CS154 Project 3: Caching
Due: 11:59pm Tuesday Nov 15, 2022

1 Overview

This project will help you better understand caching. In this project you will write (in `csim.c`) a small C program (about 250-350 lines) that simulates the behavior of a cache memory.

Doing an `svn update` in your `CNETID-cs154-aut-22` checkout should create a new directory `p3cache`. This contains all the files you need for Project 3. We will be grading your work according to the modifications of the file `csim.c` (and no other files) that are committed prior to the deadline.

The file you check in will be graded by the staff's copy of the `driver.py` script included in your `p3cache` directory. The grade produced by this script will be your grade. **If the script fails, or your code fails to compile, you will receive no credit.**

(If you work on Mac, you can use `driver-mac.py` to test your code while you are working on it. But note that **we will grade your work on linux machine** (linux.cs.uchicago.edu). So when you finish your work, **do test it on linux machine using** `driver-mac.py`**.** If you get different grades on mac and linux, please report to TA.)

2 Project: Writing a Cache Simulator

2.1 Related Textbook Material

This project is highly related to your *textbook Section 6.3 and 6.4*. We **strongly recommend** you read them before starting implementing this project.

2.2 Reference Trace Files

This project involves a cache *simulator*: we are not studying the utilization of the real cache on the computer's CPU, but of a simple made-up cache that is implemented in software. However, the cache utilization is assessed according to the sequence of memory references from real programs, which are "replayed against" the cache simulator. The sequence of memory references are stored in *reference trace files*, which are contained in the `traces` subdirectory. We use these to evaluate the correctness of the cache simulator you write in this project. The trace files are generated by a Linux program called `valgrind`. For example, typing

```
linux> valgrind --log-fd=1 --tool=lackey -v --trace-mem=yes ls -l
```

on the command line runs the executable program "`ls -l`", captures a trace of each of its memory accesses in the order they occur, and prints them on `stdout`.

`Valgrind` memory traces have the following form:

```
I 0400d7d4,8
 M 0421c7f0,4
 L 04f6b868,8
 S 7ff0005c8,8
```

Each line denotes one or two memory accesses. The format of each line is

```
[space]operation address,size
```

The *operation* field denotes the type of memory access: "I" denotes an instruction load, "L" a data load, "S" a data store, and "M" a data modify (i.e., a data load followed by a data store). There is never a space before each "I". There is always a space before each "M", "L", and "S". The *address* field specifies a 64-bit hexadecimal memory address. The *size* field specifies the number of bytes accessed by the operation.

2.3 Description

In this project, you will write a cache simulator in `csim.c` that takes a `valgrind` memory trace as input, simulates the hit/miss behavior of a cache memory (with different E, B, and S parameters) on this trace, and outputs the total number of hits, misses, and evictions. For this project, use the allocate-on-write policy to handle write misses in your implementation of the simulator.

We have provided you with the binary executable of a *reference cache simulator*, called `csim-ref`, that simulates the behavior of a cache with arbitrary size and associativity on a `valgrind` trace file. It uses the LRU (least-recently used) replacement policy when choosing which cache line to evict.

The reference simulator takes the following command-line arguments:

```
Usage: ./csim-ref [-hv] -s <s> -E <E> -b <b> -t <tracefile>
```

- `-h`: Optional help flag that prints usage info
- `-v`: Optional verbose flag that displays trace info
- `-s <s>`: Number of set index bits ($S = 2^s$ is the number of sets)
- `-E <E>`: Associativity (number of lines per set)
- `-b <b>`: Number of block bits ($B = 2^b$ is the block size)
- `-t <tracefile>`: Name of the `valgrind` trace to replay

The command-line arguments are based on the notation (s, E, and b) from page 597 of the CS:APP2e textbook. For example:

```
linux> ./csim-ref -s 4 -E 1 -b 4 -t traces/yi.trace
hits:4 misses:5 evictions:3
```

The same example in verbose mode:

```
linux> ./csim-ref -v -s 4 -E 1 -b 4 -t traces/yi.trace
L 10,1 miss
M 20,1 miss hit
L 22,1 hit
S 18,1 hit
L 110,1 miss eviction
L 210,1 miss eviction
M 12,1 miss eviction hit
hits:4 misses:5 evictions:3
```

Your job for this project is to fill in the given `csim.c` file so that it produces the identical output as the reference simulator. This will consist of implementing the `simulate` function that opens and parses each line of whatever trace file is given with the `-t` option, and tracks what that memory operation would do within the context of the cache that you are simulating. Your simulator will consist of new global variables to represent the state of the simulator, and functions (called by `simulate`) that operate on the simulator in response to the memory operations read from the trace file.

2.4 Programming Rules

- Your `csim.c` file must compile without warnings in order to receive full credit.
- Your simulator must work correctly for arbitrary s, E, and b. This means that you will need to allocate storage for your simulator's data structures using the `malloc` function.
- For this project, we are interested only in data cache performance, so your simulator should ignore all instruction cache accesses (lines starting with "I"). Recall that `valgrind` always puts "I" in the first column (with no preceding space), and "M", "L", and "S" in the second column (with a preceding space). This may help you parse the trace, although there are many different legitimate ways to do so.
- To receive credit for this project, you must (as it is now in `csim.c`) call the function `printSummary`, with the total number of hits, misses, and evictions, at the end of your `main` function:

  ```
  printSummary(hitCount, missCount, evictionCount);
  ```

- For this project, you should assume that memory accesses are aligned properly, such that a single memory access never crosses block boundaries. By making this assumption, you can ignore the request sizes in the `valgrind` traces.

2.5 Evaluation (Max Score = 100 points)

For this project, we will run your cache simulator using different cache parameters and traces. There are ten test cases, the first 6 are worth 8 points each, and the last 4 are worth 13 points each:

```
linux> ./csim -s 1 -E 1 -b 1 -t traces/yi2.trace
linux> ./csim -s 4 -E 2 -b 4 -t traces/yi.trace
linux> ./csim -s 2 -E 1 -b 4 -t traces/dave.trace
linux> ./csim -s 2 -E 1 -b 3 -t traces/trans.trace
linux> ./csim -s 2 -E 2 -b 3 -t traces/trans.trace
linux> ./csim -s 2 -E 4 -b 3 -t traces/trans.trace
linux> ./csim -s 5 -E 1 -b 5 -t traces/trans.trace
linux> ./csim -s 5 -E 1 -b 5 -t traces/long.trace
linux> ./csim -s 4 -E 4 -b 5 -t traces/long.trace
linux> ./csim -s 1 -E 8 -b 8 -t traces/long.trace
```

You can use the reference simulator `csim-ref` to obtain the correct answer for each of these test cases. During debugging, use the `-v` option for a detailed record of each hit and miss.

For each test case, outputting the correct number of cache hits, misses, and evictions will give you full credit for that test case. For each of the first 6 tests, 4 (out of 8) points are awarded for the correct number of hits, and 2 point each for the correct number of misses and evictions, respectively. For the last 4 tests, 5 (out of 13) points are awarded for the correct number of hits, and 4 point each for the correct number of misses and evictions, respectively. For example, here a particular test case is worth 13 points, and if your simulator outputs the correct number of hits and misses, but reports the wrong number of evictions, then you will earn 9 points.

2.6 Hints and Suggestions

We have provided you with an autograding program, called `test-csim`, that tests the correctness of your cache simulator on the reference traces. Be sure to compile your simulator before running the test:

```
linux> make
linux> ./test-csim
                        Your simulator     Reference simulator
Points (s,E,b)    Hits  Misses  Evicts    Hits  Misses  Evicts
     8 (1,1,1)       9       8       6       9       8       6  traces/yi2.trace
     8 (4,2,4)       4       5       2       4       5       2  traces/yi.trace
     8 (2,1,4)       2       3       1       2       3       1  traces/dave.trace
     8 (2,1,3)     167      71      67     167      71      67  traces/trans.trace
     8 (2,2,3)     201      37      29     201      37      29  traces/trans.trace
     8 (2,4,3)     212      26      10     212      26      10  traces/trans.trace
    13 (5,1,5)     231       7       0     231       7       0  traces/trans.trace
    13 (5,1,5)  265189   21775   21743  265189   21775   21743  traces/long.trace
    13 (4,4,5)  268525   18439   18375  268525   18439   18375  traces/long.trace
    13 (1,8,8)  272531   14433   14417  272531   14433   14417  traces/long.trace
```

```
    100

TEST_CSIM_RESULTS=100
```

For each test, it shows the number of points you earned, the cache parameters, the input trace file, and a comparison of the results from your simulator and the reference simulator.

Here are some hints and suggestions for working on this project:

- Do your initial debugging on the small traces, such as `traces/dave.trace`.
- The reference simulator takes an optional `-v` argument that enables verbose output, displaying the hits, misses, and evictions that occur as a result of each memory access. You are not required to implement this feature in your `csim.c` code, but we strongly recommend that you do so. It will help you debug by allowing you to directly compare the behavior of your simulator with the reference simulator on the reference trace files.
- Each data load (L) or store (S) operation can cause at most one cache miss. The data modify operation (M) is treated as a load followed by a store to the same address. Thus, an M operation can result in two cache hits, or a miss and a hit plus a possible eviction.
- Use svn. When you get something working, check in that version (you can always back up to it later). Never add to a version that works without first committing your code to the repo.

2.7 Grading

We have provided you with a *driver program*, called `./driver.py`, that performs a complete evaluation of your simulator and transpose code. This is the same program your instructor uses to evaluate your code for this project. The driver uses `test-csim` to evaluate your simulator. Then it prints a summary of your results and the points you have earned. To run the driver, type:

```
linux> ./driver.py
```

The output of our copy of driver.py will be your grade. If it does not work, or your code fails to compile, you will receive no credit. Test before you commit your final version.

3 Acknowledgements

This assignment was created by the text book authors and their TAs. It has been modified by the CS154 instructors.

Note that before remote learning, p3 consists of two parts: (a) Writing a cache simulator, and (b) Optimizing the Matrix transpose. To reduce project load in remote learning, we have removed part (b).