CSCI-UA 201 sec. 3
Lab Assignment 2: The Cache Lab
Due: April 18, 2015 at 11:59PM EST

The tutorial for this lab will be held on April 3 at 2:00pm in room 109WWH.

Introduction

In this lab you will practice anlyzing cache use and how to tweak C code to make it cache friendly.

Hand-in instructions

Solutions to all problems should be handed in in electronic format on NYU Classes. Combine all of them into a single zip file (or tar file) for submission. No scanned documents, please! If you prefer to hand in solutions to problems 2 and 3 on paper, you may do so **by the end of the class on April 15** - notice that this is before the due date for the rest of the lab.

Problem 1 (50 points)

Download the following file into your virtual machine.

```
http://cs.nyu.edu/~joannakl/cs201.03_s15/assignments/lab2_p1.tar
```

Run:

`tar -xvf lab2_p1.tar`

to extract the lab files

You will get a directory called cache-lab that contains the following files:

- `run` - a small shell script that you use to compile and run your program. If your program is called `refcode.c` then command to compile, execute and get cache miss rates measurements is:

  `./run refcode`

- `refcode.c` - a source file that contains naive implementation of the functions you need to optimize.

For this problem you need to tweak the functions `level_1()`, ..., `level_5()` to make them produce lower number of cache misses. For each function that you optimize, you need to provide an explanation of why your improvement results in smaller number of cache misses. Provided that in comments above each function.

Notes:

- Do not modify the code where the comments state so.
- You can add any variables or data structure you want, but you are not allowed to remove/modify available variables or data structures.
- Feel free to add additional helper functions. If you do, name them starting with the name of the function in which they are used. For example, if you want a helper function for `level_1()` function, then that helper function should be called something like `level_1_helper_A()`. The cache misses that occur within the helper function will count towards the function that calls it.
- Do not use recursion.
- Your changes should not change the underlying algorithm. They should make the existing implementation more efficient. For example, if one of the functions implements a bubble sort, do not change it to quick sort.
- Always rememeber that cache friendly means locality of data access, both temporal and spatial.
- You should first understand what the function does, then attempt to optimize it.

Hand-in instructions

- Make a copy of `refcode.c` and call it `optimized.c`.
- Put your names in the comments at the top of that file.
- Apply your newly learned wisdom about caches to optimize the code in `optimized.c`. Check the results using `./run optimized`.
- Submit the final version (together with the solutions to the other problems) to NYU Classes.

How will we grade this?

- The main measure of success is to reduce the number of cache misses for level 1 cache (L1 cache). These results are shown in the output of `run` script unde the columns **d1mr** and **d1mw**.
- We will compare the number of cache misses in the provided `refcode.c` and in you `optimized.c`.

- Each level is worth 10 points, calculated as follows:
  - 3 points - explanation of the improvement
  - up to 7 points for the improvement:
    - improvement ratio of 0.9 or higher: 0 points
    - improvement ratio of 0.7 - 0.9: 2 points
    - improvement ratio of 0.5 - 0.7: 4 points
    - improvement ratio of 0.3 - 0.5: 6 points
    - improvement ratio of 0.3 or lower: 7 points

Problem 2 (12 points)

The following table gives the parameters for a number of different caches, where m is the number of physical address bits, C is the cache size (number of data bytes), B is the block size in bytes, and E is the number of lines per set. For each cache, determine the number of cache sets (S), tag bits (t), set index bits (s), and block offset bits (b).

Cache	m	C	B	E	S	t	s	b
1.	32	1024	4	4				
2.	32	1024	4	256				
3.	32	1024	8	1				
4.	32	1024	8	128				
5.	32	1024	32	1				
6.	32	1024	32	4				

Problem 3 (15 points)

Consider the following matrix transpose routine:

```
typedef int array[4][4]

void transpose(array dst, array src) {
```

```
  int i, j;
  for (i=0; i < 4; i++) {
    for (j=0; j < 4; j++) {
       dst[j][i] = src[i][j];
    }
  }
}
```

Assume this code runs on a machine with the following properties:

- `sizeof(int)==4`
- The `src` starts at address 0 and the `dst` starts at address 64 (decimal).
- There is a single L1 data cache that is direct-mapped, write-through, write-allocate, with a block size of 16 bytes.
- The cache has total size of 32 data bytes and the cache is initially empty.
- Accesses to the `src` and `dst` arrays are the only sources of read and write misses.

Part A

For each `row` and `col`, indicate if the access to `src[row][col]` and `dst[row][col]` is a hit (h) or a miss (m). For example, both reading `src[0][0]` and writing to `dst[0][0]` are misses.

`dst` array

	Col 0	Col 1	Col 2	Col 3
Row 0				
Row 1				
Row 2				
Row 3				

`src` array

	Col 0	Col 1	Col 2	Col 3
Row 0				
Row 1				
Row 2				
Row 3				

Part B

Repeat **Part A**, but for cache with total size of 64 data bytes.

`dst` array

	Col 0	Col 1	Col 2	Col 3
Row 0				
Row 1				
Row 2				
Row 3				

`src` array

	Col 0	Col 1	Col 2	Col 3
Row 0				
Row 1				
Row 2				
Row 3				

Extra Credit: Problem 4 (10 points)

Research a particular processor (other than Core i7). Prepare 1-2 page report about its memory hierarchy (types, sizes, placement). Include information about number of cores, caches that are core specific and shared. Make sure to include information about your sources.