15-213/18-213/15-513/18-613, Spring 2020
Cache Lab: Understanding Cache Memories
Assigned: Tue. February 18, 2020
Due: Thu. February 27, 11:00PM
Last Possible Time to Hand in: Sun. March 1, 11:00PM

1 Logistics

This is an individual project. All handins are electronic.

You must do this lab on a class shark machine.

To get your lab materials, click “Download Handout" on autolab, enter your Andrew ID, and follow the instructions. You will receive an email to the repository within a few minutes. It will be located at

`https://github.com/cmu15213s20/cachelab-s20-`*`yourgithubid`*

2 Overview

This lab will help you understand the impact that cache memories can have on the performance of your C programs.

The lab consists of three parts.

You will first write several traces to test the behavior of a cache simulator. Next, you will write a small C program (about 200-300 lines) that simulates the behavior of a hardware cache memory. Finally, you will optimize a small matrix transpose function, with the goal of minimizing the number of cache misses.

3 Downloading the assignment

Your lab materials are available at the GitHub link above. You can acquire it by accepting the invitation, then following the same cloning procedure as datalab. You will be working on two files: `csim.c` and `trans.c`. To compile these files, type:

```
linux> make clean
linux> make
```

Note that the file `csim.c` doesn't exist in your initial handout directory. You'll need to create it from scratch before running `make`.

4 Evaluation

This section describes how your work will be evaluated. The maximum score for this lab is 112 points:

Traces: 10 Points

Part A: Cache Simulator 60 Points

Part B: Matrix Transpose 30 Points

Style: 12 Points

Note: If you did not submit traces by its deadline, you no longer have an opportunity to earn those points.

Evaluation for Style

There are 12 points for coding style, for just your simulator. These will be assigned manually by the course staff. Style guidelines can be found on the course website. Of these, 2 points are dedicated to the accuracy and verbosity of your git commit history.

5 Part A: Writing a Cache Simulator

5.1 Description

In Part A you will write a cache simulator in `csim.c`. Your simulator should take a memory trace as input and simulate the hit/miss/evict behavior of a cache memory on this trace. At the end of the simulation it should output the total number of hits, misses, evictions, as well as the number of dirty bytes that have been evicted and the number of dirty bytes in the cache at the end of the simulation.

As a reminder, a dirty bit is a bit associated with a cache block that is set in the case where the payload of that block has been modified, but not yet written back to main memory. A dirty byte is any payload byte whose corresponding cache block's dirty bit is set.

We have provided you with the binary executable of a *reference cache simulator*, called `csim-ref`. It simulates the behavior of a cache with arbitrary size and associativity on a trace file. It uses the LRU (least-recently used) replacement policy when choosing which cache line to evict, and follows a write-back, write-allocate policy.

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 memory trace to replay

The command-line arguments are based on the notation (s, E, and b) from page 617 of the CS:APP3e textbook. For example:

```
linux> ./csim-ref -s 4 -E 1 -b 4 -t traces/yi.trace
hits:4 misses:5 evictions:3 dirty_bytes_in_cache:32 dirty_bytes_evicted:16
```

The same example in verbose mode:

```
linux> ./csim-ref -v -s 4 -E 1 -b 4 -t traces/yi.trace
L 10,1 miss
L 20,1 miss
S 20,1 hit
L 22,1 hit
S 18,1 hit
L 110,1 miss eviction
L 210,1 miss eviction
L 12,1 miss eviction
S 12,1 hit
hits:4 misses:5 evictions:3 dirty_bytes_in_cache:32 dirty_bytes_evicted:16
```

Your job for Part A is write a simulator in `csim.c` so that it takes the same command line arguments and produces the identical output as the reference simulator (except your verbose output may differ). Notice that `csim.c` doesn't exist in your initial handout directory. You'll need to write it from scratch.

5.2 Programming Rules for Part A

- Include your name and Andrew ID in the header comment for `csim.c`.
- Your `csim.c` file must compile without warnings in order to receive credit.
- All of your code must be in `csim.c` and it must compile with the provided make file.

- Your simulator must work correctly for arbitrary s, E, and b. This means that you will need to allocate storage for your simulator's data structures using the `malloc` or `calloc` function. Type "`man malloc`" for information about these functions.
- To receive credit for Part A, you must call the function `printSummary`, with the total number of hits, misses, and evictions, at the end of your `main` function:

```
printSummary(long hit_count, long miss_count, long eviction_count,
long dirty_bytes_in_cache_count, long dirty_bytes_evicted_count);
```

- For this lab, you can 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 memory traces.
- Your cache implementation must be able to handle caches of arbitrary size. In particular, you may not use stack frames to allocate your cache, since this will prevent large-sized caches from being created. While your code will not be required to produce the correct output on unreasonably large values, your simulator should not crash in any case.

5.3 Evaluation for Part A

For Part A, we will run your cache simulator using different cache parameters and traces. Note that while we show the following parameters in the same order, we will vary their order in testing. There are ten test cases, each worth 5 points, except for the last case, which is worth 10 points:

```
linux> ./csim -s 0 -E 1 -b 0 -t traces/wide.trace
linux> ./csim -s 2 -E 1 -b 2 -t traces/wide.trace
linux> ./csim -s 3 -E 2 -b 2 -t traces/load.trace
linux> ./csim -s 1 -E 1 -b 1 -t traces/yi2.trace
linux> ./csim -s 4 -E 2 -b 4 -t traces/yi.trace
linux> ./csim -s 2 -E 1 -b 4 -t traces/dave.trace
linux> ./csim -s 2 -E 1 -b 3 -t traces/trans.trace
linux> ./csim -s 2 -E 2 -b 3 -t traces/trans.trace
linux> ./csim -s 14 -E 1024 -b 3 -t traces/trans.trace
linux> ./csim -s 5 -E 1 -b 5 -t traces/trans.trace
linux> ./csim -s 5 -E 1 -b 5 -t traces/long.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, computing 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, evictions, dirty bytes evicted, and dirty bytes in cache is worth 1/5 of the credit for that test case. That is, if a particular test case is worth 5 points, and your simulator outputs the correct number of hits and misses, but reports the wrong number of evictions, dirty bytes evicted, and dirty bytes in cache, then you will earn 2 points.

5.4 Working on Part A

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:

```
linux> make
linux> ./test-csim
                        Your simulator                          Reference simulator
Points ( s,   E,b)    Hits  Misses  Evicts D_Cache D_Evict    Hits  Misses  Evicts D_Cache D_Evict
     5 ( 0,   1,0)       1      18      17       1       6       1      18      17       1       6  traces/wide.trace
     5 ( 2,   1,2)       3      16      12       4      20       3      16      12       4      20  traces/wide.trace
     5 ( 3,   2,2)       6       3       0       0       0       6       3       0       0       0  traces/load.trace
     5 ( 1,   1,1)       9       8       6       4       8       9       8       6       4       8  traces/yi2.trace
     5 ( 4,   2,4)       4       5       2      32      16       4       5       2      32      16  traces/yi.trace
     5 ( 2,   1,4)       2       3       1      32      16       2       3       1      32      16  traces/dave.trace
     5 ( 2,   1,3)     167      71      67       8     264     167      71      67       8     264  traces/trans.trace
     5 ( 2,   2,3)     201      37      29      32     152     201      37      29      32     152  traces/trans.trace
     5 (14,1024,3)     215      23       0     120       0     215      23       0     120       0  traces/trans.trace
     5 ( 5,   1,5)     231       7       0     160       0     231       7       0     160       0  traces/trans.trace
    10 ( 5,   1,5)  265189   21777   21745      96  556608  265189   21777   21745      96  556608  traces/long.trace
    60
```

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.

Here are some hints and suggestions for working on Part A:

- To get started, you'll need to create a file called `csim.c` with an empty `main` routine. If you try running `make` before creating `csim.c`, it will fail with the message:

  ```
  make: *** No rule to make target 'csim.c', needed by 'csim'.  Stop.
  ```

- The `printSummary` function is implemented in `cachelab.c`. In order to call it from `csim.c`, you will need to include the header file called `cachelab.h`:

  ```
  #include "cachelab.h"
  ```

- After calling `printSummary`, the `main` routine in `csim.c` should return a status of zero. If you return with a non-zero status value, the autograders will assume that there was an error.

- 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 behavior of the reference simulator on the reference trace files.

- We recommend that you use the `getopt` function to parse your command line arguments. You'll need the following header files:

  ```
  #include <stdlib.h>
  #include <unistd.h>
  #include <getopt.h>
  ```

  See "`man 3 getopt`" for details.

- Each data load (`L`) or store (`S`) operation can cause at most one cache miss.
- If you would like to use C0-style contracts from 15-122, you can include `contracts.h`, which we have provided in the handout directory for your convenience.
- Do not forget that the addresses in the trace are 64-bit hexadecimal memory addresses.

6 Part B: Optimizing Matrix Transpose

6.1 Description

In Part B you will write a transpose function in `trans.c` that uses as few clock cycles as possible, where the number of clock cycles is computed artificially using a cache simulator. The clock cycle computation captures the property that cache misses require significantly more clock cycles (100) than cache hits (4).

Let A denote a matrix, and $a_{i,j}$ denote the component at row i and column j. The *transpose* of A, denoted A^T, is a matrix such that $a^T_{i,j} = a_{j,i}$.

To help you get started, we have given you several example transpose functions in `trans.c` that compute the transpose of $N \times M$ matrix A and store the results in $M \times N$ matrix B. An example of one such function is:

```
char trans_desc[] = "Simple row-wise scan transpose";
void trans(size_t M, size_t N, double A[N][M], double B[M][N], double *tmp);
```

Argument `tmp` is a pointer to an array of 256 elements that can be used to hold data as an intermediate step between reading from `A` and writing to `B`.

The example transpose functions are correct, but they have poor performance, because the access patterns result in many cache misses, resulting in a high number of clock cycles.

Your job in Part B is to write a similar function, called `transpose_submit`, that minimizes the number of clock cycles across different sized matrices:

```
char transpose_submit_desc[] = "Transpose submission";
void transpose_submit(size_t M, size_t N, double A[N][M], double B[M][N],
                      double *tmp);
```

Do *not* change the description string ("`Transpose submission`") for your `transpose_submit` function. The autograder searches for this string to determine which transpose function to evaluate for credit.

6.2 Programming Rules for Part B

- Include your name and Andrew ID in the header comment for `trans.c`.
- Your code in `trans.c` must compile without warnings to receive credit.

- All of your code must be in `trans.c`, and it must compile with the provided make file.
- You may not make out-of-bounds references to any array.
- You may use helper functions. Indeed, you will find this a useful way to structure your code.
- Your transpose function may not modify array `A`. You may, however, read and/or write the contents of `B` and `tmp` as many times as you like.
- You may not store any array data outside of `A`, `B`, and `tmp`. This includes in any local variables, structs, or arrays in your code.
- You may use recursion.
- You are NOT allowed to use any variant of `malloc`.
- Since our style guidelines prohibit the use of "magic numbers," you should refer to the maximum number of elements in `tmp` with the compile-time constant `TMPCOUNT`.
- These restrictions apply to *all* functions in your `trans.c` file, not just those that are called as part of the official submission.
- You may customize your functions to use different approaches depending on the values of M and N. Indeed, you will find this necessary to achieve the required performance objectives.

Although the compiler is configured to detect some violations of these guidelines automatically, your programs will also be manually checked for compliance at the same time style points are assigned. If they are violated, you will receive no credit for Part B. Make sure you understand and follow the rules!

6.3 Evaluation for Part B

For Part B, we will evaluate the correctness of your `transpose_submit` function on ten different matrices. You will receive no credit for part B if your code gives incorrect results for any of these.

We will evaluate its performance on two different-sized matrices:

- 32×32 ($M = 32$, $N = 32$)
- 63×65 ($M = 63$, $N = 65$)

6.3.1 Performance (30 pts)

For each matrix size, the performance of your `transpose_submit` function is evaluated by using `LLVM`-based instrumentation to extract the address trace for your function, and then using the reference simulator to replay this trace on a cache with parameters ($s = 5$, $E = 1$, $b = 6$).

Using the reference cache simulator, each transpose function will be assigned some number of clock cycles m. A cache miss is worth 100 clock cycles, while a cache hit is worth 4. Your performance score for each matrix size will scale linearly with m, up to some threshold. The scores are computed as:

- 32×32: 20 points if $m < 36{,}000$, 0 points if $m > 45{,}000$
- 63×65: 10 points if $m < 280{,}000$, 0 points if $m > 350{,}000$

For example, a solution for the 32×32 matrix with 1764 hits and 284 misses ($m = 1764 \cdot 4 + 284 \cdot 100 = 35456$) would receive 20 of the possible 20 points.

You can optimize your code specifically for the two cases in the performance evaluation. In particular, it is perfectly OK for your function to explicitly check for the matrix sizes and implement separate code optimized for each case.

6.4 Working on Part B

We have provided you with an autograding program, called `test-trans.c`, that tests the correctness and performance of each of the transpose functions that you have registered with the autograder.

You can register up to 100 versions of the transpose function in your `trans.c` file. Each transpose version has the following form:

```
/* Header comment */
char trans_simple_desc[] = "A simple transpose";
void trans_simple(size_t M, size_t N, double A[N][M], double B[M][N],
                  double *tmp)
{
    /* your transpose code here */
}
```

Register a particular transpose function with the autograder by making a call of the form:

```
registerTransFunction(trans_simple, trans_simple_desc);
```

in the `registerFunctions` routine in `trans.c`. At runtime, the autograder will evaluate each registered transpose function and print the results. Of course, one of the registered functions must be the `transpose_submit` function that you are submitting for credit:

```
registerTransFunction(transpose_submit, transpose_submit_desc);
```

See the default `trans.c` function for an example of how this works.

The autograder takes the matrix size as input. It uses `LLVM` to generate a trace of each registered transpose function. It then evaluates each trace by running the reference simulator on a cache with parameters ($s = 5$, $E = 1$, $b = 6$).

For example, to test your registered transpose functions on a 32×32 matrix, rebuild `test-trans`, and then run it with the appropriate values for M and N:

```
linux> make
linux> ./test-trans -M 32 -N 32
Function 0 (2 total)
```

```
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=6)
func 0 (Transpose submission): hits:868, misses:1180, evictions:1148,
clock_cycles:121472

Function 1 (2 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=6)
func 1 (Row-wise scan transpose): hits:868, misses:1180, evictions:1148,
clock_cycles:121472

Summary for official submission (func 0): correctness=1 cycles=121472
```

In this example, we have registered two different transpose functions in `trans.c`. The `test-trans` program tests each of the registered functions, displays the results for each, and extracts the results for the official submission.

Here are some hints and suggestions for working on Part B.

- The `test-trans` program saves the trace for function i in file `trace.f`i. These trace files are invaluable debugging tools that can help you understand exactly where the hits and misses for each transpose function are coming from. To debug a particular function's performance, simply run its trace through the reference simulator with the verbose option:

  ```
  linux> ./csim-ref -v -s 5 -E 1 -b 6 -t trace.f0
  L 704600,8 miss
  S 784600,8 miss eviction
  L 704608,8 miss eviction
  S 784700,8 miss
  L 704610,8 hit
  ...
  ```

- Since your transpose function is being evaluated on a direct-mapped cache, conflict misses are a potential problem, both within the individual matrices, between them, and between the matrices and the temporary data. Think about the potential for conflict misses in your code, especially along the diagonal. Try to think of access patterns that will decrease the number of these conflict misses. This will in turn lower the number of clock cycles required.

- You are guaranteed that matrices `A` and `B`, and temporary storage `tmp` all align to the same positions in the cache. That is, if a_A, a_B, and a_t are the starting addresses of `A`, `B`, and `tmp`, respectively, then $a_A \bmod C = a_B \bmod C = a_t \bmod C$, where C is the cache size (in bytes). Also, these all begin on a cache-block boundary. That is, $a_A \bmod B = a_B \bmod B = a_t \bmod B = 0$, where $B = 2^b$ is the block size.

- It is not likely that you will want to use 256 temporaries in any of your transpose routines. However, having this many allows you to strategically choose which ones to use in order to avoid conflicts with the elements of `A` and `B` you are reading and writing.

- Blocking is a useful technique for reducing cache misses. See

`http://csapp.cs.cmu.edu/public/waside/waside-blocking.pdf`

for more information. You will need to experiment with a number of different blocking strategies.

7 Putting it all Together

We have provided you with a *driver program*, called `./driver.py`, that performs a complete evaluation of your simulator and transpose code. This is the same program that Autolab uses when it autogrades your handins. The driver uses `test-csim` to evaluate your simulator, and it uses `test-trans` to evaluate your submitted transpose function for correctness (ten matrix sizes) and performance (two matrix sizes). Then it prints a summary of your results and the points you have earned.

To run the driver, type:

```
linux> ./driver.py
```

8 Handing in Your Work

To create a tar file to submit to autolab, run `make handin`. This will create `handin.tar` that contains your current `csim.c` and `trans.c` files.

Then upload this file (and only this file!) to Autolab, which will autograde your submission and record your scores. You may handin as often as you like until the due date.

You can also run `make submit` which uses the autolab command line interface to hand in your work.

IMPORTANT: Do not create this file on a Windows or Mac machine, and do not upload files in any other archive format, such as `.zip`, `.gzip`, or `.tgz` files.

A Appendix

A.1 Trace File Format

The `traces` subdirectory of the handout directory contains a collection of *reference trace files* that we will use to evaluate the correctness of the cache simulator you write in Part A.

The memory traces have the following form:

```
L 04f6b868,8
S 7ff0005c8,8
```

Each line denotes one memory access. The format of each line is

Op *Addr*`,`*Size*

The *Op* field denotes the type of memory access: "`L`" a data load, and "`S`" a data store. The *Addr* field specifies a 64-bit hexadecimal memory address. The *Size* field specifies the number of bytes accessed by the operation. For example, the first line above attempts to *load 8 bytes* from the address *0x04f6b868*.

A.2 Clang format

To help you with style, we will be using `clang-format` to format your `csim.c` and `trans.c` files. This will format your code based on the rules in the file `.clang_format`. When you run `make handin` to create your tar file to handin to autolab, it will also format your files. If you want to modify the way your code is formatted, you can change the `.clang_format` file. For information about how to change the rules in the file please see `https://clang.llvm.org/docs/ClangFormatStyleOptions.html`.