Benjamin Patrick Schreiber 11790684 PA 1 Cache Simulator

Cache Simulator Implementation

Introduction

This C program operates as a set associative memory cache simulator. The program reads in a `valgrind` trace file (see traces/) and simulates the behavior of the cache with the given dimensions and associativity. The program outputs the number of hits, misses, and evictions that occur during the simulation.

Set up for simulation

I began the program by dividing the `main` function into three steps:
1. Parse the command line arguments
2. Open the file from the arguments
3. Run the cache simulation

The command line is parsed using the `getopt` function. The program then attempts to open the file by the given path. If the file is not found, the program will exit with an error message.

In order to read the file, I created an `iterator` seen in `iterator.h`.
An enum `v_opt_flag` keeps track of all the valgrind flags that can be read from the file.
Then, a struct `trace_line_t` holds the flag as well as the address and size of the memory access.

With this, I used a lazy functional-programming inspired approach with the `yield_trace_line` function, which
reads the next line from the file and returns a `trace_line_t` struct. This function is used in the main loop of the program to read the file line by line. When the file is finished, the function returns a `trace_line_t` with the flag `U` for undefined.

Cache Simulation

Now that we can read from the trace files, we can simulate the cache.

`cache_t` is a struct that holds the cache's dimensions and associativity. It also holds the cache's data in a 2D array of `cache_line_t` structs. Each `cache_line_t` struct holds a valid bit, tag, and block through the `cache_block_t` struct.

Since we have to use LRU eviction, the `cache_block_t` holds a timestamp that is updated every time the block is accessed.

To create a cache, you call `create_cache(dimensions)`. This will return a cache with all valid bits set to 0, and all timestamps set to 0.

Reads, Writes

Reading and writing is exactly the same in this program since there are no write-backs.
The `read_cache` and `write_cache` function returns a `cache_result_t` which holds the stats of the read operation. This will reveal to our program if the read was a hit, miss, or eviction.

In order to access the cache, we must calculate the set index and tag.

To get the set_index, I use a bit mask to get the bits that represent the set index. I then shift the address to the right by the number of bits in the block and set the index to get the tag.

To get the tag, I shift the address to the right by the number of bits in the set index and block.

Now that we have the set index and tag, we can go directly to the set in the cache and check if the tag matches any of the blocks.

cache_block_t *set = cache.data + (set_index * cache.E);

From here, we need to iterate `E` times (the amount of lines in the set) to check if the tag matches any of the blocks.

While checking the tag, we also check if the block is valid. On an invalid block, we can immediately return a miss.
We can also fill the spot in the cache with a valid value after this.

If we find a matching tag, we return a hit.

If we do not find a matching tag, we must evict a block. We do this by finding the least recently used block in the set and replacing it with the new block. Loop through the set and find the block with the lowest timestamp.

Tests

I developed this program using test-driven development, via unit tests for the individual components, and integration tests for the entire program. I could have even done snapshot testing by saving the output of the program and comparing it to a known good output (the ones provided), but the integration tests were sufficient for this project.