
cs281: Introduction to Computer Systems

Project Lab 6: The Cachelab – Simulating a Cache Controller

Distributed: Monday, Nov. 10, Due: Monday, Nov. 17

Introduction

This assignment will help you understand the operation of cache memories by simulating the actions that a Cache Controller must take to process memory addresses. In particular, your simulator must manage a *virtual* cache against a sequence of address references, determine hits versus misses, and, in a virtual sense, replace lines already present in the cache with a line being requested. The thing that makes your simulator different than an actual cache controller is that you have no requirement to load or store the actual data associated with the memory references. However, to make the assignment more realistic, we will use a program that can determine all of the instruction and data addresses referenced by a target program and log them to a file. This log file will be the input to your simulator.

This is an individual project. You must run this lab on a 64-bit x86-64 machine, such as the Ubuntu machines in Olin 219. Be sure and use Piazza to post questions and get replies from your instructor, your TAs, and from fellow students. Students sending questions directly to the professor by email will be politely asked to post on Piazza for an answer.

Your simulator is a C program (about 200-400 lines) that simulates the behavior of a cache memory.

1 Downloading the assignment

Start by copying `cachelab-handout.tar`, available on the course web site, 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. You will be modifying a single file: `csim.c`. To compile, type:

```
linux> make clean
linux> make
```

2 Detailed Description

2.1 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

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

on the command line runs the executable program “ls”, 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.

2.2 Writing a Cache Simulator

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 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 is to fill in the `csim.c` file so that it takes the same command line arguments and produces the identical output as the reference simulator. Notice that this file is almost completely empty. You'll need to write it from scratch.

Programming Rules

- Include your name and loginID in the header comment for `csim.c`.
- Your `csim.c` file must compile *without warnings* 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.
- 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.
- 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);
```

- For this 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*.

3 Evaluation

This section describes how your work will be evaluated. The full score for this lab is 50 points:

- Correctness: 36 Points
- Style, Performance, Absence of Memory Leaks: 14 Points

3.1 Evaluation for Correctness

We will run your cache simulator using different cache parameters and traces. There are eight *known* 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, 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 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.

In addition to the known test cases, worth 27 total points, your instructors will use 2 additional tests, one worth 3 points, and another more complicated case, worth 6 points. This brings the correctness total to 36 points.

3.2 Evaluation for Style

There are 14 points for coding style. These will be assigned manually by your instructor. Your program will be assessed for good functional decomposition, good variable and function naming, and overall clarity. It will also be inspected for efficiency and for the potential of memory leaks.

4 Working on the Lab

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
```

Points	(s,E,b)	Your simulator			Reference simulator			
		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, 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:

- 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.
- We recommend that you use the `getopt` function to parse your command line arguments. You'll need the following header files:

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

See “man 3 getopt” for details.

- 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.

5 Handing in Your Work

Each time you type `make` in the `cachelab-handout` directory, the Makefile creates a tarball, called `userid-handin.tar`, that contains your current `csim.c` file.

Email your `userid-handin.tar` to your instructor by the due date and time.