PA #3: Simulating Data Cache Behavior

Due 11:59 pm, June 17th

1. Overview

This project is intended to help you understand the principle of caching by implementing a data cache. The main part of this project is to simulate a data cache for evaluating memory access traces. The cache should be configurable to adjust capacity, associativity, and block size with command-line options. You must support an extra option to print out the content of the cache.

2. Simulation Details

In this project, you will implement a simple cache simulator running for memory trace files. Each input file consists of a sequence of read (load) or write (store) operations with memory addresses. For each step, the cache simulator reads a line of the input file and simulates the internal operation of the cache. The write policy of the cache must be `write-allocate` and `write-back`. The replacement policy must be the `perfect LRU`.

2.1 Input Traces

The input trace we provided comes from selected SPEC-CPU benchmarks, which are widely used in academia and industry to evaluate the processor architecture. Each trace (benchmark) has its own behavior accessing memory so that we can observe different locality characteristics for the traces. Here are the first 10 lines of the `sample_input/simple` file.

```
$ head sample_input/simple
R 0x10001000
R 0x10001020
R 0x10001040
R 0x10001060
R 0x10001000
R 0x10001040
R 0x10001080
R 0x100010a0
W 0x10001004
W 0x10001024
```

The first column presents the access type, either read (`R`) or write (`W`). The memory address is in the second column. Note that our trace does not include the actual content (data) corresponding to the memory address because the goal of this project is to simulate the cache behavior instead of caching the contents.

2.2 Setting Up the Cache (Task #1)

To simulate the data cache, we need to define the capacity, associativity, and block size. These three parameters must be configurable through a command-line interface. The skeleton code already implemented the routine handling of the options. So, you are supposed to complete the `build_cache()` function in the **`cache.c`** file. In **`cache.h`**, we define the cache data structure, called `struct l1cache`, which contains sets (`struct cache_set`), ways (`n_ways`), and configurations (`n_set_bits` and `n_data_bits`). The first task is to understand the structure of `l1cache` and then complete the `build_cache()` function. Again, please note that our cache does not hold the actual data for simulation purposes. Instead, we only keep the tag value.

2.3 Evaluating Traces with the Cache (Task #2)

Once you complete task #1, it is time to work on evaluating the cache for the trace files. In this project, we provide the skeleton code for reading the trace from the file and extracting the access type and memory address. Your task is to implement the `access_cache()` function in the **`cache.c`** file. In the function, you need to write code looking up the cache for a given memory address and then figure out whether the requested memory can be found in the cache or not. The tag matching procedure should be implemented. If you can find the requested cache block (as known as cache hit), you need to update the age field (which is used for LRU replacement) of the cache. Also, you may update the dirty bit depending on the access type.

If the requested memory block is not in your cache, you need to allocate a cache block by either finding an empty space or evicting a cache block that is least recently used in the given cache set. After that, you need to update the tag information and other fields (valid, age, and dirty). Since we assume that our cache is **`write-allocate`**, you need to properly set the relevant field according to the access type.

Finally, our simulator counts the total number of hits, misses, and write-backs for a given trace file. So, we need to update the values through call-by-reference in `access_cache()`.

3. Getting the Skeleton Code

You can download the skeleton code from the GitHub repository to the server or local machines. Then you are ready to start the project.

→ Go to the following page: https://github.com/Sys-KU/kece343-comm307-project3

Be sure to read the **`README.md`** file for some useful information. It includes an explanation of each file and which files you are allowed to modify for this project. You are only allowed to modify the `cache.c` file because the grading script works on the provided `sample_input` and `sample_output` files described in the following section.

4. Building and Running the Cache Simulator

As Project 1, we provide the Makefile to build and test your code. If you are not familiar with how Makefile works, we highly recommend you visit this website.

4.1 Building your code

Step1: Let's move to the directory we cloned

```
$ git clone https://github.com/Sys-KU/kece343-comm307-project3.git
$ ls
kece343-comm307-project3

# TIPS: Try to use the Tab key when navigating directories or files
$ cd kece343-comm307-project3
```

Step 2: Trying to build the skeleton code

```
$ ls
Makefile README.md cache.c cache.h cache_simulator.c handout
sample_input sample_output

$ make # the warning messages come from the skeleton code
gcc -g -Wall -O0   -c -o cache_simulator.o cache_simulator.c
gcc -g -Wall -O0   -c -o cache.o cache.c
cache.c:8:12: warning: 'index_bits' defined but not used
[-Wunused-function]
    8 | static int index_bits(int n){
      |            ^~~~~~~~~~
gcc -o cache_simulator cache_simulator.o cache.o
```

Step 3: You can find the executable binary (`cache_simulator`)

```
$ ls
Makefile    README.md    cache.c    cache.h    cache.o    cache_simulator
cache_simulator.c cache_simulator.o handout sample_input sample_output
```

4.2 Testing the cache we built

Step 1: Our simulator requires a memory trace file

```
$ ./cache_simulator
Usage: ./cache_simulator [-c cap:assoc:block_size] [-x]
input_trace
```

The skeleton code handles the three parameters specified with '`-c`' option. You can change the parameters and see the cache statistics such as the number of hits and misses. For correctness, we provide the option of printing out the cached address (tag) information with '`-x`'.

- `-c:` cache configuration (capacity:associativity:block_size)
- `-x` : dump the cached address (tag) at the end of simulation

Step 2: Running the simulator with the provided sample inputs

```
$ ls sample_input/
gcc  libquantum  milc  simple


# Run the simulator with options
# Note that the below result comes from reference code, not the skeleton code.
$ ./cache_simulator -c 1024:8:8 -x sample_input/simple
Cache Configuration:
-------------------------------------
Capacity: 1024B
Associativity: 8way
Block Size: 8B

Cache Stat:
-------------------------------------
Total reads: 12
Total writes: 7
Write-backs: 0
Read hits: 6
Write hits: 7
Read misses: 6
Write misses: 0

Cache Tags:
-------------------------------------
          WAY[0]      WAY[1]      WAY[2]      WAY[3]      WAY[4]
WAY[5]      WAY[6]      WAY[7]
SET[0]:   0x10001000  0x10001080  0x00000000  0x00000000  0x00000000
0x00000000  0x00000000  0x00000000
SET[1]:   0x00000000  0x00000000  0x00000000  0x00000000  0x00000000
0x00000000  0x00000000  0x00000000
SET[2]:   0x00000000  0x00000000  0x00000000  0x00000000  0x00000000
0x00000000  0x00000000  0x00000000
SET[3]:   0x00000000  0x00000000  0x00000000  0x00000000  0x00000000
0x00000000  0x00000000  0x00000000
SET[4]:   0x10001020  0x100010a0  0x00000000  0x00000000  0x00000000
0x00000000  0x00000000  0x00000000
SET[5]:   0x00000000  0x00000000  0x00000000  0x00000000  0x00000000
0x00000000  0x00000000  0x00000000
SET[6]:   0x00000000  0x00000000  0x00000000  0x00000000  0x00000000
0x00000000  0x00000000  0x00000000
SET[7]:   0x00000000  0x00000000  0x00000000  0x00000000  0x00000000
0x00000000  0x00000000  0x00000000
```

```
SET[8]:   0x10001040  0x00000000  0x00000000  0x00000000  0x00000000
0x00000000  0x00000000  0x00000000
SET[9]:   0x00000000  0x00000000  0x00000000  0x00000000  0x00000000
0x00000000  0x00000000  0x00000000
SET[10]:   0x00000000  0x00000000  0x00000000  0x00000000  0x00000000
0x00000000  0x00000000  0x00000000
SET[11]:   0x00000000  0x00000000  0x00000000  0x00000000  0x00000000
0x00000000  0x00000000  0x00000000
SET[12]:   0x10001060  0x00000000  0x00000000  0x00000000  0x00000000
0x00000000  0x00000000  0x00000000
SET[13]:   0x00000000  0x00000000  0x00000000  0x00000000  0x00000000
0x00000000  0x00000000  0x00000000
SET[14]:   0x00000000  0x00000000  0x00000000  0x00000000  0x00000000
0x00000000  0x00000000  0x00000000
SET[15]:   0x00000000  0x00000000  0x00000000  0x00000000  0x00000000
0x00000000  0x00000000  0x00000000
```

Once you run the `cache_simulator` binary with a trace file, it will print the cache configuration first, and then you can see the statistics such as how many reads and writes are performed and the number of cache hits and misses. If you have the option of printing out the cache tags, you are able to see the tag information that is cached. Note that when running from skeleton code, you may probably face the segmentation error. This is because the skeleton code does not have a proper `build_cache()` function.

4.3 Comparing the generated output with the reference output

```
$ ./cache_simulator -c 1024:8:8 -x sample_input/simple | diff
-Naur sample_output/simple -
--- sample_output/simple   2022-11-29 21:26:57.398061766 +0900
+++ - 2022-11-29 21:36:06.979967252 +0900
@@ -1,36 +0,0 @@
-Cache Configuration:
---------------------------------------
-Capacity: 1024B
-Associativity: 8way
-Block Size: 8B
-
-Cache Stat:
---------------------------------------
-Total reads: 12
-Total writes: 7
-Write-backs: 0
-Read hits: 6
-Write hits: 7
-Read misses: 6
-Write misses: 0
-
-Cache Content:
---------------------------------------
-          WAY[0]      WAY[1]      WAY[2]      WAY[3]      WAY[4]
```

```
WAY[5]       WAY[6]       WAY[7]
-SET[0]:   0x10001000  0x10001080  0x00000000  0x00000000
0x00000000  0x00000000  0x00000000  0x00000000
-SET[1]:   0x00000000  0x00000000  0x00000000  0x00000000
0x00000000  0x00000000  0x00000000  0x00000000
-SET[2]:   0x00000000  0x00000000  0x00000000  0x00000000
0x00000000  0x00000000  0x00000000  0x00000000
-SET[3]:   0x00000000  0x00000000  0x00000000  0x00000000
0x00000000  0x00000000  0x00000000  0x00000000
-SET[4]:   0x10001020  0x100010a0  0x00000000  0x00000000
0x00000000  0x00000000  0x00000000  0x00000000
-SET[5]:   0x00000000  0x00000000  0x00000000  0x00000000
0x00000000  0x00000000  0x00000000  0x00000000
-SET[6]:   0x00000000  0x00000000  0x00000000  0x00000000
0x00000000  0x00000000  0x00000000  0x00000000
-SET[7]:   0x00000000  0x00000000  0x00000000  0x00000000
0x00000000  0x00000000  0x00000000  0x00000000
-SET[8]:   0x10001040  0x00000000  0x00000000  0x00000000
0x00000000  0x00000000  0x00000000  0x00000000
-SET[9]:   0x00000000  0x00000000  0x00000000  0x00000000
0x00000000  0x00000000  0x00000000  0x00000000
-SET[10]:   0x00000000  0x00000000  0x00000000  0x00000000
0x00000000  0x00000000  0x00000000  0x00000000
-SET[11]:   0x00000000  0x00000000  0x00000000  0x00000000
0x00000000  0x00000000  0x00000000  0x00000000
-SET[12]:   0x10001060  0x00000000  0x00000000  0x00000000
0x00000000  0x00000000  0x00000000  0x00000000
-SET[13]:   0x00000000  0x00000000  0x00000000  0x00000000
0x00000000  0x00000000  0x00000000  0x00000000
-SET[14]:   0x00000000  0x00000000  0x00000000  0x00000000
0x00000000  0x00000000  0x00000000  0x00000000
-SET[15]:   0x00000000  0x00000000  0x00000000  0x00000000
0x00000000  0x00000000  0x00000000  0x00000000
```

To figure out which cache sets and ways are different from the reference output, we use the `diff` utility, or you can use `Makefile` to test individual samples. Since the skeleton code does not generate the correct output, it prints all the lines that are different. If you implement the code as intended, the `diff` utility will print nothing like the below. It means that your output file is the same as the reference output in the `sample_output` directory.

5. Grading Policy

Grades will be given based on the 4 examples provided for this project provided in the `sample_input` directory. Your simulator should print the same output as the files in the `sample_output` directory.

We will be automating the grading procedure by seeing any differences between the files in the

`sample_output` directory and the result of your simulator executions. Please make sure that your outputs are identical to the files in the `sample_output` directory.

You are encouraged to use the `diff` command to compare your outputs to the provided outputs. If there are any differences (including whitespaces), the diff program will print the different lines. If there are no differences, nothing will be printed as mentioned before. Furthermore, we have provided a simple checking mechanism in the `Makefile`. Executing the following command will automate the checking procedure.

```
$ make test
```

There are 4 code segments to be graded, and you will be granted 20 points for each correct binary code and being **`Correct`** means that every digit and location is the same as the given output of the example. If a digit is not the same, you will receive **0 points** for the example.

6. Submission

Before submitting your code to `Gradescope`, it is highly recommended to complete the work on your local Linux system environment. In this project, you just need to upload the `cache.c` file to gradescope (**Do not modify file name**). After that, you should check that your code works in your local environment. Of course, you can test your code on `Gradescope` as many times as you want. Please make sure that you can see the same result on the submission site as well.

In this assignment, you are supposed to submit a 1-page document that should include the following:

1) Explain the structure of the provided skeleton and how it works as per your understanding
2) Describe how you implemented the `build_cache()` and `access_cache()` codes
3) What you learned from this assignment
4) (Optional) Suggest anything to improve this assignment if you have

- You do not need to include the cover page. Please just put your name and student ID
- Do not include any code or screenshots in the document
- Your document should be the PDF format. If you turn in doc or hwp format, you will get 0 points for this document.

7. Updates/Announcements

If there are any updates to the project, including additional tools/inputs/outputs, or changes, we will post a notice on the LMS and send you an e-mail using the LMS system. **Frequently check your e-mail account or the LMS notice board for updates.**

8. Misc

We will accept your late submissions, but your score will lose up to 50%. Please do not give up the project.

Be aware of plagiarism! Although it is encouraged to discuss with others and refer to extra materials, **copying other students or opening code publicly is strictly banned**. The TAs will compare your source code with other team's code. If you are caught, you will receive a penalty for plagiarism.

Last semester, we found a couple of plagiarism cases through an automated tool. Please do not try to cheat TAs. If you have any requests or questions regarding administrative issues (such as late submission due to an unfortunate accident, `Gradescope` is not working), please send an e-mail to the TA (whchoi@csl.korea.ac.kr).