[redacted] Computer Systems 2023

Cache Sim Report

Tom Taylor [redacted]

Table of Contents

Introduction

This report details the design and implementation of *sim*, a small Rust program which simulates a simplified model of a memory cache.

The report will give a brief overview of the code, offer the reasoning behind decisions made and cover the testing methodology used.

Scope and Requirements

Overview

The simulated cache consists of a defined number of **sets** each containing a quantity of **lines** which each contain **blocks** of data fetched from external memory as well as metadata. The location in which to store each address in the cache is derived from certain bit fields of their address.

The number of **sets**, **lines** and **blocks** in the cache is determined by the user via the command line upon execution and is intrinsically linked to the bit field index sizes.

There are three possible operations to the cache. A **Load** represents loading an address from memory, and a **Store** represents saving an address into memory, a **Modify** can be thought of a subsequent **Load** then **Store**.

When an instruction and address are provided to the simulation, several results are possible. If the address is already in the cache then a **Hit** is recorded, if the address is not present but there is a space in the cache then we register a **Miss**, if the address is not present and there is no space, then we register an **Eviction** as the **Least-Recently-Used** address is evicted from that section of the cache and replaced with the requested address.

Input

The *sim* program receives from the command line a set of parameters and the location of a trace file, if any of these parameters are missing or erroneous the program will exit with a descriptive error message. The program must handle terminal input and the opening of a file from disk.

The trace file is a plain text file stored on the local filesystem which contains multiple lines representing an operation for the cache; an **Instruction**, an **Address** (a 64-bit integer in hexadecimal) and a **Block** index (which is unused). e.g.

```
:  L  ffff,2
```

Output

Upon successful processing of the provided source file, the frequencies of Hits, Misses and Evictions is printed to the terminal and the program exits. e.g.

```
: hits:1 misses:6 evictions:1
```

If the program is run with the '*verbose*' flag then at each line of the trace file, the program will print the line input and the result of that operation, and then terminate normally as above.

A demonstration implementation has been given with which to test and compare functionality during development. (*sim-ref*)

Assumptions

Assumptions are made for the handling of user and file input, namely that they are logical and have the correct format. Any egregious errors should be handled and halt the running of the program but given the limits of the brief, we can not handle every possible edge case. The file is assumed to be in the correct format and to not be so large as to crash the program. The parameters given for the cache are assumed to be sensible choices and not so large or small as to cause issues, for example having a combined length longer than a 64 bit integer.

Implementation Description

Data Structures

Cache

This is a struct which represents the cache itself, it has the following fields;

- *sets*, representing the cache contents
- *set_bits, block_bits* to store the bit fields for address processing
- *miss, hit, evict* to store the frequencies of each result on the cache

Sets is a Vector of sets which themselves contain multiple Vectors containing Lines. (Nesting Vectors is not recommended for performance reasons but profiling is outside the scope of this project.)

A vector has to be used instead of a fixed sized array because we do not know the amount of sets and lines at compile time. Because the lines objects will be used in a LRU manner, experiments were made using Vector Double-Ended Queues to represent the lines, as this offered no advantage in speed the simplest option (Vector) was chosen.

The vector expands and contracts as objects are added so there is no need to store a 'valid' bit to signify that a segment contains data as we might in a static array. As the cache should always be working with 64 bit addresses it makes sense to have the lines contents, set_bits and block_bits values as unsigned 64 bit integers (u64).

Representing a cache as a struct with 'impl' functions provides a clean abstraction as opposed to global variables, facilitates ownership for the borrow checker and allows multiple instances of caches within the same program for testing and further experimentation.

Structs and Enums

To provide a level of abstraction and ease development and readability, enumerations were used to represent certain values in the program rather than using strings or 'magic numbers'. Structs were employed to represent key objects and facilitate the passing of multiple values between functions.

Values represented as *enums*:

- CacheInstruction (Representing the 4 operations Load, Store, Modify, Instruction)
- CacheResult (One of 3 results; Hit, Miss, Eviction)

Objects defined as *structs*:

- Address (an address split into the tag bits and set bits)
- Cmd (representing an address and instruction parsed from the trace file)
- ArgFlags (representing the flags passed from command line values)

Key Functions Overview

Cache Functions

new - This function takes values for set bits, block bits, lines and returns a new 'cache' struct with the correct number and capacity of lines and sets. The function assumes that the values given would result in a valid and logical cache configuration.

process_address - This function takes a single u64 integer and splits it into an enum containing the tag and set bits, depending on the settings of the cache. The block bits are discarded. This is performed once upfront to avoid having to re-calculate multiple times in our code. The simplest way to perform this operation with unsigned integers and bitfields, is with bitwise operations and masking.

operate - This function takes a single integer argument, it converts it into a valid **Address** with *process_address* then performs a cache access to determine if the result is a **Hit, Miss** or **Eviction** and the cache statistics are updated. An action may have multiple results and they are appended to a Vector which is returned.

run_command - This function is responsible for operating the cache in the appropriate manner for the operation read. If the operation is **Modify**, then it is necessary to check the cache twice. If the operation is **Load** or **Store,** the cache only needs checking once and if the operation is an **Instruction** we do nothing. This function concatenates all the results for the operation and returns them to the calling function.

Using these two functions *operate* and *run_command* simplifies the program logic and prevents unnecessary repetition of code and long branching structures for each combination of operation and result.

In addition there are functions which take and address perform the atomic cache actions; *check_hit, update, check_free, insert* and *evict*. After processing the instructions, *cache_results* is used to access the global statistics of the cache for printing to output.

File Functions

process_input_file - This is the main "loop" of the program which opens the provided trace file and iterates over each line. The address and instruction is extracted and passed into the cache. If the program is running in verbose mode, the vector of "Results" returned is printed to the screen along with the line.

Terminal Functions

process_args - takes a slice of the input arguments from the command line using the *getopts* crate and creates a struct of the various flags. This function is mostly boiler plate to parse values, check for errors, perform integer conversion and print error messages.

Directory Structure

The directory layout is as follows:

```
sim/
├── Cargo.lock
├── Cargo.toml
├── src/
│   ├── lib.rs
│   └── main.rs
├── target/
├── tests/
│   └── test.rs
└── tools/
    ├── linecount.sh
    └── linecount.txt
```

Files Description

src/main.rs

This is the main "entry-point" into the program, it calls the functions defined in lib.rs to read a file, create a cache and process the data into the cache.

src/lib.rs

This is the "library" for the program, containing all the functions and structure definitions.

target/

Directory where the compiled program is stored.

tests/test.rs

This is the location for all the integration tests for the program.

Testing Methodology

Automated Testing

Integration tests were written using Rust's built-in Cargo utilities. Each aspect of the program was tested where possible and the scope of each test is explained in the source code comments.

Tests were written for checking the various parsing functions (*process_args, line_to_command*) to ensure behaviour is as expected and that bad inputs cause an exception or error message. Tests with different cache sizes and tracefile inputs were compared against results from sim-ref.

All the cases tested pass and all the tracefiles tested return the same results as sim-ref. Invalid command line arguments cause the program to exit, causing problems for the automated tests. In addition the arguments were tested manually from the terminal. See Appendix for test reports.

Development Cycle and Decisions

Cycle

During development git was used to create backups, track the project history and create new branches for experimenting with new functionality and refactoring. Using the Cargo testing framework ensured that helper functions could be developed and tested incrementally, before being integrated in the eventual implementation.

Structure and Methodology

The original intent for the project structure was to adopt a semi-modular approach and isolate each functionality into separate modules, the idea being that it would make it easier to test and produce a stable foundation for further expansion. During development it became clear that this was **not** the intended approach from the brief so the project was combined into one crate to keep the source code size within the set limits.

The quality-of-life features added to the code, such as use of enums and abstraction do add considerably to the verbosity but this is a fair trade off for readability and robustness, and are optimized to the same result by the compiler.

The data structures employed to represent the cache are suboptimal and efficiency could possibly be improved by switching to a structure such as a hybrid hash map and list, which could offer linear time for checking whether an address is in the cache, for removing/adding items to the cache as well as keeping track of the ordering for the LRU algorithm. The implementation of this is beyond the scope of this project and with Rust might require the use of *unsafe* code or redundancy. The current data structure implementations are more than adequate for the use case.

Further Development

If this project were to be developed further it would make sense to re-divide the code into separate modules for file handling, command line handling and cache functions. This would make the source code easier to manage.

A more methodical system for error handling would be essential if this were to be expanded, as there are situations where the program may crash unexpectedly or produce undefined behaviour.

It would be worthwhile implementing extra eviction policy algorithms (LFU, MRU) and the ability to have offsets of blocks. The program could be expanded to allow for chaining L1-L2-L3 stage caches, or even a custom configuration. As previously mentioned the cache itself could be represented in a more efficient way, to allow for faster execution with large tracefiles.

A textual or even graphical visualisation of what is happening at each stage of the cache operation would be an interesting feature, perhaps with the ability to advance execution step-by-step.

[Word Count : 1933]

Appendix

Test Reports

Cargo Output (More tests to be added...)

```
running 14 tests
test tests::line_to_command_none ... ok
test tests::line_to_command_test ... ok
test tests::line_to_command_panic - should panic ... ok
test tests::run_cache_ibm ... ok
test tests::run_cache_yi ... ok
test tests::test_lru ... ok
test tests::test_direct_cache ... ok
test tests::test_args_success ... ok
test tests::test_operation_insert ... ok
test tests::test_tiny_cache ... ok
test tests::test_tiny_cache_2 ... ok
test tests::test_yi_example_inst ... ok
test tests::run_cache_trans ... ok
test tests::run_cache_long ... ok

test result: ok. 14 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
finished in 2.62s
```

Command Line Arguments Testing

```
$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.00s
     Running `target/debug/sim`
Error: Required option 's' missing
Usage: target/debug/sim [-hv] -s <num> -E <num> -b <num> -t <file>

Options:
    -h, --help          Print this help message.
    -v, --verbose       Optional verbose flag.
    -s num              Number of set index bits.
    -E num              Number of lines per set.
    -b num              Number of block offset bits.
    -t file             Trace file.

$ cargo run - -h
    Finished dev [unoptimized + debuginfo] target(s) in 0.00s
     Running `target/debug/sim - -h`
Usage: target/debug/sim [-hv] -s <num> -E <num> -b <num> -t <file>

Options:
    -h, --help          Print this help message.
    -v, --verbose       Optional verbose flag.
    -s num              Number of set index bits.
    -E num              Number of lines per set.
    -b num              Number of block offset bits.
    -t file             Trace file.

$ cargo run - -s 4
    Finished dev [unoptimized + debuginfo] target(s) in 0.00s
     Running `target/debug/sim - -s 4`
Error: Required option 'E' missing
Usage: target/debug/sim [-hv] -s <num> -E <num> -b <num> -t <file>
```

```
Options:
    -h, --help          Print this help message.
    -v, --verbose       Optional verbose flag.
    -s num              Number of set index bits.
    -E num              Number of lines per set.
    -b num              Number of block offset bits.
    -t file             Trace file.

$ cargo run - -s 4 -E 4
    Finished dev [unoptimized + debuginfo] target(s) in 0.00s
     Running `target/debug/sim - -s 4 -E 4`
Error: Required option 'b' missing
Usage: target/debug/sim [-hv] -s <num> -E <num> -b <num> -t <file>

Options:
    -h, --help          Print this help message.
    -v, --verbose       Optional verbose flag.
    -s num              Number of set index bits.
    -E num              Number of lines per set.
    -b num              Number of block offset bits.
    -t file             Trace file.

$ cargo run - -s 4 -E 4 -b 4
    Finished dev [unoptimized + debuginfo] target(s) in 0.00s
     Running `target/debug/sim - -s 4 -E 4 -b 4`
Error: Required option 't' missing
Usage: target/debug/sim [-hv] -s <num> -E <num> -b <num> -t <file>

Options:
    -h, --help          Print this help message.
    -v, --verbose       Optional verbose flag.
    -s num              Number of set index bits.
    -E num              Number of lines per set.
    -b num              Number of block offset bits.
    -t file             Trace file.

$ cargo run - -s 4 -E 4 -b 4 -t
    Finished dev [unoptimized + debuginfo] target(s) in 0.00s
     Running `target/debug/sim - -s 4 -E 4 -b 4 -t`
Error: Argument to option 't' missing
Usage: target/debug/sim [-hv] -s <num> -E <num> -b <num> -t <file>

Options:
    -h, --help          Print this help message.
    -v, --verbose       Optional verbose flag.
    -s num              Number of set index bits.
    -E num              Number of lines per set.
    -b num              Number of block offset bits.
    -t file             Trace file.

$ cargo run - -s 4 -E 4 -b 4 -t blank
    Finished dev [unoptimized + debuginfo] target(s) in 0.00s
     Running `target/debug/sim - -s 4 -E 4 -b 4 -t blank`
blank: No such file or directory

$ cargo run - -s 4 -E 4 -b 4 -t ../traces/yi.trace
    Finished dev [unoptimized + debuginfo] target(s) in 0.00s
     Running `target/debug/sim - -s 4 -E 4 -b 4 -t ../traces/yi.trace`
hits:5 misses:4 evictions:0
```

```
$ cargo run - -s 4 -E 4 -b 4 -v -t ../traces/yi.trace
    Finished dev [unoptimized + debuginfo] target(s) in 0.00s
     Running `target/debug/sim - -s 4 -E 4 -b 4 -v -t ../traces/yi.trace`
L 10,1 miss
M 20,1 miss hit
L 22,1 hit
S 18,1 hit
L 110,1 miss
L 210,1 miss
M 12,1 hit hit
hits:5 misses:4 evictions:0
```

Documentation and References

Documentation

The Rust Programming Language - *https://doc.rust-lang.org/book/*

The Cargo Book - *https://doc.rust-lang.org/cargo/*

Rust by Example - *https://doc.rust-lang.org/rust-by-example/*

Getopts Documentation - *https://docs.rs/getopts/0.2.20/getopts/index.html*

Articles and References

Yury Zhauniarovich, Testing Errors in Rust - *https://zhauniarovich.com/post/2021/2021-01-testing-errors-in-rust/*

Luca Palmieri, Error Handling in Rust: A Deep Dive - *https://www.lpalmieri.com/posts/error-handling-rust/*

Jim Blandy, Jason Orendorff, and Leonora F.S. Tindall, Programming Rust, 2nd Edition, 2021