CS 4290 - Summer 2021
Project 1 : Cache Hierarchy Simulator

Due: June 21^{st} 2021 @ 11:59 PM

Patrick Lavin

Version : 1.0

I Rules

- **This is an individual assignment. ABSOLUTELY NO COLLABORATION IS PERMITTED.** All cases of honor code violations will be reported to the Dean of Students. See Appendix A for more details.
- The due date at the top of the assignment is final. Late assignments will not be accepted.
- Please use office hours and Piazza for getting your questions answered. If you are unable to make it to office hours, please email me and I'll be happy to meet you some other time.
- This is a tough assignment that requires a good understanding of concepts before you can start writing code. **Make sure to start early**.
- Read the entire document before starting. Critical pieces of information and hints might be provided along the way.
- Unfortunately, experience has shown that there is a chance that errors in the project description will be found and corrected after its release. **You are responsible for checking for updates to the project. Any changes will be announced on Canvas.**
- Make sure that all your code is written according to **C++11** standards, using only the standard libraries.

II Introduction

Caches are complex memories that are often difficult to understand. One way to understand them is to build them. However, building caches is time consuming (and kinda infeasible, requiring clean rooms and what not), and as is often the case with computer architecture , writing a simulator is nearly as effective and far more time-efficient. Hence, in this project, you will write a cache simulator that can simulate memory traces containing data addresses, verify its functional correctness on traces from the SPEC 2017 benchmark suite, and finally run experiments on a given workload to find an optimal cache configuration for that workload.

The simulator should support a configurable data cache with an optional victim cache. The replacement policy is configurable between *LRU, FIFO and LFU-NotMRU*, while the write policy is *WBWA*. Section III provides the specifications of the cache organization, and section IV provides useful information concerning the simulator.

III Simulator Specifications

i Cache Hierarchy Layout

Your simulator should model an L1 Data Cache which is also connected to a Victim cache (Figure 1). Detailed specifications are below:

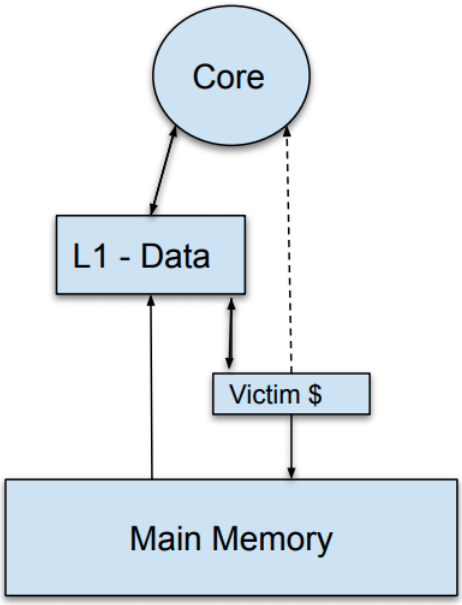

Figure 1: The cache hierarchy organization showing only the data movement lines. Note that not all data movement lines are explicitly modeled in the simulator.

- Accesses, either *Loads* (`type==Load`) or *Stores (type==Store)*, first access the L1 cache, and on a miss access the Victim Cache. The Victim Cache sends retrieved data to both the CPU and the L1-Data cache.
- The L1 cache is represented with three parameters (C, B, S) where:
  - 2^C is the cache size in bytes
  - 2^B is the size of each block in bytes
  - 2^S is the number of ways in each set
- The victim cache is configurable, i.e. it may or may not be present in the simulation. When it is present, it holds `V` blocks, found in the configuration file. `V` will always be less than or equal to 4. The Victim Cache uses the *FIFO* replacement policy (defined below, in the bullet point on L1 replacement policies).
- Memory addresses for each access are 64-bits long.
- The *Miss Penalty* for accessing main memory is 60 cycles.
- The caches are *byte* addressable.
- The L1 cache implements the *WBWA* - Write Back, Write Allocate policy
- The L1 cache uses one of the following replacement policies:
  - *LRU* - Least Recently Used (the oldest block in age of use is kicked out)

  - *FIFO* - First In First Out (the oldest block in age of first use is kicked out)
  - *LFU-NotMRU* - Least Frequently Used-Not the Most Recently Used (The least Frequently block which is not the Most Recently Used block is kicked out)

- All valid bits and dirty bits are set to 0 when the simulation begins.
- Each simulation is configured via a configuration file which describes the cache parameters for the L1 cache, the replacement policy, and the size of the victim cache (if applicable). A sample file is shown in listing 1.

Listing 1: Sample Configuration File

```
{
  "L1 Data" : {
    "C" : 15,
    "B" : 6,
    "S" : 3
  },
  "Victim Cache" : {
    "V" : 4
  },
  "Replacement Policy" : "LRU",
}
```

ii Cache Operations

- First, the L1 cache is checked for the block. A miss triggers a search in the victim cache. If the block is found in the victim cache, it is promoted to the L1 cache. If the block is not found in the victim cache, it is fetched from main memory. In reality, the Victim cache is checked in parallel with the L1 cache, and its access time is encompassed by the access time of the L1 cache. This effect is achieved in the simulation by assuming that hit time of the victim cache is 0.
- Blocks installed in the L1 cache from the victim cache follow the replacement policy for insertion. That is, an installation from victim cache to the L1 cache is similar to fetching the block from main memory, except the block's dirty bit is preserved.
- Replacement policy information is updated on hitting or installing a block in a cache. More specifically update the replacement information if:
  - An access hits in the L1 cache
  - A block is installed in the L1-Data cache from the Victim Cache
  - A block is installed in either of the L1-caches from main memory
- If the access to the data cache is a write (i.e. a store instruction), the dirty bit should be set
- The victim cache stores blocks evicted from the L1 cache. It uses FIFO replacement, which means that if a new block has to be inserted in the victim cache when it is full, the oldest block in the victim cache is kicked out. If the victim block is dirty, it is written back to main memory.
- The LFU-NotMRU replacement policy can have conflicts wherein two blocks could have the same use frequency. In that scenario, ties can be arbitrarily broken. In our simulator, we break ties by evicting the block with the higher tag. For example, if one block has tag `0x100`, and the other has tag `0x200`, and they both have the same use frequency, the block with tag `0x200` is evicted.

iii Cache Statistics (The Output)

For every access performed by the cache hierarchy, you will be updating the appropriate statistic described in the below structure definition. Note that the `l1data_hit_time` stat is set by the driver according to Table 1 below, and the `l1data_miss_penalty` stat is set by the driver to be 60. Everything else in this struct should be set by your program.

```
struct sim_stats_t {
    // L1 Data Cache (L1D) statistics
    uint64_t l1data_num_accesses;           // Total L1D Accesses
    uint64_t l1data_num_accesses_loads;     // L1D Accesses which are Loads
    uint64_t l1data_num_accesses_stores;    // L1D Accesses which are Stores
    uint64_t l1data_num_misses;             // Total L1D Misses
    uint64_t l1data_num_misses_loads;       // L1D Misses which are Loads
    uint64_t l1data_num_misses_stores;      // L1D Misses which are Stores
    uint64_t l1data_num_evictions;          // Total blocks evicted from L1D

    // Victim Cache (VC) statistics
    uint64_t victim_cache_accesses;         // Number of accesses to the VC
    uint64_t victim_cache_misses;           // Accesses that miss in the VC
    uint64_t victim_cache_hits;             // hits in the VC

    // Memory bus statistics
    uint64_t num_writebacks;                // Number of writebacks
    uint64_t bytes_transferred_from_mem;    // Number of bytes transferred from
                                            // main memory
    uint64_t bytes_transferred_to_mem;      // Number of bytes transferred to
                                            // main memory

    // Performance Statistics
    double l1data_hit_time;                 // L1 Data Hit Time
    double l1data_miss_penalty;             // L1 Data Miss Penalty
    double l1data_miss_rate;                // L1 Data Miss Rate
    double victim_cache_miss_rate;          // Miss rate of the victim cache

    double avg_access_time;                 // Average Access Time per access
};
```

Table 1: L1 Hit Time (Cycles)

Config	512	1K	2K	4K	8K	16K	32K
DM	1	1	2	2	2	2	2
$2W$	1	2	2	2	2	3	3
$4W$	2	2	2	3	3	3	4
$8W$	2	3	3	3	3	3	4
FA	4	4	4	5	5	6	6

IV Implementation Details

You have been provided with the following files in `Project1.tar.gz`:

- `src/cache.hpp` : A header file with declaration of functions you will be filling out and the `struct sim_stats_t` definition.
- `src/cache.cpp` : The file containing the functions you will be writing
- `src/cachesim_driver.cpp` : The `main` function and driver for the simulation framework.
- `CMakeLists.txt` : A cmake file that will generate a makefile for you.

- `config/` : A number of configure files utilizing the different functionality of your cache.
- `traces/` : A folder containing read-write address traces from real SPEC 2017 programs. Each trace contains 10 Million Load/Store accesses. A trace looks like:

```
L 0x7ffd31a002ac    // Load Access
L 0x55eeafee88f4
S 0x55eeafee88f7    // Store Access
L 0x7ffd31a00298
...
```

- `verify.py` : This file is used for validation. See section V.
- `solution.log` : This file is used for validation. See section V.

  There is another file, `experiments.tar.gz` containing more traces that you will use in section VI.

i Provided Framework

I have provided you with a framework that reads the address trace line by line, and calls the cache access function, one access at a time. Make sure you carefully read the provided code to fully understand what is going on before you start implementing your solution. **You only need to edit** `cache.cpp`. The following functions need to be filled out, but you may add helper functions and global variables as you see fit.

1. `void sim_init(struct sim_config_t *conf)`

   Initialize your cache hierarchy and any globals that you might need here. The struct `sim_config_t` is defined below and is populated by the driver file:

```
// Replacement policies
enum replacement_policy {LRU = 1, LFU = 2, FIFO = 3};

// Struct for storing per Cache parameters
struct cache_config_t {
    uint64_t c;
    uint64_t b;
    uint64_t s;
};

// Struct for tracking the simulation parameters
struct sim_config_t {
    struct cache_config_t l1;
    bool has_victim_cache;
    uint64_t victim_cache_size;
    enum replacement_policy rp; // replacement policy
 };
```

2. `void cache_access(uint64_t addr, char type, struct sim_stats_t *sim_stats, struct sim_config_t *sim_conf);`

   Subroutine for simulating cache events one access at a time. The addresses are all 64-bit unsigned values, the The access type (`type`) can be either `LOAD` or `STORE` type which are defined as `char` constants in the `cache.hpp` header file. The simulation configuration and statistics structs are defined in the above sections.

3. `void sim_cleanup(struct sim_stats_t *stats)`

Perform any memory cleanup and finalize required statistics (such as average access time, miss rates, etc.) in this subroutine.

ii Building the Simulator

I have provided you with a `CMakeLists.txt` file that is used to build the simulator. CMake generates a makefile depending on the system configuration you are trying to build the simualator on. Follow these steps on a UNIX like machine:

```
Unix: $ cd <Project Directory>
Unix: $ mkdir build && cd build
Unix: $ cmake ..
Unix: $ make
```

This will generate an executable `cachesim` in the `<Project Directory>/build` folder. This is all that you need to know about using `cmake`. If you are interested in learning more about this build system, you can start by reading this link.

The generated executable `cachesim` can be run with the following command:

```
Unix: $ ./cachesim -c <configuration file> -i <trace file>
```

For example, if the executable is in the build folder as described above and the other directory structures have been preserved, you can run a simulation for the default configuration on the gcc trace by:

```
Unix: $ ./cachesim -c ../config/default.conf -i ../traces/gcc.trace
```

Note: You will need to have cmake installed on the machine you are using. You should be able to install it using the package manager on your choice of Linux distribution or from this link if you are using a machine running MacOS.

iii Things to Watch Out For and Hints

This section describes the design choices the validation solution has made. You will need to follow these guidelines to perfectly match the validation log.

- Before you write even a single line of code, I strongly suggest understanding all the cache concepts and figuring out how and when each level of the hierarchy is accessed. Writing the flow of accesses on a piece of paper will be a good starting point!
- LRU and FIFO require some notion of time. However, this does not mean that you should use the time utility to implement LRU. Maintaining a global clock in the simulation is sufficient.
- LFU-NotMRU is a special case of LFU where the most recently used block is not evicted. One way to achieve this is to keep track of the most recent block per set in some data-structure.
- The Tag width of the L1 cache and the victim cache may not be the same size for a given set of parameters.
- The program uses C++ in case any of you wish to use C++ features. If you are more familiar with C you can just write C and it should work fine.
- You may wish to create simpler traces of your own for debugging purposes.

V Validation Requirements

You must run your simulator and debug it until the statistics from your solution **perfectly (100 percent)** match the statistics in the validation log for all test configurations. This requirement must be completed before you can proceed to the next section.

We have provided you with a `Python` run script and configuration files which will run the simulator for various configurations and traces, and generate a log file (called `verify.out`). You can then compare this generated log file against the `solution.log` file. The script assumes that the simulator executable will be in the `build` folder. To run the verification script and compare against the given solution log:

```
Unix: $ python3 verify.py
Unix: $ diff -w verify.out solution.log
```

VI Experiments

After you have debugged and validated your cache simulator, you will find the optimum cache configuration for the two versions of the same workload, a matrix-matrix multiplication kernel and the tiled version of the same. These traces are in `experiments.tar.gz`. The best configuration should follow the following constraints:

- Have the lowest average access time. Recall that AAT will primarily depend on *Hit Time* and *Miss Rate*. Try to play around with Cache parameters to find trends.
- The total size of the L1 cache, including both the Tag Store and the Data Store, must not exceed *48 KiB* (KiB = Kibi-byte = 2^{10} bytes). Note that you will have to workout the metadata size including tag, valid and dirty bits. For simplicity, you **should not** include the replacement bits as a part of this budget.
- The victim cache **may not have more than 4 entries**.
- I strongly encourage you try a few distinct configurations and understand the trends rather than trying to brute force search the entire search space. These traces are larger than the ones used for validation so it will take longer to simulate them.
- **Note:** Metadata calculation should not be based on the size of your class/struct in the simulator code, but rather on what the (C, B, S) parameters would yield in reality.

After you are done evaluating the two versions of the workload, write a short report on your findings, describing the optimum cache configuration for each, and explain any surprising or unexpected results. Please limit this report to a maximum of **2 pages**. Ensure that the report is in a file named `<last name>_report.pdf` (please substitute your own last name into the filename and do not include the angle brackets).

VII What to Submit to Canvas

You need to submit the following files to Canvas:

- `cache.cpp`
- `<last name>_report.pdf`

VIII Grading

You will be evaluated on the following criterion:

+0 :	You don't turn in anything (significant) by the deadline
+50 :	You turn in well commented significant code that compiles and runs but does **not** match the validation
+30 :	Your simulator **completely matches** the validation output
+15 :	You ran experiments and found the optimum configuration for the 'experiments' workload and presented sufficient evidence and reasoning
+5 :	Your code is well formatted, commented and does not have any memory leaks! Check out the section on helpful tools

Appendix A - Plagiarism

We take academic plagiarism very seriously in this course. Any and all cases of plagiarism are reported to the Dean of Students. You may not do the following in addition to the Georgia Tech Honor Code:

- Copy/share code from/with your fellow classmates or from people who might have taken this course in prior semesters.
- Look up solutions online. Trust us, I will know if you copy from online sources.
- Debug other people's code. You can ask for help with using debugging tools (Example: Hey XYZ, could you please show me how GDB works), but you may not ask or give help for debugging the cache simulator.
- You may not reuse any code from earlier courses even if you wrote it yourself. This means that you cannot reuse code that you might have written for this class if you had taken it in a prior semester. You must write all the code yourself and during this semester.

Appendix B - Helpful Tools

You might the following tools helpful:

- **GDB:** The GNU debugger will be really helpful when you eventually run into that seg fault. The cmake file provided to you enables the debug flag which generates the required symbol table for GDB by default.
- **Valgrind:** Valgrind is really useful for detecting memory leaks. Use the below command to track all leaks and errors.

```
Unix: $ valgrind --leak-check=full
   --show-leak-kinds=all --track-fds=yes
   --track-origin=yes -v ./cachesim <cachesim
   arguments as you would usually provide them>
```