GNU gprof

The GNU Profiler

Jay Fenlason and Richard Stallman

This manual describes the GNU profiler, `gprof`, and how you can use it to determine which parts of a program are taking most of the execution time. We assume that you know how to write, compile, and execute programs. GNU `gprof` was written by Jay Fenlason.

This manual was edited January 1993 by Jeffrey Osier.

1 Why Profile

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

Since the profiler uses information collected during the actual execution of your program, 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.

Profiling has several steps:

- You must compile and link your program with profiling enabled. See Chapter 2 [Compiling], page 3.
- You must execute your program to generate a profile data file. See Chapter 3 [Executing], page 5.
- You must run `gprof` to analyze the profile data. See Chapter 4 [Invoking], page 7.

The next three chapters explain these steps in greater detail.

The result of the analysis is a file containing two tables, the *flat profile* and the *call graph* (plus blurbs which briefly explain the contents of these tables).

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. See Chapter 5 [Flat Profile], page 11.

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. See Chapter 6 [Call Graph], page 13.

2 Compiling a Program for Profiling

The first step in generating profile information for your program is to compile and link it with profiling enabled.

To compile a source file for profiling, specify the '-pg' option when you run the compiler. (This is in addition to the options you normally use.)

To link the program for profiling, if you use a compiler such as cc to do the linking, simply specify '-pg' in addition to your usual options. The same option, '-pg', alters either compilation or linking to do what is necessary for profiling. Here are examples:

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

The '-pg' option also works with a command that both compiles and links:

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

If you run the linker ld directly instead of through a compiler such as cc, you must specify the profiling startup file '/lib/gcrt0.o' as the first input file instead of the usual startup file '/lib/crt0.o'. In addition, you would probably want to specify the profiling C library, '/usr/lib/libc_p.a', by writing '-lc_p' instead of the usual '-lc'. This is not absolutely necessary, but doing this gives you number-of-calls information for standard library functions such as read and open. For example:

```
ld -o myprog /lib/gcrt0.o myprog.o utils.o -lc_p
```

If you compile only some of the modules of the program with '-pg', you can still profile the program, but you won't get complete information about the modules that were compiled without '-pg'. The only information you get for the functions in those modules is the total time spent in them; there is no record of how many times they were called, or from where. This will not affect the flat profile (except that the calls field for the functions will be blank), but will greatly reduce the usefulness of the call graph.

3 Executing the Program to Generate Profile Data

Once the program is compiled for profiling, you must run it in order to generate the information that `gprof` needs. Simply run the program as usual, using the normal arguments, file names, etc. 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.

The way you run the program—the arguments and input that you give it—may have a dramatic effect on what the profile information shows. The profile data will describe the parts of the program that were activated for the particular input you use. For example, if the first command you give to your program is to quit, the profile data will show the time used in initialization and in cleanup, but not much else.

You program will write the profile data into a file called '`gmon.out`' just before exiting. If there is already a file called '`gmon.out`', its contents are overwritten. There is currently no way to tell the program to write the profile data under a different name, but you can rename the file afterward if you are concerned that it may be overwritten.

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.

The '`gmon.out`' file is written in the program's *current working directory* at the time it exits. This means that if your program calls `chdir`, the '`gmon.out`' file will be left in the last directory your program `chdir`'d to. If you don't have permission to write in this directory, the file is not written. You may get a confusing error message if this happens. (We have not yet replaced the part of Unix responsible for this; when we do, we will make the error message comprehensible.)

4 gprof Command Summary

After you have a profile data file 'gmon.out', you can run gprof to interpret the information in it. The gprof program prints a flat profile and a call graph on standard output. Typically you would redirect the output of gprof into a file with '>'.

You run gprof like this:

```
gprof options [executable-file [profile-data-files...]] [> outfile]
```

Here square-brackets indicate optional arguments.

If you omit the executable file name, the file 'a.out' is used. If you give no profile data file name, the file 'gmon.out' is used. If any file is not in the proper format, or if the profile data file does not appear to belong to the executable file, an error message is printed.

You can give more than one profile data file by entering all their names after the executable file name; then the statistics in all the data files are summed together.

The following options may be used to selectively include or exclude functions in the output:

`-a` The '-a' option causes gprof to suppress the printing of statically declared (private) functions. (These are functions whose names are not listed as global, and which are not visible outside the file/function/block where they were defined.) Time spent in these functions, calls to/from them, etc, will all be attributed to the function that was loaded directly before it in the executable file. This option affects both the flat profile and the call graph.

`-e` *function˙name*
: The '-e *function*' option tells gprof to not print information about the function *function_name* (and its children...) in the call graph. The function will still be listed as a child of any functions that call it, but its index number will be shown as '[not printed]'. More than one '-e' option may be given; only one *function_name* may be indicated with each '-e' option.

`-E` *function˙name*
: The -E *function* option works like the -e option, but time spent in the function (and children who were not called from anywhere else), will not be used to compute the percentages-of-time for the call graph. More than one '-E' option may be given; only one *function_name* may be indicated with each '-E' option.

`-f` *function˙name*
: The '-f *function*' option causes gprof to limit the call graph to the function *function_name* and its children (and their

children...). More than one '-f' option may be given; only one *function_name* may be indicated with each '-f' option.

`-F` *function`name*
: The '-F *function*' option works like the `-f` option, but only time spent in the function and its children (and their children...) will be used to determine total-time and percentages-of-time for the call graph. More than one '-F' option may be given; only one *function_name* may be indicated with each '-F' option. The '-F' option overrides the '-E' option.

`-k` *from... to...*
: The '-k' option allows you to delete from the profile any arcs from routine *from* to routine *to*.

`-v`
: The '-v' flag causes `gprof` to print the current version number, and then exit.

`-z`
: If you give the '-z' option, `gprof` will mention all functions in the flat profile, even those that were never called, and that had no time spent in them. This is useful in conjunction with the '-c' option for discovering which routines were never called.

The order of these options does not matter.

Note that only one function can be specified with each `-e`, `-E`, `-f` or `-F` option. To specify more than one function, use multiple options. For example, this command:

```
gprof -e boring -f foo -f bar myprogram > gprof.output
```

lists in the call graph all functions that were reached from either `foo` or `bar` and were not reachable from `boring`.

There are a few other useful `gprof` options:

`-b`
: If the '-b' option is given, `gprof` doesn't print the verbose blurbs that try to explain the meaning of all of the fields in the tables. This is useful if you intend to print out the output, or are tired of seeing the blurbs.

`-c`
: The '-c' option causes the static call-graph of the program to be discovered by a heuristic which examines the text space of the object file. Static-only parents or children are indicated with call counts of '0'.

`-d` *num*
: The '-d *num*' option specifies debugging options.

`-s`
: The '-s' option causes `gprof` to summarize the information in the profile data files it read in, and write out a profile data file called '`gmon.sum`', which contains all the information from the profile data files that `gprof` read in. The file '`gmon.sum`' may

be one of the specified input files; the effect of this is to merge the data in the other input files into 'gmon.sum'. See Chapter 8 [Sampling Error], page 23.

Eventually you can run gprof again without '-s' to analyze the cumulative data in the file 'gmon.sum'.

-T
: The '-T' option causes gprof to print its output in "traditional" BSD style.

5 How to Understand the Flat Profile

The *flat profile* shows the total amount of time your program spent executing each function. Unless the '-z' option is given, functions with no apparent time spent in them, and no apparent calls to them, are not mentioned. Note that if a function was not compiled for profiling, and didn't run long enough to show up on the program counter histogram, it will be indistinguishable from a function that was never called.

This is part of a flat profile for a small program:

```
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
...
```

The functions are sorted by decreasing run-time spent in them. The functions '`mcount`' and '`profil`' are part of the profiling aparatus and appear in every flat profile; their time gives a measure of the amount of overhead due to profiling.

The sampling period estimates the margin of error in each of the time figures. A time figure that is not much larger than this is not reliable. In this example, the '`self seconds`' field for '`mcount`' might well be '`0`' or '`0.04`' in another run. See Chapter 8 [Sampling Error], page 23, for a complete discussion.

Here is what the fields in each line mean:

`% time`
: This is the percentage of the total execution time your program spent in this function. These should all add up to 100%.

`cumulative seconds`
: This is the cumulative total number of seconds the computer spent executing this functions, plus the time spent in all the functions above this one in this table.

`self seconds`
: This is the number of seconds accounted for by this function alone. The flat profile listing is sorted first by this number.

`calls`
: This is the total number of times the function was called. If the function was never called, or the number of times it was called cannot be determined (probably because the function was not compiled with profiling enabled), the *calls* field is blank.

`self ms/call`
: This represents the average number of milliseconds spent in this function per call, if this function is profiled. Otherwise, this field is blank for this function.

`total ms/call`
: This represents the average number of milliseconds spent in this function and its descendants per call, if this function is profiled. Otherwise, this field is blank for this function.

`name`
: This is the name of the function. The flat profile is sorted by this field alphabetically after the *self seconds* field is sorted.

6 How to Read the Call Graph

The *call graph* shows how much time was spent in each function and its children. From this information, you can find functions that, while they themselves may not have used much time, called other functions that did use unusual amounts of time.

Here is a sample call from a small program. This call came from the same `gprof` run as the flat profile example in the previous chapter.

```
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]
-----------------------------------------------
```

The lines full of dashes divide this table into *entries*, one for each function. Each entry has one or more lines.

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 (also called *children* when we speak of the call graph).

The entries are sorted by time spent in the function and its subroutines.

The internal profiling function `mcount` (see Chapter 5 [Flat Profile], page 11) is never mentioned in the call graph.

6.1 The Primary Line

The *primary line* in a call graph entry is the line that describes the function which the entry is about and gives the overall statistics for this function.

For reference, we repeat the primary line from the entry for function `report` in our main example, together with the heading line that shows the names of the fields:

```
index  % time    self  children called     name
...
[3]    100.0    0.00    0.05       1         report [3]
```

Here is what the fields in the primary line mean:

`index`
: Entries are numbered with consecutive integers. Each function therefore has an index number, which appears at the beginning of its primary line.

  Each cross-reference to a function, as a caller or subroutine of another, gives its index number as well as its name. The index number guides you if you wish to look for the entry for that function.

`% time`
: This is the percentage of the total time that was spent in this function, including time spent in subroutines called from this function.

  The time spent in this function is counted again for the callers of this function. Therefore, adding up these percentages is meaningless.

`self`
: This is the total amount of time spent in this function. This should be identical to the number printed in the `seconds` field for this function in the flat profile.

`children`
: This is the total amount of time spent in the subroutine calls made by this function. This should be equal to the sum of all the `self` and `children` entries of the children listed directly below this function.

`called`
: This is the number of times the function was called.

  If the function called itself recursively, there are two numbers, separated by a '`+`'. The first number counts non-recursive calls, and the second counts recursive calls.

In the example above, the function `report` was called once from `main`.

`name` This is the name of the current function. The index number is repeated after it.

If the function is part of a cycle of recursion, the cycle number is printed between the function's name and the index number (see Section 6.4 [Cycles], page 16). For example, if function `gnurr` is part of cycle number one, and has index number twelve, its primary line would be end like this:

```
gnurr <cycle 1> [12]
```

6.2 Lines for a Function's Callers

A function's entry has a line for each function it was called by. These lines' fields correspond to the fields of the primary line, but their meanings are different because of the difference in context.

For reference, we repeat two lines from the entry for the function `report`, the primary line and one caller-line preceding it, together with the heading line that shows the names of the fields:

```
index  % time    self  children called     name
...
                 0.00     0.05      1/1           main [2]
[3]    100.0     0.00     0.05      1          report [3]
```

Here are the meanings of the fields in the caller-line for `report` called from `main`:

`self` An estimate of the amount of time spent in `report` itself when it was called from `main`.

`children` An estimate of the amount of time spent in subroutines of `report` when `report` was called from `main`.

The sum of the `self` and `children` fields is an estimate of the amount of time spent within calls to `report` from `main`.

`called` Two numbers: the number of times `report` was called from `main`, followed by the total number of nonrecursive calls to `report` from all its callers.

`name and index number`
The name of the caller of `report` to which this line applies, followed by the caller's index number.

Not all functions have entries in the call graph; some options to `gprof` request the omission of certain functions. When a caller has no entry of its own, it still has caller-lines in the entries of the functions it calls.

If the caller is part of a recursion cycle, the cycle number is printed between the name and the index number.

If the identity of the callers of a function cannot be determined, a dummy caller-line is printed which has '`<spontaneous>`' as the "caller's name" and all other fields blank. This can happen for signal handlers.

6.3 Lines for a Function's Subroutines

A function's entry has a line for each of its subroutines—in other words, a line for each other function that it called. These lines' fields correspond to the fields of the primary line, but their meanings are different because of the difference in context.

For reference, we repeat two lines from the entry for the function `main`, the primary line and a line for a subroutine, together with the heading line that shows the names of the fields:

```
index  % time    self  children called     name
...
[2]    100.0    0.00    0.05       1         main [2]
                0.00    0.05       1/1           report [3]
```

Here are the meanings of the fields in the subroutine-line for `main` calling `report`:

`self` An estimate of the amount of time spent directly within `report` when `report` was called from `main`.

`children` An estimate of the amount of time spent in subroutines of `report` when `report` was called from `main`.

The sum of the `self` and `children` fields is an estimate of the total time spent in calls to `report` from `main`.

`called` Two numbers, the number of calls to `report` from `main` followed by the total number of nonrecursive calls to `report`.

`name` The name of the subroutine of `main` to which this line applies, followed by the subroutine's index number.

If the caller is part of a recursion cycle, the cycle number is printed between the name and the index number.

6.4 How Mutually Recursive Functions Are Described

The graph may be complicated by the presence of *cycles of recursion* in the call graph. A cycle exists if a function calls another function that (directly or indirectly) calls (or appears to call) the original function. For example: if `a` calls `b`, and `b` calls `a`, then `a` and `b` form a cycle.

Whenever there are call-paths both ways between a pair of functions, they belong to the same cycle. If `a` and `b` call each other and `b` and `c` call each other, all three make one cycle. Note that even if `b` only calls `a` if it was not called from `a`, `gprof` cannot determine this, so `a` and `b` are still considered a cycle.

The cycles are numbered with consecutive integers. When a function belongs to a cycle, each time the function name appears in the call graph it is followed by '`<cycle` *number*`>`'.

The reason cycles matter is that they make the time values in the call graph paradoxical. The "time spent in children" of `a` should include the time spent in its subroutine `b` and in `b`'s subroutines—but one of `b`'s subroutines is `a`! How much of `a`'s time should be included in the children of `a`, when `a` is indirectly recursive?

The way `gprof` resolves this paradox is by creating a single entry for the cycle as a whole. The primary line of this entry describes the total time spent directly in the functions of the cycle. The "subroutines" of the cycle are the individual functions of the cycle, and all other functions that were called directly by them. The "callers" of the cycle are the functions, outside the cycle, that called functions in the cycle.

Here is an example portion of a call graph which shows a cycle containing functions `a` and `b`. The cycle was entered by a call to `a` from `main`; both `a` and `b` called `c`.

```
index  % time    self  children called     name
----------------------------------------
                 1.77        0    1/1        main [2]
[3]     91.71    1.77        0    1+5    <cycle 1 as a whole> [3]
                 1.02        0    3          b <cycle 1> [4]
                 0.75        0    2          a <cycle 1> [5]
----------------------------------------
                                  3          a <cycle 1> [5]
[4]     52.85    1.02        0    0      b <cycle 1> [4]
                                  2          a <cycle 1> [5]
                    0        0    3/6        c [6]
----------------------------------------
                 1.77        0    1/1        main [2]
                                  2          b <cycle 1> [4]
[5]     38.86    0.75        0    1      a <cycle 1> [5]
                                  3          b <cycle 1> [4]
                    0        0    3/6        c [6]
----------------------------------------
```

(The entire call graph for this program contains in addition an entry for `main`, which calls `a`, and an entry for `c`, with callers `a` and `b`.)

```
index  % time    self  children called     name
                                             <spontaneous>
[1]    100.00       0     1.93    0      start [1]
                 0.16     1.77    1/1        main [2]
```

```
-----------------------------------------------
                0.16    1.77    1/1        start [1]
[2]   100.00    0.16    1.77    1      main [2]
                1.77       0    1/1        a <cycle 1> [5]
-----------------------------------------------
                1.77       0    1/1        main [2]
[3]    91.71    1.77       0    1+5    <cycle 1 as a whole> [3]
                1.02       0    3          b <cycle 1> [4]
                0.75       0    2          a <cycle 1> [5]
                   0       0    6/6        c [6]
-----------------------------------------------
                                3          a <cycle 1> [5]
[4]    52.85    1.02       0    0      b <cycle 1> [4]
                                2          a <cycle 1> [5]
                   0       0    3/6        c [6]
-----------------------------------------------
                1.77       0    1/1        main [2]
                                2          b <cycle 1> [4]
[5]    38.86    0.75       0    1      a <cycle 1> [5]
                                3          b <cycle 1> [4]
                   0       0    3/6        c [6]
-----------------------------------------------
                   0       0    3/6        b <cycle 1> [4]
                   0       0    3/6        a <cycle 1> [5]
[6]     0.00       0       0    6      c [6]
-----------------------------------------------
```

The `self` field of the cycle's primary line is the total time spent in all the functions of the cycle. It equals the sum of the `self` fields for the individual functions in the cycle, found in the entry in the subroutine lines for these functions.

The `children` fields of the cycle's primary line and subroutine lines count only subroutines outside the cycle. Even though `a` calls `b`, the time spent in those calls to `b` is not counted in `a`'s `children` time. Thus, we do not encounter the problem of what to do when the time in those calls to `b` includes indirect recursive calls back to `a`.

The `children` field of a caller-line in the cycle's entry estimates the amount of time spent *in the whole cycle*, and its other subroutines, on the times when that caller called a function in the cycle.

The `calls` field in the primary line for the cycle has two numbers: first, the number of times functions in the cycle were called by functions outside the cycle; second, the number of times they were called by functions in the cycle (including times when a function in the cycle calls itself). This is a generalization of the usual split into nonrecursive and recursive calls.

The `calls` field of a subroutine-line for a cycle member in the cycle's entry says how many time that function was called from functions in the cycle. The total of all these is the second number in the primary line's `calls` field.

In the individual entry for a function in a cycle, the other functions in the same cycle can appear as subroutines and as callers. These lines show how many times each function in the cycle called or was called from each other function in the cycle. The `self` and `children` fields in these lines are blank because of the difficulty of defining meanings for them when recursion is going on.

7 Implementation of Profiling

Profiling works by changing how every function in your program is compiled so that when it is called, it will stash away some information about where it was called from. From this, the profiler can figure out what function called it, and can count how many times it was called. This change is made by the compiler when your program is compiled with the '-pg' option.

Profiling also involves watching your program as it runs, and keeping a histogram of where the program counter happens to be every now and then. Typically the program counter is looked at around 100 times per second of run time, but the exact frequency may vary from system to system.

A special startup routine allocates memory for the histogram and sets up a clock signal handler to make entries in it. Use of this special startup routine is one of the effects of using 'gcc ... -pg' to link. The startup file also includes an 'exit' function which is responsible for writing the file 'gmon.out'.

Number-of-calls information for library routines is collected by using a special version of the C library. The programs in it are the same as in the usual C library, but they were compiled with '-pg'. If you link your program with 'gcc ... -pg', it automatically uses the profiling version of the library.

The output from gprof gives no indication of parts of your program that are limited by I/O or swapping bandwidth. This is because samples of the program counter are taken at fixed intervals of run time. Therefore, the time measurements in gprof output say nothing about time that your program was not running. For example, a part of the program that creates so much data that it cannot all fit in physical memory at once may run very slowly due to thrashing, but gprof will say it uses little time. On the other hand, sampling by run time has the advantage that the amount of load due to other users won't directly affect the output you get.

8 Statistical Inaccuracy of `gprof` Output

The run-time figures that `gprof` gives you are based on a sampling process, so they are subject to statistical inaccuracy. If a function runs only a small amount of time, so that on the average the sampling process ought to catch that function in the act only once, there is a pretty good chance it will actually find that function zero times, or twice.

By contrast, the number-of-calls figures are derived by counting, not sampling. They are completely accurate and will not vary from run to run if your program is deterministic.

The *sampling period* that is printed at the beginning of the flat profile says how often samples are taken. The rule of thumb is that a run-time figure is accurate if it is considerably bigger than the sampling period.

The actual amount of error is usually more than one sampling period. In fact, if a value is *n* times the sampling period, the *expected* error in it is the square-root of *n* sampling periods. If the sampling period is 0.01 seconds and `foo`'s run-time is 1 second, the expected error in `foo`'s run-time is 0.1 seconds. It is likely to vary this much *on the average* from one profiling run to the next. (*Sometimes* it will vary more.)

This does not mean that a small run-time figure is devoid of information. If the program's *total* run-time is large, a small run-time for one function does tell you that that function used an insignificant fraction of the whole program's time. Usually this means it is not worth optimizing.

One way to get more accuracy is to give your program more (but similar) input data so it will take longer. Another way is to combine the data from several runs, using the '`-s`' option of `gprof`. Here is how:

1. Run your program once.
2. Issue the command '`mv gmon.out gmon.sum`'.
3. Run your program again, the same as before.
4. Merge the new data in '`gmon.out`' into '`gmon.sum`' with this command:

   `gprof -s` *executable-file* `gmon.out gmon.sum`
5. Repeat the last two steps as often as you wish.
6. Analyze the cumulative data using this command:

   `gprof` *executable-file* `gmon.sum >` *output-file*

9 Estimating `children` Times Uses an Assumption

Some of the figures in the call graph are estimates—for example, the `children` time values and all the the time figures in caller and subroutine lines.

There is no direct information about these measurements in the profile data itself. Instead, `gprof` estimates them by making an assumption about your program that might or might not be true.

The assumption made is that the average time spent in each call to any function `foo` is not correlated with who called `foo`. If `foo` used 5 seconds in all, and 2/5 of the calls to `foo` came from `a`, then `foo` contributes 2 seconds to `a`'s `children` time, by assumption.

This assumption is usually true enough, but for some programs it is far from true. Suppose that `foo` returns very quickly when its argument is zero; suppose that `a` always passes zero as an argument, while other callers of `foo` pass other arguments. In this program, all the time spent in `foo` is in the calls from callers other than `a`. But `gprof` has no way of knowing this; it will blindly and incorrectly charge 2 seconds of time in `foo` to the children of `a`.

We hope some day to put more complete data into '`gmon.out`', so that this assumption is no longer needed, if we can figure out how. For the nonce, the estimated figures are usually more useful than misleading.

10 Incompatibilities with Unix gprof

GNU gprof and Berkeley Unix gprof use the same data file 'gmon.out', and provide essentially the same information. But there are a few differences.

- For a recursive function, Unix gprof lists the function as a parent and as a child, with a calls field that lists the number of recursive calls. GNU gprof omits these lines and puts the number of recursive calls in the primary line.
- When a function is suppressed from the call graph with '-e', GNU gprof still lists it as a subroutine of functions that call it.
- The blurbs, field widths, and output formats are different. GNU gprof prints blurbs after the tables, so that you can see the tables without skipping the blurbs.

Table of Contents