Computer Systems
baiCOSY06, Fall 2016
Cache Lab: Understanding Cache Memories
Assigned: Oct. 3, Due: Wed., Oct. 12 11:59

Arnoud Visser (A.Visser@uva.nl) is the lead person for this assignment. Giulio Stramondo, Janosh Haber, Kyrian Maat, Tycho van der Ouderaa, Joop Pascha and Thomas Schaper are the teaching assistants.

1 Overview

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

2 Downloading the assignment

You can find the the file `cachelab-handout.tar` at the location `/opt/prac/cs_ai/cachelab` on the machine `acheron.fnwi.uva.nl` or on location `https://staff.fnwi.uva.nl/a.visser/education/CS/perf/cachelab-handout.tar`.

Start by copying `cachelab-handout.tar` to a protected Linux directory 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 the file `trans.c`. To compile these files, type:

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

3 valgrind

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. As root, you could easily install it with the command `sudo aptitude install valgrind kcachegrind`[1]. You could add this package to your environment on acheron in the following way:

- Open your `$HOME/.bashrc`
- Add the following lines:

```
# User specific aliases and functions
PACKAGEPATH=/opt/prac/cs_ai/pkg:/usr/local/pkg; export PACKAGEPATH
if which softpkg > /dev/null ]]; then
    eval "$(softpkg -b)"
elif [[ -x /opt/prac/cs_ai/packages/softpkg2.4/bin/softpkg ]]; then
    eval "$(/opt/prac/cs_ai/packages/softpkg2.4/bin/softpkg -b)"
fi
```

- Alternatively, overwrite your `.bashrc` with the version of this course by the command

  ```
  cp  /opt/prac/cs_ai/etc/bashrc ~/.bashrc
  ```

  In addition, you should add the package `valgrind-3.11.0` to your `.pkgrc`-file. Alternatively, overwrite your `.pkgrc` by the command

  ```
  cp  /opt/prac/cs_ai/etc/pkgrc ~/.pkgrc
  ```

  Test your installation by typing `source .bashrc` followed by `which valgrind`.

4 Description

The lab as given at CMU has two parts. Part A is about the implemention of a cache simulator. You can skip that part at the UvA. In Part B you will write a matrix transpose function that is optimized for cache performance.

4.1 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, 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:

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

```
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]);
```

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 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.[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`.

4.2 Evaluation for Part B

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

[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.

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

4.2.1 Performance

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 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 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.

4.3 Labbook

The labbook should make the improvement of the performance of the transpose-function traceable. The Labbook will be evaluated on content, structure, wording and completeness, as described in

`http://www.practicumav.nl/onderzoeken/labboek.html`

The course staff will inspect your code in Part B for illegal arrays and excessive local variables. The balance in the grading between labbook and code performance will be 12 to 27.

5 Working on the Lab

5.1 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(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.

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.[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.

- Valgrind running amok. Sometimes valgrind is still checking in the background, creating huge log-files. You could check this with `ps -Afu uvanetid | grep valgrind`. You could kill this processes with the command `kill -9 ‘pidof valgrind‘`.

5.2 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. 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
```

6 Handing in Your Work

Two BlackBoard will be created, one for your labbook and one for your code (`trans.c` file). If you have a better implementation, feel free to increase the version number and do another submission.

[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.