Performance Engineering of Software Systems September 21, 2010
Massachusetts Institute of Technology 6.172
Professors Saman Amarasinghe and Charles E. Leiserson Handout 6

Profiling

Project 2-1

Evaluating Performance Via Profiling

Last Updated: September 22, 2010

Generally, when you are concerned about the performance of a program, the best approach is to implement something is correct and then to evaluate it. In some cases, many parts of this initial implementation (or even the entire thing) may be adequate for your needs. However, when you need to improve performance, you must first decide where to focus your efforts. This is where profiling becomes useful.

Profiling (and dynamic binary instrumentation) let you examine how different aspects of a program perform in a variety of situations. In larger projects, profiling can help you identify where your program spends most of its execution time. In smaller pieces of code, it can help you figure out why your code isn't as performant as you hoped it might be and to correct any issues holding your code back.

1 Getting started

Introduction

In this part of the project, we want you to reverse engineer the provided programs (that is, figure out what they are doing) using your profiling tools, namely *gprof* and *perf events*. In particular, we'll be looking at an image rotation program, a variety of sorting programs, and a second look at profiling pentominoes. In the next part of the project, you'll reimplement these programs.

Getting the code

As with project 1, you will get the project code using git:

```
% git clone /afs/csail/proj/courses/6.172/\
    student-repos/project2.1/username/ project2.1
```

Using perf events

The Linux Perf Events subsystem uses a sampling approach to gather data about important hardware and kernel events, such as cache misses, branch misses, page faults, and context switches. The `perf` program, distributed in the linux-tools package, records data during profiling runs and displays the results in the terminal. We're only going to use the `stat` subcommand, which is invoked as `perf stat` *`my_command`*, and produces output like the following, if you run it on `make`:

```
$ perf stat make -j

    Performance counter stats for make -j:

    8117.370256  task clock ticks     #      11.281 CPU utilization factor
            678  context switches     #       0.000 M/sec
            133  CPU migrations       #       0.000 M/sec
         235724  pagefaults           #       0.029 M/sec
    24821162526  CPU cycles           #    3057.784 M/sec
    18687303457  instructions         #    2302.138 M/sec
      172158895  cache references     #      21.209 M/sec
       27075259  cache misses         #       3.335 M/sec

          Wall-clock time elapsed:   719.554352 msecs
```

You can choose specific events, such as L1-dcache-load-misses, with the `-e` option. You can see a full list of events by running `perf list`. For more examples of how to use perf, refer to the link below.
`https://perf.wiki.kernel.org/index.php/Perf_examples`

2 Image rotation

One of Snailspeed Ltd.'s larger consumer-facing services is the Snailsnap photo hosting service. Among other things, Snailsnap lets users rotate their photographs. Since this is by far the most popular image manipulation operation among users of the service (who are mostly professional photographers), any performance improvements that can be made to the image rotation program will significantly reduce the number of backend servers required by the service.

Each image is represented as a two-dimensional $n \times n$ matrix of 64-bit integers. Each integer contains four channels of 16 bits each; the channels represent the red, blue, green, and alpha values of the pixel, respectively. Each program creates a new $n \times n$ matrix which contains a rotated version of the original matrix.

While you should be able to come up with a more efficient solution, a naive one might look something like this.

- The matrix M is transposed, producing M^T.
- Row i of M^T is exchanged with row $n-1+i$.

In code, this simple solution would look something like this. The arguments **m1** and **m2** are the source and destination matrices, respectively, and **n** is the length of one side of each matrix. Since C only supports one-dimensional arrays (as pointers into memory), we use a common technique to produce what is effectively a two-dimensional array. Consider that a two-dimensional array can be mapped into a one-dimensional array by laying out consecutive rows end-to-end in memory (so that $a[n-1]$ is the last element of the first row, and $a[n]$ is the first element of the second row). Let $i = (y*n)+x$. Then, if b is the imaginary two-dimensional matrix and a is the underlying one-dimensional array, you can access the element $b[x][y]$ by referring to $a[i]$.

```
void rotate(const uint64_t* m1, uint64_t* m2, const int n)
{
  for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j)
      m2[(j * n) + i] = m1[(i * n) + (n - 1 - j)];
}
```

Keep in mind that if you reverse x and y, this mapping will change! (You'll be switching between a row-major and a column-major layout.)

2.1

Run each of the three image rotation programs with $n = 10,000$. You can do this for the nth program by typing

```
pnqsub ./rotateN 10000
```

Report the execution time of each program.

2.2

From scribblings that Harvey left in the margin of the "high society" section of the Boston Globe, you learn that `rotate1` is the naive implementation presented above, and that `rotate2` is exactly the same except that the order in which the loops are nested has been reversed.

For each of the two programs, show the order in which the elements of the two matrices are accessed. You should draw two diagrams, each consisting of two squares (one for **m1** and one for **m2**). Use arrows to depict the direction of memory accesses.

2.3

Draw an additional pair of diagrams for each of the two programs, showing how memory is accessed in the one-dimensional array that is being used to represent the two-dimensional matrices. In your diagrams, assume that memory is contiguous in the horizontal direction. Think about the difference in performance between the two programs; do these diagrams provide you with any additional insight?

2.4

Assuming that the matrices are laid out in a single, contiguous block of memory, how do you think the cache miss rates compare?

2.5

Verify your answer to the previous question by obtaining the exact L1 cache miss rates for both programs using `perf`. Run the command below for both binaries, and report the cache miss ratio and the cycles per instruction (CPI). The CPI is cycles divided by instructions, and the cache miss ratio is L1-dcache-load-misses value divided by L1-dcache-loads.

```
% pnqsub perf stat -e cycles -e instructions ./rotateN
% pnqsub perf stat -e L1-dcache-loads -e L1-dcache-load-misses ./rotateN
```

For CPI, the ideal value is about 0.25, since Core i7's can dispatch about 4 instructions a cycle, and lower is better. If you have lots of stalls, CPI may be above 1, meaning less than one instruction is retired per cycle. For cache miss rates, the ideal miss rate is zero, and lower is obviously better.

Now, profile `rotate3` and obtain its CPI and cache miss rate. Note that while the program's internal timer separates setup time from the execution time, the `perf` tool does not differentiate, and so the total cache reference and miss numbers are polluted by setup. Therefore, the miss rate may not be as low as you would expect it to be, even though the third program runs much more quickly.

2.6

Come up with a memory access pattern that might achieve the miss rate that you have observed. Draw the same diagrams that you drew for the first two programs to illustrate this memory access pattern.

Hint: You should construct an access pattern that minimizes the miss rate as much as possible.

Hint: In class, we discussed techniques for revisiting cache lines before they are evicted.

3 Sorting 64-bit integers

Profiling code can often give you valuable insight into how a program works and why it performs the way it does. In this exercise, you will be provided 4 unnamed sorting binaries (`sort1` to `sort4`) and be asked to infer what they do based on their performance characteristics. You may find it helpful to know that these binaries were compiled with a highly optimizing compiler designed to take advantage of architecture-specific features of the Intel Xeon processors in the cloud machines.

Note that we've provided `int32_sorts` and `int64_sorts` variants – these are all 64-bit programs, but sort arrays of 32-bit integers and 64-bit integers, respectively.

Hint: Feel free to use general reference materials if you think they will help you.

3.1

Compare the first three 64-bit sorting programs by running them for arrays of $1,000,000$ elements and comment on their timing output.

3.2

Now, repeat the test for the 32-bit sorts. How does the performance differ?

Did any of the results surprise you? Can you think of possible explanations for your observations?

3.3

Use `perf` to obtain figures for the execution time, the CPI, the L1 data cache miss rate, and the branch misprediction rate for each binary. For branch prediction, use the "branches" and "branch-misses" events. Explain what each of these is, and what it measures. How does each impact performance? Briefly comment on the results of your profiling.

3.4

Comment on the trends you see. With the data you have collected, you should be able to identify what type of sorting algorithm has been implemented in each of the first three programs; do so, and explain what led you to each conclusion. If necessary, you are welcome to collect additional data points.

3.5

Now we will examine the final sorting program, `sort4`. This program is designed to sort very small arrays quickly, so in order to obtain meaningful timing results we will have to run it a number of times. The second argument to each sort program can be used to control the number of times the sorting algorithm is run. Run the 64-bit version of each sorting algorithm for an array size of $n = 7$ for $10,000,000$ iterations.

```
pnqsub perf stat ./sort4 7 10000000
```

What do you notice?

3.6

Now, run `sort4` on input sizes of $n = 6$, 7, and 8. While doing so, use `perf` to inspect the L1 *instruction* cache miss rate, using the following command.

```
pnqsub perf stat -e L1-icache-misses -e L1-icache-loads ./sort4 n 10000000
```

What do you notice?

3.7

Disassemble `sort4` binary with the following command.

```
objdump -d sort4 > sort4.asm
```

Open `sort4.asm` with a text editor and find the beginning of the function named “b” by searching for the string “<b>”. This function sorts an array of three elements. You may also want to look at “c”, which is similar but sorts an array of four elements. Look at the size of these two functions; can you explain the results of the previous section?

How does this program sort arrays?

4 Profiling pentominoes

For this exercise, we’ll be using `gprof`. Unlike perf events, gprof uses static instrumentation to gather information about program execution. “Instrumentation” refers to bits of code that are inserted into the program to record data. By “static” instrumentation, we mean that is inserted into the program by the compiler. The tradeoff between event sampling and instrumentation is that sampling has low overhead and uses performance counters, while instrumentation is more precise but has different overhead characteristics.

To use `gprof`, we have modified the Makefile to pass the `-pg` flag to the compiler and linker when building your code, which instruments the binary. If you run `make`, it will print the command used to invoke the compiler, and verify that `-pg` is present. We also added `-fno-inline-functions` to prevent the compiler from inlining all of the helper functions into `solve_internal`.

When you run your program, the instrumentation will count the number of calls to each function and take timing and call stack samples. Recording a backtrace and checking a timer on every function call and return is too expensive, so gprof uses a sampling based approach to gather this data, while the call count data is precise. When the program exits, the profiling data is dumped to a gmon.out file in the current directory.

4.1

Gather and display the flat profile and callgraph from the instrumented pentominoes binary by running

```
% ./pentominoes -b ’0,0 1,1 2,2 3,3’ -n 1000
```

```
% gprof ./pentominoes
```

NOTE: You may have to scroll up to see the gprof output.

4.2

Suppose you could choose to optimize `fill_piece` to run 2x faster. Based on the data you have collected, what would be the new runtime?

4.3

Perform the same computation for the `can_fill_piece` routine.

5 Turn-in

When you’ve written up answers to all of the above questions, turn in your writeup by uploading it to Stellar.