# Introduction to Profiling

2024 Fall ECE454: Computer Systems Programming
Jon Eyolfson

Lecture 2
1.0.1

## We Could Time the Functions We're Interested In

We did it in the last lecture using the `CLOCK_PROCESS_CPUTIME_ID` clock

It measures actual CPU time consumed by the process

However this too much manual effort, and we may not know where to focus

## We Need to Measure Our Programs with Profilers

We have to **profile** to see what is taking up execution time in a program

Profilers have two kinds of output: flat, and call-graph

They collect data from: statistical sampling, or instrumentation

## The Difference Between Profiler 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

## How Sampling Profilers Collect Data

Mostly done by taking samples of the system state at a set rate

At a sample: check the system state

Will have some slowdown, but not much

## How Instrumenting Profilers Collect Data

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)

## The Basic Guidelines for Large Software Projects

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!

## Some Sanity Checks When Looking at the Data

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

## Tools We'll Use in This Course—`gcov` and `gprof`

`gcov` is a coverage tool that instruments your program

It tells you how many times each line executes
(needs debugging information)

This does not give you performance numbers!

`gprof` is a profiler that uses sampling and instrumentation

and tells you how long each function executes

Reference: GCC Instrumentation Options

## `gcov` Usage

Use the compiler flag `--coverage` when compiling and linking
(adds the `-fprofile-arcs` and `-ftest-coverage` flags)

(or use `b_coverage=true` with `meson`)

When you run your program every object file generates a `.gcda` and `.gcno`

Generate a report using `gcovr`, or using `meson`:

```
meson setup -Db_coverage=true build
meson compile -C build
ninja -C build coverage
```

## `gprof` Usage

Use the `-pg` flag with `gcc` when compiling (also linking)

Run your program as you normally would (it now generates a `gmon.out` file)

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

Example commands using `meson`, assuming the output `calc`:

```
meson setup -Dc_args=-pg -Dc_link_args=-pg build
meson compile -C build
build/calc
gprof build/calc
```

## Flat Profile Example

When we look at the profiling data, this is the first thing we see:

```
 Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ns/call  ns/call  name
 34.49      0.75     0.75 300000000     2.51     2.51  int_power
 29.43      1.39     0.64                              _init
 28.51      2.02     0.62 300000000     2.07     2.07  float_power
  2.76      2.08     0.06 100000000     0.60     7.32  float_math
  2.30      2.13     0.05 100000000     0.50     4.65  float_math_helper
  1.84      2.17     0.04                              main
  0.46      2.18     0.01 100000000     0.10     7.72  int_math
  0.46      2.19     0.01 100000000     0.10     5.11  int_math_helper
```

## Flat Profile Reference

There is one function per line, the columns are:

**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

**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

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
[1]     70.6    0.04    1.50                 main [1]
                0.01    0.76 100000000/100000000     int_math [2]
                0.06    0.67 100000000/100000000     float_math [4]
-----------------------------------------------
                0.01    0.76 100000000/100000000     main [1]
[2]     35.3    0.01    0.76 100000000               int_math [2]
                0.01    0.50 100000000/100000000     int_math_helper [7]
                0.25    0.00 100000000/300000000     int_power [3]
-----------------------------------------------
                0.25    0.00 100000000/300000000     int_math [2]
                0.50    0.00 200000000/300000000     int_math_helper [7]
[3]     34.4    0.75    0.00 300000000               int_power [3]
-----------------------------------------------
                0.01    0.50 100000000/100000000     int_math [2]
[7]     23.4    0.01    0.50 100000000               int_math_helper [7]
                0.50    0.00 200000000/300000000     int_power [3]
```

## 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 (callers)

Lines below are functions which were called by this function (callees)

**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 Callers in the Call Graph

The callers are the 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
divided by number of nonrecursive calls to primary function

## Reading the Callees in the Call Graph

The callees are the 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

Note: `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 divided by number of nonrecursive calls to current function