Chapter 1



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

n 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

stopwatch

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.



FIGURE 1.1 Timing dbench with stopwatch.

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).

🕼 Shell - Konsole 🧶

Session Edit View Bookmarks Settings Hep sfb1:/usr/src/dbench/dbench-2.1 # date && ./dbench 20 && date Tue Jun 1 17:22:25 PDT 2004 20 clients started 0 62477 3.87 MB/sec Throughput 3.87242 MB/sec 20 procs Tue Jun 1 17:52:24 PDT 2004 sfb1:/usr/src/dbench/dbench-2.1 #

FIGURE 1.3 Using date to measure dbench timing.

time

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.

Session Edit View Bookmarks	s Settings Help	
sfb1:/usr/src # time .:		
	kdb	linux-2.6.4-rc1.tar.gz
	kernel-modules	lsof
2.2.9	kprof	ltt
2.4.16	limon	memwatch
cdrecord	linux	mesa
clock	linux-2.4.21	nfsac1-2.6.1-0.8.67.tar.gz
dbench	linux-2.4.21-99	packages
ddd	linux-2.4.21-99-include	patch-2.4.24-pre1
ea-2.4.24-0.8.68.dif		php
electric	linux-2.4.23	ppcboot
gdb	linux-2.4.23.tar.gz	sample3
get	linux-2.4.26	sformat
gettime	linux-2.4.26.tar.gz	suse90-linux
graphuiz	linux-2.6.2	timing
hmckernel	linux-2.6.2.tar.gz	valgrind
insight	linux-2.6.3	yamd
jfsutils	linux-2.6.3.tar.gz	
kbd	linux-2.6.4-rc1	
./2.2.9:		
./2.4.16:		
linux		
linux-2.4.16.tar.gz		

FIGURE 1.4

Timing the **Is** command with **time**.

6

Chapter 1 • Profiling

Figure 1.5 shows the finishing up of the ls command and the three timings (**real**, **user**, and **sys**) produced by time.

	Konsole <3>	A 10				statute dross
	View Bookmarks					
Makef	`ile Makefil	le.in filt	er_discards filter	_test_paths unus	sed ug_reg	test.in
./valgrin	d∕valgrind-2	2.0.0/tests	∕.deps:			
0.0	d∕valgrind-2	0 0 dtaata	(unucod)			
.voargrin			pth_signal1.c	pth_simple_thread:	s.c. simwai	t_all.c
	oneparar		pth_signal2.c	pth_shipic_chicaa	twopara	
Makefile	pth_cano		pth_signal_gober.c		twopara	
Makefile.			pth_sigpending.c	signal1.c	9. 7 .21	
Makefile.	in pth_sema	phore1.c	pth_simple_mutex.c	signal3.c		
ma		ory1.c mem	ory1.c~yamd-0.32	yamd-0.32.tar.gz		
ma		do-syms.o	libyamd-dynamic.so	tests yamd	-gcc.o	yamd.os
ma ./yamd/ya	md-0.32: README	do-syms.o first.c	3	tests yamd- yamd-g++ yamd-	-gcc.o -memory1 -memory1.c	yamd.os
	umd-0.32: README TODO dbgcom.dif	do-syms.o first.c	libyamd-dynamic.so libyamd.a	tests yamd- yamd-g++ yamd-	-memory1 -memory1.c	yamd.os
ma ./yamd/ya COPYING Makefile	umd-0.32: README TODO dbgcom.dif	do-syms.o first.c first.o	libyamd-dynamic.so libyamd.a libyamdf.a	tests yamd- yamd-g++ yamd- yamd-g++.o yamd-	-memory1 -memory1.c .c	yamd.os
.∕yamd∕ya COPYING Makefile NEWS	md-0.32: README TODO dbgcom.dif do-syms do-syms.c	do-syms.o first.c first.o first.os gdb.dif	libyamd-dynamic.so libyamd.a libyamdf.a run-yamd	tests yamd- yamd-g++ yamd- yamd-g++.o yamd- yamd-gcc yamd	-memory1 -memory1.c .c	yamd.os
ma ./yamd/ya COPYING Makefile NEWS ./yamd/ya	md-0.32: README TODO dbgcom.dif do-syms	do-syms.o first.c first.o first.os gdb.dif cs:	libyand-dynamic.so libyand.a libyandf.a run-yand run-yand.in	tests yand- yamd-g++ yamd- yamd-g++.o yamd- yamd-gcc yamd yamd-gcc.c yamd	-memory1 -memory1.c .c .o	1000 - 1000 mls
ma yamd/ya Makefile MEWS /yamd/ya . Makef	md-0.32: README TODO dbgcom.dif do-syms do-syms.c md-0.32/test `ile test1.c	do-syms.o first.c first.o first.os gdb.dif cs: test11.	libyand-dynamic.so libyand.a libyandf.a run-yand run-yand.in	tests yand- yand-g++ yand- yand-g++.o yand- yand-gcc yand yand-gcc.c yand c test3.c test5.u	-memory1 -memory1.c .c .o c test7.c	1000 - 1000 mls
ma ./yamd/ya COPYING Makefile NEWS ./yamd/ya . Makef main.	md-0.32: README TODO dbgcom.dif do-syms do-syms.c md-0.32/test ile test1.c c test10.	do-syms.o first.c first.o first.os gdb.dif cs: test11.	libyamd-dynamic.so libyamd.a libyamdf.a run-yamd run-yamd.in c test13.c test15.	tests yand- yand-g++ yand- yand-g++.o yand- yand-gcc yand yand-gcc.c yand c test3.c test5.u	-memory1 -memory1.c .c .o c test7.c	1000 - 1000 mls
ma ./yamd/ya COPYING Makefile NEWS ./yamd/ya . Makef main. real 4	md-0.32: README TODO dbgcom.dif do-syms.c und-0.32/test `ile test1.c c test10. m58.045s	do-syms.o first.c first.o first.os gdb.dif cs: test11.	libyamd-dynamic.so libyamd.a libyamdf.a run-yamd run-yamd.in c test13.c test15.	tests yand- yand-g++ yand- yand-g++.o yand- yand-gcc yand yand-gcc.c yand c test3.c test5.u	-memory1 -memory1.c .c .o c test7.c	1000 - 1000 mls
ma ./yamd/ya COPYING Makefile NEWS ./yamd/ya . Makef main. real 4 user 0	md-0.32: README TODO dbgcom.dif do-syms do-syms.c md-0.32/test ile test1.c c test10.	do-syms.o first.c first.o first.os gdb.dif cs: test11.	libyamd-dynamic.so libyamd.a libyamdf.a run-yamd run-yamd.in c test13.c test15.	tests yand- yand-g++ yand- yand-g++.o yand- yand-gcc yand yand-gcc.c yand c test3.c test5.u	-memory1 -memory1.c .c .o c test7.c	1000 - 1000 mls

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. clock

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>
  #include <ctime>
2
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
15
    start time = clock();
    finish time = clock();
16
    double_delay_time = (double) (finish_time - start_time);
17
18
19
     cout<<"Delay time:"<<(double)delay time<<" seconds."</pre>
      <<endl;
20
21
    // start timing
22
23 start time = clock();
24
25
   // Begin the timing
26
27
    for (int i = 0; i < 100000; i++)
28
      ł
```

```
cout <<"In:"<<i<" loop" << endl;</pre>
29
30
31
    // End the timing
32
33
34
    // finish timing
35
    finish time = clock();
36
37
38
    // compute the running time without the delay
39
40
    double elapsed iter time = (double) (finish time - start
      time);
41
    elapsed iter time -= delay time;
42
43
    // convert to second format
44
    double elapsed time = elapsed iter time / CLOCKS PER SEC;
45
46
    // output the time elapsed
47
48
    cout<<"Elapsed time:"<<(double)elapsed time<<" seconds."</pre>
49
      <<endl;
50
51
    // get the delay of executing the clock() function
52
53
    start_time = clock();
54
55
    finish time = clock();
    delay_time = (double)(finish_time - start_time);
56
57
58
    cout<<"Delay time:"<<(double)delay time<<" seconds."<<endl;</pre>
59
60
    // now see what results we get by doing the measurement
    // of the loop by cutting the loop in half
61
62
    // start timing
63
64
    start time = clock();
65
66
67
    // Begin the timing
68
69
    for (int i = 0; i < 50000; i++)
70
      cout <<"In:"<<i<" loop" << endl;</pre>
71
72
73
74
    // End the timing
75
76
    // finish timing
77
    finish time = clock();
78
79
```

```
clock
```

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

The sampleclock.cc program can be built by executing the **make** command.

Figure 1.6 shows the building and running of the sampleclock program.

🖫 si	iell - Konsole 🧕
Session	Edit View Bookmanks Settings Help
	usr/src/clock # make
	-Wall sampleclock.cc -o sampleclock
	usr/src/clock # ./sampleclock
	time:0 seconds.
In:0 1 In:1 1	oop
In:2 1	
In:3 1	
In:4 1	
In:5 1	
In:6 1	
In:7 1	
In:8 1	
In:9 1	
In:10	loop
In:11	loop
In:12	loop
In:13	
In:14	
In:15	
In:16	
In:17	
In:18	
In:19	
In:20	
	loop
In:22 In:23	
In:24	
111.23	TOOP



Figure 1.7 shows the elapsed time for the first loop as 3.11 seconds.

🕼 Shell - Konsole 🍥
Session Edit View Bookmarks Settings Help
In:99992 loop
In:99993 loop
In:99994 loop
In:99995 loop
In:99996 loop
In:99997 loop
In:99998 loop
In:99999 loop
Elapsed time:3.11 seconds.
Delay time:0 seconds.
In:0 loop
In:1 loop
In:2 loop
In:3 loop
In:4 loop
In:5 loop
In:6 loop
In:7 loop
In:8 loop
In:9 loop
In:10 loop
In:11 loop
In:12 loop
In:13 loop
In:14 loop
In:15 loop
In:16 loop
In:17 loop
In:18 loop
In:19 loop

FIGURE 1.7 The timing for loop 1.

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.

10

gettimeofday

5	S	ıell -	Konso	e 9		
Sessi	on	Edit	View	Bookmarks	Settings	Неір
In:4	99	72	loop			
In:4	99	73	loop			
In:4	99	74	loop			
In:4	99	75	loop			
In:4	99	76	loop			
In:4	99	77	loop			
In:4	99	78	loop			
In:4			loop			
In:4			loop			
In:4			loop			
In:4			loop			
In:4			loop			
In:4			loop			
In:4			loop			
In:4			loop			
In:4			loop			
In:4			loop			
In:4			loop			
In:4			loop			
In:4			loop			
In:4			loop			
In:4			loop			
In:4			loop			
In:4			loop			
In:4			loop			
In:4 In:4			loop			
In:4			loop			
			loop	1.66 sec	obroo	
				clock t		
2101	./	usr	/SLC	CIUCK 1	•	

FIGURE 1.8

The timing for loop 2.

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
```

```
struct timeval start, finish ;
4
5
   int msec;
6
7
   int main ()
8
9
    gettimeofday (&start, NULL);
10
    sleep (200); /* wait ~ 200 seconds */
11
12
13
    gettimeofday (&finish, NULL);
14
15
    msec = finish.tv sec * 1000 + finish.tv usec / 1000;
    msec -= start.tv sec * 1000 + start.tv usec / 1000;
16
17
    printf("Time: %d milliseconds\n", msec);
18
19
```

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 200009 milliseconds.

```
Session Edit View Bookmarks Settings Heb

Session Edit View Bookmarks Settings Heb

Sfb1:/usr/src/gettime # gcc sampletime.c -o sampletime

Sfb1:/usr/src/gettime # ./sampletime

Time: 200009 milliseconds

sfb1:/usr/src/gettime #
```

```
FIGURE 1.9
Timing using gettimeofday.
```

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 */
```

Performance Tuning Using GNU gprof

```
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
14
15
    /* get time of day after write */
    if (gettimeofday( &tp end, NULL ) == -1 )
16
17
           error message gettimeofday failed */
18
19
20
    /*
      calculate elapse time new */
21
    elapse time = elapse time new - elapse time start;
22
    /* compute throughput */
    printf( "elapse time for write: %d \n", elapse time );
23
```

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.

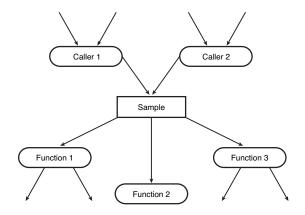
Profiling tells you where a program is spending its time and which functions are called while the program is being executed. With profile information, you can determine which pieces of the program are slower than expected. These sections of the code can be good candidates to be rewritten to make the 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.





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.

gcc Option Needed for gprof

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

```
#include <stdlib.h>
1
2
   #include <stdio.h>
3
   int prime (int num);
4
5
6
   int main()
7
   {
8
    int i;
9
    int colcnt = 0;
10
    for (i=2; i <= 50000; i++)
      if (prime(i)) {
11
12
         colcnt++;
13
         if (colcnt%9 == 0) {
            printf("%5d\n",i);
14
15
            colcnt = 0;
16
17
      else
18
          printf("%5d ", i);
19
      }
      putchar('\n');
20
21
      return 0;
22 }
23
24 int prime (int num)
        /* check to see if the number is a prime? */
25
26
       int i;
27
       for (i=2; i < num; i++)
28
       if (num %i == 0)
29
          return 0;
30
       return 1;
31 }
```

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.

🕼 si	iell - Kon	sole S)											
Session	Session Edit View Bookmanks Settings Help													
sfb1:/	sfb1:/usr/src/sample1 # gcc -pg -o sample1 sample1.c sfb1:/usr/src/sample1 # ./sample1													
sfb1:/	usr/sr	c/samp	le1 #	./samp	le1									
2	З	5	7	11	13	17	19	23						
29	31	37	41	43	47	53	59	61						
67	71	73	79	83	89	97	101	103						
107	109	113	127	131	137	139	149	151						
157	163	167	173	179	181	191	193	197						
199	211	223	227	229	233	239	241	251						
257	263	269	271	277	281	283	293	307						
311	313	317	331	337	347	349	353	359						
367	373	379	383	389	397	401	409	419						
421	431	433	439	443	449	457	461	463						
467	479	487	491	499	503	509	521	523						
541	547	557	563	569	571	577	587	593						
599	601	607	613	617	619	631	641	643						
647	653	659	661	673	677	683	691	701						
709	719	727	733	739	743	751	757	761						
769	773	787	797	809	811	821	823	827						
829	839	853	857	859	863	877	881	883						
887	907	911	919	929	937	941	947	953						
967	971	977	983	991	997	1009	1013	1019						
1021	1031	1033	1039	1049	1051	1061	1063	1069						
1087	1091	1093	1097	1103	1109	1117	1123	1129						
1151	1153	1163	1171	1181	1187	1193	1201	1213						
1217	1223	1229	1231	1237	1249	1259	1277	1279						
1283	1289	1291	1297	1301	1303	1307	1319	1321						
1327	1361	1367	1373	1381	1399	1409	1423	1427						
1429	1433	1439	1447	1451	1453	1459	1471	1481						
1483	1487	1489	1493	1499	1511	1523	1531	1543						
1549	1553	1559	1567	1571	1579	1583	1597	1601						

FIGURE 1.11

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.



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:
Each sample counts as 0.01 seconds.
  %
        cumulative
                      self
                                           self
                                                     total
 time
         seconds
                    seconds
                               calls
                                        ms/call
                                                  ms/call
                                                            name
 99.93
          103.67
                    103.67
                               49999
                                          2.07
                                                            prime
                                                    2.07
  0.07
          103.74
                       0.07
                                                            main
                   Call graph
```

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

103.74 seconds

index	% time	self	children	called	name
					<spontaneous></spontaneous>
[1]	100.0	0.07	103.67		main [1]
		103.67	0.00	49999/49999	prime [2]

18

Chapter 1 • Profiling

[2]	99.9	103.67 103.67	49999/49999 49999	main [1] prime [2]
Index	by funct	tion name		
[1]] main		[2] prime	

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—**-fprofile-arcs** and **-ftest-coverage**, as shown in Figure 1.13. These options let you look at the program using gcov, as shown in Figure 1.14.

-	ell-Kon: Edit Viev	_	9	ings Hab								
					a -fpr	ofile-	arcs -	-ftest-	-coverage	-0	sample1 s	ample1.c
				./samp				1 0000	obtorage	0	bullpror b	unprozito
Z	3	5	7	11	13	17	19	23				
29	31	37	41	43	47	53	59	61				
67	71	73	79	83	89	97	101	103				
107	109	113	127	131	137	139	149	151				
157	163	167	173	179	181	191	193	197				
199	211	223	227	229	233	239	241	251				
257	263	269	271	277	281	283	293	307				
311	313	317	331	337	347	349	353	359				
367	373	379	383	389	397	401	409	419				
421	431	433	439	443	449	457	461	463				
467	479	487	491	499	503	509	521	523				
541	547	557	563	569	571	577	587	593				
599	601	607	613	617	619	631	641	643				
647	653	659	661	673	677	683	691	701				
709	719	727	733	739	743	751	757	761				
769	773	787	797	809	811	821	823	827				
829	839	853	857	859	863	877	881	883				
887	907	911	919	929	937	941	947	953				
967	971	977	983	991	997	1009	1013	1019				
1021	1031	1033	1039	1049	1051	1061	1063	1069				
1087	1091	1093	1097	1103	1109	1117	1123	1129				
1151	1153	1163	1171	1181	1187	1193	1201	1213				
1217	1223	1229	1231	1237	1249	1259	1277	1279				
1283	1289	1291	1297	1301	1303	1307	1319	1321				
1327	1361	1367	1373	1381	1399	1409	1423	1427				
1429	1433	1439	1447	1451	1453	1459	1471	1481				
1483	1487	1489	1493	1499	1511	1523	1531	1543				
1549	1553	1559	1567	1571	1579	1583	1597	1601				

FIGURE 1.13

Building sample1 with gcov options.

47777		w Bookm	nde Sat										
		Session Edit View Bookmanks Settings Hepp											
47857	47779	47791	47797	47807	47809	47819	47837	47843					
11031	47869	47881	47903	47911	47917	47933	47939	47947					
				47981									
				48119									
				48221									
				48337									
				48437									
				48523									
				48611									
				48731									
48767	48779	48781	48787	48799	48809	48817	48821	48823					
				48871									
				49003									
				49081									
				49171									
49207	49211	49223	49253	49261	49277	49279	49297	49307					
				49367									
				49451									
				49537									
49603	49613	49627	49633	49639	49663	49667	49669	49681					
				49741									
				49823									
			49921	49927	49937	49939	49943	49957					
49991													
			le1 #	gcov s	sample:	L.C							
File `													
		ted:100											
sample	1.c:cr	reating	(`sam)	ple1.c	.gcov'								

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.

Listing 1.7

Output from gcov for sample1

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

20

Chapter 1 • Profiling

```
int i;
        1:
               8:
                   int colcnt = 0;
        1:
               9:
    50000:
                   for (i=2; i <= 50000; i++)
              10:
    49999:
              11:
                      if (prime(i)) {
                        colcnt++;
     5133:
              12:
     5133:
              13:
                        if (colcnt%9 == 0) {
                   printf("%5d\n",i);
      570:
              14:
      570:
              15:
                    colcnt = 0;
              16:
        -:
        - :
              17:
                    else
     4563:
              18:
                      printf("%5d ", i);
              19:
        -:
                        putchar('\n');
        1:
              20:
        1:
              21:
                        return 0;
              22: }
        -:
        _ •
              23:
    49999:
              24: int prime (int num) {
                       /* check to see if the number is a prime?
        -:
              25:
                       */
    49999:
                       int i;
              26:
121337004:
              27:
                       for (i=2; i < num; i++)
                       if (num %i == 0)
121331871:
              28:
              29:
    44866:
                          return 0;
     5133:
              30:
                       return 1;
              31:
        - :
                       }
        -:
              32:
```

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 inListing 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 {
```

```
10
     int i;
11
     int colcnt = 0;
     for (i=2; i <= 50000; i++)
12
13
       if (prime(i)) {
          colcnt++;
14
15
         if (colcnt%9 == 0) {
            printf("%5d\n",i);
16
17
            colcnt = 0;
18
19
       else
20
            printf("%5d ", i);
21
22
         putchar('\n');
23
         return 0;
24 }
25
26 int prime (int num)
27
       /* check to see if the number is a prime? */
28
      int i;
       for (i=2; i <= faster(num); i++)</pre>
29
30
       if (num %i == 0)
31
          return 0;
32
       return 1;
33
    }
34
35 int faster (int num)
36 {
37
    return (int) sqrt( (float) num);
38
```

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:					
	sample counts	as 0.01	seconds.			
010	cumulative	self		self	total	
time	seconds	seconds	calls	us/call	us/call	name
52.68	1.48	1.48	1061109	1.39	1.39	faster
46.61	2.78	1.30	49999	26.10	55.60	prime
0.71	2.80	0.02			ma	in

Call graph

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

index	% time	self	children	called	name <spontaneous></spontaneous>
[1]	100.0	0.02 1.30	2.78 1.48	49999/49999	main [1] prime [2]
[2]	99.3	1.30 1.30 1.48	1.48 1.48 0.00	49999/49999 49999 1061109/1061109	main [1] prime [2] 9 faster [3]
[3]	52.7	1.48 1.48	0.00 0.00	1061109/1061109 1061109	prime [2] faster [3]

Index by function name

[3] faster	[1] main	[2]
prime		

$\begin{array}{cccccccccccccccccccccccccccccccccccc$	a state of the	iell - Kon	-			-			
$ sfb1: vusr vsrc vsample2 \\ 2 3 5 7 11 13 17 19 23 \\ 29 31 37 41 43 47 53 59 61 \\ 67 71 73 79 83 89 97 101 103 \\ 107 109 113 127 131 137 139 149 151 \\ 157 163 167 173 179 181 191 193 197 \\ 199 211 223 227 229 233 239 241 251 \\ 257 263 269 271 277 281 283 239 241 251 \\ 257 263 269 271 277 281 397 449 353 359 \\ 367 373 379 383 389 397 401 409 419 \\ 421 431 433 439 443 449 457 461 463 \\ 461 494 495 7461 463 \\ 467 479 487 491 499 503 509 521 523 \\ 541 547 557 563 569 571 577 587 593 \\ 599 601 607 613 617 619 631 641 643 \\ 647 653 659 661 673 677 683 691 701 \\ 709 719 727 733 739 743 751 757 761 \\ 769 773 787 797 809 811 821 823 827 \\ 829 839 853 857 859 863 877 881 883 \\ 887 907 911 919 929 937 941 947 953 \\ 967 971 977 983 991 997 1009 1013 1019 \\ 1021 1031 1033 1039 1049 1051 1061 1063 1069 \\ 1067 1091 1093 1097 1103 1109 1117 1123 1129 \\ 1051 1153 1153 1163 1171 1181 1187 1117 1123 1129 \\ 127 1223 1229 1231 1237 1249 1259 1277 1279 128 128 128 128 128 128 128 128 128 128$	Session	Edit Viev	v Bookma	arks Sett	ings Help	8			
$ \begin{array}{cccccccccccccccccccccccccccccccccccc$							ampleZ	sampl	e2.c -lm
$\begin{array}{cccccccccccccccccccccccccccccccccccc$							1010133	10121020	\$16,257.5
$ \begin{array}{cccccccccccccccccccccccccccccccccccc$									
$ \begin{array}{cccccccccccccccccccccccccccccccccccc$									
$\begin{array}{cccccccccccccccccccccccccccccccccccc$									
$\begin{array}{cccccccccccccccccccccccccccccccccccc$									
$\begin{array}{cccccccccccccccccccccccccccccccccccc$									
$\begin{array}{cccccccccccccccccccccccccccccccccccc$									
$\begin{array}{cccccccccccccccccccccccccccccccccccc$									
$\begin{array}{cccccccccccccccccccccccccccccccccccc$									
$\begin{array}{cccccccccccccccccccccccccccccccccccc$	367	373	379	383	389	397	401	409	419
541 547 557 563 569 571 577 587 593 599 601 607 613 617 619 631 641 643 647 653 659 661 673 677 683 691 701 709 719 727 733 739 743 751 757 761 769 773 787 797 809 811 821 823 827 839 853 857 859 863 877 881 883 867 907 911 919 929 937 941 947 953 967 971 977 983 991 997 1009 1013 1019 1087 1031 1039 1097 1051 1061 1063 1069 1087 1091 1093 1097 1031 1091 111123 1123 1087	421	431	433	439	443	449	457	461	463
$\begin{array}{cccccccccccccccccccccccccccccccccccc$	467	479	487	491	499	503	509	521	523
$ \begin{array}{ccccccccccccccccccccccccccccccc$	541	547	557	563	569	571	577	587	593
709 719 727 733 739 743 751 757 761 769 773 787 797 809 811 821 823 827 829 839 853 857 859 863 877 881 883 867 907 911 919 929 937 941 947 953 967 971 977 983 991 997 1009 1013 1019 1021 1031 1033 1039 1049 1051 1063 1069 1087 1091 1093 1097 103 109 1117 1123 1123 1151 1153 1163 1171 1181 1187 1193 1201 1213 1243 1229 1231 1237 1249 1259 1277 1279 1283 1289 1291 1297 1301 1303 1307 1321	599	601	607	613	617	619	631	641	643
$\begin{array}{cccccccccccccccccccccccccccccccccccc$	647	653	659	661	673	677	683	691	701
829 839 853 857 859 863 877 881 883 887 907 911 919 929 937 941 947 953 967 971 977 983 991 997 1009 1013 1019 1021 1031 1033 1039 1049 1051 1061 1065 1066 1087 1091 1093 1097 1103 1109 1117 1123 1129 1151 1153 1163 1171 1181 1187 1193 1201 1231 1283 1289 1291 1297 1301 1309 1319 1321 1327 1361 1367 1373 1381 1399 1409 1423 1427	709	719	727	733	739	743	751	757	761
887 907 911 919 929 937 941 947 953 967 971 977 983 991 997 1009 1013 1019 1021 1031 1033 1039 1049 1051 1063 1069 1087 1091 1093 1097 1103 1109 1117 1123 1123 1151 1153 1163 1171 1181 1187 1193 1201 1213 1217 1223 1229 1231 1237 1249 1259 1277 1279 1283 1289 1291 1297 1301 1307 1319 1321 1327 1361 1367 1373 1381 1399 1409 1423 1422	769	773	787	797	809	811	821	823	827
967 971 977 983 991 997 1009 1013 1019 1021 1031 1033 1039 1049 1051 1061 1063 1069 1087 1091 1093 1097 1103 1109 11123 1123 1151 1153 1163 1171 1181 1187 1193 1201 1213 1217 1223 1229 1231 1237 1249 1259 1277 1279 1283 1291 1297 1301 1303 1307 1321 1321 1327 1361 1373 1381 1399 1409 1423 1429	829	839	853	857	859	863	877	881	883
1021 1031 1033 1039 1049 1051 1061 1063 1069 1087 1091 1093 1097 1103 1109 1117 1123 1129 1151 1153 1163 1171 1181 1187 1193 1201 1213 1217 1229 1229 1231 1237 1249 1259 1277 1279 1283 1289 1291 1291 1301 1303 1307 1319 1321 1327 1361 1367 1373 1381 1399 1409 1423 1427	887	907	911	919	929	937	941	947	953
1087 1091 1093 1097 1103 1109 1117 1123 1129 1151 1153 1163 1171 1181 1187 1193 1201 1213 1217 1223 1229 1231 1237 1249 1259 1277 1279 1283 1289 1291 1297 1301 1303 1307 1319 1321 1327 1361 1367 1373 1381 1399 1409 1423 1427	967	971	977	983	991	997	1009	1013	1019
$\begin{array}{cccccccccccccccccccccccccccccccccccc$	1021	1031	1033	1039	1049	1051	1061	1063	1069
1151 1153 1163 1171 1181 1187 1193 1201 1213 1217 1223 1229 1231 1237 1249 1259 1277 1279 1283 1289 1291 1297 1303 1307 1319 1321 1327 1361 1367 1373 1381 1399 1409 1423 1427	1087	1091	1093	1097	1103	1109	1117	1123	1129
1217 1223 1229 1231 1237 1249 1259 1277 1279 1283 1289 1291 1297 1301 1303 1307 1319 1321 1327 1361 1367 1373 1381 1399 1409 1423 1427	1151				1181		1193	1201	1213
1283 1289 1291 1297 1301 1303 1307 1319 1321 1327 1361 1367 1373 1381 1399 1409 1423 1427									1279
1327 1361 1367 1373 1381 1399 1409 1423 1427									1321
									1427
									1481

FIGURE 1.15 Building and running sample2.

🕼 s	hell - Kor	ısole ⊲2>	9					
Session	Edit Vie	w Bookn	narks Sett	ings Help	r.			
47309	47317	47339	47351	47353	47363	47381	47387	47389
47407	47417	47419	47431	47441	47459	47491	47497	47501
47507	47513	47521	47527	47533	47543	47563	47569	47581
47591	47599	47609	47623	47629	47639	47653	47657	47659
47681	47699	47701	47711	47713	47717	47737	47741	47743
							47837	47843
		47881					47939	
47951		47969		47981		48023	48029	
48073		48091		48119			48157	
	48187			48221				48271
		48311		48337			48371	
							48473	
		48491		48523			48539	
48563				48611			48647	48649
48661		48677				48751		
							48821	
							48907	
							49031	
49037		49057	49069			49109	49117	49121
49123	49139	49157				49193	49199	49201
		49223					49297	
		49339		49367				49409
		49429					49477	
		49529					49559	
49603		49627			49663	49667	49669	
49697	49711					49757	49783	49787
		49807						49871
			49921	49927	49937	49939	49943	49957
	49993		.1-2 #					
STDT:	usr/s	rc/samj	piez #	gprof	-b sar	prez		

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.

						e-arcs	-ftes	t-coverage	e -o sa	mple2	sampl	e2.c -1
	usr/sr					100000	100000	10000				
2	3	5	7	11	13	17	19	23				
29	31	37	41	43	47	53	59	61				
67	71	73	79	83	89	97	101	103				
107	109	113	127	131	137	139	149	151				
157	163	167	173	179	181	191	193	197				
199	211	223	227	229	233	239	241	251				
257	263	269	271	277	281	283	293	307				
311	313	317	331	337	347	349	353	359				
367	373	379	383	389	397	401	409	419				
421	431	433	439	443	449	457	461	463				
467	479	487	491	499	503	509	521	523				
541	547	557	563	569	571	577	587	593				
599	601	607	613	617	619	631	641	643				
647	653	659	661	673	677	683	691	701				
709	719	727	733	739	743	751	757	761				
769	773	787	797	809	811	821	823	827				
829	839	853	857	859	863	877	881	883				
887	907	911	919	929	937	941	947	953				
967	971 1031	977	983	991	997	1009	1013	1019				
1021 1087	1031	1033 1093	1039 1097	1049 1103	1051 1109	1061 1117	1063 1123	1069 1129				
1151	1153	1163	1171	1181	1187	1193	1201	1213				
1217	1223	1229	1231	1237	1249	1259	1201	1213				
1283	1223	1225	1231	1301	1303	1307	1319	1321				
1327	1361	1367	1373	1381	1399	1409	1423	1427				
1429	1433	1439	1447	1451	1453	1405	1425	1427				
1423	1433	1439	1493	1451	1455	1523	1531	1543				
1549	1553	1559	1567	1571	1579	1525	1551	1601				

FIGURE 1.17

Building sample2 with gcov and running sample2.

😱 s	ihell - Kor	1sole <2>	9					
Session	Edit Vie	w Bookn	narks Sett	tings Help				
	47779							
	47869							
	47963							
	48079							
	48187							
48281	48299	48311	48313	48337	48341	48353	48371	48383
48397	48407	48409	48413	48437	48449	48463	48473	48479
48481	48487	48491	48497	48523	48527	48533	48539	48541
	48571							48649
48661	48673	48677	48679	48731	48733	48751	48757	48761
48767	48779	48781	48787	48799	48809	48817	48821	48823
48847	48857	48859	48869	48871	48883	48889	48907	48947
48953	48973	48989	48991	49003	49009	49019	49031	49033
49037	49043	49057	49069	49081	49103	49109	49117	49121
49123	49139	49157	49169	49171	49177	49193	49199	49201
	49211							
49331	49333	49339	49363	49367	49369	49391	49393	49409
49411	49417	49429	49433	49451	49459	49463	49477	49481
49499	49523	49529	49531	49537	49547	49549	49559	49597
49603	49613	49627	49633	49639	49663	49667	49669	49681
49697	49711	49727	49739	49741	49747	49757	49783	49787
49789	49801	49807	49811	49823	49831	49843	49853	49871
49877	49891	49919	49921	49927	49937	49939	49943	49957
49991	49993	49999						
sfb1:	/usr/si	rc/sam	le2 #	gcov :	samplea	2.c		
File	sample	eZ.c'						
Lines	execu	ted:100	0.00%	of 20				
sample	e2.c:ci	reating	sam)	ple2.c	.gcov'			
sfb1:	/usr/s	rc/samj	le2 #					

FIGURE 1.18

Running sample2 and getting gcov output.

Listing 1.10 shows gcov output for the sample2 program.

Listing 1.10

```
Output of sample2.c.gcov
```

```
0:Source:sample2.c
    -:
           0:Graph:sample2.bbg
    - :
           0:Data:sample2.da
    -:
           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:{
    1:
         10:
    1:
                 int i;
                 int colcnt = 0;
for (i=2; i <= 50000; i++)</pre>
    1:
          11:
50000:
          12:
```

```
49999:
                      if (prime(i)) {
            13:
   5133:
            14:
                        colcnt++;
                       if (colcnt%9 == 0) {
   5133:
            15:
            16:printf("%5d\n",i);
    570:
    570:
            17:colcnt = 0;
       - :
            18:
            19:
                   else
      -:
   4563:
                      printf("%5d ", i);
            20:
            21:
      - :
      1:
            22:
                       putchar(' n');
      1:
            23:
                       return 0;
            24: }
      - :
            25:
      - :
  49999:
            26: int prime (int num) {
                      /* check to see if the number is a
      - :
            27:
                     prime? */
  49999:
            28:
                     int i;
1061109:
                      for (i=2; i <= faster(num); i++)</pre>
            29:
            30:
1055976:
                      if (num %i == 0)
  44866:
            31:
                        return 0;
   5133:
            32:
                      return 1;
            33:
      -:
      - :
            34:
            35: int faster (int num)
      - :
1061109:
            36: {
1061109:
            37:
                  return (int) sqrt( (float) num);
            38: }
      - :
      - :
            39:
```

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

```
1
   #include <iostream>
2
   void function1() {
3
       for(int i=0;i<1000000;i++);</pre>
4
5
   }
6
7
   void function2() {
8
       function1();
9
       for (int i=0;i<2000000;i++);</pre>
10
   }
11
12 void function3() {
13
       function1();
14
       function2();
       for (int i=0;i<3000000;i++);</pre>
15
16
           function1();
17 }
18
19 int main(){
20
       for(int i=0;i<10;i++)</pre>
21
       function1();
22
23
       for (int i=0;i<5000000;i++);</pre>
24
25
       for(int i=0;i<10;i++)</pre>
26
           function2();
27
           for(int i=0; i<13;i++);</pre>
28
29
               function3();
30
               function2();
31
               function1();
32
33
```

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.

Shell-Konsole 2 Session Edit View Bookmarks Settings Heb sfb1:/usr/src/sample3 # g++ sample3.cpp -pg -o sample3 sfb1:/usr/src/sample3 # ./sample3 sfb1:/usr/src/sample3 # gprof ./sample3 > sample3.gprof sfb1:/usr/src/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.

Flat pr	ofile:					
Each sa	ample counts a	as 0.01 sec	onds.			
00	cumulative	self		self	total	
time	seconds	seconds	calls	ms/call	ms/call	name
43.36	4.21	4.21	12	0.35	0.52	function2()
42.84	8.37	4.16	25	0.17	0.17	function1()
8.65	9.21	0.84				main
5.15	9.71	0.50	1	0.50	1.35	function3()
0.00	9.71	0.00	1	0.00	0.00	global constructors
						keyed to function1()
0.00	9.71	0.00	1	0.00	0.00	
stati	.c_initializat	ion_and_de	structio	on_0(int,	int)	

The sample3.gprof should look similar to this:

Field	Description
% time	The percentage of the program's total running time used by this function.
cumulative seconds	A running sum of the number of seconds accounted for by this function and those listed above it.
self seconds	The number of seconds accounted for by this function alone. This is the major sort for this listing.
calls	The number of times this function was invoked if this func- tion is profiled; otherwise, it is blank.
self ms/call	The average number of milliseconds spent in this function per call if this function is profiled; otherwise, it is blank.
total ms/call	The average number of milliseconds spent in this function and its descendents per call if this function is profiled; oth- erwise, it is blank.
name	The function's name. This is the minor sort for this listing. The index shows the location of the function in the gprof listing. If the index is in parentheses, it shows where it would appear in the gprof listing if it were to be printed.

Call gr	aph (ex	planatio	on follows)			
granula	arity: e	ach sam <u>p</u>	ole hit covers 4	4 byte(s)	for 0.10% of	9.71 seconds
index	% time	self	children	called	name	
					< 5	pontaneous>
[1]	100.0	0.84	8.87		main	[1]
		3.86	1.83	11/12		function2() [2]

		1.83	0.00	11/25	function1() [3]
			0.85		function3() [4]
			0 17		function3() [4]
				11/12	main [1]
[2]			2.00		function2() [2]
		2.00	0.00	12/25	
			0.00		function3() [4]
		1.83	0.00	11/25	main [1]
		2.00	0.00	12/25	function2() [2]
			0.00		function1() [3]
			0.85		main [1]
[4]	13.9	0.50	0.85	1	function3() [4]
		0.35	0.17	1/12	function2() [2]
			0.00	2/25	function1() [3]
			0.00		do_global_ctors_aux [13]
[11]	0.0	0.00	0.00	1	global constructors keyed to
functi	on1() [11]			
		0.00	0.00	1/1	
stat	ic_init	ializati	.on_and_de	estruction_0(int,	int) [12]
		0.00	0.00	1/1	global constructors keyed to
functi	on1() [11]			
[12]		0.0	0.00	0.00	1
				estruction_0(int,	int) [12]

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:

Field Description

index A unique number given to each element of the table. Index numbers are sorted numerically. The index number is printed next to every function name so that it is easier to look up the function in the table.
% time 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%.

Field	Description
self	The total amount of time spent in this function.
children	The total amount of time propagated into this function by its children.
called	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.
name	The name of the current function. The index number is printed after it. If the function is a member of a cycle, the cycle number is printed between the function's name and the index number.

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

Field	Description
self	The amount of time that was propagated directly from the function into this parent.
children	The amount of time that was propagated from the function's chil- dren into this parent.
called	The number of times this parent called the function and the total number of times the function was called. Recursive calls to the func- tion 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:

Field	Description
self	The amount of time that was propagated directly from the child into the function.
children	The amount of time that was propagated from the child's children to the function.

kprof

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
[11] global constructors keyed to function1() [3] function1() [4] function3()
[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 nstalled 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.

🕼 Shell - Konsole <3> 🧕	-		X	1
Session Edit View Bookmarks Settings Help				
sfb1:/usr/src/graphviz # tar zxuf graphviz-1.12.tar.gz && cd graphviz-1.12 && ./configure && make install	åå	ma)	ке	

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

kprof

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



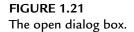


Figure 1.22 shows the flat profile view of the sample3 program. This screen shot also shows that function1 is called by function2, function3, and main.

<u>File Tools S</u> ettings	Нер							
<u>Flat Profile</u> <u>Hierar</u>	hical Profile	Object Profile	Graph	View Metho	od View			
Filter:								
Function/Method 🗢	Count	Total (s)	%	Self (s)	Total ms/call	Self ms/call	-	
static_initialization	đ	9.700	0.000	0.000	0.000	0.000		
function1	25	8.360	42.780	4.150	0.170	C	alied By:	
function2	12	4.210	43.400	4.210	0.520		unction3()	
function3	1	9.700	5.150	0.500	1.350		main	
global constructors k	1	9.700	0.000	0.000	0.000			
main	0	9.200	8.660	0.840	0.000		function2()	

FIGURE 1.22 The flat profile view.

Eile <u>T</u> ools <u>S</u> ettings <u>H</u> elp						
Elat Profile Hierarchical Profi	le Object Pr	ofile G <u>r</u> aph Vi	ew <u>M</u> et	hod View		
Function/Method 👻	Count	Total (s)	%	Self (s)	Total ms/call	Self ms/call
Hierarchy						
	1	9.700	0.000	0.000	0.000	0.000
function1	25	8.360	42.780	4.150	0.170	0.170
- function2	12	4.210	43.400	4.210	0.520	0.350
I function1	25	8.360	42.780	4.150	0.170	0.170
function3	1	9.700	5.150	0.500	1.350	0.500
± function2	12	4.210	43.400	4.210	0.520	0.350
-global constructo	1	9.700	0.000	0.000	0.000	0.000
^I static_initia	1	9.700	0.000	0.000	0.000	0.000
l main	0	9.200	8.660	0.840	0.000	0.000
i. ⊢ function2	12	4.210	43.400	4.210	0.520	0.350
I-function1	25	8.360	42.780	4.150	0.170	0.170
function3	1	9.700	5.150	0.500	1.350	0.500

Figure 1.23 shows the hierarchical profile view of the sample3 program.

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

📕 sample3.prof1 - KProf 🗢 🧕	
<u>File Tools S</u> ettings <u>H</u> ep	
Eat Profile Hierarchical Profile Object Profile Graph View Method View	
::function2 ::function1 ::function1	

FIGURE 1.24 The graph view.

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.

Web Resources for Profiling

http://www.gnu.org/software/binutils/manual/	
gprof-2.9.1/gprof.html Documentation for gprof	
http://kprof.sourceforge.net/ kprof home page	
http://www.research.att.com/sw/tools/graphviz/ download.html graphviz home page	
http://samba.org/ftp/tridge/dbench/ dbench download page	