem, you will experiment with various cache configurations and measure the cache performance of number of processors. The cache configurations that you should try are:

- Cache size: vary from 32KB, 64KB, 128KB, 256KB, 512KB, while keeping the cache block size at 64B.
- Cache associativity: keep it constant at 8-way
- Cache-block size: vary from 64B, 128B, and 256B, while keeping the cache size at 1MB.
- Protocol: MSI, MOSI and MOESI.

Do all the above experiments for each protocol. For each simulation, run and collect the following statistics for each cache:

1. Number of read transactions the cache has received.
2. Number of read misses the cache has suffered.
3. Number of write transactions the cache has received.
4. Number of write misses the cache has suffered.
5. The total miss rate of the cache
6. Number of dirty cache blocks written back to the main memory.
7. *Total number of Invalid to Exclusive transitions for the applicable protocol*

8. *Total number of Invalid to Shared transitions for the applicable protocol*
9. *Total number of Modified to Shared transitions for the applicable protocol*
10. *Total number of Exclusive to Shared transitions for the applicable protocol*
11. *Total number of Shared to Modified transitions for the applicable protocol*
12. *Total number of Invalid to Modified transitions for the applicable protocol*
13. *Total number of Exclusive to Modified transitions for the applicable protocol*
14. *Total number of Owned to Modified transitions for the applicable protocol*
15. *Total number of Modified to Owned transitions for the applicable protocol*
16. Number of transactions of the cache with the memory. This includes the total number of times a read and write is performed from the memory. Writebacks count as writes, BusRds where there is no Flush or FlushOpt will count as reads.
17. Number of cache-to-cache transfers from the requestor cache perspective (i.e., how many lines this cache has received from other peer caches).
18. Number of interventions. (To avoid confusion, we are counting interventions as Modified to Owned state transition only)
19. Number of invalidations (for a bus action, every-time a cache block's state transitions to Invalid) .
20. Number of Flushes (Only compulsory flushes, not FlushOpt).
21. Number of BusRds that have been issued.
22. Number of BusRdXs that have been issued.
23. Number of BusUpgrs that have been issued.

(NOTE: Values 7 to 15 are tracked to enable easy debugging; a mis-match for these will not incur a penalty but if these are wrong, you can be sure that some other metric will mis-match.)

For the bus operations, you should count the number that have been issued on the bus. Do not count one bus transaction for each cache that each operation is applied to. In other words, if there are four processors and P1 issues a BusRd, this counts as only one bus operation, not 4. Overall, the report should

- Present the statistics in tabular format as well as figures in your report.
- Discuss trends with respect to change in the configuration of the system as well as across the protocol.
- Discuss the various aspects of the coherence protocols. You can discuss the bus traffic, number of memory transactions, and complexity of the protocols.

5. Grading

- 20%: Your code compiles successfully
- 60%: Your output matches exactly for runs on all three caches that you are implementing (points will be equally distributed). There will be debug runs provided to you (here are the debug runs for Firefly) and mystery runs which will be carried out on your submitted code.
- 20%: Report. Credit will be given on the statistics shown and discussion presented.

6. Submission

Create a compressed folder named 'project2.zip' and containing the files :

- code files main.cc, cache.cc, msi.cc, mosi.cc, moesi.cc and their headers. (Do not include any object files)
- Makefile that is provided
- Report capturing all the statistics as mentioned in section 4 and named as report.pdf

Use "zip project2 *.cc *.h Makefile report.pdf"
Any deviation from the format mentioned for the files and zip folder will result in deduction of 5 points..

7. Suggestions

1. Read the main, Cache, and MSI classes carefully, and understand how a single cache works.
2. Most of the code given to you is well encapsulated, so you do not have to modify most of the existing functions. You may need to add more functions as deemed necessary.
3. You might create an array of caches, based on the number of processors used in the system.
4. In cache.h, you might need to define new functions, counters, or any protocol specific states and variables.
5. Start early and post your questions on Piazza.