New York University
Computer Science Department
Courant Institute of Mathematical Sciences

Course Title: Computer Systems Organization **Course Number:** CSCI-UA.0201-007
Instructor: Jean-Claude Franchitti **Session:** 27

C Programming Lab #3 – Optional Extra Credit

I. Due

May 16, 2022 by midnight via NYU Brightspace.

II. Objectives

The purpose of this programming lab is for you to become more familiar with memory hierarchy and caching. In particular, this lab will help you understand the impact that cache memories can have on the performance of your C programs. You will develop this knowledge by writing a general-purpose cache simulator, and then optimize a small matrix transpose kernel to minimize the number of misses on your simulated cache. The lab consists of two parts. In the first part you will write a small C program (about 200-300 lines maximum) that simulates the behavior of a cache memory. In the second part, you will optimize a small matrix transpose function, with the goal of minimizing the number of cache misses.

III. References

1. Slides and handouts posted on the course Web site for session 26
2. Textbook Part I, Sections 6.1-6.7
3. Lab3 material contained in Unix tar file named “lab3.tar”

IV. Software Required

1. Microsoft Word.
2. WinZip as necessary.
3. Linux utilities and programs as needed (e.g., tar, gcc, make)

Important: You should do all your work on the 64-bit x86-64 Linux machines to which you have access within the CIMS cluster (i.e., i6.cims.nyu.edu).

V. Assignment Steps and Questions

1. Retrieve “lab3.tar”, copy it to a directory in which you plan to do your work, and run the following command:

```
access2> tar -xvf lab3.tar
```

2. This will create a directory called `lab3-handout` that contains a number of files. You will be modifying two files: `csim.c` and `trans.c`.

3. To compile these files, type:

```
access2> make clean
access2> make
```

4. Review and understand reference trace files:

   The `traces` subdirectory of the `lab3-handout` directory contains a collection of *reference trace files* that will be used 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

```
access2> 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. Part A - Writing a cache simulator:

   As mentioned earlier the lab has two parts. In Part A you will implement a cache simulator. In Part B you will write a matrix transpose function that is optimized for cache performance.

   The cache simulator that you will implement in `csim.c` 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.

   A binary executable of a *reference cache simulator*, called `csim-ref`, is provided to you in `lab3-handout` directory. It 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 replay

The command-line arguments are based on the notation (s, E, and b) from page 615 of the CSO course textbook. For example:

```
access2> ./csim-ref -s 4 -E 1 -b 4 -t traces/yi.trace
hits:4 misses:5 evictions:3
```

The same example in verbose mode:

```
access2> ./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.

Programming Rules for Part A:

- Include your name and loginID 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, you should focus on data cache performance. Therefore, your simulator should ignore all instruction cache accesses (lines starting with "I"). As mentioned earlier, `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` at the end of your `main` function to report the total number of hits, misses, and evictions:

  ```
  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.

6. Part B - Optimizing Matrix Transpose

In Part B you will write a transpose 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 *transpose* of A, denoted A^T, is a matrix such that $A_{ij} = A^T_{ji}$.

To help you get started, you are given an example transpose function in `trans.c` that computes the transpose of $N \times M$ matrix A and stores the results in $M \times N$ matrix B:

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

The example transpose function is correct, but it is inefficient because the access pattern results in relatively many cache misses.

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

```
char transpose_submit_desc[] = "Transpose submission";
void transpose_submit(int M, int N, int A[N][M], int B[M][N]);
```

Please, do *not* change the description string ("`Transpose submission`") for your `transpose_submit` function. The automated grading program, searches for this string to determine which transpose function to evaluate for credit.

Programming Rules for Part B:

- Include your name and loginID 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 transpose function.[1]
- You are not allowed to side-step the previous rule by using any variables of type `long` or by using any bit tricks to store more than one value in a single variable.
- Your transpose function may not use recursion.
- 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 transpose function. For example, if your transpose 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 stated above.
- Your transpose function may not modify array A. You may, however, do whatever you want with the contents of array B.
- You are NOT allowed to define any arrays in your code or to use any variant of `malloc`.

7. Evaluation:

This section describes how your work will be evaluated. The full score for this lab is 60 points as follows:

- Part A: 27 Points
- Part B: 26 Points
- Style: 7 Points

7.1 Evaluation for Part A:

For Part A, grading will consist of running 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:

```
access2> ./csim -s 1 -E 1 -b 1 -t traces/yi2.trace
access2> ./csim -s 4 -E 2 -b 4 -t traces/yi.trace
access2> ./csim -s 2 -E 1 -b 4 -t traces/dave.trace
access2> ./csim -s 2 -E 1 -b 3 -t traces/trans.trace
access2> ./csim -s 2 -E 2 -b 3 -t traces/trans.trace
access2> ./csim -s 2 -E 4 -b 3 -t traces/trans.trace
```

[1] The reason for this restricion is that the lab3 testing code is not able to count references to the stack. Therefore, you need to limit you references to the stack and focus on the access patterns of the source and destination arrays.

```
access2> ./csim -s 5 -E 1 -b 5 -t traces/trans.trace
access2> ./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.

7.2 Evaluation for Part B:

For Part B, grading will consist of evaluating the correctness and performance of your `transpose_submit` function on three different-sized output matrices:

- 32×32 ($M = 32, N = 32$)
- 64×64 ($M = 64, N = 64$)
- 61×67 ($M = 61, N = 67$)

7.2.1. Performance (26 pts)

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

Your performance score for each matrix size scales linearly with the number of misses, m, up to some threshold:

- 32×32: 8 points if m < 300, 0 points if m > 600
- 64×64: 8 points if m < 1, 300, 0 points if m > 2, 000
- 61×67: 10 points if m < 2, 000, 0 points if m > 3, 000

Your code must be correct to receive any performance points for a particular matrix size. Your code only needs to be correct for these three cases and you can optimize it specifically for these three cases. In particular, it is perfectly OK for your function to explicitly check for the input sizes and implement separate code optimized for each case.

7.3 Evaluation for Style:

Please note that there are 7 points for C coding style based on style guidelines documentation provided on the course website or any other related material (see http://www.nyu.edu/classes/jcf/CSCI-UA.0201-007/handouts/resources.html).

Your Part B code will be inspected for illegal arrays and excessive local variables.

8. Working on the lab:

8.1 Working on Part A:

An automated grading program, called `test-csim`, is provided to you in the `lab3-handout` directory. The grading program tests the correctness of your cache simulator on the reference traces. Be sure to compile your simulator before running the test:

```
access2> make
access2> ./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 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 recommend that you use the `getopt` function to parse your command line arguments. You'll need the following header files:

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

See "`man 3 getopt`" for details.

- 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.

8.2 Working on Part B:

An automated grading program, called `test-trans.c`, is provided to you in the `lab3-handout` directory. The grading program tests the correctness and performance of each of the transpose functions that you are registering with it.

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(int M, int N, int A[N][M], int B[M][N])
{
     /* your transpose code here */
}
```

You can register a particular transpose function with the automated grading program by making a call of the form:

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

within the `registerFunctions` routine in `trans.c`. At runtime, the automated grading program will evaluate each registered transpose function and print the corresponding 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` code for an example of how the `registerTransFunction` works.

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

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:

```
access2> make
access2> ./test-trans -M 32 -N 32
Step 1: Evaluating registered transpose funcs for correctness:
func 0 (Transpose submission): correctness: 1
func 1 (Simple row-wise scan transpose): correctness: 1
func 2 (column-wise scan transpose): correctness: 1
```

```
func 3 (using a zig-zag access pattern): correctness: 1

Step 2: Generating memory traces for registered transpose funcs.

Step 3: Evaluating performance of registered transpose funcs (s=5, E=1, b=5)
func 0 (Transpose submission): hits:1766, misses:287, evictions:255
func 1 (Simple row-wise scan transpose): hits:870, misses:1183, evictions:1151
func 2 (column-wise scan transpose): hits:870, misses:1183, evictions:1151
func 3 (using a zig-zag access pattern): hits:1076, misses:977, evictions:945
Summary for official submission (func 0): correctness=1 misses=287
```

In this example, we have registered four different transpose functions in `trans.c`. The `test-trans` program tests each one of the registered functions, displays the results for each one of them, and extracts the results for 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*.[2] 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, simply run its trace through the reference simulator with the verbose option:

```
access2> ./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
        ...
```

- Since your transpose function is being evaluated on a direct-mapped cache, conflict misses are a potential problem. You should 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 such conflict misses.
- Blocking is a useful technique for reducing cache misses. See

  http://www.nyu.edu/classes/jcf/CSCI-UA.0201-007/handouts/blocking.pdf

  for more information.

8.3 Putting it all together

A driver program, called `./driver.py`, is provided to you in the `lab3-handout` directory. It performs a complete evaluation of your simulator and transpose code. This is the same program your instructor will uses to evaluate your handins. The driver program uses `test-csim` to evaluate your simulator, and it uses `test-trans` to

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

evaluate your submitted transpose function on the three matrix sizes. It then prints a summary of your results and the points you have earned.

To run the driver, type:

```
i6> ./driver.py
```

9. Archiving your work:

   Each time you type `make` in the `lab3-handout` directory, the Makefile creates a tarball, called `userid-handin.tar`, that contains your current `csim.c` and `trans.c` files.

10. Provide details (in short report form) related to your completion of Parts A and B following a strict set of *coding rules*.

    Save your report as a Word document.

    Name the file "**firstname_lastname_lab_3.docx**" (e.g., "john_doe_lab_3.docx").

    Submit your lab assignment via NYU Brightspace, including your report word document and your named archive `userid-handin.tar`, by the due date.

    Note: Pre-approved delayed submissions must be emailed directly to the professor. Please use the following naming convention in the subject line of the eMail for delayed submissions:

    > "**CSO - firstname lastname - lab 3**"
    > (e.g.: "CSO - John Doe - lab3").

    Since source code is submitted, include your name as a comment at the top of each file (note: all files submitted should include your name). Submit the electronic copy of the report and the code via NYU Brightspace.

VII. <u>Deliverables</u>

1. Electronic submission:

   Your report and lab assignment files must be submitted via NYU Brightspace. The files must be created and sent by the due date/time shown at the beginning of this document. After that date/time, the lab assignment is late. The email clock is the official clock. You may submit your work as often as you like until the due date of the lab assignment.

   Please note the following:

   - To receive credit, you will need to upload your report file and `csim.c` and `trans.c` files within your named archive (i.e., `userid-handin.tar`).

   - At any point in time, your most recently uploaded files are your official submission. You may submit files as often as you like.

- You must remove any extraneous print statements from your `csim.c` and `trans.c` files before handing them in.

2. Cover page and other formatting requirements (to be included in the report document):

   The cover page supplied on the next page must be the first page of your report.

   Fill in the blank area for each field.

   <u>NOTE</u>:

   The sequence of the electronic submission is:

   1. **Cover sheet**
   2. **Report sheet(s)**

3. Grading guidelines (grade normalized to 100):

   Report Layout (15%)
   - Report is neatly assembled on 8 1/2 by 11 layout.
   - Report cover page with your name (last name first followed by a comma then first name), NYU ID and course section number with a signed statement of independent effort is included.
   - File name is correct.

   Answers to Individual Questions (85%):
   - Solutions to Parts A and B included in your `csim.c` and `trans.c` files are correct from a coding and performance standpoint and your name is included at the beginning of each file.
   - Assumptions provided as required.

   (100 points total, all questions weighted equally)

VIII. Sample Cover Sheet:

Name ________________________ Date: ____________
(last name, first name)

NYU ID: __________

Section: ___________

Lab 3

Total in points (100 points total): ____________________

Professor's Comments: