1 Lab 11

Date: Oct 31, 2019

This document first describes the aims of this lab. It then provides necessary background. It then describes the exercises which need to be performed.

1.1 Aims

The aim of this lab is to familiarize you with various methods for measuring the timing performance of programs under Unix. After completing this lab, you should be familiar with the following topics:

- The use the `time` command built-in to most Unix shells.
- The use of basic block counting to find out how often a line of code was executed.
- Appreciation for the different factors which can affect the time performance of a program.

1.2 Background

The factors which affect the time performance of a program can include the following:

Algorithmic Choices The choice of algorithms used by a program can make a dramatic change in the time performance of a program. For example, a program may see a drastic improvement in performance if its main algorithm which takes time which increases quadratically with the size of its input is replaced by one whose time simply increases linearly with the size of its input.

System Interactions Many programs can run without being affected by the parameters of the computer system on which they are running, but other programs may not be as lucky. Computer system parameters like the organization of memory or the use of pipelining can change the time performance of such programs.

Micro Changes If a construct is used frequently enough by a program then replacing it with a slightly more efficient construct can improve the time performance of a program.

Donald Knuth: *Premature optimization is the root of all evil (or at least most of it) in programming.* Hence before attempting any optimization of code, it is necessary to measure the time performance of a program and only optimize if necessary and only optimize code which is a bottleneck.

Under Unix, the tools available to analyze the time performance include the following:

`time` A command built-in to most Unix shells. Prints out the amount of time needed to execute a command. The printed time components include the the real (AKA *wall-clock*) time, system time (the amount of time spent executing code in the kernel) and user time (the amount of time spent executing code in user-space).

The `time` report for jobs which run on computer systems where a single CPU is being shared among many different users will often show that the wall-clock time is much greater than the sum of the user and system times. This is expected on such a system.

OTOH, if a computer system has multiple CPUs, then it is possible that the sum of the system and user time is greater than the wall-clock elapsed time.

`times()` This is a C library call which returns data-structures representing somewhat more detail than that returned by the shell `time` command. This lab will not explore this further.

`gcov` When a program is compiled with special options, then each run creates a special binary file which records how many times each *line* in the program was executed. This tool analyzes the generated binary file to produce an annotated listing of the original source code documenting how many times each line was executed.

Besides allowing insight into where a program is spending most of its time (as represented by line count), this tool is also invaluable as a test coverage tool to ensure that a set of tests minimally exercises each line of code.

`gprof` This tool also measures the time spent by different portions of a program execution. The approach often used is PC sampling: while the program is running the PC is sampled every few milli (or maybe nano) seconds. The sampled value of the PC is translated back into a line number in the program. This lab will not be using this tool but it is well worth a look.

1.3 Exercises

1.3.1 Starting up

Use the startup directions from the earlier labs to create a `work/lab11` directory and fire up a terminal whose output you are logging using the `script` command. Make sure that your `lab11` directory contains a copy of the files directory.

For the exercises which follow it is **imperative that you be running bash**.

1.3.2 Exercise 1: Use of the time Command

The `time` command which is built-in to many Unix shells is simple to use: simply prefix a normal shell command by the word `time`. For example:

```
$ time ls ~/cs220/labs/lab11/files
coverage  int-search  matmul-cache  parity

real    0m0.002s
user    0m0.000s
sys     0m0.000s
```

You should get output with a very similar format. If not, verify that you are running the `bash` shell. Make sure you are not running the `time` program (available as `/usr/bin/time`) which produces output in a different format.

As mentioned in the background section, the command reports the amount of real or elapsed time (AKA *wall-clock* time), amount of time spent in kernel space and amount of time spent in user space.

Try timing a command which produces more interesting output:

```
$ time sleep 5
```

Note the large difference between the elapsed time and the other times.

Time a command which searches a large directory:

```
$ time wc -l ‘find /etc -type f 2>/dev/null‘ 2>/dev/null | tail
```

[The command within the backquotes `find`'s all files (`-type f`) within the `/etc` directory, discarding any errors (`2>/dev/null`). Those results are counted using `wc -l` (the results of the backquotes `find` are treated as though they were typed in to the `wc -l` command), with errors again being discarded. The last portion of the results are displayed using the `tail` command.]

Notice here that the system time is non-trivial and measures the time used for starting and synchronizing the multiple sub-processes involved in the above command.

Run it again; you may get a much faster result if your system has cached the necessary files in its kernel buffers.

Try timing a couple more non-trivial shell commands.

1.3.3 Exercise 2: Speeding Up Code using Assembly

In the previous lab we explored using the parity flag (available on most architectures) to evaluate the parity of a word. In this exercise, we will use `time` to evaluate how much of a performance boost this technique provides.

Change over to the parity directory and look at the files: main.c is simply a driver program which uses its command-line arguments to figure out how many parities it should evaluate over random integers; parity-c.c evaluates parity using the ones-counting technique covered earlier in the course and finally parity-s.c uses inline assembly.

Build all programs by typing `make`. Then run:

```
$ ./parity-c -d 10
```

followed by:

```
./parity-s -d 10
```

The `-d` flag is a debug flag and prints out the randomly generated integers (limited to one byte) and the computed parity. Both of them should produce the same results even though the first program uses the counting-ones technique to evaluate parity whereas the second program evaluates parity by accessing the parity flag using inline assembly.

Now time the two programs (with the `-d` flags turned off). As both programs are pretty fast, you may need to increase the number of tests to around 100 million or so, depending on the system you are running on. If your results are similar to mine, you should observe that the version which uses inline assembly is about twice as fast as the version which is written in pure C. Also observe that almost all the time is spent in user space as the program uses minimal kernel services.

1.3.4 Exercise 3: Observing Cache Effects

Change over to the matmul-cache directory. The main.c file is a simple driver program which accesses the command-line arguments to get the size of a (square) matrix and the number of tests to run. It generates two random double matrices and then calls a matrix multiply routine to multiply them together.

Look at the simple-matmul.c file: it contains the classic code for matrix multiplication. Recall that matrices in C are laid out by row; hence in the inner-most loop

```
for (int k = 0; k < n; k++) {
  c[i][j] += a[i][k]*b[k][j];
}
```

the entries in `a[][]` are being accessed sequentially in memory within a row, but the entries in `b[][]` are not being accessed sequentially in memory (in fact, they are being accessed by column so that successive accesses will have wildly different memory addresses).

Recall that modern computer systems use a hierarchy of memory systems with a small amount of expensive high-speed *cache* memory and a larger amount of cheaper (but slower) main memory. Assuming that the program has good *locality of reference* the overall memory speed should be close to that of the high-speed cache at overall cost close to the cost of the main memory.

Unfortunately, the columnar access to `b[][]` in the inner-loop of the matrix multiply destroys locality of reference. If the matrix is sufficiently large, then it is likely that successive columns of `b[][]` are not in the cache and hence the system will need to fetch the `b[][]` entries from the slow main memory resulting in poor overall performance.

However, there is a simple fix which avoids the problem. Simply transpose the matrix `b[][]` before the multiply begins and then within the inner loop, instead of accessing `b[][]` column-wise access the transposed matrix row-wise. The row-wise access should take care of the cache problems.

Look at the transpose-matmul.c file and notice the changes within `matrix_multiply()`. However, the `matrix_transpose()` function has been left incomplete: please add in the code to make its meet the given specification.

First make sure that your fix is correct: build all the programs by typing `make`. Then simply run the simple matrix multiply on a small matrix using something like:

```
$ ./simple-matmul 4 1
```

The result matrix should be printed out.

Then run the transposed matrix multiply:

```
$ ./transpose-matmul 4 1
```

If your `matrix_transpose()` function is correct, the results should match. If not, iterate until you fix that function.

Now run:

```
$ time ./simple-matmul 1500 1
```

followed by

```
$ time ./transpose-matmul 1500 1
```

Depending on your system, you may or may not see that the second solution is faster.

However, as you increase the size of the matrix, you should definitely see the difference in performance:

```
$ for s in `seq 1000 500 2500`; \
> do \
>   echo -n "*** Size $s: simple"; time ./simple-matmul $s 1 ; echo ; \
>   echo -n "*** Size $s: transpose"; time ./transpose-matmul $s 1 ; echo ; \
> done
```

You should definitely see a large difference in performance as the matrix size increases.

1.3.5 Exercise 4: Algorithmic Changes Win

Change over to the int-search directory. The driver program in main.c creates a random integer array of size specified by the first command-line argument and runs a number of searches for elements in that array specified by the second command-line argument.

Two implementations of the search are provided:

1. An incomplete *linear search* implementation is provided. It should search sequentially through the array looking for the specified element: if found, it should return its index, if not found (the entire array has been searched) it should return a negative value.

   If there are `n` elements in the array, on average `n/2` elements will be compared for a successful search but `n` elements must be compared if the search proves ultimately unsuccessful.

2. A complete *binary search* implementation is provided. This provides identical functionality to the linear search but uses the `bsearch()` library function discussed in class.

   If there are `n` elements in the array, a maximum of $\lceil \log_2 n \rceil$ comparisons are necessary for searching for an element.

As `n` increases, the difference in performance can be dramatic.

To measure the increase, first add in the code for the linear search in linear-search.c. Build all the programs by typing `make`. Test your linear search on a small example by typing:

```
$ ./linear-search 100 1
```

If it fails, you should get an assertion failure.

Now time both linear search and binary search:

```
$ time ./linear-search 2000 2000
```

```
$ time ./binary-search 2000 2000
```

You should see that the binary search is dramatically faster.

1.3.6 Exercise 5: Ensuring Test Coverage

Change over to the coverage directory and build the `coverage` program by typing `make`. The `coverage` program is a silly program which reads 3 `int`'s `a`, `b` and `c` at a time from standard input and then calls a `compute()` function which computes some function `compute(a, b, c)`. The code for `compute()` contains a lot of conditional code.

The program has been compiled with special options (see the `Makefile`). When compiled, it will produce a `coverage.gcno` binary file which records information about the control flow in the program. Each run of the program generates a binary file `coverage.gcda` which contains information about which lines were executed during that run.

Run the program by typing `./coverage`, provide 3 integers like `200 2000 2999` and then type ^D to indicate EOF. You should see the generated file in your directory.

Now run the program `gcov` on the generated file: i.e. `gcov coverage.gcda`. It should create a file `coverage.c.gcov` which is the source file with each line annotated with a count of the number of times that line was executed. Look at `coverage.c.gcov` to see your results.

Now look at `coverage.c` and generate sufficient test data to ensure that every line in `compute()` is covered by your test data. Because of the regular branching structure within `compute()` it should be clear that 8 sets of `a`, `b`, `c` values should be sufficient.

Run `gcov` to validate that your tests do indeed exercise each line of `compute()`.

1.4 References

- Ulrich Drepper, *What Every Programmer Should Know About Memory*. This is the source of the cache example.
- *gcov docs*.