CS105 – Computer Systems Spring 2023

Assignment 6: Cache Lab

Due: Tuesday, March 7, 2023 at 11:59pm

For this assignment, you will emulate a direct-mapped cache at the user level. As usual, you should complete this assignment with a partner; you may choose your partner for this assignment.

The starter code for this assignment is available both on the course website and on the course VM. You may complete it either on your local machine or on the course VM (the file path is `/cs105/starters/cachelab.tar`. Note that you will need to be connected to the Pomona WiFi network or the Pomona VPN to access the VM.

As usual, you can unpack the starter code with the command "`tar xvf /cs105/starters/cachelab.tar`". This will create a subdirectory named `cachelab-handout` containing the starter code `caching.c` and a `Makefile` that you should use to compile your code with the command "`make`".

1 Background

Recall that caches are used as smaller, faster places to store copies of bytes from main memory. When a process attempts to access a value in memory, it actually first checks the cache. If there is a copy of the value from that address in the cache, it reads the value directly from cache. If the requested value is not in the cache, it (1) reads the value from memory and (2) updates the cache.

2 Starter Code

The starter code for this assignment contains most of the code necessary to simulate a direct mapped cache. Main memory is implemented as an array of bytes; for the purposes of this simulation, we'll assume we have **24-bit** addresses implemented as the type `ptr_24`.

The cache in this assignment has 256 cachelines and is implemented as an array of values of type `cacheline_t`. This type stores all the data from a single cacheline (the valid bit, the tag, and the data block); it is defined as follows:

```
typedef struct {
    bool valid; // 1 bit valid (actually 8 bits, but we'll pretend)
    byte tag; // 8 bit tag
    byte datablock[256]; // 256 byte data blocks
} cacheline_t;
```

The main function in the starter code initializes main memory. It then performs a series of matrix multiplications on one-dimensional matrices (aka vectors) of various lengths by calling the function `vector_multiplication`. In all cases, we assume that the first vector starts at address 0 and the second vector is immediately after the first vector in memory.

The simulation uses a wrapper function `access_direct` to both simulate how computers access memory: it first checks whether a value is in the cache. If it is, it retrieves the value from the cache. If it is not, it retrieves the value from memory, updates the cache, and the returns the value. To do so, it uses a series of four helper functions: `parse_addr`, `is_in_direct_cache`, `lookup_int_in_direct_cache`, and `update_direct_cache`. Unfortunately, these helper functions are not (correctly) implemented.

I recommend that you try compiling (`make`) and running (`./caching`) the starter code before you get started. Make a note of the values you get as answers; you'll want to make sure you get the same answers at the end! Note that all of the tests initially have a hit rate of 0 and a miss rate of 1; this is to be expected, since you have not yet implemented the part of the simulation that uses the cache.

3 Simulating Caching

Your first task is to implement the four missing functions:

1. `parse_addr`: This function should take a 24-bit address and separated it into the tag, index, and offset.
2. `is_in_direct_cache`: This function should return a boolean value: 1 if the value corresponding to the specified tag/index/offset is currently stored in the cache (a cache hit) and 0 if it is not (a cache miss).
3. `lookup_int_in_direct_cache`: This function should read an integer from the cache.
4. `update_direct_cache`: This function should update the cacheline at `index` with to contain the value at `addr` (and the nearby bytes). Hint: the library function `memcpy` is your friend.

4 Analysis

After you have your simulation working, save the output to a textfile using the command `./caching > results.txt` Then create a new plaintext file called `analysis.txt` and answer the following four questions:

1. Explain (in detail) why the cache hit rate/cache miss rate you get for length 2 is correct.
2. Explain why you would expect the cache hit rate to go up as the vectors get longer.
3. Explain why the cache hit rate suddenly drops for large vectors.
4. What would be a better cache configuration to improve the hit rate for large vectors? Hint: no, you can't make your cache any bigger.

Feedback

Create a file called `feedback.txt` that answers the following questions:

1. How long did each of you spend on this assignment?
2. Any comments on this assignment?

As always, how you answer these questions **will not affect your grade**, but whether you answer them will.

Submission

Submit the following four files on Gradescope: your source code `caching.c`, your evaluation results `results.txt`, your analysis `analysis.txt`, and your feedback file `feedback.txt`. Make sure that you tag your partner as a collaborator when you submit. Also, be sure the names of all team members are *clearly* and *prominently* documented in the comments at the top of all submitted files.