CS 449 Fall 2020
Cache Lab: Understanding Cache Memories

1 Overview

This lab will help you understand how cache memories work. For this purpose, you will write a small C program that simulates the behavior of a cache memory.

2 Logistics

This is an *individual* project. All submissions are electronic using an autograder service. Before you begin, please take the time to review the course policy on academic integrity at the course web site.

You must run this lab on a 64-bit x86-64 machine. All submissions are electronic. We strongly advise you to test your code on the Thoth machine before submitting it. Always test your submission on Gradescope in advance. We will not return any points lost because you did the lab in a different environment and have not tested it on Gradescope.

Start early to get it done before the due date. Assume things will not go according to plan, and so you must allow extra time for heavily loaded systems, dropped Internet connections, corrupted files, traffic delays, minor health problems, etc.

3 Downloading the assignment

Your lab materials are contained in an archive file called `cachelab-handout.zip`, which you can download to Thoth as follows.

```
$ wget https://bit.ly/39bePaT -O cachelab-handout.zip
```

Start by copying `cachelab-handout.zip` to a protected directory in which you plan to do your work. Then give the command below

```
$ unzip cachelab-handout.zip
```

This will create a directory called `cachelab-handout` that contains a number of files.

WARNING: Do not let the Windows WinZip program open up your `.zip` file (many Web browsers are set to do this automatically). Instead, save the file to a private directory on Thoth and use the Linux `unzip` program to extract the files. In general, for this class you should NEVER use any platform other than Linux to modify your files. Doing so can cause loss of data (and important work!).

4 Input Trace Files

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

```
$ 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
```

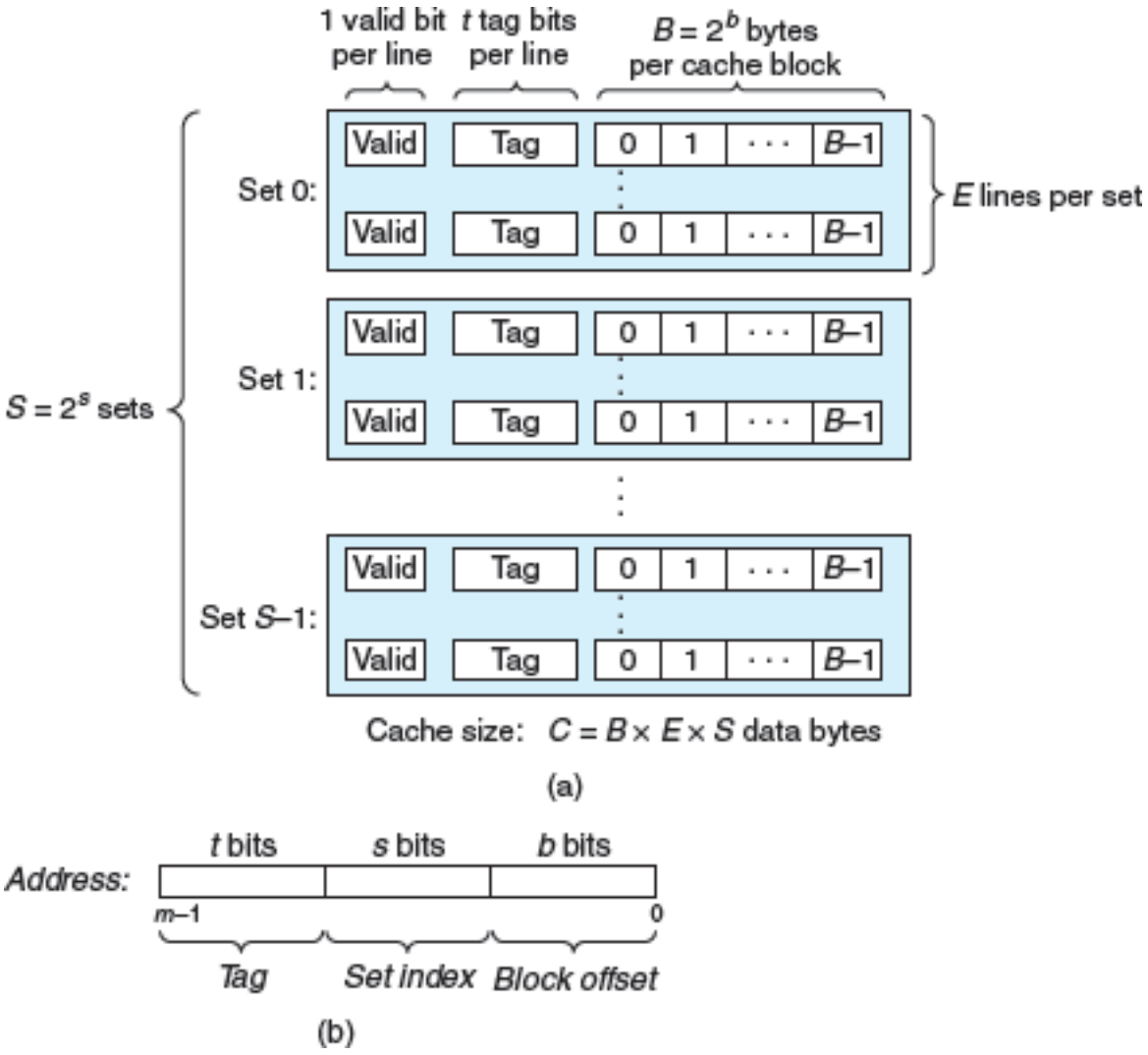
The *operation* field denotes the type of memory access: "I" denotes an instruction load, "L" a data load, "S" a data store, and "M" a data modify (i.e., a data load followed by a data store). There is never a space before each "I". There is always a space before each "M", "L", and "S". The *address*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

5 Programming Task

You will write a cache simulator in the provided file `csim.c` 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 cache is organized so that it can find the requested word by simply inspecting the bits of the address, similar to a hash table with an extremely simple hash function.

Figure 1: General organization of cache (S, E, B, m). (a) A cache is an array of sets. Each set contains one or more lines. Each line contains a valid bit, some tag bits, and a block of data, (b) The cache organization induces a partition of the `m` address bits into `t` tag bits, `s` set index bits, and `b` block offset bits.



The simulation follows the cache organization and notation (s, E, and b) as described in the CS:APP3e textbook (see Figure 1 for the illustration taken from the textbook [Page 616]). The `s` set index bits in the address form an index into the array of `S` sets. The first set is set 0, the second set is set 1, and so on. When interpreted as an unsigned integer, the set index bits tell us which set the word must be stored in. Once we know which set the word must be contained in, the `t` tag bits in the address tell us which line (if any) in the set contains the word. A line in the set contains the word if and only if the valid bit is set and the tag bits in the line match the tag bits in the address A. Once we have located the line identified by the tag in the set identified by the set index, then the `b` block offset bits give us the offset of the word in the B-byte data block.

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

For example:

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

The same example in verbose mode:

```
$ ./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 task for this project is to fill in the `csim.c` file so that it takes the same command line arguments and produces identical output to the reference simulator. This file already has some code, but you need to figure out what data structures you need and how to use them! (Look again at Figure 1)

As you read L / S / M accesses from the input trace, you will need (at least) to:

- split the memory address;
- use the "set" portion to select the correct set (in some data structure);
- use the "tag" portion to search for a line (in some data structure);
- update counters/metadata to keep track of which line is the least recently accessed (for LRU policy).

In your implementation, you must adhere to the following rules.

- Your `csim.c` 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 dynamically allocate storage for your simulator's data structures using the `malloc` function.
- For this lab, 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 lab, 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.

Important: 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 (even when only 1 byte is modified).

6 Evaluation

This section describes how your work will be evaluated. The full score for this lab is 21 points. We will run your cache simulator using different cache parameters and traces. There are seven test cases, each worth 3 points:

```
$ ./csim -s 1 -E 1 -b 1 -t traces/yi2.trace
$ ./csim -s 4 -E 2 -b 4 -t traces/yi.trace
$ ./csim -s 2 -E 1 -b 4 -t traces/dave.trace
$ ./csim -s 2 -E 1 -b 3 -t traces/trans.trace
$ ./csim -s 2 -E 2 -b 3 -t traces/trans.trace
$ ./csim -s 2 -E 4 -b 3 -t traces/trans.trace
$ ./csim -s 5 -E 1 -b 5 -t traces/trans.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 give you full credit for that test case. Each of your reported number of hits, misses and evictions is worth 1/3 of the credit

for that test case. That is, if a particular test case is worth 3 points, and your simulator outputs the correct number of hits and misses, but reports the wrong number of evictions, then you will earn 2 points.

We have provided you with an autograding 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:

```
$ make
$ ./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
    21
```

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

7 Hints

Here are some hints and suggestions for working on this assignment:

- 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.c` 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.
- You are allowed to search the Internet for the documentation of the following C library functions which will be helpful: `getopt` (to parse command-line arguments), `fopen` and `fclose` (to open/-close files), `fgets`, `fscanf`, or `getline` (to read from files), `sscanf` (to parse fields within a string).
- Note that `sscanf` requires the format string to contain the expected format. For example, if you wanted to parse two int separated by a comma and a space you would have to use a format string like: `"%d, %d"` that contains the expected placeholders at the expected locations.
- Use `gdb` to debug your code, for instance, to help you figure out which line of code is causing a segmentation fault. See `http://www.unknownroad.com/rtfm/gdbtut/gdbsegfault.html`.

8 Submitting Your Work

You need to submit your `csim.c` file using Gradescope. You may resubmit your work until the deadline, with your most recent submission counting for credit.

9 Optional Extra Credit

IMPORTANT NOTICE: Do NOT spend time on this part until you have finished and turned in your cache simulator assignment. You won't receive any credit for this part if you haven't successfully completed your simulator assignment.

In this extra credit portion, Chip D. Signer, Ph.D., is trying to reverse engineer a competitor's microprocessors to discover their cache geometries and has recruited you to help. Instead of running programs on these processors and inferring the cache layout from timing results, you will approximate his work by using the cache simulator.

Your extra lab materials are contained in an archive file called `cachelab-mystery.zip`, which you can download to Thoth as follows.

```
$ wget https://bit.ly/37q9z27 -O cachelab-mystery.zip
```

Start by copying `cachelab-mystery.zip` to a protected Linux directory in which you plan to do your work. Then give the command below

```
$ unzip cachelab-mystery.zip
```

This will create a directory called `cachelab-mystery` that contains a number of files. Below are the important files:

- **cache-test-skel.c** – Skeleton code for determining cache parameters (to be modified)
- **test-cache.sh** – Script that makes and runs cache-test with all the cache object files
- **caches/cache_*.o** – "Caches" with known parameters for testing your code
- **support/mystery-cache.h** – Defines the function interface that the object files export

You will write some C code to identify the cache parameters of some given mystery "caches". Each of the "processors" is provided as an object file (.o file) against which you will link your code. The file `mystery-cache.h`, shown below, defines the function interface that these object files export.

```
typedef unsigned long long addr_t;
typedef unsigned char bool_t;
#define TRUE 1
```

```
#define FALSE 0

/* Lookup an address in the cache. Returns TRUE if the access hits,
    FALSE if it misses. */
bool_t access_cache(addr_t address);


/* Clears all words in the cache (and the victim buffer, if
    present). Useful for helping you reason about the cache
    transitions, by starting from a known state. */
void flush_cache(void);
```

It includes a typedef for a type `addr_t` (an unsigned 8-byte integer) which is what these (pretend) caches use for "addresses", or you can use any convenient integer type.

Your job is to fill in the function stubs in `cache-test-skel.c` which, when linked with one of these cache object files, will determine and then output the cache size, associativity, and block size. You will use the functions above to perform cache accesses and use your observations of which ones hit and miss to determine the parameters of the caches. In particular, you will complete the 3 functions in `cache-test-skel.c` that have `/* YOUR CODE GOES HERE */` comments in them.

IMPORTANT: You should NOT be calling any functions other than `flush_cache` and `access_cache` inside of your functions. For example, you cannot call function `get_block_size()` say inside of `get_cache_assoc`.

Some of the provided object files are named with this information (e.g. `cache_65536c_2e_16k.o` is a 65536 **Byte c**apacity, 2-way set-associativ**e** cache with 16 **Byte** bloc**k**s) to help you check your work.

We have provided you with a Makefile that includes a target cache-test. To use it, set `TEST_CACHE` to the object file to link against on the command line. That is, from within the working directory run the command:

```
$ make cache-test TEST_CACHE=caches/cache_65536c_2e_16k.o
```

This will create an executable **cache-test** that will run your cache-inference code against the supplied cache object. You can also run the following shell script to test all the cache objects provided:

```
$ sh test-cache.sh
```

To receive extra credit, you need to submit your modified `cache-test-skel.c` file using Gradescope. In the electronic evaluation, there will be 4 mystery cache object files, whose parameters you must discover on your own. You can assume that the mystery caches have sizes that are powers of 2 and use a least recently used replacement policy. You cannot assume anything else about the cache parameters except what you can infer from the cache size. Finally, the mystery caches are all pretty realistic in their geometries, so use this fact to sanity check your results.