Georgia Institute of Technology

School of Computer Science

CS 4290/6290, ECE 4100/6100: Spring 2018 (Prof. Conte)

Project 1: Cache design

Version 0.1

Due dates:

Checkpoint 1: Sunday, Feb 4th at 11:55pm

Final due date: Sunday, Feb 11th at 11:55pm

Rules

This is the first project for the course --- here are some rules:

1. Sharing of code between students is viewed as cheating and will receive appropriate action in accordance with University policy.

2. It is acceptable for you to compare your results, and only your results, with other students to help debug your program. It is not acceptable to collaborate either on the code development or on the final experiments.

3. You should do all your work in the C or C++ programming language, and should be written according to the C99 or C++11 standards, using only the standard libraries.

4. Unfortunately experience has shown that there is a very high chance that there are errors in this project description. The online version will be updated as errors are discovered. *It is your responsibility to check the website often and download new versions of this project description as they become available*.

5. A Makefile with the frontend will be given to you; you will only need to fill in the empty functions and any additional subroutines you will be using. You will also need to fill in the statistics structure that will be used to output the results.

Project Description:

Cache memories are hard to understand! One way to understand them is to actually build a cache. We do not have time to do this in this class. However, writing a simulator for a cache can also make caches easier to understand. So in this project, you will design a parametric cache simulator and use it to design data caches well suited to the SPEC benchmarks.

Logistics:

There are two parts to this project. In the first part, you will implement a basic cache. Your outputs should match the reference outputs **exactly.** You need to submit this by checkpoint 1's due date. In the second part, you'll add more features and experiments, which you need to turn in by the final deadline. For any questions and clarifications, we encourage you to post on piazza and/or see the TAs during office hours. We will update this document with details about the second part shortly. We'll send an email when the second part is posted. **You should download a revised copy when it is posted.**

Simulator specifications:

Cache simulation capabilities:

The simulator should model a cache with 2^C bytes of data storage, having 2^B-byte blocks, and with sets of 2^S blocks per set (note that S=0 is a direct-mapped cache, and $S = C - B$ is a fully associative cache). The cache uses the **LRU** replacement policy.

Notes:

- The provided traces have a stream of read and write addresses. You can assume that each access is a 1 byte access. i.e., a single access won't span multiple cache blocks.
- In your cache, the set index bits are the lowest contiguous bits in the address before the block offset.
- The cache has a write-back write-allocate (**WBWA**) policy. You need to keep track of dirty cache blocks with a dirty bit.
- Additionally, there is a valid bit per block, which is initially 0 when the simulation starts.
- When accounting for storage requirements, you also need to account for tag storage.

The following are used to calculate the average access time as seen by the CPU (AAT). Hit time is denoted by HT, miss penalty by MP and miss ratio by MR:

- AAT = HT + MR * MP
- HT = 2 + 0.2*S

- MP = 20
- Time units are in nanoseconds

In general, (*C, B, S*) completely specifies the caching system

Explanation of Provided Framework

We are providing you with a framework to build the cache simulator. You must fill in the following functions in the framework:

```
void setup_cache(uint64_t c, uint64_t b, uint64_t s);
```

Subroutine for initializing the cache. You many add and initialize any global or heap variables as needed.

```
void cache_access(char type, uint64_t arg, cache_stats_t* p_stats);
```

Subroutine that simulates the cache one trace event at a time. Type can be either READ or WRITE, which is each defined in cachesim.hpp. A READ event is a memory load operation of 1 byte to the address specified in arg. A WRITE event is a memory store operation of 1 byte to the address specified in arg.

```
void complete_cache(cache_stats_t *p_stats);
```

Subroutine for cleaning up memory and calculating overall system statistics such as miss rate or average access time.

The code skeleton is only a loose guideline to help you get started. You are free to modify it in any way you see fit.

***Statistics (output)*: The simulator must output the following statistics after completion of the run:**

For each entry in the trace file, print "H" or "M" for a hit or a miss in the cache.
Also print these statistics:

a. number of reads, writes and total accesses to the cache

b. number of read hits and misses

c. number of write hits and misses

d. total hits and misses

e. hit and miss ratios for reads and writes separately as well as overall ratios

f. number of write-backs to the lower level of memory hierarchy

g. the average access time (AAT) for the cache

The output of these variables should be handled by the driver code, and you only need to fill in the structure cache_statistics_t, defined in cachesim.hpp.

Experiments:

Checkpoint 1: Proof that your simulator works. This is done via validating your simulator (see the section on *validation requirement* below).

Checkpoint 2: We'll update this document with additional features and experiments in the next few days.

Validation Requirement for Checkpoint 1:

Four sample simulation outputs will be provided on canvas by the TAs. You must run your simulator and debug it until it matches 100% all the statistics in the validation outputs posted on the website. You are required to hand in this validated output with your code by checkpoint 1 (see below).

What to hand in via Canvas for checkpoint 1:

1. Output from your simulation showing it matches 100% all the statistics in the validation outputs posted on the website for each validation run

2. The commented source code for the simulator program itself.

3. Remember that your code must compile and run on a current variant of Linux (i.e., Debian, Red Hat, Ubuntu) running on an x86 architecture (i.e., Intel or AMD).

4. Code that cannot be compiled will result in a zero.

5. **Note that late projects will not be accepted.**

Grading:

Checkpoint 1:

- 0% You do not hand in anything by the deadline *or* you hand in someone else's simulator [Plagiarism will not be tolerated]
- +50% Your code runs and matches all validation runs

Checkpoint 2:

- TBD