CSCI 2021, Fall 2018
Cache Lab: Understanding Cache Memories
Assigned: Wednesday, December 5, 2018
Due: Wednesday, December 12 11:55PM
Last Possible Time to Turn in: December 13 11:55PM

1 Logistics

This is an individual project. You must run this lab on a 64-bit x86-64 machine.

2 Overview

This lab will help you understand the impact that cache memories can have on the performance of your C programs.

The lab consists of two parts. In the first part you will complete a small C program (about 200-300 lines) that simulates the behavior of a cache memory. In the second part, you will optimize a small C function function, with the goal of optimizing cache performance (e.g. minimizing the number of cache misses), branch efficiency, and other performance techniques you have learned.

3 Downloading the assignment

> The handout tarball is located at
> `/web/classes/Fall-2018/csci2021-010/ha/5/ha5-handout.tar`

Start by copying `ha5-handout.tar` to a protected Linux directory in which you plan to do your work. Then give the command

```
linux> tar xvf ha5-handout.tar
```

This will create a directory called `ha5-handout` that contains a number of files. You will be modifying two files: `csim.c` and `encoder.c`. To compile these files, type:

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

4 Description

The lab has two parts. In Part A you will complete the implementation of a cache simulator. In Part B you will optimize the program for cache performance.

4.1 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 in Part A. 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.

4.2 Part A: Writing a Cache Simulator

In Part A 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. Much of the simulator has already been written so your task is to complete the program.

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 Part A 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.The current cache simulator is missing the functionality for the functions `freeCache`, `accessData`, and a portion of the main function. Knowing what you know about how cache works, it is your job to complete these functions such that they produce the same thing that the reference file does.

Programming Rules for Part A

- Include your name and loginID in the header comment for `csim.c`.
- 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 for Part A, 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.

4.3 Part B: Program Optimization

Your job in part B is to optimize the procedure `analyze`, any sub-procedures, and data structures in the file `encoder.c`. Currently, it is written in such a way that it does not take advantage of the cache to decrease the run-time. It is up to you to identify the areas of the function that could be improved using optimizations you have learned during lecture (reducing unnecessary calls to procedures, removing aliasing, alignment, etc.).

The encoder program computes the average of some data and prints the result to stdout. It also performs some string operations (an "encoding") and writes that result to the file `output.txt`. The data is defined in `data.o`, but is otherwise hidden from view.

NOTE: You will not be allowed to change the function so that it ruins the generality of the program. For instance, if the program were designed to give a specific value on certain input, you cannot simply change the function so that it returns the value that it expects. To account for this, we will be testing your updated program against different sets of data in order to grade the correctness of your program.

Programming Rules for Part B

- Your code must return the correct value and write the correct output file to receive credit.
- Your code should utilize cache-based optimizations.
- Do not change any code in `main` or `setupDocuments`. All other code is fair game.

5 Evaluation

This section describes how your work will be evaluated. The full score for this lab is 100 points:

- Part A: 45 Points
- Part B: 55 Points

5.1 Evaluation for Part A

For Part A, we will run your cache simulator using different cache parameters and traces. There are eight test cases, each worth 5 points, except for the last case, which is worth 10 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.

5.2 Evaluation for Part B

For Part B, we will evaluate the correctness and performance of the function that you will optimize. 55 points are possible To get a sense of the score that you will receive on part B of this assignment, the `compute_performance.sh` script is provided for you. Run it with your encode binary as an argument and it will compute the estimated number of cycles (CEST). Your performance score for Part B will be determined by comparing your CEST score against a series of solutions. For example the CEST score for encoder without any optimizations:

```
linux> make encoder
linux> ./compute_performance.sh encoder
24623636
```

Your score for Part B will be zero if your optimizations change the program output or the content of the file `output.txt`. A python module `correct.py` is included to automatically assess the correctness of your program. The copy of `encoder.c` you received is correct. Run the module with your program as an argument:

```
linux> ./correct.py encoder
Correct
```

The more you optimize the program, the more points you will receive on Part B. Full points will be awarded to programs that produce CEST$\leq$1244182, and fewer points depending on where they fall in comparison

to a sequence of partial solutions. Programs that acheive 759000≤CEST<1244182 will be rewarded by up to 10 points extra credit. The included `driver.py` program to see how many points your CEST score is worth:

```
linux> ./driver.py
...
Part B: Testing program optimization
Testing encoder output for correctness...
Running encoder through valgrind and computing cycles...
  Correct                1
  CEST                   24623636
  Performance Points     0
...
```

NOTE: There may be a slight difference between the CEST calculated by `driver.py` and `compute_performance.sh`, but the former will be used in the final grading.

6 Working on the Lab

6.1 Working on Part A

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
     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 for working on Part A:

- 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.

6.2 Working on Part B

Here are some hints and suggestions for working on Part B.

- Profile the program to see where the most optimization work will be needed. Finding where the most time is spent in a program will focus your efforts.
- Don't try to rewrite the entire program. You are only required to optimize the current version of the program.
- First, find the small things that you could optimize and then work your way up to optimizing larger portions of the code.
- Think of how memory is layout out when you are trying to read it, i.e how can you read something like a matrix more efficiently to utilize cache.

6.3 Putting it all Together

We have provided you with a *driver program*, called `./driver.py`, that performs a complete evaluation of your simulator and encoder code. This is the same program your instructor uses to evaluate your handins. The driver uses `test-csim` to evaluate your simulator, and it measures the CEST of the encoder with `compute_performance`. To run the driver, type:

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

7 Handing in Your Work

Each time you type `make` in the `ha5-handout` directory, the Makefile creates a tarball, called `userid-handin.tar`, that contains your current `csim.c` and `encoder.c` files. **Do not rename these files**.

Upload your `userid-handin.tar` on moodle before the due date.

IMPORTANT: Do not create the handin tarball on a Windows or Mac machine, and do not handin files in any other archive format, such as `.zip`, `.gzip`, or `.tgz` files.