CS {4/6}290 & ECE {4/6}100 - Spring 2020
Project 1 : Cache Hierarchy Simulator

Dr. Thomas Conte

Due: February 14^{th} 2020 @ 11:59 PM

Version : 0.97

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 for getting your questions answered. If you are unable to make it to office hours, please email the TAs.
- 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 high chance there errors in the project description will be found and corrected after release. **It is your responsibility to check for updates on Canvas, and download updates if any.**
- Make sure that all your code is written according to **C99 or 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 as the case with computer architects often is, you will instead write a simulator. More specifically, in this project, you will write a cache hierarchy simulator that can simulate memory traces containing both instruction and data addresses, and then run experiments on the given workload traces (from the SPEC 2017 benchmark suite) to find the best cache configurations for each of them. The caches will be organized into configurable, split L1 caches (instruction and data), and a unified L2 cache. The hierarchy will also follow configurable replacement and write policies. Section III provides the specifications of the cache hierarchy, and section IV provides useful information concerning the simulator.

Note: This lab has to be done individually. Please follow all the rules from section I and checkout Appendix A.

III Simulator Specifications

i Cache Hierarchy Layout

Your simulator should model an Instruction-L1 cache and a Data-L1 cache connected to a Unified-L2 cache (Figure 1). Detailed specifications are below:

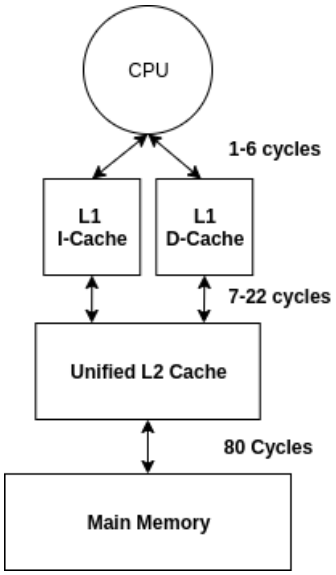

Figure 1: The cache hierarchy organization

- Instruction accesses `(type==Instruction)` access the L1-Instruction cache while data accesses `(type==Load or type==Store)` access the L1-Data cache. On a miss, the Unified L2 cache is access.
- Each cache in the hierarchy 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 caches are organized in a *non-inclusive* hierarchy. This means that cache blocks (i.e. the data of a cache block) present in the L1 cache need not be in the L2 cache. However, unlike the exclusive hierarchy, there is no requirement that the L1 and L2 cannot have the same data block.
- Memory addresses for each access are 64-bit long.
- The caches are *byte* addressable.
- The caches implement one of two write policies:
  - *WBWA* - Write back, write allocate
  - *WTWNA* - Write through, write no allocate
- All the caches use the same replacement policy, which is one of:
  - *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* - Least Frequently Used (the block which has been accessed least times is kicked out)

- All valid bits and dirty bits (If applicable) are set to 0 when the simulation begins.
- Each simulation is configured via a configuration file which describes the cache parameters for all the caches, and the replacement and write policies for the simulation. A sample file is shown in listing 1.

Listing 1: Sample Configuration File

```
{
  "L1 Instruction" : {
    "C" : 15,
    "B" : 6,
    "S" : 3
  },
  "L1 Data" : {
    "C" : 16,
    "B" : 6,
    "S" : 3
  },
  "L2 Unified" : {
    "C" : 20,
    "B" : 6,
    "S" : 3
  },
  "Replacement Policy" : "LRU", // "LRU", "LFU" or "FIFO"
  "Write Policy" : "WBWA",      // "WBWA" or "WTWNA"
}
```

ii Cache Operations

- Based on the type of access the appropriate L1 cache is checked. If the L1 cache misses, the unified L2 cache is checked, and if the block is found there, it is installed in the L1 cache. The victim block from the L1 cache (based on replacement policy) is installed in the L2 cache.
- If the L2 cache also misses on an access, the block is fetched from main memory, first installed in the L2 cache, and then installed in the appropriate L1 cache. This could cause evictions from both the L2 and L1 caches. Dirty L2 evictions (where applicable) should be written back to main memory and all L1 evictions should be installed in the L2 cache. Make sure to update the dirty bit in the L2 block when installing a dirty L1 eviction into the L2 cache.
- If the access was a store, only the top level cache (L1 cache in this case) will be marked dirty if applicable (based on the write policy).
- Replacement policy information is updated on hitting or installing in a cache. More specifically update the replacement information if:
  - An access hits in either the L1 cache or the L2 cache
  - A block is installed in the L1 cache from the L2 cache
  - A block is installed in the L2 cache either from main memory or from the L1 cache (i.e the L1 victim which is now being installed in the L2 cache updates the L2 cache's replacement information for that block)

  Note: More than one block's replacement information may be updated for a single demand access based on the above rules.

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:

```
struct sim_stats_t {

    // L1 Instruction Cache statistics
    uint64_t l1inst_num_accesses;           // Total L1 Inst Accesses
    uint64_t l1inst_num_misses;             // Total L1 Inst Misses
    uint64_t l1inst_num_evictions;          // Total blocks evicted from
        L1-Instruction cache

    double l1inst_hit_time;                 // L1 Inst Hit Time
    double l1inst_miss_penalty;             // L1 Inst Miss Penalty
    double l1inst_miss_rate;                // L1 Inst Miss Rate
    double l1inst_AAT;                      // L1 Inst Average Access Time

    // L1 Data Cache statistics
    uint64_t l1data_num_accesses;           // Total L1 Data Accesses
    uint64_t l1data_num_accesses_loads;     // L1 Data Accesses which are Loads
    uint64_t l1data_num_accesses_stores;    // L1 Data Accesses which are Stores
    uint64_t l1data_num_misses;             // Total L1 Data Misses
    uint64_t l1data_num_misses_loads;       // L1 Data Misses which are Loads
    uint64_t l1data_num_misses_stores;      // L1 Data Misses which are Stores
    uint64_t l1data_num_evictions;          // Total blocks evicted from
        L1-Data cache

    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 l1data_AAT;                      // L1 Data Average Access Time

    // Unified L2 Cache statistics
    uint64_t l2unified_num_accesses;        // Total L2 Accesses
    uint64_t l2unified_num_accesses_loads;  // L2 Accesses that are Loads
    uint64_t l2unified_num_accesses_stores; // L2 Accesses that are Stores
    uint64_t l2unified_num_accesses_insts;  // L2 Accesses that are instructions
    uint64_t l2unified_num_misses;          // Total L2 Misses
    uint64_t l2unified_num_misses_insts;    // L2 Misses that are instructions
    uint64_t l2unified_num_misses_loads;    // L2 Misses that are Loads
    uint64_t l2unified_num_misses_stores;   // L2 Misses that are Stores
    uint64_t l2unified_num_evictions;       // Total blocks evicted from the L2
    uint64_t l2unified_num_write_backs;     // Total write backs from L2 to
        Main Memory
    uint64_t l2unified_num_bytes_transferred; // Total bytes written from L2 to
        Main Memory

    double l2unified_hit_time;              // L2 Hit Time
    double l2unified_miss_penalty;          // L2 Miss Penalty
    double l2unified_miss_rate;             // L2 Miss Rate
    double l2unified_AAT;                   // L2 Average Access Time

    // Performance Statistics
    double inst_avg_access_time;            // Average Access Time per access
        for Instructions
    double data_avg_access_time;            // Average Access Time per access
        for Data (Loads and Stores)
    double avg_access_time;                 // Average Access Time per access -
        A weighed average of instruction and data accesses
};
```

Comments About the Output

- As you will observe, the provided simulator driver prints the output from this structure at the end of a simulation run. This output will be used for all validations (described in detail later). Note that the hit time for the caches depends on various factors such as their size and associativity. The driver function for the simulator will set the hit times for each cache based on tables 1 and 2 for each simulation run.
- The Average Access Time can be computed using the following equation:

$$AAT(\text{Average Access Time}) = HT(\text{Hit Time}) + MR(\text{Miss Rate}) \times MP(\text{Miss Penalty})$$

- The overall AAT can be computed using a weighted average of the total number of instruction and data accesses in the simulation.

Table 1: L1 Access 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

Table 2: L2 Access Time (Cycles)

Config	128K	256K	512K	1M	2M
DM	7	8	8	9	10
$2W$	7	8	9	10	11
$4W$	8	9	10	11	12
$8W$	10	12	13	14	15
FA	15	16	18	20	22

IV Implementation Details

You have been provided with the following files:

- `src/cache.{h/hpp}` : A header file with declaration of functions you will be filling out and the `struct sim_stats_t` definition.
- `src/cache.{c/cpp}` : The file containing the functions you will be writing
- `src/cachesim_driver.{c/cpp}` : The `main` function and driver for the simulation framework.
- `CMakeLists.txt` : A cmake file that will generate a makefile for you.
- `config/default.conf` : A basic simulation configuration file to serve as an example
- `traces/` : A folder containing read-write address traces from real SPEC 2017 programs. Each trace contains 10 Million instructions worth of accesses, so including loads and stores (which typically account for 30% of instructions), there are a total of 13-14 Million cache accesses performed during each simulation. A trace looks like:

```
I 0x55eeafee88f0    // Instruction Access
L 0x7ffd31a002ac    // Load Access
I 0x55eeafee88f4
I 0x55eeafee88f7
S 0x7ffd31a00298    // Store Access
...
```

NOTE: Additional configuration files, a run script to run those test cases and a validation log that you will be matching against will be provided shortly.

i Provided Framework

We 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 will need to fill the following functions found in the `cache.{c/cpp}` file:

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:

```
// Write policies and Replacement policies
enum write_policy {WBWA = 1, WTWNA = 2};
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 l1data;
    struct cache_config_t l1inst;
    struct cache_config_t l2unified;
    enum write_policy wp; // write policy
    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, bool debug, FILE *df);`

   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 `INST`, `LOAD` or `STORE` type which are defined as `char` constants in the `cache.{h/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

We have provided you with a `CMakeLists.txt` file that can be 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

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

- Where applicable, writes to blocks are only performed in the top level cache (L1 cache in this case). This means if an access is a write, the dirty bit is set only for the block in the L1 cache. For example, if there is a write miss in both the L1 cache and the L2 cache, the missed block is first installed in the L2 cache, and then in the L1 cache. However, only the L1 block is marked dirty.
- The tag sizes in the L1 caches, and the L2 cache are not the same. Keep this in mind when performing look ups in multiple levels of the hierarchy on misses.
- When moving blocks between the caches, make sure to move the dirty and valid statuses as well.
- Before you write even a single line of code, we 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!

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.

VI Experiments

You must find the best cache hierarchy configuration for each of the given workloads (traces) under a certain budget. More details will be added to this section shortly.

Keep an eye for announcements detailing the updated experiments section!

VII What to Submit to Canvas

You will submit the following files to Canvas in a compressed tarball (`tar.gz`) named `<last name>_project1.tar.gz`:

- `src/cache.{h/hpp}`
- `src/cache.{c/cpp}`
- `src/cache_driver.{c/cpp}`
- `src/util/jsmn.h`
- `CMakeLists.txt`
- `<last name>_report.pdf`

Make sure you untar and check this tarball to ensure that all the required files are present in the tar before submitting to Canvas!

VIII Grading

You will be evaluated on the following criterion:

+0 :	You don't turn anything in by the deadline
+50 :	You turn in well commented significant code that compiles but does not match the validation
+20 :	Your simulator matches the validation output
+20 :	You ran experiments and found the optimum configuration for each workload (optimum will be defined in section VI)
+5 :	You report is award winning. That means you have justified each optimum graph with graphs, tables and a persuasive argument
+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, we 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>
```