CS105 – Computer Systems Fall 2023

Assignment 5: Cache Lab

Due: Tuesday, October 24, 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 (`itbdcv-lnx04p.campus.pomona.edu`). 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` containing four files: `eval.c`, `caching.h`, `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 is comprised of three files:

- `caching.h` This is a header file that defines types, global variables, and function headers that are shared between the two C files. Most critically, this file defines values `NUM_CACHELINES` and `DATA_BLOCK_LEN` (both set equal to 256) as well as types `byte`, `bool`, and `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)
      long tag; // 48 bit tag (actually 64 bits, but we'll pretend)
      byte datablock[DATA_BLOCK_LEN]; // 256 byte data blocks
  } cacheline_t;
  ```

  The simulated cache is declared in this file as a global variable of type `cacheline_t*`, aka an array of values of type `cacheline_t`. **You should not make any changes to this file.**

- `eval.c` This file contains most of the code needed to simulate and evaluate a direct-mapped cache. You do not need to make changes to this file, although you may find in helpful to temporarily modify the value `MAX_LEN` defined at the top to a smaller value for debugging purposes.

The main function in `eval.c` initializes the simulation. 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, the first vector `a` and the second vector `b` are immediately sequential in memory (these addresses are explicitly defined inside the loop in the `main` function).

The simulation uses a wrapper function `access_direct` (also defined in file `eval.c`) to 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 various helper functions implemented in file `caching.c`.

- `caching.c` implements a series of four helper functions: `parse_addr`, `is_in_direct_cache`, `lookup_int_in_direct_cache`, and `update_direct_cache`. Unfortunately for you, these helper functions are not yet (correctly) implemented.

3 Simulating Caching

Your first task is to implement the four functions in `caching.c`:

1. `parse_addr`: This function should take an address of type `void*` 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 data at `addr` (and the nearby bytes). Hint: the library function `memcpy` is your friend.

Getting started. 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.

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 comment at the top of `caching.c`.