# Project 1 -- Cache Replacement Policies Simulator

**This is an *individual* project.** You may only collaborate with your classmates according to the CS Collaboration Policy. Plagiarism will be punished severely.

## Learning Objectives

- Create a simulator for a set-associative cache system.
- Become familiar with cache organization and cache lookup operations.
- Solidify your understanding of cache replacement policies.
- Compare cache performance across replacement policies, set sizes and cache lines per set.

## Overview

In this project you will be creating a cache simulator which can be used to observe how the different cache parameters effect the efficacy of a caching system.

It is ***highly recommended*** that you develop this project on a Linux system. If you choose to use a different type of system (such as macOS or Windows), the instructors and TAs will not be able to assist you with any platform-specific issues.

### Rubric

This project is worth **150** points, distributed as follows:

| # | Item | Points |
|---|---|---|
| 1 | LRU replacement policy | 90 |
| 2 | RAND replacement policy | 20 |
| 3 | LRU_PREFER_CLEAN replacement policy | 40 |
| | **TOTAL** | **150** |

**NOTE:** Since your submission will be partially auto-graded on Gradescope, you will be only be allowed a total of three submissions. Be sure to test your code before submitting!

## Cache Simulation

Your simulator must simulate a set-associative cache with three different replacement policies. The cache size, the number of cache lines, the associativity, and the replacement policy are configurable via command line arguments. Your program must read in a trace of the memory accesses from `stdin` and output the results to `stdout`.

You must implement a `RANDOM` replacement policy, a `LRU` replacement policy, and a `LRU_PREFER_CLEAN` replacement policy. See the Replacement Policies section below for details.

You may complete this project using any language, however *significant* starter code is provided in C. Your code **must** satisfy all of the requirements, regardless of what language you choose to implement the project in. (See the Starter Code Overview section below for details.)

### Cache Simulation Behavior

The following describes how the simulation must behave. *Note that some of this functionality is already implemented by the starter code.*

**Assumptions:**

- At the start of the simulation, no data is stored in the cache.
- The machine you are simulating is a 32-bit machine.

**Each line of `stdin` must be processed as follows:**

1. If there is a *cache hit*, then go to step 3.
2. If there is a *cache miss*, then you need to find where to store the new cache line, evicting from the cache as necessary. Then store the cache line containing the missed address in the cache. Specifically, the following operations must be performed:
   1. Calculate the set in the cache where the cache line needs to be stored.
   2. If the set is full (no more space to store a new cache line in a previously unoccupied spot), then a cache line must chosen for eviction according to the specified *replacement policy*.
   3. The cache line's tag needs to be updated with the new cache line.
3. If the operation was a write, the cache line should be marked as dirty.
4. The replacement policy state must be updated.

### Replacement Policies

You must implement three replacement policies:

- `LRU` (**L**east **R**ecently **U**sed, item 1.1 in the rubric) --- For this policy, the least recently used cache line should be evicted.
- `RANDOM` (item 1.2 in the rubric) --- For this policy, a random cache line should be evicted.
- `LRU_PREFER_CLEAN` (item 1.3 in the rubric) --- For this policy, the least recently used clean (not dirty) cache line should be evicted. If there are no clean cache lines, then the least recently used dirty cache line should be evicted.

## Starter Code Overview

The starter code provides a C project that can be compiled using `make`. The only dependency for compiling the code is GCC.

The starter code should compile as is, however it will not behave correctly. I recommend that you attempt to build the starter code before starting to make your own modifications so that you know that you have something that is working.

The starter code provides an easy way to create a properly formatted submission TAR.GZ file using `make submission`. This calls the `bin/makesubmission.sh` script. This script requires `sh` and `tar`.

*Note for Nix package manager users*: a `shell.nix` file is provided with the starter code. Running `nix-shell` will start a shell with the necessary dependencies installed. If you also use direnv, running `direnv allow` will add of the environment variables from the Nix shell to your current environment when you `cd` to this directory.

### Downloading the Source

Please download the project file on Canvas page.

### Building and Running

You can build the starter code by running `make` from this directory. This will create a `cachesim` executable that you can run. See the Inputs section for details on what inputs need to be passed in and what the parameters are.

You can also run the automated test script which includes some of the inputs that will be run by the grader script by running the following command:

```
$ make grade
```

This will compile your program and then run the grader script. The grader script requires Python 3 and scipy.

### Top-Level Organization

The following tree shows an overview of the important files and directories in the starter code repository.

```
/aca-project1                   project root
|-> bin/                        contains some utility shell scripts
|-> expected/                   contains expected outputs
|-> inputs/                     contains a set of sample input trace files
|-> Makefile                    a Makefile for compiling the project
|-> README.md                   this README file
'-> src/                        all of the source code for the project
```

### Source Organization

All of the places where you potentially need to add code are marked with a `TODO`.

There are extensive comments at the top of each file explaining what each one does. There are also comments throughout the code explaining in detail the most important parts of the codebase.

## Full Requirements

*The following requirements are automatically fulfilled by the starter code (assuming correct usage).* They are included so that if you choose to write your simulator without using the starter code, your submission will be able to be graded.

### Compilation and Runtime Environment

Your submission must compile and run on **Ubuntu 18.04** and the execution of the simulator must not utilize any network resources. The compilation process may utilize network resources only for downloading any compilers required to compile your program. Your program must not error on any well-formed inputs, and must exit with 0 as the exit code. If the input is malformed, the behavior of the program is undefined.

**If your submission fails to compile or run on Ubuntu 18.04 without using network resources during execution, *you will receive a score of 0 for this project.***

### Submission Format

**Failure to follow the submission format described in this section will result in a score of 0 for this project.**

You must submit a TAR file with *all* of your source code. The TAR file can optionally be XZ or GZ compressed. The filename of your submission must match the following regular expression (case is ignored):

```
(\w+)-project1.(tar(.gz|.xz)?)
-----              ---------
  ▲                      ▲
your MultiPass username  |
                         |
                     optional compression of TAR file
```

Your TAR file must contain a `Makefile` in the root of the archive. Running `make` should compile your code and create an executable file called `cachesim` in the same directory as the `Makefile`. This executable must be your cache simulator implementation.

### Inputs

Your program must accept four positional command line arguments:

- **Replacement policy**: this will be one of the following: `LRU`, `RAND`, or `LRU_PREFER_CLEAN` representing the replacement policy to use in the cache simulation.
- **Cache size**: this will be an integer representing the number of bytes that can be stored in the cache.
- **Cache lines**: this will be an integer representing the number of cache lines in the cache. (This can be used to calculate the size of each cache line.)
- **Associativity**: this will be an integer representing the number of cache lines per set.

Your program must accept a trace of memory access locations via `stdin` (one per line), formatted as two space-separated values that indicate whether the memory operation is a read (`R`) or write (`W`) and which memory location was accessed as a hexadecimal number, respectively. For example:

```
R 0xbfe74c2c
R 0xbfe748b8
```

```
W 0xbfe748b8
R 0xbfe74c30
```

You can also look in the `inputs` directory for more examples.

Example execution with LRU replacement policy, cache size of 32 KiB, 2048 cache lines, 4-way associativity, and passing the contents of the `./inputs/trace1` file via `stdin`.

```
$ ./cachesim LRU 32768 2048 4 < ./inputs/trace1
```

## Output

Your program must provide output on `stdout`. **Output to `stderr` or to a file will not be graded.**

Additionally, *only lines that start with `OUTPUT` will be graded.* This means that you can output as much as you want to stdout for debugging purposes as long as those lines don't begin with `OUTPUT`.

### Statistics Output

The `OUTPUT` lines that are required are the statistics output which should be printed after the simulation is complete. Specifically, the following statistics must be output in order:

- Accesses (int): the total number of memory accesses
- Hits (int): the total number of cache hits
- Misses (int): the total number of cache misses
- Dirty Evictions (int): the total number of evictions of dirty cache lines which would require a write-back.
- Hit Ratio (float): the ratio of cache hits to total accesses

The Hit Ratio should be formatted with 8 decimal places and a leading zero. Each of the statistics should be its own line, and should be written in all caps. For example:

```
OUTPUT ACCESSES 496611
OUTPUT HITS 494165
OUTPUT MISSES 2446
OUTPUT DIRTY EVICTIONS 245
OUTPUT HIT RATIO 0.99507462
```

## Other Information

If you have any questions regarding this project, please contact the instructor or the teaching assistants.