ECE/CSC 406/506: Architecture of Parallel Computers
Project 2. Coherence Protocols

Due: Wednesday, November 25th, 11:59 pm

The purpose of this project is to give you an idea of how parallel architecture designs are evaluated and how to interpret performance data.

In this project, you will create a simulator that will allow you to compare different coherence protocol optimizations. You will simulate a 4-processor system with ***Modified MSI*** and ***Dragon Protocol.*** Your job in this project is to instantiate these peer caches and to *maintain coherence* across them by applying coherence protocols.

Your simulator should output various statistics, such as the number of cache hits, misses, memory transactions, issued coherence signals, etc.

The simulator is functional, i.e., the number of cycles and data transfer are not under consideration. It makes the project easy to implement.

Introduction

In this section, the provided infrastructure is described. Particular tasks are described in Parts 1 and 2.

Organization

- src/ - starter source code
    - cache.cc
    - cache.h
    - main.cc
    - Makefile
- trace/ - memory access traces
- val/ - validation outputs for traces

The directory val contains validation outputs you are asked to match. You have to match the output files. If your output does not match the given validations in terms of results and formats, you will be deducted points. You can use the `make val` target to perform a local check.

Starter code

Class Cache simulates the behavior of a single cache. It provides implementation of all basic cache methods and cache line states. Actions that should be performed during cache access are

described in the Access() method. Please study that class thoroughly. Note counter increment and LRU updates.

You can derive your coherent caches from that class by providing a new Access() method (possible with a new Snoop() method) and adding new cache line states. You are free to make **any changes** to the code as long as it compiles into a single binary and has required command line arguments.

Trace format

The trace reading routine is already provided in main.cc. If you want to create your traces for debugging (highly recommended), please use the following format.

Each line in the trace file is one memory transaction by one of the processors. Each transaction consists of three elements: processor(0-3) operation(r,w) address(in hex).

For example, if you read the line `5 w 0xabcd` from the trace file, processor 5 is writing to the address "0xabcd" in its local cache. You need to propagate this request down to cache 5, and cache 5 should take care of that request (maintaining coherence at the same time).

Self-grader and submission

Gradescope self-grader will be provided. To prepare your source code for submission, run `make pack`. You are allowed to add new source files. Make your first submission as early as possible to understand the requirements. Number of attempts is not limited. After you are done with your code and the report, attach the PDF file to your submission.

Cache parameters

Size: 8192B, associativity: 8, block size: 64B

Command line arguments

```
./smp_cache <cache_size> <assoc> <block_size> <num_processors> <protocol> <trace_file>
```

Compiling and running the simulator

Run all commands in the src/ directory. Check targets and parameters in the Makefile

To compile, run `make`. **Common problem:** not running `make clean` after modifying header files.

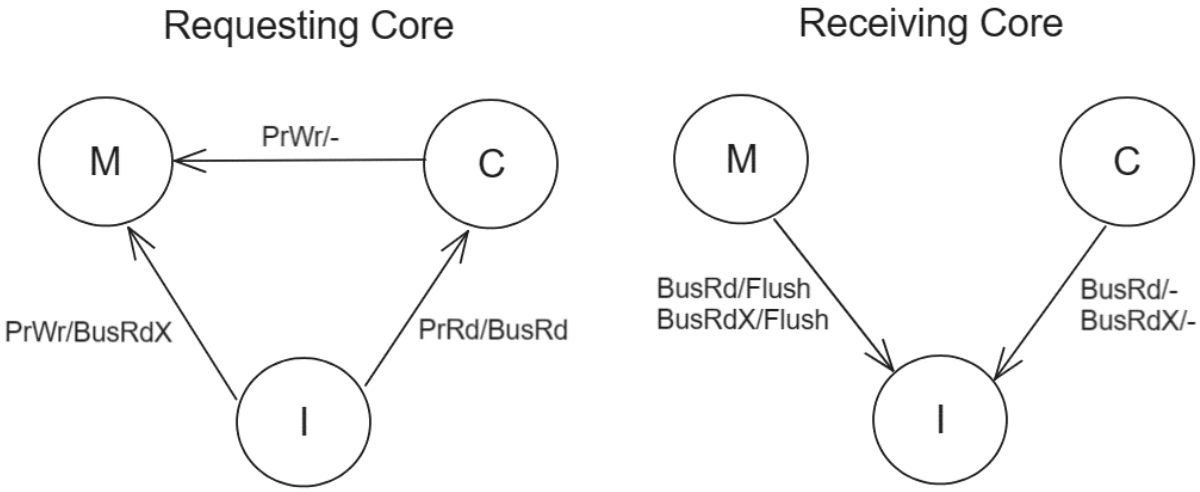
To make sure that your implementation doesn't fail, run `make run.`

To validate your implementation, run `make validate.` If the output looks strange, check that your simulator outputs something with `make run.`

Implementation suggestions

1. Read the given code carefully, cache.cc, cache.h, and understand how a single cache works. Most of the code given to you is well-encapsulated, so you do not have to modify most of the existing methods. You may need to add more functions as deemed necessary.
2. In cache.cc, there is a function called Cache::Access() , representing the entry point to the cache module; you might need to call this function from the main and pass any required parameters to it.
3. Recall that each processor has a separate cache, and for each request in one cache (requestor), there should be some handling in other caches (snoopers).
4. In cache.h, you might need to define new methods, counters, states, or protocol-specific states and variables.
5. You might create an array of caches based on the number of processors used in the system.
6. Feel free to create a class hierarchy and divide your code into multiple files.
7. Start early and do self-grading after each part.

Part 1. Modified MSI



a) (40 pt) Implement Modified MSI protocol as seen in the diagram.

Implementation hint: add an instance of cache per processor, add coherence state support to a cache line, and ensure that Access() in the requestor emits coherence signals and that other snooping caches process those signals.

Output of the simulator

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. Total miss rate (rounded to 2 decimals; should use percentage format)
6. Number of dirty blocks written back to the main memory (due to cache eviction and Flush)
7. Number of memory transactions
8. Number of invalidations (any state->Invalid)
9. Number of flushes to the main memory
10. Number of issued BusRdX transactions

b) (10 pt) Plot the number of memory transactions for the long trace.

Part 2. Dragon Protocol

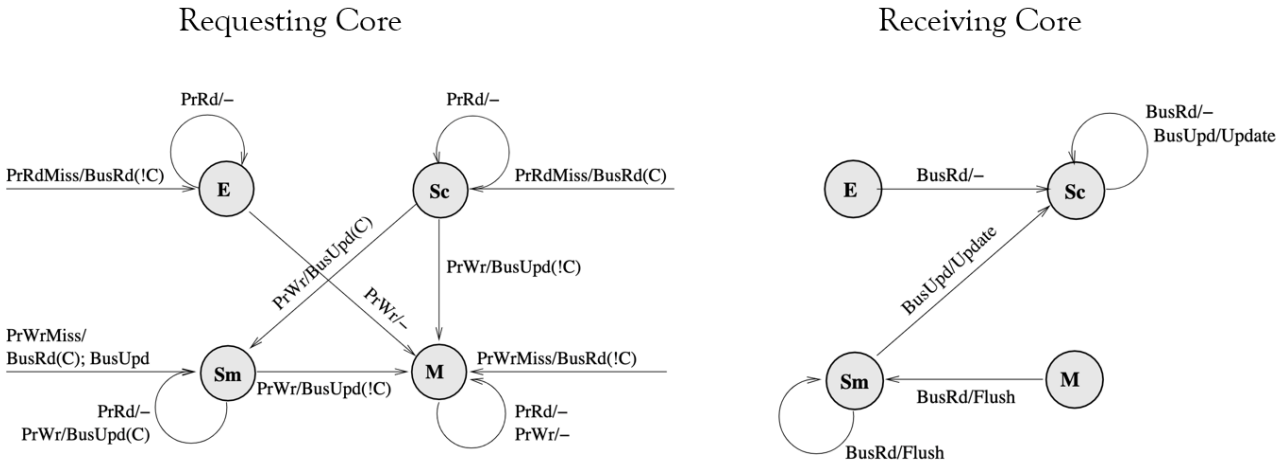


Figure 7.8 page 233 from the book

a) (40 pt) Implement Dragon protocol as seen in the diagram.

b) (10 pt) Compare Modified MSI optimization and Dragon protocols. Explain the results briefly.

Output of the simulator

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. Total miss rate (rounded to 2 decimals; should use percentage format)
6. Number of dirty blocks written back to the main memory (due to cache eviction and Flush)
7. Number of memory transactions
8. Number of interventions (refers to the total number of times that E/M transits to Shared states)
9. Number of flushes to the main memory
10. Number of issued Bus transactions (BusUpd)