CSC/ECE 506: Computer Architecture and Multiprocessing
Program 4: Simulating DSM Coherence
Due: Friday, April 20, 2022

1. Overall Problem Description

In this project, you will add new features to a trace-driven DSM (distributed shared memory) simulator. You are provided with superclass cache.cc along with main.cc. You need to subclass the directory to create a full bit-vector directory. The skeleton of directory.cc is provided and the relevant methods need to be implemented. Based on this, Implement the MESI protocol as a subclass of cache.cc. Then, subclass directory.cc again to create a Simplified Scalable Coherent Interface (SSCI) directory. Your project should build on a Linux machine. The purpose of this project is to give you an understanding of DSM architecture, evaluate it and interpret performance data. The most challenging part of this project is to understand how caches, coherence protocols and directories are implemented and handled in a DSM environment. Once you understand this, the rest of the assignment should be straightforward.

2. Simulator

The specifications of the DSM system are as follows

- The system will consist of 16 nodes. There will be a single-level cache for all the nodes, which follow the MESI invalidation coherence protocol. All the caches will have the same configuration, which will be given as input parameters to the simulator. This part is the same as in Program 3.
- In a DSM system, the memory is spread across all the nodes in the system. However, for the simulation, you don't have to take the memory organization into consideration. You only need to handle a directory structure along with the caches.
- In a real DSM system, the directory is also distributed among the nodes. However, in this simulation, for simplicity we will use a single directory structure. In operations on the directory, the processor number will be passed as a parameter.
- The directory information will be stored either as a full bit-vector or as a linked list. The directory structure will have a limited size, equal to the total number of the blocks that can be cached. e.g. if the cache size is 32768 (32KB) and block size is 64 , then the total number of blocks that can be cached at one processor is

$$\frac{32768}{64} = 512$$

  Therefore, the total number of directory entries in the system is 512 × 16 = 8192. This format is known as a sparse directory format, which is described in the textbook. In Lecture 21, we called it "reducing the height" of the directory.

How to build the simulator

You are provided with a working program for a 16-processor system, which implements a coherence protocol but not the directory. The Cache class implements what is common to each class, and your MESI class implements functionality that is unique to the MESI protocol. You are given all the methods for the full implementation of the MESI protocol. You should study these methods because you might be asked about them on the final exam.

You will need to build a directory structure. It will contain the directory entries for different blocks. The dir_entry class in directory.cc is an interface class. The functions declared here are all virtual. You need to implement these functions as part of derived classes fbv.cc and ssci.cc. The fbv.cc expects you to implement a full bit-vector, while ssci.cc will use a linked-list implementation. Each entry in fbv.cc and ssci.cc will consist of a tag to identify the block, the cache state of the block and a full bit-vector or a

linked list, as described in the lecture and textbook. You need to implement the methods below in directory.cc

- `dir_entry *find_dir_line(ulong tag)`: This will return the directory line if present. Else it will return NULL.
- `dir_entry *find_empty_line(ulong tag)`: This will traverse the directory based on its size and return an uncached line.

One way to implement a full bit vector is using an array of 16 chars or Booleans (if using C++). The tag will consist of the address bits – the block offset bits. So, if the block size is 64 bytes, there are 6 block offset bits. Hence the tag will 32 – 6 = 26 MSB bits of the address

For every memory reference, you will first check whether it is a hit or miss in the cache. Depending on this, access the directory, modify the state of blocks in peer cache, modify the directory entry, and then assign an updated state in the cache of the requesting processor. This way you'll have handled all the transactions explained in the notes.

In case a directory entry for the block is not found in the directory structure, then reuse the directory entry of an unused block. You will need to notify the directory when a block is evicted from any cache. When a block no longer exists in any of the caches, the full bit-vector becomes 0000 0000 0000 0000, whereas the linked list will not have any nodes. When this occurs, mark the entry as U (uncached). At the start of the simulation, all the entries will be uncached. You need to implement the methods below in both fbv.cc and ssci.cc.

- `void add_sharer_entry(int proc_num)`: This caches an entry in directory for a given address by updating the FBV or linked list. This has been provided in the files.
- `void remove_sharer_entry(int proc_num)`: This uncaches a directory entry for a given address by deleting the node in the linked list or by resetting the corresponding bit in the bit-vector.
- `int is_cached(int np)`: This checks whether the given entry is cached in the directory.
- `void sendInt_to_sharer(ulong addr, int num_ proc , int proc_num)`: This function sends the intervention to all the sharers of the block.
- `void sendInv_to_sharer(ulong addr, int num_proc , int proc_num)`: This function sends the invalidation to all the sharers of the block. **(This is provided)**

When a cache access results in a miss and a block needs to be evicted, perform the eviction before caching the block, as the eviction will change the directory contents.

You are also 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 in the hierarchy). The `main()` method is simpler than for Program 3, because there is no bus action that needs to be undertaken after most kinds of cache accesses.

The reference simulator implements write-back, write-allocate (WBWA) and the LRU replacement policy. So, in case you are planning to create or use your own simulator please keep these policies in mind.

3. Getting Started

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:

```
------------------------------------------------------------
-----------SP2022-506 DSM SIMULATOR                    ----------
------------------------------------------------------------
Compilation Done ---> nothing else to make :)
```

An executable called dsm will be created.

In order to run your simulator, you need to execute the following command:
input format: ./dsm ⟨cache_size⟩ ⟨assoc⟩ ⟨block_size⟩ ⟨num_processors⟩ ⟨dir_type⟩ ⟨trace_file⟩

where—

- dsm: Executable of the DSM simulator generated after making
- cache_size: Size of each cache in the system (as in Program 3, all caches are the same size, for example, 65536)
- 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)
- dir_type: 0 – Full bit vector, 1 - SSCI
- trace_file: The input file that has the multithreaded workload trace.

The trace files are at the locations given below.

`/afs/eos.ncsu.edu/lockers/workspace/csc/CSC506-1/trace/swaptions_truncated`
`/afs/eos/lockers/research/csc/traces/x264_trace/TimingCPU_x264_simlarge`

Each trace file consists of a sequence of memory transactions; each transaction consists of three elements:
⟨processor (0-15)⟩ ⟨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`" to its local cache. You need to propagate this request down to cache 3, and cache 3 should take care of it.

The Swaptions trace is about 3.8 GB and the x264 trace is 14 GB. The simulator should take about 10 min. to process the longer trace, and you can run multiple simulations in parallel.

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 `dsm`. You should check that the output of both runs is the same.

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

`sh module.sh ⟨cache_size⟩ ⟨assoc⟩ ⟨block_size⟩ ⟨num_processors⟩ <0> ⟨trace_file⟩ ⟨number_of_references(optional)⟩`

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. Setting, e.g., `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.

4. Report

For this problem, you will experiment with various cache configurations and measure the cache performance of number of processors across the two different trace files mentioned above. 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. (Please note that 1KB is 1024bytes ➔ 128KB is 131072bytes) (Not giving the proper values will result in the deduction of marks)
- Cache associativity: keep it constant at 4-way
- Cache block size: vary from 64B, 128B, and 256B, while keeping the cache size at 1MB

Do all the above experiments for SSCI and FBV using both traces. For each simulation run, collect the following statistics:

1. Number of read transactions the caches have received.
2. Number of read misses the caches have suffered.
3. Number of write transactions the caches have received.
4. Number of write misses the caches have suffered.
5. The total miss rate of the caches.
6. Number of writebacks when lines are replaced (victimized) from a cache.
7. Number of cache-to-cache transfers.
8. Number of `SignalRd`s
9. Number of `SignalRdX`s
10. Number of `SignalUpgr`s
11. Number of invalidations
12. Number of interventions

For your report, you should only print out aggregate statistics (e.g., the total number of read transactions that all 16 caches have received), though you may find it useful to print out statistics on individual caches for debugging purposes.

Present the statistics in tabular format as well as figures in you report. Discuss the trends with respect to change in the configuration of the system. Discuss the various aspects of the coherence protocols. You can discuss the bus traffic, number of memory transactions, and complexity of the protocols.

Answer the following questions, making additional simulation runs as necessary.

- What is the "best" block size for each trace in a 256K cache? In a 512K cache? In a 1M cache?
- Is "high" associativity more important in a small cache or a large cache? How can you tell?

- Does the miss rate continue to fall as the cache size increases? Why or why not? Explain any apparent anomalies in the data.
- How can the number of invalidations rise with increasing cache size?
- Look up the characteristics of the two benchmarks (Swaptions and x264) on the Internet. Can you explain the different behavior (e.g., miss rates, number of invalidations, or changes in them as cache parameters are varied) by referring to the benchmark characteristics? For example, why does one benchmark exhibit a steeper curve on increasing the cache size compared to the other benchmark?

5. Submission

Create a compressed folder named `project4.zip` containing the files—

- code files `main.cc`, `cache.cc`, `mesi.cc`, `directory.cc` and their associated header files. (Do not include any object files)
- the `Makefile` that is provided
- A prose report on your results, including all the statistics as mentioned in Section 4 and named `report.pdf`.

To create the zip file, sue the command `zip project4 *.cc *.h Makefile report.pdf` .
Any deviation from the format mentioned for the files and zip folder will result in deduction of 5 points.

6. Grading

- 15%: Your code compiles successfully
- 25%: Your output matches exactly for runs on all three caches that you are implementing (points will be equally distributed). There will be debug runs provided to you and mystery runs which will be carried out on your submitted code.
- 60%: Report. Credit will be given on the statistics shown and discussion presented.

7. Suggestions

1. Read the main, cache, and directory 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.