15-213/18-213, Spring 2012
Lab Assignment L4: Cache Lab
Assigned: Tuesday Feb. 21
Due: Thursday, Mar. 1, 11:59PM
Last Possible Time to Turn in: Saturday, Mar. 3, 11:59PM

1 Logistics

This is an individual project. All handins are electronic.

You must do this lab on one of the class shark machines or an Andrew Linux machine.

Please contact the 15-213 staff through Piazza for all questions about assignment.

2 Overview

This lab will help you understand cache memories. As you've seen in lecture and read in your textbook, understanding cache memories is important because they can have a significant impact on the performance of your C programs.

The lab consists of two parts. In the first part you will write a small C program (about 200-300 lines) that simulates the behavior of a cache memory. In the second part, you will optimize a small matrix transpose function, with the goal of minimizing the number of cache misses on a reference version of the cache simulator.

3 Downloading the assignment

Your lab materials are contained in a Unix tar file called `cachelab-handout.tar`, which you can download from Autolab. After logging in to Autolab at

```
http://autolab.cs.cmu.edu
```

you can retrieve the `cachelab-handout.tar` file by selecting "Cache Lab - Download lab materials" and then hitting the "Save File" button.

Start by copying `cachelab-handout.tar` to a protected directory in Andrew in which you plan to do your work. Then give the command "`tar xvf cachelab-handout.tar`". This will create a directory called `cachelab-handout` that contains the following files (some of them are generated by 'make'):

- `cachesim.c`: Your cache simulator file.
- `trans.c`: Your matrix transpose file.
- `Makefile`: Used by 'make' to generate binaries.
- `test`: A binary you can use to test/evaluate your work
- `driver.py`: A script that grades your work and submits an unofficial score to Autolab.
- `traces`: A folder containing some reference traces.
- `andrewID_handin.tar`: A tarball containing your cachesim.c and trans.c files that you can upload to Autolab for credit. This file is updated each time you run 'make'.

You will be modifying two files: `cachesim.c` and `trans.c`. To compile these files, simply type:

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

Note that "make" also creates a tarball containing `cachesim.c` and `trans.c`. You will submit this tarball (and only this tarball) to Autolab.

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

NOTE: Your lab submission must compile without warnings.

4 The Cache Lab

This lab has two parts. In Part (a) you will implement a cache simulator. In Part (b) you will write a matrix transpose function that is optimized for cache performance.

4.1 Part (a): Building a Cache Simulator

4.1.1 Generating Memory Traces

On the shark machines, there is a tool called `valgrind`, which can be used to record all of the memory accesses performed by the execution of a given binary. For example, you can execute:

```
  linux> valgrind --log-fd=1 --tool=lackey -v --trace-mem=yes echo cachelab
```

The above command runs ”echo cachelab” with valgrind and displays a trace of the memory accesses to `stdout`.

In the output, you should see entries such as the following:

```
I 0400d7d4,8
 M 0421c7f0,4
 L 04f6b868,8
 S 7ff0005c8,8
```

The format each line in the trace is ”Operation Address,Size”. In the operation field, ”I” denotes an instruction load; ”L” denotes a data load; ”S” denotes a data store; and ”M” denotes a data modify, **which should be treated as a load followed by a store**. The memory address is given in hex format, followed by the number of bytes accessed.

4.1.2 Cache Simulator

You goal for Part (a) is to write a cache simulator that can take the memory traces from valgrind as input, simulate a cache, and output the number of cache hits, misses and evictions.

Your cache simulator should be able to handle different cache sizes and associativity. More specifically, your cache simulator should take the following arguments on the command line:

```
    -b: Number of block bits (so 2^b is the block size)
    -s: Number of set index bits (so 2^s is the number of sets)
    -E: Associativity (number of lines per set)
    -t: File name of the trace to replay
```

Note we use the same notation (s, S, b, B, E) as in your CS:APP2e textbook (page 597). Your cache simulator should use the LRU (least recently used) replacement policy for evictions.

Note that if you are using getopt function to parse your arguments you need to add

```
 #include <getopt.h>
 #include <stdlib.h>
 #include <unistd.h>
```

to your includes.

We have provided you with the binary of a reference cache simulator called `cachesim.example`. Type

```
    linux> ./cachesim.example
```

to see its usage and command line arguments.

Your job for Part (a) is to fill in the empty `cachesim.c` file so that it takes the same command line arguments and produces the same output as the reference simulator. Notice that this file is completely empty. You'll need to write it from scratch. In fact, until you add a `main` function, the `make` command will fail.

Note that `cachesim.example` can take an optional argument `-v`, which enables verbose output. Using this option will help you debug your cache simulator—it prints the hits, miss, and evictions after each memory access. You do NOT need to implement this `-v` feature in your `cachesim.c`, but we strongly recommend that you do so, as it will help you debug your code.

Important Notes:

- In order for us to evaluate your results, you must call the function `printCachesimResults()` with the number of cache hits, misses and evictions. For example:

  ```
  printCachesimResults(hit_count, miss_count, eviction_count);
  ```

  This function sends the results to the driver for evaluation. It should be called at the end of your `main` function in `cachesim.c`. You must call this function to get credit for Part (a).

- For this lab, we are interested only in data cache performance, so you should ignore the instruction cache accesses (lines starting with I). Notice that Valgrind always put ”I” in the first column, and ”M”, ”L”, ”S” in the second column. This may help you parse the trace.

- For the purpose of this lab, 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.

- If you wish to use C0 style contracts, from 15-122, you can include `contracts.h`. The provided Makefile will build `cachesim-debug` and `test-debug`, in addition to `cachesim` and `test`, which will have your contracts enabled.

4.2 Part (b): Optimizing Matrix Transpose

Your goal for Part (b) is to write a function in `trans.c` that computes the transpose of a matrix. This function you write should minimize the number of cache misses. We have included an inefficient transpose function to help get you started.

4.2.1 Matrix Transpose

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 $A_{ij} = A^T_{ji}$.

Note that inside `trans.c`, we have given you an example function that performs the transpose:

```
char trans_desc[] = "Simple row-wise scan transpose";
void trans(int M, int N, int A[N][M], int B[M][N])
```

You will need to write a similar function, called `transpose_submit()`, that computes the transpose of matrix A and saves the results in matrix B. Note that the size of A and B are given at runtime. We will evaluate your transpose function using matrices of different sizes (32×32, 64×64, and 61×67).

Your access to local variables (M, N, and your own local variables) will not result in recorded cache misses. This allows you to focus on optimizing matrix access. **However, you must not use arrays on the stack or heap.** That is, inside the transpose function, you cannot declare local array variables or use `malloc` to obtain extra space.

Important: Please do NOT change the description of `transpose_submit`. The description tells the test program to evaluate that function for credit.

Hint: Sometimes you may want to test multiple transpose functions and compare their performance. In `trans.c`, you can declare a number of your own transpose functions. As long as you register your function (see the end of `trans.c` for examples on how to do this), then the test function will evaluate it and print the number of cache misses.

For every function you register, you should also provide a short description. This string should be short and not contain newline characters. The test program prints this string for your convenience.

Note: If you use C0 contracts in your code, you should use `test`, the executable with contracts disabled, when evaluating your code's performance.

5 Evaluation

This section describes how your work will be evaluated. The full score for this lab is 53 points:

- Part (a): 21 Points
- Part (b): 27 Points
- Style: 5 Points

5.1 Evaluation for Part (a)

For Part (a), we will run your cache simulator using different parameters (s, b, E), and on different traces. For each test case, outputting the correct number of cache hits, misses AND evictions will give you full credit for that test case. Each of your reported number of hits, misses and evictions is worth 1/3 of the credit for that test case. That is, if a particular test case is worth 3 points, and your simulator outputs the correct number of hits and misses, but reports the wrong number of evictions, then you will earn 2 points. There are six test cases, worth 3, 3, 3, 3, 3 and 6 points respectively. The driver runs the following test cases:

```
  linux> ./cachesim -s 1 -E 1 -b 1 -t traces/yi2.trace
  linux> ./cachesim -s 4 -E 2 -b 4 -t traces/yi.trace
```

```
linux> ./cachesim -s 2 -E 1 -b 4 -t traces/dave.trace
linux> ./cachesim -s 4 -E 2 -b 4 -t traces/simple_trans.trace
linux> ./cachesim -s 5 -E 1 -b 5 -t traces/simple_trans.trace
linux> ./cachesim -s 5 -E 1 -b 5 -t traces/long_trace.trace
```

You can obtain the correct answer for these test cases using the reference cache simulator.

5.2 Evaluation for Part (b)

Your score for Part (b) consists of 1 correctness point and 26 performance points.

5.2.1 Correctness

We will use the function `transpose_submit` you provide to transpose matrices of different sizes.

The transpose functions is first evaluated for its correctness. The function will be evaluated using the reference cache simulator, with the following assumptions:

- There is a single 1KB direct mapped cache with 32-byte blocks ($E = 1$, $C = 1024$, $B = 32$). There are 5 set index bits ($s = 5$) and 5 block offset bits ($b = 5$).

The reference cache simulator will be invoked on trace files generated from your transpose function using the following arguments:

- `linux> ./cachesim.example -s 5 -E 1 -b 5 -t <trace file name>`

Your transpose function earns 1 point if it is correct. Only one point is given for correctness because we have already provided you with a correct transpose function.

5.2.2 Performance

If your function is incorrect, it earns 0 points. Depending on the number of misses, your performance score scales linearly. Your performance score is based solely on the following cases:

1. Transposing 32×32 matrices:
8 points if your function causes less than 300 cache misses.
0 points if your function causes more than 600 cache misses.

2. Transposing 64×64 matrices:
8 points if your function causes less than 1300 cache misses.
0 points if your function causes more than 2000 cache misses.

3. Transposing 61×67 matrices:
10 points if your function causes less than 2000 cache misses.

0 points if your function causes more than 3000 cache misses.

You do NOT get extra credit for reducing the number of misses further, although of course there are bragging rights at stake on the Autolab scoreboard.

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.

5.3 Autograding Your Work

5.3.1 Working on Part (a)

In order to help you to develop your own simulator, we have provided you with a reference binary called `cachesim.example`. You can execute `./cachesim.example` to see its usage and command line arguments. we have also provided you with a set of traces that are located in your handout directory in the folder `traces`.

Hints:

- It is better to debug your simulator by starting with very small traces such as `traces/dave.trace`.
- Each load/store can cause at most one cache miss. Data modify (M) is treated as a load followed by a store to the same address. Hence, an M operation can result in two cache hits, or a miss and a hit plus a possible eviction.
- Don't forget to add a comment at the beginning of the `cachesim.c` file with your name and Andrew ID.

5.3.2 Working on Part (b)

The test program:
We provide you with a test tool that you can use to evaluate the performance of your tranpose function. For example:

```
linux> ./test -M 32 -N 32
```

where M and N are the matrix dimensions. The test program will determine:

- Correctness of your `cachesim.c` simulator
- Correctness of your transpose function in `trans.c`
- Number of cache misses of your transpose function

For grading, we will run the test program on three matrices::

```
linux> ./test -M 32 -N 32
linux> ./test -M 64 -N 64
linux> ./test -M 61 -N 67
```

Generated traces:
In addition, you can use the trace files that the `test` tool will generate at every run: `trace.full` and `trace.filtered`. The first one has everything that is generated by Valgrind, including instructions addresses. You might need it only if you want to find the exact instruction that cased a hit or a miss in your transpose function.

The most useful trace is the second one. It includes only those memory addresses that correspond to your transpose function call. You can use `trace.filtered` to analyze your hit/miss ratio the same way as you did in Part (a). It can be helpful in improving your transpose function performance. The example of a potential use:

```
linux> ./cachesim -s 5 -E 1 -b 5 -t trace.filtered
```

The driver program:
We have also provided you with the driver program `./driver.py`. This is the same program that Autolab uses when it autogrades your handin. To run the driver, type:

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

It is a simple wrapper that makes all three calls of the test tool, as described above, and then submits an unofficial score to Autolab.

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

does the same thing but skips the unofficial submission to Autolab.

5.4 Coding Style

There are 5 points for coding style. Style guidelines can be found on the course website.

6 Handing in Your Work

Executing the command ”make”, besides compiling your code, creates a file named `andrewID_handin.tar`. This is tarball containing your `cachesim.c` and `trans.c` files. To hand in your work, you will need to upload this tarball (and only this tarball) to Autolab.

Note that grading this lab takes a few seconds, so the score might not appear immediately after you upload your work.