Cache Lab

Yehowshua Immanuel and Tushar Krishna

March 7, 2021

Lab Description

The lab files can be downloaded here :assignment.zip

In this lab, you will write a cache simulator that you will use to evaluate and compare the cache hit rates for several cache architectures. **This assignment is designed to be completed in C**. Coding this simulator in C will best prepare you for future classes that use this class as a prerequisite. All starter code we provide you is written in C. You may use C++ as well – you are responsible for any changes necessary (ex: changing the file extension).

Given a cache memory organization, your simulator should read a file containing a memory trace (a sequence of memory addresses) and determine which of the memory references will cause cache hits (and which will cause misses). The program should keep track of the number of hits and misses generated by each cache over the entire trace. For every memory access you will need to update your cache data structures and update any cache statistics (more on this later).

Trace File

Three trace files are available in the Traces folder. Each line has three fields.

- The first field gives the type of the memory reference (2 for instruction fetch, 1 for a store's data memory write, and 0 for a load's data memory read).
- The second field on the line is the address.
- The third field gives the instruction value for a fetch and is always 0 for loads and stores (Not used in this lab).

For e.g., the gcc trace file has 1,000,000 memory references (one per line) and was collected from the gcc while it was running on a MIPS processor.

The Simulator

Your task is to simulate an architecture with a writeback cache –a data cache (D-cache). The cache will be placed between the CPU and memory. The data cache will be used to cache data reads/writes. The cache will use an LRU replacement policy.

When initializing the cache, you will be given a block size, a cache size and an associativity (maximum of 32) – these will always be powers of 2.

For every access your cache makes, you will be provided an address and an access type (mem read, mem write, or instruction read). For every cache access, you will need to update the data structures you are using and the cache statistics to reflect the occurrence of that cache access. The statistics you will need to keep track of are the total accesses, total misses, D-cache accesses, D-cache misses and writebacks.

Simulator Basics

- You only need to implement the cache directory and not the data section. For each cache block you need to keep track of the tag, valid bit, and dirty bit (was the cache line modified?). For each set you need to keep track of the LRU status of the blocks in the set. Dynamically allocate and create the directory data structure.
- Read and process each address from the trace file – read a memory reference, extract the index and tag, determine whether it is a hit or a miss, update the directory structure and any other data structures you add (for example counters keeping track of the number of hits or number of references). Do not forget to update the LRU status.
- You are simulating only one cache configuration at a time.

It is imperative that you implement the simulator on your own.

Assignment submissions will be checked with code similarity analysis tools including against publicly and previously available code bases.

Understanding the Given Code

To help you get started on the lab, some starter code has been provided for you. The code provided to you is as follows:

- cachesim.h: contains some declarations that define data types, structs and functions for the simulator. We have provided some structs for a cache block and a cache set, but feel free to update anything that is not marked "Do not modify".
- cachesim.c: This file contains your main simulator code. This file already contains code to fetch the cache configuration, read the trace file, and run the full cache simulation. Your main job is to complete the cachesim_init and cachesim_access functions. All items are commented, so please check the code for more details on what/where changes are necessary. Please take note of some parts of the code we ask you not to change, as these parts are important for your simulator's outputs to stay standardized.
- lrustack.h: This file contains a potential interface for your LRU stack, with all functions thoroughly documented. Note that you can also ignore our starter code and write the LRU stack from scratch if you so desire.
- lrustack.c: This file contains a potential interface for your LRU stack, with all functions thoroughly documented. Note that you can also ignore our starter code and write the LRU stack from scratch if you so desire.
- lrustacktest.c: This file contains some basic tests for the LRU stack (if you implemented it using the interface we set up). You can add tests if you want to test your LRU stack more thoroughly. You can rewrite the tests completely if you changed the interface. You will not be graded on your LRU Stack tests in this file – this is simply provided to help ensure your LRU stack implementation works correctly before integrating it into the full simulator.

All locations where you will need to make changes are marked with "TODO" comments – feel free to remove these comments when done implementing your cache simulator.

Compiling, Running and Debugging

- Compiling LRU stack test:

```
gcc lrustack.h lrustack.c lrustacktest.c -o lrustacktest
```

- Running LRU stack test:

```
./lrustacktest
```

- Compiling the cache simulator:

```
gcc lrustack.h lrustack.c cachesim.h cachesim.c -o cachesim
```

- Running the cache simulator:

```
./cachesim <trace> <block_size>, <cache_size>,
<associativity>
```

- Expected outputs:

The Tests folder lists the sample outputs for the three traces provided. However, your code will also be evaluated against additional traces.

Make sure all code compiles and runs without issue before making any changes.

A good debugging tool is GDB. You can find a good intro to using GDB at https://www.geeksforgeeks.org/gdb-step-by-step-introduction/

Assignment (100 pts)

Determine hit rates, miss rates, and writeback traffic for various configuration of a **64 Kbyte unified set associative cache** (instruction and data), assuming a cold cache for each trace - the cache is initially empty and all lines are in the invalid state. The cache configuration is a combination of the line size and associativity. **Configuration parameters include: cache block size (32 byte, 64 byte, 128 byte, 256 byte, and 512 byte) and associativity (2, 4, 8)**.

Submission Requirements

- Your code
  - **If you coded in C++**: update the create_submission.sh to zip the files you used – by default, they try to zip the .c and .h files.
  - You can use create_submission.sh to create **ece3058_cachelab_submission.tar.gz**. **It is your responsibility to make sure all files necessary to run the simulator are included in the submission.**
  - Please make sure your code is well documented.
  - **Do not include the trace file or points will be deducted.**

Grading Guidelines

IMPORTANT NOTES:

1. If your code does not compile your assignment cannot be graded.
2. If your code crashes/seg. faults or produces unreadable output it cannot be graded.
3. Your code will be graded according to a randomly selected combination of parameters (cache size, line size, associativities and trace files). Its accuracy will be compared against our solution simulator.

GRADING RUBRIC:

Program compiles and executes	25 points
Results are largely correct, but answers may be *slightly* incorrect	60 points
Results are precisely correct	70 points
Code is documented	5 points

Program compiles and executes	25 points
TOTAL	**100 points**

Note: You must achieve a minimum average of 50% in the lab assignments to pass the course.

Extra Credit (50 points)

This portion will only be graded once you have completed the above parts of the assignment.**

Extend your simulator to

1. Have a separate 16 Kbyte direct mapped I-cache and 16 Kbyte direct mapped D-Cache each with 64 byte lines.
2. Include a 256 Kbyte, K-way set associative unified L2 cache with a line size of 64 bytes and a write-back update policy. The L2 cache should inclusive (details below), meaning all the data in the L1 cache is also in the L2 cache (keep in mind what this means for the associativity/number of sets of the L2 cache with respect to the L1 cache).
3. Consider the configuration of the L2 with 64 byte lines and associativity 8
   a. Compute the local miss rates and the global miss rates for each trace for instructions and data.
   b. What value of associativity minimizes the global miss rate for data.
   c. What is the total volume of traffic between the L1 D-cache and the L2 for each trace.
4. Write your own test cases for your L2 Cache, and be prepared to demo how you test case fully showcases the benefits of an additional L2 Cache.

Inclusive L2

In the case of multilevel caches (assume two-level -L1 and L2), if the contents of L1 are always contained in L2, caching is said to be inclusive.

Read or Write Misses

- If the block is not found in either L1 or L2, then it is fetched from the main memory and placed in *both* L1 and L2.

Read Hits

- Suppose there is a read request for block (i.e., line) X. If the block is found in L1 cache (hit), then the data is read from L1 cache and returned to the processor. If the block is not found in the L1 cache (L1 miss), but present in the L2 cache (L2 hit), then the cache block is fetched from the L2 cache and placed in L1.

Write Hits

- When you write to a line in L1 (now dirty), you don't need to worry about updating L2 till this line is evicted from L1 due to replacement. In this case, you will update L2 (note - this won't be a writeback since you aren't updating memory). If you miss in L1 but find the block in L2, you will bring it into L1 and update it at L1.

Evictions

- If a block is evicted from L1, there is no involvement of L2.

- If a block is evicted from L2, the L2 cache sends a back invalidation to the L1 cache, so that inclusion is not violated. When an eviction in L2 causes a dirty L1 line to be invalidated, in this case, you will update the memory and increment writebacks.

Traffic

- The traffic between L1 and L2 constitutes (i) data is brought from L2 into L1 on a L1 miss but L2 hit, (ii) invalidations sent to L1 when a line is evicted from L2, (iii) updates to L2 due to a dirty line being evicted from L1.

Breaking it Down Into Bite-Sized Pieces

1. Read the instructions again.
2. Brainstorm and Design: Spend time thinking through the design, i.e., the directory data structure and all the counters you will need to record the requested information. Specify the data structures you need for the tag directory and LRU stacks. It is highly recommended that you draw the data structures and mentally walk through the processing of a reference. Do this analysis before you ever write a line of code. It will save you a lot of time and stress. Most people spend time debugging and hacking away at a piece of code, which was written before they thought through the details.
3. Create your own traces of addresses rather than starting with the original traces. Create sequences for which you know the behavior. For example, create a sequence of 100 identical addresses – you will have one miss and 99 hits. Another test case is a repeating sequence of addresses that will conflict in the cache – for example two addresses with the same index but different tag and a direct mapped cache. You know the answer to these cases and can easily check the results for correctness.
4. Parse Address: Write a test program to just be able to read the trace and parse the address into tag, and set. Make sure you can read the trace correctly and print the set index and tag. Once you are sure of this, then you can focus on the cache data structures.
5. LRU Replacement: You will need some form of LRU replacement in your simulator. Coding this is often one of the harder parts of this lab. In the starter code, we have isolated the LRU code into lrustack.h/lrustack.c, so you can test your LRU implementation independent of the cache. The file lrustack.h/lrustack.c defines a potential interface to interact with an LRU stack. If you implement LRU according to the provided comments, you will be able to sanity check it against some basic tests we provide in lrustacktest.c. Note, these tests may not be exhaustive, but they are a good sanity check to ensure your LRU implementation is correct. If you want more exhaustive tests, feel free to add them! **If you want to implement an LRU stack your own way, feel free to do that as well!** We still recommend testing the LRU stack separately before proceeding on with the rest of the simulator.
6. Cache simulator initialization: Write the function to initialize your cache simulator.
7. Cache access: Write the function to run a cache access.
8. Partial Verification: Run the simulator with your traces and verify it behaves as expected.
9. Full Verification: Now run the full traces. We will provide you with statistics the TA implementation gets for certain configurations so you can verify your cache simulator is working.
10. Experiments and Report: See above instructions.

Thinking through the design of your simulator up front should minimize the time it takes you to complete the assignment. As mentioned earlier, debugging time is typically the biggest component of projects. Some clear up-front thinking will save you much of that time. The actual code size is not very large for a C program.

START EARLY and utilize recitation/office hours!