CS154 Project 3: Caching
Due: 11:59pm Thursday November 14, 2019

1 Overview

This project will help you understand the impact that cache memories can have on C program performance.

The project consists of two parts. In Part A you will write (in `csim.c`) a small C program (about 250-350 lines) that simulates the behavior of a cache memory. In Part B, you will optimize (in `trans.c`) a small matrix transpose function, with the goal of minimizing the number of cache misses.

Doing an `svn update` in your `CNETID-cs154-aut-19` checkout should create a new directory `p3cache`. This contains all the files you need for Project 3. We will be grading your work according to the modifications of files `csim.c` and `trans.c` (and no other files) that are committed prior to the deadline.

The files you check in will be graded by the staff's copy of the `driver.py` script included in your `p3cache` directory. The grade produced by this script will be your grade. **If the script fails, or your code fails to compile, you will receive no credit.**

2 Motivation

We have already discussed in class how changing the software can speed up code and reduce cache misses. This is such a fundamentally important program optimization that we would be remiss not to give you some practical experience in this area. This project requires you to understand the workings of the cache well enough to implement one in software. It then requires that you understand how to modify code to improve cache behavior. Professors in the department have devoted countless hours to this type of optimization on real systems. This kind of optimization is even more important on multicore, but you must first understand how to do it for a single cache – hence this assignment.

3 Part A: Writing a Cache Simulator

3.1 Reference Trace Files

This project involves a cache *simulator*: we are not studying the utilization of the real cache on the computer's CPU, but of a simple made-up cache that is implemented in software. However, the cache utilization

is assessed according to the sequence of memory references from real programs (including the matrix transpose in Part B), which are "replayed against" the cache simulator. The sequence of memory references are stored in *reference trace files*, which are contained in the `traces` subdirectory. We use these 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

```
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.

3.2 Description

In Part A, you will write a cache simulator in `csim.c` that takes a `valgrind` memory trace as input, simulates the hit/miss behavior of a cache memory (with different E, B, and S parameters) on this trace, and outputs the total number of hits, misses, and evictions. For this project, use the allocate-on-write policy to handle write misses in your implementation of the simulator.

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.

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 597 of the CS:APP2e textbook. For example:

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

The same example in verbose mode:

```
linux> ./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 given `csim.c` file so that it produces the identical output as the reference simulator. This will consist of implementing the `simulate` function that opens and parses each line of whatever trace file is given with the `-t` option, and tracks what that memory operation would do within the context of the cache that you are simulating. Your simulator will consist of new global variables to represent the state of the simulator, and functions (called by `simulate`) that operate on the simulator in response to the memory operations read from the trace file.

3.3 Programming Rules

- Your `csim.c` file must compile without warnings in order to receive full 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.
- For this project, we are interested only in data cache performance, so your simulator should ignore all instruction cache accesses (lines starting with "I"). Recall that `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, although there are many different legitimate ways to do so.
- To receive credit for Part A, you must (as it is now in `csim.c`) call the function `printSummary`, with the total number of hits, misses, and evictions, at the end of your `main` function:

```
printSummary(hitCount, missCount, evictionCount);
```

- For this project, 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.

3.4 Evaluation (Part A Max Score = 60 points)

For Part A, we will run your cache simulator using different cache parameters and traces. There are ten test cases, the first 6 are worth 4 points each, and the last 4 are worth 9 points each:

```
linux> ./csim -s 1 -E 1 -b 1 -t traces/yi2.trace
linux> ./csim -s 4 -E 2 -b 4 -t traces/yi.trace
linux> ./csim -s 2 -E 1 -b 4 -t traces/dave.trace
linux> ./csim -s 2 -E 1 -b 3 -t traces/trans.trace
linux> ./csim -s 2 -E 2 -b 3 -t traces/trans.trace
linux> ./csim -s 2 -E 4 -b 3 -t traces/trans.trace
linux> ./csim -s 5 -E 1 -b 5 -t traces/trans.trace
linux> ./csim -s 5 -E 1 -b 5 -t traces/long.trace
linux> ./csim -s 4 -E 4 -b 5 -t traces/long.trace
linux> ./csim -s 1 -E 8 -b 8 -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. For each of the first 6 tests, 2 (out of 4) points are awarded for the correct number of hits, and 1 point each for the correct number of misses and evictions, respectively. For the last 4 tests, each of your reported number of hits, misses, and evictions is worth 1/3 of the credit for that test case. For example, here a particular test case is worth 9 points, and if your simulator outputs the correct number of hits and misses, but reports the wrong number of evictions, then you will earn 6 points.

3.5 Working on Part A

We have provided you with an autograding program, called `test-csim`, that tests the correctness of your cache simulator on the reference traces. Be sure to compile your simulator before running the test:

```
linux> make
linux> ./test-csim
                        Your simulator     Reference simulator
Points (s,E,b)    Hits  Misses  Evicts    Hits  Misses  Evicts
     4 (1,1,1)       9       8       6       9       8       6  traces/yi2.trace
     4 (4,2,4)       4       5       2       4       5       2  traces/yi.trace
     4 (2,1,4)       2       3       1       2       3       1  traces/dave.trace
     4 (2,1,3)     167      71      67     167      71      67  traces/trans.trace
```

```
 4 (2,2,3)     201      37      29     201      37      29  traces/trans.trace
 4 (2,4,3)     212      26      10     212      26      10  traces/trans.trace
 9 (5,1,5)     231       7       0     231       7       0  traces/trans.trace
 9 (5,1,5)  265189   21775   21743  265189   21775   21743  traces/long.trace
 9 (4,4,5)  268525   18439   18375  268525   18439   18375  traces/long.trace
 9 (1,8,8)  272531   14433   14417  272531   14433   14417  traces/long.trace
60

TEST_CSIM_RESULTS=60
```

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 the 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.
- 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.
- Use svn. When you get something working, check in that version (you can always back up to it later). Never add to a version that works without first committing your code to the repo.

4 Part B: Optimizing Matrix Transpose

4.1 Description

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.

4.2 Programming Rules

- Your code in `trans.c` must compile without warnings to receive full 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 matrix A. You may, however, do whatever you want with the contents of matrix B.
- You are NOT allowed to define any arrays in your code or to use any variant of `malloc`.

4.3 Evaluation (Part B Max Score = 40 points)

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

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

[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 matrices.

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: 15 points if $m < 300$, 0 points if $m > 600$
- 32×64: 15 points if $m < 700$, 0 points if $m > 1,000$
- 64×64: 10 points if $m < 1,300$, 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 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.

4.4 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.[2] These trace files are invaluable performance 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
  ...
  ```

[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.

- 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.
- When optimizing code, you will often reach a point where changes actually make things worse. To avoid losing work, again, *use svn*. Make sure that you have committed your code every time you reach a new level of performance. If you don't do this, you will be extremely frustrated at times when you can't remember exactly how you got the better result earlier.

5 Putting it all Together

The maximum score for the whole Project 3 is 100, where

- Part A max score = 60 points
- Part B max score = 40 points

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 code for part A and B. The driver uses `test-csim` to evaluate your simulator, and it uses `test-trans` to evaluate your submitted transpose function on the different matrix sizes. Then it prints a summary of your results and the points you have earned. To run the driver, type:

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

The output of our copy of driver.py will be your grade. If it does not work, or your code fails to compile, you will receive no credit. Test before you commit your final version.

6 Acknowledgements

This assignment was created by the text book authors and their TAs. It has been modified by the CS154 instructors.