Lecture 20 - Basic Profiling

ECE 459: Programming for Performance

Jon Eyolfson

University of Waterloo

February 27, 2012

Reminder

Midterm

- **Date:** This Friday
- **Time:** 6:30 PM
- **Location:** RCH 105 (A-K), RCH 110 (L-Z)
  - Organized by last names

Content

- Closed-book
- Simple calculators (no other aides)
- 4 Questions
  - Definitions (pick two of three)
  - Calculations (Amdahl's/Gustafson's law)
  - Data races/thread-safety
  - Dependencies

Style

- Mostly consistent with last year
- Content is mainly from lectures 1-7 (although future lectures have some better explanation, i.e. thread-safety in lecture 15)
- I touched on critical paths in lecture 7, I'll mention it again

Preparation

- Friday's tutorial time will be open office hours here
- 1:30 PM in DWE 3522
- As always, you can e-mail me or TAs to set up office hours

Introduction

- So far we've been looking at small problems
- We have to **profile** to see what is taking up execution time in a large program
- Two main outputs:
  - Flat
  - Call-graph
- Two main data gathering methods:
  - Statistical
  - Instrumentation

Outputs

Flat Profiler

- Only computes the average time in a particular function
- Does not include anymore information such as: callee's

Call-graph Profiler

- Computes the call times
- Frequency of function calls
- Call graph, showing what called the function

Data Gathering

Statistical

- Mostly done by taking samples of the system state
- Every 2ns, check the system state
- Will have some slowdown, but not much

Instrumentation

- Add additional instructions at specified program points
- You can do this at compile time or run time (expensive)
- Also, either manually or automatically
- Like conditional breakpoints

Guide

For any large software projects you should:

- Write clear and consise code, not trying to do any premature optimizations (focus on correctness)
- Profile to get a baseline of your performance
  - Allows you to easily track any performance changes
  - Allows you to re-design your program before it's too late
- Focus your optimization efforts on the code that matters

Things to Look For

- Time is spent in the right part of the system
- Majority of time should not be spent in any error-handling, non-critical code or exceptional cases
- Time is not unnecessarily spent in the operating system

Introduction

- Statistical based with some instrumentation for calls
- Runs completly in User-space
- Only requires a compiler

Usage

- Use the `-pg` flag with `gcc` when compiling (also linking)
- Run your program as you normally would
  - Your program will now create a `gmon.out` file
- Use gprof to interpret the results `gprof <executable>`

Example

- A program that has 100 million calls to two math functions

```
int main() {
    int i,x1=10,y1=3,r1=0;
    float x2=10,y2=3,r2=0;

    for(i=0;i<100000000;i++) {
        r1 += int_math(x1,y1);
        r2 += float_math(y2,y2);
    }
}
```

- Looking at the code, we have no idea what takes longer
- Probably would guess floating point math taking longer
- Overall, silly example

Example (Integer Math)

```
int int_math(int x, int y){
    int r1;
    r1=int_power(x,y);
    r1=int_math_helper(x,y);
    return r1;
}

int int_math_helper(int x, int y){
    int r1;
    r1=x/y*int_power(y,x)/int_power(x,y);
    return r1;
}

int int_power(int x, int y){
    int i, r;
    r=x;
    for(i=1;i<y;i++){
        r=r*x;
    }
    return r;
}
```

Example (Float Math)

```
float float_math(float x, float y) {
    float r1;
    r1=float_power(x,y);
    r1=float_math_helper(x,y);
    return r1;
}

float float_math_helper(float x, float y) {
    float r1;
    r1=x/y*float_power(y,x)/float_power(x,y);
    return r1;
}

float float_power(float x, float y){
    float i, r;
    r=x;
    for(i=1;i<y;i++) {
        r=r*x;
    }
    return r;
}
```

Flat Profile

- When we run the program and look at the profiling data, this is the first thing we see

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ns/call  ns/call  name
 32.58      4.69     4.69 300000000    15.64    15.64  int_power
 30.55      9.09     4.40 300000000    14.66    14.66  float_power
 16.95     11.53     2.44 100000000    24.41    55.68  int_math_helper
 11.43     13.18     1.65 100000000    16.46    45.78  float_math_helper
  4.05     13.76     0.58 100000000     5.84    77.16  int_math
  3.01     14.19     0.43 100000000     4.33    64.78  float_math
  2.10     14.50     0.30                             main
```

- One function per line
- **time:** the percent of the total execution time in this function
- **self:** seconds in this function
- **cumulative:** addition of this function plus any above in table

Flat Profile

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ns/call  ns/call  name
 32.58      4.69     4.69 300000000    15.64    15.64  int_power
 30.55      9.09     4.40 300000000    14.66    14.66  float_power
 16.95     11.53     2.44 100000000    24.41    55.68  int_math_helper
 11.43     13.18     1.65 100000000    16.46    45.78  float_math_helper
  4.05     13.76     0.58 100000000     5.84    77.16  int_math
  3.01     14.19     0.43 100000000     4.33    64.78  float_math
  2.10     14.50     0.30                             main
```

- **calls:** number of times this function was called
- **self ns/call:** just self nanoseconds / calls
- **total ns/call:** average time of function execution, including any other calls the function makes

Call Graph Example (1)

- After the flat profile gives you a feel of the costly functions, you can get a better story from the call-graph

```
index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.30   14.19                 main [1]
                0.58    7.13 100000000/100000000     int_math [2]
                0.43    6.04 100000000/100000000     float_math [3]
-----------------------------------------------
                0.58    7.13 100000000/100000000     main [1]
[2]     53.2    0.58    7.13 100000000         int_math [2]
                2.44    3.13 100000000/100000000     int_math_helper [4]
                1.56    0.00 100000000/300000000     int_power [5]
-----------------------------------------------
                0.43    6.04 100000000/100000000     main [1]
[3]     44.7    0.43    6.04 100000000         float_math [3]
                1.65    2.93 100000000/100000000     float_math_helper [6]
                1.47    0.00 100000000/300000000     float_power [7]
```

Reading the Call Graph

- The line with the index is the current function being looked at **(primary line)**
- Lines above are functions which called this function
- Lines below are functions which were called by this function (children)

Primary Line

- **time:** total percentage of time spent in this function and it's children
- **self:** same as flat profile
- **children:** time spent in all calls made by the function
  - It should be equal to self + children of all functions below

Reading the Call Graph Callers

Callers (functions above the primary line)

- **self:** time spent in primary function, when called from current function
- **children:** time spent in primary function's children, when called from current function
- **called:** number of times primary function was called from current function / number of nonrecursive calls to primary function

Reading the Call Graph Callees

Callees (functions below the primary line)

- **self:** time spent in current function when called from primary function
- **children:** time spent in current function's children calls when called from primary function
  - self + children is an estimate of time spent in current function when called from primary function
- **called:** number of times current function was called from primary function / number of nonrecursive calls to current function

Call Graph Example (2)

```
index % time    self  children    called     name

                2.44    3.13 100000000/100000000     int_math [2]
[4]     38.4    2.44    3.13 100000000          int_math_helper [4]
                3.13    0.00 200000000/300000000     int_power [5]
-----------------------------------------------
                1.56    0.00 100000000/300000000     int_math [2]
                3.13    0.00 200000000/300000000     int_math_helper [4]
[5]     32.4    4.69    0.00 300000000          int_power [5]
-----------------------------------------------
                1.65    2.93 100000000/100000000     float_math [3]
[6]     31.6    1.65    2.93 100000000          float_math_helper [6]
                2.93    0.00 200000000/300000000     float_power [7]
-----------------------------------------------
                1.47    0.00 100000000/300000000     float_math [3]
                2.93    0.00 200000000/300000000     float_math_helper [6]
[7]     30.3    4.40    0.00 300000000          float_power [7]
```

- We can now see where most of the time comes from, and pin-point any locations that makes unexpected calls, etc.
- This example isn't too exciting, and we could simplify the math

Summary

- Saw how to use `gprof` (one option for Assignment 3)
- Profile early and often
- Make sure your profiling shows what you expect
- We'll see other profiles we can use as well
  - OProfile
  - Valgrind
  - AMD CodeAnalyst

Assignment 3

- Hopefully out Wednesday
- Travelling salesman problem
- Improving a genetic algorithm in C++
- Now is your time to get into groups of 2, e-mail me with your WatIDs