ECE459: Programming for Performance Winter 2020

Lecture 24 — Profiling

Jeff Zarnett & Patrick Lam *2019-12-09*

Profiling

Think back to the beginning of the course when we did a quiz on what operations are fast and what operations are not. The important takeaway was not that we needed to focus on how to micro-optimize this abstraction or that hash function, but that our intuition about what is fast and what is slow is often wrong. Not just at a macro level, but at a micro level. You may be able to narrow down that this computation of x is slow, but if you examine it carefully... what parts of it are slow?

If you don't use tools, then you end up guessing. You just make some assumptions about what you think is likely to be slow and try to change it. You've probably heard the famous quotation before, but here it is in its full form:

> *Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.*
>
> – Donald Knuth

So going about this blindly is probably a waste of time. You might be fortunate and optimize a slow part[1] but we should really follow one of my favourite rules: "don't guess, measure!"[2] So, to make your programs or systems fast, you need to find out what is currently slow and improve it (duh!). Up until now in the course it's mostly been about "let's speed this up", but we did not take much time to decide what we should speed up (though you maybe did this on your assignment 2...?).

The general idea is, collect some data on what parts of the code are taking up the majority of the time. This can be broken down into looking at what functions get called, or how long functions take, or what's using memory...

There is always the "informal" way of doing this; it sort of works but it's not exactly the best plan. You probably know that when developing a program you can "debug" it without using any tools (e.g., `gdb`) by inserting a lot of print statements to the console or the log file. So when you enter function `foo` you print a nice little line on the console that say something like "entering function foo", associated with a timestamp and then when you're ready to return, a corresponding print function that says "exiting" appears, also with a timestamp.

This approach kind of works, and I've used it myself to figure out what blocks of a single large function are taking a long time (updating exchange rates... yeah). But this approach is not necessarily a good one. It's an example of "invasive" profiling – we are going in and changing the source code of the program in question – to add instrumentation (log/debug statements). Plus we have to do a lot of manual accounting. Assuming your program is fast and goes through functions quickly and often, trying to put the pieces together manually is hopeless. It worked in that one example because the single function itself was running in the half hour range and I could see that the save operation was taking twelve minutes. Not kidding.

(Also like debugging, if you get to be a wizard you can maybe do it by code inspection, but that technique of speculative execution inside your head is a lot harder to apply to performance problems than it is to debugging.)

So we should all agree, we want to use tools and do this in a methodical way.

[1] There is a saying that even a blind squirrel sometimes finds a nut.

[2] Now I am certain you are sick of hearing that.

Now that we agree on that, let's think about how profiling tools work

- 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"[3].

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 [Can06].

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.

[3] `http://en.wikipedia.org/wiki/Unobtainium`

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[4], 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 `clang` when compiling and linking. (It's currently broken in gcc, unless you give `-no-pie`.) 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.

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

```
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;
}
```

```
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.

Introduction to gperftools

Next, we'll talk about the Google Performance Tools.

`http://google-perftools.googlecode.com/svn/trunk/doc/cpuprofile.html`

They include:

- a CPU profiler
- a heap profiler
- a heap checker; and
- a faster (multithreaded) `malloc`.

We'll mostly use the CPU profiler. Characteristics include:

- supposedly works for multithreaded programs;
- purely statistical sampling;
- no recompilation required (typically benefit from re-linking); and
- better output, including built-in graphical output.

You can use the profiler without any recompilation. But this is not recommended; you'll get worse data. Use `LD_PRELOAD`, which changes the dynamic libraries that an executable uses.

```
% LD_PRELOAD="/usr/lib/libprofiler.so" CPUPROFILE=test.prof ./test
```

The other (more-recommended) option is to link to the profiler with `-lprofiler`.

Both options read the `CPUPROFILE` environment variable, which specifies where profiling data goes.

You can use the profiling library directly as well:

```
#include <gperftools/profiler.h>
```

Then, bracket code you want profiled with:

```
ProfilerStart()
// ...
ProfilerStop()
```

You can change the sampling frequency with the `CPUPROFILE_FREQUENCY` environment variable (default value 100 interrupts/second).

`pprof` **usage.** `pprof` is like `gprof` for Google Perf Tools. It analyzes profiling results. Here are some usage examples.

```
% pprof test test.prof
    Enters "interactive" mode
% pprof --text test test.prof
    Outputs one line per procedure
% pprof --gv test test.prof
     Displays annotated call-graph via 'gv'
% pprof --gv --focus=Mutex test test.prof
    Restricts to code paths including a .*Mutex.* entry
% pprof --gv --focus=Mutex --ignore=string test test.prof
    Code paths including Mutex but not string
% pprof --list=getdir test test.prof
    (Per-line) annotated source listing for getdir()
% pprof --disasm=getdir test test.prof
    (Per-PC) annotated disassembly for getdir()
```

Can also output `dot`, `ps`, `pdf` or `gif` instead of `gv`.

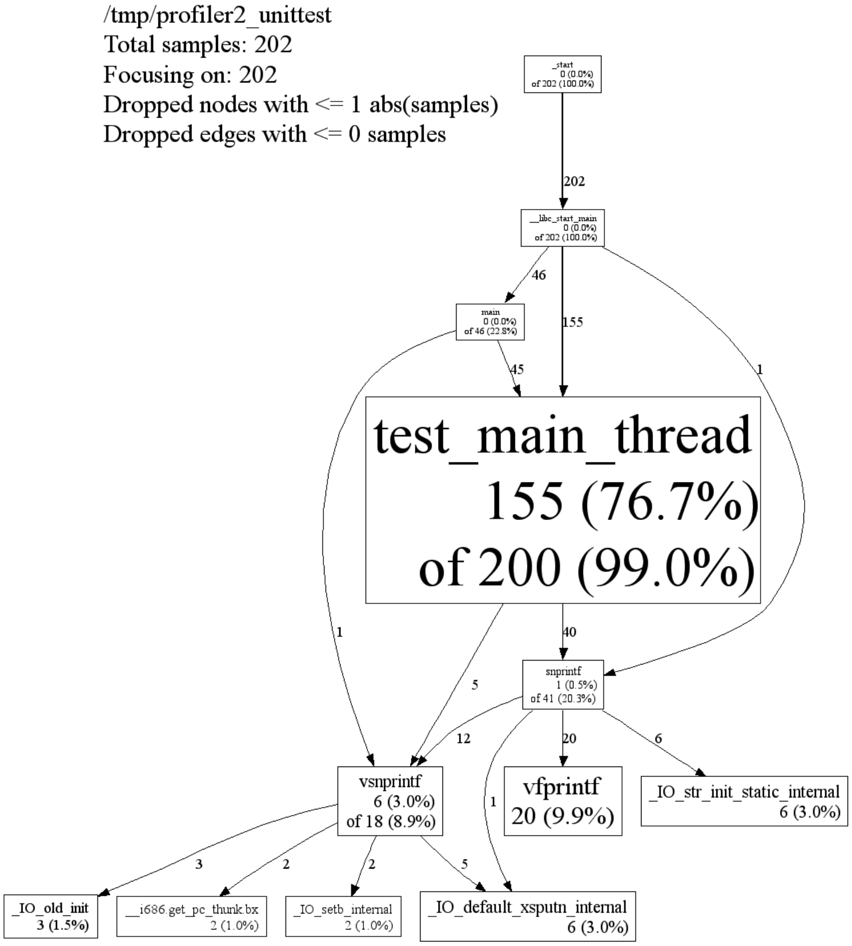
`pprof` **text output.** This is similar to the flat profile in `gprof`.

```
jon@riker examples master % pprof --text test test.prof
Using local file test.
Using local file test.prof.
Removing killpg from all stack traces.
Total: 300 samples
      95  31.7%  31.7%      102  34.0% int_power
      58  19.3%  51.0%       58  19.3% float_power
      51  17.0%  68.0%       96  32.0% float_math_helper
      50  16.7%  84.7%      137  45.7% int_math_helper
      18   6.0%  90.7%      131  43.7% float_math
      14   4.7%  95.3%      159  53.0% int_math
      14   4.7% 100.0%      300 100.0% main
       0   0.0% 100.0%      300 100.0% __libc_start_main
       0   0.0% 100.0%      300 100.0% _start
```

Columns, from left to right, denote:

- Number of samples in this function.
- Percentage of samples in this function (same as **time** in `gprof`).
- Percentage of checks in the functions printed so far (equivalent to **cumulative**, but in %).
- Number of checks in this function and its callees.
- Percentage of checks in this function and its callees.
- Function name.

Graphical Output. Google Perf Tools can also produce graphical output:



This shows the same numbers as the text output. This particular graphical example is on a different program than the text output, and that's why they look different. Directed edges denote function calls. Note:

$$\text{\# of samples in callees} = \text{\# in ``this function + callees''} - \text{\# in ``this function''}.$$

For example, in `float_math_helper`, we have "51 (local) of 96 (cumulative)". Here,

$$96 - 51 = 45 \text{ (callees)}.$$

- callee `int_power` = 7 (bogus)
- callee `float_power` = 38
- callees total = 45

Note that the call graph is not exact. In fact, it shows many bogus relations which clearly don't exist. For instance, we know that there are no cross-calls between `int` and `float` functions.

As with `gprof`, optimizations will change the graph.

You'll probably want to look at the text profile first, then use the `-focus` flag to look at individual functions.

References

[Can06] Bryan Cantrill. Hidden in Plain Sight, 2006. Online; accessed 20-Janaury-2016. URL: `http://queue.acm.org/detail.cfm?id=1117401`.