Cache Simulator Coursework

1936 Words
by Kai Bowers

Table of Contents

1. Introduction

This document describes the development, usage and testing procedures of the cache simulation software located at https://github.com/University-of-London/coursework-KaiBowers99.

The software simulates the behaviour of a cache, specifically tracking hits, misses, and evictions based on the instructions of a provided valgrind trace file.

The program is authored in Rust and requires Cargo version 1.76.0 for building and testing. Make sure to have Rust and Cargo installed on your system to proceed with building and testing the program.

2. Usage

The software can be integrated as a library within your own software or utilized as a stand-alone tool accessible from the command line.

All the following arguments are always required for both cases:

`-s` refers to the number of set index bits.
`-E` refers to the associativity, which are the number of lines per set.
`-b` refers to the size of the blocks.
`-t` refers to a path of a valgrind trace file containing instructions for the cache simulator. “I” Instructions are ignored.

for `s`, `E` and `b` a integer larger than zero is required. A error is printed otherwise.

2.1. Command-line

To execute the tool from the command line, simply run the sim software with your desired parameters using:

```
sim -s -E -b -t
```

Here's an example of how to use the tool, provided your console is within the 'sim' directory:

```
./target/release/sim -s 4 -E 3 -b 2 -t
/home/codio/workspace/traces/yi2.trace
```

the arguments can be given in any order. If the flag order is unspecified, the above format “`-s -E -b -t`” is assumed:

```
./target/release/sim 4 3 2 /home/codio/workspace/traces/yi2.trace
```

2.2. Integrated

The core logic of the software resides in two files located within the 'sim/src' directory: "main.rs" and "cache.rs". To use the code directly, only "cache.rs" is necessary. You can include it and invoke the function as follows:

```
pub fn simulate_cache(s: u32 , e: u32, b: u32, file_path: &String) ->
String
```

It produces the result in the terminal and simultaneously returns it as a string.

2.3. Building

if you wish to build the sim project, run the following command within the sim directory:

```
cargo build --release
```

3. Output

The software counts `hits`, `misses` and `evictions`.

`Hits` occur when the processor requests data from memory, and the data is found in the cache.

`Misses` occur when the processor requests data from memory, and the data is not found in the cache.

`Evictions` occur when the cache reaches its capacity and existing data must be removed to make space for new data.

The software outputs a string using the following format:

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

4. Scope and Assumptions

The term “cache” in this document refers to a simulated cache meaning that this software does not store any data.

It is assumed that the provided trace file contains one instruction per line in the following format**:**

```
Z x,y
```

`Z` is a single character representing the Instruction type (***L*** (load), ***M*** (modify) or ***S*** (store))
x is a positive number containing the address of the operation
y is a positive number (>0) containing the size of the data

if `Z` is the character “**I**”, it is ignored, as implementing this instruction is not within the scope of this project.

While it's not strictly required, it is assumed that the user has already navigated into the 'sim' directory. Traces should be placed in the 'traces' directory, and the software should reside within the 'sim/target/release/' directory. Although the software is designed to accommodate any directory for traces and execution, testing has been conducted primarily in this configuration. Hence, it is strongly advised to follow this setup for your own testing purposes.

There are various checks in place which verify that the above assumptions are met by the user, which can be found in the main.rs and cache.rs files.
The test/validity_tests.rs file contains a list of false inputs, purposefully showing how to trigger the software's error messages.

5. Project Structure

The project contains six key files, each serving the purposes outlined below. All files regarding the logic are located in **sim/src/,** while all files handling testing can be found in **sim/tests/.**

sim/src/cache.rs contains the main logic of the cache simulator.

sim/src/main.rs checks and processes the users the arguments and passes them to the cache.rs file. Important when the user runs the software from the command-line.

sim/tests/unit_tests.rs includes a comprehensive suite of tests designed to validate a wide range of inputs across all trace files without any reference to the reference simulator.

sim/tests/integration_tests.rs contains tests which verify that the output of the simulated cache are identical to that of the reference simulator.

sim/tests/validity_tests.rs includes a comprehensive suite of tests designed to check purposefully wrong inputs, and verifies correct responses from the program.

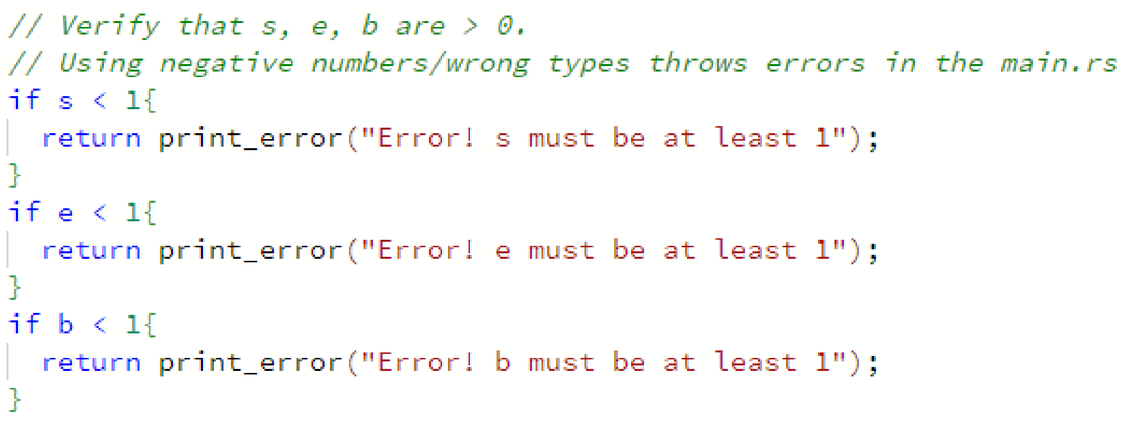
sim/target/release/sim contains the current newest build of the software, which can be called with the `-s -E -b -t` arguments.

6. Implementation

In this section, I will describe the internal workings of the software, which are encapsulated within the "main.rs" and "cache.rs" files.

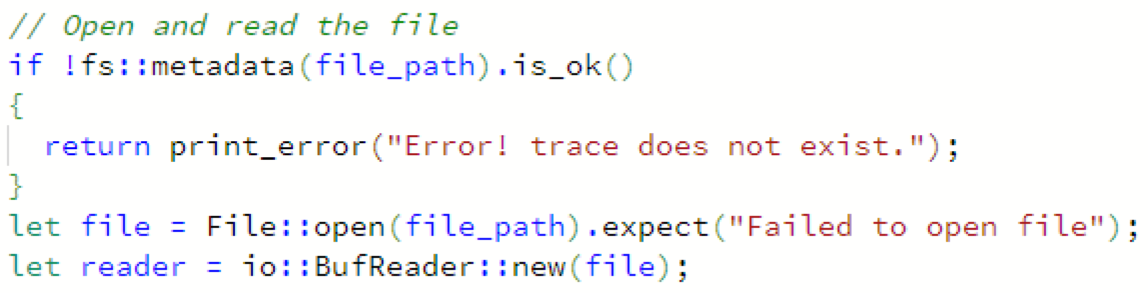
The main.rs file oversees the validation of input correctness and the parsing of arguments necessary for the cache.rs file. This includes verifying the argument length, type and order. All of this functionality is contained within the main function.

The main function passes these arguments into the cache.rs file, where the ranges for `s`, `E` and `b` are checked and verified.



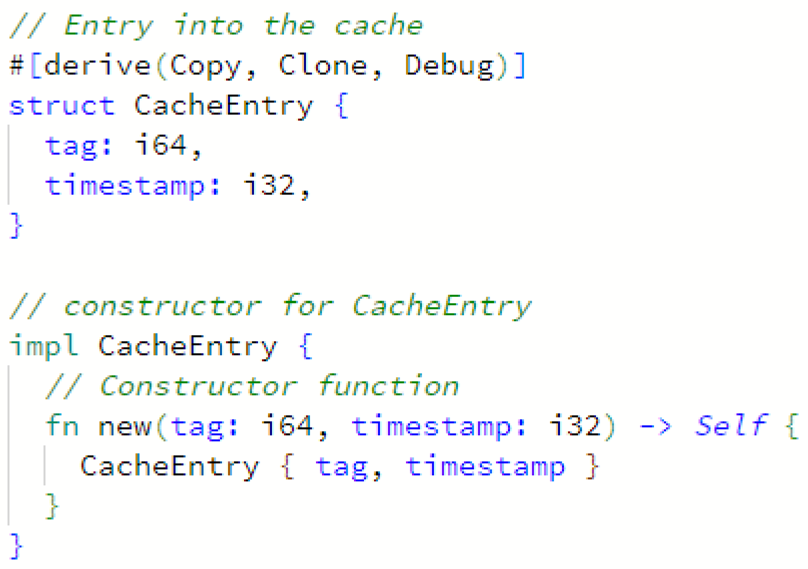
```
// Verify that s, e, b are > 0.
// Using negative numbers/wrong types throws errors in the main.rs
if s < 1{
    return print_error("Error! s must be at least 1");
}
if e < 1{
    return print_error("Error! e must be at least 1");
}
if b < 1{
    return print_error("Error! b must be at least 1");
}
```

At this point it is assumed that `s`, `E` and `b` are correct and valid inputs. If there is a error in the trace file (non-existent, broken) errors are thrown in the upcoming simulation process.



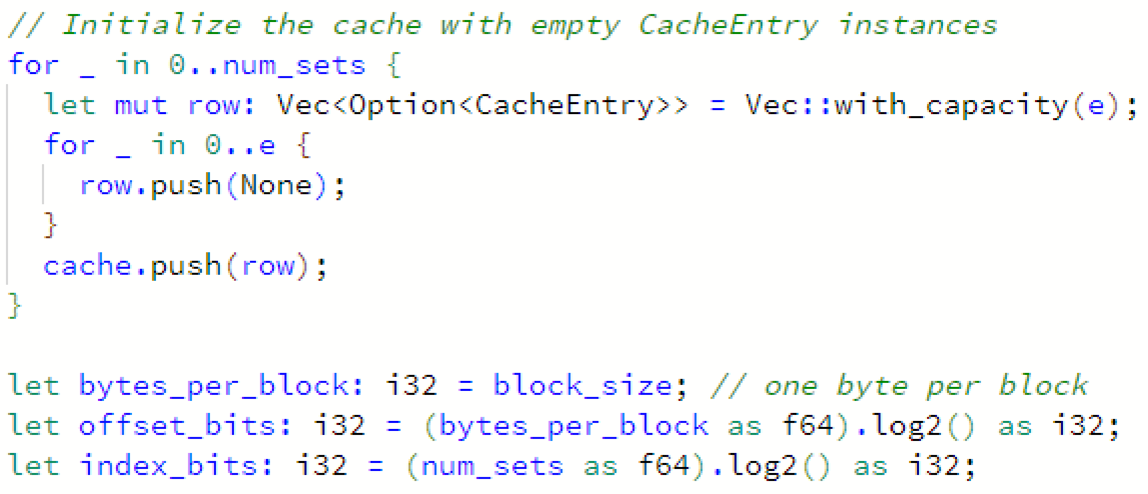
```
// Open and read the file
if !fs::metadata(file_path).is_ok()
{
    return print_error("Error! trace does not exist.");
}
let file = File::open(file_path).expect("Failed to open file");
let reader = io::BufReader::new(file);
```

To simulate the cache, the following data structure was created:



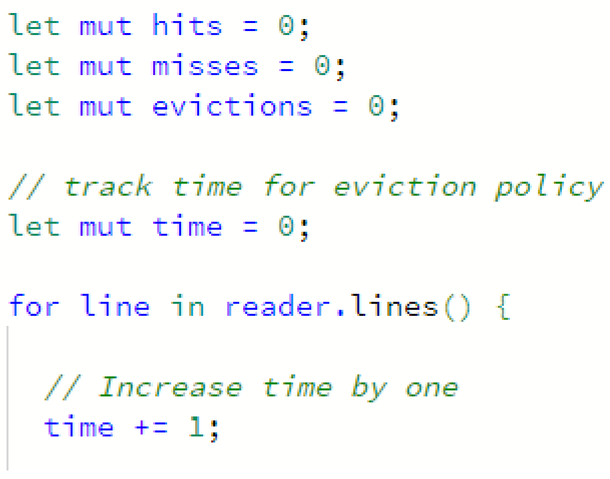
```rust
// Entry into the cache
#[derive(Copy, Clone, Debug)]
struct CacheEntry {
  tag: i64,
  timestamp: i32,
}

// constructor for CacheEntry
impl CacheEntry {
  // Constructor function
  fn new(tag: i64, timestamp: i32) -> Self {
    CacheEntry { tag, timestamp }
  }
}
```

A cache entry contains the tag and the timestamp of the data entry. The timestamp is recorded on entry for the Least-Recently-Used eviction policy.



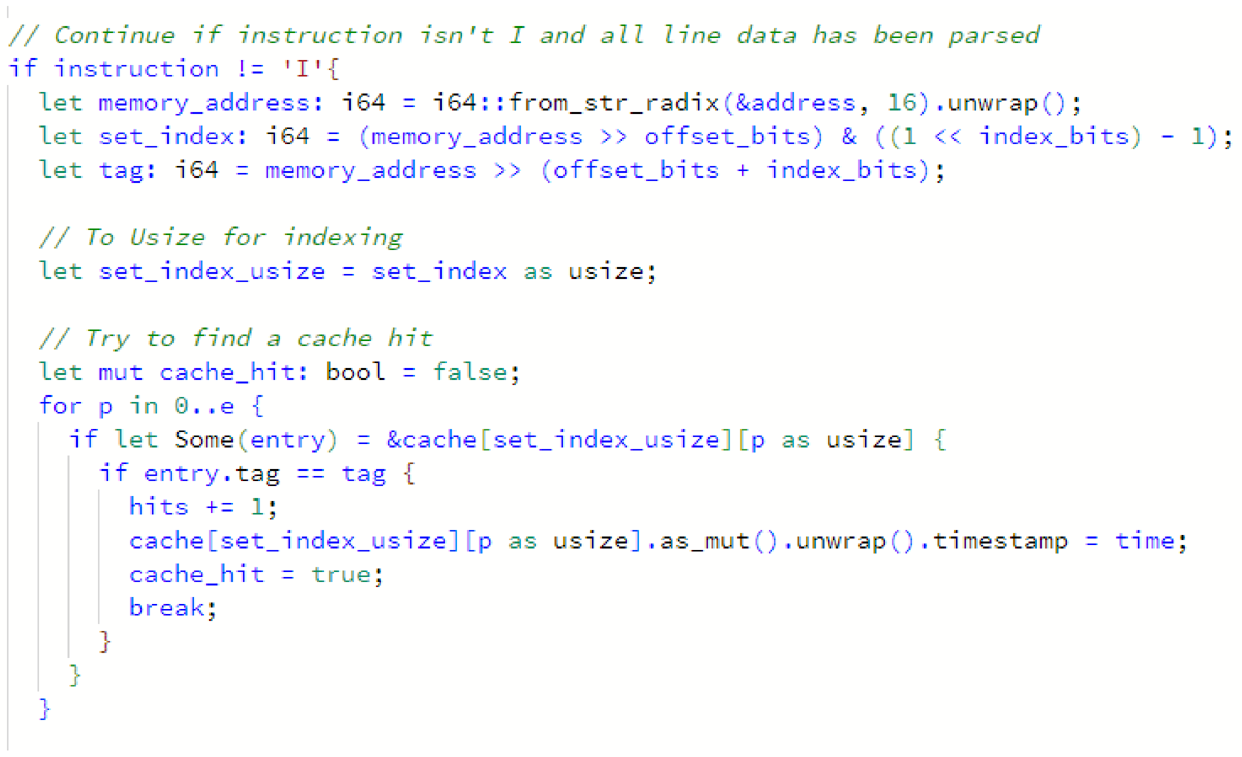
```rust
// Initialize the cache with empty CacheEntry instances
for _ in 0..num_sets {
  let mut row: Vec<Option<CacheEntry>> = Vec::with_capacity(e);
  for _ in 0..e {
    row.push(None);
  }
  cache.push(row);
}

let bytes_per_block: i32 = block_size; // one byte per block
let offset_bits: i32 = (bytes_per_block as f64).log2() as i32;
let index_bits: i32 = (num_sets as f64).log2() as i32;
```

After initializing a empty cache, the software iterates through each line in the trace file, and starts counting hits, misses and evictions.



```
let mut hits = 0;
let mut misses = 0;
let mut evictions = 0;

// track time for eviction policy
let mut time = 0;

for line in reader.lines() {

    // Increase time by one
    time += 1;
```

If a instruction (line in trace file) cannot be interpreted, a error is thrown. When a instruction is successfully read, the software first checks for a hit within the cache. This is done by checking the location and tag within the cache. This occurs regardless of the instruction type (except "I" of course).



```
// Continue if instruction isn't I and all line data has been parsed
if instruction != 'I'{
  let memory_address: i64 = i64::from_str_radix(&address, 16).unwrap();
  let set_index: i64 = (memory_address >> offset_bits) & ((1 << index_bits) - 1);
  let tag: i64 = memory_address >> (offset_bits + index_bits);

  // To Usize for indexing
  let set_index_usize = set_index as usize;

  // Try to find a cache hit
  let mut cache_hit: bool = false;
  for p in 0..e {
    if let Some(entry) = &cache[set_index_usize][p as usize] {
      if entry.tag == tag {
        hits += 1;
        cache[set_index_usize][p as usize].as_mut().unwrap().timestamp = time;
        cache_hit = true;
        break;
      }
    }
  }
```

When there is no hit, we record a miss. Furthermore, it is checked if a eviction occurs. This occurs if a spot In the cache must be replaced according the Least-Recently-Used eviction policy.

The function `find_empty` finds a empty valid spot within the cache. `find_oldest` is used when no valid empty spot could be found and a eviction must occur (-1). These functions are described in more detail in the next chapter.

```
//Miss!
if !cache_hit {
    misses += 1;

    //Attempt to find a empty place in the cache
    let find_empty = find_empty(set_index_usize, e, cache.clone());

    if find_empty == -1 {

        // Attempt has failed, replace oldest instead.
        let newest_index = find_oldest(set_index_usize, e, cache.clone()) as usize;
        if cache[set_index_usize][newest_index].is_some() {
            evictions += 1;
        }
        cache[set_index_usize][newest_index] = Some(CacheEntry::new(tag, time));

    } else {

        //Attempt was successful, add here.
        cache[set_index_usize][find_empty as usize] = Some(CacheEntry::new(tag, time));
    }
}
```

A additional hit is recorded when the instruction is set to modify:

```
if instruction == 'M'{
    hits += 1; //Modify always adds a additional hit
}
```

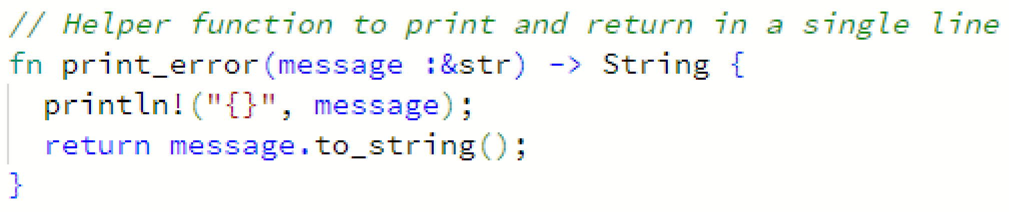
After the above procedure is completed for all lines within the trace file, the following final code is executed:

```
let result_string = format!("hits:{} misses:{} evictions:{}", hits, misses, evictions);
println!("{}", result_string); // final result printing
return result_string; //returned for tests
```

7. Functions

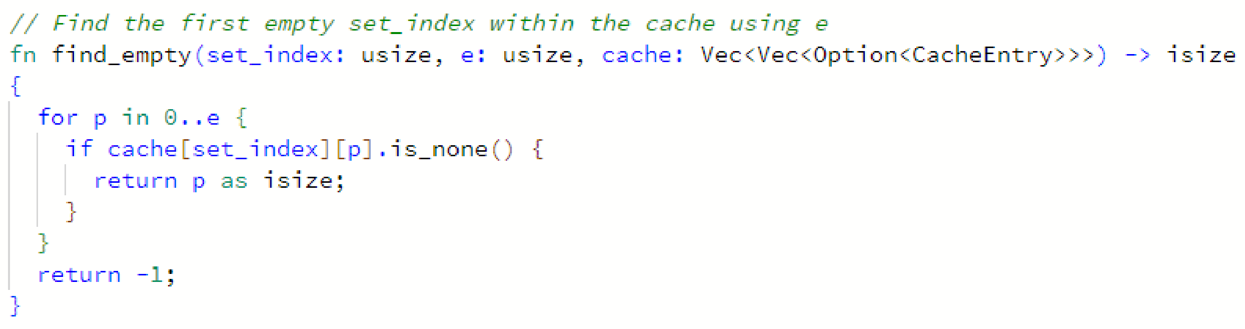
There are three helper functions within cache.rs

`print_error` is a function which prints a message and returns it. It was created to save space.



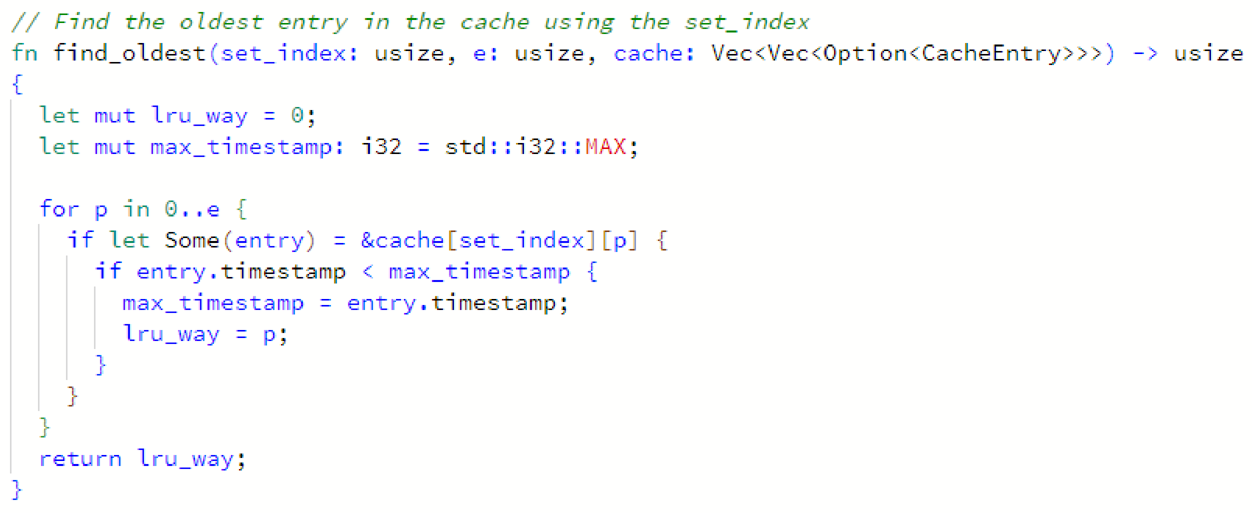
```rust
// Helper function to print and return in a single line
fn print_error(message :&str) -> String {
  println!("{}", message);
  return message.to_string();
}
```

`find_empty` is used to find a empty place within a cache using `setindex` and `E`. `setindex` is calculated using the instruction's address while `E` is a parameter given by the user.
if no empty item can be found, -1 is returned, which means a item must be evicted. If the function is successful in finding a empty place, the index is returned.



```rust
// Find the first empty set_index within the cache using e
fn find_empty(set_index: usize, e: usize, cache: Vec<Vec<Option<CacheEntry>>>) -> isize
{
  for p in 0..e {
    if cache[set_index][p].is_none() {
      return p as isize;
    }
  }
  return -1;
}
```

`find_oldest` is used to find the oldest item in the cache using `set_index` and `E`. This function is used for the least-recently used policy, and returns the index of the item that must be evicted next. Using this function assumes a item must be evicted.



```rust
// Find the oldest entry in the cache using the set_index
fn find_oldest(set_index: usize, e: usize, cache: Vec<Vec<Option<CacheEntry>>>) -> usize
{
  let mut lru_way = 0;
  let mut max_timestamp: i32 = std::i32::MAX;

  for p in 0..e {
    if let Some(entry) = &cache[set_index][p] {
      if entry.timestamp < max_timestamp {
        max_timestamp = entry.timestamp;
        lru_way = p;
      }
    }
  }
  return lru_way;
}
```

8. Testing

Tests can be run within the sim directory with the command “`Cargo Test`”. All tests have been verified to pass. The testing is compromised of the following three main testing suites:

8.1. Unit Tests

The unit tests evaluate the simulator's behaviour using a variety of valid inputs and all of the provided traces. The unit tests are designed to test a large range of values to validate the accuracy of the output.

There are a total of 7 unit tests, all following this layout:

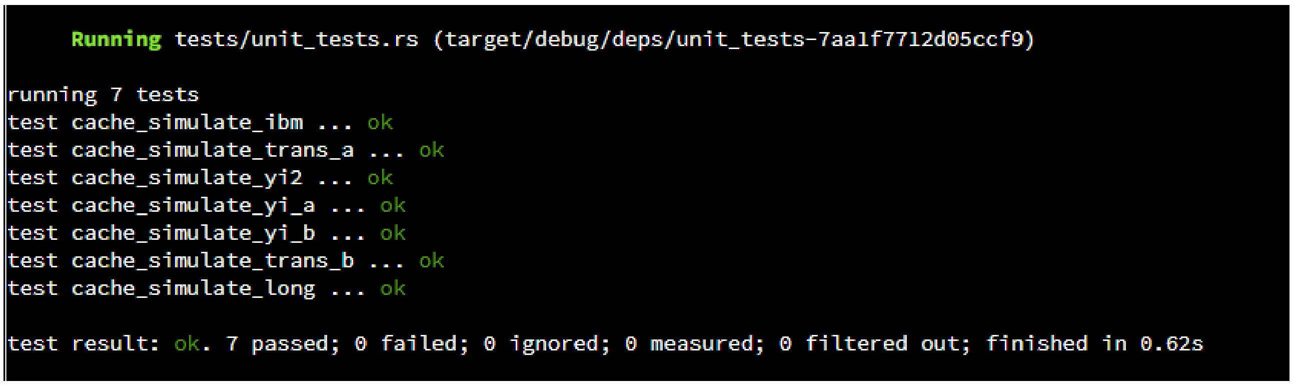
```
#[test]
fn cache_simulate_yi_a() {
    let input_path = "/home/codio/workspace/traces/yi.trace";
    let (input_s, input_e, input_b) = (4, 1, 4);

    let output_expected = "hits:4 misses:5 evictions:3";
    let output_actual : String = simulate_cache(input_s, input_e, input_b, &input_path.to_string());

    assert_eq!(&output_expected, &output_actual);
}
```

At first, we define the input parameters: the file path, s, E and b.
the `output_expected` variable contains what we want the cache to output, while `output_actual` contains the actual output of the cache.

The function simulate_cache is taken directly from sim/src/cache.rs:

```
#[path = "../src/cache.rs"]
mod cache;
use cache::simulate_cache;
```

`assert_eq!` confirms that the actual output matches with our expected values:



```
     Running tests/unit_tests.rs (target/debug/deps/unit_tests-7aa1f7712d05ccf9)

running 7 tests
test cache_simulate_ibm ... ok
test cache_simulate_trans_a ... ok
test cache_simulate_yi2 ... ok
test cache_simulate_yi_a ... ok
test cache_simulate_yi_b ... ok
test cache_simulate_trans_b ... ok
test cache_simulate_long ... ok

test result: ok. 7 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.62s
```

8.2. Integration Tests

The integration tests compare the output of the cache simulator with that of the reference simulator. It ensures that the program behaves consistently with the reference simulator, validating its functionality in a broader context.

The five Integration tests are setup like this:

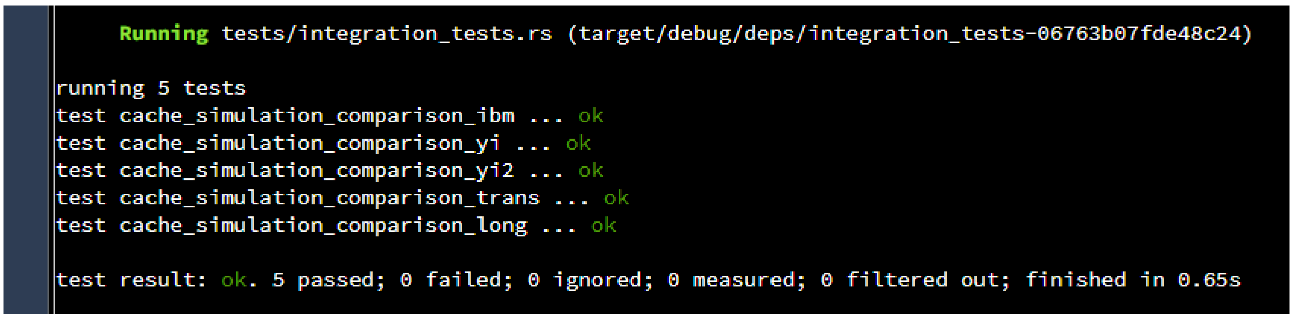
```rust
#[test]
fn cache_simulation_comparison_yi() {
    verify_identical_output_to_simref("/home/codio/workspace/traces/yi.trace")
}
```

Each of the five tests uses a different trace file. `verify_identical_output_to_simref` is a function which checks that the output from our cache meets the output from the reference simulator:

```rust
fn verify_identical_output_to_simref(path : &str) {

    // Default Input
    let (input_s, input_e, input_b) = (2, 4, 6);

    // Start sim-ref
    let output = Command::new("/home/codio/workspace/sim-ref")
        .arg("-s").arg(input_s.to_string())
        .arg("-E").arg(input_e.to_string())
        .arg("-b").arg(input_b.to_string())
        .arg("-t").arg(path)
        .output()
        .expect("Failed to execute command");

    if output.status.success() {
        // Get the output from sim-ref
        let output_expected = String::from_utf8_lossy(&output.stdout);

        // Get the output from cache.rs
        let output_actual : String = simulate_cache(input_s, input_e, input_b, &path.to_string());

        //Verify identical output. Sim-Ref contains additional \n.
        assert_eq!(&output_expected, &(output_actual.to_owned() + "\n"));

    } else {
        let error_str = String::from_utf8_lossy(&output.stderr);
        panic!("Error : {}", error_str);
    }
}
```

Similar to the unit tests, we include the `simulate_cache` function and define `output_actual` vs `output_expected`. To receive the real output of the sim-ref, we include the following:

```
#[path = "../src/cache.rs"]
mod cache;
use std::process::{Command};
use cache::simulate_cache;
```



```
     Running tests/integration_tests.rs (target/debug/deps/integration_tests-06763b07fde48c24)

running 5 tests
test cache_simulation_comparison_ibm ... ok
test cache_simulation_comparison_yi ... ok
test cache_simulation_comparison_yi2 ... ok
test cache_simulation_comparison_trans ... ok
test cache_simulation_comparison_long ... ok

test result: ok. 5 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.65s
```

8.3. Validity Tests

The seven validity tests verify whether the cache produces correct outputs when provided with invalid inputs. They aim to assess the ability to handle unexpected or incorrect inputs.

The initial three tests ensure that the program stops when the variables `s`, `E`, and `b` are set to zero. Each of these tests are structured like the example below:

```
#[test]
fn cache_simulate_s_is_zero() {
  let input_path = "/home/codio/workspace/traces/yi.trace";
  let (input_s, input_e, input_b) = (0, 1, 4);

  let output_expected = "Error! s must be at least 1";
  let output_actual : String = simulate_cache(input_s, input_e, input_b, &input_path.to_string());

  assert_eq!(&output_expected, &output_actual);
}
```

It's worth mentioning that the program encounters failures with values below zero as well. In these cases, the program triggers an error either via the main.rs in the command line or through a build error when incorporated into your own code (input type). The reason we specifically check for exactly zero is because this error had to be intentionally designed to be triggered, unlike negative values. Negative value issues prompt errors from Rust's error handling mechanism.

The fourth test checks the output of the program in cases where a argument is missing. Here we call the command to run our cache simulator to verify that the program responds with the default usage message:

```rust
#[test]
fn cache_simulate_missing_b() {

    // Inputs
    let (input_s, input_e) = (4, 8);
    let path = "/home/codio/workspace/traces/ibm.trace";

    // Start sim
    let output = Command::new("/home/codio/workspace/sim/target/release/sim")
        .arg("-s").arg(input_s.to_string())
        .arg("-E").arg(input_e.to_string())
        .arg("-t").arg(path)
        .output()
        .expect("Failed to execute command");

    let expected_output = "Usage: /home/codio/workspace/sim/target/release/sim <s> <E> <b> <trace_file_path>\n";
    let output_actual = String::from_utf8_lossy(&output.stdout);

    assert_eq!(output_actual, expected_output);
}
```

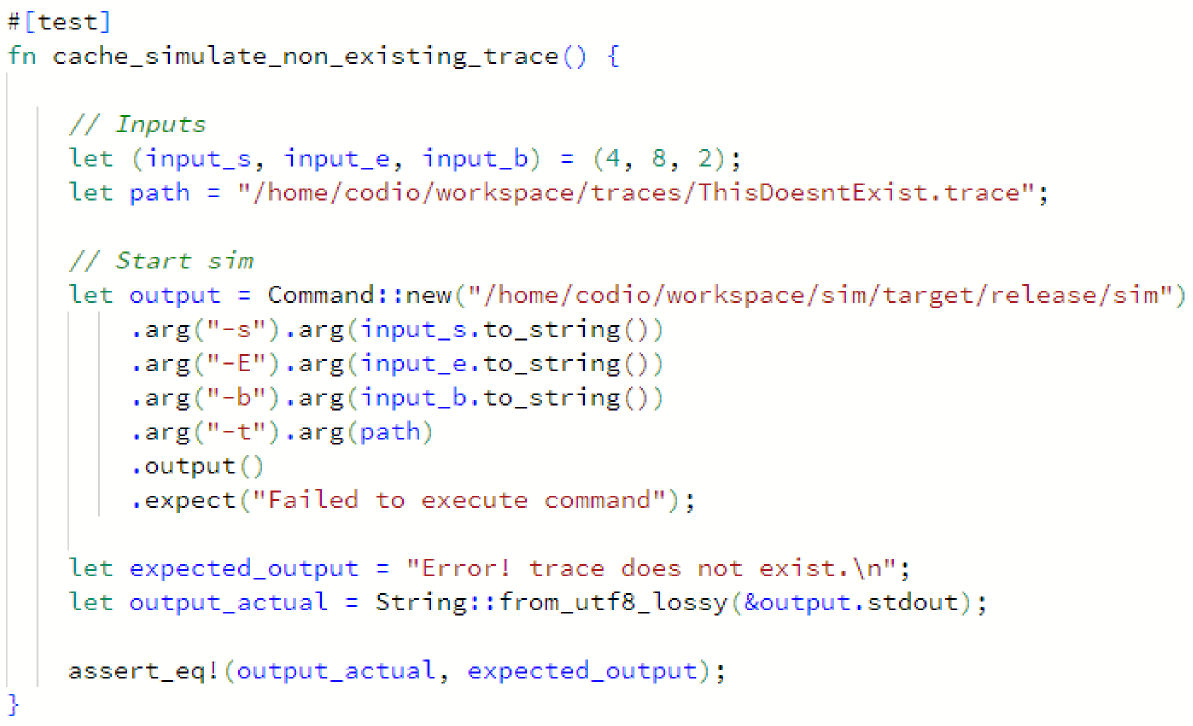
The fifth test is similar, but here we don't provide a trace:

```rust
#[test]
fn cache_simulate_missing_trace_input() {

    // Inputs
    let (input_s, input_e, input_b) = (4, 8, 2);

    // Start sim
    let output = Command::new("/home/codio/workspace/sim/target/release/sim")
        .arg("-s").arg(input_s.to_string())
        .arg("-E").arg(input_e.to_string())
        .arg("-b").arg(input_b.to_string())
        .output()
        .expect("Failed to execute command");

    let expected_output = "Usage: /home/codio/workspace/sim/target/release/sim <s> <E> <b> <trace_file_path>\n";
    let output_actual = String::from_utf8_lossy(&output.stdout);

    assert_eq!(output_actual, expected_output);
}
```

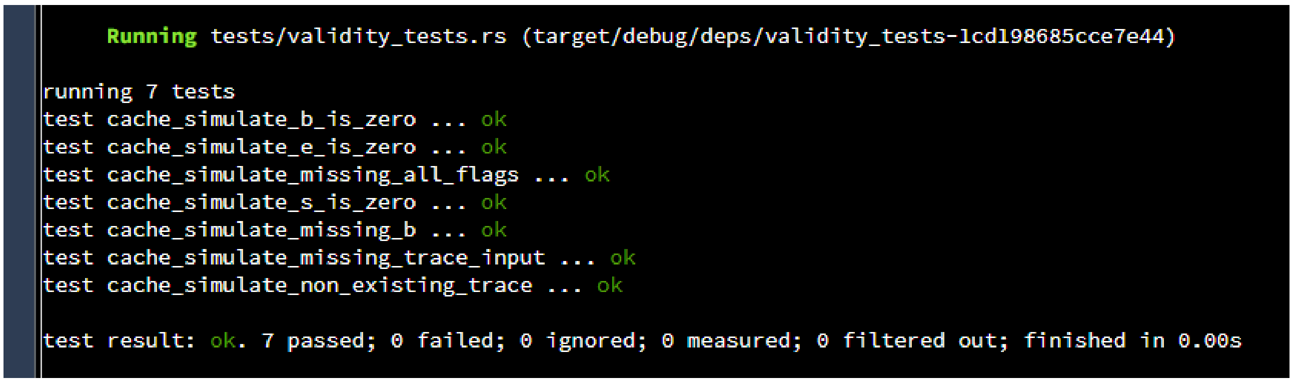
In the sixth test, we do not provide any arguments at all:

```rust
#[test]
fn cache_simulate_missing_all_flags() {
    // Start sim
    let output = Command::new("/home/codio/workspace/sim/target/release/sim")
        .output()
        .expect("Failed to execute command");

    let expected_output = "Usage: /home/codio/workspace/sim/target/release/sim <s> <E> <b> <trace_file_path>\n";
    let output_actual = String::from_utf8_lossy(&output.stdout);

    assert_eq!(output_actual, expected_output);
}
```

In the final test case we do provide all flags, but provide a trace path to a file which doesn't exist:



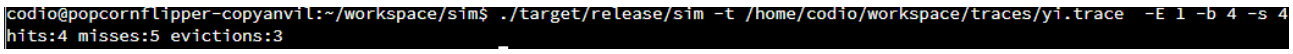
```rust
#[test]
fn cache_simulate_non_existing_trace() {

    // Inputs
    let (input_s, input_e, input_b) = (4, 8, 2);
    let path = "/home/codio/workspace/traces/ThisDoesntExist.trace";

    // Start sim
    let output = Command::new("/home/codio/workspace/sim/target/release/sim")
        .arg("-s").arg(input_s.to_string())
        .arg("-E").arg(input_e.to_string())
        .arg("-b").arg(input_b.to_string())
        .arg("-t").arg(path)
        .output()
        .expect("Failed to execute command");

    let expected_output = "Error! trace does not exist.\n";
    let output_actual = String::from_utf8_lossy(&output.stdout);

    assert_eq!(output_actual, expected_output);
}
```



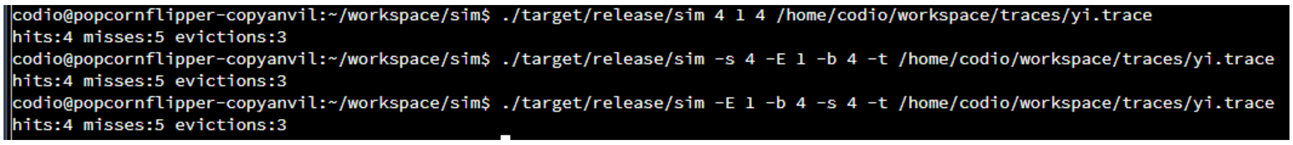
```
     Running tests/validity_tests.rs (target/debug/deps/validity_tests-1cd198685cce7e44)

running 7 tests
test cache_simulate_b_is_zero ... ok
test cache_simulate_e_is_zero ... ok
test cache_simulate_missing_all_flags ... ok
test cache_simulate_s_is_zero ... ok
test cache_simulate_missing_b ... ok
test cache_simulate_missing_trace_input ... ok
test cache_simulate_non_existing_trace ... ok

test result: ok. 7 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s
```

8.4. Additional Tests

Additionally I have confirmed that all flags can be provided in random order. Furthermore, if no specific flag order is given, the default order of -s -E -b -t is utilized:



```
codio@popcornflipper-copyanvil:~/workspace/sim$ ./target/release/sim -t /home/codio/workspace/traces/yi.trace  -E 1 -b 4 -s 4
hits:4 misses:5 evictions:3
```



```
codio@popcornflipper-copyanvil:~/workspace/sim$ ./target/release/sim 4 1 4 /home/codio/workspace/traces/yi.trace
hits:4 misses:5 evictions:3
codio@popcornflipper-copyanvil:~/workspace/sim$ ./target/release/sim -s 4 -E 1 -b 4 -t /home/codio/workspace/traces/yi.trace
hits:4 misses:5 evictions:3
codio@popcornflipper-copyanvil:~/workspace/sim$ ./target/release/sim -E 1 -b 4 -s 4 -t /home/codio/workspace/traces/yi.trace
hits:4 misses:5 evictions:3
```

9. Development Cycle

As a newcomer to Rust, this project has significantly contributed to my skill development in this programming language. But as a newcomer, I chose to create a prototype using C# first, a language I'm already comfortable with. This allowed me to fully understand cache simulators before diving into Rust.

The initial C# code is located within the project directory at sim/other/prototype.cs. It's worth noting that this code isn't essential for the project at all and can be disregarded or removed entirely. It's not designed for practical use.

I included the C# code solely to illustrate the development cycle and to provide a comparison of the logic between Rust and C#, which was valuable for me personally.