Computer Systems
Fall 2018
Lab Assignment 3: Understanding Cache Memories
Assigned: October 2, Due: October 10, **20:00**

Giulio Stramondo (`giuliostramondo+kicks2018@gmail.com`) is the lead person for this assignment.

1 Introduction

This lab will help you understand the impact that cache memories can have on the performance of your C programs. Specifically, you will optimize a small-matrix transpose function, with the goal of minimizing the number of cache misses.

2 Logistics

Please read the assignment carefully before you start working.

For this assignment you are requested to work in pairs. You MUST work with a different partner than in the previous 2 labs. Each pair is requested to submit the code and a lab report (see section 7 for more details).

Clarifications and corrections will be posted on the Canvas course page, if the need arises.

3 Getting the assignment

We expect you to work on your own machine, either directly using your Linux environment or using a virtual machine. You can download all necessary files directly from Canvas.

To work locally, you should just download the code from Canvas, store it in an appropriate folder (e.g., cachelab) and decompress it.

To work on `acheron`, you must download the code from Canvas to your local machine, copy the code from the local machine to the server (using `scp`), and finally login to the server (using `ssh`) to work.

4 Preparing the environment

This lab measures the performance of your code with the package valgrind: a tool suite for profiling Linux which is installed in the virtual machine of our bachelor, but not on acheron or your local machine.

Install Valgrind locally

As root, you could easily install valgrind with the command
`sudo aptitude install valgrind kcachegrind` [1].

You should have already done that during the first lab. If not - please make sure you will install it now.

Valgrind on acheron

To execute on acheron, you need to install valgrind locally. To do so: login (ssh) and perform the following actions in your home directory, where UVANETID must be substituted with your student id.

```
cd
mkdir .pkgs
cd .pkgs/
yumdownloader valgrind
rpm2cpio valgrind-3.8.1-8.el6.x86\_64.rpm | cpio -idv
export VALGRIND_LIB="/home/UVANETID/.pkgs/usr/lib64/valgrind"
export PATH=\$PATH:/home/UVANETID/.pkgs/usr/bin
```

To be able to use valgrind after login:

```
export VALGRIND_LIB="/home/UVANETID/.pkgs/usr/lib64/valgrind"
export PATH=\$PATH:/home/UVANETID/.pkgs/usr/bin
```

Alternatively, to export valgrind's location permanently one can modify the .bashrc file adding those two lines.

To test if everything went well, type `valgring`.

You should see:

```
valgrind: no program specified
valgrind: Use --help for more information.
```

From the cachelab-handout directory, type `./driver.py`.

You should see:

[1] `http://sbt.science.uva.nl/bterwijn/KI2016/INSTALL.txt`

```
Part A: Testing cache simulator
Running ./test-csim
[ ... more text ... ]

Part B: Testing transpose function
Running ./test-trans -M 32 -N 32
Running ./test-trans -M 64 -N 64
Running ./test-trans -M 61 -N 67
[ ... more text ... ]
```

5 Description

Your task is to write a matrix transpose function that is optimized for cache performance. The original lab has two parts. We focus only on part B.

5.1 Reference Trace Files

The trace files are generated by `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 Optimizing Matrix Transpose

You have to modify the given transpose function in `trans.c` such that it causes as few cache misses as possible.

Let A denote a matrix, and A_{ij} denote the component on the i-th row and j-th column. The *transpose* of A, denoted A^T, is a matrix such that $A_{ij} = A^T_{ji}$.

To help you get started, we have given you 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 new function 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]);
```

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.

Programming Rules

- Include your name and student 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 transpose function.[2]
- 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 to 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.
- 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`.

[2]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.

6 Working on the Lab

6.1 Objectives

You must provide a correct, and high-performing implementation of a matrix transpose function for three different-sized output matrices:

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

Your code must be correct (i.e., it must correctly transpose the matrix) to receive any performance points for a particular 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 recommended for your function to explicitly check for the input sizes and implement separate code optimized for each case*. Moreover, you need to minimize the number of cache misses for every case: less misses directly correlate with higher performance. We have provided an autograder to help with these measurements.

6.2 Using the autograder

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(int M, int N, int A[N][M], int B[M][N])
{
    /* 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 `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:

```
linux> make
linux> ./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 of the registered functions, displays the results for each, and extracts the results for the official submission.

6.3 Expected performance

For each matrix size, the performance of your `transpose_submit` function is evaluated by the autograder (i.e., 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:

32x32 : max points (8) if $m < 300$, 0 points if $m > 720$

61x67 : max points (10) if $m < 2000$, 0 points if $m > 3600$

64x64 : max points (8) if $m < 1300$, 0 points if $m > 2400$

6.4 Hints and suggestions

Here are some hints and suggestions for optimizing the matrix transpose:

- The `test-trans` program saves the trace for function i in file `trace.f`i.[3] 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:

```
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
...
```

- Since your transpose function is being evaluated on a direct-mapped cache, conflict misses are a potential problem. 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.

6.5 Checking the overall results

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 your instructor uses to evaluate your submission. The driver uses `test-csim` to evaluate your simulator, and it uses `test-trans` to evaluate your submitted transpose function on the three matrix sizes. Then it prints a summary of your results and the points you have earned.

To run the driver, type:

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

7 Lab Report

You will present on your approach to solve this assignment, the challenges you encountered, and the lessons you learned into a lab report.

[3] 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.

See the page 'Het labboek'[4] on the website on Academic Skills for some general pointers about what a lab journal/lab report should contain.

For this assignment, you will receive a template of such a lab report, and you will be requested to answer the questions in the provided text (a bit like "fill in the blanks", only with longer answers). You will be graded for these answers only, but make sure they are coherent, they make sense in the context of the report you submit, and they are readable. We do not correct grammar mistakes, but text that is not readable will not be graded. Correct answers need not be excessive in length (usually, one paragraph per (sub)question suffices), and can include images, data, or (psuedo)code snippets, if needed.

Each question has an allocated score. When a total score (i.e., the sum of all questions' scores) of a report exceeds 10, you simply have more opportunities to get a 10. Moreover, if your grade does exceed 10, excess points can be used to improve scores for other reports (your TA will manage these grades). Thus, it is in your overall advantage to try your best to answer all the questions.

Once you have filled in your answers, generate the PDF file of the report using your favorite Latex compiler (e.g., online, Overleaf or SharedLatex). The PDF of your report is the file you hand in as your lab report.

Like most of the course material, the lab report is provided in English. If you have problems writing your answers in English please inform your TA or contact the coordinators (Giulio, Ana) asap.

8 Evaluation

Your score will consist of two grades: code and report.

The code is evaluated for each matrix size. The maximum number of points is 26, which is equivalent to a grade of 10 for the code. See above in Section 6.3 for the exact numbers.

Your report will be again based on a number of questions, which give you the opportunity to explain your approach for each matrix size. Overall, the maximum grade for your report is 10.

9 Submission instructions

When you have completed the lab, you will hand in two files on Canvas: your code (`trans.c`) and your lab report. Make sure your code file contains the student names and IDs of both students.

Please make sure you will rename the file as `trans_XXXXXX_YYYYYY.c` where "XXXXXX" and "YYYYYY" stand for the student numbers in your pair. Likewise, the report should be named `Report_CacheLab_xxxxx_yyyyy.pdf`

Delayed submissions (up to 4h late after 20:00) are penalized with 1 point per every 1h. This applies to whichever part of the submission is delayed (solution, report, or both). Submissions that are received later than 4h (i.e., after midnight) are NOT graded (thus, will receive 0 points).

[4] `http://www.practicumav.nl/onderzoeken/labboek.html`