

1. Coursework: January to March 2024 session

Submission Deadline: Monday 1st April 2024 at 13.00 UK time

This coursework is weighted at 100% of final mark for the module.

Please Note:

- You are permitted to upload your Coursework in the final submission area as many times as you like before the deadline.
- If you upload the wrong version of your Coursework, you are able to upload the correct version of your Coursework via the same submission area and submit your new version before the deadline. In doing so, this will delete the previous version which you submitted and your new updated version will replace it.
- Please note, when the due date is reached, the version you have submitted last, will be considered as your final submission and it will be the version that is marked.
- Once the due date has passed, it will not be possible for you to upload a different version of your assessment. Therefore, you must ensure you have submitted the correct version of your assessment which you wish to be marked, by the due date.

2. Coursework Description

The summative assessment for this module requires the implementation a program simulating the performance of cache memory and includes the submission of a written report.

The coursework is designed to assess your critical skills developed during this module and requires the development of a practical solution.

The deadline for submission is in Week 12.

- Code must be submitted via [CODIO](#) and must be written in the [Rust](#) programming language.
- The written report must be submitted via [Turn It In](#).

Coursework Brief

The `sim` project folder is provided for the coursework and the `main.rs` file is intentionally empty; you will need to implement all functionality from scratch.

Write a cache simulator with source code placed in file `./sim/main.rs` which takes a memory trace file as input, simulates the hit/miss behaviour of a cache memory on this trace, and outputs the total number of hits, misses, and evictions.

Your program should be able to receive command line arguments (detailed below) and produce identical output as a reference simulator provided as part of the coursework specification (also described in detail below).

3. Software installation

If the repository does not contain the `Rust` infrastructure, or you wish to develop on your own machine, then you will need to:

- install the `C` compiler
- install the `Rust` system
- install the `getopt` crate
- install the `valgrind` utility

The following should be carried out in a terminal window.

Install `C` compiler

```
sudo apt install build-essential
```

Install `Rust` system

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

Select option 1.) for a default install.

Add `getopt` to project

Assuming you have created a Cargo project called `sim`, then change your current directory to `./sim` and type:

```
cargo add getopt
```

Install `valgrind`

```
sudo apt-get install valgrind
```

4. Tasks

This programming task asks you to explore the impact that cache memories can have on the performance of your programs. The task consists of writing a small Rust program that simulates the behaviour of cache memory.

Recall that caching is an optimisation technique, discussed in more detail in Topic 7 of this module, employed in computer systems to keep recent or often-used data in memory locations that are faster or computationally cheaper to access.

Furthermore, recall from Topic 7, that our full model of caching incorporates three distinct levels:

1. L1 cache is located almost as close to the CPU as the registers and hence it is almost as fast. The internal structure of L1 can be represented as a lookup table which employs a memory address (in the RAM) as the key to retrieve the contents of that memory address which can be either data or instructions.
2. L2 cache is farther away from L1 and hence has higher latency but typically more L2 is available.
3. L3 cache is yet farther away from the CPU and has a slower clock but is still much faster than RAM. Much more L3 is typically available, sometimes up to 32 MB.

You can view the cache memory allocated to your Linux virtual machine on Codio using the following command:

```
linux> lscpu
L1d cache: 32K
L1i cache: 32K
L2 cache: 1024K
L3 cache: 36608K
```

The result reveals that the virtual machine has 36608K of L3 and 1024K of L2 cache. As noted, L1 cache stores both data and instruction and therefore it is divided in two components:

- L1d: L1 cache for data
- L1i: L1 cache for instruction

L2 and L3 cache memory stores only data.

When the CPU requires specific data to carry out a particular operation, it first attempts to find it in L1. If it does not, it proceeds to look for it in L2 and then L3. If there is a match, the condition is referred to as a cache hit. If the CPU cannot find the data in any level of the cache memory, it attempts to access it from RAM. This condition is known as a cache miss. In this case, a new entry is created in the cache after each miss to store the data retrieved from RAM.

This behaviour is often implemented using the Least-Recently Used algorithm, which means that when a new item is added to the cache, the least recently used item is deleted, a condition known as cache eviction, to make space for the new entry.

Considering cache misses specifically, there are different reasons why this can occur in practice:

- A cold miss is inevitable and caused by the first access to a block of memory because it would not have been cached yet.
- A conflict miss occurs when even though cache is large enough, multiple data objects map to the same block.
- A capacity miss occurs when the set of active cache blocks is larger than the cache.

In this coursework task we will simplify the above model so that we shall consider all three levels of cache memory as a single unit.

Therefore, we do not differentiate between cache hits which locate data in L1, in L2 or in L3. Moreover, we consider cache memory to consist of S sets, with each set made up of E lines. Each line is made up of blocks with capacity of B bytes. This view of cache memory is depicted in Figure 1 below.

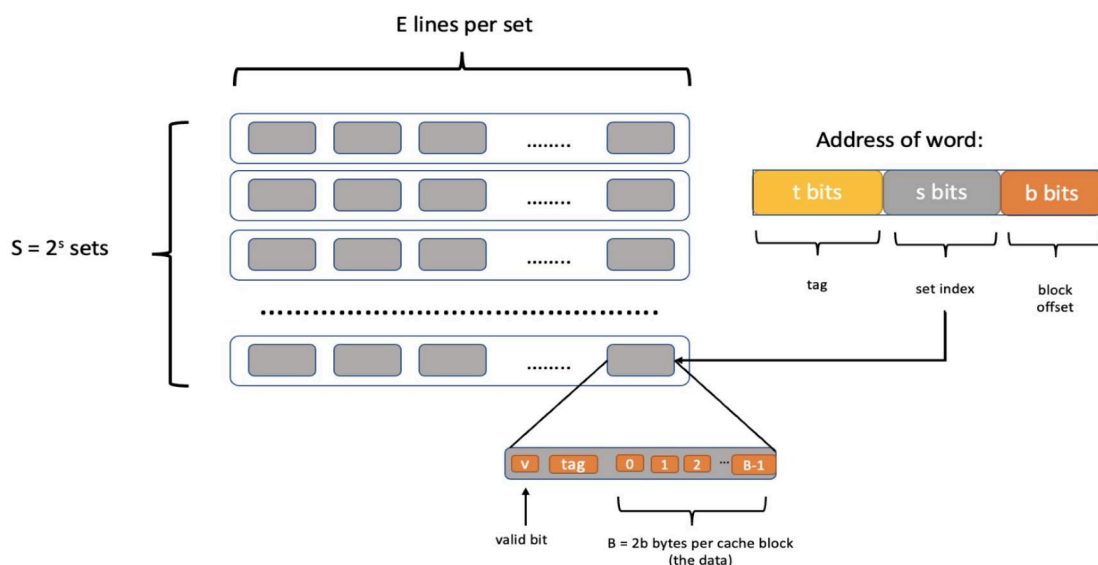


Figure 1. Visual depiction of simplified cache memory structure.

Moreover, we will consider memory addresses which have the following format:

memory address

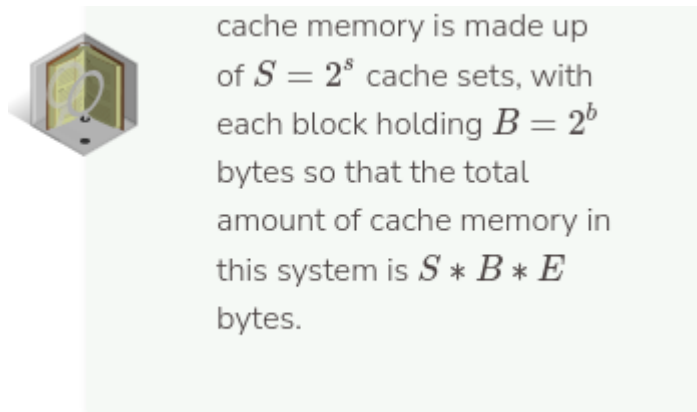


That is, memory addresses consist of three parts which are used to obtain the information required to look up and fetch a block of data from main memory into the cache:

- the *block offset* corresponds to the b least significant bits of the address;
- the *set index* corresponds to the bits following the block bits; and finally,

- the *tag bits* correspond to the remaining $AddressSize - b - s$ bits, i.e., the most significant bits in the memory address after set bits and block bits are taken out.

Following the above, we can observe that the length and format of memory addresses employed in a particular system that is the specific choice of b and s , imply that:



Furthermore, recall that data is copied in block sized units. You may also recall the concept of associativity from Lecture 9 of Topic 7 where we observe that setting $E=1$ results in a direct-mapped cache.

5. Detailed Task Description

In your implementation of `sim` the following requirements shall be fulfilled:

1. To receive full credit, the `sim` file must compile without any warnings.
2. The simulator must work correctly for arbitrary choices of s , E , and b . To achieve this, the program should dynamically allocate storage for the simulation data structures using the appropriate functions.
3. To receive full credit, the last step before your simulator terminates must result in printing the result of the simulation precisely matching the format below. This is critical for testing the correctness of your code and any other formatting of the output will result in an error and thus no credit given for correctness. For example,

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

4. For this problem, we only consider data cache performance. As such, the simulator shall ignore all instruction cache accesses, i.e., lines in the traces starting with `"I"`. To help you parse the trace, please note that in the trace files an `"I"` is always in the first column with no preceding space, and `"M"`, `"L"`, and `"S"` are always in the second column and the line includes a preceding space character.

5. For this problem, you should assume that memory accesses are aligned properly, such that any single memory access never crosses block boundaries. By making this assumption, you can ignore the request sizes in the traces, i.e., the `size` value in the trace format shown above.

6. Additional key points to consider

1. The cache simulator is **NOT** a cache! You are not required to implement the actual behaviour of cache memory but simply count hits, misses, and evictions. In particular, it is **NOT** required to store memory contents.
2. Block offsets are **NOT** used, that is the b least significant bits in the address do not matter.
3. Focus on using *meta-data* in the trace file to calculate hits, misses, and evictions.
4. Your cache simulator needs to work for different values of s , b , and E , which shall be input at run time.
5. Use the `getopt` crate (or similar) to parse command line options and retrieve the parameters above.
6. Use *Least-Recently-Used* as the replacement policy, i.e., evict the least recently used block from the cache to make room for the next block.
7. Each data load “`L`” or store “`S`” operation can cause at most one cache miss.
8. Each data modify operation “`M`” should be treated as a *load* followed by a *store* to the same address. Thus, an “`M`” operation can result in two cache hits, or a miss and a hit plus a possible eviction.

7. Reference trace files

To evaluate the correct operation of the cache simulator developed in Task 1, we will use a collection of reference trace files. These files were generated from a live `Linux` system using the command line utility `valgrind`.

You can generate your own trace files by typing the following in the command line of your Linux container on `Codio`.

```
valgrind --log-fd=1 --tool=lackey -v --trace-mem=yes ls -l
```

The above results in executing the command `ls -l` and capturing a trace of each of its memory accesses in the order they occur. By default, the utility prints the results on `stdout`.

To understand the output produced (which can scroll over several pages and include additional meta-data that we will not use), consider the following snippet from such a file:

```
I 0400d7d4,8
```

M 0421c7f0,4

L 04f6b868,8

S 7ff0005c8,8

In the above, each line represents one or two memory accesses following the format:

[space] operation address, size

The operation field denotes the type of memory access:

- “I” is an instruction load,
- “L” is a data load,
- “S” is a data store, and
- “M” is a data modify (that is, a data load followed by a data store).

There is never a space character before an “I”. There is always a space character before each “M”, “L”, and “S” (see below for further details on parsing the file).

The address field specifies a 64-bit memory address in hexadecimal.

The size field specifies the number of bytes accessed by the operation.

Note that the trace files contain several additional fields to the above, but we are only using memory address for this cache simulator.

8. Reference simulator

In addition to sample trace files, the binary executable `sim-ref` of a reference cache simulator is provided.

This can be used to check that your program produces correct results.

The reference simulator takes the following command-line arguments:

Usage:

```
./sim-ref [-hv] -s <s> -E <E> -b <b> -t <tracefile>
```

`-h`: Optional help flag that prints usage info

`-v`: Optional verbose flag that displays trace info `-s <s>`:

Number of set index bits ($S = 2^s$ is the number of sets)

`-E <E>`: Associativity (number of lines per set)

`-b `: Number of block bits ($B = 2^b$ is the block size)

`-t <tracefile>`: Name of the trace to replay

The command-line arguments are based on the notation above. For example:

```
> ./sim-ref -s 4 -E 1 -b 4 -t traces/yi.trace
```

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

The same example in verbose mode produces the following output:

```
> ./sim-ref -v -s 4 -E 1 -b 4 -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 eviction
```



```
L 210,1 miss eviction
```

```
M 12,1 miss eviction hit
```

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

9. How to submit

Work on your coursework using the repository allocated to you in the module github classroom.

To claim your repository, please follow this link: <https://classroom.github.com/a/NcKbwo41>

Make sure you use the repository provided in the classroom throughout development.

Note: Make sure that in addition to your source code you also include any other files or directories that are needed for your program to run.

To submit your work for marking, clone the version of the github repository that you wish to be considered by cloning it to Codio.

- Go to the coursework assignment in Codio.
- Open the terminal (using Tools-> Terminal) and type

```
>> git clone YOUR_GITHUBCLASSROOM_REPO_LINK
```

where `YOUR_GITHUBCLASSROOM_REPO_LINK` is the URL of your assigned GitHub project repository.

- Enter the credentials of your GitHub account. Remember that you need to use a personal access token as your password.

Note: You must provide the following to complete the assessment:

- the GitHub repo with the final version of your coursework (and history)
- The written report.

All entries contribute to your mark.

Assessment Criteria:

Please refer to Appendix C of the Programme Regulations for detailed Assessment Criteria.

Plagiarism:

This is cheating. Do not be tempted and certainly do not succumb to temptation. Plagiarised copies are invariably rooted out and severe penalties apply. All assignment submissions are electronically tested for plagiarism. More information may be accessed via:

<https://learn.london.ac.uk/mod/page/view.php?id=3214>