ENSC 254 Lab Assignment 4

Important Logistics:

- Lab 4 weighs 9% of the final marks. It includes 470 points in total, which will be scaled to 9% of the final marks.
- Lab 4 will be done and graded per 2-student group, as detailed in the lab logistics. Some points, breakdown and guidelines on workload division are provided later in the description.
- Please make sure your code can compile and run correctly on the lab computers (i.e., FAS-RLA Linux computers). If your code cannot compile on the lab computers, then you get 0 mark for lab 4. If your code can compile, but cannot run correctly, then you only get the points where your code runs correctly on the lab computers.

Introduction to Lab 4:

The purpose of this assignment is to enhance your understanding of the cache design and its performance. This includes 1) the organization of a cache (including direct mapped cache, fully associative cache, and set associative cache), 2) the organization of a memory address to index the cache (including tag bits, set index bits, and block byte offset bits), and 3) common cache operations (including cache block identification, placement, and replacement). Specifically, you will develop a simple, configurable single-level cache simulator in lab 4, which will simulate the cache accesses (hits, misses, and evictions) for a given data memory access trace. With this cache simulator, you will be able to explore different cache configurations to choose the best one for your given benchmarks (memory traces in lab 4). Note in lab 4, the memory address width is 64-bits (traces extracted on the real X86 CPU) instead of 32 bits.

Take the challenge and have fun :-)

Framework Code

Major data structures:

In ***cache.h***, we have provided you the following major data structures to simulate a cache.

1) Data structure to represent a cache line (***Line***):

```
typedef struct _line {
    /* Record cache block (i.e., cache line) address for tracing purpose.
     * Block address (block_addr) is the original cache access address
     * whose byte offset bits are cleared to 0.
     * Note in lab 4, we do not record the actual data of a cache block.
     */
    unsigned long long block_addr;
    bool valid;  // valid bit for the cache block
```

```
    unsigned long long tag; // tag bits of the cache block
    /* lru_clock is used to maintain the LRU clock of the cache block.
     * The high lru_clock is, the more recent the cache block is accessed.
     * lru_clock per cache block is updated based on the current global
     * lru_clock inside its corresponding cache set.
     */
    unsigned long long lru_clock;

    /* access_counter is used to implement LFU replacement policy.
     * It tracks how many times the current cache block is accessed.
     */
    int access_counter;
} Line;
```

2) Data structure to represent a cache set (***Set***):

```
typedef struct _set {
    Line *lines;  // List/Array of cache lines/block in the cache set

    /* global lru_clock inside each cache set, which starts from 0.
     * Whenever the cache set is accessed, no matter which cache block
     * inside the set is accessed, it increments the global lru_clock by 1.
     */
    unsigned long long lru_clock;
} Set;
```

3) Data structure to represent a cache (***Cache***):

```
typedef struct _cache {
    // Cache parameters, which users configure from the command line
    //-s <num>: Number of set index bits. i.e., number of cache sets = 2^s
    int setBits;
    //-E <num>: Number of lines per set. i.e., associativity, number of ways
    int linesPerSet;
    //-b <num>: Number of block offset bits. i.e., number of bytes per block = 2^b
    int blockBits;

    // Core cache data structure
    Set *sets; // List/Array of cache sets in the cache

    // Cache stats to count the total number of hits, misses, and evictions
    int hit_count;
    int miss_count;
    int eviction_count;

    // Cache replacement policy
    // 0: Least Recently Used (LRU), 1: Least Frequently Used (LFU)
    int lfu;
```

```
    bool displayTrace; // Used for verbose prints
    char* name; // name for the cache
} Cache;
```

Basically, a ***Cache*** includes a number of cache ***Set***s, where each ***Set*** includes a number of cache ***Line***s (i.e., cache blocks). Through the command line parameters, a user can configure the number of cache sets, the number of cache lines per set, the size (how many bytes) of each cache line, and the cache replacement policy. The ***Cache*** structure also records the total number of cache hits, cache misses, and cache evictions (a special cache miss: there is no empty cache line in the cache set, cache line replacement/eviction needed), which will be printed out at the end after we simulate a trace of memory accesses.

Main function:

The main function in ***main.c*** configures a ***Cache*** with the user configuration parameters, takes in a memory trace file, simulates each memory access, and prints out the summary statistics at the end. Below is a list of command line options supported by the main function.

- -h : help flag that prints usage info
- -s <num>: Number of set index bits. i.e., number of cache sets = 2^s
- -E <num>: Number of lines per set. i.e., associativity, number of ways
- -b <num>: Number of block offset bits. i.e., number of bytes per block = 2^b
- -L : use Least Recently Used (LRU) cache replacement policy.
- -F : use Least Frequently Used (LFU) cache replacement policy.
- -t <file> : Trace file to be processed. It reads and simulates memory trace file line by line.
- -v : verbose mode. Displays what happens to each line of memory trace file.

In the ***main*** function, it calls the ***runTrace*** function to simulate the memory trace in the configured cache. Each iteration, it reads in one line of the memory trace (i.e., current memory address), calls the ***operateCache*** function to simulate the cache access behavior of the current memory address, and returns a ***result***, which could be printed out in the verbose mode.

The *operateCache* function is the entry point of the cache simulator, which you will focus on in lab 4 and will be detailed later. It returns a ***result*** type (in ***cache.h***) object as shown below:

```
typedef struct _result {
    int status;                 // 0: miss 1: hit 2: evict
    unsigned long long victim_block_addr; // block address of the victim line.
    unsigned long long insert_block_addr; // block address of inserted line.
} result;
```

Memory trace file:

The ***traces/input*** folder contains a collection of memory trace files that we will use as inputs to evaluate the correctness of your cache simulator; the golden reference outputs expected by the cache simulator are provided in ***traces/ref*** folder. **The real memory trace files are generated by a Linux program called valgrind on the host 64-bit X86 CPU**: ***each memory address is 64 bits***. For example, typing "***valgrind --log-fd=1 --tool=lackey -v --trace-mem=yes ls -l***" on the command

line, valgrind runs the executable program "***ls -l***", captures a trace of each of its memory accesses in the order they occur, and prints them to stdout.

Memory traces have the following format:

M 0421c7f0,4

L 04f6b868,8

S 7ff0005c8,8

Each line denotes one or two memory accesses. The format of each line is "***operation address,size***":

- The ***operation*** field denotes the type of memory access: L for data load, S for data store, and M for data modify (i.e., a data load followed by a data store, which is supported by X86, but not by RISC-V).
    - **Note:** For the M type of access, the data store in the second half is always a hit, as the data load in the first half already ensures the data is in the cache. Therefore, in the ***operateCache*** function, we increment the hit_count by one for M type of access. ***As a result, when you simulate the M type of access inside the operateCache function, you treat it as if it is a single load access, as we already take care of the second store access.***
    - **Note:** valgrind traces could also include an ***I*** type operation, which is for the instruction access. In lab 4, we focus on data cache simulation and do not consider ***I*** type operation.
- The ***address*** field specifies a ***64-bit hexadecimal*** memory address.
- The ***size*** field specifies the number of bytes accessed by the operation.

Beside the two real memory traces (real1.trace and real2.trace), we also provide five synthetic memory traces files (syn1.trace to syn5.trace). The synthetic trace files are artificially created following the above format, such that you can easily debug your cache simulator to verify that it is working as expected: you can figure out the correct cache behavior using paper and pencil ©

- **Note:** the synthetic traces follow exactly the same format as shown above. For example, the trace "L 18,4" means a load of 4 bytes data starting at 64-bit memory address 0x18.

Your TODO list:

In lab 4, to implement the simulator, you will only need to work on one file: *cache.c.* Please do not change any other files. Specifically, you will implement the following functions. The function prototypes have been given; don't change them. Please change the function bodies, which are indicated by the comment "/* YOUR CODE HERE */". If you want to add extra helper functions, it's ok; but don't change existing function prototypes.

- **operateCache**: This is the ***entry point*** to operate the cache for a given address in the trace file, which is called by the ***runTrace*** function in main.c. All other functions are helper functions.
    - First, is increments the global lru_clock in the corresponding cache set for the address.
    - Second, it checks if the address is already in the cache using the "**probe_cache**" function.
    - If yes, it is a cache hit:

        - call the "**hit_cacheline**" function to update the counters inside the hit cache line, including its lru_clock and access_counter.
        - record a hit status in the return "result" struct and update hit_count
    - Otherwise, it is a cache miss:
        - call the "**insert_cacheline**" function, trying to find an empty cache line in the cache set and insert the address into the empty line.
        - if the "**insert_cacheline**" function returns true, record a miss status and the inserted block address in the return "result" struct and update miss_count
        - otherwise, if the "**insert_cacheline**" function returns false:
            - call the "**victim_cacheline**" function to figure out which victim cache line to replace based on the cache replacement policy (LRU and LFU).
            - call the "**replace_cacheline**" function to replace the victim cache line with the new cache line to insert.
            - record an eviction status, the victim block address, and the inserted block address in the return "result" struct. Update both miss_count and eviction_count.
- **cacheSetUp**: allocate the memory space for the cache with the given cache parameters and initialize the cache sets and lines. Also initialize the cache name to the given name. This function is called in the **main** function.
- **deallocate**: deallocate the memory space for the cache. This function is called in the **main** function.
- **address_to_block**: helper function. Given an address, return the block (aligned) address, i.e., byte offset bits are cleared to 0.
- **cache_tag**: helper function. Return the tag of an address.
- **cache_set**: helper function. Return the cache set index of the address.
- **probe_cache**: check if the address is found in the cache. If so, return true; otherwise, return false. This function is called in the **operateCache** function.
- **hit_cacheline**: access address in cache and update the LRU and LFU counters. This function is called in the **operateCache** function if **probe_cache** returns true.
- **insert_cacheline**: This function is called in the **operateCache** function if **probe_cache** returns false. It will try to find an empty (i.e., invalid) cache line for the address to insert. If it cannot find an empty one, it returns false. Otherwise, if it can find an empty one:
    - it inserts the address into that cache line (marking it valid).
    - it updates the cache line's lru_clock based on the global lru_clock in the cache set and initiates the cache line's access_counter.
    - it returns true.
- **victim_cacheline**: If there is no empty cacheline, this function figures out which cacheline to replace depending on the cache replacement policy (LRU and LFU). It returns the block address of the victim cacheline; note we no longer have access to the full address of the victim. This function is called in the **operateCache** function.
    - For LRU, we select the cache line that is least recently accessed (i.e., with the smallest lru_clock) as the victim.
    - For LFU, we select the cache line that is least frequently accessed (i.e., with the smallest access_counter) as the victim. When there is a tie, i.e., both cache lines have the smallest access_counter, we choose the one with the smallest lru_clock as the victim.

- **replace_cacheline**: Replace the victim cacheline with the new address to insert. Note for the victim cachline, we only have its block address. For the new address to be inserted, we have its full address. Remember to update the new cache line's lru_clock based on the global lru_clock in the cache set and initiate the cache line's access_counter. This function is called in the **operateCache** function.

Cache design exploration: Once you finish implementing the cache simulator, you have the right tool to explore different cache design configurations for a given set of benchmarks, i.e., two real memory traces in lab 4, and choose the best cache configuration that achieves the highest cache hit rate. We will give more details in the following sections.

Basic Commands and Testing:

- To build your cache simulator, type the following command on a lab computer (all commands are typed when you are inside the lab4 folder):

  make

- To run your simulator with a test input, type the following command:

  ./cache -v -s 1 -E 1 -b 1 -L -t traces/input/syn1.trace

  The command line options are explained earlier. This simulates a cache configuration with 2 sets, direct mapped, block size = 2 Bytes, and LRU replacement policy. The input trace is *traces/input/syn1.trace* and it will verbosely print out each line's access information.

  To redirect this output into an output file ("*traces/out/syn1_1L.out*"), instead of printing it onto the screen, type the following command:

  ./cache -v -s 1 -E 1 -b 1 -L -t traces/input/syn1.trace > traces/out/syn1_1L.out

- To validate the correctness of the above test, type the following command:

  diff ./traces/out/syn1_1L.out ./traces/ref/syn1_1L.ref

  For every input file, we have also provided the golden reference file (.ref files in the *traces/ref* folder) to compare against. Where there is a mismatch, you can use gdb or cgdb to help the debugging and code fix.

More Testing and Grading:

- **[226 points] Test 1: test each synthetic memory trace and cache configuration for both LRU and LFU replacement polices in verbose mode. Matching one line of the golden reference output file gives you 1 point for lab 4.**
  - [34 points] Cache config: 2 sets, direct mapped, block size = 2 Bytes, LRU vs LFU. Input trace: *syn1.trace.*

    To perform the LRU test (17 points), type the following commands:

    ./cache -v -s 1 -E 1 -b 1 -L -t traces/input/syn1.trace > traces/out/syn1_1L.out
    diff ./traces/out/syn1_1L.out ./traces/ref/syn1_1L.ref

To perform the LFU test (17 points), type the following commands:

./cache -v -s 1 -E 1 -b 1 -F -t traces/input/syn1.trace > traces/out/syn1_1F.out
diff ./traces/out/syn1_1F.out ./traces/ref/syn1_1F.ref

Below are 6 more synthetic testcases (points are evenly divided for LRU and LFU tests), following a similar naming convention and the commands are also similar. A complete set of commands is provided in ***run-cache-trace.sh***.

- [34 points] Cache config: 1 set (i.e., fully associative), 2-way set associative, block size = 2 Bytes, LRU vs LFU. Input trace: *syn1.trace.*
- [22 points] Cache config: 2 sets, 2-way set associative, block size = 2 Bytes, LRU vs LFU. Input trace: *syn2.trace.*
- [30 points] Cache config: 2 sets, 4-way set associative, block size = 4 Bytes, LRU vs LFU. Input trace: *syn3.trace.*
- [58 points] Cache config: 2 sets, 4-way set associative, block size = 4 Bytes, LRU vs LFU. Input trace: *syn4.trace.*
- [48 points] Cache config: 1 set (i.e., fully associative), 8-way set associative, block size = 8 Bytes, LRU vs LFU. Input trace: *syn5.trace.*

- **[180 points] Test 2: test each real memory trace and cache configuration for both LRU and LFU replacement polices in concise mode. Matching total number of hits give you 15 points; matching total number of misses give you 15 points; matching total number of evictions give you 15 points.**
  - [90 points] Cache config (64KB): 256 sets, 4-way set associative, block size = 64 Bytes, LRU vs LFU. Input trace: *real1.trace.*

    To perform the LRU test (45 points), type the following commands:

    ./cache -s 8 -E 4 -b 6 -L -t ./traces/input/real1.trace > traces/out/real1_L.out
    diff ./traces/out/real1_L.out ./traces/ref/real1_L.ref

    To perform the LFU test (45 points), type the following commands:

    ./cache -s 8 -E 4 -b 6 -F -t ./traces/input/real1.trace > traces/out/real1_F.out
    diff ./traces/out/real1_F.out ./traces/ref/real1_F.ref
  - [90 points] Cache config (32KB): 128 sets, 8-way set associative, block size = 32 Bytes, LRU vs LFU. Input trace: *real2.trace.*

    The commands are similar, and a complete set of commands is provided in ***run-cache-trace.sh***.
- **[64 points] Cache design exploration**: With your developed cache simulator, now you can explore different cache design configurations for a given set of benchmarks, i.e., two real memory traces in lab 4. Let's assume your cache block size is 64 bytes (i.e., b=6, fixed, do not change this), and your total cache size is ***CSIZE*** (note we use conventional cache size, which includes the actual data part only and excludes cache tag and status bits). You have the freedom to explore different cache configurations, including the number of cache sets (2^s), the cache associativity (E, assuming E is a power-of-two number), and cache

replacement policy (LRU vs LFU). Please explore and report the best cache configuration that achieves the highest cache hit rate.

- [30 points] Assume ***CSIZE*** = 2 KB. Explore all possible cache configurations, report the cache hit rate of each cache configuration for each real memory trace in a table and a line chart. And report your final choice of the cache configuration that achieves the highest cache hit rate.
  - [12 points] Report (in .pdf format) for cache hit rate of each cache configuration for *real1.trace*. Report this in both a table and a line chart, explain your results, and report your final choice.
  - [12 points] Report (in .pdf format) for cache hit rate of each cache configuration for *real2.trace*. Report this in both a table and a line chart, explain your results, and report your final choice.
  - [6 points] Make your final choice of one cache configuration, that achieves the highest cache hit rate for both traces. Report this in a ***dse_2k.txt*** file and following this format:
    - In line 1, type ***-s YOUR_SET_BITS***, e.g., ***-s 3***;
    - In line 2, type ***-E YOUR_ASSOCIATIVITY***, e.g., ***-E 4***;
    - In line 3, type ***-L*** for LRU or ***-F*** for LFU;
    - **Do not include anything else in this dse_2k.txt; violating this will get 0 point for this part.**
- [34 points] Assume ***CSIZE*** = 4 KB. Explore all possible cache configurations, report the cache hit rate of each cache configuration for each real memory trace in a table and a line chart. And report your final choice of the cache configuration that achieves the highest cache hit rate.
  - [14 points] Report (in .pdf format) for cache hit rate of each cache configuration for *real1.trace*. Report this in both a table and a line chart, explain your results, and report your final choice.
  - [14 points] Report (in .pdf format) for cache hit rate of each cache configuration for *real2.trace*. Report this in both a table and a line chart, explain your results, and report your final choice.
  - [6 points] Make your final choice of one cache configuration, that achieves the highest cache hit rate for both traces. Report this in a ***dse_4k.txt*** file and following the same format as specified above for ***CSIZE*** = 2 KB.
- Summary of extra report files needed to be included in your final .zip file: ***dse_report.pdf*** (with 4 tables and 4 line charts, explanations, and final choices), ***dse_2k.txt*** (final choice for CSIZE = 2 KB), ***dse_4k.txt*** (final choice for CSIZE = 4 KB).

- **Guidance for workload distribution**. Below is an example distribution, you may use a different distribution. **Please document your workload distribution in the given *WorkDistForGrading.csv* file inside the lab4 folder. In the TA tutorial, TA will show you how to fill in this .csv file. Please strictly follow the format to make it easy for the auto-grading; otherwise, you will have 10% mark deduction penalty.**
  - **Student 1 is responsible for all LRU tests and reports, which count for 235 points**.
  - **Student 2 is responsible for all LFU tests and reports, which count for 235 points**.

Assignment Submission:

Your lab assignment 4 will be submitted electronically through Canvas. You will need to submit a single ***YOUR_LAB_GROUP.zip*** file (e.g., LA01_01.zip). This format makes it easy for TA to do auto-grading. Please double check the format correctness. **If you fail to comply with this format, you will lose 10% of the total points of this lab**. To zip your files in Linux,

1. *Go to your lab4 folder*
2. *make deepclean //**make sure you clean up your files***
3. *cd .. //go one level up*
4. *mv lab4 **YOUR_LAB_GROUP** //rename your lab4 folder using your lab group (e.g., LA01_01)*
5. *zip -r YOUR_LAB_GROUP.zip YOUR_LAB_GROUP //zip all your lab4 files into a single .zip*

Submission Deadline:

Your lab assignment 4 is due at **11:59:59pm on Friday, Jun 28th, 2024**. You need to meet the deadline: every 10 minutes late for submission, you lose 10% of the points; that is, 100 minutes late, you will get zero for this lab.

Lab Demonstration:

To finalize your lab 4 points, you will have to demo your lab code to your TA in the following lab sessions that you enrolled on **Wednesday (Jul 3rd, 2024) and Thursday (Jul 4th, 2024).** Due to the large enrollment, TAs will **randomly** select some students (about 1/2 student groups) to do the lab 4 demo. If you are NOT selected to do the demo, the auto-graded points are your final points.

For those who are selected to do the demo, each student group has around 10 minutes to explain their code to the TA. Only code from your Canvas submission is allowed in the lab demo. If you fail to do the demo (without a medical note), you will be awarded zero on this lab assignment. **If it is determined that you do not understand your code being evaluated (the part you are responsible for based on your workload distribution), you will be awarded zero and considered as CHEATING on this lab assignment.** Also please show up in the demo day on time (TA will send out your scheduled time), otherwise you will lose 10% of the lab 4 points.