CS {4/6}290 & ECE {4/6}100 - Spring 2019
Project 1 : Cache Simulator

Dr. Thomas Conte
Due: February 16th 2019 @ 11:55 PM

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 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. In this project, you will write a data cache hierarchy simulator that can simulate memory address traces, and run experiments on the given workload traces to find the best cache configurations for each of them. Section III provides the specifications of the cache hierarchy, and section IV provides useful information concerning the simulator.

This lab has to be done individually. Please follow all the rules as per section I.

III Simulator Specifications

i Cache Hierarchy Layout

An L1 cache, a victim cache (VC) and an L2 cache. Detailed specifications are below:

- The simulator should model a cache hierarchy with:

  - 2^c bytes of storage in the L1 Data cache
  - 2^C bytes of storage in the L2 cache
  - 2^s blocks per set in the L1 cache
  - 2^S blocks per set in the L2 cache
  - 2^b bytes per block in both caches
  - A victim cache which can hold V blocks
  - Command line parameters `c, C, s, S, b` and `V` can be used to modify the above values

- Memory addresses are 64-bit long.
- The caches are byte addressable.
- The caches implement the write-back, write-allocate (WBWA) policy.
- Both caches use the least recently used (LRU) block replacement policy.
- All valid bits are set to 0 when the simulation begins.
- The victim cache (which can hold V blocks) has the following properties:
  - Is fully associative
  - Uses the first in first out (FIFO) replacement policy
  - `V` can vary in the range $[0 \ldots 4]$
- There is a prefetcher that prefetches K next blocks on a miss. Command line parameter `K` can vary in the range $[0 \ldots 4]$. More details are provided later in this document.
- Cache accesses are either READ or WRITE type. A READ event loads 1 byte from an address and a WRITE event stores 1 byte to an address.
- In general, (`c, C, s, S, b, V, K`) completely specifies the cache hierarchy.

ii Cache Optimizations

Details of the victim cache, the "Non-inclusive" two level hierarchy, and the prefetcher are below:

1. **The Victim Cache**
   - When the L1 cache misses, first the victim cache is checked. If the block is present in the victim cache, the LRU block of the L1 cache set ("the victim") is replaced with the VC block that was just found. If the block is not present in the VC it is fetched from the L2 cache. The victim block from the L1 cache is put in the VC. Remember that the victim cache uses a FIFO replacement policy.
   - When a block ("the victim") is evicted from the L1 cache, it is written to the VC. Blocks evicted from the VC are written to the main memory if they are clean. If the block evicted from the victim cache is dirty, it is installed in the L2 cache.
2. **Level Two (L2) Cache**
   - The L2 cache is checked when there is an L1 cache miss and a victim cache miss. If the block is found in the L2 cache, it is installed in the L1 cache. This can cause an L1 eviction to the victim cache, and consequently a victim cache eviction to the L2 cache **only if the victim cache "victim" is dirty**. Note, this eviction from the victim cache to the L2 cache can also cause an eviction from the L2 cache.

- If the L2 cache also misses, the missed block is fetched from main memory and installed in first the L2 cache and then in the L1 cache. These installations can cause evictions from both the caches, and should be handled appropriately (Remember: An L1 eviction is installed in the Victim Cache, a Victim Cache eviction, if dirty, is installed in the L2 cache, and any L2 evictions if dirty are written back to memory). If the access is a write, only the top level cache (The L1 cache in this case) is marked dirty.

3. **The Prefetcher**

   - This cache hierarchy also implements a basic `+k` prefetcher. When a block is fetched from main memory for insertion into the L2 cache, `k` blocks with increasing *block addresses* are pre-fetched from main memory and **installed in the L2 cache**. Keep in mind that these pre-fetched insertions can cause evictions from the L2 cache. All prefetched blocks are inserted **in the LRU position**.
   - The first time a prefetched block is accessed, make sure to increment the statistic `num_useful_prefetches`. Prefetched blocks **don't affect any other statistics** in any way.
   - Block_address(X) is the address X where the offset bits are set to zero.
   - Note: When scanning the set for the least recently used block, always start scanning from way-0.

iii Cache Statistics (The Output)

The simulator will output the following statistics after completion of the run:

- `uint64_t num_accesses` : Total number of accesses made to the memory hierarchy
- `uint64_t num_accesses_reads` : Number of read accesses made to the memory hierarchy
- `uint64_t num_accesses_writes` : Number of write accesses made to the memory hierarchy
- `uint64_t num_misses_l1` : Total number of misses in the L1 cache
- `uint64_t num_misses_reads_l1` : Number of read misses in the L1 cache
- `uint64_t num_misses_writes_l1` : Number of write misses in the L1 cache
- `uint64_t num_hits_vc` : Total number of victim cache hits
- `uint64_t num_misses_vc` : Total number of victim cache misses
- `uint64_t num_misses_reads_vc` : Number of read misses in the victim cache
- `uint64_t num_misses_writes_vc` : Number of write misses in the victim cache
- `uint64_t num_misses_l2` : Total number of misses in the L2 cache
- `uint64_t num_misses_reads_l2` : Number of read misses in the L2 cache
- `uint64_t num_misses_writes_l2` : Number of write misses in the L2 cache
- `uint64_t num_write_backs` : Total number of blocks written back to memory
- `uint64_t num_bytes_transferred` : Total number of bytes transferred on the L2 and Memory bus. This includes bytes transferred for writebacks, for prefetches and for any miss repairs to the L2 cache.

- `uint64_t num_prefetches` : Total number of blocks prefetched
- `uint64_t num_useful_prefetches` : Number of useful prefetches. Only count this the first time you hit a prefetched block.
- `double hit_time_l1` : L1 cache hit time = $2.0 + 0.2 \times s$ cycles
- `double hit_time_l2` : L2 cache hit time = $8.0 + 0.4 \times S$ cycles
- `double hit_time_mem` : Memory hit time = 80.0 cycles
- `double miss_rate_l1` : L1 cache miss rate
- `double miss_rate_vc` : Victim cache miss rate
- `double miss_rate_l2` : L2 cache miss rate
- `double avg_access_time` : Average access time of the memory hierarchy

As you will observe, the provided driver will print out the statistics in the desired format. You will only have to fill in the `cache_stats_t` struct passed in to the various functions you will be modifying.

IV Implementation Details

You have been provided with the following files:

- `cache.{h/hpp}` : A header file with declaration of functions you will be filling out and the `struct cache_stats_t` definition.
- `cache.{c/cpp}` : The file containing the functions you will be modifying.
- `cache_driver.{c/cpp}` : The `main` function and driver for the simulator framework.
- `CMakeLists.txt` : A cmake file that will generate a makefile for you.
- `runscript.sh` : A runscript with some basic test cases for cache validation. Note: Your cache may be evaluated against additional test cases that may be released at a later time. **A more detailed runscript will be uploaded to Canvas shortly!**
- `validation.log` : A validation file from the TAs' solution. **This file will be uploaded to Canvas shortly!**
- `traces/` : A folder containing read-write address traces from real programs.

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 in the framework:

```
void cache_init(struct cache_config_t *conf)
```

Initialize your cache hierarchy and any globals that you might need here. `cache_config_t` has the following fields which the driver populates for you:

```
    struct cache_config_t {
        uint64_t c; // 2^c bytes of storage in the L1 cache
        uint64_t C; // 2^C bytes of storage in the L2 cache
        uint64_t s; // 2^s ways in the L1 cache
        uint64_t S; // 2^S ways in the L1 cache
        uint64_t b; // 2^b bytes in cache blocks in both caches
        uint64_t v; // v blocks in the victim cache
        uint64_t k; // prefetch k blocks
    };
```

```
void cache_access(uint64_t addr, char rw, struct cache_stats_t
  *stats)
```

Subroutine for simulating cache events one access at a time. The access type (`rw`) can be either `READ` or `WRITE` type. Update the `stats` struct as necessary.

```
void cache_cleanup(struct cache_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 simulator 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.

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.

In order to use the provided runscript, copy it to the `<Project Directory>/build` folder. The script looks for the `traces` folder in the top level project directory.

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.

- 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 cache, the victim cache and the L2 cache are not the same. Keep this in mind when performing lookups in multiple levels of the hierarchy on misses.

- All prefetched blocks should be installed in the LRU position. One way to achieve this is to mark the timestamp of the prefetched blocks as:

  $(\min(\text{Timestamp of all other blocks in the set}) - 1)$.

- The victim cache is fully associative, and follows the FIFO replacement policy (This means that the oldest timestamp block is evicted first).
- Evicted blocks from the Victim Cache that are dirty are written back to the L2 cache. Dirty blocks evicted from the L2 cache are written back to memory.
- When moving blocks between the various 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.

The exact runscript and validation log for this requirement will be uploaded to Canvas shortly. **Please watchout for an announcement!**

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 tarball.

- `cache.{h/hpp}`
- `cache.{c/cpp}`
- `cache_driver.{c/cpp}`
- `CMakeLists.txt`
- `<last_name>_report.pdf`

You can use the makefile generated by cmake (sub-section: ii of section IV) to generate a tarball using the following command:

```
Unix: $ make submit
```

This will generate a tarball `project1_submit.tar.gz` in the `<Project Directory>/build/` directory. **Make sure you untar and check this tarball to ensure that all the required files are present here 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 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 will be 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 ./cachesim <cachesim arguments
   as you would usually provide them>
```