Program Assignment 4 Cache Simulation

Jun 28, 2020 (2020-06-28T00:00:00+08:00)

By Jing-Jia Liou

Contents

1 Problems

- In this homework, there are three main parts.
  1. In Part I (60%), we will implement the LFU policy in a cache simulator.
  2. In Part II (10%), please answer the questions in the Google form.
  3. In Part III (30%), we will implement an optimized L1 cache.

1.1 Penalty of plagiarism

- If your final version (last submission before due date) is a plagiarized code, no credit will be given.

1.2 Part I LFU replacement policy (60%)

Due: 23:59 on June 28, 2020

Proportion: 60%

- We've prepared a Cache Simulator programmed in C++. Please download codes by the following command:

  ```
  $ git clone https://gitlab.com/amy26656/pa4_template.git pa4
  ```

- Please implement **LFU replacement policy** in our cache simulator. LFU stands for "least frequent used" blocks. LFU blocks will be replaced if conflict happens. LFU is calculated based the number of accesses (including both read and write) over a period of time (please use 100 memory accesses as a period in this program assignment). We only record data for cache blocks in the cache. If one block is replaced, there will be no record (set frequency 0).

  1. In `pa4/part1/src/main_cache.cpp`, please complete two functions: `_HitHandle()` and `_GetIndexByLFU()`

     - **`_HitHandle(const addr_t &addr)`**:

       This function is always provoked when the cache is accessed with a given address `addr`.

     - **`_GetIndexByLFU(const addr_t &addr)`**:

       This function is provoked in the process of writing data to cache. Given an address, determine which cache block is going to be over-written by new data from address `addr`.

```
void MainCache::_HitHandle(const addr_t &addr) {
     // # TODO

     // switch(property.associativity) {
     // case full_associative:

     // case set_associative:

     // default:
     //     break;
     // }
}

ulint MainCache::_GetIndexByLFU(const addr_t &addr) {
     ulint res(0);
     // # TODO

     // switch(property.associativity) {
     // case full_associative:

     // case set_associative:

     // default:
     //     break;
     // }
     return res;
}
```

2. You can declare any additional variable or method in `MainCache` class

- Compiling the simulator

  The program directory structure looks like this

```
pa4/
|   part1/
|   |----CMakeLists.txt
|   |----include/
|   |----src/
|   |     |----main_cache.cpp
|   |     |----main_cache.hpp
|   |     |----main.cpp
        ...
```

  Here we use **CMake** instead of traditional Makefile to build your project, you can add any C++ source files in `src/` directory and cmake will detect them automatically.

  Use the following commands to build your simulator (Assume you are already in the project root folder):

```
$ mkdir build && cd build
$ cmake ..
$ make
```

Then the binary executable file will be generated and named as `cache_sim`.

- We also prepared a few **trace files** under `pa4/part1/trace/` directory.

  Trace files are load/store records dumped from benchmark programs. We will use them to represent program memory accesses. The format of a trace file is as follows:

  ```
  l 0x000000001fffff50
  ```

  `l` or `s` means **load** or **store**.

  `0x000000001fffff50` is the 64-bit address in hexadecimal, note that the block size should be at least 8 Bytes (due to RISC-V double word load/store).

- There are three(3) sample config files under `pa4/part1/config/` directory. Config file is used to specify a cache architecture:

  ```
  {
      "multi-level": false,
      "content": [
          {
              "cache-size": 256,
              "block-size": 8,
              "associativity": "direct-mapped",
              "replacement-policy": "random"
          }
      ]
  }
  ```

  ```
  {
      "multi-level": false,
      "content": [
          {
              "cache-size": 64,
              "block-size": 32,
              "associativity": "set-associative",
              "number-of-way": 4,
              "replacement-policy": "random"
          }
      ]
  }
  ```

```
{
    "multi-level": false,
    "content": [
        {
            "cache-size": 8,
            "block-size": 64,
            "associativity": "full-associative",
            "replacement-policy": "random"
        }
    ]
}
```

- Run simulation and check your simulation results.

  Assume you are running with `gcc.trace` and `cache1.json`. Use the following command to run simulation.

  ```
  $ ./cache_sim -t ../trace/gcc.trace -c ../config/cache1.json
  ```

  You can use the following command to run simulation and dump the results into text file `gcc.txt`.

  ```
  $ ./cache_sim -t ../trace/gcc.trace -c ../config/cache1.json > gcc.txt
  ```

- Your simulator output will be as follows:

  ```
  ========================================
  Test file: ../TestData/gcc.trace
  ----------------------------------------
  # L1 Cache
  Cache size: 256KB
  Cache block size: 8B
  Associativity: direct-mapped
  Replacement policy: Random
  ----------------------------------------
  Number of cache access: 515683
  Number of cache load: 318197
  Number of cache store: 197486
  Number of total cache hit: 494203
  Cache hit rate: 0.958347
  Average Memory Access Time: 5.165 cycles
  ========================================
  ```

- To verify your LFU implementation, please use following tables to check your outputs.

Test file	**gcc**	**gzip**	**mcf**	**swim**	**twolf**	**maze**
Total inst.	515683	481044	727230	303193	482824	13388
Load	318197	320441	5972	220668	351403	12074

Test file	gcc	gzip	mcf	swim	twolf	maze
Store	197486	160603	721258	82525	131421	1314

- 256KB, 8 Bytes/line, direct-mapped, None

Test file	gcc	gzip	mcf	swim	twolf	maze
Hit Rate	0.958347	0.667072	0.010379	0.934319	0.988443	0.935614

- 64KB, 32 Bytes/line, 4-way set-associative, LFU

Test file	gcc	gzip	mcf	swim	twolf	maze
Hit Rate	0.987165	0.668263	0.752448	0.976827	0.996317	0.979609

- 8KB, 64 Bytes/line, fully-associative, LFU

Test file	gcc	gzip	mcf	swim	twolf	maze
Hit Rate	0.962605	0.668375	0.875951	0.974891	0.976134	0.976696

- Note that your simulation results should match above tables for the same cache architecture and trace files.

- In our Autolab system, we prepare 9 different trace files as testcases for grading your implementation. You already have 6 of 9 trace files as public test cases, the other 3 test cases are non-public.

1.3 Part II: Observe and Analyze (10%)

Due: 23:59 on June 28, 2020

Proportion: 10%

- Please answer the questions listed in the form.
  1. EE3450-pa4-extra.

1.4 Part III Cache Optimization (30%)

Due: 23:59 on June 28, 2020

Proportion: 30%

- In this part, we will design an optimized cache architecture with a cache size of 64KB (cache size is defined by the available space to store user's data, not including bits necessary for cache management) . The design parameters are cache block size, number of ways. Our objective is to have an average hit rate as high as possible under the 64KB constraint over several trace files.

- We will use the trace files under `pa4/part3/trace/` directory to evaluate average hit rate.
- If two cache architecture has similar average hit rates (difference < 0.1%), we will use hardware cost to find the best cache design. The hardware cost is basically the total memory bits used in the cache. Note that we will actually use a SRAM simulator (CACTI) to estimate the total memory area (not just bit numbers).
- Cache designs from all students will be ranked in Poisson distribution percentile for grading.
- Please use LFU as your replacement policy, remember to **copy the LFU you write in part1 to part3**, in `pa4/part3/src/` directory.
- To design your cache architecture, please modify **cache_arc.cfg** under `pa4/part3/` directory.
- To evaluate the average hit rate and hardware cost, we prepare an evaluation tool. The tool will be activated by the following command.

  ```
  $ make
  ```

  The tool will ask for your cache configuration and generate hit rate and hardware cost for your cache design. The terminal output will looks like the following. In this case, the hardware cost of the design is 96.98347245600002 and the average hit rate is 0.998051. Note that the lower hardware cost, your cache needs less hardware (better).

  ```
  Block-size: 64
  Associativity(2^n with n >= 0): 4
  Hardware cost: 96.98347245600002
  Average Hit-Rate: 0.998051
  $
  ```

  Also note that this tool will generate a file named **score.txt** in current folder. You will need to submit this score file, too. Remember to use `make clean`, if you want to re-evaluate the hit rate and hardware cost.

2 Submission

2.1 Part I

- Files to submit

1. Project directory (pa4): Including all C++ source files and CMakeLists.txt, but **DO NOT** submit the build directory and trace files.
    - The detailed file structure is listed below.

```
pa4/
|---CMakeLists.txt
|---include/
|---src/
|   |--- main_cache.cpp
|   |--- main_cache.hpp
|   |--- main.cpp
|   |--- any .cpp files you created
```

1. Compress your project directory as submission.tar

    ```
    $ tar cvf submission.tar pa4
    ```

2. Submit the tar file on Autolab website (https://autolab.larc-nthu.net/courses/EE3450/assessments/pa4).

2.2 Part II

- Just answer the form, no other file needs to be submitted.

2.3 Part III

- Files to submit

1. Compress the files "score.txt" and "cache_arc.cfg" as "pa4_part3.tar"

    ```
    $ tar cvf pa4_part3.tar score.txt cache_arc.cfg
    ```

2. Submit the tar file on Autolab website part3 (https://autolab.larc-nthu.net/courses/ee3450_2020_spring/assessments/pa4part3).

3. You can check your rank in Autolab scoreboard (https://autolab.larc-nthu.net/courses/ee3450_2020_spring/assessments/pa4part3/scoreboard).

3 Grading

- No credit for dead or crashed codes.
- No credit for codes with wrong output formats.
- The hit rate of each simulation should be identical to the tables listed. If some value mismatches the table, only partial credits will be given.

- For Part I, if some value in non-public testcases mismatches the correct answer, only partial credits will be given.
