ECE463/563 – Microprocessor Architecture
Project #3
Due date: April 29, 2021

Objective

The goal of this project is to design and implement a cache simulator (level-1 cache only).

Code organization

The `project3_code.tar.gz` archive contains the C++ code templates for the simulator, as well as test cases to validate its operation.

In order to extract the code in a Linux environment, you can invoke the following command:

```
tar -xzvf project3_code.tar.gz
```

This will create a `project3_code` folder containing the following:

- `cache.h/cache.cc`: code templates for the cache simulator. These are the only files that you need to modify to implement the simulator.
- `testcases`: test cases to validate the operation of the L1 cache simulator. This folder contains six test cases (`testcase0-5`). For each of them, you will find two files: `testcaseN.cc` and `testcaseN.out`. The former contains the test case implementation, and the latter the expected output of the test case. You should not modify any of the test case files.
- `Makefile`: `Makefile` to be used to compile the code. The use of this `Makefile` will cause an object file (`.o`) to be created for each C++ file that is part of the project. If the compilation succeeds, the binaries corresponding to the test cases will be generated in the `bin` folder. You don't need to modify the `Makefile`.
- `traces`: memory access traces used by the test cases. `simple.t` is a short synthetic trace to help you debug your code (and is used in `testcase0`). `GCC.t`, `MCF.t` and `LBM.t` are significantly longer memory access traces from real applications from the SPEC2006 benchmark suite (https://www.spec.org/cpu2006/), and they are used in `testcase1-5`.
- `bin`: once you compile the code, the test case binaries will be saved into this folder.

Important

The `cache.h` header file is commented and contains details on the functions/methods that you need to implement. Be sure to read the comments in this header file before you start coding.

Assumptions & Requirements

1. The tool should allow simulating an L1 cache with configurable capacity, cache line size, associativity, write hit and write miss policies, hit time and miss penalty. In addition, the width of the memory addresses should also be configurable. All these parameters can be set through the `cache` constructor:

```
cache(unsigned cache_size,                      // cache size (in bytes)
              unsigned cache_associativity,     // cache associativity
              unsigned cache_line_size,         // cache block size (in bytes)
              write_policy_t write_hit_policy,  // write-back or write-through
              write_policy_t write_miss_policy, // write-allocate or no-write-allocate
              unsigned cache_hit_time,          // cache hit time (in clock cycles)
              unsigned cache_miss_penalty,      // cache miss penalty (in clock cycles)
              unsigned address_width            // number of bits in memory address
        );
```

2. The simulator does not need to support fully associative caches.
3. The simulated cache should use the **LRU replacement policy**.
4. The cache simulator should simulate cache read and write accesses. The trace files in the `traces` folder contain the sequences of memory accesses that must be simulated. Each line of the trace files has the following syntax:
   `<r|w> <memory address>`
   where `r` and `w` represent read and write operations, respectively.
   For convenience, the parser is already included in the `run` function. The only modification to this function that is required is the insertion of invocations of the `read` and `write` functions.
5. In addition to the `read` and `write` functions, you must implement the following methods:
   - `print_configuration` outputs the configuration of the cache simulator (cache size, associativity, cache line size, write hit and miss policies, cache hit time and miss penalty, and memory address width). An example output is shown below (please use the same format).

```
CACHE CONFIGURATION
size = 32 KB
associativity = 4-way
cache line size = 32 B
write hit policy = write-back
write miss policy = write-allocate
cache hit time = 5 CLK
cache miss penalty = 100 CLK
memory address width = 48 bits
```

   - `print_statistics` outputs the statistics collected during the processing of a memory access trace (number of memory accesses, number of read/write accesses and read/write misses, number of cache evictions, number of memory writes and average memory access time). An example output is shown below (please use the same format).

```
STATISTICS
memory accesses = 2000000
read = 1251647
read misses = 21142
write = 748353
write misses = 12659
evictions = 33289
memory writes = 30005
average memory access time = 6.69005
```

- `print_tag_array` prints the index, the dirty bit (when applicable) and the tag **of the *valid* cache entries**. Please use the format of the example below:

```
BLOCKS 0
  index dirty         tag
     20     1  0x574fbe187
     52     1  0x574f4e930
     54     1  0x574f4e930
     58     1  0x574f4e930
     66     1  0x574fbe33a
     82     1  0x574fc0362
    104     1  0x574fc0c49
    106     1  0x574fc0c49
    150     0  0x574fc0c49
    154     1  0x574fbe2f5
    170     0  0x574fc0c49
    172     0  0x574fc0c49
    220     1  0x574fc04a1
BLOCKS 1
  index dirty         tag
     52     1  0x584f4e930
     54     1  0x584f4e930
     58     1  0x604f4e930
BLOCKS 2
  index dirty         tag
BLOCKS 3
  index dirty         tag
```

  In the format above, the *index* field is 7-character wide, the *dirty-bit* field is 6-character wide, and the *tag* field is 4-character wider than the tag (in hexadecimal format). You can use the `setw` function to set the width of the various fields. For example:

```
cout << setfill(' ') << setw(7) << "index" << setw(6) << "dirty" << setw(4+tag_bits/4)
<< "tag" << endl;
```

[ECE563 students only] Write a small tool that, given 3 matrices with size $N \times N$ (N being a parameter to the tool) and a 32-bit memory address, generates a trace with the sequence of memory operations triggered by a matrix multiplication code operating on the given matrices. The trace should contain only data accesses to the matrices (you can ignore instruction accesses and the loop iterator variable). Assume that the 3 matrices are laid out contiguously in memory starting at the specified address. Further, assume that the matrix multiplication code does not use tiling and that the matrices are laid out in row-major order. *Note*: the content of the matrices is irrelevant to this problem.

Report

Your report should be no longer than 5 pages (11 pt. Times New Roman font), including figures. For ECE463 students, the report should contain an analysis of the cache behavior of GCC (trace `GCC.t`). For ECE563 students, the report should contain an analysis of the cache behavior of the matrix multiplication code. More details on the analysis to be performed are provided below. For this project, *you do not need to include a description of your code in the report*. The report should cover the following aspects:

- **[ECE463 students]** Analysis of the miss rate of GCC (i.e., `GCC.t` trace) when varying the cache size, associativity, block size and write-hit/miss policies. In particular, your analysis should cover the following settings: (i) cache size: 16KB, 32KB, 64KB, 128KB, 256KB, 512KB; (ii) cache associativity: 1-way, 2-way, 4-way, 8-way, 16-way; (iii) block size: 32B, 64B, 128B, 256B, (iv) write-back/write-allocate and write-through/no-write-allocate write hit/miss policy combinations. When you analyze the impact of the cache size on the miss rate, you should fix the write-hit and write-miss policies, the associativity and block size (to a couple of relevant values), and vary the cache size. You should then plot the results in a chart with the cache size on the x-axis, the miss rate on the y-axis, and a data series for each relevant (associativity, block size) setting. You can draw two different charts for write-back/write-allocate and write-through/no-write-allocate policies. Similar charts should be plotted when analyzing the effect of the cache associativity and block size on the miss rate. You should include in your report a discussion of the results. Based on the results of your experiments, provide an estimate of the working set size of GCC.

- **[ECE563 students]** Assume that *N* is a multiple of the cache block size. Estimate the miss rate generated by the matrix multiplication code on a fully associative cache with write-back, write-allocate policy as a function of N, cache size and block size (i.e., write an equation that expresses the miss rate as a function of these parameters). Perform experiments to both evaluate the cache behavior of the matrix multiplication code and to validate your estimate. In the experiments, consider: (i) cache size: 16KB, 64KB, 256KB; (ii) cache associativity: 1-way, 2-way, 4-way, 16-way; (iii) block size: 32B, 64B, 128B, 256B, (iv) write-back/write-allocate policy, (v) two representative matrix sizes, one leading to cache evictions and one not. While your simulator does not need to support fully associative caches, you can assume that a 16-way cache approximates the behavior of a fully associative cache. Plot charts similar to the ones described above for ECE463 students, and discuss the results.

Tip to make data collection fast: As indicated above, the analysis requires you to perform a number of experiments with different cache size, associativity and block size configurations. The fastest way to collect the results is to write a test case that runs the experiments one after the other. For example, have a look at the code of test cases 4 and 5. Test case 4 includes a loop to test several cache associativity settings, and test case 5 includes a loop to go through multiple cache block sizes. You can write your scripts by copying/pasting&modifying these test case files. You can then add the "miss rate" to the values printed out at the end of each simulation. You can use the `cat` and `awk` utilities to quickly extract the values you are looking for from the output files. For example, say that I want to find all the values associated to "`read misses`" in the `testcase5.out` file.

```
>> cat testcase5.out | grep "read misses"
read misses = 34938
read misses = 34938
read misses = 37014
read misses = 52701

>> cat testcase5.out | grep "read misses" | awk '{print $4}'
34938
34938
37014
52701
```

You don't need to add an access latency and power analysis to your report. However, if you are interested in seeing how the cache configuration affects the cache hit time and energy consumption, you can use CACTI: http://www.cs.utah.edu/~rajeev/cacti6/.

Testing

As mentioned above, the compilation process generates a separate binary for each test case in the `testcases` folder. To execute `testcaseX`, you can invoke:

```
./bin/testcaseX
```

To check if your output is correct, you can compare it with file `testcaseX.out` in the `testcases` folder. On Linux, you can use the `diff` utility to do so.

For example, you can invoke

```
./bin/testcaseX > my_output

diff my_output testcases/testcaseX.out
```

The first command will run the test case and save its output into `my_output`. The second command will compare your output with the reference output line-by-line.

Grading guide

- Report = 35 points (if your code does not run correctly, use the outputs of the test cases provided to make an analysis of the data generated by the test cases for GCC).
- Code = max 35 points
- Test cases = 30 points (5 points per test case)

Submission instructions

1. **Report:** The format and content of the report are detailed above. Save your report in *pdf* format, with file name `project3_report.pdf`. Include the report in the `project3_code` folder.
2. **Test cases:** You should not modify any of the test cases. All the functionality should be included in the `cache.h` and `cache.c/cc`, files.
3. **Code:** Independently of the development environment and operating system you used to develop your code, your code should compile and run on the `grendel.ece.ncsu.edu` Linux machine, and it should compile using the provided `Makefile`. Have a look at Piazza for information on how to access the `grendel` machine.
4. You should invoke "`make clean`" before submitting your code. That is, your submission should not contain any object or binary files.
5. Your `project3_code` folder should contain the `project3_report.pdf` and the code.
6. Go to the parent folder of the `project3_code` folder. Compress the whole `project3_code` directory into a *tar.gz* file.

   ```
   tar -zcvf project3_code.tar.gz project3_code
   ```

7. Submit your project through Moodle.