Birkbeck
UNIVERSITY OF LONDON

Software Report: Developing a Cache Simulator in Rust

Word Count of report (excluding references): 1917

Introduction

This software aims to simulate the behaviour of a cache, and dependent upon whether the cache replacement policy is Direct Mapped (DMC), Fully Associative (FAC) or Set Associative (SAC), print out the correct number of Cache Misses, Hits and Evictions - hereafter referred to as CM, CH and CE.

Scope and Requirements

Overview and Input

My program must compile or provide usage guidance when given the following four parameters:

```
cargo run -- -s <num> -E <num> -b <num> -t <directory_of_trace_file>
```

Initially my program reads arguments from the command line, informing the user if those commands are appropriate and providing usage guidance if not. I then assign all of these values to variables in the broadest scope possible, allowing access to this information throughout the entire program.

Output

The output adheres strictly to the criteria in the coursework relying on an implemented function within the custom data structure ArrayRepresentationOfCache to do so.

Assumptions

As we do not need to store data I chose to only use the inputted amount of block bits to determine which bits to remove, and to take the following set and tag bits.

If the E value equals to 1 we proceed as if the cache structure is a DMC, as only one cache line exists if this value is 1.

The program should process the trace file with a FAC if the s bits value is equal to 1 (as this results in 1 Set within which you can input whatever tag bits the program is provided with).

And finally the program should assume a SAC if the input of both s and e does not equal one.

Implementation Description

Data Structures

The following data structures were used in the program in order to process the trace file into a usable format as well as simulate the various methods of the cache.

This struct was somewhat of a placeholder and was only used within another function in order to split the binary and tag, and then pass those values onto the final location of the binary which is BinaryInTagSetBlock.

```
struct TupleOfTagAndAddress<'a>{
    tag: &'a str,
    binary: String
}
```

This struct only contains the data that has been properly processed and manipulated, storing the full binary in its tag bit and set bit parts respectively. It also stores the type of memory access which would be Modify (M), Store (S) or Load (L).

```
    struct BinaryInTagSetBlockParts<>{
        type_of_mem_access: String,
        tag_bits: String,
        set_bits: String,
    }
```

This final struct is responsible for holding and passing on the values below to the various methods associated with it. It generates and stores a 2d array which provides a useful abstraction of a cache - this is dynamically generated based on input.

```
pub struct ArrayRepresentationOfCache{
    pub value_of_s_as_usize: usize,
    pub value_of_e_as_usize: usize,
    pub rows_or_cache_sets: usize,
    pub cols_or_cache_lines: usize,
    pub two_d_array: Vec<Vec<String>>,
    pub cache_hits: i32,
    pub cache_misses: i32,
    pub cache_evictions: i32
}
```

Key Functions Overview

Hexadecimal Conversion to Binary

I use a very simple and legible method of conversion, and whilst Hash Maps are not the most efficient means of converting from hex to binary I wanted this function to be easily read as the rest of the program is dependent upon accurate conversion.

Splitting the File

This is perhaps an overly verbose manner of splitting the file, but the process is as follows:

First the trace file is split at every line break (as Valgrind outputs every instruction and then a new line) and these unmodified instructions are placed into a vector of lines, with each element representing one instruction from the CPU.

I then take this vector of lines and using a helper data struct, I split it into the memory address type, and the hexadecimal (which at this point is converted to binary). After this I iterate through every element of the vector of unconverted instructions, splitting them up into the custom data struct BinaryInTagSetBlockParts. I create a vector of all the BinaryInSetTagBlockParts and output this return this from the function for later processing. But to summarise this results in a vector of a custom struct that will be processed in order to establish CH, CM and CE.

DMC Processing

In the eventuality that the cache is a DMC, I created the two following methods to use within my custom struct:

```
pub fn dmc_process(&mut self, set_bits: String, tag_bits: String,
type_of_instruction: String)

pub fn create_two_d_array_with_index_if_dmc(&mut self)
```

The dmc_process first takes the set bits and uses them to find the index that this data would be stored in. As in a DMC there is only a single line that every block from main memory can go into, and as such we do not need to iterate through every line but just check the single index provided to us by the set bits. Once we access that index we can check if the line is empty and if it is we insert the tag - changing the default “empty” String of the two dimensional array to a tag, indicating that this line is now full. This counts as a CM.

The CH occurs when the above process results in the tag being found in the cache meaning that the data is theoretically already stored there.

And finally a CE occurs when none of the other conditions are met, meaning another block that is also stored in the same line has been requested by the CPU, and as in a DMC every block is associated with a particular line an eviction has to occur.

In order to simulate the behaviour of the M type of memory address I have used an if statement to check if this is the case, and increment the data structs cache hits every time it is processed, as the CPU would always access the data after storing it.

In addition to this, we have a method that manipulates the two_d_array in order to give it indexes dependent on the amount of set bits. If, for example, the set bits are 4 bits long we can determine there are 16 potential locations and as such we need 16 unique binary indexes which are generated and inserted by this method.

FAC Processing

Here are the methods used:

```
pub fn has_cache_got_empty_tag_fully_associative(&self) -> bool

fn is_tag_in_cache_fully_associative(&self, block_id: String)->
Option<usize>

pub fn
modify_two_d_array_to_be_correct_rows_and_correct_col_for_fully_associat
ive (&mut self)

pub fn insert_into_cache_if_fully_associative(&mut self, set_bits:
String, tag_bits: String, type_of_instruction: String)
```

The modify_two_d_array_to_be_correct_rows_and_correct_col_for_fully_associative is my solution to the issue that the two_d_array would initialise itself as having 2 sets as opposed to one - as 0 was not allowed as an input for set bits and as s^2 = Sets an input of 1 would result in 2 sets, as opposed to 1. A FAC has only one monolithic set, and this provides that.

The other methods are called within insert_into_cache insert_into_cache_if_fully_associative. This begins with combining the tag bits with the set bits in order to form a full_block_id (which is used in effect as the tag). The block_id is the only element required to determine whether the block is the cache and as such the process of determining if it is a CH; CM is relatively simple. As any block can go anywhere, the only point at which we have to deal with CE is when the cache is full, or no more elements in the set equal to my default value of “empty”.

SAC Processing

Here are the methods used:

```
pub fn modify_cache_structure_for_set_associative(&mut self)

pub fn is_specific_set_empty_set_associative(&self, set_bits: String)->
Option<usize>
```

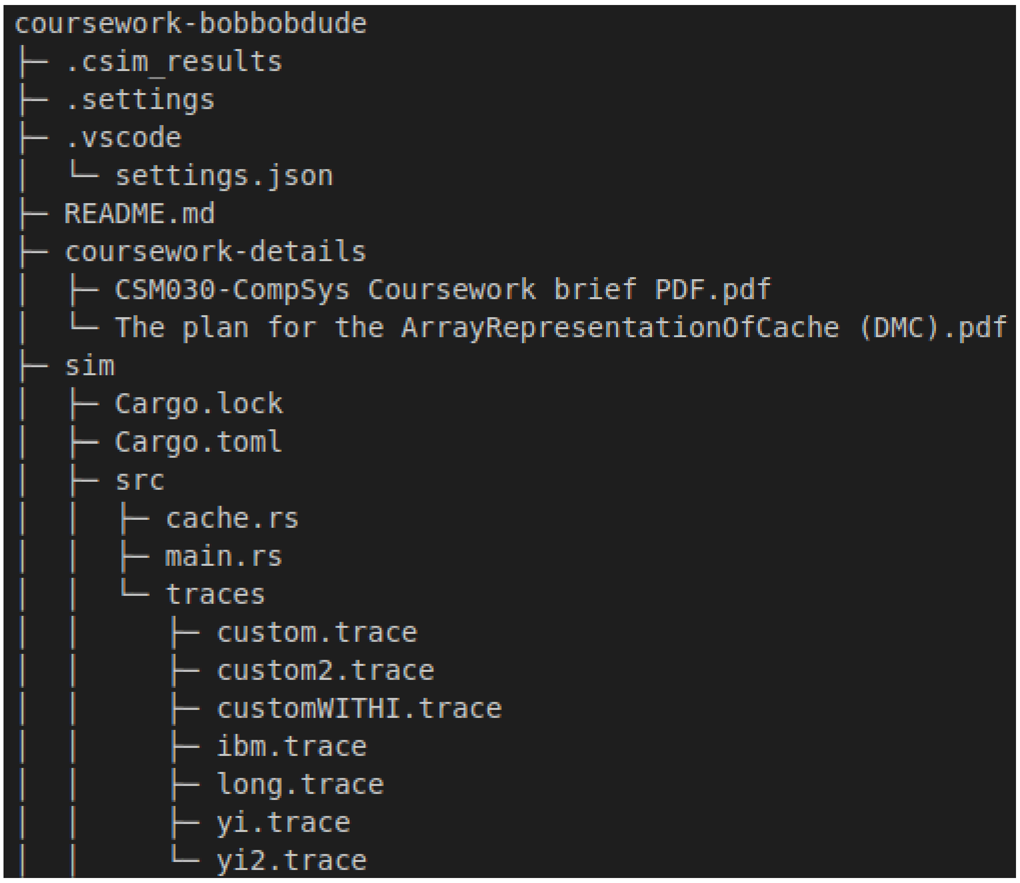
```
pub fn is_tag_in_set(&self, set_bits: String, tag_bits:String) ->
Option<usize>

pub fn set_associative_process(&mut self, set_bits: String, tag_bits:
String, type_of_address: String)
```

The modify_cache_structure_for_set_associative changes the first element of every vector within the 2d vector to a set_address. For example, if the input is -s >= 2 -E >= 2 and b <num> we can conclusively determine this to be a SAC. If it is a SAC we take the number of Sets, which can also be considered the length of the 2d vector created and create a binary representation of every number between 0 and the length. We then input this on the first column and never change it - these can be compared to the set bits of the addresses.

The three following methods are very similar to the above, and check in particular sets if the criteria is met as opposed to every set. Within each set I use a LRU policy, using pop and insert throughout.

Directory Structure and Description

I eventually split my extensive custom data struct out into its own file called cache.rs, making all the functions and the struct itself public so it was available in the main file as well. All of the tests are under a #[cfg(test)] to ensure they do not compile and run when cargo run is used in the terminal.



```
coursework-bobbobdude
├─ .csim_results
├─ .settings
├─ .vscode
│   └─ settings.json
├─ README.md
├─ coursework-details
│   ├─ CSM030-CompSys Coursework brief PDF.pdf
│   └─ The plan for the ArrayRepresentationOfCache (DMC).pdf
├─ sim
│   ├─ Cargo.lock
│   ├─ Cargo.toml
│   ├─ src
│   │   ├─ cache.rs
│   │   ├─ main.rs
│   │   └─ traces
│   │       ├─ custom.trace
│   │       ├─ custom2.trace
│   │       ├─ customWITHI.trace
│   │       ├─ ibm.trace
│   │       ├─ long.trace
│   │       ├─ yi.trace
│   │       └─ yi2.trace
```

Testing Methodology

Testing

All of my tests reside in my main.rs file under the #[cfg(test)] Rust attributes ensure they are not compiled. They cover all of the functions in main, as well as simulating and testing the cache functionality as well, with the test only being completed if the output matches the reference binary output for the same input.

Development Cycle and Decisions

Structure and Methodology

Although this perhaps led to overly verbose code I wanted to ensure that all my methods in cache.rs were different for every cache situation (DMC, FAC, and SAC). This allowed me to experiment with different techniques as I learnt more about rust, gradually improving my ability and confidence over time. To give you a tangible example of this, I have inserted below a line from one of the final commits:

```
self.two_d_array[index_of_set_to_check].retain(|item| item.to_string()
!= tag_bits);
```

After researching various methods of removing tag bits from the array, I came across this. It made my code easier to debug and more succinct than the inefficient combination of saving an index value derived from a for loop, and then using that to delete the value. This methodology and structure of working suited a novice in Rust (such as myself) well.

One mistake I did make was my insistence of using Strings which led to me having to clone values causing my program to be slower and less efficient. My reason for doing this was due to believing that we would have to edit valid, lock and dirty bits at a later stage, which was not the case. So in future, ensuring a comprehensive understanding of the task at hand would help with this.

Further Development

I think the obvious choice of further development for me would be creating a fully working cache memory by storing the data and retaining the block bits that were discarded at the start of my program, to offset into the block. But beyond this, I think working out the anomalies such as my program incorrectly reporting cache evictions sometimes would be a practical first step towards the overarching goal of a functioning cache.

Appendix

Usage

While your working directory is;

```
coursework-bobbobdude/sim$
```

Run the following:

```
cargo run -- -s <num> -E <num> -b <num> -t <directory_of_trace_file>
```

References

Intermation (2020a). *Ep 073: Introduction to Cache Memory*. *YouTube*. Available at: https://www.youtube.com/watch?v=Bz49xnKBH_0 [Accessed 4 Apr. 2024].

Intermation (2020b). *Ep 074: Fully Associative Caches and Replacement Algorithms*. *YouTube*. Available at: https://www.youtube.com/watch?v=A0vR-ks3hsQ [Accessed 4 Apr. 2024].

Intermation (2020c). *Ep 075: Direct Mapped Caches*. *YouTube*. Available at: https://www.youtube.com/watch?v=zocwH0g-qQM [Accessed 4 Apr. 2024].

Intermation (2020d). *Ep 076: Set-Associative Caches*. *YouTube*. Available at: https://www.youtube.com/watch?v=gr5M9CULUZw [Accessed 4 Apr. 2024].

Intermation (2020e). *Ep 077: Cache Write Policies, Flag Bits, and Split Caches*. *YouTube*. Available at: https://www.youtube.com/watch?v=Y7q2ECeFWE8 [Accessed 4 Apr. 2024].

simonsan (2024). *Constructor - Rust Design Patterns*. [online] Github.io. Available at: https://rust-unofficial.github.io/patterns/idioms/ctor.html [Accessed 4 Apr. 2024].

University of London (2024a). *Class Collaborate*. [online] Bbcollab.com. Available at: https://eu.bbcollab.com/collab/ui/session/playback/load/1435ece84ffa4f18ae956997199e6e1d?authToken=eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJiYkNvbGxhYkFwaSIsInJlY29yZGluZ1VpZCI6IjE0MzVlY2U4NGZmYTRmMThhZTk1Njk5NzE5OWU2ZTFkIiwia

XNzIjoiYmJDb2xsYWJBcGkiLCJleHAiOjE3MTIyNzM3NDQsInR5cGUiOjEsImlhdCI6MTcxMjI3MzQ0NCwiY29uc3VtZXIiOiI2YTJiYmFjYTQ4NDQ0NDA3YjJhNGE2NTEyZWVlMGQ1NCJ9.8XbG_n-MI_N6wdYjiDRNhsmAG1dmcniVdY8iz6PORPo [Accessed 4 Apr. 2024].

University of London (2024b). *CSM030-2024-JAN: 7.12 Lecture 8: Cache memory | MyLondon*. [online] London.ac.uk. Available at: https://learn.london.ac.uk/mod/page/view.php?id=146455 [Accessed 4 Apr. 2024].

University of London (2024c). *CSM030-2024-JAN: 7.13 Lecture 9: Cache placement policies | MyLondon*. [online] London.ac.uk. Available at: https://learn.london.ac.uk/mod/page/view.php?id=146456&forceview=1 [Accessed 4 Apr. 2024].

University of London (2024d). *CSM030-2024-JAN: 7.14 Lecture 10: Cache replacement policies | MyLondon*. [online] London.ac.uk. Available at: https://learn.london.ac.uk/mod/page/view.php?id=146457&forceview=1 [Accessed 4 Apr. 2024].

University of London (2024e). *CSM030-2024-JAN: 7.15 Reading: Cache memory principles | MyLondon*. [online] London.ac.uk. Available at: https://learn.london.ac.uk/mod/page/view.php?id=146458&forceview=1 [Accessed 4 Apr. 2024].