GPROF Tutorial – How to use Linux GNU GCC Profiling Tool

by Himanshu Arora on August 10, 2012

Profiling is an important aspect of software programming. Through profiling one can determine the parts in program code that are time consuming and need to be re-written. This helps make your program execution faster which is always desired.

In very large projects, profiling can save your day by not only determining the parts in your program which are slower in execution than expected but also can help you find many other statistics through which many potential bugs can be spotted and sorted out.

In this article, we will explore the GNU profiling tool 'gprof'.

How to use gprof

Using the gprof tool is not at all complex. You just need to do the following on a high-level:

- Have profiling enabled while compiling the code
- Execute the program code to produce the profiling data
- Run the gprof tool on the profiling data file (generated in the step above).

The last step above produces an analysis file which is in human readable form. This file contains a couple of tables (flat profile and call graph) in addition to some other information. While flat profile gives an overview of the timing information of the functions like time consumption for the execution of a particular function, how many times it was called etc. On the other hand, call graph focuses on each function like the functions through which a particular function was called, what all functions were called from within this particular function etc So this way one can get idea of the execution time spent in the sub-routines too.

Lets try and understand the three steps listed above through a practical example. Following test code will be used throughout the article :

```c
//test_gprof.c
#include<stdio.h>

void new_func1(void);

void func1(void)
{
    printf("\n Inside func1 \n");
    int i = 0;

    for(;i<0xffffffff;i++);
    new_func1();

    return;
}

static void func2(void)
{
    printf("\n Inside func2 \n");
    int i = 0;

    for(;i<0xffffffaa;i++);
    return;
}

int main(void)
{
    printf("\n Inside main()\n");
    int i = 0;

    for(;i<0xffffff;i++);
    func1();
    func2();

    return 0;
}
```

```c
//test_gprof_new.c
#include<stdio.h>

void new_func1(void)
{
    printf("\n Inside new_func1()\n");
    int i = 0;

    for(;i<0xffffffee;i++);

    return;
}
```

Note that the ‘for’ loops inside the functions are there to consume some execution time.

Step-1 : Profiling enabled while compilation

In this first step, we need to make sure that the profiling is enabled when the compilation of the code is done. This is made possible by adding the ‘-pg’ option in the compilation step.

From the man page of gcc :

> -pg : Generate extra code to write profile information suitable for the analysis program gprof. You must use this option when compiling the source files you want data about, and you must also use it when linking.

So, lets compile our code with '-pg' option :

```
$ gcc -Wall -pg test_gprof.c test_gprof_new.c -o test_gprof
$
```

Please note : The option '-pg' can be used with the gcc command that compiles (-c option), gcc command that links(-o option on object files) and with gcc command that does the both(as in example above).

Step-2 : Execute the code

In the second step, the binary file produced as a result of step-1 (above) is executed so that profiling information can be generated.

```
$ ls
test_gprof  test_gprof.c  test_gprof_new.c

$ ./test_gprof

 Inside main()

 Inside func1

 Inside new_func1()

 Inside func2

$ ls
gmon.out  test_gprof  test_gprof.c  test_gprof_new.c

$
```

So we see that when the binary was executed, a new file 'gmon.out' is generated in the current working directory.

Note that while execution if the program changes the current working directory (using chdir) then gmon.out will be produced in the new current working directory. Also, your program needs to have sufficient permissions for gmon.out to be created in current working directory.

Step-3 : Run the gprof tool

In this step, the gprof tool is run with the executable name and the above generated ‘gmon.out’ as argument. This produces an analysis file which contains all the desired profiling information.

```
$  gprof test_gprof gmon.out > analysis.txt
```

Note that one can explicitly specify the output file (like in example above) or the information is produced on stdout.

```
$ ls
analysis.txt  gmon.out  test_gprof  test_gprof.c  test_gprof_new.c
```

So we see that a file named ‘analysis.txt’ was generated.

On a related note, you should also understand how to debug your C program using gdb.

Comprehending the profiling information

As produced above, all the profiling information is now present in ‘analysis.txt’. Lets have a look at this text file :

```
Flat profile:

Each sample counts as 0.01 seconds.
%    cumulative self          self   total
time seconds    seconds calls s/call s/call name
33.86 15.52     15.52    1    15.52  15.52  func2
33.82 31.02     15.50    1    15.50  15.50  new_func1
33.29 46.27     15.26    1    15.26  30.75  func1
0.07  46.30     0.03                        main

% the percentage of the total running time of the
time program used by this function.

cumulative a running sum of the number of seconds accounted
seconds for by this function and those listed above it.

self the number of seconds accounted for by this
seconds function alone. This is the major sort for this
listing.

calls the number of times this function was invoked, if
this function is profiled, else blank.

self the average number of milliseconds spent in this
ms/call function per call, if this function is profiled,
else blank.

total the average number of milliseconds spent in this
ms/call function and its descendents per call, if this
function is profiled, else blank.

name the name of the function. This is the minor sort
for this listing. The index shows the location of
the function in the gprof listing. If the index is
in parenthesis it shows where it would appear in
the gprof listing if it were to be printed.

Call graph (explanation follows)

granularity: each sample hit covers 2 byte(s) for 0.02% of 46.30 seconds

index % time self children called name

[1]   100.0  0.03  46.27          main [1]
             15.26 15.50    1/1      func1 [2]
             15.52 0.00     1/1      func2 [3]
-----------------------------------------------
             15.26 15.50    1/1      main [1]
```

```
[2]   66.4   15.26 15.50    1     func1 [2]
             15.50 0.00     1/1      new_func1 [4]
-----------------------------------------------
             15.52 0.00     1/1      main [1]
[3]   33.5   15.52 0.00     1     func2 [3]
-----------------------------------------------
             15.50 0.00     1/1      func1 [2]
[4] 33.5     15.50 0.00     1     new_func1 [4]
-----------------------------------------------
```

This table describes the call tree of the program, and was sorted by
the total amount of time spent in each function and its children.

Each entry in this table consists of several lines. The line with the
index number at the left hand margin lists the current function.
The lines above it list the functions that called this function,
and the lines below it list the functions this one called.
This line lists:
index A unique number given to each element of the table.
Index numbers are sorted numerically.
The index number is printed next to every function name so
it is easier to look up where the function in the table.

% time This is the percentage of the `total' time that was spent
in this function and its children. Note that due to
different viewpoints, functions excluded by options, etc,
these numbers will NOT add up to 100%.

self This is the total amount of time spent in this function.

children This is the total amount of time propagated into this
function by its children.

called This is the number of times the function was called.
If the function called itself recursively, the number
only includes non-recursive calls, and is followed by
a `+' and the number of recursive calls.

name The name of the current function. The index number is
printed after it. If the function is a member of a
cycle, the cycle number is printed between the
function's name and the index number.

For the function's parents, the fields have the following meanings:

self This is the amount of time that was propagated directly
from the function into this parent.

children This is the amount of time that was propagated from
the function's children into this parent.

called This is the number of times this parent called the
function `/' the total number of times the function
was called. Recursive calls to the function are not
included in the number after the `/'.

name This is the name of the parent. The parent's index
number is printed after it. If the parent is a
member of a cycle, the cycle number is printed between
the name and the index number.

If the parents of the function cannot be determined, the word
`' is printed in the `name' field, and all the other
fields are blank.

For the function's children, the fields have the following meanings:

self This is the amount of time that was propagated directly
from the child into the function.

children This is the amount of time that was propagated from the

```
child's children to the function.

called This is the number of times the function called
this child `/' the total number of times the child
was called. Recursive calls by the child are not
listed in the number after the `/'.

name This is the name of the child. The child's index
number is printed after it. If the child is a
member of a cycle, the cycle number is printed
between the name and the index number.

If there are any cycles (circles) in the call graph, there is an
entry for the cycle-as-a-whole. This entry shows who called the
cycle (as parents) and the members of the cycle (as children.)
The `+' recursive calls entry shows the number of function calls that
were internal to the cycle, and the calls entry for each member shows,
for that member, how many times it was called from other members of
the cycle.

Index by function name

[2] func1 [1] main
[3] func2 [4] new_func1
```

So (as already discussed) we see that this file is broadly divided into two parts :

1. Flat profile
2. Call graph

The individual columns for the (flat profile as well as call graph) are very well explained in the output itself.

Customize gprof output using flags

There are various flags available to customize the output of the gprof tool. Some of them are discussed below:

1. Suppress the printing of statically(private) declared functions using -a

If there are some static functions whose profiling information you do not require then this can be achieved using -a option :

```
$ gprof -a test_gprof gmon.out > analysis.txt
```

Now if we see that analysis file :

```
Flat profile:

Each sample counts as 0.01 seconds.
%        cumulative self           self    total
time  seconds       seconds calls  s/call  s/call  name
67.15 30.77         30.77     2    15.39  23.14    func1
33.82 46.27         15.50     1    15.50  15.50    new_func1
0.07   46.30         0.03                          main

...
...
...

Call graph (explanation follows)

granularity: each sample hit covers 2 byte(s) for 0.02% of 46.30 seconds

index   %time        self  children  called  name

[1]     100.0        0.03   46.27             main [1]
                     30.77  15.50     2/2      func1 [2]
-----------------------------------------------------
```

```
                     30.77  15.50     2/2      main [1]
[2]     99.9         30.77  15.50     2      func1 [2]
                     15.50   0.00     1/1      new_func1 [3]
----------------------------------------------------
                     15.50   0.00     1/1      func1 [2]
[3]        33.5      15.50 0.00       1      new_func1 [3]
-----------------------------------------------

...
...
...
```

So we see that there is no information related to func2 (which is defined static)

2. Suppress verbose blurbs using -b

As you would have already seen that gprof produces output with lot of verbose information so in case this information is not required then this can be achieved using the -b flag.

```
$ gprof -b test_gprof gmon.out > analysis.txt
```

Now if we see the analysis file :

```
Flat profile:

Each sample counts as 0.01 seconds.
%       cumulative    self           self    total
time    seconds       seconds  calls  s/call  s/call   name
33.86 15.52            15.52      1    15.52  15.52    func2
33.82 31.02            15.50      1    15.50  15.50    new_func1
33.29 46.27            15.26      1    15.26  30.75    func1
0.07   46.30            0.03                           main

Call graph

granularity: each sample hit covers 2 byte(s) for 0.02% of 46.30 seconds
index % time self children called name

[1]   100.0  0.03  46.27          main [1]
             15.26 15.50    1/1      func1 [2]
             15.52 0.00     1/1      func2 [3]
-----------------------------------------------
             15.26 15.50    1/1      main [1]
[2]   66.4   15.26 15.50    1     func1 [2]
             15.50 0.00     1/1      new_func1 [4]
-----------------------------------------------
             15.52 0.00     1/1      main [1]
[3]   33.5   15.52 0.00     1     func2 [3]
-----------------------------------------------
             15.50 0.00     1/1      func1 [2]
[4] 33.5     15.50 0.00     1     new_func1 [4]
-----------------------------------------------
Index by function name

[2] func1 [1] main
[3] func2 [4] new_func1
```

So we see that all the verbose information is not present in the analysis file.

3. Print only flat profile using -p

In case only flat profile is required then :

```
$ gprof -p -b test_gprof gmon.out > analysis.txt
```

Note that I have used(and will be using) -b option so as to avoid extra information in analysis output.

Now if we see that analysis output:

```
Flat profile:

Each sample counts as 0.01 seconds.
%       cumulative    self            self   total
time    seconds       seconds  calls  s/call  s/call  name
33.86   15.52          15.52      1   15.52   15.52    func2
33.82   31.02          15.50      1   15.50   15.50    new_func1
33.29   46.27          15.26      1   15.26   30.75    func1
0.07    46.30          0.03                            main
```

So we see that only flat profile was there in the output.

4. Print information related to specific function in flat profile

This can be achieved by providing the function name along with the -p option:

```
$ gprof -pfunc1 -b test_gprof gmon.out > analysis.txt
```

Now if we see that analysis output :

```
Flat profile:

Each sample counts as 0.01 seconds.
%          cumulative     self            self     total
time       seconds        seconds  calls  s/call   s/call  name
103.20     15.26          15.26     1     15.26   15.26    func1
```

So we see that a flat profile containing information related to only function func1 is displayed.

5. Suppress flat profile in output using -P

If flat profile is not required then it can be suppressed using the -P option :

```
$ gprof -P -b test_gprof gmon.out > analysis.txt
```

Now if we see the analysis output :

```
Call graph

granularity: each sample hit covers 2 byte(s) for 0.02% of 46.30 seconds

index % time self children called name

[1]   100.0  0.03  46.27          main [1]
             15.26 15.50    1/1      func1 [2]
             15.52 0.00     1/1      func2 [3]
-----------------------------------------------
             15.26 15.50    1/1      main [1]
[2]   66.4   15.26 15.50    1     func1 [2]
             15.50 0.00     1/1      new_func1 [4]
-----------------------------------------------
             15.52 0.00     1/1      main [1]
[3]   33.5   15.52 0.00     1     func2 [3]
-----------------------------------------------
             15.50 0.00     1/1      func1 [2]
[4] 33.5     15.50 0.00     1     new_func1 [4]
-----------------------------------------------
Index by function name

[2] func1 [1] main
[3] func2 [4] new_func1
```

So we see that flat profile was suppressed and only call graph was displayed in output.

Also, if there is a requirement to print flat profile but excluding a particular function then this is also possible using -P flag by passing the function name (to exclude) along with it.

```
$ gprof -Pfunc1 -b test_gprof gmon.out > analysis.txt
```

In the above example, we tried to exclude 'func1' by passing it along with the -P option to gprof. Now lets see the analysis output:

```
Flat profile:

Each sample counts as 0.01 seconds.
%         cumulative      self              self    total
time      seconds         seconds   calls   s/call  s/call   name
50.76     15.52            15.52      1     15.52   15.52    func2
50.69     31.02            15.50      1     15.50   15.50    new_func1
0.10      31.05            0.03                              main
```

So we see that flat profile was displayed but information on func1 was suppressed.

6. Print only call graph information using -q

```
gprof -q -b test_gprof gmon.out > analysis.txt
```

In the example above, the option -q was used. Lets see what effect it casts on analysis output:

```
Call graph

granularity: each sample hit covers 2 byte(s) for 0.02% of 46.30 seconds
index % time self children called name

[1]   100.0  0.03  46.27          main [1]
             15.26 15.50    1/1      func1 [2]
             15.52 0.00     1/1      func2 [3]
-----------------------------------------------
             15.26 15.50    1/1      main [1]
[2]   66.4   15.26 15.50    1     func1 [2]
             15.50 0.00     1/1      new_func1 [4]
-----------------------------------------------
             15.52 0.00     1/1      main [1]
[3]   33.5   15.52 0.00     1     func2 [3]
-----------------------------------------------
             15.50 0.00     1/1      func1 [2]
[4] 33.5     15.50 0.00     1     new_func1 [4]
-----------------------------------------------
Index by function name

[2] func1 [1] main
[3] func2 [4] new_func1
```

So we see that only call graph was printed in the output.

7. Print only specific function information in call graph.

This is possible by passing the function name along with the -q option.

```
$ gprof -qfunc1 -b test_gprof gmon.out > analysis.txt
```

Now if we see the analysis output:

```
Call graph

granularity: each sample hit covers 2 byte(s) for 0.02% of 46.30 seconds
index % time self children called name

             15.26 15.50    1/1      main [1]
[2]   66.4   15.26 15.50    1     func1 [2]
             15.50 0.00     1/1      new_func1 [4]
```

```
-----------------------------------------------
                15.50 0.00     1/1      func1 [2]
[4]    33.5    15.50 0.00     1      new_func1 [4]
-----------------------------------------------
Index by function name

[2] func1 (1) main
(3) func2 [4] new_func1
```

So we see that information related to only func1 was displayed in call graph.

8. Suppress call graph using -Q

If the call graph information is not required in the analysis output then -Q option can be used.

```
$ gprof -Q -b test_gprof gmon.out > analysis.txt
```

Now if we see the analysis output :

```
Flat profile:

Each sample counts as 0.01 seconds.
%       cumulative    self            self    total
time    seconds       seconds   calls  s/call  s/call    name
33.86 15.52            15.52      1   15.52   15.52     func2
33.82 31.02            15.50      1   15.50   15.50     new_func1
33.29 46.27            15.26      1   15.26   30.75     func1
0.07   46.30            0.03                            main
```

So we see that only flat profile is there in the output. The whole call graph got suppressed.

Also, if it is desired to suppress a specific function from call graph then this can be achieved by passing the desired function name along with the -Q option to the gprof tool.

```
$ gprof -Qfunc1 -b test_gprof gmon.out > analysis.txt
```

In the above example, the function name func1 is passed to the -Q option.

Now if we see the analysis output:

```
Call graph

granularity: each sample hit covers 2 byte(s) for 0.02% of 46.30 seconds
index % time self children called name

[1]    100.0  0.03   46.27          main [1]
              15.26 15.50     1/1      func1 [2]
              15.52 0.00      1/1      func2 [3]
-----------------------------------------------
              15.52 0.00      1/1      main [1]
[3]    33.5   15.52 0.00      1      func2 [3]
-----------------------------------------------
              15.50 0.00      1/1      func1 [2]
[4]    33.5   15.50 0.00      1      new_func1 [4]
-----------------------------------------------
Index by function name

(2) func1 [1] main
[3] func2 [4] new_func1
```

So we see that call graph information related to func1 was suppressed.

If you enjoyed this article, you might also like..

1. 50 Linux Sysadmin Tutorials
2. 50 Most Frequently Used Linux Commands (With Examples)
3. Top 25 Best Linux Performance Monitoring and Debugging Tools
4. Mommy, I found it! – 15 Practical Linux Find Command Examples
5. Linux 101 Hacks 2nd Edition eBook Free

- Awk Introduction – 7 Awk Print Examples
- Advanced Sed Substitution Examples
- 8 Essential Vim Editor Navigation Fundamentals
- 25 Most Frequently Used Linux IPTables Rules Examples
- Turbocharge PuTTY with 12 Powerful Add-Ons

Tagged as: Gprof Call Graph

{ 25 comments… add one }

- Ethan August 10, 2012, 7:16 pm

  It is easy to use,clearly and instructive.

  Link

- Dario August 13, 2012, 8:28 am

  Very interesting, didn't know about this command!
  Thank you very much, keep up the good work!

  Link
- Mike Dunlavey October 2, 2012, 1:10 pm

  Please stop recommending GPROF. It's authors never even said it was much good for finding bottlenecks, and it isn't.
  Some more reasons: here, and here.

  Link
- Rasna October 17, 2013, 11:17 pm

  Could you please add details about how to get gmon.out for a running process?

  Link
- Antoha Bikineev October 30, 2013, 10:12 am

  I think it is better to make bigger time discrete in functions. In flat profile we expect that func1 works longer than func2. But because difference in loops is small, every time program executes we can get different sort results in flat profile. Sorry for bad English and thank you very much for this article

  Link
- Mark Richter February 13, 2014, 12:13 pm

  I have been using gprof to isolate a performance issue in a large scale business application, but recent attempts to do this have stalled. What we're seeing is that at the end of the program's execution, the CPU hangs at 100% utilization in the program, and it either takes hours (or days) to finish, or it never finishes.

  This never happens when the non-profiling version is run.

  We typically use SLES 11 for the build base, and either SLES 11 or RHEL 6 for the execution. This always happens on RHEL 6, and with large datasets on SLES 11.

  Why would the profiling version hang up at the end of program execution like that?

  Thanks.

  Link
- Mike Dunlavey February 14, 2014, 6:38 am

  GPROF is not very good for what you need. Try this instead: http://stackoverflow.com/a/378024/23771
  Plenty of people have used it, and it gets results.
  In a large application like yours, 99.9% of the time is spent in a deep call stack terminating in system functions, often doing I/O.
  GPROF is blind to I/O, so you have no idea how much wall-clock time any functions in your system are actually responsible for. The flat profile is mainly about self time, which in a large program is usually irrelevant because the real problems are mid-stack. GPROF does not do a good job of getting inclusive time for functions, and it gives you no information at the level of lines of code, and if there's recursion, it gives up.

  Link
- Alex February 21, 2014, 10:41 am

  Stuck at "Step-2 : Execute the code"

  Executing the code does not produce the gmon.out.

  The code that I have, runs indefinitely, until I do Ctrl+C

Any ideas why no gmon.out?

P.S. it should not change directory.
—
Update1: Well, I ran the proram again, and this time it produced the gmon.out file.
I still stopped it by “Ctrl+C”, so I have no idea why it did not work the other time
—
Update2: It produced the gmon.out because I ran the program with the –help switch, which means clean exit.

So doing a “Ctrl+C” prevents the program from producing the gmon.out

Link

- Cody February 27, 2014, 7:36 pm

  Mike, that’s funny. The only reason I came here is I was trying to remember why gmon.out was not produced (the article didn’t help – I remembered you have to pass it to the compiler _and_ the linker and I then remembered you cannot send it a signal – it has to exit the execution in the normal way)… anyway, I took my own code, profiled it, made some adjustments to my code and dropped the CPU time from 10% to < 1% for this task in question. As usual in life, it is a matter of using the tool in the right way (in addition to using the right tool) and that involves having enough experience in order to interpret it (you know, sort of like you don't use a hammer to put a screw in a screw hole? yeah, that applies here too). Just because _some_ say it is not useful for X does not mean it is _not_ useful for X. The same goes the other way around too. Bottom line: suggesting not to use a profiling tool as it isn't designed for finding bottlenecks (which incidentally, only you wrote that word.. the author was simply explaining gprof usage) or whatever else is unhelpful. The man page, by the way, suggests this:

  "The flat profile shows how much time your program spent in each function, and how many times that function was called. If you simply want to know which functions burn most of the cycles, it is stated concisely here.

  The call graph shows, for each function, which functions called it, which other functions it called, and how many times. There is also an estimate of how much time was spent in the subroutines of each function. This can suggest places where you might try to eliminate function calls that use a lot of time."

  In other words, in the hands of an experienced programmer, it is a very valuable tool for exactly what you claim it isn't good for. (Bugs listed are irrelevant as again, just because something has a problem does not mean it is useless or never useful in any way shape or form. Oh no, the world is NOT perfect and only a fool would think otherwise or think that only perfection is valuable). You can go on all you want about how gprof knows nothing about I/O but that's not a problem (and if you know what problem you have – e.g., cpu usage is higher than it should be – then you can choose the right tool) and only a naive person would think otherwise (why do you think there is 'nice' and 'ionice' utilities ? Exactly, CPU cycles is only ONE variable and nothing in life is so simple). Lastly, on the subject of 'nice' versus 'ionice', perhaps you need to remember the Unix philosophy of do one thing very well (and that is why the pipe can be so powerful – chaining commands together that can really make for incredibly processing power) when thinking about telling others that gprof cannot account for I/O and so is therefore useless.

  Link

- Mike Dunlavey February 28, 2014, 7:33 am

  Hi Cody,

  You raise a lot of valid points, but let me itemize my objections to gprof. I’ll try to be brief.
  It’s not just about gprof itself, because many newer profilers have corrected some of these objections.
  It’s about the mental habits that go along with it, i.e. “myths”:

  1. That program counter sampling is useful (as opposed to stack sampling).

  2. That measuring time of functions is good enough (as opposed to lines of code or even instructions).

  3. That the call graph is important (as opposed to the information in stack samples).

4. That recursion is a tricky confusing issue (it only is a problem when trying to construct an annotated call graph).

5. That accuracy of measurement is important (as opposed to accuracy of identifying speedup opportunities).

6. That invocation counting is useful (as opposed to getting inclusive time percent).

7. That samples need not be taken during IO or other blockage (as opposed to sampling on wall-clock time).

8. That self time matters (as opposed to inclusive time, which includes self time).

9. That samples have to be taken at high frequency (they do not).

10. That you are looking for "the bottleneck" (as opposed to finding all of them – there often are several).

I could go into greater detail on any of these if necessary.

I'm not saying you can't find problems with these tools.
I'm saying, in big software, there are problems they won't find, and if you really need performance, those are killers.

If you need highest performance, in big software, and you can't just kill-it-with-iron, these tools are nowhere near aggressive enough.
The human eye can recognize similarities between state samples (stack and data)
that no summarizing backend of any profiler has any hope of exposing to the user.

For what it's worth, I made a very amateur 8-minute video of how this works:
https://www.youtube.com/watch?v=xPg3sRpdW1U

Cheers

Link

- Mike Dunlavey February 28, 2014, 8:20 am

  Rewording 3rd from last paragraph:

  I'm not saying you can't find some problems with these tools.
  I'm saying, in big software, there are additional problems they won't find, and if you really need performance, those are killers.
  You can think the software is as fast as possible, when in fact it could be much faster.

  Link

- Cody February 28, 2014, 11:07 am

  Hi Mike,

  Well your points are also valid. I think point 10 is exactly what I was getting at: that there are many variables (pardon the pun) and that gprof is only one tool of many that can help but it can still help.

  Responding to your revised third paragraph:

  Indeed, it can always get faster and that is the con and pro of higher level languages; on the one hand, you can get more done sooner but on the other hand the executable will be larger and the executable will not be as efficient (or as fast). And hey, even if [you] were to write everything in assembly (or even machine code), there's always room for improvement* and even without improvement we as humans always strive for faster, better, etc. (hence why CPUs of today are so much faster, better, can handle more at one time, and so on, compared to the [example from when I last did any major assembly] 16 bit days).

*For instance, the following assembly is not as efficient as it could be (unless CPUs and their instructions set or rather the assemblers have improved so much that they optimise it nowadays):
mov ax, 0
versus
xor ax, ax

So yes, you're absolutely right – there is no one size fits all and each tool has their own strengths and weaknesses and the ability to recognise those strengths and weaknesses is what really helps. Shortly, I think we're on the same page more than I thought initially and if I sounded at all arrogant (about life, about the fact there is no such thing as perfection or anything else) then I apologise. The main thing I was getting at is gprof has its uses and to dismiss it entirely is not always helpful (but then so would be dismissing your points – they are valid). But there's always room for improvement and it really is a matter of perspective (your point about software being able to be much faster for example versus my point about assembly versus HLL and even that assembly instructions can be improved upon).

Hopefully that clears up any points from my end (you did indeed clear up your points and I – besides being surprised you responded to me responding from your response from 2012 – appreciate it).

(As a quick addendum: looking at your points again, I think 2 is another excellent one to consider [and something I was getting at too albeit it takes knowing the function and what it calls and where; in fact, that is how I improved my program – I knew the functions that took significant time and called the most, and combined it with the knowledge (from writing the entire [fairly large] program myself) of what called it and what it called, and was able to more or less optimise it out in most cases]. Also, I agree that recursion is not all that difficult. All in all, your points are very valid and whilst you may dislike gprof I've found it useful. On the other hand, I'll take a look at what you suggested too, because as I noted, there is always room for improvement and I strive to better myself and to always learn more).

Thanks for the productive discussion (I didn't expect it but I'm more glad I responded now than I was initially).

Cheers,
Cody

Link

- Cody February 28, 2014, 11:12 am

  Alex,

  The reason ctrl-c prevents it from producing a gmon.out file is not so much that you hit ctrl-c (by itself) but rather what ctrl-c does: it sends an interrupt signal. The problem is gprof won't generate the output unless it calls the exit or returns normally. This means that sending SIGINT or SIGTERM or SIGHUP to your program through the kill utility (and presumably the kill system call or raise library call) will also prevent the generation of the output file. So you need it to exit from program termination (normal termination).

  Link

- Mark Richter February 28, 2014, 5:44 pm

  My only comment here is that what we did get from the few profiling runs that ran to completion helped us identify exactly what the performance problems were.

  I was much more concerned about why the profiling build of the app hung at the end, thereby preventing us from collecting the gmon,out files.

  Yes, I understand the limitations of most program analysis tools. None of them are perfect, but a great many help us mere humans look in the right direction to let our brains figure out what went slow/wrong. It's much easier than just staring at million line-long logs that contain gobs of relatively useless information, typically enough that slogging through it is not worth the effort as compared to narrowing the focus with profiling tools so we have a clue what to look for.

  I take it that no one knows why a -pg program just hangs at the end of execution?

[Link]()

- Cody February 28, 2014, 10:36 pm

  Mark,

  Indeed – that we're imperfect is something that can be turned into a strength, exactly as you described (the utility is not perfect but it still has its uses just like all things in life and even the concept of 'good' comes with 'bad' and 'bad' comes with 'good' – always). Anyway, as for why it would hang, a question and a suggestion on figuring out where its having issues :

  Question is: how does the program end (Does it directly call exit or does it return from main (assuming that it is C or C++) ? Okay, make that two questions (three if you count the previous one as two): does the program do this when _not_ compiled with -pg ? Anything else that is different should be kept in mind too (including – just saying and not suggesting this is it – system load).
  Suggestion: while debugging is truly an art form (which by the way, if you are troubleshooting programs, I highly suggest that if you can, you learn this art as it is incredibly invaluable and that is coming from experience with this ability) even if you can do basic debugging you might be able to figure out at least where the problem lies. Can you compile the program with debugging symbols (compiler option would be '-g', linker does not need it unlike '-pg' [typically the compiler will pass certain options to the linker but apparently -pg is not passed to it – at least according to the man page]) ? If yes, after that, if you find the task's PID (e.g., via ps or if there's only one instance of it, pgrep, assuming Linux [so /proc filesystem.. far as I recall, pgrep uses that but maybe my tired head is mixed up]) and attach to the program _during_ its hang up. You could do that with (examples) GDB (option -p) or it might be easier to use something like strace (since it will show you the function running without having to look at 'bt' [also 'backtrace' – same command in GDB). strace invocation would be like :

  strace -p
  (obviously replace with the pid of the task)
  GDB would be more involved once in the program but GDB you have more control over and that includes line by line execution, break points, watch points and indeed seeing where the execution is at the point you stopped it [backtrace or bt] (and whether strace needs debugging symbols or not I'm not even sure about 100% but I think not: strace ls seems to work and I highly doubt I have debugging symbols for such utility). One final note is that it is almost always not a system library bug when you see something hanging or crashing in a system library (e.g., exit can and does call other functions), despite what many developers would some times wish (because it means there is a problem in their code and so something they have to fix). Just mentioning that because I see that complaint a lot.

  From attaching to the program during execution (where you need to investigate), you then have an idea (well, often) where the problem is in which case you can get closer to solving the problem. I have followups enabled so if you respond maybe I can help more.

  [Link]()

- Sameer March 11, 2014, 1:00 am

  What about cases where i need to know the time spent by each routine in nano seconds precession ?

  Say for high performance applications.

  [Link]()

- Anonymous July 30, 2014, 6:58 am

  excellent…….

  [Link]()

- Andreas August 25, 2014, 2:54 am

  Thanks for this great article!
  But: I don't userstand the line

  67.15 30.77 30.77 2 15.39 23.14 func1

How does it come that this function is called 2 times?

Link

- Vivek Kumar December 30, 2014, 9:48 pm

  Hi

  Is it possible to set the gmon.out file path, when compiling the -pg option?

  Thanks and it is very simple. short and meaningful post.

  Link

- Kiranjp September 3, 2015, 1:26 am

  Superb one keep up the good work man…..

  Link

- Praveen Andhale March 12, 2016, 4:08 am

  Thanks ..very helpful …

  Link

- Ehsan March 16, 2016, 7:56 pm

  Hi, I'm doing everything right but my flat profile is empty? (Do you have any idea why? thank you

  Link

- Anonymous March 21, 2016, 4:07 pm

  From the man page of gcc :

  -pg : Generate extra code to write profile information suitable for the analysis program gprof. You must use this option when compiling the source files you want data about, and you must also use it when linking.
  can you explan , please? what is th man page gcc?

  Link

- FrAnKenStEiN MC May 14, 2016, 5:02 pm

  In my case the CALL GRAPH is not shown

  Link

- Will B June 2, 2016, 1:32 pm

  Thank you for this, Ramesh! 🙂

  Link

Leave a Comment

Name

Email

Website

Comment

Submit

☐ Notify me of followup comments via e-mail

Next post: 11 Linux diff3 Command Examples (Compare 3 Files Line by Line)

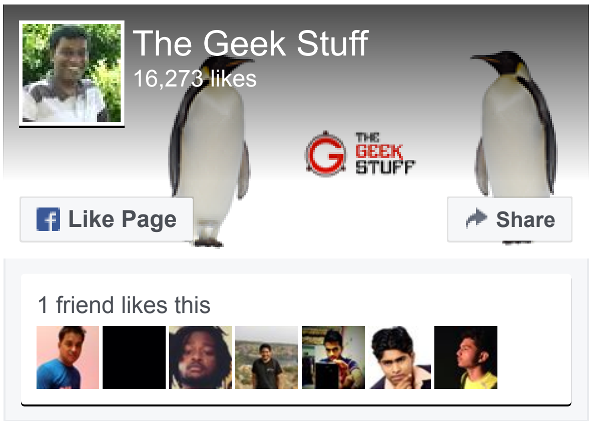
Previous post: Journey of a Data Packet in the Internet

RSS | Email | Twitter | Facebook | Google+

Google™ Custom Search Search

EBOOKS

- Free Linux 101 Hacks 2nd Edition eBook - Practical Examples to Build a Strong Foundation in Linux
- Bash 101 Hacks eBook - Take Control of Your Bash Command Line and Shell Scripting
- Sed and Awk 101 Hacks eBook - Enhance Your UNIX / Linux Life with Sed and Awk
- Vim 101 Hacks eBook - Practical Examples for Becoming Fast and Productive in Vim Editor
- Nagios Core 3 eBook - Monitor Everything, Be Proactive, and Sleep Well



POPULAR POSTS

- 12 Amazing and Essential Linux Books To Enrich Your Brain and Library
- 50 UNIX / Linux Sysadmin Tutorials
- 50 Most Frequently Used UNIX / Linux Commands (With Examples)
- How To Be Productive and Get Things Done Using GTD
- 30 Things To Do When you are Bored and have a Computer
- Linux Directory Structure (File System Structure) Explained with Examples
- Linux Crontab: 15 Awesome Cron Job Examples
- Get a Grip on the Grep! – 15 Practical Grep Command Examples
- Unix LS Command: 15 Practical Examples
- 15 Examples To Master Linux Command Line History
- Top 10 Open Source Bug Tracking System
- Vi and Vim Macro Tutorial: How To Record and Play
- Mommy, I found it! -- 15 Practical Linux Find Command Examples
- 15 Awesome Gmail Tips and Tricks
- 15 Awesome Google Search Tips and Tricks
- RAID 0, RAID 1, RAID 5, RAID 10 Explained with Diagrams
- Can You Top This? 15 Practical Linux Top Command Examples
- Top 5 Best System Monitoring Tools
- Top 5 Best Linux OS Distributions
- How To Monitor Remote Linux Host using Nagios 3.0
- Awk Introduction Tutorial – 7 Awk Print Examples
- How to Backup Linux? 15 rsync Command Examples
- The Ultimate Wget Download Guide With 15 Awesome Examples
- Top 5 Best Linux Text Editors
- Packet Analyzer: 15 TCPDUMP Command Examples
- The Ultimate Bash Array Tutorial with 15 Examples
- 3 Steps to Perform SSH Login Without Password Using ssh-keygen & ssh-copy-id
- Unix Sed Tutorial: Advanced Sed Substitution Examples
- UNIX / Linux: 10 Netstat Command Examples
- The Ultimate Guide for Creating Strong Passwords
- 6 Steps to Secure Your Home Wireless Network
- Turbocharge PuTTY with 12 Powerful Add-Ons

CATEGORIES

- Linux Tutorials
- Vim Editor
- Sed Scripting
- Awk Scripting
- Bash Shell Scripting
- Nagios Monitoring
- OpenSSH

- [IPTables Firewall]
- Apache Web Server
- MySQL Database
- Perl Programming
- Google Tutorials
- Ubuntu Tutorials
- PostgreSQL DB
- Hello World Examples
- C Programming
- C++ Programming
- DELL Server Tutorials
- Oracle Database
- VMware Tutorials

Ramesh Natarajan

G+ Follow

About The Geek Stuff

My name is **Ramesh Natarajan**. I will be posting instruction guides, how-to, troubleshooting tips and tricks on Linux, database, hardware, security and web. My focus is to write articles that will either teach you or help you resolve a problem. Read more about Ramesh Natarajan and the blog.

Contact Us

Email Me : Use this Contact Form to get in touch me with your comments, questions or suggestions about this site. You can also simply drop me a line to say hello!.

Follow us on Google+

Follow us on Twitter

Become a fan on Facebook

Support Us

Support this blog by purchasing one of my ebooks.

Bash 101 Hacks eBook

Sed and Awk 101 Hacks eBook

Vim 101 Hacks eBook

Nagios Core 3 eBook