CSC/ECE 506: Architecture of Parallel Computers
Program 3: Bus-Based Cache Coherence Protocols
Due: Wednesday, March 23, 2022

1. Problem Description

This project asks you to add new features to a trace-driven cache-coherence simulator. It is supposed to give you an idea of how parallel architectures handle coherence, and how to interpret performance data. You are given a C++ cache simulator implementing the MSI and Firefly protocols, and you need to extend that simulator to implement the **MOSI** and **Dragon** 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 BusWr(ulong addr)`
- `cache_line *allocate_line(ulong addr)`
- `boolean writeback_needed(cache_state state)`

Note that some protocols do not implement some of the above methods. 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 `BusWr, BusUpgd etc.` 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. The `allocateLine` method is different 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 are different 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). The provided code—

- reads in a parameter representing the protocol, and instantiates the appropriate kind of cache;
- handles bus operations, applying them to each of the caches.

Also `main.cc` has methods `sendBusRd` and `sendBusRdX` that apply BusRd and BusRdX, respectively, 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`.

In a real architecture, a signal would just be sent out on the bus, and all caches would see it. In the simulator, each cache needs to be separately informed that a BusRd or BusRdX is occurring.

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 a single private L1 cache connected to the main memory directly through a shared bus, as shown in Figure 1.
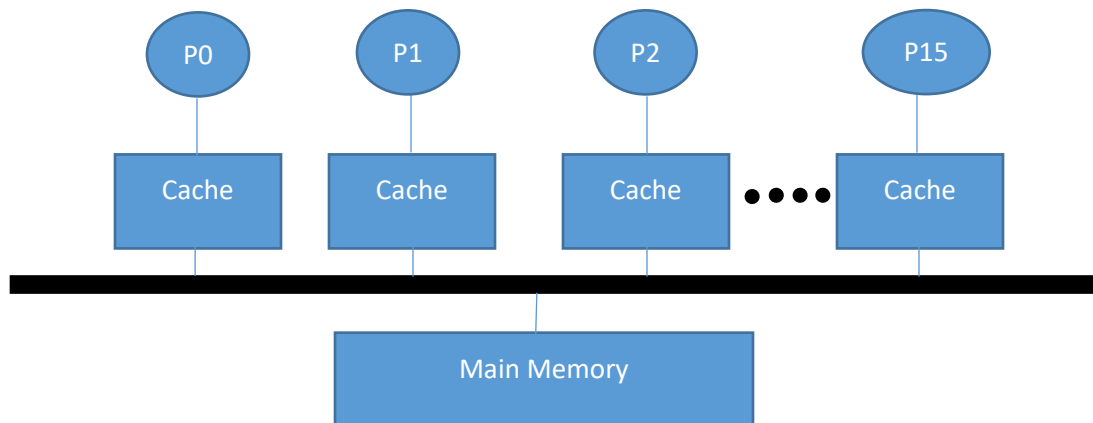
Figure 1. A homogenous SMP system consisting of four processors, each connected to a private L1 cache. All caches are connected to the main memory through a shared bus.

Note: The given simulator's write policy is write-back, write-allocate (WBWA) and it implements 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 and Dragon protocols and match the results produced by the reference simulator exactly

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

We have provided a 3.6 GB trace file for this project, which is too large to download reliably. The file is located at

`/afs/eos.ncsu.edu/lockers/workspace/csc/CSC506-1/trace/swaptions_truncated`

We are trying to get another benchmark trace ready, so that you can see how different traces behave on the same system. If we succeed, we will diminish the number of parameter combinations you need to test, so that the amount of work stays roughly the same.

The trace file(s) can be accessed from any host (such as remote.eos.ncsu.edu) that has access to AFS. We recommend that your run your code on `remote.csc.ncsu.edu`, which is a special virtual server set up for our class. You may write your code on that machine, or transfer it there via scp or WinSCP, etc.

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:
```
--------------------------------------------------------------
------------ SPR2022-506 BUS-BASED CACHE SIMULATOR ------------
--------------------------------------------------------------

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 ⟨project_name⟩ ⟨cache_size⟩ ⟨assoc⟩ ⟨block_size⟩ ⟨num_processors⟩ ⟨protocol⟩ ⟨trace_file⟩`

where—

- project_name: For this program , you should give this parameter the value `smp`
- `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`: 0 = MSI, 1=MOSI, 2 = Firefly, 3 = Dragon
- trace_file : The input file that has the multithreaded workload trace. The trace file to use is swaptions_truncated.

You can use other trace files to debug you code, and only include results from swaptions_truncated into your report. Each trace file has a sequence of cache transactions; each transaction consists of three elements:

> ⟨processor(0-15)⟩ ⟨operation (r,w) ⟩ ⟨address (8 hex chars) ⟩

For example, if you read the line `7 r 0xabcd` from the trace file, that means processor 7 is reading to the address `0xabcd` to its local cache. You need to propagate this request down to cache 7, and cache 7 should take care of that request (maintaining coherence at the same time). Please make sure that your program will run on `remote.eos.ncsu.edu`.

To help you debug your code, we have provided a reference executable called `simulate_cache_ref`. You can run it using the command above, substituting `simulate_cache_ref` for `simulate_cache`. The output from both of the runs should be the same.

We've also provided a shell script that will run both simulators, yours and the reference simulator:

> `sh module.sh ⟨project_name⟩ ⟨cache_size⟩ ⟨assoc⟩ ⟨block_size⟩ ⟨num_processors⟩ ⟨protocol⟩ ⟨trace_file⟩ ⟨number_of_references⟩`

where—

- `module.sh`: Script that will run your code as well as the reference code
- number_of_references: Setting this parameter to k runs both the reference simulator as well as your code on the first k references from the trace file. For example, setting `number_of_references` to `10000` runs the simulator(s) on the first 10000 references in the trace file. If your output is diverging from the correct output, you can use this tool to pinpoint the exact instruction where the first discrepancy occurs.

- Once you run the `module.sh` command, the output will be stored in two files, `results.txt` and `ref_result.txt`.

The diff between these two output files will be stored in the file `difference.txt`. Your output should match the given validation runs in terms of given results and format.

Ensure that you test for all the corner cases and permutations of cache size, associativity etc. You are given an executable file (simulate_cache_ref) rather than validations to help test that none of the corner cases are missed.

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 128KB, 256KB, and 512KB while keeping the cache block size at 64B.
- Cache associativity: vary between 1-way, 2-way, and 4-way.
- Cache-block size: vary from 32B, 64B, 128B, and 256B, while keeping the cache size at 256KB.
- Protocol: MSI, MOSI, Firefly, and Dragon.

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. 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. Both write-throughs and writebacks count as writes, as well as BusWrs in Firefly, since BusWrs update main memory in that protocol.
8. Number of cache-to-cache transfers from the requestor cache perspective (i.e., how many lines this cache has received from other peer caches).
9. Number of interventions.
10. Number of invalidations.
11. Number of flushes.
12. Number of BusRds that have been issued.
13. Number of BusRdXs that have been issued.
14. Number of BusUpgrs that have been issued.
15. Number of BusWrs that have been issued.

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 sixteen processors and P1 issues a BusRd, this counts as only one bus operation, not four. 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.
- Consider the following issues.
  1. Which helps more on miss rate? Doubling the associativity? Doubling the block size? Doubling the cache size? Does it depend on which protocol is in use?
  2. Does larger associativity benefit a large cache or a small cache?
  3. Which protocol (MSI, MOSI, Firefly, Dragon) has the most memory transactions? Which one has the least? Why?

4. Compare the number of read and write misses in MOSI vs. Dragon. Explain your observation.
5. Compare the number of bus transactions and writebacks in MOSI vs. Dragon. Explain your observation.
6. For the Dragon protocol, given the same cache size and associativity, compare the number of BusRds and BusUpds for different block sizes. Explain what happens and why.

5. Grading

- 20%: Your code compiles successfully
- 40%: Your output matches exactly with the one from the reference simulator.
- 40%: Report. Credit will be given on the statistics shown and discussion presented.

6. Submission

Create a compressed folder named ⟨unityID1[_unityID2]⟩.tar.gz containing the files—

- The provided reference simulator
- Code directory including all .cc files. (Do not include any object files, run **make clean** before submission)
- A report of your results, including all the statistics as mentioned in Section 4, and name it report.pdf.

Any deviation from the format mentioned for the files and zip folder will result in deduction of 5 points.

The command to compress:

```
tar -czvf ⟨unityID1[_unityID2]⟩.tar.gz /path/to/your/project3
```

7. Suggestions

1. Read the main, Cache, MSI, and Firefly 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.

8. Resource Information

1. Log on to remote.eos.ncsu.edu using putty or mobaxterm, by using the following command:
   ```
   ssh <unityID>@remote.eos.ncsu.edu
   ```

2. You can transfer the code zip file by using the scp command:
   ```
   scp -r <source_path> <unityID>@remote.eos.ncsu.edu:<dest_path>
   ```