Profiling programs based on user defined execution time - GPROF

SHUBHENDRA PAL SINGHAL*,
Department of Computer Science and Engineering
National Institute of Technology, Tiruchirappalli

This paper focuses on the explanation of the architecture of profilers particularly gprof and how to profile a program according to the user defined input of execution time . Gprof is a profiler available open source in the package of binutils. Gprof records the flow of the program including the callee and caller information and their respective execution time. This information is represented in the form of a call graph which is explained in more detail in section 4. Profilers at the time of execution creates a call graph file which indicates the full flow of the program including the individual execution time as well. This paper aims at providing a better understanding of the data structure used to store the information and how is a profiler(gprof) actually using this data structure to give user a readable format. The next section of this paper solves one of the limitation of gprof i.e. edit the time of block of code without understanding the call graph. Any changes in the execution time of a particular block of code would affect the total execution time. So if we edit the gprof in such a way that its consistent and platform independent, then it can yield various results like testing execution time after parallelism, before even designing it by replacing the values with theoretical/emulated ones and see if the total execution time is getting reduced by a desired number or not? Gprof edit can help us figure out that what section of code can be parallelized or which part of code is taking the most time and which call or part can be changed to reduce the execution time[6].

Additional Key Words and Phrases: Profiling, Gprof, binutils, Execution time, Histograms

1 INTRODUCTION

Profiling is a dynamic program analysis[5] that measures the time complexity of every instruction in a program. The information is stored by using -pg flag in gcc[3]. This flag when enabled switches debugging and profiling on during the compilation of the program. This produces an executable file. This executable file when run produces a file "gmon.out" which contains the call graph information of the program. This is a binary file and is not readable by a user. So, gprof uses the symbol table and histograms and records every instance in a histogram. Histogram contains the address and the execution time of that instance. The symbol table consists of the function name and its corresponding assigned index. This information is traced by the profiler to print the call graph of the program in a readable format[3]. The profiler uses the approach of traversing the path from root to the last child (in that arc) in the graph which helps to figure out that what calls have been made to whom and what was the time taken inside a parent and the child separately. This information is stored in the form of arcs making parent as the start pointer. The information is then printed in such a way that there are two ways to discern the data flow[7] :
1.) Flat Profile : The flat profile shows the total amount of time your program spent executing each function. The functions are sorted by first by decreasing run-time spent in them, then by decreasing number of calls, then alphabetically by name. Generally, sample count is 0.01 sec indicating that the number 'x' units in the profile represents (100 * x) seconds of the execution.[7]
2.) Call Graph Profile : The call graph shows how much time was spent in each function and its children. In each entry, the primary line is the one that starts with an index number in square brackets. The end of this line says which function the entry is for. The preceding lines in the entry describe the callers of this function and the following lines describe its subroutines(children).[7]

Author's address: Shubhendra Pal Singhal, 106116088@nitt.edu,
Department of Computer Science and Engineering
National Institute of Technology, Tiruchirappalli
.

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
74.43	6.88	6.88	3	2.29	2.29	func5(){8} 100%
20.52	8.78	1.90	1	1.90	7.04	func1(){12} 100%
3.71	9.13	0.34	2	0.17	2.47	func4(){9} 100%
2.18	9.33	0.20	1	0.20	2.67	func3(){10} 100%
0.11	9.34	0.01	1	0.01	5.15	func2(){11} 100%

Fig. 1. Flat Profile

2 INSTALLATION OF GPROF IN LINUX ENVIRONMENT

sudo apt-get install binutils is the command which will install the package binutils. Gprof is a sub-directory inside this directory[4].

3 TERMS USED

1.) Call Graph : It is a graph which describes the calling pattern of the functions and analyses the execution time of every call even if it involves the same parent and child.[7]
2.) Self time : The total execution time of the function excluding the time taken by its subroutines(children) is called self time.[7]
3.) Child time : The time taken by the children of a function i.e. the difference between the time instant when it returns back to caller and the time instant the callee is called by the caller.[7]
4.) Arc : Arc denoted the path from the root to leaf indicating the calls made by the root and then subsequent callers(children of root) till the point the last child returns back to its caller[7].
5.) Histogram : It is a data structure which consists of a low base address and high end address which represents the storage of the callee and its corresponding caller. The bin_count represents the total execution time taken by an instance. It can be used to describe the self time and child time. If the function represents the child of a function, then the corresponding bin_count will represent the self time of child. For example if func1() calls func2(), then the arc func1()—-> func2() will represent the time taken by func2() alone. The func1() time will be calculated by the arc main()—>func1(). So like this there will be one root function which will not have any parent. This function is named as spontaneous and the time recorded for this function is stored in the arc representing spontaneous as the tag. The hist time represents the self time taken by the function and the "child_time" represents the time taken by its children.
6.) Propagate time : The self time taken by the function in an arc is called prop.self time. The prop.child time represents the time taken by the children of that function keeping in mind that if already traversed arcs includes the child, then it needs to be included.
7.) Bin-count : In this paper, the bin_count is referred to as the time corresponding to the bin of histogram. The code has a totally different interpretation of bin_count which is different from what is used in this paper. It is not the count of the bin in histogram but its the time which is obtained after scaling using the sample count as a metric.
Every profile shows the average time per call[8].

3.1 Illustration of how call graph is formed

Fig3. represents the call graph of the program. The calculation of time is always a bottom top approach as the time can be calculated when the call returns from its child. So the call-graph profile is topologically sorted and

```
granularity: each sample hit covers 2 byte(s) for 0.11% of 9.34 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.00    9.34                 main [1]
                1.90    5.15       1/1           func1() [2]
                2.29    0.00       1/3           func5() [3]
-----------------------------------------------
                1.90    5.15       1/1           main [1]
[2]     75.4    1.90    5.15       1         func1() [2]
                0.01    5.14       1/1           func2() [4]
-----------------------------------------------
                2.29    0.00       1/3           main [1]
                4.59    0.00       2/3           func4() [5]
[3]     73.7    6.88    0.00       3         func5() [3]
-----------------------------------------------
                0.01    5.14       1/1           func1() [2]
[4]     55.1    0.01    5.14       1         func2() [4]
                0.20    2.47       1/1           func3() [6]
                0.17    2.29       1/2           func4() [5]
-----------------------------------------------
                0.17    2.29       1/2           func3() [6]
                0.17    2.29       1/2           func2() [4]
[5]     52.8    0.34    4.59       2         func4() [5]
                4.59    0.00       2/3           func5() [3]
-----------------------------------------------
                0.20    2.47       1/1           func2() [4]
[6]     28.6    0.20    2.47       1         func3() [6]
                0.17    2.29       1/2           func4() [5]
-----------------------------------------------
```

Fig. 2. Call Graph Profile

if there any recursive calls or cycles then that cycle is considered as one strongly connected component and the further topological sorting is done considering it as one node. The call graph profile thus starts with the arc representing func3(), func4() and func5(). Calculation of this arc means that first calculation of func3() can be done as both func4() and func5() are getting called in the subsequent execution. Now, the func4() is called once and func5() is called once. So, when we proceed to index 5, we need to include two calls of func4()–> func5() as there are 2 calls of similar kind. The average of time taken by two calls is taken. Similarly, the full call graph is constructed as shown in Fig 2.

The Fig 1. shows the percent of time spent and their respective total time. Cumulative seconds helps to determine the total time taken i.e. total execution time of the program.

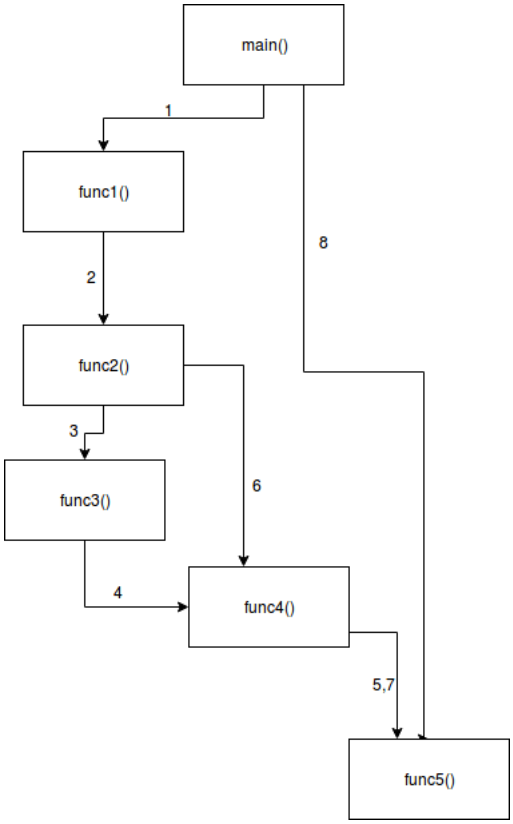

Fig. 3. Illustration of Call Graph

4 IMPLEMENTATION

Gprof uses the gmon.out to extract the execution time in histogram specified in hist.c. Call graph is constructed from gmon.out by "call_graph.c" These bins then are collectively merged to get the total and self time of a function in hist.c. This is used to create flat profile. Using this "cg_arcs.h", gprof constructs the arc from the parent to the leaf of the graph. This arc is then printed in an order specified by call graph profile and the execution time is taken from flat profile. Supposedly, a function took 30sec averagely with 3 calls, the cal graph will show 10sec/call[1] in the call graph profile, thus extracting this average time from flat profile, shown in file cg_print.c". Apart from these files, rest are all utility or library based files that support these files - hist.c, cg_arcs.h, cg_print.c[7].

4.1 Changing the printing pattern of gprof

In fig3., suppose we want to change the self time taken by func5(). An "if" statement would suffice it stating that if the index is equal to 5, then change the time. At this point of time, there is no problem, but if we move a step up and analyze func3() time, then we cannot trace that func3() called func5() effectively i.e. though func3() called func4(), but the execution time of func3() will change as the time taken by func4() is also changed. So, this particular change can be propagated to its immediate next parent i.e. func4(), but not to the whole graph as we don't have any means to store every function calling children. Visiting every child when a parent is encountered

will take a huge amount of time and the time also will not be stored anywhere meaning that if we want to retrieve any arc time that is getting affected by this change, then we cannot get it as we are just printing and not changing the values of assigned data structure i.e. histogram [5].

4.2 Editing data structure of gprof

Suppose, the part of program can be replaced by some coding paradigm[5] which helps to reduce the execution time of that particular block of code, then we have to replace that time and see that how execution pattern gets affected. This requires a change in the record of histogram corresponding to the block. Identifying the block can be done by 2 ways :
1.) Address - The address of the block can be found via printing and the corresponding block can then be changed. But there is one flaw here and that is if we run the program next time, we are not sure whether the program will load in the exact same address or not. This is what we call as platform dependency where every instance is changed.
2.) Index - The above idea seems to be a bit infeasible. So in this situation we analyze that what is not platform dependent. Address, time etc. are all machine dependent, but the call graph is not. So we assign the index to every call and then change the time for those calls(corresponding to the block we need to change). Index is assigned to every call and the respective id can be printed so that the user can identify the call by noticing its parent and respective child. The time corresponding to this id can be changed in the code. This change will be propagated and the final execution time will also get changed by the same amount.
The histogram consists of bin_count which is used to indicate the time taken in execution.
The algorithm described below is edited in the file "hist.c" which contains the histogram data structure [5] that is built from "gmon.out" file. The function "static void hist_assign_samples_1 (histogram *r)" consists of assigning the address and time to a histogram, so if we change the time here corresponding to the id that we want to change, the the entire pattern can be generated. The id that needs to be changed is known by printing it for every call initially and then figuring out which id corresponds to which call. A table of id and corresponding call is printed which helps to identify which id needs to be changed by looking at which call has to be modified. Ids are not machine dependent which makes it a feasible way of changing time of a specific block.

Assume min to be lower limit and max as upper limit of block that needs to be replaced. c be the changed value. In case of different values for every id, it needs to be specified in the form of an array of values.
$id = 0$, $total\_bin = 0$
if $id >= min - 1 and id <= max - 1$ **then**
    $total\_bin \mathrel{+}= bin\_count$
    $bin\_count = c$
end
$count = count + 1$
$count\_time = bin\_count$

Algorithm 1: Algorithm for changing execution time of a certain part of code

5 COMPILATION AND RESULTS OBTAINED

There are two types of gprofs that are required :
1.) Gprof version1 : hist.c is edited to print the ids and the corresponding calls. 2.) Gprof version2 : After identifying the call that needs to be changed, the algorithm is included in hist.c
After editing hist.c, gprof has to be compiled [3] and the changes needs to be incorporated, so we use :

```
Id is : 2

--->_Z5func4v--->
[assign_samples] bin_low_pc=0x4006a8, bin_high_pc=0x4006ac, bin_count=500

Z5func5v


Id is : 3

--->_Z5func4v--->
[assign_samples] bin_low_pc=0x4006ac, bin_high_pc=0x4006b0, bin_count=166

Z5func5v
```

Fig. 4. Bin id print for version1

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 72.72      1.90     1.90        1     1.90     2.57  func1(){12}  100%
 13.15      2.24     0.34        2     0.17     0.23  func4(){9}  100%
  7.74      2.44     0.20        1     0.20     0.43  func3(){10}  100%
  6.96      2.62     0.18        3     0.06     0.06  func5(){8}  100%
  0.39      2.63     0.01        1     0.01     0.68  func2(){11}  100%
Tot is: 2.634857
Total time in bin is : 666
```

Fig. 5. Flat Profile version2

1.) **make** (rectify the errors if any)
2.) **make install**
This will create version1 and version2 of the gprof. Gprof is used on the executable file and "gmon.out" as :
<directory-name>/gprof -d <.exe file> gmon.out > <file-name>
e.g. The program shown in Fig3.[2] is executed and the exe file is test. The gprof directory is /home/. So, the command is : /home/gprof test gmon.out > a.txt
This will save the output in a.txt file. So we need to create two such text files corresponding to the two versions of gprof.

6 ANALYSIS OF CHANGES OBSERVED IN THE SYSTEM

Suppose, we want to change the time of arc func4()—> func5() to 0.01 seconds, then this particular bin id checked is 2 and 3 as shown in fig4. The func5() time needs to be changed indicating that func4() should be the parent and func5() must be the child in the representation and only bin ids 2 and 3 correspond to this arc. Suppose we change both of them to 1 which implies that the total time taken by func5() must reduce from 6.88 to 0.16(start−>func5()) + 0.01*2(1 sec*sample-size = 1*0.01 = 0.01, and there are two such calls) which equals 0.18. Difference 6.88 and 0.18 = 6.7 i.e. the total execution time should reduce by 6.7, i.e. from 9.34 to 9.34-6.7 = 2.64 which is close to the actual result obtained in fig 5. Also when we analyze the call graph profile, then we notice that func5() takes just 0.06 seconds per call which is correct because 0.18/3 = 0.06 in fig5 and fig6.

```
granularity: each sample hit covers 2 byte(s) for 0.38% of 2.63 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.00    2.63                 main{13}  100%  [1]
[cmp_arc] main{13}  100%  [1] calls func5(){8}  100%  [6] 6.057143 + 0.000000 1/3
[cmp_arc] main{13}  100%  [1] calls func1(){12}  100%  [2] 189.790476 + 67.638095 1/1

                1.90    0.68       1/1           func1(){12}  100%  [2]
                0.06    0.00       1/3           func5(){8}  100%  [6]
-----------------------------------------------
                1.90    0.68       1/1           main{13}  100%  [1]
[2]     97.7    1.90    0.68       1         func1(){12}  100%  [2]
                0.01    0.67       1/1           func2(){11}  100%  [3]
-----------------------------------------------
                0.01    0.67       1/1           func1(){12}  100%  [2]
[3]     25.7    0.01    0.67       1         func2(){11}  100%  [3]
[cmp_arc] func2(){11}  100%  [3] calls func4(){9}  100%  [4] 17.161905 + 6.057143 1/2
[cmp_arc] func2(){11}  100%  [3] calls func3(){10}  100%  [5] 20.190476 + 23.219048 1/1

                0.20    0.23       1/1           func3(){10}  100%  [5]
                0.17    0.06       1/2           func4(){9}  100%  [4]
-----------------------------------------------
[cmp_arc] func3(){10}  100%  [5] calls func4(){9}  100%  [4] 17.161905 + 6.057143 1/2
[cmp_arc] func2(){11}  100%  [3] calls func4(){9}  100%  [4] 17.161905 + 6.057143 1/2

                0.17    0.06       1/2           func3(){10}  100%  [5]
                0.17    0.06       1/2           func2(){11}  100%  [3]
[4]     17.6    0.34    0.12       2         func4(){9}  100%  [4]
                0.12    0.00       2/3           func5(){8}  100%  [6]
-----------------------------------------------
                0.20    0.23       1/1           func2(){11}  100%  [3]
[5]     16.5    0.20    0.23       1         func3(){10}  100%  [5]
                0.17    0.06       1/2           func4(){9}  100%  [4]
-----------------------------------------------
[cmp_arc] func4(){9}  100%  [4] calls func5(){8}  100%  [6] 12.114286 + 0.000000 2/3
[cmp_arc] main{13}  100%  [1] calls func5(){8}  100%  [6] 6.057143 + 0.000000 1/3

                0.06    0.00       1/3           main{13}  100%  [1]
                0.12    0.00       2/3           func4(){9}  100%  [4]
[6]      6.9    0.18    0.00       3         func5(){8}  100%  [6]
```

Fig. 6. Call Graph Profile version2

7 SOLUTION TO LIMITATION OF GPROF

One of the limitation of gprof[8] that the arc time in call graph will always be average[1]. Even if one of the arc takes very less amount of time in one case and the same arc takes more time in another case, still the time displayed will be average of both and not the actual one. This involves the need to understand the call graph before editing the block of code according to user defined input. But big programs like SAT has a lot of functions and thus, it makes a bit difficult to trace all functions and then edit the time for that block of code. So, the solution to this problem is that we just have to see the id of the corresponding and change it with the user defined time as shown in fig 7. Thus, without understanding the call graph, we can analyze the pattern of how is the total execution time getting affected.

8 CONCLUSION

Changing the time of a particular block of code could be done in several ways but considering the factor that code should be portable[6], reference to a call by its id turned out to be a suitable method. Gprof printing pattern cannot be changed as the calls cannot be traced back as seen in the section 4.1. Adding an extra attribute to a data structure is a small change that can profile a program according to the user needs. This strategy can be used to test that how much does emulated time of a block of code affects the total execution time. If its an expected number, then the hardware design[6] for the same can be implemented on the chip instead to first designing it and then finding it to be futile. This can improve the hardware design process and also can help see whether any kind of parallelism is helping in execution.

REFERENCES

[1] Jay Fenlason. Nov 2008. GNU Gprof. https://sourceware.org/binutils/docs/gprof/Assumptions.html#Assumptions

[2] David Flater. March 2013. Configuration of profiling tools for C/C++ applications under 64-bit Linux. https://nvlpubs.nist.gov/nistpubs/TechnicalNotes/NIST.TN.1790.pdf

[3] R.Stallman J. Fenlasonand. 2000. GNU gprof:the GNU profiler. https://www.cs.tufts.edu/comp/150PAT/tools/gprof/gprof.pdf

[4] Himanshu Arora Uqnic Network Pte Ltd. Aug 15, 2014. How to Profile a C program in Linux using GNU gprof. https://www.maketecheasier.com/profile-c-program-linux-using-gprof/

[5] Dmitrijs Zaparanuks Matthias Hauswirth. June 11 - 16, 2012. Algorithmic profiling. (June 11 - 16, 2012), Pages 67–76. https://doi.org/10.1145/2254064.2254074

[6] David B. Stewart. 2006. Measuring Execution Time and Real-Time Performance. https://pdfs.semanticscholar.org/e255/041e179f96a46f772c7959e381710a6a5a94.pdf

[7] Marshall K. McKusick Susan L. Graham, Peter B. Kessler. 1982. gprof: a Call Graph Execution Profiler. (1982). https://doi.org/10.1145/800230.806987

[8] Dominic A. Varley. APRIL 1993. PRACTICAL EXPERIENCE OF THE LIMITATIONS OF GPROF. (APRIL 1993). https://pdfs.semanticscholar.org/5f0f/69b65304898a38e7528dfab6cefc140e4466.pdf

```
int count = 0;
int count_arg = 0;
int tot_bin = 0;
static void
hist_assign_samples_1 (histogram *r)
{
  bfd_vma bin_low_pc, bin_high_pc;
  bfd_vma sym_low_pc, sym_high_pc;
  bfd_vma overlap, addr;
  unsigned int bin_count;
  unsigned int i, j, k;
  double count_time, credit;

  bfd_vma lowpc = r->lowpc / sizeof (UNIT);

  /* Iterate over all sample bins.  */
  for (i = 0, k = 1; i < r->num_bins; ++i)
    {
      bin_count = r->sample[i];
      if (! bin_count)
        continue;
      count_arg++;
      DBG(SAMPLEDEBUG,printf ("----------------------------  "));
      DBG(SAMPLEDEBUG,printf ("\n\nId is : %d\n",count_arg));
      bin_low_pc = lowpc + (bfd_vma) (hist_scale * i);
      bin_high_pc = lowpc + (bfd_vma) (hist_scale * (i + 1));
      if(count>=132 && count <=250) {
        tot_bin+=bin_count;
        DBG(SAMPLEDEBUG,printf ("Inside bin_count change"));
        bin_count = 1;
      }
      count++;
      count_time = bin_count;
      DBG (SAMPLEDEBUG,
     printf ("\n--->%s--->\n",symtab.base[k].name));

      DBG (SAMPLEDEBUG,
           printf (
      "[assign_samples] bin_low_pc=0x%lx, bin_high_pc=0x%lx, bin_count=%u\n",
                  (unsigned long) (sizeof (UNIT) * bin_low_pc),
                  (unsigned long) (sizeof (UNIT) * bin_high_pc),
                  bin_count));
      total_time += count_time;
```

Fig. 7. Editing hist.c for user defined execution time

Count corresponds to id of the bin. The id corresponding to the change is written in if condition e.g. here it is 133-251. See the total bin count which is being replaced and getting stored in tot_bin. User defined input is bin_count = 1.