CS180, Fall 2021
Cache assignment: Understanding Cache Memories

1 Logistics

This is an individual project. You must run this assignment on a 64-bit x86-64 machine.

2 Overview

This assignment will help you understand the foundations of a CPU cache.

You will write a small C++ program (about 200-300 lines) that simulates the behavior of a cache memory.

3 Downloading the assignment

Download the assignment archive from the course site.

Start by extracting it to a protected Linux directory in which you plan to do your work.

You will be modifying the file `csim.cpp`. To compile the project files, type:

```
linux> make clean
linux> make
```

4 Description

You will implement a cache simulator to keep track of a program's cache hit statistics. Specifically it will count how many cache hits, misses and evictions there are.

Your simulator will model **one** cache where it's size and associativity are customizable.

Note that it will not need to know nor keep track of the programs variable values. It will only process memory access addresses.

4.1 Reference Trace Files

Your program will be provided with a trace file that contains a history of address memory accesses.

The `traces` subdirectory of the handout directory contains a collection of *reference trace files* that we will use to evaluate the correctness of the cache simulator you write. The trace files are generated by a Linux program called `valgrind`. For example, typing

```
linux> valgrind --log-fd=1 --tool=lackey -v --trace-mem=yes ls -l
```

on the command line runs the executable program `ls -l`, captures a trace of each of its memory accesses in the order they occur, and prints them on `stdout`.

`Valgrind` memory traces have the following form:

```
I 0400d7d4,8
 M 0421c7f0,4
 L 04f6b868,8
 S 7ff0005c8,8
```

Each line denotes one or two memory accesses. The format of each line is

```
[space]operation address,size
```

The *operation* field denotes the type of memory access: "I" denotes an instruction load, "L" a data load, "S" a data store, and "M" a data modify (i.e., a data load followed by a data store). There is never a space before each "I". There is always a space before each "M", "L", and "S". The *address* field specifies a 64-bit hexadecimal memory address. The *size* field specifies the number of bytes accessed by the operation. For this assignment we will ignore the *size* field.

Your program will need to be able to parse these trace files. Here is an example of how one could get started.

```
#include <fstream>
#include <string>
#include <iostream>
using namespace std;
int main(void)
{
    ifstream  fin{ "dave.trace" };
    char      type{'\0'};
    long long addr{0};
    string ignore{};
    while (fin >> type >> std::hex >> addr >> ignore)
    {
        cout << type << " " << std::hex << addr << '\n';
    }
    return 0;
}
```

Here we are opening up an input file stream and parsing the information we expect line by line. We use a `char` to get the type of instruction, an `long long` and `std::hex` to parse the integer address value and lastly we using a `std::string` to eat the remaining characters which we will ignore for this assignment.

4.2 Writing a Cache Simulator

You will write a cache simulator in `csim.cpp` that takes a `valgrind` memory trace as input, simulates the hit/miss behavior of a cache memory on this trace, and outputs the total number of hits, misses, and evictions.

We have provided you with the binary executable of a *reference cache simulator*, called `csim-ref`, that simulates the behavior of a cache with arbitrary size and associativity on a `valgrind` trace file. It uses the LRU (least-recently used) replacement policy when choosing which cache line to evict.

The reference simulator takes the following command-line arguments:

```
Usage: ./csim-ref [-hv] -s <s> -E <E> -b <b> -t <tracefile>
```

- `-h`: Optional help flag that prints usage info
- `-v`: Optional verbose flag that displays trace info
- `-s <s>`: Number of set index bits ($S = 2^s$ is the number of sets)
- `-E <E>`: Associativity (number of lines per set)
- `-b <b>`: Number of block bits ($B = 2^b$ is the block size)
- `-t <tracefile>`: Name of the `valgrind` trace to replay

The command-line arguments are based on the notation (s, E, and b) from the textbook. For example:

```
linux> ./csim-ref -s 4 -E 1 -b 4 -t traces/yi.trace
hits:4 misses:5 evictions:3
```

The same example in verbose mode:

```
linux> ./csim-ref -v -s 4 -E 1 -b 4 -t traces/yi.trace
L 10,1 miss
M 20,1 miss hit
L 22,1 hit
S 18,1 hit
L 110,1 miss eviction
L 210,1 miss eviction
M 12,1 miss eviction hit
hits:4 misses:5 evictions:3
```

Your job is to fill in the `csim.cpp` file so that it takes the same command line arguments and produces the identical output as the reference simulator. Notice that this file is almost completely empty. You'll need to write it from scratch.

Programming Rules

- Include your name and loginID in the header comment for `csim.cpp`.
- Your `csim.cpp` file must compile without warnings in order to receive credit.
- Your simulator must work correctly for arbitrary s, E, and b. This means that you will need to allocate storage dynamically for your simulator's data structures.
- For this assignment, we are interested only in **data cache performance**, so your simulator should ignore all instruction cache accesses (lines starting with "I"). Recall that `valgrind` always puts "I" in the first column (with no preceding space), and "M", "L", and "S" in the second column (with a preceding space). This may help you parse the trace.
- To receive credit, you must call the function `printSummary`, with the total number of hits, misses, and evictions, at the end of your `main` function:

```
printSummary(hit_count, miss_count, eviction_count);
```

- For this this assignment, you should assume that memory accesses are aligned properly, such that a single memory access never crosses block boundaries. By making this assumption, you can ignore the request sizes in the `valgrind` traces.
- We are not interested in your C programming skills, so you do not need to write your own logic for something like a dynamic array. Utilize the C++ standard library to keep your program as simple and as straightforward as you can. The point of this assignment is demonstrate your understanding of the logic behind CPU caches and to express hat through code.

5 Working on the Assignment

We have provided you with an testing program, called `test-csim`, that tests the correctness of your cache simulator on the reference traces. Be sure to compile your simulator before running the test:

```
linux> make
linux> ./test-csim
                        Your simulator     Reference simulator
Points (s,E,b)    Hits  Misses  Evicts    Hits  Misses  Evicts
     3 (1,1,1)       9       8       6       9       8       6  traces/yi2.trace
     3 (4,2,4)       4       5       2       4       5       2  traces/yi.trace
     3 (2,1,4)       2       3       1       2       3       1  traces/dave.trace
     3 (2,1,3)     167      71      67     167      71      67  traces/trans.trace
     3 (2,2,3)     201      37      29     201      37      29  traces/trans.trace
```

```
 3 (2,4,3)     212      26      10     212      26      10  traces/trans.trace
 3 (5,1,5)     231       7       0     231       7       0  traces/trans.trace
 6 (5,1,5)  265189   21775   21743  265189   21775   21743  traces/long.trace
27
```

For each test, it shows the number of points, the cache parameters, the input trace file, and a comparison of the results from your simulator and the reference simulator.

Here are some hints and suggestions:

- Do your initial debugging on the small traces, such as `traces/dave.trace`.
- The reference simulator takes an optional `-v` argument that enables verbose output, displaying the hits, misses, and evictions that occur as a result of each memory access. You are not required to implement this feature in your `csim.cpp` code, but we strongly recommend that you do so. It will help you debug by allowing you to directly compare the behavior of your simulator with the reference simulator on the reference trace files.
- We recommend that you use the `getopt` function to parse your command line arguments. You'll need the following header files:

  ```
  #include <getopt.h>
  #include <stdlib.h>
  #include <unistd.h>
  ```

  See "`man 3 getopt`" for details.
- Each data load (L) or store (S) operation can cause at most one cache miss. The data modify operation (M) is treated as a load followed by a store to the same address. Thus, an M operation can result in two cache hits, or a miss and a hit plus a possible eviction.

6 Grading Rubric

- ☐ **[core]** Create a simple cache simulator
- ☐ **[core]** Results match the reference simulator with the provided trace files and with other trace files
- ☐ Utilized `getopt` for implementing command line arguments
- ☐ Created verbose mode
- ☐ Used only C++ standard libraries and not the C standard libraries.
- ☐ All source files compile without warnings and without errors.
- ☐ Correct files submitted. No unnecessary files submitted
- ☐ Source Code has a proper header comment: Name, Assignment Name, Course Number, Term & Year.

Score	Assessment
Zero	Nothing turned in at all
Failing	Attempted something
Rudimentary	Close to meeting core requirements
Satisfactory	Meets all of the core requirements
Good	Clearly meets all requirements
Excellent	High quality, well beyond the requirements

6.1 Matching the Reference Simulator

We will run your cache simulator using different cache parameters and traces. There are eight test cases.

```
linux> ./csim -s 1 -E 1 -b 1 -t traces/yi2.trace
linux> ./csim -s 4 -E 2 -b 4 -t traces/yi.trace
linux> ./csim -s 2 -E 1 -b 4 -t traces/dave.trace
linux> ./csim -s 2 -E 1 -b 3 -t traces/trans.trace
linux> ./csim -s 2 -E 2 -b 3 -t traces/trans.trace
linux> ./csim -s 2 -E 4 -b 3 -t traces/trans.trace
linux> ./csim -s 5 -E 1 -b 5 -t traces/trans.trace
linux> ./csim -s 5 -E 1 -b 5 -t traces/long.trace
```

You can use the reference simulator `csim-ref` to obtain the correct answer for each of these test cases. During debugging, use the `-v` option for a detailed record of each hit and miss.

For each test case, outputting the correct number of cache hits, misses and evictions will indicate a correct implementation for that test case.

Running `test-csim` will give you a score to communicate how well your implementation matches the reference simulator. This will be used to gauge if your implementation is on the right track or not.

7 Submission

Submit your `csim.cpp` file on the course site.