How do I profile C++ code running on Linux? [closed]

Asked 16 years ago Modified 18 days ago Viewed 737k times

2170

Closed. This question needs to be more focused. It is not currently accepting answers.

Want to improve this question? Update the question so it focuses on one problem only by editing this post.

Closed 17 days ago.

Improve this question

How do I find areas of my code that run slowly in a C++ application running on Linux?

c++ linux profiling

Share

Improve this question

Follow

edited Jul 4, 2022 at 22:44

Mateen Ulhaq

27.1k ● 21 ● 117 ● 152

asked Dec 17, 2008 at 20:29

Gabriel Isenberg

26.3k ● 4 ● 38 ● 58

2 It is already answered on the following link: stackoverflow.com/questions/2497211/… – Kapil Gupta May 22, 2012 at 10:12

9 Most of the answers are `code` profilers. However, priority inversion, cache aliasing, resource contention, etc. can all be factors in optimizing and performance. I think that people read information into *my slow code*. FAQs are referencing this thread. – artless-noise-bye-due2AI Mar 17, 2013 at 18:44

7 CppCon 2015: Chandler Carruth "Tuning C++: Benchmarks, and CPUs, and Compilers! Oh My!" – maddouri Oct 11, 2015 at 22:06

5 I used to use pstack randomly, most of the time will print out the most typical stack where the program is most of the time, hence pointing to the bottleneck. – Jose Manuel Gomez Alvarez Dec 15, 2016 at 9:26

1 cachegrind/callgrind ? – v.oddou Feb 22, 2019 at 1:27

20 Answers

Sorted by: Highest score (default)

1623 ✓

If your goal is to use a profiler, use one of the suggested ones.

However, if you're in a hurry and you can manually interrupt your program under the debugger while it's being subjectively slow, there's a simple way to find performance problems.

Execute your code in a debugger like gdb, halt it and each time look at the call stack (e.g. backtrace) several times. If there is some code that is wasting some percentage of the time, 20% or 50% or whatever, that is the probability that you will catch it in the act on each sample. So, that is roughly the percentage of samples on which you will see it. There is no educated guesswork required. If you do have a guess as to what the problem is, this will prove or disprove it.

You probably have multiple performance problems of different sizes. If you clean out any one of them, the remaining ones will take a larger percentage, and be easier to spot, on subsequent passes. This *magnification effect*, when compounded over multiple problems, can lead to truly massive speedup factors.

Caveat: Programmers tend to be skeptical of this technique unless they've used it themselves. They will say that profilers give you this information, but that is only true if they sample the entire call stack, and then let you examine a random set of samples. (The summaries are where the insight is lost.) Call graphs don't give you the same information, because

1. They don't summarize at the instruction level, and
2. They give confusing summaries in the presence of recursion.

They will also say it only works on toy programs, when actually it works on any program, and it seems to work better on bigger programs, because they tend to have more problems to find. They will say it sometimes finds things that aren't problems, but that is only true if you see something *once*. If you see a problem on more than one sample, it is real.

P.S. This can also be done on multi-thread programs if there is a way to collect call-stack samples of the thread pool at a point in time, as there is in Java.

P.P.S As a rough generality, the more layers of abstraction you have in your software, the more likely you are to find that that is the cause of performance problems (and the opportunity to get speedup).

Added: It might not be obvious, but the stack sampling technique works equally well in the presence of recursion. The reason is that the time that would be saved by removal of an instruction is approximated by the fraction of samples containing it, regardless of the number of times it may occur within a sample.

Another objection I often hear is: "*It will stop someplace random, and it will miss the real problem*". This comes from having a prior concept of what the real problem is. A key property of performance problems is that they defy expectations. Sampling tells you something is a problem, and your first reaction is disbelief. That is natural, but you can be sure if it finds a problem it is real, and vice-versa.

Added: Let me make a Bayesian explanation of how it works. Suppose there is some instruction `I` (call or otherwise) which is on the call stack some fraction `f` of the time (and thus costs that much). For simplicity, suppose we don't know what `f` is, but assume it is either 0.1, 0.2, 0.3, ... 0.9, 1.0, and the prior probability of each of these possibilities is 0.1, so all of these costs are equally likely a-priori.

Then suppose we take just 2 stack samples, and we see instruction `I` on both samples, designated observation `o=2/2`. This gives us new estimates of the frequency `f` of `I`, according to this:

```
Prior
P(f=x) x  P(o=2/2|f=x) P(o=2/2&&f=x)  P(o=2/2&&f >= x)

0.1    1     1             0.1           0.1
0.1    0.9   0.81          0.081         0.181
0.1    0.8   0.64          0.064         0.245
0.1    0.7   0.49          0.049         0.294
0.1    0.6   0.36          0.036         0.33
0.1    0.5   0.25          0.025         0.355
0.1    0.4   0.16          0.016         0.371
0.1    0.3   0.09          0.009         0.38
0.1    0.2   0.04          0.004         0.384
0.1    0.1   0.01          0.001         0.385

                  P(o=2/2) 0.385
```

The last column says that, for example, the probability that `f` >= 0.5 is 92%, up from the prior assumption of 60%.

Suppose the prior assumptions are different. Suppose we assume `P(f=0.1)` is .991 (nearly certain), and all the other possibilities are almost impossible (0.001). In other words, our prior certainty is that `I` is cheap. Then we get:

```
Prior
P(f=x) x  P(o=2/2|f=x) P(o=2/2&& f=x)  P(o=2/2&&f >= x

0.001  1    1              0.001         0.001
0.001  0.9  0.81           0.00081       0.00181
0.001  0.8  0.64           0.00064       0.00245
0.001  0.7  0.49           0.00049       0.00294
0.001  0.6  0.36           0.00036       0.0033
0.001  0.5  0.25           0.00025       0.00355
0.001  0.4  0.16           0.00016       0.00371
0.001  0.3  0.09           0.00009       0.0038
0.001  0.2  0.04           0.00004       0.00384
0.991  0.1  0.01           0.00991       0.01375

                  P(o=2/2) 0.01375
```

Now it says `P(f >= 0.5)` is 26%, up from the prior assumption of 0.6%. So Bayes allows us to update our estimate of the probable cost of `I`. If the amount of data is small, it doesn't tell us accurately what the cost is, only that it is big enough to be worth fixing.

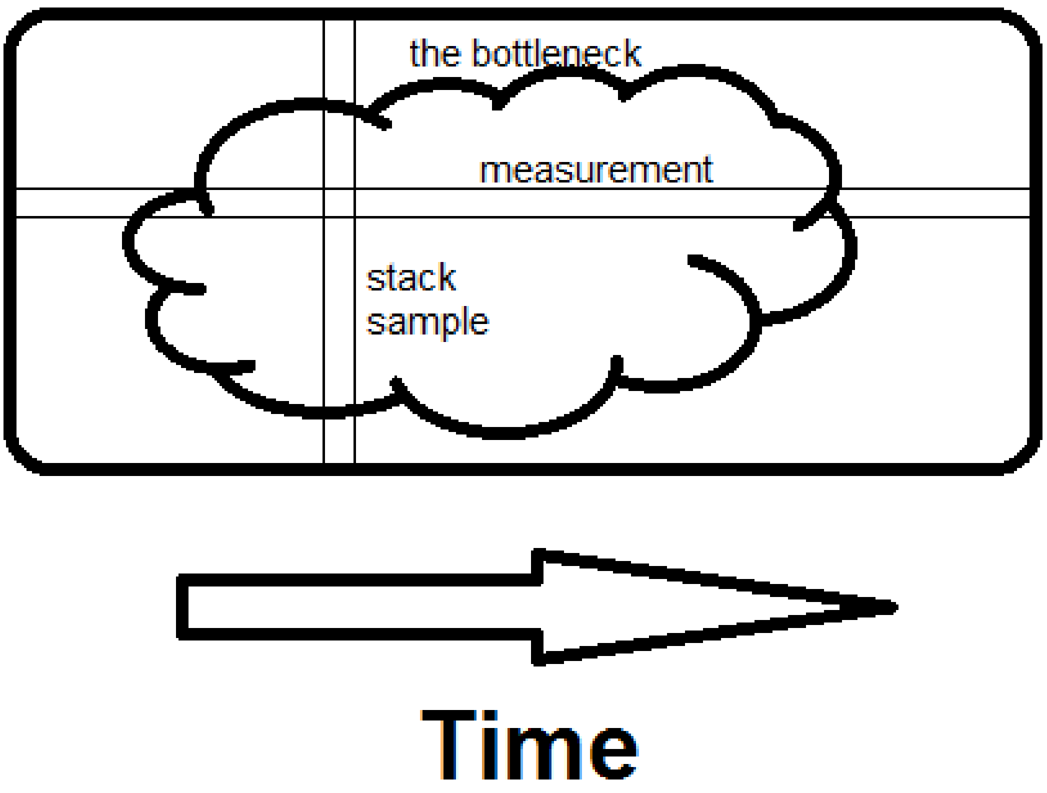
Yet another way to look at it is called the Rule Of Succession. If you flip a coin 2 times, and it comes up heads both times, what does that tell you about the probable weighting of the coin? The respected way to answer is to say that it's a Beta distribution, with average value `(number of hits + 1) / (number of tries + 2) = (2+1)/(2+2) = 75%`.

(The key is that we see `I` more than once. If we only see it once, that doesn't tell us much except that `f` > 0.)

So, even a very small number of samples can tell us a lot about the cost of instructions that it sees. (And it will see them with a frequency, on average, proportional to their cost. If `n` samples are taken, and `f` is the cost, then `I` will appear on `nf+/-sqrt(nf(1-f))` samples. Example, `n=10`, `f=0.3`, that is `3+/-1.4` samples.)

Added: To give an intuitive feel for the difference between measuring and random stack sampling: There are profilers now that sample the stack, even on wall-clock time, but *what comes out* is measurements (or hot path, or hot spot, from which a "bottleneck" can easily hide). What they don't show you (and they easily could) is the actual samples themselves. And if your goal is to *find* the bottleneck, the number of them you need to see is, *on average*, 2 divided by the fraction of time it takes. So if it takes 30% of time, 2/.3 = 6.7 samples, on average, will show it, and the chance that 20 samples will show it is 99.2%.

Here is an off-the-cuff illustration of the difference between examining measurements and examining stack samples. The bottleneck could be one big blob like this, or numerous small ones, it makes no difference.



Measurement is horizontal; it tells you what fraction of time specific routines take. Sampling is vertical. If there is any way to avoid what the whole program is doing at that moment, *and if you see it on a second sample*, you've found the bottleneck. That's what makes the difference - seeing the whole reason for the time being spent, not just how much.

Share Improve this answer Follow

edited Dec 2 at 23:30

community wiki
28 revs, 6 users 67%
Mike Dunlavey

339 This is basically a poor man's sampling profiler, which is great, but you run the risk of a too-small sample size which

will possibly give you entirely spurious results.
– Crashworks May 22, 2009 at 21:56

119 @Crash: I won't debate the "poor man" part :-) It's true that statistical measurement precision requires many samples, but there are two conflicting goals - measurement and problem location. I'm focussing on the latter, for which you need precision of location, not precision of measure. So for example, there can be, mid-stack, a single function call A(); that accounts for 50% of time, but it can be in another large function B, along with many other calls to A() that are not costly. Precise summaries of function times can be a clue, but every other stack sample will pinpoint the problem.
– Mike Dunlavey May 23, 2009 at 1:14

51 ... the world seems to think that a call-graph, annotated with call counts and/or average timing, is good enough. It is not. And the sad part is, for those that sample the call stack, the most useful information is right in front of them, but they throw it away, in the interests of "statistics".
– Mike Dunlavey May 24, 2009 at 18:08

39 I don't mean to disagree with your technique. Clearly I rely quite heavily on stack-walking sampling profilers. I'm just pointing out that there are some tools that do it in an automated way now, which is important when you're past the point of getting a function from 25% to 15% and need to knock it down from 1.2% to 0.6%. – Crashworks Jun 2, 2009 at 3:27

20 -1: Neat idea, but if you're getting paid to work in even a moderately performance oriented environment this is a waste of everyone's time. Use a real profiler so we don't have to come along behind you and fix the actual problems. – Sam Harwell Apr 8, 2010 at 13:26

Use Valgrind with the following options:

709

```
valgrind --tool=callgrind ./(Your binary)
```

This generates a file called `callgrind.out.x`. Use the `kcachegrind` tool to read this file. It will give you a graphical analysis of things with results like which lines cost how much.

Share Improve this answer Follow

edited Jul 4, 2022 at 22:00

Mateen Ulhaq
27.1k ● 21 ● 117 ● 152

answered Apr 21, 2009 at 4:09

Ajay
10.2k ● 8 ● 33 ● 34

74 valgrind is great, but be warned that it will make your program darn slow – neves Jan 25, 2012 at 20:07

39 Check out also Gprof2Dot for an amazing alternative way to visualize the output. `./gprof2dot.py -f callgrind callgrind.out.x | dot -Tsvg -o output.svg` – Sebastian May 22, 2013 at 13:42

6 @neves Yes Valgrind is just not very helpful in terms of speed for profiling "gstreamer" and "opencv" applications real-time. – enthusiasticgeek May 22, 2013 at 20:20

4 @Sebastian: `gprof2dot` is now here: github.com/jrfonseca/gprof2dot – John Zwinck Apr 27, 2017 at 3:19

4 One thing to bear in mind is to compile WITH debug symbols included but WITH optimisation, to get something explorable yet with the speed characteristics similar to the actual "release" build. – BIOStheZerg Oct 29, 2019 at 12:01

388

I assume you're using GCC. The standard solution would be to profile with gprof.

Be sure to add `-pg` to compilation before profiling:

```
cc -o myprog myprog.c utils.c -g -pg
```

I haven't tried it yet but I've heard good things about google-perftools. It is definitely worth a try.

Related question here.

A few other buzzwords if `gprof` does not do the job for you: Valgrind, Intel VTune, Sun DTrace.

Share Improve this answer Follow

edited Aug 12, 2017 at 17:35
try-catch-finally
7,614 ● 6 ● 49 ● 71

answered Dec 17, 2008 at 20:34
Nazgob
8,573 ● 4 ● 42 ● 42

6 I agree that gprof is the current standard. Just a note, though, Valgrind is used to profile memory leaks and other memory-related aspects of your programs, not for speed optimization. – Bill the Lizard Dec 18, 2008 at 15:02

74 Bill, In vaglrind suite you can find callgrind and massif. Both are pretty useful to profile apps – Salvatore Dario Minonne Dec 18, 2008 at 15:05

9 @Bill-the-Lizard: Some comments on **gprof** : stackoverflow.com/questions/1777556/alternatives-to-gprof/… – Mike Dunlavey Mar 4, 2010 at 13:23

7 gprof -pg is only an approximation of callstack profiling. It inserts mcount calls to track which functions are calling which other functions. It uses standard time based sampling for, uh, time. It then apportions times sampled in a function foo() back to the callers of foo(), in proprtion to the numberr of calls. So it doesn't distinguish between calls of different costs. – Krazy Glew Apr 28, 2012 at 5:45

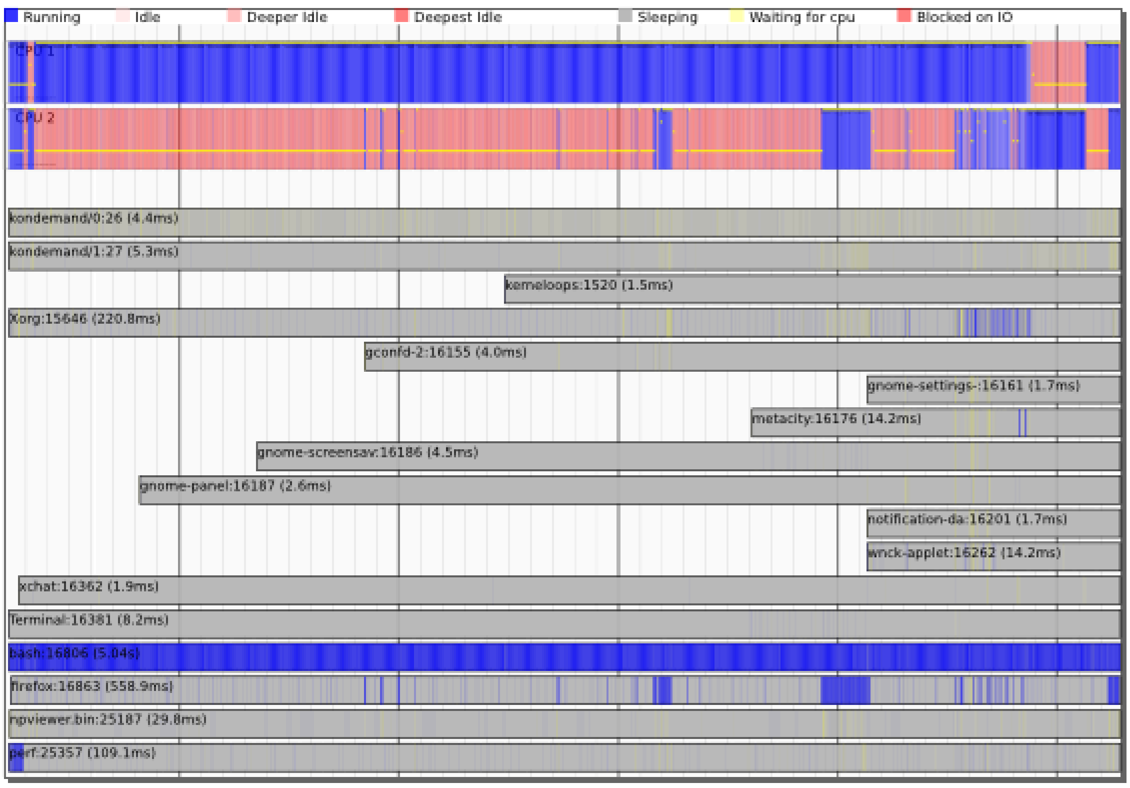
2 With clang/clang++, one might consider using gperftools's CPU profiler. Caveat: Have not done so myself. – einpoklum Oct 14, 2019 at 14:27

299

Newer kernels (e.g. the latest Ubuntu kernels) come with the new 'perf' tools (`apt-get install linux-tools`) AKA perf_events.

These come with classic sampling profilers (man-page) as well as the awesome timechart!

The important thing is that these tools can be **system profiling** and not just process profiling - they can show the interaction between threads, processes and the kernel and let you understand the scheduling and I/O dependencies between processes.



Share Improve this answer Follow

edited Aug 12, 2017 at 17:30

try-catch-finally
7,614 ● 6 ● 49 ● 71

answered Aug 17, 2010 at 11:48

Will
75.6k ● 43 ● 175 ● 255

14 Great tool! Is there anyway for me to get a typical "butterfly" view that starts from "main->func1->fun2" style? I can't seem to figure that out... `perf report` seems to give me the function names with the call parents... (so it's sort of an inverted butterfly view) – kizzx2 Oct 1, 2010 at 6:17

1 Will, can perf show timechart of thread activity; with CPU number information added? I want to see when and which thread was running on every CPU. – osgx Dec 6, 2011 at 4:24

3 @kizzx2 - you can use `gprof2dot` and `perf script`. Very nice tool! – dashesy May 14, 2012 at 23:55

3 Even newer kernels like 4.13 have eBPF for profiling. See brendangregg.com/blog/2015-05-15/ebpf-one-small-step.html and brendangregg.com/ebpf.html – Andrew Stern Oct 13, 2017 at 15:00

3 This should be the accepted answer. Using a debugger introduces too much noise in the samples. Performance counters for linux works for multiple threads, multiple processes, user and kernel space, which is great. You can also retrieve many useful information such as branch and cache misses. In the same website @AndrewStern mentioned, there is a flamegraph which is very useful for this kind of analysis: flame graphs. It generates SVG files that can be opened with a web browser for interactive graphs! – Jorge Bellon Dec 17, 2018 at 12:25

The answer to run `valgrind --tool=callgrind` is not quite complete without some options. We usually do not want to profile 10 minutes of slow startup time under

101

Valgrind and want to profile our program when it is doing some task.

So this is what I recommend. Run program first:

```
valgrind --tool=callgrind --dump-instr=yes -v --instr-
```

Now when it works and we want to start profiling we should run in another window:

```
callgrind_control -i on
```

This turns profiling on. To turn it off and stop whole task we might use:

```
callgrind_control -k
```

Now we have some files named callgrind.out.* in current directory. To see profiling results use:

```
kcachegrind callgrind.out.*
```

I recommend in next window to click on "Self" column header, otherwise it shows that "main()" is most time consuming task. "Self" shows how much each function itself took time, not together with dependents.

Share Improve this answer Follow

edited Sep 5, 2015 at 10:44

Peter Mortensen
31.6k ● 22 ● 109 ● 133

answered Jun 8, 2012 at 8:01

Tõnu Samuel
2,896 ● 3 ● 22 ● 31

13 Now on some reason callgrind.out.* files were always empty. Executing callgrind_control -d was useful to force dump of data to disk. – Tõnu Samuel Jul 31, 2014 at 4:25

4 Can't. My usual contexts are something like whole MySQL or PHP or some similar big thing. Often even do not know what I want to separate at first. – Tõnu Samuel Nov 21, 2015 at 22:50

3 Or in my case my program actually loads a bunch of data into an LRU cache, and I want not to profile that. So I force-load a subset of the cache at startup, and profile the code using only that data (letting the OS+CPU manage the memory use within my cache). It works, but loading that cache is slow and CPU intensive across code that I'm trying to profile in a different context, so callgrind produces badly polluted results. – Code Abominator Mar 17, 2016 at 3:49

3 there is also `CALLGRIND_TOGGLE_COLLECT` to enable/disable collection programmatically; see stackoverflow.com/a/13700817/288875 – Andre Holzner Aug 29, 2017 at 16:59

2 @TõnuSamuel, for me also callgrind.out.* was empty. In my case, the program was getting crahsed while profiling. Once the reason for crash was resolved, I am able to see contents in callgrind.out.* file. – explorer Jun 4, 2020 at 4:29

91

Survey of C++ profiling techniques: gprof vs valgrind vs perf vs gperftools

In this answer, I will use several different tools to a analyze a few very simple test programs, in order to concretely compare how those tools work.

Summary of results:

tool	recompile	slowdown
gprof	`-pg`	1x (2x-3x reported)
perf	no	1x
valgrind	no	15x (80x reported)
gperftools	no	

The following test program is very simple and does the following:

- `main` calls `fast` and `maybe_slow` 3 times, one of the `maybe_slow` calls being slow

  The slow call of `maybe_slow` is 10x longer, and dominates runtime if we consider calls to the child function `common`. Ideally, the profiling tool will be able to point us to the specific slow call.

- both `fast` and `maybe_slow` call `common`, which accounts for the bulk of the program execution

- The program interface is:

```
./main.out [n [seed]]
```

and the program does `O(n^2)` loops in total. `seed` is just to get different output without affecting runtime.

main.c

```
#include <inttypes.h>
#include <stdio.h>
#include <stdlib.h>

uint64_t __attribute__ ((noinline)) common(uint64_t n,
    for (uint64_t i = 0; i < n; ++i) {
        seed = (seed * seed) - (3 * seed) + 1;
    }
    return seed;
}

uint64_t __attribute__ ((noinline)) fast(uint64_t n, u
    uint64_t max = (n / 10) + 1;
    for (uint64_t i = 0; i < max; ++i) {
        seed = common(n, (seed * seed) - (3 * seed) +
    }
    return seed;
}

uint64_t __attribute__ ((noinline)) maybe_slow(uint64_
is_slow) {
    uint64_t max = n;
    if (is_slow) {
        max *= 10;
    }
    for (uint64_t i = 0; i < max; ++i) {
        seed = common(n, (seed * seed) - (3 * seed) +
    }
    return seed;
}

int main(int argc, char **argv) {
    uint64_t n, seed;
    if (argc > 1) {
            n = strtoll(argv[1], NULL, 0);
        } else {
            n = 1;
        }
        if (argc > 2) {
            seed = strtoll(argv[2], NULL, 0);
        } else {
            seed = 0;
        }
        seed += maybe_slow(n, seed, 0);
        seed += fast(n, seed);
        seed += maybe_slow(n, seed, 1);
        seed += fast(n, seed);
        seed += maybe_slow(n, seed, 0);
        seed += fast(n, seed);
        printf("%" PRIX64 "\n", seed);
        return EXIT_SUCCESS;
}
```

Default compilation unless otherwise specified:

```
gcc -ggdb3 -O3 -std=c99 -Wall -Wextra -pedantic -o mai
```

gprof

gprof requires recompiling the software with instrumentation, and it also uses a sampling approach together with that instrumentation. It therefore strikes a balance between accuracy (sampling is not always fully accurate and can skip functions) and execution slowdown (instrumentation and sampling are relatively fast techniques that don't slow down execution very much).

gprof is built-into GCC/binutils, so all we have to do is to compile with the `-pg` option to enable gprof. We then run the program normally with a size CLI parameter that produces a run of reasonable duration of a few seconds (`10000`):

```
gcc -pg -ggdb3 -O3 -std=c99 -Wall -Wextra -pedantic -o
time ./main.out 10000
```

For educational reasons, we will also do a run without optimizations enabled. Note that this is useless in practice, as you normally only care about optimizing the performance of the optimized program:

```
gcc -pg -ggdb3 -O0 -std=c99 -Wall -Wextra -pedantic -o
./main.out 10000
```

First, `time` tells us that the execution time with and without `-pg` were the same, which is great: no slowdown! I have however seen accounts of 2x - 3x slowdowns on complex software, e.g. as shown in this ticket.

Because we compiled with `-pg`, running the program produces a file `gmon.out` file containing the profiling data.

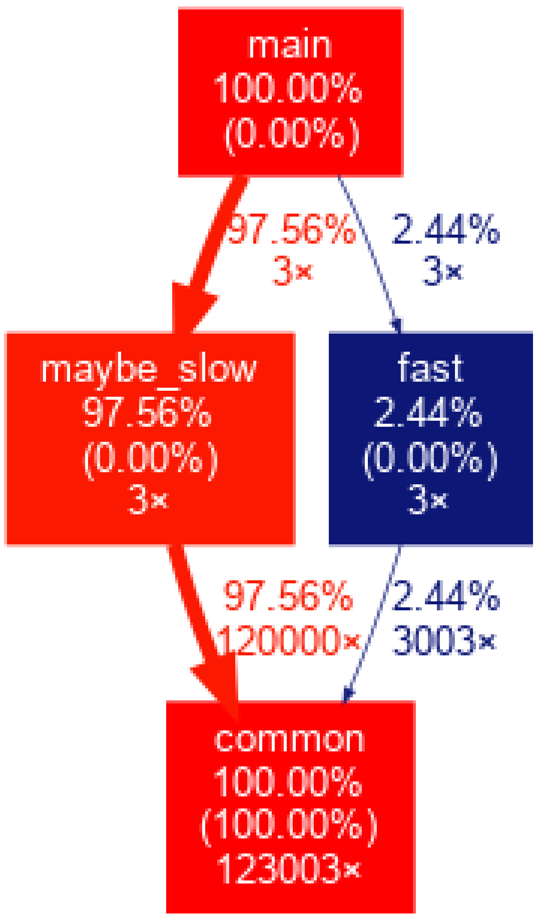
We can observe that file graphically with `gprof2dot` as asked at: Is it possible to get a graphical representation of gprof results?

```
sudo apt install graphviz
python3 -m pip install --user gprof2dot
gprof main.out > main.gprof
gprof2dot < main.gprof | dot -Tsvg -o output.svg
```

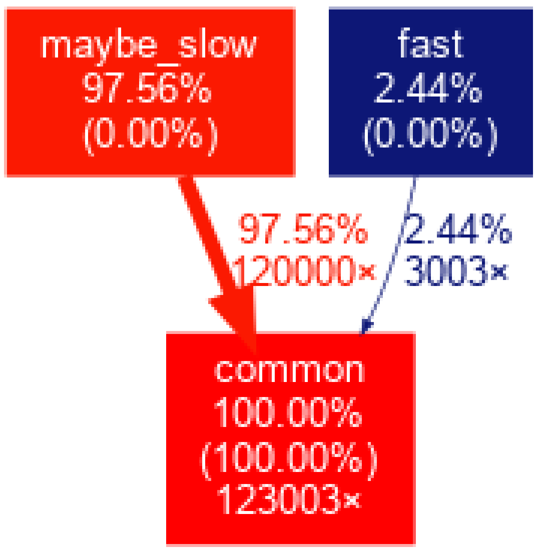
Here, the `gprof` tool reads the `gmon.out` trace information, and generates a human readable report in `main.gprof`, which `gprof2dot` then reads to generate a graph.

The source for gprof2dot is at: https://github.com/jrfonseca/gprof2dot

We observe the following for the `-O0` run:



and for the `-O3` run:



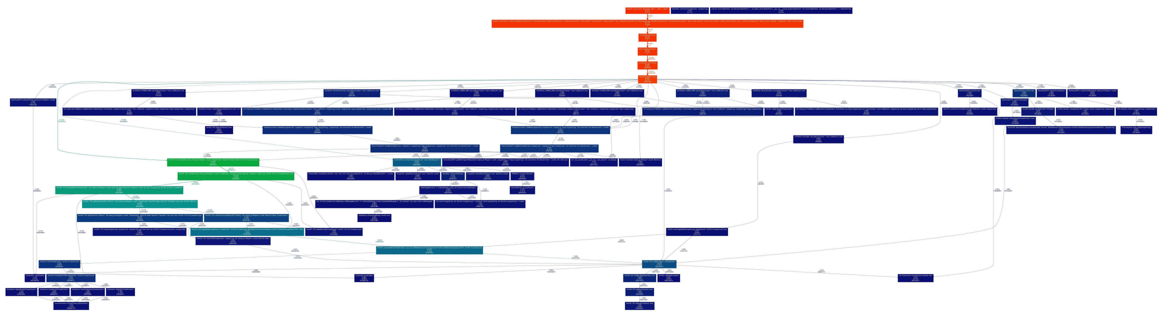
The `-O0` output is pretty much self-explanatory. For example, it shows that the 3 `maybe_slow` calls and their child calls take up 97.56% of the total runtime, although execution of `maybe_slow` itself without children accounts for 0.00% of the total execution time, i.e. almost all the time spent in that function was spent on child calls.

TODO: why is `main` missing from the `-O3` output, even though I can see it on a `bt` in GDB? Missing function from GProf output I think it is because gprof is also sampling based in addition to its compiled instrumentation, and the `-O3` `main` is just too fast and got no samples.

I choose SVG output instead of PNG because the SVG is searchable with Ctrl + F and the file size can be about 10x smaller. Also, the width and height of the generated image can be humoungous with tens of thousands of pixels for complex software, and GNOME `eog` 3.28.1 bugs out in that case for PNGs, while SVGs get opened by my browser automatically. gimp 2.8 worked well though, see also:

- https://askubuntu.com/questions/1112641/how-to-view-extremely-large-images
- https://unix.stackexchange.com/questions/77968/viewing-large-image-on-linux
- https://superuser.com/questions/356038/viewer-for-huge-images-under-linux-100-mp-color-images

but even then, you will be dragging the image around a lot to find what you want, see e.g. this image from a "real" software example taken from this ticket:



Can you find the most critical call stack easily with all those tiny unsorted spaghetti lines going over one another? There might be better `dot` options I'm sure, but I don't want to go there now. What we really need is a proper dedicated viewer for it, but I haven't found one yet:

- View gprof output in kcachegrind
- Which is the best replacement for KProf?

You can however use the color map to mitigate those problems a bit. For example, on the previous huge image, I finally managed to find the critical path on the left when I made the brilliant deduction that green comes after red, followed finally by darker and darker blue.

Alternatively, we can also observe the text output of the `gprof` built-in binutils tool which we previously saved at:

```
cat main.gprof
```

By default, this produces an extremely verbose output that explains what the output data means. Since I can't explain better than that, I'll let you read it yourself.

Once you have understood the data output format, you can reduce verbosity to show just the data without the tutorial with the `-b` option:

```
gprof -b main.out
```

In our example, outputs were for `-O0`:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call
100.35      3.67     3.67   123003     0.00     0.00
  0.00      3.67     0.00        3     0.00     0.03
  0.00      3.67     0.00        3     0.00     1.19

            Call graph


granularity: each sample hit covers 2 byte(s) for 0.27

index % time    self  children    called     name
                0.09    0.00    3003/123003      fast
                3.58    0.00  120000/123003      maybe
[1]    100.0    3.67    0.00  123003         common [1
-----------------------------------------------
```

```
                                                     <spon
[2]     100.0     0.00     3.67                  main [2]
                  0.00     3.58       3/3            maybe
                  0.00     0.09       3/3            fast
-----------------------------------------------
                  0.00     3.58       3/3            main
[3]      97.6     0.00     3.58       3          maybe_slo
                  3.58     0.00  120000/123003       commo
-----------------------------------------------
                  0.00     0.09       3/3            main
[4]       2.4     0.00     0.09       3          fast [4]
                  0.09     0.00    3003/123003       commo
-----------------------------------------------

Index by function name

   [1] common                      [4] fast
```

and for `-O3`:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call
100.52      1.84     1.84   123003    14.96    14.96

            Call graph


granularity: each sample hit covers 2 byte(s) for 0.54

index % time    self  children    called     name
                0.04    0.00    3003/123003      fast
                1.79    0.00  120000/123003      maybe
[1]     100.0   1.84    0.00  123003         common [1
-----------------------------------------------
                                                 <spon
[2]      97.6   0.00    1.79                 maybe_slo
                1.79    0.00  120000/123003      commo
-----------------------------------------------
                                                 <spon
```

```
[3]       2.4     0.00     0.04                 fast [3]
                  0.04     0.00    3003/123003      commo
-----------------------------------------------

Index by function name

   [1] common
```

As a very quick summary for each section e.g.:

```
                  0.00     3.58       3/3           main
[3]      97.6     0.00     3.58       3         maybe_slo
                  3.58     0.00  120000/123003      commo
```

centers around the function that is left indented (`maybe_slow`). `[3]` is the ID of that function. Above the function, are its callers, and below it the callees.

For `-O3`, see here like in the graphical output that `maybe_slow` and `fast` don't have a known parent, which is what the documentation says that `<spontaneous>` means.

I'm not sure if there is a nice way to do line-by-line profiling with gprof: [`gprof` time spent in particular lines of code]()

`perf` from `linux-tools`

`perf` works by sampling which part of the programming is running regularly. This makes it very simple to setup without any recompilation, it can even attach to programs that are already running. The downside is that we lose a bit of data compared to gprof instrumented code: notably we don't know how many times each function is called.

This section was tested on Ubuntu 23.10:

```
sudo apt install linux-tools-common linux-tools-generi
```

And then you also need:

```
sudo sysctl kernel.perf_event_paranoid=-1 kernel.kptr_
```

or to persist those options that across reboots:

```
printf 'kernel.perf_event_paranoid = -1
kernel.kptr_restrict = 0
' | sudo tee -a /etc/sysctl.conf
```

Collect data:

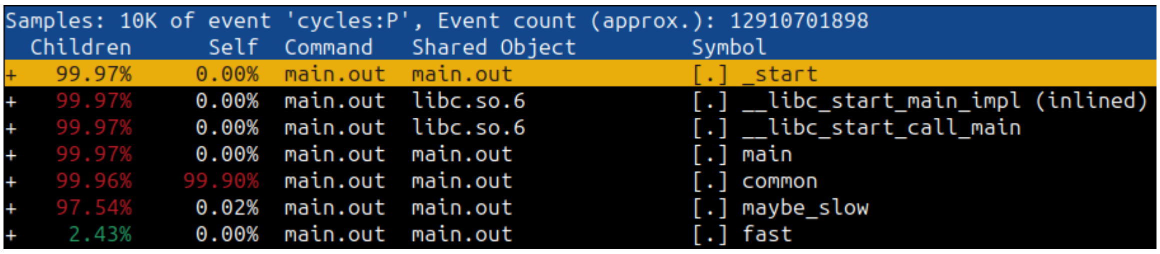
```
time perf record --call-graph dwarf ./main.out 10000
```

This produces a 42 MB file `perf.data` and adds a fixed 1 s overhead to execution, so it's fine time-wise. Ther `--call-graph dwarf` option is mentioned at: [Call stack in the perf profiler]()

Next we can inspect the data interactively with:

```
perf report
```

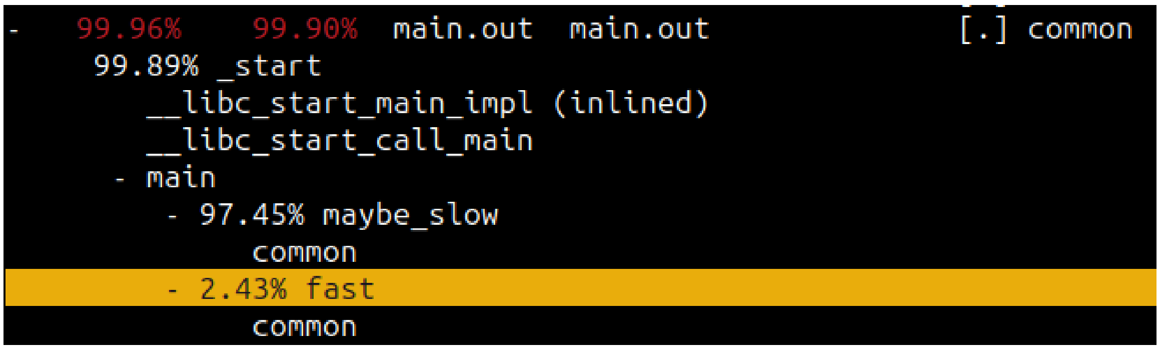
Initially this gives us a view:



```
Samples: 10K of event 'cycles:P', Event count (approx.): 12910701898
  Children      Self  Command   Shared Object          Symbol
+   99.97%     0.00%  main.out  main.out               [.] _start
+   99.97%     0.00%  main.out  libc.so.6              [.] __libc_start_main_impl (inlined)
+   99.97%     0.00%  main.out  libc.so.6              [.] __libc_start_call_main
+   99.97%     0.00%  main.out  main.out               [.] main
+   99.96%    99.90%  main.out  main.out               [.] common
+   97.54%     0.02%  main.out  main.out               [.] maybe_slow
+    2.43%     0.00%  main.out  main.out               [.] fast
```

TODO how to always sort by "Self"? [Sorting by self column in perf report]() It seemed to randomly sort by one column or another from time to time?

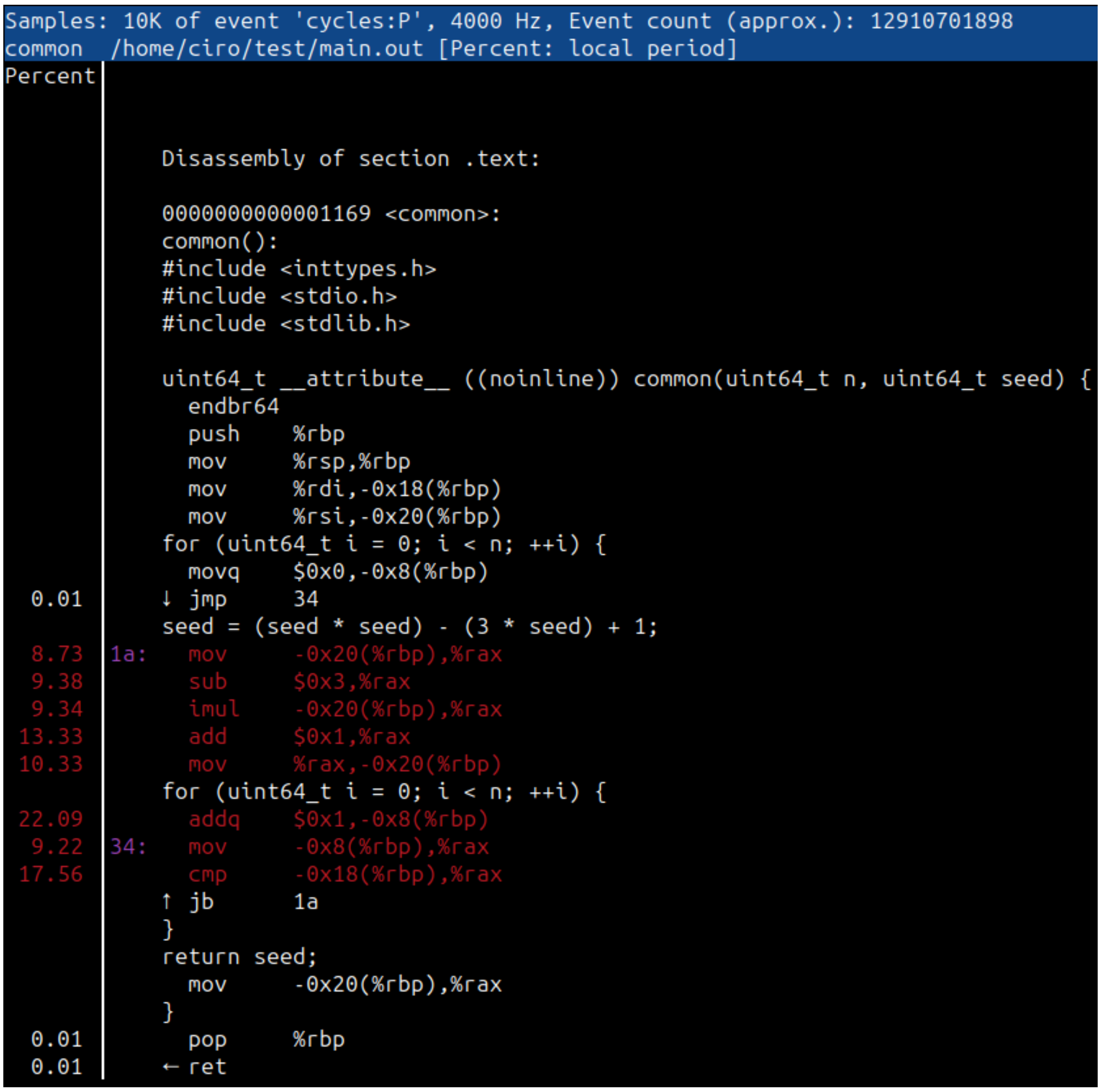
This immediately tells us that `common` is what takes up most of the execution time. If we select the "common" line and hit `+` (plus sign) or `e` (expand) and keep expanding recursively we finally see:



```
-   99.96%    99.90%  main.out  main.out               [.] common
     99.89% _start
        __libc_start_main_impl (inlined)
        __libc_start_call_main
      - main
         - 97.45% maybe_slow
              common
         - 2.43% fast
              common
```

which tells us exactly what we want: `maybe_slow` calls to `common` are the slow ones.

Another mega-cool thing we can do is to hit `a` (annotate) on the `common` line, which gives us this assembly level breakdown of the function

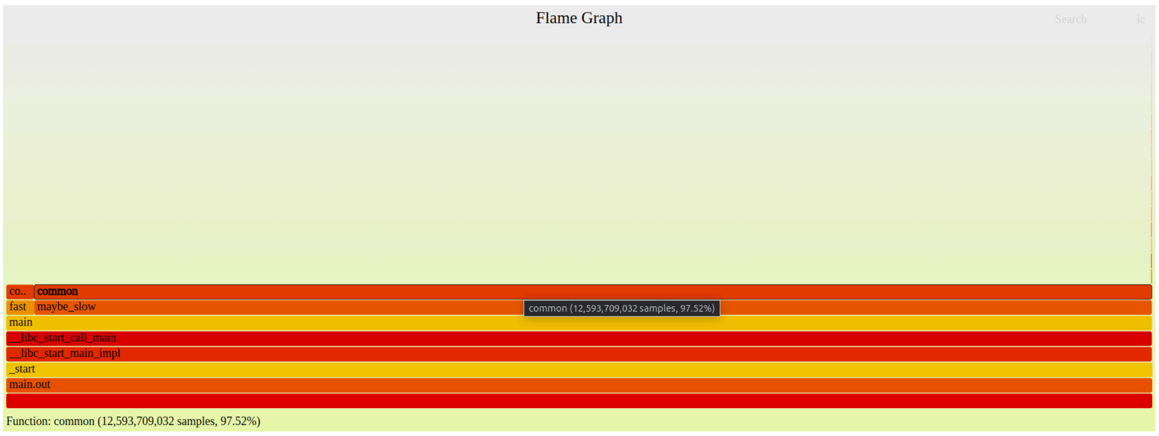


```
Samples: 10K of event 'cycles:P', 4000 Hz, Event count (approx.): 12910701898
common  /home/ciro/test/main.out [Percent: local period]
Percent│
       │
       │
       │      Disassembly of section .text:
       │
       │      0000000000001169 <common>:
       │      common():
       │      #include <inttypes.h>
       │      #include <stdio.h>
       │      #include <stdlib.h>
       │
       │      uint64_t __attribute__ ((noinline)) common(uint64_t n, uint64_t seed) {
       │        endbr64
       │        push   %rbp
       │        mov    %rsp,%rbp
       │        mov    %rdi,-0x18(%rbp)
       │        mov    %rsi,-0x20(%rbp)
       │      for (uint64_t i = 0; i < n; ++i) {
       │        movq   $0x0,-0x8(%rbp)
  0.01 │      ↓ jmp    34
       │      seed = (seed * seed) - (3 * seed) + 1;
  8.73 │1a:     mov    -0x20(%rbp),%rax
  9.38 │        sub    $0x3,%rax
  9.34 │        imul   -0x20(%rbp),%rax
 13.33 │        add    $0x1,%rax
 10.33 │        mov    %rax,-0x20(%rbp)
       │      for (uint64_t i = 0; i < n; ++i) {
 22.09 │        addq   $0x1,-0x8(%rbp)
  9.22 │34:     mov    -0x8(%rbp),%rax
 17.56 │        cmp    -0x18(%rbp),%rax
       │      ↑ jb     1a
       │      }
       │      return seed;
       │        mov    -0x20(%rbp),%rax
       │      }
  0.01 │        pop    %rbp
  0.01 │      ← ret
```

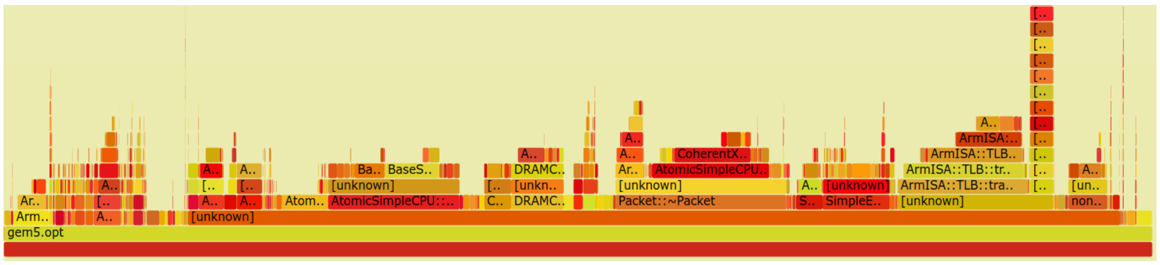
Another interesting visualization is with https://github.com/brendangregg/FlameGraph which is also mentioned at: http://www.brendangregg.com/perf.html#FlameGraphs

```
git clone https://github.com/brendangregg/FlameGraph
git -C FlameGraph/ checkout cd9ee4c4449775a2f867acf31c
perf script | FlameGraph/stackcollapse-perf.pl | Flame
flamegraph.svg
```

and we get:



On the a more complex example I got from another program it becomes clearer what the graph means:



TODO there are a log of `[unknown]` functions in that example, why is that?

Another perf GUI interfaces which might be worth it include:

- Eclipse Trace Compass plugin: https://www.eclipse.org/tracecompass/

  But this has the downside that you have to first convert the data to the Common Trace Format, which can be done with `perf data --to-ctf`, but it needs to be enabled at build time/have `perf` new enough, either of which is not the case for the perf in Ubuntu 18.04

- https://github.com/KDAB/hotspot

> The downside of this is that there seems to be no Ubuntu package, and building it requires Qt 5.10 while Ubuntu 18.04 is at Qt 5.9.
>
> But David Faure mentions in the comments that there is no an AppImage package which might be a convenient way to use it.

valgrind callgrind

valgrind runs the program through the valgrind virtual machine. This makes the profiling very accurate, but it also produces a very large slowdown of the program. I have also mentioned kcachegrind previously at: Tools to get a pictorial function call graph of code

callgrind is the valgrind's tool to profile code and kcachegrind is a KDE program that can visualize cachegrind output.

First we have to remove the `-pg` flag to go back to normal compilation, otherwise the run actually fails with `Profiling timer expired`, and yes, this is so common that I did and there was a Stack Overflow question for it.

So we compile and run as:

```
sudo apt install kcachegrind valgrind
gcc -ggdb3 -O3 -std=c99 -Wall -Wextra -pedantic -o mai
time valgrind --tool=callgrind valgrind --dump-instr=y
  --collect-jumps=yes ./main.out 10000
```

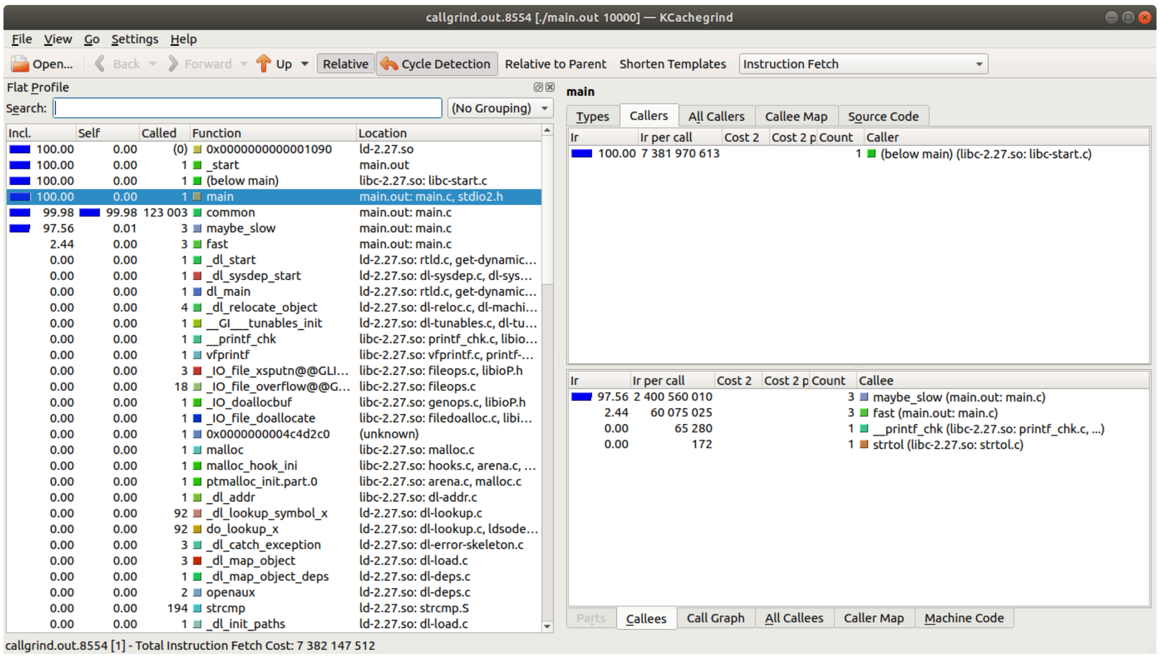
I enable `--dump-instr=yes --collect-jumps=yes` because this also dumps information that enables us to view a per assembly line breakdown of performance, at a relatively small added overhead cost.

Off the bat, `time` tells us that the program took 29.5 seconds to execute, so we had a slowdown of about 15x on this example. Clearly, this slowdown is going to be a serious limitation for larger workloads. On the "real world software example" mentioned here, I observed a slowdown of 80x.

The run generates a profile data file named `callgrind.out.<pid>` e.g. `callgrind.out.8554` in my case. We view that file with:

```
kcachegrind callgrind.out.8554
```

which shows a GUI that contains data similar to the textual gprof output:


callgrind.out.8554 [./main.out 10000] — KCachegrind
File View Go Settings Help
Open... Back Forward Up Relative Cycle Detection Relative to Parent Shorten Templates Instruction Fetch
Flat Profile
Search:
(No Grouping)
main
Types Callers All Callers Callee Map Source Code
Parts Callees Call Graph All Callees Caller Map Machine Code

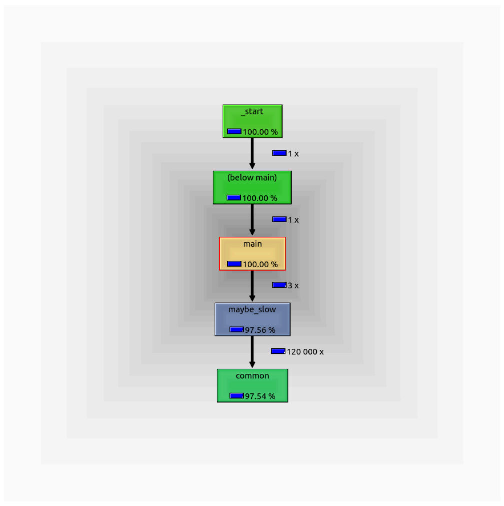
Incl.	Self	Called	Function	Location
100.00	0.00	(0)	0x0000000000001090	ld-2.27.so
100.00	0.00	1	_start	main.out
100.00	0.00	1	(below main)	libc-2.27.so: libc-start.c
100.00	0.00	1	main	main.out: main.c, stdio2.h
99.98	99.98	123 003	common	main.out: main.c
97.56	0.01	3	maybe_slow	main.out: main.c
2.44	0.00	3	fast	main.out: main.c
0.00	0.00	1	_dl_start	ld-2.27.so: rtld.c, get-dynamic…
0.00	0.00	1	_dl_sysdep_start	ld-2.27.so: dl-sysdep.c, dl-sys…
0.00	0.00	1	dl_main	ld-2.27.so: rtld.c, get-dynamic…
0.00	0.00	4	_dl_relocate_object	ld-2.27.so: dl-reloc.c, dl-machi…
0.00	0.00	1	__GI___tunables_init	ld-2.27.so: dl-tunables.c, dl-tu…
0.00	0.00	1	__printf_chk	libc-2.27.so: printf_chk.c, libio…
0.00	0.00	1	vfprintf	libc-2.27.so: vfprintf.c, printf-…
0.00	0.00	3	_IO_file_xsputn@@GLI…	libc-2.27.so: fileops.c, libioP.h
0.00	0.00	18	_IO_file_overflow@@G…	libc-2.27.so: fileops.c
0.00	0.00	1	_IO_doallocbuf	libc-2.27.so: genops.c, libioP.h
0.00	0.00	1	_IO_file_doallocate	libc-2.27.so: filedoalloc.c, libi…
0.00	0.00	1	0x00000000004c4d2c0	(unknown)
0.00	0.00	1	malloc	libc-2.27.so: malloc.c
0.00	0.00	1	malloc_hook_ini	libc-2.27.so: hooks.c, arena.c, …
0.00	0.00	1	ptmalloc_init.part.0	libc-2.27.so: arena.c, malloc.c
0.00	0.00	1	_dl_addr	libc-2.27.so: dl-addr.c
0.00	0.00	92	_dl_lookup_symbol_x	ld-2.27.so: dl-lookup.c
0.00	0.00	92	do_lookup_x	ld-2.27.so: dl-lookup.c, ldsode…
0.00	0.00	3	_dl_catch_exception	ld-2.27.so: dl-error-skeleton.c
0.00	0.00	3	_dl_map_object	ld-2.27.so: dl-load.c
0.00	0.00	1	_dl_map_object_deps	ld-2.27.so: dl-deps.c
0.00	0.00	2	openaux	ld-2.27.so: dl-deps.c
0.00	0.00	194	strcmp	ld-2.27.so: strcmp.S
0.00	0.00	1	_dl_init_paths	ld-2.27.so: dl-load.c

Ir	Ir per call	Cost 2	Cost 2 p	Count	Caller
100.00	7 381 970 613			1	(below main) (libc-2.27.so: libc-start.c)

Ir	Ir per call	Cost 2	Cost 2 p	Count	Callee
97.56	2 400 560 010			3	maybe_slow (main.out: main.c)
2.44	60 075 025			3	fast (main.out: main.c)
0.00	65 280			1	__printf_chk (libc-2.27.so: printf_chk.c, ...)
0.00	172			1	strtol (libc-2.27.so: strtol.c)

callgrind.out.8554 [1] - Total Instruction Fetch Cost: 7 382 147 512

Also, if we go on the bottom right "Call Graph" tab, we see a call graph which we can export by right clicking it to obtain the following image with unreasonable amounts of white border :-)



I think `fast` is not showing on that graph because kcachegrind must have simplified the visualization because that call takes up too little time, this will likely be the behavior you want on a real program. The right click menu has some settings to control when to cull such nodes, but I couldn't get it to show such a short call after a quick attempt. If I click on `fast` on the left window, it does show a call graph with `fast`, so that stack was actually captured. No one had yet found a way to show the complete graph call graph: Make callgrind show all function calls in the kcachegrind callgraph

TODO on complex C++ software, I see some entries of type `<cycle N>`, e.g. `<cycle 11>` where I'd expect function names, what does that mean? I noticed there is a "Cycle Detection" button to toggle that on and off, but what does it mean?

gperftools

Previously called "Google Performance Tools", source: https://github.com/gperftools/gperftools Sample based.

First install gperftools with:

```
sudo apt install google-perftools
```

Then, we can enable the gperftools CPU profiler in two ways: at runtime, or at build time.

At runtime, we have to pass set the `LD_PRELOAD` to point to `libprofiler.so`, which you can find with `locate libprofiler.so`, e.g. on my system:

```
gcc -ggdb3 -O3 -std=c99 -Wall -Wextra -pedantic -o mai
LD_PRELOAD=/usr/lib/x86_64-linux-gnu/libprofiler.so \
  CPUPROFILE=prof.out ./main.out 10000
```

Alternatively, we can build the library in at link time, dispensing passing `LD_PRELOAD` at runtime:

```
gcc -Wl,--no-as-needed,-lprofiler,--as-needed -ggdb3 -
-pedantic -o main.out main.c
CPUPROFILE=prof.out ./main.out 10000
```

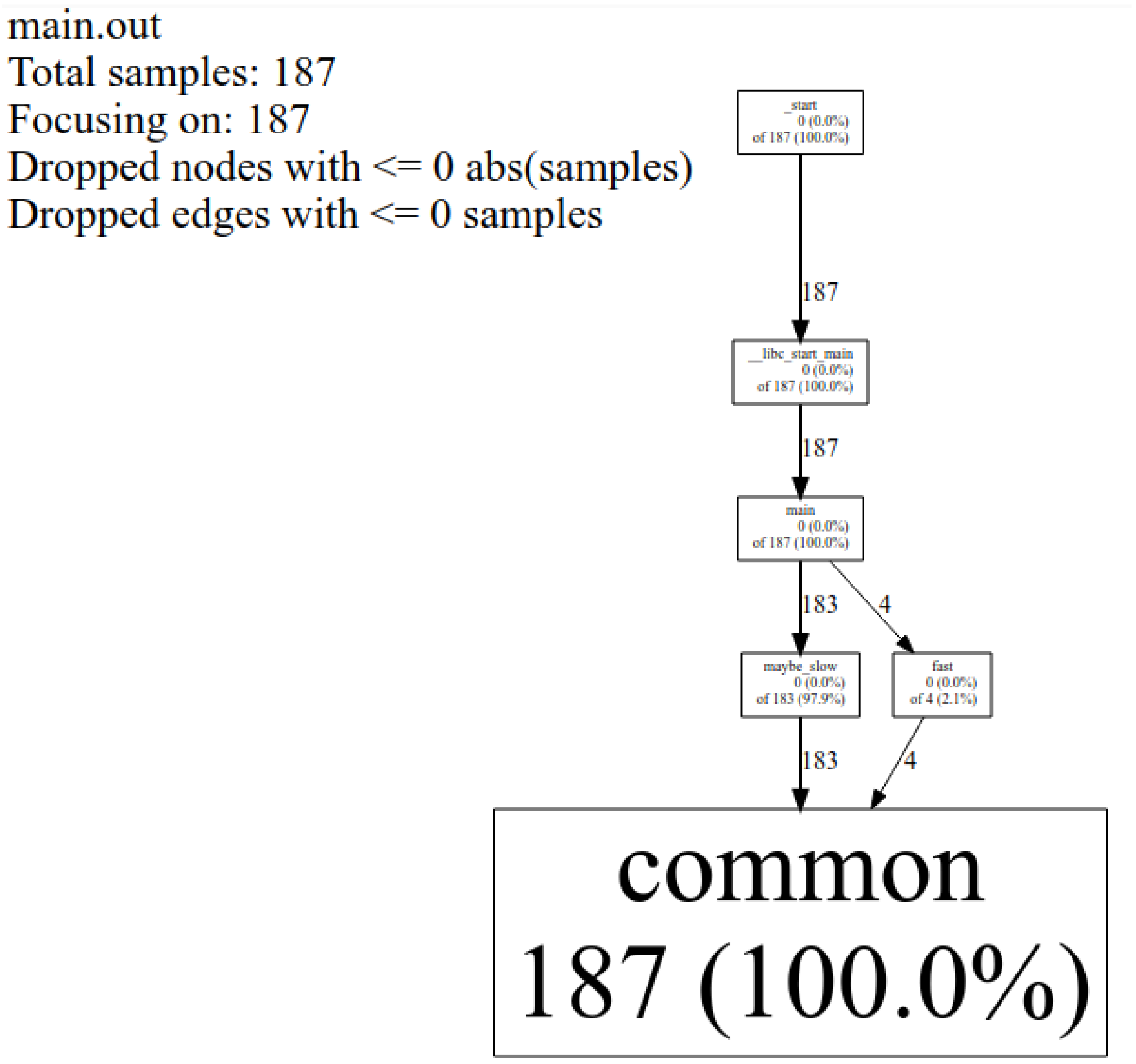
See also: gperftools - profile file not dumped

The nicest way to view this data I've found so far is to make pprof output the same format that kcachegrind takes as input (yes, the Valgrind-project-viewer-tool) and use kcachegrind to view that:

```
google-pprof --callgrind main.out prof.out  > callgrin
kcachegrind callgrind.out
```

After running with either of those methods, we get a `prof.out` profile data file as output. We can view that file graphically as an SVG with:

```
google-pprof --web main.out prof.out
```



```
main.out
Total samples: 187
Focusing on: 187
Dropped nodes with <= 0 abs(samples)
Dropped edges with <= 0 samples
```

which gives as a familiar call graph like other tools, but with the clunky unit of number of samples rather than seconds.

Alternatively, we can also get some textual data with:

```
google-pprof --text main.out prof.out
```

which gives:

```
Using local file main.out.
Using local file prof.out.
Total: 187 samples
     187 100.0% 100.0%      187 100.0% common
       0   0.0% 100.0%      187 100.0% __libc_start_ma
       0   0.0% 100.0%      187 100.0% _start
       0   0.0% 100.0%        4   2.1% fast
       0   0.0% 100.0%      187 100.0% main
       0   0.0% 100.0%      183  97.9% maybe_slow
```

See also: How to use google perf tools

Instrument your code with raw `perf_event_open` syscalls

I think this is the same underlying subsystem that `perf` uses, but you could of course attain even greater control by explicitly instrumenting your program at compile time with events of interest.

This is likely too hardcore for most people, but it's kind of fun. Minimal runnable example at: Quick way to count number of instructions executed in a C program

Intel VTune

https://en.wikipedia.org/wiki/VTune

This seems to be closed source and x86-only, but it is likely to be amazing from what I've heard. I'm not sure how free it is to use, but it seems to be free to download. TODO evaluate.

Tested in Ubuntu 18.04, gprof2dot 2019.11.30, valgrind 3.13.0, perf 4.15.18, Linux kernel 4.15.0, FLameGraph 1a0dc6985aad06e76857cf2a354bd5ba0c9ce96b, gperftools 2.5-2.

Share Improve this answer Follow

edited Feb 10 at 6:54

answered Feb 17, 2020 at 15:15

Ciro Santilli OurBigBook.com
380k ● 117 ● 1.3k ● 1.1k

---

6 By default perf record uses the frame pointer register. Modern compilers don't record the frame address and instead use the register as a general purpose. The alternative is to compile with `-fno-omit-frame-pointer` flag or use a different alternative: record with `--call-graph "dwarf"` or `--call-graph "lbr"` depending on your scenario. – Jorge Bellon Mar 30, 2020 at 15:27

---

1 KDAB's hotspot ships with an AppImage these days, making it really easy to use. – David Faure Jun 28, 2022 at 16:23

Great answer. Side note, `[unknown]` frames might be caused by stack too deep if DWARF method is used --- although I haven't investigated very carefully. gist.github.com/user202729/b6ea134715ddf851efd0047b3197fc33 has some resources. – user202729 Feb 2 at 13:58

Great answer. But seems like gprof can't profile shared libraries. – Alex Che Feb 28 at 23:33

For `gprof` I found that `gprof main.out > main.gprof` gives the error `gmon.out: "not in executable format"`, and the program was required as an input for the call, i.e. `gprof ./myprog main.out > main.gprof` – Wolfie Oct 1 at 8:30

88

I would use Valgrind and Callgrind as a base for my profiling tool suite. What is important to know is that Valgrind is basically a Virtual Machine:

> (wikipedia) Valgrind is in essence a virtual machine using just-in-time (JIT) compilation techniques, including dynamic recompilation. Nothing from the original program ever gets run directly on the host processor. Instead, Valgrind first translates the program into a temporary, simpler form called Intermediate Representation (IR), which is a processor-neutral, SSA-based form. After the conversion, a tool (see below) is free to do whatever transformations it would like on the IR, before Valgrind translates the IR back

> into machine code and lets the host processor run it.

Callgrind is a profiler build upon that. Main benefit is that you don't have to run your aplication for hours to get reliable result. Even one second run is sufficient to get rock-solid, reliable results, because Callgrind is a **non-probing** profiler.

Another tool build upon Valgrind is Massif. I use it to profile heap memory usage. It works great. What it does is that it gives you snapshots of memory usage -- detailed information WHAT holds WHAT percentage of memory, and WHO had put it there. Such information is available at different points of time of application run.

Share Improve this answer Follow

answered May 22, 2009 at 21:44

anon

68

This is a response to Nazgob's Gprof answer.

I've been using Gprof the last couple of days and have already found three significant limitations, one of which I've not seen documented anywhere else (yet):

1. It doesn't work properly on multi-threaded code, unless you use a workaround
2. The call graph gets confused by function pointers. Example: I have a function called `multithread()`

which enables me to multi-thread a specified function over a specified array (both passed as arguments). Gprof however, views all calls to `multithread()` as equivalent for the purposes of computing time spent in children. Since some functions I pass to `multithread()` take much longer than others my call graphs are mostly useless. (To those wondering if threading is the issue here: no, `multithread()` can optionally, and did in this case, run everything sequentially on the calling thread only).

3. It says here that "... the number-of-calls figures are derived by counting, not sampling. They are completely accurate...". Yet I find my call graph giving me 5345859132+784984078 as call stats to my most-called function, where the first number is supposed to be direct calls, and the second recursive calls (which are all from itself). Since this implied I had a bug, I put in long (64-bit) counters into the code and did the same run again. My counts: 5345859132 direct, and 78094395406 self-recursive calls. There are a lot of digits there, so I'll point out the recursive calls I measure are 78bn, versus 784m from Gprof: a factor of 100 different. Both runs were single threaded and unoptimised code, one compiled `-g` and the other `-pg`.

This was GNU Gprof (GNU Binutils for Debian) 2.18.0.20080103 running under 64-bit Debian Lenny, if that helps anyone.

Share Improve this answer

Follow

edited Sep 15, 2018 at 9:33

HugoTeixeira
4,884 ● 3 ● 25 ● 33

answered Jun 30, 2011 at 19:30

Rob_before_edits
1,173 ● 10 ● 13

---

1 Yes, it does sampling, but not for number-of-calls figures. Interestingly, following your link ultimately led me to an updated version of the manual page I linked to in my post, new URL: sourceware.org/binutils/docs/gprof/… This repeats the quote in part (iii) of my answer, but also says "In multi-threaded applications, or single threaded applications that link with multi-threaded libraries, the counts are only deterministic if the counting function is thread-safe. (Note: beware that the mcount counting function in glibc is not thread-safe)." – Rob_before_edits Jun 22, 2012 at 4:30

1 It is not clear to me if this explains my result in (iii). My code was linked -lpthread -lm and declared both a "pthread_t *thr" and a "pthread_mutex_t nextLock = PTHREAD_MUTEX_INITIALIZER" static variable even when it was running single threaded. I would ordinarily presume that "link with multi-threaded libraries" means actually using those libraries, and to a greater extent than this, but I could be wrong! – Rob_before_edits Jun 22, 2012 at 6:05

33

Use Valgrind, Callgrind and KCacheGrind:

```
valgrind --tool=callgrind ./(Your binary)
```

generates file *callgrind.out.x*. Read it using KCacheGrind.

Use gprof (add `-pg`):

```
cc -o myprog myprog.c utils.c -g -pg
```

(not so good for multi-threads, function pointers)

Use google-perftools:

Uses time sampling, I/O and CPU bottlenecks are revealed.

Intel VTune is the best (free for educational purposes).

Others: AMD CodeAnalyst (since replaced with AMD CodeXL), OProfile, and 'perf' tools (`apt-get install linux-tools`)

Share Improve this answer

Follow

edited Dec 3 at 0:27

Peter Mortensen

31.6k ● 22 ● 109 ● 133

answered Feb 23, 2017 at 21:28

Ehsan

1,418 ● 15 ● 13

8

Also worth mentioning are

1. HPCToolkit (http://hpctoolkit.org/) - Open-source, works for parallel programs and has a GUI with which to look at the results multiple ways

2. Intel VTune (https://software.intel.com/en-us/vtune) - If you have intel compilers this is very good
3. TAU (http://www.cs.uoregon.edu/research/tau/home.php)

I have used HPCToolkit and VTune and they are very effective at finding the long pole in the tent and do not need your code to be recompiled (except that you have to use -g -O or RelWithDebInfo type build in CMake to get meaningful output). I have heard TAU is similar in capabilities.

Share Improve this answer Follow

answered Sep 14, 2018 at 22:56

raovgarimella
166 ● 1 ● 8

7

For single-threaded programs you can use **igprof**, The Ignominous Profiler: https://igprof.org/ .

It is a sampling profiler, along the lines of the... long... answer by Mike Dunlavey, which will gift wrap the results in a browsable call stack tree, annotated with the time or memory spent in each function, either cumulative or per-function.

Share Improve this answer Follow

answered Mar 17, 2018 at 12:20

fwyzard
2,667 ● 1 ● 22 ● 22

It looks interesting, but fails to compile with GCC 9.2. (Debian/Sid) I made an issue on github.

– Basile Starynkevitch Jan 11, 2020 at 21:47

5

Actually a bit surprised not many mentioned about google/benchmark , while it is a bit cumbersome to pin the specific area of code, specially if the code base is a little big one, however I found this really helpful when used in combination with `callgrind`

IMHO identifying the piece that is causing bottleneck is the key here. I'd however try and answer the following questions first and choose tool based on that

1. is my algorithm correct ?
2. are there locks that are proving to be bottle necks ?
3. is there a specific section of code that's proving to be a culprit ?
4. how about IO, handled and optimized ?

`valgrind` with the combination of `callgrind` and `kcachegrind` should provide a decent estimation on the points above, and once it's established that there are issues with some section of code, I'd suggest to do a micro bench mark - `google benchmark` is a good place to start.

Share Improve this answer Follow

edited Jun 25, 2021 at 19:58
Moritz Barsnick
35 ● 1 ● 14

answered Nov 3, 2019 at 14:47

asio_guy
3,767 ● 3 ● 21 ● 35

I found that my google benchmark numbers looked more accurate than gprof when i measured sections of code. As you said its really good for micro benchmarking. but If you want a more holistic picture you need a different approach. – Rahul Ravindran Jun 19, 2020 at 16:22

4

These are the two methods I use for speeding up my code:

For CPU bound applications:

1. Use a profiler in DEBUG mode to identify questionable parts of your code
2. Then switch to RELEASE mode and comment out the questionable sections of your code (stub it with nothing) until you see changes in performance.

For I/O bound applications:

1. Use a profiler in RELEASE mode to identify questionable parts of your code.

N.B.

If you don't have a profiler, use the poor man's profiler. Hit pause while debugging your application. Most developer suites will break into assembly with commented line numbers. You're statistically likely to land in a region that is eating most of your CPU cycles.

For CPU, the reason for profiling in **DEBUG** mode is because if your tried profiling in **RELEASE** mode, the compiler is going to reduce math, vectorize loops, and inline functions which tends to glob your code into an un-mappable mess when it's assembled. **An un-mappable mess means your profiler will not be able to clearly identify what is taking so long because the assembly may not correspond to the source code under optimization**. If you need the performance (e.g. timing sensitive) of **RELEASE** mode, disable debugger features as needed to keep a usable performance.

For I/O-bound, the profiler can still identify I/O operations in **RELEASE** mode because I/O operations are either externally linked to a shared library (most of the time) or in the worst case, will result in a sys-call interrupt vector (which is also easily identifiable by the profiler).

Share Improve this answer Follow

edited Sep 5, 2015 at 23:35

answered Nov 28, 2013 at 18:21

seo
1,969 ● 24 ● 18

2 +1 The poor man's method works just as well for I/O bound as for CPU bound, and I recommend doing all performance tuning in DEBUG mode. When you're finished tuning, then turn on RELEASE. It will make an improvement if the

program is CPU-bound in your code. *Here's a crude but short video of the process.* – Mike Dunlavey Jun 27, 2014 at 20:55

4 I wouldn't use DEBUG builds for performance profiling. Often have I seen that performance critical parts in DEBUG mode are completely optimized away in release mode. Another problem is the use of asserts in debug code which add noise to the performance. – gast128 Jul 21, 2014 at 18:55

3 Did you read my post at all? "If you need the performance (e.g. timing sensitive) of RELEASE mode, disable debugger features as needed to keep a usable performance", "Then switch to RELEASE mode and comment the questionable sections of your code (Stub it with nothing) until you see changes in performance."? I said check for possible problem areas in debug mode and verify those problems in release mode to avoid the pitfall you mentioned. – seo Jul 22, 2014 at 15:54

4

You can use the iprof library:

https://gitlab.com/Neurochrom/iprof

https://github.com/Neurochrom/iprof

It's cross-platform and allows you not to measure performance of your application also in real-time. You can even couple it with a live graph. Full disclaimer: I am the author.

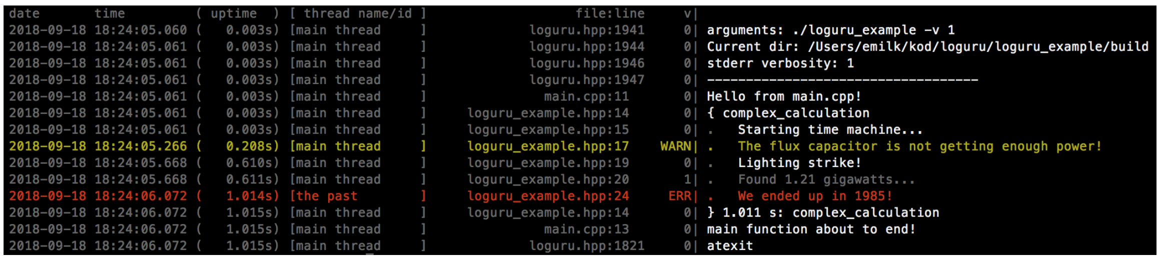
Share Improve this answer Follow

answered Feb 24, 2019 at 18:01

N3UR0CHR0M
313 ● 2 ● 5

4

You can use a logging framework like `loguru` since it includes timestamps and total uptime which can be used nicely for profiling:



```
date       time         ( uptime  ) [ thread name/id ]                   file:line     v|
2018-09-18 18:24:05.060 (   0.003s) [main thread     ]             loguru.hpp:1941     0| arguments: ./loguru_example -v 1
2018-09-18 18:24:05.061 (   0.003s) [main thread     ]             loguru.hpp:1944     0| Current dir: /Users/emilk/kod/loguru/loguru_example/build
2018-09-18 18:24:05.061 (   0.003s) [main thread     ]             loguru.hpp:1946     0| stderr verbosity: 1
2018-09-18 18:24:05.061 (   0.003s) [main thread     ]             loguru.hpp:1947     0| -----------------------------------
2018-09-18 18:24:05.061 (   0.003s) [main thread     ]              main.cpp:11        0| Hello from main.cpp!
2018-09-18 18:24:05.061 (   0.003s) [main thread     ]      loguru_example.hpp:14      0| { complex_calculation
2018-09-18 18:24:05.061 (   0.003s) [main thread     ]      loguru_example.hpp:15      0| .   Starting time machine...
2018-09-18 18:24:05.266 (   0.208s) [main thread     ]      loguru_example.hpp:17   WARN| .   The flux capacitor is not getting enough power!
2018-09-18 18:24:05.668 (   0.610s) [main thread     ]      loguru_example.hpp:19      0| .   Lighting strike!
2018-09-18 18:24:05.668 (   0.611s) [main thread     ]      loguru_example.hpp:20      1| .   Found 1.21 gigawatts...
2018-09-18 18:24:06.072 (   1.014s) [the past        ]      loguru_example.hpp:24    ERR| .   We ended up in 1985!
2018-09-18 18:24:06.072 (   1.015s) [main thread     ]      loguru_example.hpp:14      0| } 1.011 s: complex_calculation
2018-09-18 18:24:06.072 (   1.015s) [main thread     ]              main.cpp:13        0| main function about to end!
2018-09-18 18:24:06.072 (   1.015s) [main thread     ]             loguru.hpp:1821     0| atexit
```

Share Improve this answer Follow

answered May 21, 2019 at 13:28

BullyWiiPlaza
19.1k ● 15 ● 131 ● 206

1 what does this have to do with profiling? – sehe Jun 28, 2023 at 13:45

@sehe: You can see how long what takes by the log timestamps – BullyWiiPlaza Jun 28, 2023 at 14:42

3

At work we have a really nice tool that helps us monitoring what we want in terms of scheduling. This has been useful numerous times.

It's in C++ and must be customized to your needs. Unfortunately I can't share code, just concepts. You use a "large" `volatile` buffer containing timestamps and event ID that you can dump post mortem or after stopping the logging system (and dump this into a file for example).

You retrieve the so-called large buffer with all the data and a small interface parses it and shows events with name (up/down + value) like an oscilloscope does with colors (configured in `.hpp` file).

You customize the amount of events generated to focus solely on what you desire. It helped us a lot for scheduling issues while consuming the amount of CPU we wanted based on the amount of logged events per second.

You need 3 files :

```
toolname.hpp // interface
toolname.cpp // code
tool_events_id.hpp // Events ID
```

The concept is to define events in `tool_events_id.hpp` like that :

```
// EVENT_NAME                                  ID       BEGIN_EN
#define SOCK_PDU_RECV_D                        0x0301   //@D0030
#define SOCK_PDU_RECV_F                        0x0302   //@F0030
```

You also define a few functions in `toolname.hpp` :

```
#define LOG_LEVEL_ERROR 0
#define LOG_LEVEL_WARN 1
// ...

void init(void);
void probe(id,payload);
// etc
```

Wherever in you code you can use :

```
toolname<LOG_LEVEL>::log(EVENT_NAME,VALUE);
```

The `probe` function uses a few assembly lines to retrieve the clock timestamp ASAP and then sets an entry in the buffer. We also have an atomic increment to safely find an index where to store the log event. Of course buffer is circular.

Hope the idea is not obfuscated by the lack of sample code.

Share Improve this answer Follow

answered May 17, 2019 at 10:13

SOKS
190 ● 1 ● 3 ● 11

3

As no one mentioned Arm MAP, I'd add it as personally I have successfully used Map to profile a C++ scientific program.

Arm MAP is the profiler for parallel, multithreaded or single threaded C, C++, Fortran and F90 codes. It provides in-depth analysis and bottleneck pinpointing to the source line. Unlike most profilers, it's designed to be able to profile pthreads, OpenMP or MPI for parallel and threaded code.

MAP is commercial software.

Share Improve this answer Follow

answered Jun 28, 2019 at 4:44

Wei
738 ● 1 ● 6 ● 19

2

There are several tools available for this purpose. Each has its own unique features that other lacks.

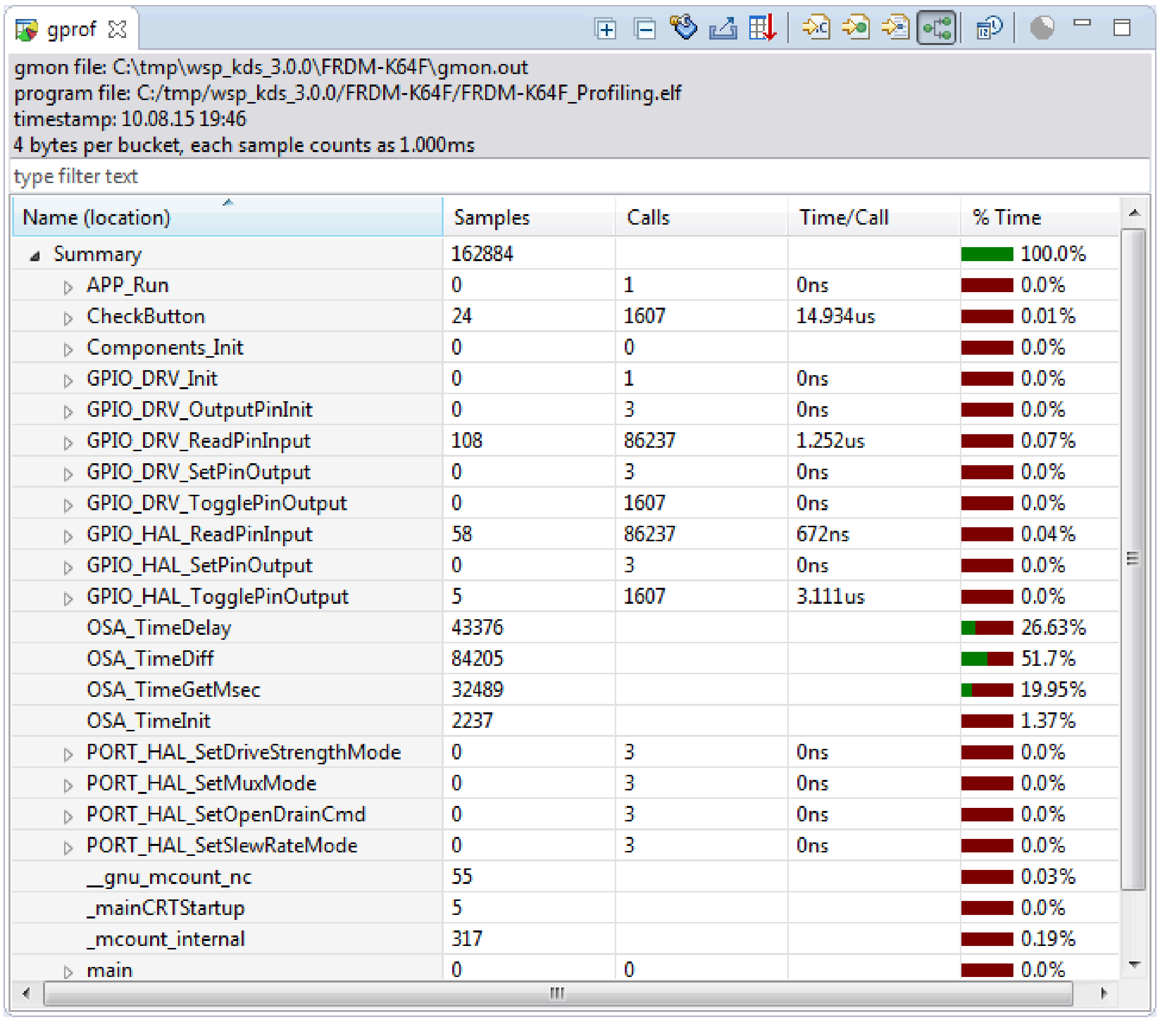
Gprof

- a profiling tool that comes with the GNU Compiler Collection (GCC).
- It also provides information on the time spent in each function of your program.
- **Usage:**

```
g++ -pg -o my_program my_program.cpp
./my_program
gprof ./my_program
```

- **Web** => gprof - GNU Project

Screenshot:



gmon file: C:\tmp\wsp_kds_3.0.0\FRDM-K64F\gmon.out
program file: C:/tmp/wsp_kds_3.0.0/FRDM-K64F/FRDM-K64F_Profiling.elf
timestamp: 10.08.15 19:46
4 bytes per bucket, each sample counts as 1.000ms

type filter text

Name (location)	Samples	Calls	Time/Call	% Time
Summary	162884			100.0%
APP_Run	0	1	0ns	0.0%
CheckButton	24	1607	14.934us	0.01%
Components_Init	0	0		0.0%
GPIO_DRV_Init	0	1	0ns	0.0%
GPIO_DRV_OutputPinInit	0	3	0ns	0.0%
GPIO_DRV_ReadPinInput	108	86237	1.252us	0.07%
GPIO_DRV_SetPinOutput	0	3	0ns	0.0%
GPIO_DRV_TogglePinOutput	0	1607	0ns	0.0%
GPIO_HAL_ReadPinInput	58	86237	672ns	0.04%
GPIO_HAL_SetPinOutput	0	3	0ns	0.0%
GPIO_HAL_TogglePinOutput	5	1607	3.111us	0.0%
OSA_TimeDelay	43376			26.63%
OSA_TimeDiff	84205			51.7%
OSA_TimeGetMsec	32489			19.95%
OSA_TimeInit	2237			1.37%
PORT_HAL_SetDriveStrengthMode	0	3	0ns	0.0%
PORT_HAL_SetMuxMode	0	3	0ns	0.0%
PORT_HAL_SetOpenDrainCmd	0	3	0ns	0.0%
PORT_HAL_SetSlewRateMode	0	3	0ns	0.0%
__gnu_mcount_nc	55			0.03%
_mainCRTStartup	5			0.0%
_mcount_internal	317			0.19%
main	0	0		0.0%

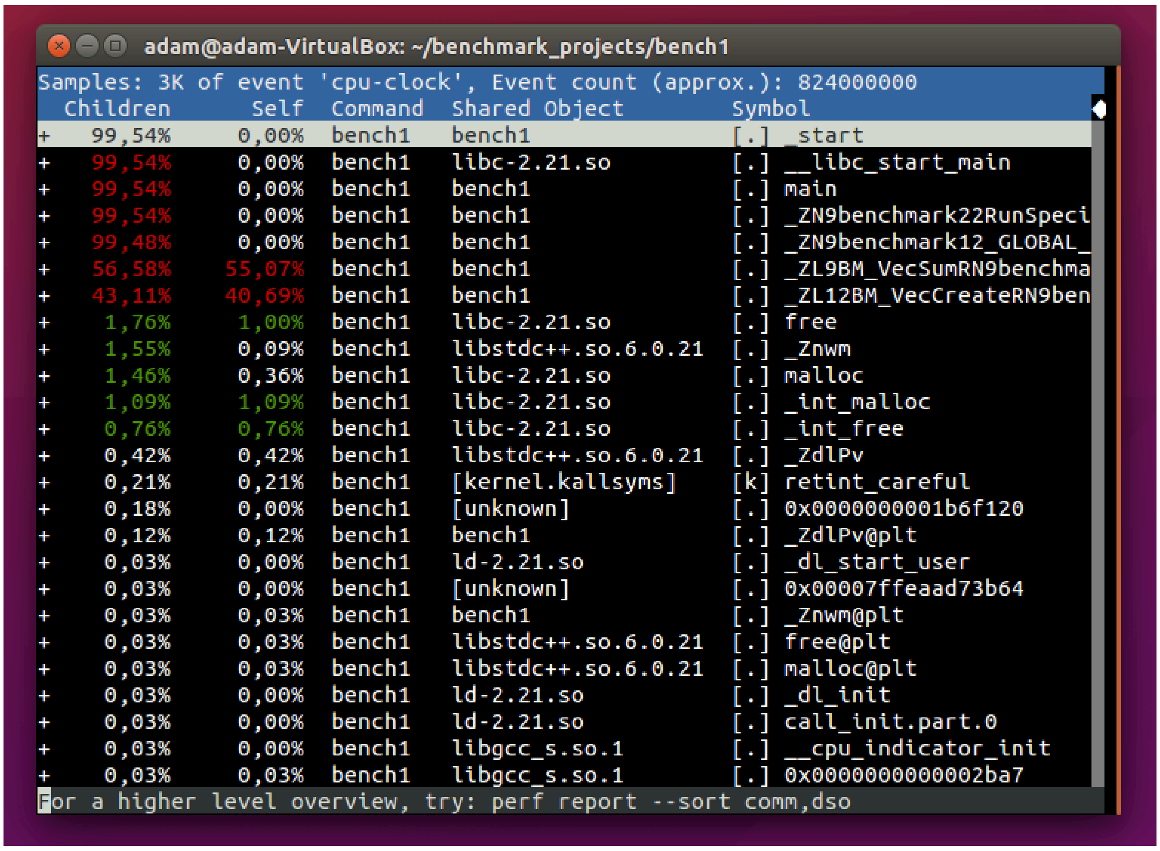
2. **perf:**

- **Description:** perf is a powerful performance analysis tool that is part of the Linux kernel. It provides a wide range of features, including CPU performance counters, tracepoints, and more.
- **Usage:**

```
perf record ./my_program
perf report
```

- **Web** => perf - Linux man page

Screenshot:



```
adam@adam-VirtualBox: ~/benchmark_projects/bench1
Samples: 3K of event 'cpu-clock', Event count (approx.): 824000000
  Children      Self  Command  Shared Object          Symbol
+   99,54%     0,00%  bench1   bench1                 [.] _start
+   99,54%     0,00%  bench1   libc-2.21.so           [.] __libc_start_main
+   99,54%     0,00%  bench1   bench1                 [.] main
+   99,54%     0,00%  bench1   bench1                 [.] _ZN9benchmark22RunSpeci
+   99,48%     0,00%  bench1   bench1                 [.] _ZN9benchmark12_GLOBAL_
+   56,58%    55,07%  bench1   bench1                 [.] _ZL9BM_VecSumRN9benchma
+   43,11%    40,69%  bench1   bench1                 [.] _ZL12BM_VecCreateRN9ben
+    1,76%     1,00%  bench1   libc-2.21.so           [.] free
+    1,55%     0,09%  bench1   libstdc++.so.6.0.21    [.] _Znwm
+    1,46%     0,36%  bench1   libc-2.21.so           [.] malloc
+    1,09%     1,09%  bench1   libc-2.21.so           [.] _int_malloc
+    0,76%     0,76%  bench1   libc-2.21.so           [.] _int_free
+    0,42%     0,42%  bench1   libstdc++.so.6.0.21    [.] _ZdlPv
+    0,21%     0,21%  bench1   [kernel.kallsyms]      [k] retint_careful
+    0,18%     0,00%  bench1   [unknown]              [.] 0x0000000001b6f120
+    0,12%     0,12%  bench1   bench1                 [.] _ZdlPv@plt
+    0,03%     0,00%  bench1   ld-2.21.so             [.] _dl_start_user
+    0,03%     0,00%  bench1   [unknown]              [.] 0x00007ffeaad73b64
+    0,03%     0,03%  bench1   bench1                 [.] _Znwm@plt
+    0,03%     0,03%  bench1   libstdc++.so.6.0.21    [.] free@plt
+    0,03%     0,03%  bench1   libstdc++.so.6.0.21    [.] malloc@plt
+    0,03%     0,00%  bench1   ld-2.21.so             [.] _dl_init
+    0,03%     0,00%  bench1   ld-2.21.so             [.] call_init.part.0
+    0,03%     0,00%  bench1   libgcc_s.so.1          [.] __cpu_indicator_init
+    0,03%     0,03%  bench1   libgcc_s.so.1          [.] 0x0000000000002ba7
For a higher level overview, try: perf report --sort comm,dso
```

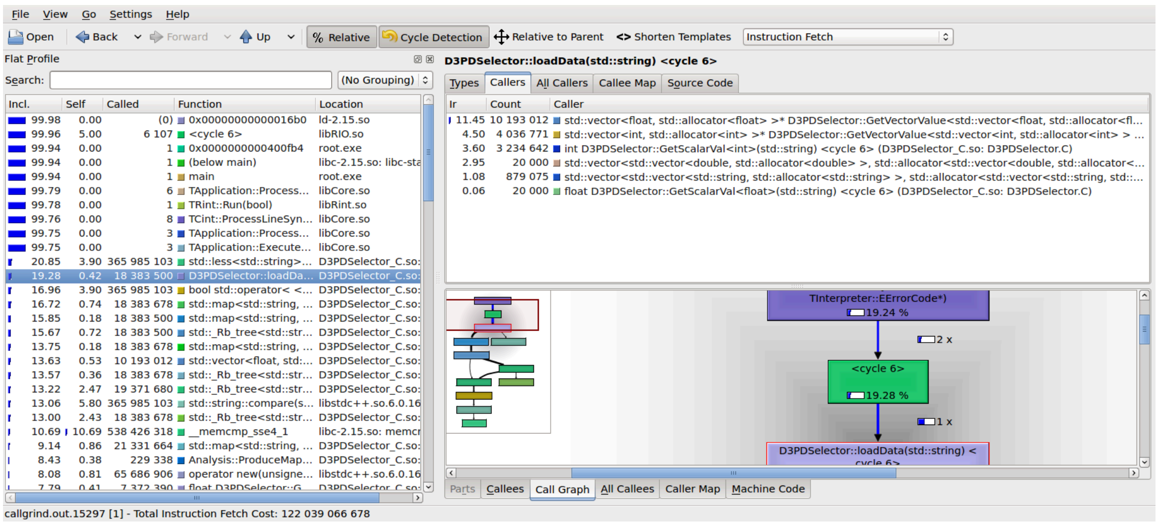
3. **Valgrind:**

- **Description:** a programming tool for memory debugging, memory leak detection, and profiling.
- **Usage:**

```
valgrind --tool=callgrind ./my_program
```

- **Web** Valgrind - Callgrind

Screenshot:



```
Incl.   Self   Called       Function                     Location
99.98   0.00          (0)   0x00000000000016b0           ld-2.15.so
99.96   5.00        6 107   <cycle 6>                    libRIO.so
99.94   0.00            1   0x0000000000400fb4           root.exe
99.94   0.00            1   (below main)                 libc-2.15.so: libc-sta
99.94   0.00            1   main                         root.exe
99.79   0.00            6   TApplication::Process...     libCore.so
99.78   0.00            1   TRint::Run(bool)             libRint.so
99.76   0.00            8   TCint::ProcessLineSyn...     libCore.so
99.75   0.00            3   TApplication::Process...     libCore.so
99.75   0.00            3   TApplication::Execute...     libCore.so
20.85   3.90  365 985 103   std::less<std::string>...    D3PDSelector_C.so:
19.28   0.42   18 383 500   D3PDSelector::loadDa...      D3PDSelector_C.so:
16.96   3.90  365 985 103   bool std::operator< <...     D3PDSelector_C.so:
16.72   0.74   18 383 678   std::map<std::string, ...    D3PDSelector_C.so:
15.85   0.18   18 383 500   std::map<std::string, ...    D3PDSelector_C.so:
15.67   0.72   18 383 500   std::_Rb_tree<std::str...    D3PDSelector_C.so:
13.75   0.18   18 383 678   std::map<std::string, ...    D3PDSelector_C.so:
13.63   0.53   10 193 012   std::vector<float, std:...   D3PDSelector_C.so:
13.57   0.36   18 383 678   std::_Rb_tree<std::str...    D3PDSelector_C.so:
13.22   2.47   19 371 680   std::_Rb_tree<std::str...    D3PDSelector_C.so:
13.06   5.80  365 985 103   std::string::compare(s...    libstdc++.so.6.0.16
13.00   2.43   18 383 678   std::_Rb_tree<std::str...    D3PDSelector_C.so:
10.69  10.69  538 426 318   __memcmp_sse4_1              libc-2.15.so: memcr
 9.14   0.86   21 331 664   std::map<std::string, ...    D3PDSelector_C.so:
 8.43   0.38      229 338   Analysis::ProduceMap...      D3PDSelector_C.so:
 8.08   0.81   65 686 906   operator new(unsigne...      libstdc++.so.6.0.16
 7.79   0.41    7 372 390   float D3PDSelector::G...     D3PDSelector_C.so:
```

D3PDSelector::loadData(std::string) <cycle 6>

```
Ir     Count        Caller
11.45  10 193 012   std::vector<float, std::allocator<float> >* D3PDSelector::GetVectorValue<std::vector<float, std::allocator<fl...
 4.50   4 036 771   std::vector<int, std::allocator<int> >* D3PDSelector::GetVectorValue<std::vector<int, std::allocator<int> > ...
 3.60   3 234 642   int D3PDSelector::GetScalarVal<int>(std::string) <cycle 6> (D3PDSelector_C.so: D3PDSelector.C)
 2.95      20 000   std::vector<std::vector<double, std::allocator<double> >, std::allocator<std::vector<double, std::allocator<...
 1.08     879 075   std::vector<std::vector<std::string, std::allocator<std::string> >, std::allocator<std::vector<std::string, std::...
 0.06      20 000   float D3PDSelector::GetScalarVal<float>(std::string) <cycle 6> (D3PDSelector_C.so: D3PDSelector.C)
```

callgrind.out.15297 [1] - Total Instruction Fetch Cost: 122 039 066 678

4. **Google Performance Tools (gperftools):**

   - **Description:** gperftools is a collection of performance tools by Google. It includes CPU and heap profilers (pprof and heap-checker), among others.
   - **Usage:**

     ```
     LD_PRELOAD=/path/to/libtcmalloc.so CPUPROFILE=
     pprof --text ./my_program ./my_program.prof
     ```

   - **Web** gperftools - GitHub

5. **KCachegrind:**

   - **Description:** KCachegrind is a visual front-end for profiling data generated by Callgrind. It provides an interactive GUI for analyzing the profiling information.
   - **Web** KCachegrind Handbook

6. **Clang's AddressSanitizer and UndefinedBehaviorSanitizer:**

   - **Description:** These sanitizers can help you find memory issues and undefined behavior in your code, which can indirectly lead to performance problems.

- **Web** AddressSanitizer

Share Improve this answer Follow

edited Dec 3 at 0:14
desertnaut
60.2k ● 31 ● 151 ● 176

answered Nov 10, 2023 at 16:15
Muhammad Ali
1,300 ● 3 ● 20 ● 23

1

Use debugging software

How can we identify where the code is running slowly?

I just think you have an obstacle. While you are in motion, then it will decrease your speed.

Like that unwanted reallocation's looping, buffer overflows, searching, memory leaks, etc. Operations consume more execution power, and it will affect adversely over the performance of the code, Be sure to add `-pg` to the compilation before profiling:

`g++ your_prg.cpp -pg` or `cc my_program.cpp -g -pg` as per your compiler

I haven't tried it yet, but I've heard good things about google-perftools. It is definitely worth a try.

`valgrind --tool=callgrind ./(Your binary)`

It will generate a file called *gmon.out* or *callgrind.out.x*. You can then use kcachegrind or a debugger tool to read this file. It will give you a graphical analysis of things with results like which lines cost how much.

i think so.

Share Improve this answer

Follow

edited Dec 2 at 22:04

Peter Mortensen
31.6k ● 22 ● 109 ● 133

answered Apr 18, 2020 at 19:23

user10306682

1 I actually would suggest adding some optimization flag, e.g. compile with `g++ -O -pg -Wall your_prg.cpp` – Basile Starynkevitch Sep 9, 2020 at 13:07

0

Use `-pg` flag when compiling and linking the code and run the executable file. While this program is executed, profiling data is collected in the file a.out.
There is two different type of profiling

1- Flat profiling:
by running the command `gprog --flat-profile a.out` you got the following data
- what percentage of the overall time was spent for the function,
- how many seconds were spent in a function—including and excluding calls to sub-functions,

- the number of calls,
- the average time per call.

2- graph profiling
us the command `gprof --graph a.out` to get the following data for each function which includes
- In each section, one function is marked with an index number.
- Above function , there is a list of functions that call the function .
- Below function , there is a list of functions that are called by the function .

To get more info you can look in
https://sourceware.org/binutils/docs-2.32/gprof/