The Cache Lab

Overview

For this lab you will implement a cache simulator in C++.  By doing so, you will develop a better understanding of how a cache is organized and accessed and improve your C++ programming skills.

Logistics

You have the option of working alone or working with someone else on this lab assignment.  If you want to work with someone else then figure out who that will be before continuing.  You cannot add a person later on.

After you've found a teammate or decided to work alone, use a browser to log into github and then open up this page:

<https://classroom.github.com/g/gcvDnJg7>

Both you and your teammate (if you have one) will visit this URL.  One of you will create the team.  This process will create a repository for you that you will need to clone on the student2 machine.

Reference Trace Files

The  traces subdirectory of the handout directory contains a collection of reference trace files that we will use to evaluate the correctness of the cache simulator you write. The trace files are generated by a Linux program called valgrind.  For example, typing

% 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, thus two sequential accesses to cache).  There is never a space before an "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.  Your simulator will ignore lines that begin with an I (instruction addresses) and the size field.

Writing a Cache Simulator

You will write a cache simulator in C++ that takes a valgrind memory trace as input, simulates the hit/miss behavior of a cache memory on this trace, and outputs the total number of hits, misses, and evictions.  
  
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 [-h] [-v] -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 616-617 of the textbook. the -h and -v arguments are optional, which is why they are in brackets. For example:

> ./csim-ref -s 4 -E 1 -b 4 -t traces/yi.trace

hits:4 misses:5 evictions:3

The same example in verbose mode:

> ./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 is to implement a cache simulator so that it takes the same command line arguments and produces the identical output as the reference simulator. The provided .C files contain some starting code, but most of the work is left for you to complete. 

Programming Rules

1. Your code must compile without warnings in order to receive full credit. It also must be free of memory errors. Specifically,

valgrind --tool=memcheck --leak-check=full ./csim -v -s 4 -E 1 -b 4 -t traces/yi.trace

must output 0 errors.  You'll need to add destructors to delete dynamically allocated data.

1. Your simulator must work correctly for arbitrary s, E, and b. This means that you will need to allocate storage for your cache's lines using the new function.
2. For this lab, 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.
3. Do not change the call to printSummary. This output is used to check the correctness of your code.
4. For this lab, 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; one access will be sufficient for getting all data bytes.
5. Be sure to do error checking on the command line arguments.  If an error is found, call the printUsage function.
6. You'll need a get bits function that takes a starting bit number (between 0 and 63), an ending bit number (between 0 and 63) and an unsigned long (uint64\_t) and returns the bits in that range.  The bit endianness is little endian (low order bit is bit 0).   Be sure to do error checking on the parameters by using the assert function.  Your get bits function will be used to grab the set index and the tag.
7. Keep the tags in a particular set in LRU order such that lines[setNum][0].tag is the most recently used tag and lines[setNum][associativity - 1].tag is the least recently used tag (assuming the set is full).  You'll need to update the order after each access (both hits and misses).
8. Don't use global variables.
9. Do your initial debugging on the small traces, such as traces/dave.trace.
10. Your simulator and the reference simulator take an optional -v argument that enables verbose output, displaying the hits, misses, and evictions that occur as a result of each memory access.  This will help you debug by allowing you to directly compare the behavior of your simulator with the reference simulator on the reference trace files.  You need to implement this in your code so that the "-v" option will cause your simulator to produce the same output as csim-ref with the "-v" option.
11. You can use the getopt  function to parse your command line arguments.  See man 3 getopt for details or use google.
12. 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.
13. Your methods should be small and single-purpose.
14. Be sure to nicely format and document your code.

Evaluation

We will run your cache simulator using different cache parameters and traces.  There are eight test cases of varying difficulty.

> ./csim -s 1 -E 1 -b 1 -t traces/yi2.trace  
> ./csim -s 4 -E 2 -b 4 -t traces/yi.trace  
> ./csim -s 2 -E 1 -b 4 -t traces/dave.trace  
> ./csim -s 2 -E 1 -b 3 -t traces/trans.trace  
> ./csim -s 2 -E 2 -b 3 -t traces/trans.trace  
> ./csim -s 2 -E 4 -b 3 -t traces/trans.trace  
> ./csim -s 5 -E 1 -b 5 -t traces/trans.trace  
> ./csim -s 5 -E 1 -b 5 -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. Each of your reported number of hits, misses and evictions is worth 1/3 of the credit for that test case.  That is, if a particular test case is worth 9 points, and your simulator outputs the correct number of hits and misses, but reports the wrong number of evictions, then you will earn 6 points.

We have provided you with an autograding program, called test-csim:, that tests the correctness of your cache simulator on the reference traces. 

> ./test-csim

                        Your simulator     Reference simulator

Points (s,E,b)    Hits  Misses  Evicts    Hits  Misses  Evicts

     9 (1,1,1)       9       8       6       9       8       6  traces/yi2.trace

    12 (4,2,4)       4       5       2       4       5       2  traces/yi.trace

     9 (2,1,4)       2       3       1       2       3       1  traces/dave.trace

     9 (2,1,3)     167      71      67     167      71      67  traces/trans.trace

    12 (2,2,3)     201      37      29     201      37      29  traces/trans.trace

    12 (2,4,3)     212      26      10     212      26      10  traces/trans.trace

    12 (5,1,5)     231       7       0     231       7       0  traces/trans.trace

    15 (5,1,5)  265189   21775   21743  265189   21775   21743  traces/long.trace

    90

TEST\_CSIM\_RESULTS=90

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.

The lab is worth 100 points:

10 points for design and documentation

90 points for correctness