15-213/18-243, Fall 2010
Lab Assignment L4: Cache Lab
Assigned: Tuesday Sep. 28
Due: Tuesday, Oct. 12, 11:59PM
Last Possible Time to Turn in: Friday, Oct. 15, 11:59PM

1 Logistics

This is an individual project. All handins are electronic.

Please contact the 15-213 staff list (`15-213-staff@cs.cmu.edu`) for questions about assignment.

2 Overview

This lab tests your understanding of memory cache by asking you to create a cache simulator. The simulator is like a cache, except that it does not actually store the memory contents—it only records the number of hits, misses and evictions. You need to understand the following concepts in order to implement the cache simulator:

- cache hit, miss, eviction
- block offset bits
- set index bits
- associativity
- replacement policy

After implementing a cache simulator, we ask you to write code to compute the transpose of a matrix. You will need to think of different ways to minize the number of cache misses.

3 Downloading the assignment

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

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

you can retrieve the `cachelab-handout.tar.gz` file by selecting "Cachelab - Download Lab Materials" and then hitting the "Save File" button.

Start by copying `cachelab-handout.tar.gz` to a protected directory in which you plan to do your work. Then give the command "`tar zxvf 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: A skeleton for the cache simulator
- trans.c: A file containing code to transpose a matrix
- Makefile: Used by 'make' to generate binaries.
- driver: A binary you can use to evaluate your work
- autograde.py: A script that grades your work and submit unofficial score to autolab
- traces: A folder containing a few reference traces
- andrewID_handin.tar.gz: A tarball containing cachesim.c and trans.c you can upload to autolab for credit (updated by 'make')

To compile code in this lab, simply do:

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

Note that "make" also creates a tarball containing cachesim.c and trans.c you can submit to autolab.

WARNING: Do not let the Windows WinZip program open up your `.tar.gz` 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!).

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 memory access of the execution of a given binary. As an example, you can execute:

```
linux> valgrind --log-fd=1 --tool=lackey --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 of the trace is "Operation Address,Size". In the operation field, "I" stands for instruction load; "L" stands for data load; "S" stands for data store; and "M" stands for 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, and simulate a cache and output the number of cache hits, misses and evictions.

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

```
-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 textbook CSAPP2e page 597. Your cache simulator should use the LRU (least recently used) replacement policy for evictions.

We have provided you with the binary of a reference cache simulator called `cachesim.example`. You can execute `./cachesim.example` to see its usage and command line arguments.

Your job for Part (a) is to fill in the empty `cachesim.c` file. To help you get started, we have included the code to parse commandline options.

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, at the end of your cache simulator, you must call the function printCachesimResults() with the number of cache hits, misses and evictions. This sends the results to the driver for evaluation. You will find it at the end of main() in `cachesim.c`, please do not delete that line or you will not 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 cross block boundaries. By making this assumption, you can ignore the request sizes in the Valgrind traces.

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 already included a template that you can mimic.

4.2.1 Matrix Transpose

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

Note that inside trans.c, we have given you an example function that does 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 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 by 32, 64 by 64, and 61 by 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 driver 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 with the driver (see the end of `trans.c` for examples on how to do that), the driver will print the number of cache misses. Please use the same function signature as `trans()`.

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

5 Evaluation

This section describes how your work is 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 simulat

5.2 Evaluation for part (b)

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

The transpose functions is first evaluated for its correctness. The function will be evaluated using the refererence 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>`

5.3 Scoring for Part (b)

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

5.3.1 Correctness

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.

5.3.2 Performance

If your function is incorrect, it earns 0 points. Depending on the number of misses, your performance score scales linearly.

1. Transposing 32*32 matrices:
8 points if your function causes less than 350 cache misses.
0 points if your function causes more than 1150 cache misses.

2. Transposing 64*64 matrices:
8 points if your function causes less than 1850 cache misses.
0 points if your function causes more than 4700 cache misses.

4. Transposing 61*67 matrices:
10 points if your function causes less than 2000 cache misses.
0 points if your function causes more than 4400 cache misses.

You do NOT get extra credit for reducing number of misses further.

5.4 Coding style

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

5.5 The driver

You can always run `./driver -M 32 -N 32` (M and N are the matrix dimensions the driver use) to determine:

1. Scoring of your cachesim simulator
2. Correctness of your transpose function
3. Number of cache misses of your transpose function

For grading, we will run:

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

Alternatively, you can just run

```
linux> ./autograde.py
```

Note that "./autograde.py" also submits an unofficial score to autolab.

6 Handing in your work

Executing the command "make", besides compiling your code, creates a file named andrewID_handin.tar.gz. This is tarball containing your cachesim.c and trans.c. 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 while, so the score does not immediately show up after you upload your work. The score will show up a few minutes after you upload the tarball to Autolab.