# Assignment 4

CS/ECE 354, Spring 2016

**Due Friday, April 8th before 9 AM**

## 1. Collaboration Policy

For this assignment, you may work in pairs (2 people). All students (whether working in a pair or not) must individually turn in the assignment. Therefore, 2 copies of the assignment will be turned in for those working as a pair. The grader will choose to grade only one copy of a pair's work, and assign the same score to both students.

If one member of a pair does something unexpected with respect to turning in the assignment (for example: bad copy of the source code, late, or perhaps not turning in portions of the assignment), then *both* of the pair get the same penalty and the same score.

If working in a pair, the names and sections of *both* students must appear as comments at the top of all files that will be turned in. Please read the handin section (section 7) of this specification carefully for details about **one extra file that should be submitted by people working in a pair**.

## 2. Learning Goals

This assignment will have 2 parts:

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

Please read the entire assignment specification before starting.

## 3. Getting the required files

Copy the file `/p/course/cs354-common/public/src/cachelab.tar.gz` to your private working directory. Enter the following commands (without "$>"):

```
$> tar -zxvf cachelab.tar.gz
```

This should create a directory named **cachelab** which has files required for the two parts of the assignment.

```
$> ls cachelab
part1/     part2/

$> ls cachelab/part1
p4questions

$> ls cachelab/part2
traces/  cachelab.c  cachelab.h  csim.c  csim-ref*  Makefile  README
test-csim*
```

# 4. Part 1 - running *pin* (40 points)

**ALERT:** The understanding about caches developed while working on part 1 would be necessary to work on part 2. **SO, COMPLETE PART 1 BEFORE WORKING ON PART 2 OF THE ASSIGNMENT.**

For this part, you will use a program called *pin* that produces cache performance statistics, given cache parameters and an executable.

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

The address traces are then used internally by a **cache simulator**. The cache simulator is a program that acts as if it is a cache, and for each trace, does a look-up to determine if that address causes a cache hit or a cache miss. The simulator tallies the hits/misses, and when it has completed simulation of all the traces, it prints these statistics for you.

To run a simulation, use a command line similar to

```
$> /p/course/cs354-common/public/cache/pin -t
/p/course/cs354-common/public/cache/source/tools/Memory/obj-ia32/allca
che.so -is 16384 -ia 1 -ib 64 -ds 16384 -da 1 -db 64 -- yourexe
```

In this command, you would need to replace "yourexe" with the name of your executable file. This command line is present at the top in file p4questions for you to copy from, in case you don't want to type it manually. 6 of these command line arguments specify cache parameters to *pin*. The simulator presumes 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.

**For the I-cache, specify**

-ia 1

This causes the set associativity of the cache to be 1, or direct mapped. This is the only value we will use in this part.

-is N

Substitute a power of 2 for N. This sets the capacity for this cache. For all our simulations in this part, use a 16KB (16384) size.

-ib N

Substitute a power of 2 for N. N is the number of bytes per block. Use a block size of 64bytes for I-cache for all simulations.

**For the D-cache, specify**

-da 1

This causes the set associativity of the cache to be 1, or direct mapped. This is the only value we will use in this part.

-ds N

Substitute a power of 2 for N. This sets the capacity for this cache. For all our simulations in this part, use a 16KB (16384) size.

-db N

Substitute a power of 2 for N. N is the number of bytes per block. **You will be changing this parameter in this part of assignment.**

As you work your way through this part, you will be answering questions that are in the file `p4questions.` This file will be turned in.

## Step One: 1-dimensional array code

Write a very small C program called cache1D.c that sets each element of an array of 100,000 integers to the value of its index. The statement that sets a single array element will be something like

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

The resulting executable program will be used with *pin* to generate statistics about cache usage for your analysis. To make the analysis easier, the C program is required to

- Declare the array as a global variable, so the declaration will be outside of main() (and prior to main() within the source code file). This requirement will cause the array to be within the global data segment, and *not* on the stack.
- Place a `for` loop inside main() to set each element of the array. One array element is set during each iteration of the for loop.

Compile your program with

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

## Step Two: 1-dimensional array code analysis

Run the 3 simulations needed to answer the questions in the p4`questions` file. Then, answer the questions.

## Step Three: 2-dimensional arrays, two ways

Repeat what you did for the 1 dimensional array with two programs that set elements of a 2-dimensional array. Many students will not have used 2-dimensional arrays in C before; the K&R book, section 5.7-5.9, starting on page 110 will be a good reference for declaring and using 2D arrays.

First, write a C program called cache2Drows.c that sets each element of a 3000 row by 500 column array of integers to the sum of the row index and the column index. The statement that sets an element will be something like

```
arr2d[row][col] = row + col;
```

Use a set of nested `for` loops, where the **inner loop works its way through the elements of single row of the array, and the outer loop iterates through the rows**.

Run the cache analysis requested in the p4questions file.

Second, write another C program called cache2Dcols.c that does the same thing as cache2Drows.c did, *but in a different order*. This program has the **inner loop work its way through the elements of a single column of the array, and the outer loop iterates through the columns**. If you truly understand your code, it should be close to trivial to copy cache2Drows.c and modify it to become cache2Dcols.c.

Last step will be to figure out, understand, and explain why these 2 programs (that accomplish exactly the same thing) result in different cache performance.

# 5. Part 2 - developing *csim* (60 points)

## 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 you write in this part. 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 should 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.

## 5.2 Writing the 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.**

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:
**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 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) from pages 597 and 598 of the CS:APP2e textbook.

For example, enter the following command:

```
$> ./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
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. **You NEED NOT support the verbose output (using the -v option) as mentioned above.**

Once you have made 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
```

The **SAMPLE OUTPUT** should look like as shown below:

```
hits:4 misses:5 evictions:3
```

**NOTE:** The number of hits, misses and evictions will vary with different cache configurations and different traces. However, the output should be single line similar to what is shown above.

## YOU SHOULD NOT PRINT ANYTHING EXTRA IN THE OUTPUT.

Your simulator must work correctly for arbitrary values of `s, E, and b`.

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

You should carefully read the skeleton code in csim.c. 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.

You **MUST READ COMMENTS in the file csim.c** which provide additional details.

# 6. Grading Scheme

This assignment will be graded for a total of 100 points as shown below:

1. **Part 1:** 40 points
2. **Part 2:** 60 points

**To check your implementation in csim.c, you can run the following command:**

```
$> ./test-csim
```

It will tell you how your implementation in csim.c compares against csim-ref. Maximum score possible using test-csim script is 48. **We will inspect your code manually** to check for implementation details and adherence to coding guidelines, which will carry 12 points.

# 7. Handing in the assignment

You will be turning in **5 files** in total for this assignment:

**Part 1:**

1. cache1D.c
2. cache2Drows.c
3. cache2Dcols.c
4. p4questions

**Part 2:**

5. csim.c

Copy all these 5 files into your handin directory. Your handin directory for this project is
/p/course/cs354-common/public/spring16.handin/login/p4
where login is your cs-login.

For example, if my login is "lokeshjindal15", my handin directory would look like following after submission:

```
$> pwd
/p/course/cs354-common/public/spring16.handin/lokeshjindal15/p4

$> ls
cache1D.c  cache2Dcols.c  cache2Drows.c  csim.c  p4questions
```

*If you are working as part of a pair*, you must turn in an **extra file**. This file will contain the names and sections of **both** students in the pair. As an example, if Kevin worked with Haseeb on this assignment, the contents of the extra file for both Kevin and Haseeb would be
Kevin Zhang  section 1
Haseeb Tariq  section 2

The name of this file is specialized to help the 354 automated grading tools identify who worked together. This file name is composed of the CS logins of the partners separated by a period. The file name is of the form <login1>.<login2>. Kevin's login is kzhang, and Haseeb's login is haseeb. The file name that both use will be kzhang.haseeb; please have both partners use the same file name. It does not matter which partner's name is first within this file name.

## Requirements

1. Include a comment at the top of the source code of all programs with your *name* and *section* (and your partner's name and section, if working in a pair).
2. Use the instructional Linux machines for this assignment! A penalty will be imposed for any program that was obviously edited on a Windows machine and transferred to the Unix machines for turning in. It is annoying to see Windows line endings (^M) at the end of every line. Points will be taken off for such mistakes.
3. Your programs must compile on an instructional Linux machine as indicated in this specification *without warnings or errors*.