CSC/ECE 506: Architecture of Parallel Computers
Program 3: Bus-Based Cache Coherence Protocols
Due: Friday, July 11, 2025

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 protocol, and you need to extend that simulator to implement the MOESI and MESI protocols. Your project should be built on a Linux machine. The most challenging part of this machine problem is understanding 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 operations to all caches, see the methods `sendBusRd`, `sendBusUpgr`, and `sendBusRdX`.

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` and `allocateLine` methods are 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). 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`, `sendBusUpgr` 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. (No one has ever done this 😊) 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.

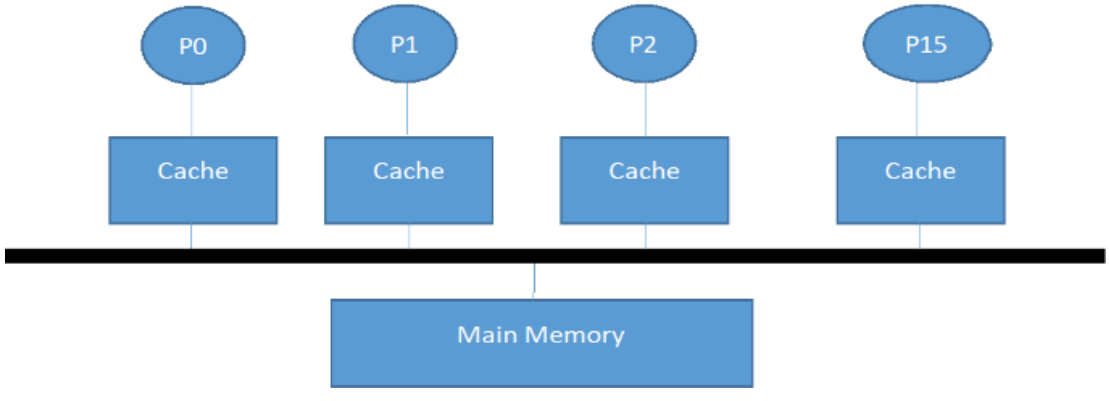

Figure 1. A homogenous SMP system consisting of sixteen processors, each connected to a private L1 cache. All caches are connected to the main memory through a shared bus.

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 MOESI and MESI 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 the following section.

3. Getting Started

We have provided five trace files for this project, which are too large to download reliably. The files are located on this drive. The trace file that you need to use to test your protocol implementations is `streamcluster.simdev.bin.` The remaining trace files are used by the question trace files (Ex: q1.sh).

After running 'make all', 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[unit]> <line_size> <associativity> <coherence>
<replacer> <trace_file> [trace_limit]
```

where—

- `cache_size`: The size of the cache in bytes. Optionally, 'k' or 'M' can be specified as the `unit`. For instance, one can specify '128k' instead of '131072'.
- `associativity`: Associativity of each cache (all caches are of the same associativity)
- `line_size`: Block size of each cache line (all caches are of the same block size)
- `coherence`: msi, mesi, moesi
- `replacer`: lru
- `trace_file`: The input file that has the multithreaded workload trace. The trace files to use are `streamcluster.simdev.bin`
- `trace_limit:` (Optional) The maximum number of traces the simulator should process

You can use **streamcluster.simdev.bin** to debug your code and for creating the tables. For answering the questions, please use the q#.sh files provided. (Note: to enable access, you will first need to run **'chmod +x q#.sh'**. Then, you can run **'./q#.sh'** where # is the question number). 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) ⟩`

CohereSim reads in memory trace files to simulate cache behavior and record statistics. These are binary files comprised of a series of 5-byte memory accesses. Each memory access is structured as follows:

The first 7 bits (7 high bits) are the CPU core ID that performs the memory access

The 8th bit (LSB) is the operation (1 = write, 0 = read)

The remaining 4 bytes are the 32-bit memory address accessed, stored in little endian byte ordering

For example, a trace of 09 70 7D 11 00 evaluates to a write operation by CPU core 4 at address 0x00117D70.

To help you debug your code, we have provided two reference shell scripts, **test_mesi.sh and test_moesi.sh**. (Note: you will need to run **'chmod +x test_mesi.sh'** and **'chmod +x test_moesi.sh'** to run these shell scripts.

4. Report

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

- Cache size: vary from 128KB, 256KB, 512KB, 1024KB, 2048KB while keeping associativity at 4 and block size 64B.
- Cache associativity: vary between 1, 2, 4, 8, and fully associative, while keeping cache size as 1024 KB and block size 64B.
- Cache-block size: vary from 32B, 64B, 128B, 256B, 512B while keeping cache size at 1048 KB and associativity at 2.
- Protocol: MESI 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. 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.
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 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, and
- consider the following issues.
    1. Does increasing the cache size benefit a heavily shared workload? In this context, a heavily shared workload refers to a workload where multiple processes are reading/writing the same data frequently. In other words, how does increasing the cache size affect the # of read/write misses? (Note: Use the **q1.sh** file to run the tests for **MESI** and look for the results in **results/q1**.) Create a graph to provide evidence for your answer.
    2. How does increasing the block size affect the number of read/write misses for

a low granularity workload (small shared data blocks) compared to a high granularity workload (large shared data blocks)? (Note: Use the **q2.sh** file to run the tests for **MESI** and look for the results in **results/q2**.) Create a graph to provide evidence for your answer.

3. Compare the overall miss rates for **MSI** vs **MESI** protocols. (Note: Use the **q3.sh** file to run the tests and look for the results in **results/q3**.) Create a graph to provide evidence for your answer.
4. Does higher associativity benefit a large cache or a small cache more? (Note: Use the **q4.sh** file to run the tests and look for the results in **results/q4**.) Create a graph to provide evidence for your answer.

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.) (Also, do not include the traces folder.)
- 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 Cache and Dragon 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. Start early and post your questions on Piazza.

8. Resource Information

1. Log on to the VCL at vcl.ncsu.edu and choose the **Ubuntu 22 GPU with Cuda (GeForce RTX 2080 Ti)** environment. Note: You will want to select 2-3 weeks for the duration to have your work stay saved in the environment, especially for the large trace files.
2. The trace files should be copied into the traces/ folder within program3.
3. The code files to edit are **src/coherence/mesi.cc** and **src/coherence/moesi.cc**.