Assignment 5

Due Date: 11:55PM April 22nd 2016

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 true 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 fixed block size, LRU replacement, and a write-back replacement policy.** Note that the simulator must be parametric, i.e., it 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 an address, 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 (200 pts)

Find the configuration for a 128 Kbyte unified set associative cache (instruction and data) that minimizes the miss rate across two benchmark traces (*trace.bubble* and t*race.merge*) – minimize the sum of the miss rates assuming a cold cache for each trace,

i.e., 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. Your analysis should include the following.

1. Compute the **overall miss rate**, the **read miss rate**, and the **write miss rate**. For example, the read miss rate is the ratio of the total number of read misses to the total number of read operations.

2. Compute the volume of **write-back traffic in bytes**.

3. Compute the **total memory access volume**. This is the total number of bytes fetched from memory (not from the cache!). Compare this against the total number of memory references. How much memory access volume (in bytes) did the cache save?

4. Provide a plot of the **miss rate vs. the line size** for line sizes of 32 bytes to 2 Kbytes and associativity 4. Note that line sizes are a power of 2.

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, *Assignment5.zip*, has been uploaded to **t-square** under **Resources**. 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.

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 (block 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 u+rwx ./sim-driver.sh`** (works on Klaus Linux labs).

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.o $tracefile $blocksize $cachesize $associativity
```

Console Output Format:

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

Test Commands

```
Command 1:./cachesim.o trace.random64k 64 8192 4
Command 2:./cachesim.o 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 **Assignment5.zip** submitted to t-square which contains

1. A PDF document (**Assignment 5.pdf**) containing a summary of your
   a. data structures (expect 4/5 lines of code),
   b. function to parse the address (expect a few lines of code),
   c. implementation of cache LRU stack (expect a few lines of code)
   d. 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. Assignment 5 will be graded on computers in the LINUX lab (Klaus 1448).
2. You are responsible for ensuring compilation and correct execution of your code on LINUX computers in Klaus 1448 before you submit.
3. If your code does not compile your assignment cannot be graded.
4. If your code crashes/seg. faults or produces unreadable output it cannot be graded.
5. **Given the assignment comes at the end of the semester, there will not be any time to re-grade.**
6. 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	40 points
Results are largely correct, but answers may be *slightly* incorrect	40 points
Results are precisely correct	60 points
PDF file contains a complete report of implementation	35 points
Code is documented well	25 points
TOTAL	**200 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 (100 points)

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

In this part of the assignment, you will get familiarized with CACTI (*CACTI: An Enhanced Cache Access and Cycle Time Model*). CACTI is an integrated cache and memory access time, cycle time, area, leakage, and dynamic power model. By integrating all these models together, users can have confidence that tradeoffs between time, power, and area are all based on the same assumptions and, hence, are mutually consistent. CACTI is intended for use by computer architects to better understand the performance tradeoffs inherent in memory system organizations.

Your task is to do design space exploration and come up with design which *you think has optimal access time and total dynamic energy trade-off*.

Instructions on using CACTI:

1. We will provide you the tarball of CACTI 6.5.
2. untar the CACTI tar ball and use the README in the cacti folder to understand how to start using cacti.

3. We will provide you the configuration file, which you will tweak to implement your parameters and do design space exploration.

Configuration that will help you come up with cache parameters are:

1. You will use 64-bit virtual address;
2. Size of cache to vary are: 16kB, 32kB, 64kB, 128kB
3. You have to generate data for both serial lookup, parallel lookup and normal lookup {These will be the modes present in the configuration file}
4. There should be separate graphs for each mode, and each graph for a particular mode should contain all the cache sizes for a particular block size.
5. Thus you will have 3(mode of access of cache) X 2(cache block size) X 2 (energy and Time) graphs.
6. For plotting time graph use data-side and tag-side component similar to the example graph shown below. For plotting Energy graph use '*Total dynamic read energy/access'* for both data array and tag array as shown in the example graph
7. Vary the associativity and cache Block size {only between 8B or 64B size}.
8. Generate the graph and add it to your pdf report "extra_credit.pdf" to be submitted only when you have completed first part.

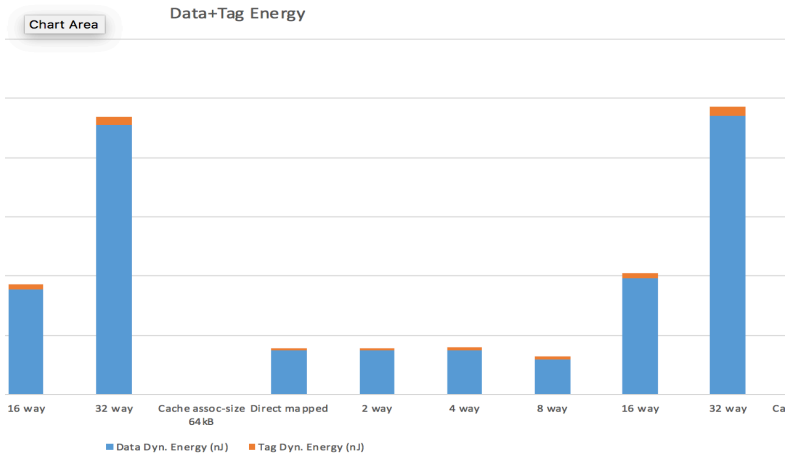
The purpose of this *extra-credit* assignment is to get you acquainted with the CACTI interface, and not necessarily the code base. So I encourage you to try to see how can different parameter be changed using configuration file from the command line.

Following should be included in your report:

Repeat these steps for *Parallel, Serial and Normal Lookup*

1. Graphs of your design space exploration, in order to get the optimal configuration as per your reasoning.
2. A single graph showing all the sizes of the cache sizes mentioned above with their access time

For example, please refer this figure below:



This is a sample graph. Here cache size and its associativity is plotted on the x-axis. Data and tag energy is plotted on the y-axis. Total energy is the sum of both data energy and tag energy, which is also the height of the whole bar.

Please follow similar graph style for your report.

Note: You have to calculate tag bits manually as per your virtual address and cache block size.

Note: Comment out your sensible parameter changes in the configuration file. **Do not delete your changes in configuration file.** As it will be graded.

Lastly, your report should contain the insight that you have got about cache while working on this assignment. [*your insight will also be graded*]

Grading rubric:

Your modified configuration file	25 marks
Graphs	50 marks
Insight	25 marks