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

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

This project aims 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 ***MESI*** and **MOESI.** 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 statistics, such as the number of cache hits, misses, memory transactions, and issued coherence signals, and calculate the total program execution.

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

1. Introduction

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

2. Organization

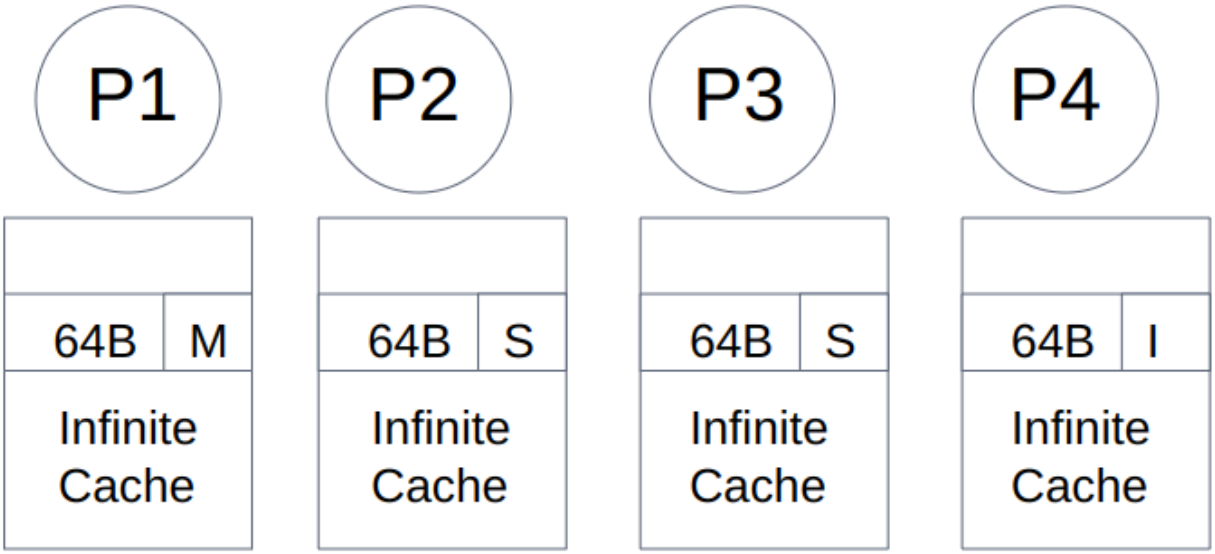
- src/ - starter source code
    - cache.cc
    - cache.h
    - main.cc
    - Makefile
- trace/ - memory access traces
- val/ - validation outputs for traces

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

3. 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.
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.

4. Cache parameters



- Each processor has an Infinite-size L1 Cache with a block size of 64B. This means the associativity is also infinite.
- You do not need to worry about them, as the infinite cache is already modeled. There is a function that updates LRU, which can be ignored as the cache would only have cold misses.

5. 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 represents a memory transaction by one of the processors. Each transaction consists of three elements: processor(0-3) operation(r,w) and address(in hex).

For example, if you read the line 5 w 0xabcd from the trace file, processor five 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).

Two traces are provided.

1. canneal.04t.debug
2. canneal.04t.longTrace

6. 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 clean after modifying header files.

There are two targets for running the script.

1. ***make test_mesi*** - It already contains the configuration to run the experiments.
2. ***make test_moesi*** - - It already contains the configuration to run the experiments.

The only change required is to change the traces; the simulation configuration remains the same.

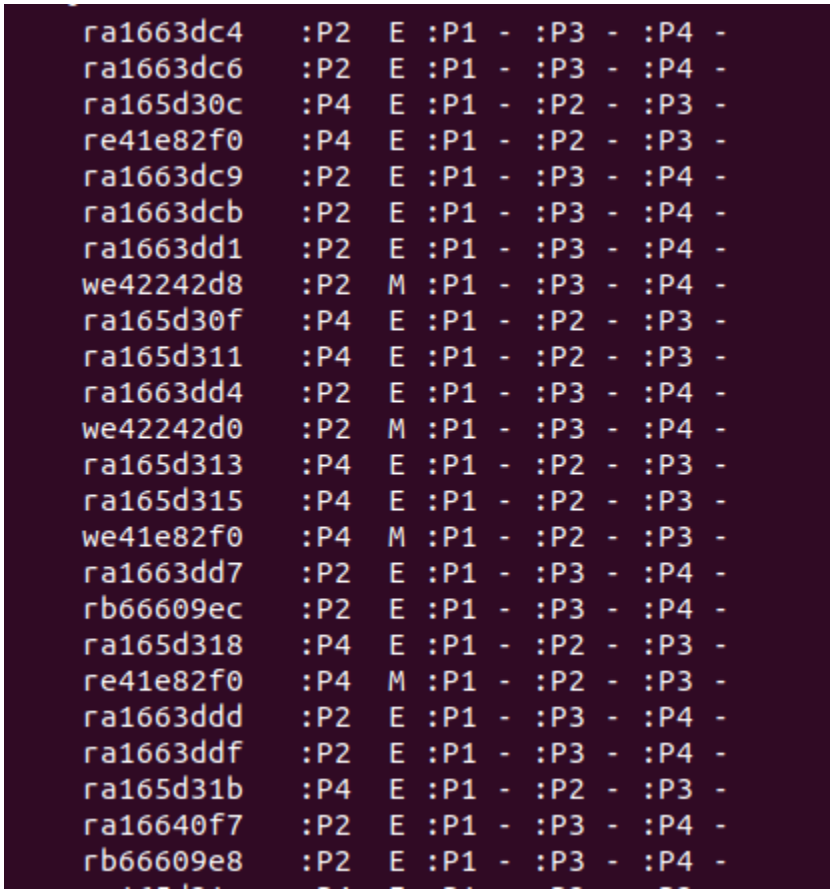
7. Implementation suggestions

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. For MESI or MOESI implementations calculating the program execution time, consider the faster method. For example, if there is a cache-to-cache transfer, use it (add that latency) instead of picking a block from memory.
4. To calculate the overall program execution time, the **latencies** of each operation are provided in **cache.h**
5. 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).
6. You might need to define new methods, counters, states, or protocol-specific states and variables in the cache. h.
7. You might create an array of caches based on the number of processors used in the system.
8. Feel free to create a class hierarchy and divide your code into multiple files.
9. Start early and do self-grading after each part.

Note:- The number of flushes is tricky as various FlushOpt operations would improve the total program execution time.

10. Problem Statement

1. **For canneal.04t.debug, print the outputs in the following format, and store them in the file cache_state.txt.**



```
ra1663dc4   :P2  E :P1 - :P3 - :P4 -
ra1663dc6   :P2  E :P1 - :P3 - :P4 -
ra165d30c   :P4  E :P1 - :P2 - :P3 -
re41e82f0   :P4  E :P1 - :P2 - :P3 -
ra1663dc9   :P2  E :P1 - :P3 - :P4 -
ra1663dcb   :P2  E :P1 - :P3 - :P4 -
ra1663dd1   :P2  E :P1 - :P3 - :P4 -
we42242d8   :P2  M :P1 - :P3 - :P4 -
ra165d30f   :P4  E :P1 - :P2 - :P3 -
ra165d311   :P4  E :P1 - :P2 - :P3 -
ra1663dd4   :P2  E :P1 - :P3 - :P4 -
we42242d0   :P2  M :P1 - :P3 - :P4 -
ra165d313   :P4  E :P1 - :P2 - :P3 -
ra165d315   :P4  E :P1 - :P2 - :P3 -
we41e82f0   :P4  M :P1 - :P2 - :P3 -
ra1663dd7   :P2  E :P1 - :P3 - :P4 -
rb66609ec   :P2  E :P1 - :P3 - :P4 -
ra165d318   :P4  E :P1 - :P2 - :P3 -
re41e82f0   :P4  M :P1 - :P2 - :P3 -
ra1663ddd   :P2  E :P1 - :P3 - :P4 -
ra1663ddf   :P2  E :P1 - :P3 - :P4 -
ra165d31b   :P4  E :P1 - :P2 - :P3 -
ra16640f7   :P2  E :P1 - :P3 - :P4 -
rb66609e8   :P2  E :P1 - :P3 - :P4 -
```

2. For ***canneal.04t.longTrace*** prints the stats in the following format.

01. Number of reads

02. Number of read misses:

03. Number of writes:

04. Number of write misses:

05. Number of write hits:

06. Number of read hits:

07. Total miss rate:

08. Number of memory accesses (exclude writebacks)

09. Number of invalidations:

10. Number of flushes:

11. Total Program Execution time.

9. Report

CSC/ECE-406 (only).

Only Implement MESI.

1. Describe the optimization that MESI does to reduce memory transactions.
2. Describe high-level details of your implementation.

CSC/ECE-506 (only).

Implement MESI and MOESI.

1. Describe the optimization that MESI does to reduce memory transactions.
2. Describe the optimization that MOESI does to reduce memory and bus transactions.
3. Describe high-level details of your implementations.

Note:- Submit your report in PDF format and do not exceed two pages.

9. Grading

CSC/ECE-406 (only).

1. Substantial effort (40)
2. All auto-grader test cases passed (30)
3. Correct cache_state.txt file (10)
4. Report(20)

CSC/ECE-50 (only).

1. Substantial effort (30)
2. All auto-grader test cases passed (30)
3. Correct cache_state.txt file(MESI) (10)
4. Correct cache_state.txt file(MOESI) (10)
5. Report(20)

10. Self-grader and submission

All submissions should be made through Gradescope, not Moodle. Make your first submission as early as possible to understand the requirements. The number of attempts is not limited.

Attach the PDF file to your submission after completing your code and the report.

11. Academic Integrity.

All students must work alone. Sharing code between students is considered cheating and will receive appropriate action in accordance with University policy. The TAs will scan source code (from current and past semesters) through various tools available for detecting cheating. Source code flagged by these tools will be dealt with severely: 0 on the project and referral to the Office of Student Conduct.