Assignment 2

Due Dates:

At 11:59 pm on Friday October 19th, 2018

1. Purpose

The goal of this assignment is to develop a good understanding of the organization and operation of cache memories by writing a cache simulator. Your goal is to simulate a single level, N-way set associative cache for a given block size and using LRU replacement. You will be provided with a header file (*cachesim.h*) and a cache model file (*cachesim.c*) that defines the API. You are permitted to discuss the solution approaches with your classmates but must **write** the simulator on your own.

2. Background

Modify *cachesim.c* to implement your 1-level cache simulator that can simulate a "N-way" set associative cache with LRU replacement and a write-back replacement policy. Note the simulator must be parametric, i.e., the following should be able to take as parameters: line size, cache size, and associativity.

The basic elements are as follows

1. You only need to implement the cache directory and not the data section. For each cache block you need to keep track of the tag, valid bit, and dirty bit (was the cache line modified?). For each set you need to keep track of the LRU status of the lines in the set. Dynamically allocate and create the directory data structure.
2. Read and process each address from the trace file – read a memory reference, extract the index and tag, determine whether it is a hit or a miss, update the directory structure and any other data structures you add (for example counters keeping track of the number of hits or number of references). Do not forget to update the LRU status.
3. A script is provided for convenience to run simulations over a design space, e.g., over caches of with different line sizes.

You are simulating only one cache configuration at a time.

It is imperative that you implement the simulator on your own. Assignment submissions will be checked with code similarity analysis tools including against publicly and previously available code bases.

3. Assignment (100 pts)

Find the best configuration for a **64 Kbyte unified set associative cache** (instruction and data) that minimizes the sum of the overall miss rate across all traces, assuming a cold cache for each trace - the cache is initially empty and all lines are in the invalid state. The cache configuration is a combination of the line size and associativity. **Configuration parameters include: cache block size (32 byte, 64 byte, 128 byte, 256 byte, and 512 byte) and associativity (2, 4, 8)**. Your analysis should include the following:

1. Provide a plot of the **miss rate vs. the line size** for line sizes of 32 bytes to 512 bytes with a fixed associativity of 4 for each trace. Note that line sizes are a power of 2.
2. Compute the **overall miss rate**, the **read miss rate**, and the **write miss rate** for

each trace. For example, the read miss rate is the ratio of the total number of read misses to the total number of read operations. (Compute this using the best overall configuration that minimizes the sum of the overall miss rate across all traces)

3. Compute the volume of **write-back traffic in bytes** for each trace. (Compute this using the best overall configuration that minimizes the sum of the overall miss rate across all traces)
4. Compute the **total memory access volume** for each trace. This is the total number of bytes fetched from memory (not from the cache!). Compare this against the total number of bytes referenced. How much main memory access volume (in bytes) did the cache save? (Compute this using the best overall configuration across the traces)

4. Recommendations

Here are some suggestions for how to proceed.

1. Spend time thinking through the design, i.e., the directory data structure and all the counters you will need to record the requested information. Specify the data structures you need for the tag directory and LRU stacks. It is highly recommended that you draw the data structures and mentally walk through the processing of a reference. **Do this analysis before you ever write a line of code. It will save you a lot of time and stress. Most people spend time debugging and hacking away at a piece of code, which was written before they thought through the details.**
2. Write a test program to just be able to read the trace and parse the address into tag, and set. Make sure you can read the trace correctly and print the set index and tag. Once you are sure of this, then you can focus on the cache data structure.
3. Create test sequences of addresses rather than starting with the original traces.

   Create sequences for which you know the behavior. For example, create a sequence of 100 identical addresses – you will have one miss and 99 hits. Another test case is a repeating sequence of addresses that will conflict in the cache – for example two addresses with the same index but different tag and a direct mapped cache. You know the answer to these cases and can easily check the results for correctness.
4. Now run the full traces.

Thinking through the design of your simulator up front should minimize the time it takes you to complete the assignment. As mentioned earlier, debugging time is typically the biggest component of projects. Some clear up-front thinking will save you much of that time. Like previous projects, the actual code size is not very large for a C program.

5. Skeleton Code and Trace Files

A zip file, *cachesim.zip*, has been uploaded to **canvas.** This file includes the C code skeleton framework. For C code, only **`cachesim.c`** should need modification so that it contains your code. **`sim_driver.sh`** may also be modified to vary the parameter space as you wish. Other files should not be modified, if you find yourself doing so, you may be doing something incorrectly or unnecessarily.

- `makefile`
- `main.c`
- `cachesim.h`
- `cachesim.c`
- `sim_driver.sh` *(Shell script runs your code with different parameters)*

Several trace files are also included. Note that these are pretty big files (5 Mb - 165 Mb)

- `trace.bubble`
- `trace.random64k`
- `trace.stream1M`
- `trace.merge`

These traces include all memory accesses for instruction fetches, reads, and writes for each one of the four benchmarks. Each line of each trace file describes one memory access in the following format, with each field separated by a space.

- 1 character 'r' for reads, 'w' for writes, and 'i' for instruction fetches (which are reads).
- 8-digit hexadecimal virtual address (you can ignore this)
- 8-digit hexadecimal physical address (this is the address used to index the cache)
- Decimal access length. (you can ignore this)

For this assignment you can ignore the virtual address. You can run the individual simulations directly, but *sim_driver.sh* may be handy to run a larger set of experiments as you explore the parameter space (line sizes, cache sizes and associativities). The set of block sizes, cache sizes and associativities explored is set in the first three lines of the script. By default, this script runs on every trace in the current directory, but this can also be changed. If you get a "Permission denied" message when trying to run *sim_driver.sh*. Use this command **`chmod +x ./sim-driver.sh`**.

6. Preliminary self-testing to validate your code

Among the trace files are trace.random64k and trace.stream1M. To spot check the functionality of your simulator you can compare against the following input/output metrics.

Input Command Format:

```
./cachesim $tracefile $blocksize $cachesize $associativity
```

Console Output Format:

```
Accesses, Hits, Misses, Writebacks
```

Test Commands

```
Command 1:./cachesim trace.random64k 64 8192 4
Command 2:./cachesim trace.random64k 64 8192 16
```

```
Results:
Output 1: 262144, 32652, 229492, 0
Output 2: 262144, 32601, 229543, 0
```

If your cache simulator can replicate these results, you are on the right track. You can run these simulations by executing your code from the command line manually or via *sim_driver.sh* (make sure its configuration covers these parameters). **Note that replicating these results is not a guarantee that your code is 100% correct (you could have bugs that are not exposed with these tests), but it is a good sign. If you cannot replicate the above results, then you have some work to do to get your simulator to work correctly.**

7. Submission Instructions

Please read these carefully and follow them precisely. Failure to do so will result in a loss of points. Brevity and precision is valued over volume in your writing. Your submission should be a single zip file called **Assignment-2.zip** submitted to t-square which contains

1. A PDF document (**Assignment-2.pdf**) containing a summary of your
   a. data structures
   b. Plots of parameters described in Section 3.
2. **ONLY** the following files should be included (please remove everything else)
   a. cachesim.c
   b. cachesim.h
   c. main.c
   d. makefile
   e. **Files generated when you run *make.*** Three files (two .o files and an executable) should be generated when you run make on the Linux Lab in Klaus. Run your code there before you submit. **20 point deduction** if these files are missing or have issues requiring recompilation of your code.
3. ****DO NOT INCLUDE THE TRACE FILES!!!**→ 50 Point Deduction if you do.**

8. Grading Guidelines

IMPORTANT NOTES:

1. If your code does not compile your assignment cannot be graded.
2. If your code crashes/seg. faults or produces unreadable output it cannot be graded.
3. Your code will be graded according to a randomly selected combination of parameters (cache size, line size, associativities and trace file). Its accuracy will be compared against our solution simulator.

GRADING RUBRIC:

Program compiles and executes	20 points
Results are largely correct, but answers may be *slightly* incorrect	40 points
Results are precisely correct	50 points
PDF file contains a complete report of implementation	5 points
Code is documented	5 points
Demo	20 points
TOTAL	**100 points**

Note: No late assignments will be accepted. You must achieve a minimum average of 50% in the assignments to pass the course.

9. Extra Credit (50 points)

This portion will only be graded once you have completed the above parts of the assignment.

Extend your simulator to

1. Have a separate 16 Kbyte direct mapped I-cache and 16 Kbyte direct mapped D-Cache each with 64 byte lines.
2. Include a 256 Kbyte, K-way set associative unified L2 cache with line sizes between 64 bytes and 256 bytes and a write-back update policy.
3. Consider the configuration of the L2 with 256-byte lines and associativity 8
    a. Compute the local miss rates and the global miss rates for each trace for instructions and data.
    b. What value of associativity minimizes the global miss rate for data.
    c. What is the total volume of traffic between the L1 D-cache and the L2 for each trace.

Submission Requirements

Create a zip file named **Assignment-2-XC.zip** that includes a detailed PDF document, main.c, makefile, cachesim.c, cachesim.h, and the files generated when running make. Do not include the trace file or points will be deducted.