It’s the time to speed-up your code with the

gprof

Abstract:

‘Perfect in performance’ will be a challengeable task, even may be a nightmare to the application developer! As per the studies, most of the developers put their much effort to find out where their code lags…

Profiling allows you to understand where your program spent its time and which functions called which other functions while it was executing. It can also tell you which functions are being called more or less often than you expected. This information can show you which pieces of your program are slower than you expected, and might be developers for rewriting to make your program execute faster. This may help you spot bugs that had otherwise been unnoticed also.

Meet “**gprof**”, a free profiling tool provided by GNU Foundation and experience how you can change your nightmare to "noon-hallucination”!

Introduction:

‘Embedded system’ always demands performance first than anything else. No matter how complex or simple the application may, but performance will give an up-stand to that in the market. This may be the primary reason, why every developer spending lot of time to make codes perfect than write a code!

But, the process of making a code perfect is not easy as you might think when it comes to manual correction. Most efforts become equivalent to counting the stars on the sky, when the reason behind the execution lagging is unknown.

It is therefore very clear that a tool, which can go inside the code and find out the real factors behind lagging, can help in vast. “Code-profiling” does this and gprof is a good tool to start with…

How to start with profiling?

gprof profiling can be enable by add *-pg* to the gcc compile flags.

Example as follows:

gcc test.c -pg -o test -O2 –lc

Once the program is compiled for profiling, you must run it in order to generate the information that gprof needs. Once you have built the application, simply run it as normal, using the normal arguments, file names, etc. as follows:

. /test 50000

The program should run normally, producing the same output as usual. It will, however, run somewhat slower than normal because of the time spent collecting and the writing the profile data.

Once this completes, you should see a file called *gmon.out* created in the current directory.
In order to write the 'gmon.out' file properly, your program must exit normally, by returning from *main* or by calling *exit*. Calling the low-level function *_exit* does not write the profile data, and neither does abnormal termination due to an unhandled signal.

Interpreting gprof's Output:

gprof can produce several different output styles, the most important of which are described below.

1. The Flat Profile:

Flat profile can be obtained with the gprof command, passing the executable itself and the gmon.out file as follows:

gprof test gmon.out -p

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.

Output:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 33.34      0.02     0.02     7208     0.00     0.00  open
 16.67      0.03     0.01      244     0.04     0.12  offtime
 16.67      0.04     0.01        8     1.25     1.25  memccpy
 16.67      0.05     0.01        7     1.43     1.43  write
 16.67      0.06     0.01                             mcount
  0.00      0.06     0.00      236     0.00     0.00  tzset
  0.00      0.06     0.00      192     0.00     0.00  tolower
  0.00      0.06     0.00       47     0.00     0.00  strlen
  0.00      0.06     0.00       45     0.00     0.00  strchr
  0.00      0.06     0.00        1     0.00    50.00  main
  0.00      0.06     0.00        1     0.00     0.00  memcpy
  0.00      0.06     0.00        1     0.00    10.11  print
  0.00      0.06     0.00        1     0.00     0.00  profil
  0.00      0.06     0.00        1     0.00    50.00  report
```

2. The Call Graph:

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.

"Call graph" can be obtain as follows:

gprof test gmon.out -q

Output:

```
granularity: each sample hit covers 2 byte(s) for 20.00% of 0.05 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]     100.0    0.00    0.05                 start [1]
                 0.00    0.05       1/1           main [2]
                 0.00    0.00       1/2           on_exit [28]
                 0.00    0.00       1/1           exit [59]
-----------------------------------------------
                 0.00    0.05       1/1           start [1]
[2]     100.0    0.00    0.05       1         main [2]
                 0.00    0.05       1/1           report [3]
-----------------------------------------------
                 0.00    0.05       1/1           main [2]
[3]     100.0    0.00    0.05       1         report [3]
                 0.00    0.03       8/8           timelocal [6]
                 0.00    0.01       1/1           print [9]
                 0.00    0.01       9/9           fgets [12]
                 0.00    0.00      12/34          strncmp <cycle 1> [40]
                 0.00    0.00       8/8           lookup [20]
                 0.00    0.00       1/1           fopen [21]
                 0.00    0.00       8/8           chewtime [24]
                 0.00    0.00       8/16          skipspace [44]
-----------------------------------------------
[4]      59.8    0.01        0.02       8+472     <cycle 2 as a whole> [4]
                 0.01        0.02     244+260         offtime <cycle 2> [7]
                 0.00        0.00     236+1           tzset <cycle 2> [26]
-----------------------------------------------
```

Conclusion:

'gprof' is free yet very powerful profiling tool for optimizing your code. Profiler uses information collected during the actual execution of your program. So, it can be used on programs that are too large or too complex to analyze by reading the source. However, how your program is run will affect the information that shows up in the profile data. If you don't use some feature of your program while it is being profiled, no profile information will be generated for that feature.

One of the main advantages of gprof over the other profiling tools is that gprof will be common in Linux machine and it preinstalled with 'gcc' in most of the system.
What are you waiting for then? Get the advantage of the free GNU profiler from today. Life makes easy…

Reference:

GNU gprof manual:
http://sourceware.org/binutils/docs/gprof/

By

Ajith P Venugopal
Member Technical- Software
ajithpv@iwavesystems.com