Machine Problem 2 for CSC/ECE 506 & ECE 492

Due date: Monday, October 31, 2011, 11:45pm

A. General Instructions

1. Machine problem 2 consists of implementing invalidation-based cache coherence protocols for a SMP system. The project can be done individually or a maximum group of 2 students.
2. You will need to make sure that your code runs on Grendel as that will be used as the test environment.
3. To do well for this machine problem, you need to follow the spec, and turn in everything that is asked. We give you a generous amount of time to work on the problem (more than 2 weeks). Hence, there will be no extensions granted and we recommend that you start working on this right away.
4. You must understand how various cache coherence protocols work before you start coding.

B. MSI, MESI and MOESI Coherence Protocols

Project Overall Description

In this project, students will implement a trace driven SMP simulator (symmetric multiprocessor simulator). You will be given a C++ generic-cache class, and you need to extend that class to work in a multi processor environment and to build various coherence protocols on top of it. You need to write it in C++ and you need to build it on Grendel cluster. 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 would be straightforward. The purpose of this project is to give you an idea of how parallel architecture designs are evaluated, and how to interpret performance data.

Simulator

Building the simulator:

You will be provided with a working C++ class for a uni-core system cache, namely (cache.cc, cache.h). You can start from this point and build up your project by instantiating multiple cache objects to build your SMP system. Then you need to implement the required coherence protocols to keep all peer caches coherent.

You have the choice 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). However, your results should match the posted validation runs exactly.

You are also provided with a basic main function (main.cc) that reads an input trace and passes the memory transactions down through the memory hierarchy (in our case we have only one level in the

hierarchy). Some of the code in “main.cc” is commented out and you need to fill in the correct code. A “Makefile” that generates the executable binary will be provided.

In this project, you are supposed to implement one level of cache only, that is, you need to maintain coherence across one level of cache. For simplicity, you will assume that each processor has only one private L1 cache connected to the main memory directly through a shared bus. As shown the figure below:

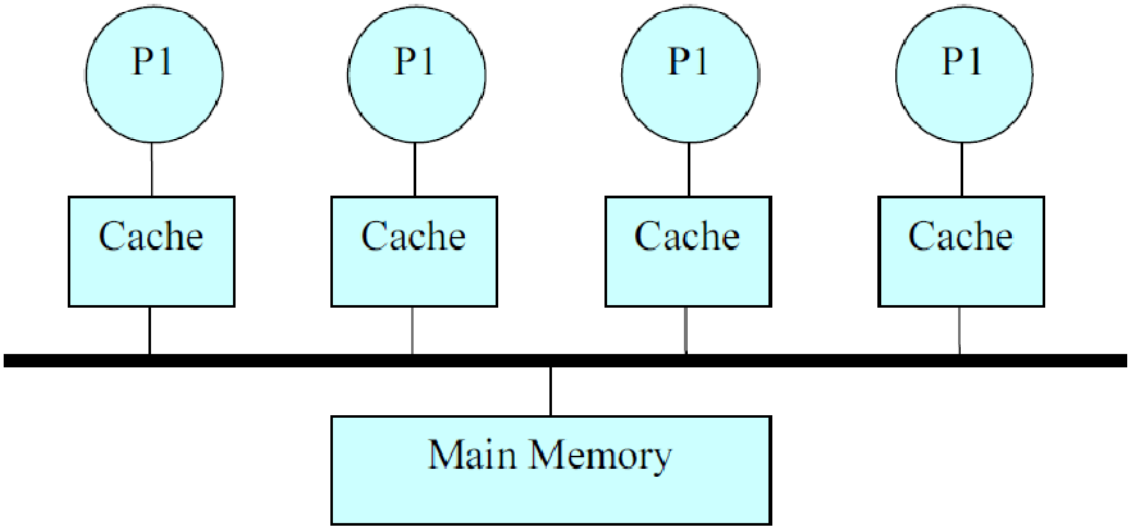

Figure1. Homogenous SMP system consists of four processors, each connected to a private L1 cache. All Caches are connected to Main Memory through a shared bus.

Your job in this project is to instantiate these peer caches, and to maintain coherence across them by applying, **MSI, MESI, and MOESI** coherence protocols.

Problem

Coding:

For this problem, you are supposed to implement MSI, MESI, and MOESI coherence protocols and to match the validation runs given to you exactly. The output will have different statistics as mentioned below. Your simulator should accept multiple arguments that specify different attributes of the multiprocessor system through command line. One of these attributes is the coherence protocol 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.

For the MSI protocol, assume that there is no BusUpgr and no cache to cache transfer. For MESI, and MOESI, follow the algorithms explained in Chapter 8 of text book.

Output of the simulator:

Your simulator should output the following statistics on the standard output (i.e. use printf or cout). Your output should match the given Validation runs or marks will be deducted

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. Number of dirty cache blocks written back to the main memory.
6. Number of transitions from invalid state to exclusive state.
7. Number of transitions from invalid state to shared state.
8. Number of transitions from modified to shared state.
9. Number of transitions from exclusive state to shared state.
10. Number of transitions from shared state to modified state.
11. Number of transitions from invalid state to modified state.
12. Number of transitions from exclusive state to modified state.
13. Number of transitions from owned state to modified state.
14. Number of transitions from modified state to owned state.
15. Number of cache-to-cache transfers from the requestor cache perspective (i.e. how many lines this cache has received from other peer caches)
16. Number of interventions (see chapter 8.)
17. Number of invalidations (any state => Invalid)
18. Number of flushes to the main memory.

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
- Cache policies: use write back and write allocate policy, and use LRU replacement policy for all cases (both are already implemented in the provided code).

In your report, answer the following questions. You should provide a detailed description and tabular/graphical representation of results with an insightful discussion to earn full credit.

- Question 1: For MESI, how does the number of cache misses and invalidations change as you increase the cache block size? What are the reasons behind this?
- Question 2: An invalidation is not necessarily harmful if it is not followed by a cache miss to the block that was invalidated. Hence, the number of coherence misses is more representative to

performance compared to the number of invalidations. Collect the number of coherence misses, and observe how they are affected as the cache size increases. Then, explain possible reasons for your observations.

- Question 3: Compare MSI vs. MESI in terms of the total number of bus transactions. The Exclusive state allows MESI to have fewer bus transactions. However, does the reduction vary with cache block size and cache size?
- Question 4: Compare MESI vs. MOESI in terms of the total number of flushes to the main memory. In general, MOESI should reduce the total number of flushes because it allows dirty sharing. However, how does the reduction affected by the cache size? Make observation and explain possible reasons for your observation.

Getting Started:

• In order to "make" your simulator, you need to execute the following command:

```
make clean; make;
```

• After making successfully, it should print out the following:

```
-----------------------------------------------------

-------------FALL11-506 SMP SIMULATOR (SMP_CACHE) ------

-----------------------------------------------------

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

• An executable called "smp_cache" should be created.

• In order to run your simulator, you need to execute the following command:

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

Where:

- smp_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 (0: MSI, 1:MESI, 2:MOESI)
- trace_file: The input file that has the multi threaded workload trace.

After you download and unzip the file named MP2.zip, you will find the following files and directories:

1. cache.cc
2. cache.h
3. main.cc
4. Makefile
5. Validation_runs

The directory "Validation_runs", contains all runs you are asked to match, you have to literally match the output files, since we are going to use "diff" in order to spot out any differences. If your output does not match the given validations in terms of results and formats, you will be deducted points. You can redirect the output of your program to a file using '>' operator

You can use the following command to check if your output matches the given validation runs:

```
diff –iw <your_file> <validation_run>
```

*****Follow the same format specified in the validation runs*****

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

```
processor(0–7) 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" in 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).

Submission:

You are supposed to submit the following files:

1. main.cc
2. cache.cc
3. cache.h
4. Makefile
5. Report (preferably pdf)

You need to zip all of the mentioned files in one zipped folder named as mp2.zip

Using the following command:

```
zip mp2.zip main.cc cache.cc cache.h Makefile report.pdf
```

Make sure you submit this zip file with the correct filename and without a folder included in the zip file, otherwise you will have points deducted even if your runs are matched.

This is due to having a script for grading your programs!

If you are working in a group, please mention your name, section and unity ID clearly.

Implementation suggestions:

1. Read the given code carefully (cache.cc, cache.h), 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. In cache.cc, there is a function called Cache::Access() , this function represents the entry point to the cache module, you might need to call this function from the main, and to pass any required parameters to it.
4. You might create an array of caches, based on the number of processors used in the system.
5. In cache.h, you might need to define new functions, counters, or any protocol specific states and variables.
6. Start early and post your questions on the message board. A forum is available especially for MP2. Please use it.

C: Reducing Off-Chip Bandwidth Usage – Enhancement Portion (Only for 506 students)

This is a follow up to Problem 1. To the generic 4-way bus based multiprocessor that we assumed in Problem 1, we are going to additionally assume that all the four processors are implemented as four cores on a single chip, as shown in Figure 2.

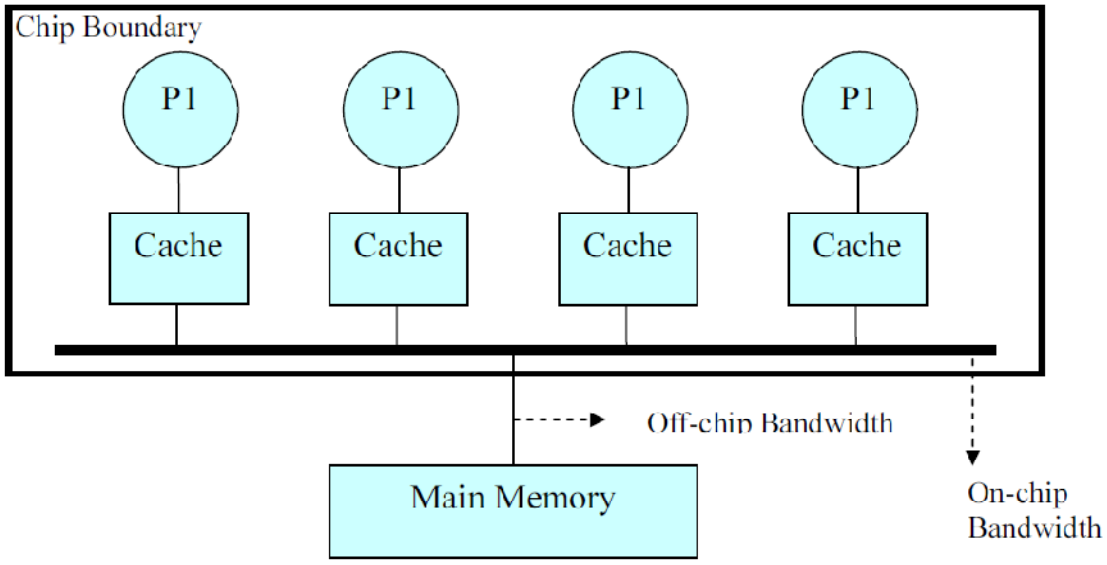

Figure 2. SMP implemented as 4-core chip.

Recent trends have shown that on-chip bandwidth is quite plentiful due to the large number of metal layers that can be used to provide on-chip interconnection. However, off-chip bandwidth is severely limited by the number of pins that can be placed on a chip, and by power consumption that comes from the number of pins used. It is projected that the growth in the number of cores will greatly outpace the availability of off-chip bandwidth. Hence, an important goal is to reduce the amount of off-chip bandwidth usage.

Assume that the execution time of a program is linearly related with the number of cache misses the program suffers from. Let M denote the number of cache misses in your system, and B is the bandwidth (total number of bytes due to cache misses, write backs, and flushes to main memory) used in your system. Your goal is to minimize the product of the two metrics, in other words B*M, by figuring out cache configuration and cache coherence protocol to use. To figure out B, you must account for command bus traffic (assume 8 bytes/64 bits per command to communicate address to the bus), and data bus traffic (your cache block size times the number of write backs, flushes to main memory, and cache misses).

Report:

In your report, show results and answer the following questions. You should provide a detailed description with an insightful discussion to earn full credit.

- For a 256KB cache with 64B block size, which protocol (MSI, MESI, and MOESI) produces the lowest B*M? Explain why.
- Question 2: Find a cache configuration and modify any of the above protocol to improve B*M. You are restricted to 1MB of cache size and 8-way associativity. You can only modify the cache block size and the coherence protocol. Your answer should not introduce new predictor structure, or use prefetching. But you are allowed to modify the cache coherence protocol such as adding new states, changing the state transition diagrams, introducing new bus transactions, etc. You are allowed to add at most 4 new states (if necessary). Explain your approach & readings. Also include the state diagram of the enhanced protocol.

Include this part in the main report. You are supposed to submit only one report

Grading Policy:

Grade will be distributed as follows:

- 20%: MSI
- 20%: MESI
- 20%: MOESI
- 15%: Three secret runs (one for each protocol)
- 10%: Enhancement (Reducing off-chip bandwidth)

- 15%: Report

**** For 492 students the 10% for enhancement will be included in report**

****If your output does not --exactly-- match any validation run, you will be given at most 10% for that part. (Instead of 20%)**

****If your output does not match the secret runs, you will get 0% for that part.**

****In order to get full credit for the report, you need to provide detailed discussion and performance evaluations results.**