

CSCI 2467, Fall 2024

Cache Lab: Understanding Cache Memories

Due: Thursday Dec 2, 11:59pm

1 Overview

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

2 Downloading the assignment

As usual, you will download the handout tar file from Autolab and put it into your 2467 directory at `systems-lab` on your `webterminal.cs.uno.edu` as you did for previous labs. Then you extract the tar file like this:

```
$ tar xvf cachelab-handout.tar
```

This will create a directory called `cachelab-handout` that contains a number of files. You will be modifying only `trans.c`. To compile, type:

```
$ make clean
$ make
```

3 Description

In this lab 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 i th row and j th column. The *transpose* of A , denoted A^T , is a matrix such that $A_{ij} = A_{ji}^T$.

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 job here 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.

3.1 Programming Rules

- Include your name and UNO email address 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 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`.

4 Evaluation

We will evaluate 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 ($M = 61, N = 67$)

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.

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

- s is the number of *set index bits* ($S = 2^s$ is the number of sets)
- E is the *associativity* (number of lines per set)
- b is the number of block bits ($B = 2^b$ is the block size)

¹The 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.

In other words, the simulated cache will have 32 sets, a block size of 32, and is 1-way set associative.

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

- 32×32 : 12 points if $m < 300$, 0 points if $m > 600$
- 64×64 : 12 points if $m < 1,300$, 0 points if $m > 2,000$
- 61×67 : 16 points if $m < 2,000$, 0 points if $m > 3,000$

Your code must be correct 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 perfectly OK for your function to explicitly check for the input sizes and implement separate code optimized for each case.

5 Working on the Lab

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 :

```
$ make
$ ./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.

Here are some hints and suggestions for working on the matrix transpose function.

- 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 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.

²Because `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.

5.1 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 your instructor uses to evaluate your handins. 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:

```
$ ./driver.py
```

6 Handing in Your Work

You will submit only `trans.c` to Autolab.

There is a scoreboard for your optional enjoyment. It lists miss counts for the three test cases. Lower values are better. How low can you get your miss count?