ECE459: Programming for Performance	Winter 2016

Lecture 20 — Profiling

Patrick Lam

Profiling

If you want to make your programs or systems fast, you need to find out what is currently slow and improve it. (duh!)

How profiling works:

- sampling-based (traditional): every so often (e.g. 100ms for gprof), query the system state; or,
- instrumentation-based, or probe-based/predicate-based (traditionally too expensive): query system state under certain conditions; like conditional breakpoints.

We'll talk about both per-process profiling and system-wide profiling.

If you need your system to run fast, you need to start profiling and benchmarking as soon as you can run the system. Benefits:

- establishes a baseline performance for the system;
- allows you to measure impacts of changes and further system development;
- allows you to re-design the system before it's too late;
- avoids the need for "perf spray" to make the system faster, since that spray is often made of "unobtainium"[1].

Tips for Leveraging Profiling. When writing large software projects:

- First, write clear and concise code.
  Don't 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.

Look for abnormalities; in particular, you're looking for deviations from the following rules:

- time is spent in the right part of the system/program;
- time is not spent in error-handling, noncritical code, or exceptional cases; and
- time is not unnecessarily spent in the operating system.

For instance, "why is `ps` taking up all my cycles?"; see page 34 of Cantrill[2].

[1] `http://en.wikipedia.org/wiki/Unobtainium`
[2] `http://queue.acm.org/detail.cfm?id=1117401`

Development vs. production. You can always profile your systems in development, but that might not help with complexities in production. (You want separate dev and production systems, of course!) We'll talk a bit about DTrace, which is one way of profiling a production system. The constraints on profiling production systems are that the profiling must not affect the system's performance or reliability.

Userspace per-process profiling

Sometimes—or, in this course, often—you can get away with investigating just one process and get useful results about that process's behaviour. We'll first talk about `gprof`, the GNU profiler tool[3], and then continue with other tools.

`gprof` does sampling-based profiling for single processes: it requests that the operating system interrupt the process being profiled at regular time intervals and figures out which procedure is currently running. It also adds a bit of instrumentation to collect information about which procedures call other procedures.

"Flat" profile. The obvious thing to do with the profile information is to just print it out. You get a list of procedures called and the amount of time spent in each of these procedures.

The general limitation is that procedures that don't run for long enough won't show up in the profile. (There's a caveat: if the function was compiled for profiling, then it will show up anyway, but you won't find out about how long it executed for).

"Call graph". `gprof` can also print out its version of a call graph, which shows the amount of time that either a function runs (as in the "flat" profile) as well as the amount of time that the callees of the function run. Another term for such a call graph is a "dynamic call graph", since it tracks the dynamic behaviour of the program. Using the `gprof` call graph, you can find out who is responsible for calling the functions that take a long time.

Limitations of `gprof`. Beyond the usual limitations of a process-oriented profiler, `gprof` also suffers limitations from running completely in user-space. That is, it has no access to information about system calls, including time spent doing I/O. It also doesn't know anything about the CPU's built-in counters (e.g. cache miss counts, etc). Like the other profilers, it causes overhead when it's running, but the overhead isn't too large.

`gprof` usage guide

We'll give some details about using `gprof`. First, use the `-pg` flag with `gcc` when compiling and linking. Next, run your program as you normally would. Your program will now create `gmon.out`.

Use gprof to interpret the results: `gprof <executable>`.

Example. Consider a program with 100 million calls to two math functions.

[3] `http://sourceware.org/binutils/docs/gprof/`

```c
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);
    }
}

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;
}
```

```c
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;
}
```

Looking at the code, we have no idea what takes longer. One might guess that floating point math takes longer. This is admittedly a silly example, but it works well to illustrate our point.

Flat Profile Example. When we run the program and look at the flat profile, 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
```

There is one function per line. Here are what the columns mean:

- **% time:** the percent of the total execution time in this function.
- **self:** seconds in this function.
- **cumulative:** sum of this function's time + any above it in table.
- **calls:** number of times this function was called.
- **self ns/call:** just self nanoseconds / calls.
- **total ns/call:** mean function execution time, including calls the function makes.

Call Graph Example. After the flat profile gives you a feel for which functions are costly, 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]
-----------------------------------------------------
                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]
```

To interpret the call graph, note that the line with the index [N] is the *primary line*, or the current function being considered.

- Lines above the primary line are the functions which called this function.
- Lines below the primary line are the functions which were called by this function (children).

For the primary line, the columns mean:

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

For 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.

For callees (functions below the primary line):

- **self:** time spent in current function when called from primary.
- **children:** time spent in current function's children calls when called from primary.
  - 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.

Based on this information, we can now see where most of the time comes from, and pinpoint any locations that make unexpected calls, etc. This example isn't too exciting; we could simplify the math and optimize the program that way.