p4: Cache Simulator

1. Logistics

1. This project must be done individually.

   It is academic misconduct to share your work with others in any form including posting it on publicly accessible web sites, such as GitHub.

   It is academic misconduct for you to copy or use some or all of a program that has been written by someone else.

2. All work for this project is to be done on one of the CS department's instructional Linux machines. You're welcome to remotely login using ssh or MobaXterm as was shown in lecture (or any other way that you figure out on your own).
3. All projects in this course are graded on CS department's instructional Linux machines. It is your responsibility to make sure that your code runs as you expect on these machines.

2. Learning Goals

This assignment has 2 parts. The purpose of the first part is to understand more about how caches work and learn a bit about simulation along the way. In second part of the assignment, you will work on developing a small cache simulator. This will make you an expert in the basics of caches and further strengthen your C programming skills.

3. Getting the Required Files

Copy the tar file `/p/course/cs354-skrentny/public/code/p4/cachelab.tgz` to your private working directory. Next enter the following command in your current working directory:

```
tar -zxvf cachelab.tgz
```

This should create a directory named **cachelab** which has the directories **part1** and **part2** required for the two parts of the assignment. Verify the contents of the cachelab directory as show below.

```
[haseeb@espresso] (47)$ ls
cachelab/ cachelab.tgz
[haseeb@espresso] (48)$ ls cachelab
part1/ part2/
[haseeb@espresso] (49)$ ls cachelab/part1
p4questions.txt
```

```
[haseeb@espresso] (50)$ ls cachelab/part2
csim.c  csim-ref*  Makefile  README  test-csim*  traces/
```

Next copy the directory "**pin-fw**" into your "**cachelab/part1**" directory. This could take a couple of minutes.

```
[haseeb@espresso] (63)$ ls
part1/  part2/
[haseeb@espresso] (64)$ cp -r /p/course/cs354-skrentny/public/code/p4/pin-fw part1/
[haseeb@espresso] (65)$ ls part1/
p4questions.txt  pin-fw/
```

4. Part 1 - cache analysis with pin:

NOTE: It is recommended that you complete part 1 of the assignment before moving on to part 2.

For this part, you will use a dynamic binary instrumentation framework called **pin** to measure cache performance statistics of an executable program by specifying certain cache parameters. Specifically you will use pin to run simulations and answer questions in the file "**p4questions.txt**".

NOTE: You will not be able to run **pin** on your personal machine. It is recommended that you complete part 1 on one of the CSL machines (either physically or remotely via ssh).

pin runs the executable to internally produce a series of address traces. These are the ordered set of memory addresses that a program reads from or writes to as it runs. Each address may represent a read for an instruction fetch, or a read/write to some data variable stored in memory.

The address traces are then used internally by the **cache simulator** tool in pin. The cache simulator is a program that acts as if it is a cache, and for each trace, it does a lookup to determine if that address causes a cache hit or a cache miss. The simulator keeps track of the hits/misses, and finally prints these statistics for you.

To run a simulation, from inside your "**part1**" directory, you will use a command line of the format:

```
pin-fw/pin -injection child -t pin-fw/source/tools/Memory/obj-ia32/allcache.so -is <c
apacity> -ia <associativity> -ib <block-size> -ds <capacity> -da <associativity> -db
<block-size> -- <your-exe>
```

In this command, you would need to replace `<your-exe>` with the name of your executable file. The simulator presumes a separate I-cache and D-cache. The I-cache holds only the machine code instructions, as read when doing an instruction fetch. The D-cache holds all other data read or written while a program runs.

There are 6 cache parameters, 3 each for I-cache and D-cache.

Capacity:

```
-is <capacity> -ds <capacity>
```

This sets the total capacity in bytes for the cache. For all your simulations (in part 1) use a 16KB or 16384 bytes size for both the I-cache and the D-cache.

```
-is 16384 -ds 16384
```

Associativity:

```
-ia <associativity> -da <associativity>
```

This sets the set associativity of the cache. For all your simulations (in part 1) use an associativity of 1 (direct-mapped) for both I-cache and the D-cache.

```
-ia 1 -da 1
```

Block size:

```
-ib <block-size> and -db <block-size>
```

This sets the block size in bytes for the cache. For all your simulations (in part 1) use a block size of 64 bytes for the I-cache.

```
-ib 64
```

The block size for the D-cache `-db <block-size>` is the parameter that you will be changing to answer the questions in the file "**p4questions.txt**".

So your final command to run a simulation, from inside your "**part1**" directory, on an executable will look like:

```
pin-fw/pin -injection child -t pin-fw/source/tools/Memory/obj-ia32/allcache.so -is 16
384 -ia 1 -ib 64 -ds 16384 -da 1 -db <block-size> -- <your-exe>
```

In order to answer the questions in "**p4questions.txt**" you will need to write 3 very small and simple programs "**cache1D.c"**, "**cache2Drows.c**" and "**cache2Dcols.c"**, and then compile them into executables as you have done in previous programs like:

```
gcc -o cache1D cache1D.c -Wall -m32 -std=gnu99
```

cache1D.c:

Declare a global array of integers of size 100,000 outside (and prior) to the **`main()`** function, so that this array will be on the data segment. Use a for loop in the **`main()`** function to iterate over the entire array and set the value of each element in the array as the index.

```
arr[i] = i;
```

Do not print to the console or do anything extra. All this program does is iterate over the global array to set its values.

Use the executable from this program to answer questions in the section cache1D of "**p4questions.txt**".

cache2Drows.c and cache2Dcols.c:

"**cache2Drows.c**" is similar to the above program except our global array is now a 2-dimensional integer array of dimensions 3000 rows x 500 columns. In our **`main()`** function we will iterate over the array with a nested loop in a row-wise order, **where the inner loop works its way through the elements of single row of the array, and the outer loop iterates through the rows**. In this order of traversal you will set every element of the array as:

```
arr2D[row][col] = row + col;
```

"**cache2Dcols.c**" does the exact same thing except the iteration is in a column-wise order, **where the inner loop works its way through the elements of a single column of the array, and the outer loop iterates through the columns**.

Use the executables from these programs to answer questions in the remaining sections of "**p4questions.txt**".

5. Part 2 - cache simulator csim:

You will write a cache simulator in "**csim.c**" that takes a valgrind memory trace as input, simulates the hit/miss/eviction behavior of a cache memory on this trace, and **outputs the total number of hits, misses and evictions**.

5.1 Reference trace files

The "**cachelab/part2/traces**" directory contains a collection of reference trace files that we will use to evaluate the correctness of the cache simulator that you will write in the next section. 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[space]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).

You will consider only memory accesses to data (L/S/M) and ignore instruction fetch (I) while working on your simulator. The address field specifies a **64-bit** hexadecimal memory address. The size field specifies the number of bytes accessed by the operation. In this course we have dealt with

32-bit computers. **Only for part 2 of this assignment, we use 64-bit computer traces** - all the addresses for accessing memory are 64-bit addresses.

NOTE: You will not have to generate any traces on your own. You need to work only with the traces provided to you in "**cachelab/part2/traces**".

5.2 Writing the cache simulator csim:

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. You can use "**csim-ref**" to compare your implementation.

The reference simulator takes the following command-line arguments:

```
./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 cache 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**).

For example, the following command gives us the output:

```
./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 -s 4 -E 1 -b 4 -t traces/yi.trace -v
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
```

You can use this verbose output from "**csim-ref**" while debugging your code ("**csim.c**"). The “hit”/”miss”/”eviction” in the verbose output above indicates if that particular memory access (L/S/M) specified by the trace file led to hit/miss/eviction in cache.

We have provided you with skeleton code in file "**csim.c**". Your job is to complete the implementation so that it outputs the correct number of hits, misses and evictions. It is highly

recommended that you support the verbose output (using the -v option) in your implementation of "csim.c". It would help you in debugging your code. No points will be deducted if you chose not to.

5.3 Testing csim:

Once you have made your changes to file "**csim.c**" and want to test your implementation, do the following:

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

NOTE: Please do not print anything else.

Notice that the output from "**csim**" matches the output from "**csim-ref**". To work correctly your simulator must match the output of "**csim-ref**" for arbitrary traces and values of **s**, **E**, and **b**.

To check your implementation in "**csim.c**" against "**csim-ref**", you can use the program "**test-csim**" and run the following command:

```
./test-csim
```

The output will show you how your implementation in "**csim.c**" compares against "**csim-ref**". The maximum score possible using the "**test-csim**" program is 48.

5.4 Guidelines:

Some important points regarding the implementation in "**csim.c**":

You should carefully read the skeleton code and the comments in "**csim.c**".

Make sure you understand the various variable types that are provided to you in the skeleton code namely **`cache_t`**, **`cache_set_t`**, and **`cache_line_t`**. You will need to decide how to implement the LRU policy for your cache. Two possible implementations are:

- List: Move the most recently used cache line to the head of the list. Evict the tail of the list as the least recently used.
- Counter: Every cache line has a counter. An access to a line would set its value to 1 greater than the current maximum value of all counters in its set. The line with the smallest counter value or least recently used is evicted.

Based on the choice you make, you would need to add an extra field to the **`cache_line_t`** struct.

Starting from **`main()`** function, the flow is described below in brief:

- Parse the command line arguments. This is already done for you.
- **`void initCache()`**:This function should allocate the data structures to hold information about the sets and cache lines using **`malloc()`** depending on the values of parameters S ($S = 2^s$) and E. You need to complete this function.

- **`void replayTrace(char* trace_fn)`**: This function parses the input trace file. This part is already done for you. It should call the **`accessData()`**function. You need to complete the missing code in this function.
- **`void accessData(mem_addr_t addr)`**: This function is the core of implementation which should use the data structures that were allocated in **`initCache()`** function to model the cache hits, misses and evictions. You need to complete this function. The most crucial thing is to update the global variables **`hit_count`**, **`miss_count`**, **`eviction_count`** inside this function appropriately. You should implement Least-Recently-Used (LRU) cache replacement policy.
- **`void freeCache()`**: This function should free up any memory you allocated using **`malloc()`** in **`initCache()`** function. This is crucial to avoid memory leaks in the code. You need to complete this function.
- **`printSummary(hit_count, miss_count, eviction_count)`**: This function prints the statistics in the desired format. This is already implemented for you.

6. Requirements

- Your program must follow these Style Guidelines.
- Your program must follow these Commenting Guidelines. You can keep the file header and function header comments we've put in the skeleton code of csim.c but you must comment the rest of your code inside the functions where necessary.
- Your programs must operate exactly as specified.
- We'll compile your programs with **`gcc -Wall -m32 -std=gnu99`** in part 1, and the Makefile in part2 on the CS Linux lab machines. It is your responsibility to ensure that your programs compile on these machines, **without warnings or errors**. Points are deducted if compiling your programs on the CS Linux lab machines produces warnings and/or errors.

7. Submitting Your Work

Submit the following **source files** under Project 4 in Assignments on Canvas on or before the deadline:

1. cache1D.c
2. cache2Drows.c
3. cache2Dcols.c
4. p4questions.txt
5. csim.c

It is your responsibility to ensure your submission is complete with the correct file names having the correct contents. The following points will seem obvious to most, but we've found we must explicitly state them otherwise some students will request special treatment for their carelessness:

- **You will only receive credit for the files that you submit.** You will not receive credit for files that you do not submit. Forgetting to submit, not submitting all the listed files, or submitting executable files or other wrong files will result in you losing credit for the assignment.
- **Do not zip, compress or submit your files in a folder.** Submit only the text files as listed.

- **Make sure your file names exactly match those listed.** If you resubmit your work, Canvas will modify the file names by appending a hyphen and a number (e.g., cache1D-1.c) and these Canvas modified names are accepted for grading.

Repeated Submission: You may resubmit your work repeatedly until the deadline has passed. **We strongly encourage you to use Canvas as a place to store a back up copy of your work.** If you resubmit, you must resubmit all of your work rather than updating just some of the files.