CS 449 Fall 2019
Cache Lab: Understanding Cache Memories

1 Logistics

This is an *individual* project. You must run this lab on a 64-bit x86-64 machine. All submissions are electronic. We strongly advise you to test your code on the CS Linux machine before submitting it. Always test your submission on Gradescope in advance. We will not return any points lost because you did the lab in a different environment and have not tested it on Gradescope.

Start early to get it done before the due date. Assume things will not go according to plan, and so you must allow extra time for heavily loaded systems, dropped Internet connections, corrupted files, traffic delays, minor health problems, etc.

2 Overview

This lab will help you understand the impact that cache memories can have on the performance of your C programs. Some of the skills tested are:

- Gain basic familiarity with cache geometries and how different associativities and line sizes present themselves.
- Evaluate a claim made in the lecture that two seemingly equivalent ways of writing a program can have vastly different performance because of cache memories
- Gain experience in designing cache-friendly code.

The lab consists of two parts. In the first part of this lab, Chip D. Signer, Ph.D., is trying to reverse engineer a competitor's microprocessors to discover their cache geometries and has recruited you to help. Instead of running programs on these processors and inferring the cache layout from timing results, you will approximate his work by using a simulator. In the second part of this lab, you will optimize a small matrix transpose function, with the goal of minimizing the number of cache misses.

3 Downloading the assignment

Your lab materials are contained in an archive file called `cachelab-handout.zip`, which you can download to your Linux machine as follows.

```
linux$ wget https://bit.ly/2BpSKGM -O cachelab-handout.zip
```

Start by copying `cachelab-handout.zip` to a protected Linux directory in which you plan to do your work. Then give the command below

```
linux$ unzip cachelab-handout.zip
```

This will create a directory called `cachelab-handout` that contains a number of files. Below are the important files for Part A:

- **cache-test-skel.c** – Skeleton code for determining cache parameters
- **testCache.sh** – Script that makes and runs cache-test with all the cache object files
- **caches/cache_*.o** – "Caches" with known parameters for testing your code
- **support/mystery-cache.h** – Defines the function interface that the object files export

Below are the important files for Part B:

- **trans.c** – Place for your transpose functions
- **grade_trans.py** – Script to run all tests and calculate score
- **support/test-trans.c** – Code to test efficiency of your functions in trans.c
- **support/tracegen.c** – Used by test-trans.c to generate memory traces
- **support/csim-ref** – Reference Cache Simulator used by test-trans

You will be modifying only two files: `cache-test-skel.c` and `trans.c`.

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

4 Description

The lab has two parts. In Part A you will write a C code to identify the cache parameters of some given mystery "caches". In Part B you will write a matrix transpose function that is optimized for cache performance.

4.1 Part A: Inferring Mystery Cache Geometries

Each of the "processors" is provided as an object file (.o file) against which you will link your code. The file `mystery-cache.h`, shown below, defines the function interface that these object files export.

```
typedef unsigned long long addr_t;
typedef unsigned char bool_t;
#define TRUE 1
#define FALSE 0

/* Lookup an address in the cache. Returns TRUE if the access hits,
    FALSE if it misses. */
bool_t access_cache(addr_t address);

/* Clears all words in the cache (and the victim buffer, if
    present). Useful for helping you reason about the cache
    transitions, by starting from a known state. */
void flush_cache(void);
```

It includes a typedef for a type `addr_t` (an unsigned 8-byte integer) which is what these (pretend) caches use for "addresses", or you can use any convenient integer type.

Your job is to fill in the function stubs in `cache-test-skel.c` which, when linked with one of these cache object files, will determine and then output the cache size, associativity, and block size. You will use the functions above to perform cache accesses and use your observations of which ones hit and miss to determine the parameters of the caches.

Some of the provided object files are named with this information (e.g. `cache_65536c_2e_16k.o` is a 65536 **Byte c**apacity, 2-way set-associativ**e** cache with 16 **Byte** bloc**k**s) to help you check your work.

Your job in Part A is to complete the 3 functions in `cache-test-skel.c` that have `/* YOUR CODE GOES HERE */` comments in them.

IMPORTANT NOTICE: You should NOT be calling any functions other than `flush_cache` and `access_cache` inside of your functions. For example, you cannot call `get_block_size()` say inside of `get_cache_assoc`.

4.2 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:

```
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.

4.3 Trace Files

Tools in this lab use `valgrind` traces of your code running on a cache simulator to evaluate the efficiency of your transpose function. 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.

Programming Rules for Part B

- Include your name and PITT 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.[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`.

5 Evaluation

This section describes how your work will be evaluated. The full score for this lab is 30 points:

- Part A: 12 Points
- Part B: 18 Points

5.1 Evaluation for Part A

In the electronic evaluation, there will be 4 mystery cache object files, whose parameters you must discover on your own. You can assume that the mystery caches have sizes that are powers of 2 and use a least recently used replacement policy. You cannot assume anything else about the cache parameters except what you can infer from the cache size. Finally, the mystery caches are all pretty realistic in their geometries, so use this fact to sanity check your results.

Each function you wrote (total of 3 functions) that identified the correct cache parameter receives 1 point, accounting for a total of 3 points per cache object. Since we will evaluate your code against 4 cache objects, the total points possible for this part will be 12.

5.2 Evaluation for Part B

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

- 32×32 ($M = 32$, $N = 32$)

[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.

- 64×64 ($M = 64$, $N = 64$)

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: 10 points if $m <= 330$, 0 points if $m > 600$
- 64×64: 8 points if $m <= 1,800$, 0 points if $m > 2,400$

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**. In particular, it is perfectly OK for your function to explicitly check for the input sizes and implement separate code optimized for each case.

We have provided you with a driver program, called `./grade_trans.py`, that performs a complete evaluation of your transpose code. This is the same program your instructor uses to evaluate your submission. The driver uses `test-trans` to evaluate your submitted transpose function on the two matrix sizes. Then it prints a summary of your results and the points you have earned. To run the driver, type:

```
linux$ make # You need to run make every time you change your code
     $ python grade_trans.py
```

6 Working on the Lab

6.1 Working on Part A

We have provided you with a Makefile that includes a target cache-test. To use it, set `TEST_CACHE` to the object file to link against on the command line. That is, from within the working directory run the command:

```
linux$ make cache-test TEST_CACHE=caches/cache_65536c_2e_16k.o
```

This will create an executable **cache-test** that will run your cache-inference code against the supplied cache object. Run this executable like so:

```
linux$ bash testCache.sh
```

6.2 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 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.

6.2.1 Hints

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:

  ```
  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/3e/waside/waside-blocking.pdf`

  for more information.

- Only memory accesses to the heap are monitored. This may be helpful... Just be careful not to break any programming rules!

- Your code only needs to work on the three array sizes given. While in CS we often try to avoid writing non-generalized code, in this case it may be quite helpful to optimize to specific cases first to reach the desired level of performance.

7 Submitting Your Work

You need to submit `cache-test-skel.c` and `trans.c` files using Gradescope. Please make sure that your final transpose code is in a function named `transpose_submit`.

[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.