4190.308 Computer Architecture, Spring 2019
Cache Lab: Understanding Cache Memories
Assigned: Wed, May 22th, Due: Wed, Jun 5th, 23:59 AoE

1 Overview

This lab will help you to understand the operation of caches by implementing your own cache simulator.

2 Handout Instructions

We distribute the directory structure and source code of the Cache Lab through git server. You can find the Cache Lab skeleton codes and files from git server url:

```
https://git.csap.snu.ac.kr
```

Follow the steps to set your environment in your workspace.

1. Press `fork` from the repository `comparchTA/Cache Lab`. This will create a copy of repository for your own git account.
2. After you forked the project, check whether your project url includes your student id like below:

   ```
   https://git.csap.snu.ac.kr/201X-XXXXX/cache-lab
   ```

   If it doesn't, change your user name in your account setting.
3. Make your repository private by checking your option at `Settings > Permissions > Project visibility`. Press `Save changes` after you change your settings.
4. Go to your project main page, and press `clone`. Copy your HTTPS url and paste it to your VM terminal command line following `git clone` command.

   ```
   $> git clone <HTTPS URL>
   ```

3 Writing a Cache Simulator

In this lab you will write a cache simulator in `cache.c/h` that takes a `valgrind` memory trace as input, simulates the hit/miss behavior of a cache memory on this trace, and outputs the total number of hits, misses, and evictions.

The input to the cache simulator are memory traces of real programs obtained by running `valgrind`. More about these traces and the trace generation can be found below.

The lab contains a reference implementation in `cachesim-ref`. The reference implementation supports six different cache line replacement strategies (round robin, random, LRU (least-recently-used), MRU (most-recently-used), LFU (least-frequently-used), and MFU (most-frequently-used)) and two write allocate strategies (write-allocate and no write-allocte). You don't have to implement all strategies (see below).

3.1 Usage

Running the reference implementation with `--help` produces the following help screen.

```
$ ./cachesim-ref -h
cachesim: a cache simulator.

Usage: $ cachesim <options>
where <options> is
  -h / --help                   Show this help screen.
  -v / --verbose                Be verbose while running.
  -c / --capacity <number>      Set the cache capacity.
  -b / --blocksize <number>     Set the block size.
  -w / --ways <number>          Set the number of ways.
  -r / --replacement <number>   Set the replacement strategy.
                                  0  round robin (default)
                                  1  random
                                  2  LRU (least-recently used)
                                  3  MRU (most-recently used)
                                  4  LFU (least-frequently used)
                                  5  MFU (most-frequently used)
  -W / --write <number>        Set the write strategy.
                                  0  write-allocate (default)
                                  1  no write-allocate
```

The capacity, blocksize, and the number of ways are mandatory parameters. The default replacement strategy (round robin) and write allocation policy (write-allocate) can be modified using the respective command line options.

The cache simulator reads its input from `stdin`. Use the pipe operator to cat a trace into the simulator as follows

```
$ cat traces/test.2.trace | ./cachesim-ref -c 256 -b 16 -w 1 -r 2
Cache configuration:
  capacity:           256
```

```
  blocksize:           16
  ways:                 1
  sets:                16
  tag shift:            8
  replacement:     LRU (least-recently used)
  on write miss:   write-allocate

Processing input...

Cache simulation statistics:
  accesses:              9
  hit:                   4
  miss:                  5
  evictions:             3
  miss ratio:     55.56%
```

The same example in verbose mode:

```
$ cat traces/test.2.trace | ./cachesim-ref -c 256 -b 16 -w 1 -r 2 -v
Cache configuration:
  capacity:          256
  blocksize:          16
  ways:                1
  sets:               16
  tag shift:           8
  replacement:     LRU (least-recently used)
  on write miss:   write-allocate

Processing input...
R       10 [t:       0,s:  1]: miss alloc(free)
R       20 [t:       0,s:  2]: miss alloc(free)
W       20 [t:       0,s:  2]: hit
R       22 [t:       0,s:  2]: hit
W       18 [t:       0,s:  1]: hit
R      110 [t:       1,s:  1]: miss evict alloc(0)
R      210 [t:       2,s:  1]: miss evict alloc(0)
R       12 [t:       0,s:  1]: miss evict alloc(0)
W       12 [t:       0,s:  1]: hit


Cache simulation statistics:
  accesses:              9
  hit:                   4
  miss:                  5
  evictions:             3
  miss ratio:     55.56%
```

3.2 Implementing Your Cache Simulator

Your job for this lab is to implement a cache simulator that takes the same command line arguments and produces the identical output as the reference simulator.

You need to implement the following cache line replacement strategies:

- round-robin
- random (use the `rand()` function from `stdlib` to produce random numbers)
- LRU (Least-Recently Used)

The other strategies implemented by the reference simulator (MRU, LFU, MFU) are optional.

Your simulator should also support the following write allocation policies:

- write-allocate
- no write-allocate

You do not need to support verbose mode (`-v`/`--verbose` option), but we strongly encourage you to implement some form of feedback to debug your simulator.

The cache is implemented in `cache.c/h`. The data structures for the cache (`structs Cache`, `Set`, and `Line`) in `cache.h` have been prepared for you but you may need to add additional fields to manage the cache. The implementation of the functionality is in `cache.c`. You will find a number of functions with `TODO` markers followed by a short explanation that give you an idea what to do.

Replacement Policy

Replacement policy is which cache line is selected when cache miss occurs. If the cache lines are full, you need to choose the cache line which is evicted. There are many replacement policies but you can implement only three of them in this project.

Round-Robin

Round-robin chooses the replacement target sequentially. There is target pointer which points the replacement target. It moves sequentially by order of cache line when cache miss occurs. When the pointer reaches the end of the cache line, it goes to the first cache line.

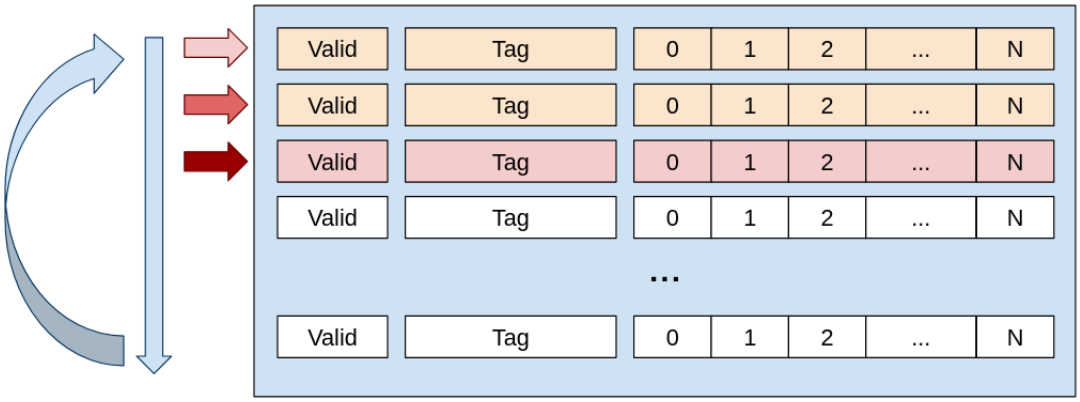

Figure 1: Round-robin replacement policy

Random

Random chooses the replacement target randomly. You can use `rand()` function in `stdlib.h`.

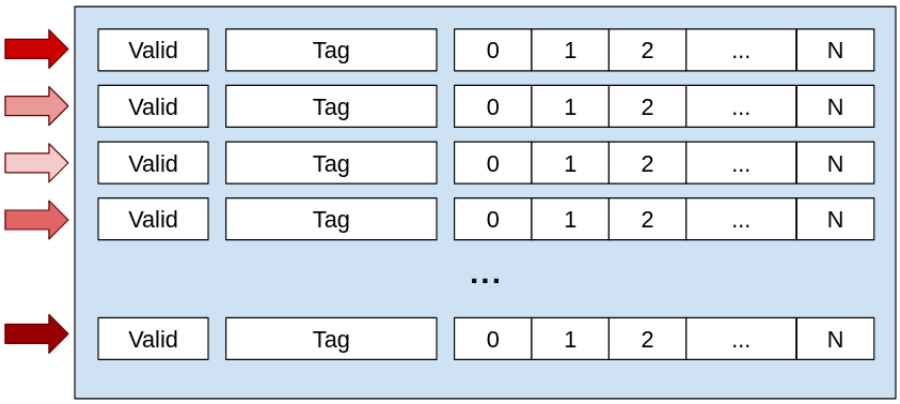

Figure 2: Random replacement policy

LRU: Least-Recently Used

LRU chooses the least recently used cache line as a replacement target. Each cache line has timestamp which records the last used time of cache line. When cache miss occurs, all timestamps are compared to select the replacement target.

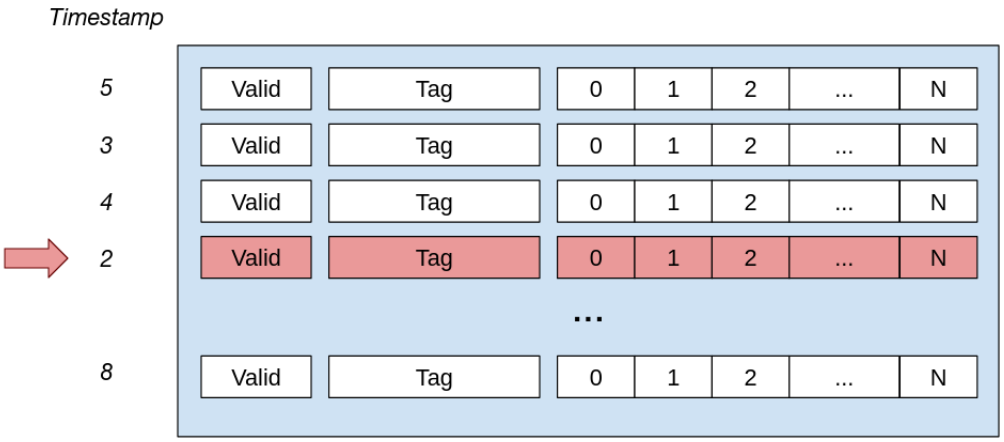

Figure 3: LRU replacement policy

Write Allocate Policy

Write allocate policy is that the way how to write the data to memory or cache. Cache can choose the write allocate policy depending on system structure or optimization techniques. In this project, you will implement only two of them, *write-allocate* and *no write-allocate*.

Write-allocate

Write-allocate policy fills the cache line when write cache miss occurs. Because it cannot write data to main memory directly, there can be burst data transfer from main memory to cache memory during handling write cache miss. When write cache miss occurs, cache takes data from main memory first, and writes the data to cache memory. The point when you implement this is there is cache line allocation when cache miss occurs.

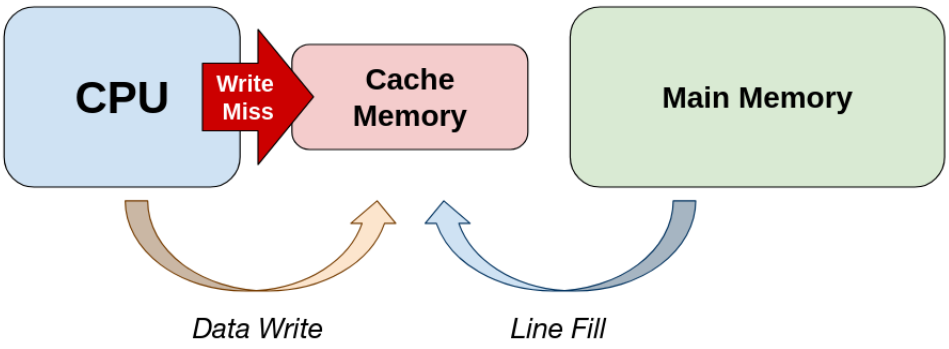

Figure 4: Write-allocate policy

No write-allocate

No write-allocate policy doesn't fill the cache line when write cache miss occurs. It can write data to main memory directly without cache write. Of course, the cache miss counts should be counted up, but there is no cache line allocation during handling cache miss.

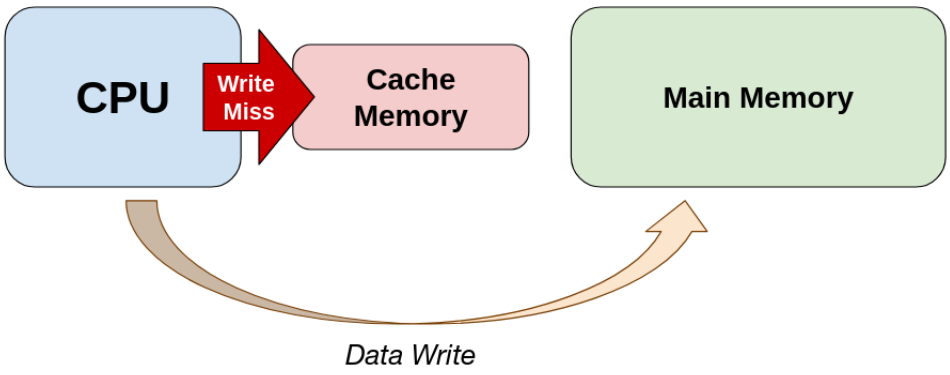

Figure 5: No write-allocate policy

Programming Rules

- Your simulator must work correctly for arbitrary input parameters. Perform parameter validation and print an error if a parameter is invalid (for example, if the capacity is not a power of two). This also means that you will need to allocate storage for your simulator's data structures using the `malloc()` function. Type "man malloc" for information about this function.
- To receive credit for this lab, you must correctly update the variables `s_acces`, `s_hit`, `s_miss`, and `s_evict`. The main simulator file `cachesim.c` uses these variables to output the cache statistics.
- For this this lab, you should assume that memory accesses are aligned properly, such that a single memory access never crosses block boundaries. By making this assumption, you can ignore the request sizes in the `valgrind` traces.
- You should be able to build your project using the command `make` only. Any other additional commands or options are not allowed.

3.3 Reference Trace Files

The `traces` subdirectory of this lab contains a collection of *reference trace files* that we will use to evaluate the correctness of your cache simulator. The trace files were generated by a Linux program called `valgrind` that records instruction and data read/write accesses of arbitrary programs.

The traces `test.?.trace` contain shorter traces that are useful when implementing and debugging your code. The four real-world traces `ls.1.trace`, `ls.2.trace`, `ls.3.trace.bz2`, and `cat.1.trace.bz2`

are the result of running `ls` and `cat` on Linux machines. Two of the traces, `ls.3.trace` and `cat.1.trace` are so long that we provide them in compressed form.

Use the pipe operator to cat a trace into the simulator. For uncompressed traces, the command is (replace `test.4.trace` with any uncompressed trace file of your choice)

```
$ cat traces/test.4.trace | ./cachesim [cache options]
```

The compressed traces need to be decompressed before feeding them into the simulator. This can be acheived by running the decompressor `bunzip2` and piping the output directly into the simulator as follows

```
$ bunzip2 -c traces/cat.1.trace.bz2 | ./cachesim [cache options]
```

4 Experiment

You can do some experiments using cache simulator you implemented. Organize following experiments and explain the results in your report.

1. When the cache size and associativity is fixed, find the relationship between miss rate and block size.
2. Find the relationship between miss rate and associativity.

You should attach some diagrams or graphs to explain the results in your report.

5 Report

Your report should follow the rules below:

- Use the report template under `report` directory.
- Before start to write a report, erase all italic font descriptions in the report template.
- Your report should include the brief statements for the project and the explanation about the program you implemented.
- Your report should not be longer than 8 pages (excluding the cover page).
- Avoid copy-pasting screenshot of your code. We already have your code.
- You can attach some diagrams to depict your implementation. If necessary.
- The file name format should match `201X-XXXXX_cachelab_report.pdf`. Replace the number at front to your student id and place it under `report` directory.

6 Evaluation

We will evaluate your simulator as follows

- round-robin replacement policy: 20 points
- random replacement policy: 20 points
- LRU replacement policy: 20 points
- write-allocate/no write-allocate policy: 10 points
- Report: 30 points

If you violate the formats, for example remote repository url format, file naming, and project directory structure, there will be some deductions on your points. The minimum point is 0 and the maximum point is 100. We test your implementation with the trace files both provided and not provided in `traces` and different cache parameters. Use the reference implementation to obtain the correct values. We will give points for (partial) implementations that produce wrong results if you were on the right track, so do not remove code even if it may not work 100% correctly.

7 Handin Instructions

You should submit the following files to the `master` branch of your own git remote repository.

- `cache.c/h`
- `cachesim.c` - You don't need to modify this file.
- `Makefile`
- `201X-XXXXX_cachelab_report.pdf` - Under the `report` directory

You can submit your files by using following `git` commands.

1. Stage your updated files.

   ```
   $> git add <path/file_name>
   ```

2. Commit your changes which you staged with commit message.

   ```
   $> git commit -m "<commit message>"
   ```

3. Push your code to remote git server.

   ```
   $> git push origin master
   ```

I recommend you to commit and push your files regularly to give yourself a chance to recover from your potential mistakes.

You should follow the project structure and file naming rules.

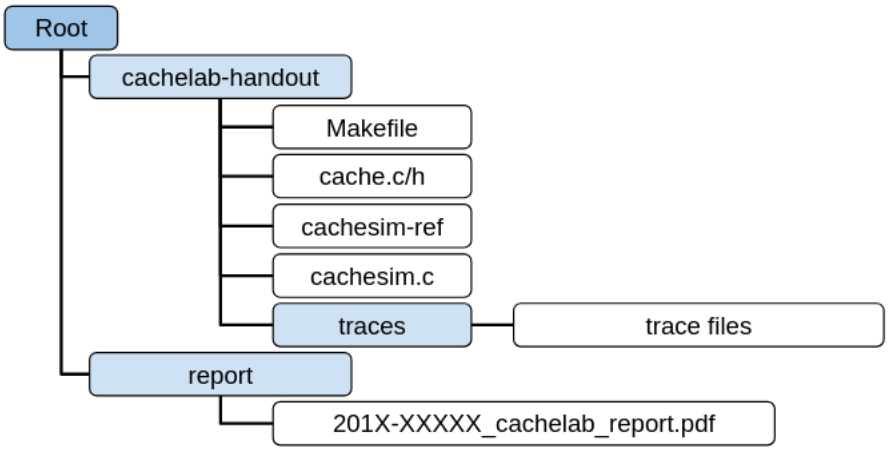

Figure 6: Project structure and Naming

If you violate the submission formats, there will be some deductions on your points.

Good Luck!