Lab 4: Cache Geometries

Assigned	Wednesday, May 13th, 2015
Due Date	*Friday, May 22st, 2015 at 5:00pm*
Files	lab4.tar.gz
Submissions	Submit a PDF file containing your answers and your modfied cache-test.skel.c file here.

Part I: An Experiment in C and Java

Assignment Goals

- Evaluate a claim made in the lecture slides that two seemingly equivalent ways of writing a program can have vastly different performance.
- Get a feel for the relative performance of Java and C.
- Get a feel for the effectiveness of turning on C compiler optimizations.

Overview

Let's test the claim that understanding the memory hierarchy can be useful in writing efficient programs. An example in the first-day lecture slides said that interchanging two loops has no effect on the correctness of the results, but can give a*21x*difference in performance. Let's see about that.

Here's the important part of the code. It computes exactly the same thing no matter which of the two loops is outermost.

```
int rep;
int i, j;

// ...

   for (i = 0; i < 2048; i++) {
      for (j = 0; j < 2048; j++) {
         // src[i][j] = i * rep;
         dst[i][j] = src[i][j];
      }
   }
```

You will download a set of three tiny programs—one in C and two in Java—that contain those loops. You'll compile them and time how long it takes them to run. For the C program, you'll compile both with and without compiler optimizations enabled, so in total you will have four programs to compare at a time

(two Java programs + one C program compiled two ways).

You will do this several times, making small modifications to see what differences they make—how the choice of language affects performance and how effective the compiler can be at optimizing your code when you:

- interchange the order of the `i`and`j`loops
- uncomment the commented line
- change the size of the array being copied from 2048 x 2048 to 4096 x 4096 (change both the array size and the bounds of the loop that copies the arrays).

You'll run each version of the code and measure how long it takes to complete. With all the permutations (4 executables x 2 loop orderings x 2 commented/uncommented line versions x 2 array sizes), that's 32 versions. (It will be easy—just read all the way through these instructions first.)

You'll then turn in a short document, described below, in which you summarize your test results and answer a few questions.

Details

Downloading

Fetch the files, which are provided as a `tar`archive:lab4.tar.gz. Save them to a directory in which you want a new directory (containing the files) created.

Now issue the command`tar xzf lab4.tar.gz`. That will un-archive the files, creating directory`lab4`. In that directory you will find these files (as well as files for part two):

File	Description
cacheExperiment.java	Rows 'Java' in your tables of test results (see below)
cacheExperimentInteger.java	Rows 'JavaInteger' in your tables
cacheExperiment.c	Rows 'C' and 'Optimized C' in your tables
run.pl	See“Automating”below

Compiling

To compile the C program without optimizations,`cd` to the`lab4`directory and type:

```
gcc -Wall cacheExperiment.c
```

That produces an executable named`a.out`. To compile the program with optimizations, type:

```
gcc -Wall -O2 cacheExperiment.c
```

(that is the capital letter o, not the number zero), which also produces an executable called`a.out`(overwriting the previous one).

To run `a.out`, you would type `./a.out`. (**Note:** You don't actually want to do this. See the next heading about obtaining timings.)

To compile `cacheExperiment.java` type:

```
javac -Xlint cacheExperiment.java
```

which produces `cacheExperiment.class`. Do the same thing for the other Java programs. To run it, type:

```
java -Xmx640M -cp . cacheExperiment
```

(Again, this is a command you need to time, so read on.)

Timing

Note: On the CSE VM, the command `/usr/bin/time` was not installed by default. Typing "time" at the bash command line will instead run bash's built in time command (which is different). To use `/usr/bin/time` on the VM you will need to install it by typing this at the command line:

```
sudo yum -y install time.
```

On Linux, you can measure the CPU time consumed by any execution using the `time` program. For example:

```
$ /usr/bin/time ./a.out
0.12user 0.03system 0:00.16elapsed 95%CPU (0avgtext+0avgdata 66704maxresid
0inputs+0outputs (0major+8287minor)pagefaults 0swaps
```

This executes the command (`./a.out`) and then prints information about the resources it consumed. (Type `man time` to obtain more information about the time program and ways to format its output.)

The only information we'll use is the user time ('0.12user', meaning 0.12 seconds of CPU time consumed while not in the operating system) and the system time ('0.03system', meaning 0.03 CPU seconds spent by the operating system doing things for this application). *The measured time we want is the sum of those two.* For this example, the measured time would be 0.15 seconds.

Measured times are likely to vary quite a bit from one run to the next, even without changing anything. (This course will explain some of the reasons why.) Note that all the programs wrap the two array-copying loops with another loop that causes the copy to be performed 10 times. One goal of that is to reduce the amount of variability in the measurements.

Automating

The distribution includes an optional script, `run.pl`, that automates some of the chore of running the four executables and gathering measurements. To run it, type `./run.pl`. It compiles each of the source files (and `cacheExperiment.c` twice; with and without optimizations), runs each with the `time` command, and reports the sum of the user and system times.

`run.pl`should work in most environments (including the CSE virtual machine). It should work for you, but it is an optional (and unsupported) tool.

So, to summarize:

1. Compile and measure each of the Java implementations as they come in the distribution. Compile and measure the C program with and without optimizations.
2. Edit each source file to uncomment the assignment to array`src`. Re-compile and re-measure.
3. Edit to switch the order of the `i`and`j`loops. Recompile and re-measure.
4. Edit to re-comment out the statement assigning to array`src`(with the `i`and`j`loops still reversed). Re-compile and re-measure.
5. Edit to put the loops back in the original order. (At this point the code is the same as it was when you first fetched it.) Change the code to copy an array of size 4096 x 4096 (change both the size of the arrays and the loop bounds). Then repeat steps 1–4 above.

Test Results

Collect your results in a short PDF documentwith the following sections:

1. **The Test System**
   - A short string describing the system your ran on (e.g., “my Mac laptop” or “the CSE home VM on my Windows laptop” or “lab Linux workstation”).
   - What the CPU is on that system. You can obtain that on any Linux system by issuing the command`cat /proc/cpuinfo`. Give us the model name, as listed.
2. **Test Results**Four tables of numbers giving the measured CPU time consumed when executing each of the four executables under the different configurations. Each table should look like this. (It doesn't have to be exactly this, to every detail of formatting, but please keep your information in the same order; it makes reading 130 copies of these tables easier if they're all laid out the same way.)

Array Size	Performing`src`assignment?	App	Time with`i`then`j`	Time with`j`then`i`
2048	No	Java		
		JavaInteger		
		C		
		Optimized C		

3. **Q&A**Answer these questions:
   1. What are the source code differences among the two Java implementations?
   2. Pick a single pair of results that most surprised you. What is it about the results that surprised you?*(That is, from the 32 measurements you collected, pick one pair of measurements whose relationship is least like what you would have guessed.)*
   3. *[Optional extra credit]*None of these programs appear to actually do anything, so one is tempted to optimize them by simply eliminating all code (resulting in an empty`main()`). Is that

a correct optimization? Related to that, try compiling this C program, with and without optimization, and then time running it:

```
#include <stdio.h>

#define SIZE 1000000

int main() {
   int i, j, k;
   int sum = 1;

   for (i = 0; i < SIZE; i++) {
      for (j = 0; j < SIZE; j++) {
         for (k = 0; k < SIZE; k++) {
            sum = -sum;
         }
      }
   }

   printf("hello, world\n");

   return 0;
}
```

Now replace the `printf` line with

```
printf("Sum is %d\n", sum);
```

and compile/run unoptimized and optimized.

Part II: Inferring Mystery Cache Geometries

Overview

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.

This lab should be done on a 64-bit machine. Use attu, the CSE VM, the lab computers, or your own personal 64-bit computer.

Instructions

Specifically, each of these "processors" is provided as an object file (.o file) against which you will link your code. See the file`mystery-cache.h` for documentation of the function interface that these object files export. 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. Some of the provided object files are named with this information (e.g.`cache_65536c_2a_16b.o`is a 65536**Byte**capacity, 2-way set-associative cache with 16 **Byte**blocks) to help you check your work. There are also 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.

You will need to complete this assignment on a Linux machine with the C standard libraries (e.g. the CSE VM, attu). All the files you need are in`lab4.tar.gz`. To extract the files from this archive, simply use the command:

```
tar xzf lab4.tar.gz
```

and the files will be extracted into a new subdirectory of the current directory named*lab4*. The provided`Makefile`includes a target`cache-test`. To use it, set`TEST_CACHE`to the object file to link against on the command line - i.e. from within the lab4 directory run the command:

```
make cache-test TEST_CACHE=cache_65536c_2a_16b.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:

```
./cache-test
```

and it will print the results to the screen.

Your Tasks

Complete the 3 functions in `cache-test-skel.c`which have `/* YOUR CODE GOES HERE */`comments in them.

Additionally, determine the geometry of each of the four mystery caches and list these in a comment, along with your name, at the top of your modified`cache-test-skel.c`.

Tips

Note that the exact style of `for`loops with which you may be familiar from Java was not allowed in C until a

later standard. Instead of writing:

```
for (int i = ...; ...; ...) { ... }
```

write this instead, declaring your loop variable outside the loop header (and at the top of your function body):

```
int i;
...
for (i = ...; ...; ...) { ... }
```

There is also a flag to tell the compiler to allow the first version, but the second version will work either way.

Submitting Your Work

Part I: Please turn in a PDF file containing your answers to Part I to theCatalyst Drop Box for this assignment.*We will not accept submissions that are not in PDF format*.

Part II: Submit your modified version of`cache-test-skel.c` to theCatalyst Drop Box for this assignment.