CSCI 2021, Spring 2015
Cache Lab: Understanding Cache Memories
Assigned: Wednesday, April 8, 2015
Due: Monday, April 27, 11:55 PM

Ravi Raj (rajxx027@umn.edu) is the lead person for this lab.

1 Introduction

The lab consists of two parts. In the first part you will write a small C program (about 200-300 lines) that simulates the behavior of a cache memory. In the second part, you will optimize a matrix wavefront function for a **matrix size of 256x256**, with the goal of minimizing the number of cache misses for a **2 way set-associative cache**. All questions can be directed primarily to the TAs. Also, feel free to ask questions on the lab forum or contact us in order to maximize your understanding of the class materials.

2 Logistics

This is an individual project. All handins are electronic. You must do this lab on a CSE LABS machine, which support 64 bit machines. Also maintain 400MB of free area in your /home/x500 directory.

3 Overview

Part A of this lab requires writing a cache simulator and Part B requires performance improvement of caches using techniques which improve temporal and spatial locality. This lab will help you in supplementing your lecture knowledge with hands on experience. This lab includes the grading program that will be used to grade your submissions. You should be able to know your approximate grade before submitting your program.

4 Downloading the assignment

Your lab materials are contained in a Unix tar file called `cachelab-handout.tar`, which you can download from Moodle. Start by copying `cachelab-handout.tar` to a protected directory in a CSE-LABS machine in which you plan to do your work. Then give the command

```
linux> tar xvf cachelab-handout.tar
```

This will create a directory called `cachelab-handout` that contains a number of files. You will be modifying and handing in two files: `csim.c` and `trans.c`. To compile these files, type:

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

WARNING: Do not let the Windows WinZip program open up your .tar file (many Web browsers are set to do this automatically). Instead, save the file to your working directory and use the Linux tar 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!).

5 Description

The lab has two parts. In Part A you will implement a cache simulator. In Part B you will write a matrix wavefront function that is optimized for cache performance.

5.1 Reference 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 the cache simulator you write in Part A. The trace files are generated by a Linux program called `valgrind`. For example, typing

```
linux> 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.

5.2 Part A: Writing a Cache Simulator

In Part A you will write a cache simulator in `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 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 re-play`

The command-line arguments are based on the notation (s, E, and b) from lecture notes. For example:

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

The same example in verbose mode:

```
linux> ./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 job for Part A is to fill in the `csim.c` file so that it takes the same command line arguments and produces the identical output as the reference simulator. Notice that this file is almost completely empty. You'll need to write it from scratch.

Programming Rules for Part A

- Include your name and ID in the header comment for `csim.c`.
- 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 allocate storage for your simulator's data structures using the malloc function. Type "man malloc" for information about this 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 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(hit_count, miss_count, eviction_count);
```

- For this 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.

5.3 Part B: Optimizing Matrix Wavefront Transport Function

In Part B you will write a matrix wavefront transport function in trans.c that causes as few cache misses as possible. Let A denote a matrix, and A ij denote the component on the ith row and jth column.
The definition of one such matrix wavefront is given by -

```
for (int j = 1; j < N; j++) {
      for (int i = 1; i < M; i++) {
            A[i][j] = A[i-1][j-1] + A[i-1][j] + A[i][j-1];
      }
}
```

To help you get started, we have given you this example of matrix wavefront transport function in `trans.c` that computes the wavefront in above fashion.

```
char matrix_wavefront_desc[] = "Simple column-wise matrix wavefront calculations";
void matrix_wavefront(int M, int N, int A[M][N], int s, int E, int b)
```

The above function is inefficient because the access pattern results in relatively many cache misses. Your job in Part B is to write a similar transport function which computes same matrix wavefront as mentioned above, called `matrix_wavefront_submit`, that minimizes the number of cache misses for matrix A.

```
char matrix_wavefront_submit_desc[] = "Matrix Wavefront submission";
void matrix_wavefront_submit(int M, int N, int A[M][N], int s, int E, int b)
```

Do not change the description string ("`Matrix Wavefront submission`") for your `matrix_wavefront_submit` function. The autograder searches for this string to determine which transport function to evaluate for credit.

Programming Rules for Part B

• Include your `name` and `ID` in the header comment for `trans.c`.

• Your code in `trans.c` must compile without warnings to receive credit.

• You are allowed to define at most 12 local variables of type int per wavefront transport function[1] **Usage of variable types other than int (like long,short, char etc.) is NOT ALLOWED. You should focus towards the optimal cache performance with spatial and temporal locality of the matrix elements.**

• You are not allowed to sidestep the previous rule by using any bit tricks to store more than one value to a single variable.

• Your matrix wavefront transport function may not use recursion.

• **In general you may not use more than 12 local variables and even** if you choose to use helper functions, you may not have more than 12 local variables on the stack at a time between your helper functions and your top level wavefront transport function. For example, if your wavefront declares 8 variables, and then you call a function which uses 4 variables, which calls another function which uses 2, you will have 14 variables on the stack, and you will be in violation of the rule.

• You are NOT allowed to define any arrays in your code or to use any variant of malloc.

6 Evaluation

[1] The reason for this restriction is that our testing code is not able to count references to the stack. We want you to limit your references to the stack and focus on the access patterns of the source and destination arrays.

This section describes how your work will be evaluated. The full score for this lab is 50 points:
• Part A: 27 Points
• Part B: 18 Points
• Style: 5 Points

6.1 Evaluation for Part A

For Part A, we will run your cache simulator using different cache parameters and traces. There are eight test cases, each worth 3 points, except for the last case, which is worth 6 points:

```
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 2 -E 4 -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, 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.

6.2 Evaluation for Part B

For Part B, we will evaluate the correctness and performance of your `matrix_wavefront_submit` function for matrix of size 256x256 on two different cache sizes:

a. s = 5, E = 2, b = 3
b. s = 8, E = 2, b = 3

6.2.1 Performance (18 pts)

For the matrix size of 256x256, the performance of your `matrix_wavefront_submit` function is evaluated by using valgrind to extract the address trace for your function, and then using the reference simulator to replay this trace on two different cache with parameters (s =

5, E = 2, b = 3) and (s = 8, E = 2, b = 3). Your performance score is evaluated as per the cache misses (m) threshold -

- `s = 5, E = 2, b = 3: 10 points if m < 35000,`
  `7 points if 35000 < m <=50000,`
  `4 points if 50000 < m <=70000,`
  `0 points if m > 70000`
- `s = 8, E = 2, b = 3: 8 points if m < 33000,`
  `5 points if 33000 < m <= 48000,`
  `2 points if 48000 < m <= 68000,`
  `0 points if m > 68000`

Your code must be correct to receive any performance points for a particular size. Your code only needs to be correct for these two cases and you can optimize it specifically for these two cases.

6.3 Evaluation For Style

There are 5 points for coding style. These will be assigned manually by the TA. The TA will inspect your code in Part B for illegal arrays and excessive local variables.

7 Working on the Lab

7.1 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*	*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*
6	*(5,1,5)*	*265189*	*21775*	*21743*	*265189*	*21775*	*21743*	*traces/long.trace*
27								

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:

• 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.

• We have already provided you with the basic skeleton of the `csim.c` file - the structure to implement the cache line, various variables required and the prototypes of functions that need to be implemented. You are required to implement those functions and make everything work. The comments included in the csim.c file should guide you with the implementation.

• 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.

7.2 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 matrix wavefront transport functions that you have registered with the autograder.

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

```
/* Header comment */
char mat_wf_simple_desc[] = "A simple Matrix Wavefront computation";
void mat_wf_simple(int M, int N, int A[N][M], int B[M][N], int s, int E, int b)
{
/* your wavefront transport code here */
}
```

Register a particular matrix wavefront transport function with the autograder by making a call of the form:

```
registerTransFunction(mat_wf_simple, mat_wf_simple_desc);
```

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

```
registerTransFunction(matrix_wavefront_submit, matrix_wavefront_submit_desc);
```

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

For example to test your registered wavefront transport functions for `256x256` matrix size on cache with features set: `s = 5, E=2, b=3` rebuild `test-trans` and run below command:

```
linux> make
linux> ./test-trans -M 256 -N 256 -s 5 -E 2 -b 3

Function 0 (3 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=2, b=3)
func 0 (Matrix Wavefront 1 submission): hits:194312, misses:65796, evictions:65732

Function 1 (3 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=2, b=3)
func 1 (Matrix Wavefront 2 submission): hits:162691, misses:97417, evictions:97353

Function 2 (3 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=2, b=3)
func 2 (Simple column-wise matrix wavefront calculations): hits:162310, misses:97798,
evictions:97734

Summary for official submission (func 0): correctness=1 misses=65796 for cache
features ( s=5 E=2 b=3 )
```

In this example, we have registered three different transport 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.

Similarly for the other cache configuration of `s=8, E=2, b=3` one should run the below command:

```
linux> ./test-trans -M 256 -N 256 -s 8 -E 2 -b 3
```

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

• The `test-trans` program saves the trace for function `i` in file trace.fi.[2] These trace files are invaluable debugging tools that can help you understand exactly where the hits and misses for each wavefront function are coming from. To debug a particular function, simply run its trace through the reference simulator with the verbose option:

```
linux> ./csim-ref -v -s 5 -E 1 -b 5 -t trace.f0

S 68312c,1 miss
L 683140,8 miss
L 683124,4 hit
L 683120,4 hit
L 603124,4 miss eviction
S 6431a0,4 miss
...
```

• 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.

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

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

for more information.

7.3 Putting it all Together

We have provided you with a driver program, called ./driver.py, that performs a complete evaluation of your simulator and matrix wavefront transport code. This is the same program

[2] Because valgrind introduces many stack accesses that have nothing to do with your code, we have filtered out all stack accesses from the trace. This is why we have banned local arrays and placed limits on the number of local variables

that will be used to autograde your handins. The driver uses test-csim to evaluate your simulator, and it uses test-trans to evaluate your submitted matrix wavefront transport function on two different 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 hand in your work for credit, run the below command :

```
linux> make handin
```

on a CSELABS machine to create the `ID_handin.tar` file that will contain your current `csim.c` and `trans.c` files. You should then upload this tarball (and only this tarball!) to Moodle. You may handin as often as you like until the due date.

Important Points Before Getting Started:

1) 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.

2) **DO NOT** make use of the Virtual Machines with hostnames - x2#-0$.cselabs.umn.edu (# can be 1 or 2 or 3 and $ can be 1 or 2 or 3 or 4 or 5).

3) Machines in **Keller 4-250, Keller -2-170 or Lind 40** are Linux operated 64 bit machines. For more details about the the lab machines you can have a look at
http://cselabs.umn.edu/labs

4) This lab **NEEDS** to be run on 64 bit machines. You can run below command on any terminal to find out the machine size.

```
linux> uname -a
```

If you get `x86_64` printed at the end of the output line, then it's a 64 bit machine.

5) Maintain some 400 MB of free space in your /home/x500 area, before starting this lab. Trace file named `trace.tmp` is generated for the simulations, which stores the combined traces for all of your implemented functions and is big in size. I suggest deleting the `trace.tmp` (which can be regenerated easily)file after your analysi due to space constraints. Alternatively you can also type the below command,

```
linux> make clean
```

Per user 500MB of /home/x500 area is allocated on CSE LABS machines and if it overshoots this limit, user accounts get locked.

6) To re-iterate, for part A , student should ONLY modify the csim.c file. For part B, ONLY trans.c file should be modified.