Chapter 1

Profiling

In this chapter

- stopwatch   page 3
- date        page 4
- time        page 5
- clock       page 6
- gettimeofday page 11
- Performance Tuning Using GNU gprof page 13
- gcc Option Needed for gprof page 15
- kprof       page 31
- Summary     page 35
- Web Resources for Profiling page 36
In general, performance tuning consists of the following steps:

1. Define the performance problem.
2. Identify the bottlenecks by using monitoring and measurement tools. (This chapter focuses on measuring from the timing aspect.)
3. Remove bottlenecks by applying a tuning methodology.
4. Repeat steps 2 and 3 until you find a satisfactory resolution.

A sound understanding of the problem is critical in monitoring and tuning the system. Once the problem is defined, a realistic goal for improvement needs to be agreed on. Once a bottleneck is found, you need to verify whether it is indeed a bottleneck and devise possible solutions to alleviate it. Be aware that once a bottleneck is identified and steps are taken to relieve it, another bottleneck may suddenly appear. This may be caused by several variables in the system running near capacity.

Bottlenecks occur at points in the system where requests are arriving faster than they can be handled, or where resources, such as buffers, are insufficient to hold adequate amounts of data. Finding a bottleneck is essentially a step-by-step process of narrowing down the problem's causes.

Change only one thing at a time. Changing more than one variable can cloud results, since it will be difficult to determine which variable has had what effect on system performance. The general rule perhaps is better stated as “Change the minimum number of related things.” In some situations, changing “one thing at a time” may mean changing multiple parameters, since changes to the parameter of interest may require changes to related parameters. One key item to remember when doing performance tuning is to start in the same state every time. Start each iteration of your test with your system in the same state. For example, if you are doing database benchmarking, make sure that you reset the values in the database to the same setting each time the test is run.

This chapter covers several methods to measure execution time and real-time performance. The methods give different types of granularity, from the program's complete execution time to how long each function in the program takes. The first three methods (stopwatch, date, and time) involve no changes to the program that need
to be measured. The next two methods (**clock** and **gettimeofday**) need to be added directly to the program’s source code. The timing routines could be coded to be on or off, depending on whether the collection of performance measurements is needed all the time or just when the program’s performance is in question. The last method requires the application to be compiled with an additional compiler flag that allows the compiler to add the performance measurement directly to the code. Choosing one method over another can depend on whether the application’s source code is available. Analyzing the source code with gprof is a very effective way to see which function is using a large percentage of the overall time spent executing the program.

Application performance tuning is a complex process that requires correlating many types of information with source code to locate and analyze performance problem bottlenecks. This chapter shows a sample program that we’ll tune using gprof and gcov.

**stopwatch**

The stopwatch uses the chronograph feature of a digital watch. The steps are simple. Reset the watch to zero. When the program begins, start the watch. When the program ends, stop the watch. The total execution time is shown on the watch. Figure 1.1 uses the file system benchmark **dbench**. The stopwatch starts when dbench is started, and it stops when the program dbench is finished.
Using the digital stopwatch method, the dbench program execution time came out to be 13 minutes and 56 seconds, as shown in Figure 1.2.

00:13.56

FIGURE 1.2
The execution time is shown on the watch.

date

The `date` command can be used like a stopwatch, except that it uses the clock provided by the system. The `date` command is issued before the program is run and right after the program finishes. Figure 1.3 shows the output of the `date` command and the dbench program, which is a file system benchmark program. The execution time is 29 minutes and 59 seconds. This is the difference between the two times shown in the figure (17:52:24 – 17:22:25 = 29 minutes 59 seconds).

FIGURE 1.3
Using `date` to measure dbench timing.
time

The **time** command can be used to measure the execution time of a specified program. When the program finishes, **time** writes a message to standard output, giving timing statistics about the program that was run. Figure 1.4 shows the timing for the list directory contents command (**ls**) with the **-R** option, which recursively lists subdirectories.

![Figure 1.4 Timing the ls command with time.](image)
Figure 1.5 shows the finishing up of the `ls` command and the three timings (real, user, and sys) produced by `time`.

```
--- Makefile  Makefile.in  filter_discards  filter_test_paths  unused  ug_regtest.in  
--/ndir/ndir-2.0.0/tests/.deps: 
  ... true rule 
--/ndir/ndir-2.0.0/tests/.deps: 
  ... blocked_signal.c  pth_signal1.c  pth_simple_threads.c  signalall.c 
  ... megan.c  pth_signal2.c  pth_threadpool.c  timeout.c 
Makefile  pth_cancel.c  pth_signal2.c  pth_uct1.c  ucp.c 
Makefile.am  pth_panic.c  pth_signal.c  signalall.c 
Makefile.in  pth_semaphore.c  pth_simple_mutex.c  signal13.c 
--/ndir: 
  ... makefile  mk-mkfile  mk-mkfile.c  yam-0.32  yam-0.32.tar.gz 
--/ndir/yam-0.32: 
  ... XDGHOME  do-sym.c  libyam-dynamic.so  tests  yam-gcc.o  yam-os 
  ... $THINC  first.c  libyam.a  yam-g++  yam-memory 
COPING  digcom.a  first.o  libyam.a  yam-g++  yam-memory 
Makefile  do-sym.c  first.o  run-yam  yam-gcc  yam.c 
MKS  do-sym.c  gdb.a  run-yam  lib-yam-gcc  yam.o 
--/ndir/yam-0.32/test: 
  ... Makefile  test1.c  test2.c  test3.c  test4.c  test5.c  test6.c  test7.c  test8.c 
  ... main.c  test10.c  test12.c  test14.c  test2.c  test4.c  test6.c  test8.c 
real  4m56.09s 
user  0m5.26s 
sys  0m26.76s 
sfb=yam-0.32 
```

**FIGURE 1.5**
The results of timing the `ls` command with `time`.

The output from `time` produces three timings. The first is **real**, which indicates that 4 minutes and 58.045 seconds elapsed during the execution of the `ls` command, that the CPU in user space (**user**) spent 9.520 seconds, and that 26.760 seconds were spent executing system (**sys**) calls.

clock

The **clock()** function is a way to measure the time spent by a section of a program. The sample program shown in Listing 1.2, called sampleclock, measures two **for** loops. The first **for** loop is on line 27 of the sampleclock program, and the second is on line 69. The **delay_time** on lines 17 and 56 calculates how long the **clock** () call takes. The makefile shown in Listing 1.1 can be used to build the sampleclock program.
Listing 1.1

The Makefile for the sampleclock Program

Makefile for sampleclock program
CC = g++
CFLAGS = -g -Wall

sampleclock: sampleclock.cc
   $(CC) $(CFLAGS) sampleclock.cc -o sampleclock

clean:
   rm -f *.o sampleclock

Listing 1.2

sampleclock.cc
1 #include <iostream>
2 #include <ctime>
3 using namespace std;
4
5 // This sample program uses the clock() function to measure
6 // the time that it takes for the loop part of the program
7 // to execute
8
9 int main()
10 {
11   clock_t start_time, finish_time;
12
13   // get the delay of executing the clock() function
14   start_time = clock();
15   finish_time = clock();
16   double delay_time = (double)(finish_time - start_time);
17
18   cout << "Delay time:" << (double)delay_time << " seconds."
19   << endl;
20
21   // start timing
22   start_time = clock();
23
24   // Begin the timing
25
26   for (int i = 0; i < 100000; i++)
27   {
28
cout << "In: " << i << " loop " << endl;
}

// End the timing

// finish timing

finish_time = clock();

// compute the running time without the delay

double elapsed_iter_time = (double)(finish_time - start_time);
elapsed_iter_time -= delay_time;

// convert to second format

double elapsed_time = elapsed_iter_time / CLOCKS_PER_SEC;

// output the time elapsed

cout << "Elapsed time: " << (double)elapsed_time << " seconds. " << endl;

// get the delay of executing the clock() function

start_time = clock();

finish_time = clock();
delay_time = (double)(finish_time - start_time);

cout << "Delay time: " << (double)delay_time << " seconds. " << endl;

// now see what results we get by doing the measurement
// of the loop by cutting the loop in half

// start timing

start_time = clock();

// Begin the timing

for (int i = 0; i < 50000; i++)
{
    cout << "In: " << i << " loop " << endl;
}

// End the timing

// finish timing

finish_time = clock();
80  // compute the running time without the delay
81  elapsed_iter_time = (double)(finish_time - start_time);
82  elapsed_iter_time -= delay_time;
83  // convert to second format
84  elapsed_time = elapsed_iter_time / CLOCKS_PER_SEC;
85  // output the time elapsed.
86  cout << "Elapsed time:" << (double)elapsed_time << " seconds."
87  << endl;
88  return 0;
89  
90 
91 }
92
93
94
95

The sampleclock.cc program can be built by executing the \texttt{make} command. Figure 1.6 shows the building and running of the sampleclock program.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{sampleclock.png}
\caption{Building and running sampleclock.}
\end{figure}
Figure 1.7 shows the elapsed time for the first loop as 3.11 seconds.

Figure 1.8 shows the elapsed time for the second loop as 1.66 seconds.

So the sampleclock program takes 3.11 seconds to execute the first `for` loop of 100000 and 1.66 seconds for the second `for` loop of 50000, which is very close to half of the time. Now let's look at another API called `gettimeofday` that can also be used to time functions in a program.
gettimeofday

gettimeofday() returns the current system clock time. The return value is a list of two integers indicating the number of seconds since January 1, 1970 and the number of microseconds since the most recent second boundary.

The sampletime code shown in Listing 1.3 uses gettimeofday to measure the time it takes to sleep for 200 seconds. The gettimeofday routine could be used to measure how long it takes to write or read a file. Listing 1.4 is the pseudocode that could be used to time a write call.

Listing 1.3

sampletime.c
1  #include <stdio.h>
2  #include <sys/time.h>
3
4 struct timeval start, finish;
5 int msec;
6
7 int main()
8 {
9   gettimeofday(&start, NULL);
10  sleep(200); /* wait ~ 200 seconds */
11  gettimeofday(&finish, NULL);
12  msec = finish.tv_sec * 1000 + finish.tv_usec / 1000;
13  msec -= start.tv_sec * 1000 + start.tv_usec / 1000;
14  printf("Time: %d milliseconds\n", msec);
15 }

Figure 1.9 shows the building of sampletime.c and the program's output. Using gettimeofday, the time for the sleep call on line 11 is 20009 milliseconds.

Listing 1.4 shows pseudocode for measuring the write call with the gettimeofday API. The gettimeofday routine is called before the write routine is called to get the start time. After the write call is made, gettimeofday is called again to get the end time. Then the elapse_time for the write can be calculated.

Listing 1.4

Pseudocode for Timing Write Code
1    /* get time of day before writing */
2    if ( gettimeofday(&tp_start, NULL) == -1 )
3        {
4        /* error message gettimeofday failed */
5  } } 
6  /* calculate elapse_time_start */
7  /* write to disk */
8  for ( i = 0; i < count; i++ )
9  {
10    if ( write( fd, buf, buf_size ) == 0 )
11      /* error message write failed */
12  }
13 /* get time of day after write */
14  if ( gettimeofday( &tp_end, NULL ) == -1 )
15  {
16    /* error message gettimeofday failed */
17  }
18 /* calculate elapse_time_new */
19  elapse_time = elapse_time_new - elapse_time_start;
20 /* compute throughput */
21  printf( "elapse time for write: %d \n", elapse_time );

Raw timings have limited usage when looking for performance issues. Profilers can help pinpoint the parts of your program that are using the most time.

Performance Tuning Using GNU gprof

A profiler provides execution profiles. In other words, it tells you how much time is being spent in each subroutine or function. You can view two kinds of extreme profiles: a sharp profile and a flat profile.

Typically, scientific and engineering applications are dominated by a few routines and give sharp profiles. These routines are usually built around linear algebra solutions. Tuning code should focus on the most time-consuming routines and can be very rewarding if successful.

Programs with flat profiles are more difficult to tune than ones with sharp profiles. Regardless of the code’s profile, a subroutine (function) profiler, gprof, can provide a key way to tune applications.

Profiling tells you where a program is spending its time and which functions are called while the program is being executed. With profile information, you can determine which pieces of the program are slower than expected. These sections of the code can be good candidates to be rewritten to make the program execute faster. Profiling is also the best way to determine how often each function is called. With this information, you can determine which function will give the most performance boost by changing the code to perform faster.
The profiler collects data during the program’s execution. Having a complete analysis of the program helps you ensure that all its important paths are while the program is being profiled. Profiling can also be used on programs that are very complex. This could be another way to learn the source code in addition to just reading it. Now let’s look at the steps needed to profile a program using gprof:

- Profiling must be enabled when compiling and linking the program.
- A profiling data file is generated when the program is executed.
- Profiling data can be analyzed by running gprof.

  gprof can display two different forms of output:

  - A flat profile displays the amount of time the program went into each function and the number of times the function was executed.
  - A call graph displays details for each function, which function(s) called it, the number of times it was called, and the amount of time that was spent in the subroutines of each function. Figure 1.10 shows part of a call graph.

  ![Call Graph Diagram](image)

  **FIGURE 1.10**
  A typical fragment of a call graph.

  gprof is useful not only to determine how much time is spent in various routines, but also to tell you which routines call (invoke) other routines. Suppose you examine gprof’s output and see that xyz is consuming a lot of time, but the output doesn’t tell you which routine is calling xyz. If there were a call tree, it would tell you where the calls to xyz were coming from.
Before programs can be profiled using gprof, they must be compiled with the -pg gcc option. To get complete information about gprof, you can use the command info gprof or man gprof.

Listing 1.5 shows the benefits that profiling can have on a small program. The sample1 program prints the prime numbers up to 50,000. You can use the output from gprof to increase this program’s performance by changing the program to sample2, shown later in Listing 1.8.

**Listing 1.5**

*sample1.c*

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

int prime (int num);

int main()
{
  int i,
  int colcnt = 0;
  for (i=2; i <= 50000; i++)
    if (prime(i)) {
      colcnt++;
      if (colcnt%9 == 0) {
        printf("%5d\n",i);
        colcnt = 0;
      }
    else
      printf("%5d ", i);
      putchar(\n');
    return 0;
}

int prime (int num) {
 /* check to see if the number is a prime? */
  int i,
  for (i=2; i < num; i++)
    if (num %i == 0)
      return 0;
  return 1;
```
Building the sample1 Program and Using gprof

The sample1.c program needs to be compiled with the option `-pg` to have profile data generated, as shown in Figure 1.11.

![Figure 1.11](image)

Building and running sample1.

When the sample1 program is run, the gmon.out file is created.

To view the profiling data, the gprof utility must be on your system. If your system is `rpm`-based, the `rpm` command shows the version of gprof, as shown in Figure 1.12.
gcc Option Needed for gprof

Figure 1.12
The version of gprof.

gprof is in the binutils package. For you to use the utility, the package must be installed on your system. One useful gprof option is -b. The -b option eliminates the text output that explains the data output provided by gprof:

```
# gprof -b ./sample1
```

The output shown in Listing 1.6 from gprof gives some high-level information like the total running time, which is 103.74 seconds. The main routine running time is 0.07 seconds, and the prime routine running time is 103.67 seconds. The prime routine is called 49,999 times.

Listing 1.6

Output from gprof for sample1

Flat profile:

<table>
<thead>
<tr>
<th>%</th>
<th>cumulative</th>
<th>self</th>
<th>self</th>
<th>total</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>seconds</td>
<td>seconds</td>
<td>calls</td>
<td>ms/call</td>
<td>ms/call</td>
</tr>
<tr>
<td>99.93</td>
<td>103.67</td>
<td>103.67</td>
<td>49999</td>
<td>2.07</td>
<td>2.07</td>
</tr>
<tr>
<td>0.07</td>
<td>103.74</td>
<td>0.07</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Call graph

granularity: each sample hit covers 4 byte(s) for 0.01% of 103.74 seconds

index % time self children called name <spontaneous>
[1] 100.0  0.07  103.67  main [1]
103.67  0.00  49999/49999 prime [2]
Next we can use the gcov program to look at the actual number of times each line of the program was executed. (See Chapter 2, “Code Coverage,” for more about gcov.)

We will build the sample1 program with two additional options—\texttt{-fprofile-arcs} and \texttt{-ftest-coverage}, as shown in Figure 1.13. These options let you look at the program using gcov, as shown in Figure 1.14.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure1_13}
\caption{Building sample1 with gcov options.}
\end{figure}
Running gcov on the source code produces the file sample1.c.gcov. It shows the actual number of times each line of the program was executed. Listing 1.7 is the output of gcov on sample1.

Listing 1.7

Output from gcov for sample1

- 0:Source:sample1.c
- 0:Graph:sample1.bbg
- 0:Data:sample1.dbg
- 1:#include <stdlib.h>
- 2:#include <stdio.h>
- 3:
- 4:int prime (int num);
- 5:
- 6:int main()
1: 7: 

FIGURE 1.14
Running sample1 and creating gcov output.

Running gcov on the source code produces the file sample1.c.gcov. It shows the actual number of times each line of the program was executed. Listing 1.7 is the output of gcov on sample1.
There are 5,133 prime numbers. The expensive operations in the routine prime are the for loop (line 27) and the if statement (line 28). The “hot spots” are the loop and the if test inside the prime routine. This is where we will work to increase the program’s performance. One change that will help this program is to use the sqrt() function, which returns the nonnegative square root function of the number passed in. sample2, shown in Listing 1.8, has been changed to use the sqrt function in the newly created function called faster.

Listing 1.8

sample2.c

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <math.h>
4
5 int prime (int num);
6 int faster (int num);
7
8 int main()
9 {
```
```c
int i;
int colcnt = 0;
for (i=2; i <= 50000; i++)
    if (prime(i)) {
        colcnt++;
        if (colcnt%9 == 0) {
            printf("%5d
", i);
            colcnt = 0;
        } else
            printf("%5d ", i);
    } else
        printf("%5d ", i);
putchar(\n);
return 0;
}

int prime (int num) {
    /* check to see if the number is a prime? */
    int i;
    for (i=2; i <= faster(num); i++)
        if (num %i == 0)
            return 0;
    return 1;
}

int faster (int num)
{
    return (int) sqrt((float) num);
}
```

Now you can build the sample2 program (see Figure 1.15) and use gprof to check how long the program will take to run (see Figure 1.16). Also, the gcov output shows the reduced number of times each line needs to be executed. In Listing 1.9, the total running time has been reduced from 103.74 seconds to 2.80 seconds.

Listing 1.9 shows the output of gprof for the sample2 program.

### Listing 1.9
**Output from gprof for sample2**

Flat profile:

<table>
<thead>
<tr>
<th>% cumulative</th>
<th>time</th>
<th>self cumulative</th>
<th>self seconds</th>
<th>calls</th>
<th>us/call</th>
<th>total seconds</th>
<th>us/call</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>52.68</td>
<td>1.48</td>
<td>1.48</td>
<td>1061109</td>
<td>1.39</td>
<td>1.39</td>
<td>faster</td>
<td></td>
<td></td>
</tr>
<tr>
<td>46.61</td>
<td>2.78</td>
<td>1.30</td>
<td>49999</td>
<td>26.10</td>
<td>55.60</td>
<td>prime</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.71</td>
<td>2.80</td>
<td>0.02</td>
<td></td>
<td></td>
<td></td>
<td>main</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Call graph

granularity: each sample hit covers 4 byte(s) for 0.36% of 2.80 seconds

<table>
<thead>
<tr>
<th>index</th>
<th>% time</th>
<th>self</th>
<th>children</th>
<th>called</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td>100.0</td>
<td>0.02</td>
<td>2.78</td>
<td></td>
<td>main [1]</td>
</tr>
<tr>
<td></td>
<td>1.30</td>
<td>1.48</td>
<td>49999/49999</td>
<td>prime [2]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.48</td>
<td>0.00</td>
<td>1061109/1061109</td>
<td>prime [2]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.48</td>
<td>0.00</td>
<td>1061109/1061109</td>
<td>faster [3]</td>
<td></td>
</tr>
<tr>
<td>[3]</td>
<td>52.7</td>
<td>1.48</td>
<td>0.00</td>
<td>1061109</td>
<td>faster [3]</td>
</tr>
</tbody>
</table>

Index by function name

prime

FIGURE 1.15
Building and running sample2.
gcc Option Needed for gprof

FIGURE 1.16
Using gprof on sample2.
Now we'll run gcov on the sample2 program, as shown in Figures 1.17 and 1.18.

### FIGURE 1.17
Building sample2 with gcov and running sample2.
Listing 1.10 shows gcov output for the sample2 program.

**Listing 1.10**

**Output of sample2.c.gcov**

```c
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

int prime (int num);
int faster (int num);

int main()
{
    int i;
    int colcnt = 0;
    for (i=2; i <= 50000; i++)
```

**FIGURE 1.18**
Running sample2 and getting gcov output.
The for loop in the prime routine has been reduced from 121 million executions to 1 million executions. Therefore, the total time has been reduced from 103.74 seconds to 2.80 seconds.

The tools gprof and gcov helped find the “hot spots” in this sample program. After the “hot spots” were found, the program was changed to increase its overall performance. It is interesting how changing a few lines of code can have a great impact on a program’s performance.

Listing 1.11, sample3.cpp, has three different functions (1, 2, and 3). It shows a more complex use of profiling, with both flat and graphic profiles. We’ll also use kprof, which can use gprof output. It presents the information in list or tree views, which make the information easier to understand when programs are more complicated. Let’s start by building the sample3.cpp program and displaying the flat and graphic profiles and then displaying the data using kprof.
Listing 1.11

sample3.cpp

```cpp
#include <iostream>

void function1()
{
    for(int i=0;i<1000000;i++);
}

void function2()
{
    function1();
    for (int i=0;i<2000000;i++);
}

void function3()
{
    function1();
    function2();
    for (int i=0;i<3000000;i++);
    function1();
}

int main()
{
    for(int i=0;i<10;i++)
        function1();
    for (int i=0;i<5000000;i++);
    for(int i=0;i<10;i++)
    {
        function3();
        function2();
        function1();
    }
}
```

Figure 1.19 shows the commands used to build and run the sample3 program. gprof is also run on sample3 to get the profile data from sample3.

FIGURE 1.19
Building and capturing gprof output for sample3.
We won’t use the -b option on the gprof output on the sample3 program so that we can see all the descriptive information that gprof can display.

The sample3.gprof should look similar to this:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>% time</td>
<td>The percentage of the program’s total running time used by this function.</td>
</tr>
<tr>
<td>cumulative</td>
<td>A running sum of the number of seconds accounted for by this function and</td>
</tr>
<tr>
<td>self</td>
<td>those listed above it.</td>
</tr>
<tr>
<td>calls</td>
<td>The number of times this function was invoked if this function is profiled;</td>
</tr>
<tr>
<td>self ms/call</td>
<td>otherwise, it is blank.</td>
</tr>
<tr>
<td>total ms/call</td>
<td>The average number of milliseconds spent in this function per call if this</td>
</tr>
<tr>
<td></td>
<td>function is profiled; otherwise, it is blank.</td>
</tr>
<tr>
<td>name</td>
<td>The function’s name. This is the minor sort for this listing. The index</td>
</tr>
<tr>
<td></td>
<td>shows the location of the function in the gprof listing. If the index is in</td>
</tr>
<tr>
<td></td>
<td>parentheses, it shows where it would appear in the gprof listing if it were</td>
</tr>
<tr>
<td></td>
<td>to be printed.</td>
</tr>
</tbody>
</table>

Call graph (explanation follows)

granularity: each sample hit covers 4 byte(s) for 0.10% of 9.71 seconds

<table>
<thead>
<tr>
<th>index</th>
<th>% time</th>
<th>self</th>
<th>children</th>
<th>called</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td>100.0</td>
<td>0.84</td>
<td>8.87</td>
<td></td>
<td>&lt;spontaneous&gt;</td>
</tr>
<tr>
<td></td>
<td>3.86</td>
<td>1.83</td>
<td>11/12</td>
<td></td>
<td>main [1]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>function2() [2]</td>
</tr>
</tbody>
</table>
This table describes the program's call tree. It is 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 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.

You see the following:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>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 that it is easier to look up the function in the table.</td>
</tr>
<tr>
<td>% time</td>
<td>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, and so on, these numbers do not add up to 100%.</td>
</tr>
<tr>
<td>Field</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>self</td>
<td>The total amount of time spent in this function.</td>
</tr>
<tr>
<td>children</td>
<td>The total amount of time propagated into this function by its children.</td>
</tr>
<tr>
<td>called</td>
<td>The number of times the function was called. If the function called itself recursively, the number includes only nonrecursive calls and is followed by a + and the number of recursive calls.</td>
</tr>
<tr>
<td>name</td>
<td>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.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>self</td>
<td>The amount of time that was propagated directly from the function into this parent.</td>
</tr>
<tr>
<td>children</td>
<td>The amount of time that was propagated from the function's children into this parent.</td>
</tr>
</tbody>
</table>
| called  | The number of times this parent called the function and the total number of times the function was called. Recursive calls to the function are not included in the number after the /.
| name    | The parent's name. 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 function's parents cannot be determined, the word <spontaneous> is printed in the name field, and all the other fields are blank.

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>self</td>
<td>The amount of time that was propagated directly from the child into the function.</td>
</tr>
<tr>
<td>children</td>
<td>The amount of time that was propagated from the child's children to the function.</td>
</tr>
</tbody>
</table>
called The number of times the function called this child and the total
number of times the child was called. Recursive calls by the child are
not listed in the number after the /.

name The child’s name. 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 the call graph has any cycles (circles), 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 how many function calls were internal
to the cycle. 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
[12] __static_initialization_and_destruction_0(int, int) [2] function2() [1]
main

kprof

kprof is a graphical tool that displays the execution profiling output generated by
the gprof profiler. kprof presents the information in list or tree view, which makes
the information easy to understand.

kprof has the following features:

- Flat profile view displays all functions and methods and their profiling
  information. (See Figure 1.22 for a view of this functionality.)

- Hierarchical profile view displays a tree for each function and method with
  the other functions and methods it calls as subelements. (See Figure 1.23
  for a view of this functionality.)

- Graph view is a graphical representation of the call tree. It requires
  Graphviz to work. (See Figure 1.24 for a view of this functionality.)

- Right-clicking a function or method displays a pop-up with the list of
callers and called functions. You can go to one of these functions directly
by selecting it in the pop-up menu. (See Figure 1.22 for a view of this
functionality.)
Installation

We've installed the kprof-1.4.2-196.i586.rpm that comes with the distribution. The following `rpm` command displays the version of the kprof application:

```
% rpm -qf /opt/kde3/bin/kprof
kprof-1.4.2-196
```

Building Graphviz, the Graph Feature

kprof supports a graph feature, but before it can be used, the Graphviz program must be built. See the Graphviz URL in the section “Web Resources for Profiling” at the end of this chapter to download the source code for Graphviz.

The version of source code for Graphviz that will be built for this section is version 1.12. The tar file graphviz-1.12.tar.gz can be downloaded.

The next steps expand the source tree. Then, using the `make` and `make install` commands, the program is built and installed to the proper location on your system, as shown in Figure 1.20.

![Building and installing Graphviz.](Best_Ch01f.qxd 8/31/05 12:55 PM Page 32)

FIGURE 1.20
Building and installing Graphviz.

After Graphviz is installed, kprof uses it to create the Graph View that can be seen in Figure 1.24.

To use kprof, the `-b` option is needed. The following command uses gprof with the `-b` option on the sample3 program. gprof’s output is saved to the sample3.prof1 file:

```
% gprof -b sample3 >sample3.prof1
```

The next step is to start kprof:

```
% kprof
```
After kprof loads, select File, Open to bring the sample3 gprof output into kprof. Figure 1.21 shows the open dialog box.

![The open dialog box.](image)

**FIGURE 1.21**
The open dialog box.

Figure 1.22 shows the flat profile view of the sample3 program. This screen shot also shows that function1 is called by function2, function3, and main.

![The flat profile view.](image)

**FIGURE 1.22**
The flat profile view.
Figure 1.23 shows the hierarchical profile view of the sample3 program.

![Hierarchical Profile View](image)

**FIGURE 1.23**
The hierarchical profile view.

Figure 1.24 shows the graph view of the sample3 program. The graph view uses Graphviz. This view shows that function1 is called by main, function2, and function3. It also shows that function2 is called by main and function3 and that function3 is called only by main.
Summary

This chapter covered five methods of timing programs or functions inside of programs. The first three methods were stopwatch, date, and time. These three methods are ways to measure the total time that the program takes to execute. These methods require no modifications to the program to measure the time spent by the program. The clock and gettimeofday routines can be added to parts of a program to measure the time spent doing a section of the program. Finally, the gprof profiler and kprof can be used to profile sample programs.

FIGURE 1.24
The graph view.
# Web Resources for Profiling

<table>
<thead>
<tr>
<th>URL</th>
<th>Description</th>
</tr>
</thead>
</table>