CS 105
Cache Lab: Understanding Cache Memories
30 Points

1 Logistics

You must run this lab on a 64-bit x86-64 machine. It will be tested on Wilkes, so you should make sure that it runs correctly there.

2 Overview

This lab will help you understand the impact that cache memories can have on the performance of your programs. You will write a small C program (about 200-300 lines) that *simulates* the behavior of a cache memory. Your cache simulator will read from a *trace file* that contains a sequence of memory address accesses, and will print the number of hits, misses, and evictions that would have occurred for that sequence of memory accesses. In addition to a trace file, your simulator accepts arguments that give the cache configuration configuration, namely, the number of set index bits, lines per set (associativity), and number of block bits. Section 4 of this writeup provides further details on writing the simulator.

IMPORTANT: Read this whole handout *completely* before starting work on the lab! There are useful hints throughout the handout.

Sections 5 and 6 give a summary of the evaluation for the lab and describe how to submit your work.

3 Downloading the assignment

This is a pair programming project that must be able to run on Wilkes.

The files needed for the lab are held in a tar archive named `cachelab-handout.tar`, which is linked from the Web page for this assignment. Start by downloading `cachelab-handout.tar` and copying it to a protected Linux directory in which you plan to do your work. Then give the command

```
linux> tar xvf cachelab-handout.tar
```

This will create a directory called `cachelab-handout` that contains a number of files.

3.1 Summary of Files

For this lab you will be modifying one file: `csim.c`. The file can be compiled by typing "`make`".

WARNING: Do not let the Windows WinZip program open up your `.tar` file (many Web browsers are set to do this automatically). Instead, save the file to your Linux directory and use the Linux `tar` program to extract the files. In general, for this class you should NEVER use any platform other than Linux to modify your files. Doing so can cause loss of data (and important work!).

These are the other files you will use, which are further described as needed throughout this writeup:

- The `traces` subdirectory contains a collection of trace files.
- `csim-ref` is a reference implementation of the cache simulator.
- `test-csim` is a testing program for your cache simulator.
- `driver.py` is a testing program (it runs `test-csim`).

(There are a few other helper files as well, e.g., a `Makefile`. See the `README` for more.)

3.2 Reference Trace Files

To test and debug the cache simulator you will build, you can run it using the *reference trace files* in the `traces` subdirectory; these same trace files will be used to evaluate the correctness of your simulator. Each trace file describes the memory references made when a program was run. They 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.

4 Writing a Cache Simulator

In this lab 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 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 [-hv] -s <s> -E <E> -b <b> -t <tracefile>
```

- `-h`: Optional help flag that prints usage information
- `-v`: Optional verbose flag that displays trace information
- `-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) discussed in class and on pages 615–617 of the CS:APP3e textbook (see especially Figure 6.26). 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 is to fill in the `csim.c` file so that it takes the same command-line arguments and produces output identical to the reference simulator. Notice that this file is almost completely empty. You'll need to write it from scratch.

Programming Rules

- Include both partners' names and login IDs in the header comment for `csim.c`.
- Your `csim.c` file must compile without warnings on Wilkes in order to receive 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. Type "man malloc" for information about this function.
- To receive credit, you must call the function `printSummary`, with the total number of hits, misses, and evictions, at the end of your `main` function:

```
printSummary(hit_count, miss_count, eviction_count);
```

Tips for Working on the Lab

Here is a list of suggestions for working on the lab; afterward are details about the testing program we have provided:

- Your cache simulator is only simulating performance with respect to hits, misses, and evictions, so each cache line *does not need to store a block of data!*
- As mentioned in the rules, you should ignore all instruction cache accesses in the trace (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 fact may help you parse the trace.
- 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.
- 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**.
- We recommend that you use the `getopt` function to parse your command line arguments. You'll need the following header files (see "`man 3 getopt`" for details):

```
#include <getopt.h>
#include <stdlib.h>
#include <unistd.h>
```

- The `fscanf` function will be useful for reading from the trace file. Since `fscanf` is complicated, you shouldn't try to understand it completely. Instead, concentrate on the `%c`, `%Lx`, and `%d` format options. Note that if you place a space character before `%c`, you can disregard valgrind's habit of putting a blank before the L, S, and M characters.

- Remember: addresses in the trace files are 64-bit **hexadecimal**. If you use `fscanf`, the format specifier you should use is `%llx`.
- 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 when you test with the reference trace files.

We have also 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
     3 (1,1,1)       9       8       6       9       8       6  traces/yi2.trace
     3 (4,2,4)       4       5       2       4       5       2  traces/yi.trace
     3 (2,1,4)       2       3       1       2       3       1  traces/dave.trace
     3 (2,1,3)     167      71      67     167      71      67  traces/trans.trace
     3 (2,2,3)     201      37      29     201      37      29  traces/trans.trace
     3 (2,4,3)     212      26      10     212      26      10  traces/trans.trace
     3 (5,1,5)     231       7       0     231       7       0  traces/trans.trace
     6 (5,1,5)  265189   21775   21743  265189   21775   21743  traces/long.trace
    27
```

For each test, the autograder 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.

Evaluation

We will run your cache simulator using different cache parameters and traces. There are eight test cases, each worth 3 points, except for the last case, which is worth 6 points:

```
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
```

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, printing 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 3 points, and your simulator outputs the correct number of hits and misses, but reports the wrong number of evictions, then you will earn 2 points.

5 Evaluation Summary

The total points for this lab are broken down as follows:

- Cache Simulator: 27 points
- Style: 3 Points

5.1 Evaluation for Style

There are 3 points for coding style. These will be assigned manually by the course staff. We expect CS70-like style.

5.2 Putting it all Together

We have provided you with a *driver program*, called `./driver.py`, that performs a complete evaluation of your simulator code. This is the same program that the course staff uses to evaluate your submitted work. 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
```

6 Handing in Your Work

Each time you type `make` in the `cachelab-handout` directory, the Makefile creates a "tarball" (`tar` archive), called *userid*`-handin.tar` (where *userid* is your Knuth login), that contains your current `csim.c` file.

When you have finished the lab, use `cs105submit` on Knuth or Wilkes to submit this tarball.

IMPORTANT: Do not create the handin tarball on a Windows or Mac machine, and do not hand in files in any other archive format, such as `.zip`, `.gzip`, or `.tgz` files (`cs105submit` will reject those anyway).