programming pearls

by Jon Bentley

PROFILERS

A physician doesn't feel dressed without a stethoscope, and a true electrical engineer is never far from an oscilloscope. Both professionals know that they need tools for studying the objects they manipulate.

What tools do you use to study your programs? Sophisticated analysis systems are now widely available, ranging from interactive debuggers to systems for program animation. But just as CAT scanners will never replace stethoscopes, complex software will never replace the simplest tool we programmers have for looking inside our programs: profilers.

This column starts by using two kinds of profilers to speed up a tiny program. The next sections sketch various uses of profilers and a profiler for a nonprocedural language. The final section discusses building profilers.

Computing Primes

In this section we'll start with a straightforward program for computing prime numbers and use profilers to decrease its run time. Although we'll pretend that our goal is to make a program faster, keep in mind that the real purpose of this section is to illustrate profilers.

Program P1 is a C program[1] to print all primes less than 1000, in order. The `prime` function returns 1 (true) if its integer argument n is prime and 0 otherwise; it tests all integers between 2 and $n - 1$ to see whether they divide n. The `main` procedure uses that routine to examine the integers 2 through 1000, in order, and prints primes as they are found.

I wrote Program P1 as I would write any program, and then compiled it with a profiling option. After the program executed, a single command generated the listing shown. (I have made minor formatting changes to a few of the outputs in this column.) The numbers to the left of each line tell how many times the line was executed. They show, for instance, that `main` was called once, it tested 999 integers, and found 168 primes. Function `prime` was called 999 times; it returned one 168 times and returned zero the other 831 times (a reassuring quick check: 168 + 831 = 999). It tested a total of 78,022 potential factors, or about 78 factors for each number examined for primality.

[1] A few C-isms: The statement `i++` increments the integer `i`. The loop `for (i=2; i<= n; i++)` iterates `i` from 2 to `n`. The statement `a=b` assigns the value of `b` to `a`; the expression `a==b` is true if the two variables are equal. The expression `a%b` is the remainder when `a` is divided by `b`, so `10%7` is 3. The `printf` routine provides a formatted print statement.


```
        prime(n)
        int n;
        {   int i;
999         for (i = 2; i < n; i++)
78022           if (n % i == 0)
831                 return 0;
168         return 1;
        }

        main()
        {   int i, n;
1           n = 1000;
1           for (i = 2; i <= n; i++)
999             if (prime(i))
168                 printf("%d\n", i);
        }
```

PROGRAM P1. Compute Primes Less Than 1000

Program P1 is correct but slow: on a VAX-11/750® it computes all primes less than 1000 in a couple of seconds, but requires three minutes to find those less than 10,000. The profile shows that most of the time is spent testing factors. Program P2 therefore considers as potential factors of n only those integers up to $\sqrt{n}$. (The integer function `root` converts its integer argument to

VAX is a trademark of Digital Equipment Corporation.

```
        root(n)
        int n;
5456    { return (int) sqrt((float) n); }

        prime(n)
        int n;
        {   int i;
999         for (i = 2; i <= root(n); i++)
5288            if (n % i == 0)
831                 return 0;
168         return 1;
        }

        main()
        {   int i, n;
1           n = 1000;
1           for (i = 2; i <= n; i++)
999             if (prime(i))
168                 printf("%d\n", i);
        }
```

PROGRAM P2. Stop Test at Square Root

floating point, calls the library function `sqrt`, then converts the floating-point answer back to an integer.)

The change was evidently effective: the line counts in Program P2 show that only 5288 factors were tested (a factor of 14 fewer than in Program P1). A total of 5456 calls were made to `root`: divisibility was tested 5288 times, and the loop terminated 168 times because `i` exceeded `root(n)`. But even though the counts are greatly reduced, Program P2 runs in 5.8 seconds, while P1 runs in just 2.4 seconds (Table II contains more details on run times). What gives?

So far we have seen only *line-count* profiles; a *procedure-time* profile gives fewer details about the flow of control but more insight into CPU time:

```
%time  cumsecs  #call  ms/call  name
 82.7     4.77                 _sqrt
  4.5     5.03    999     0.26 _prime
  4.3     5.28   5456     0.05 _root
  2.6     5.43                 _frexp
  1.4     5.51                 __doprnt
  1.2     5.57                 _write
  0.9     5.63                 mcount
  0.6     5.66                 _creat
  0.6     5.69                 _printf
  0.4     5.72      1    25.00 _main
  0.3     5.73                 _close
  0.3     5.75                 _exit
  0.3     5.77                 _isatty
```

The procedures are listed in decreasing order of run time; the time is displayed both in (cumulative) seconds and as a percent of the total. The three procedures in the source program (`main`, `prime`, and `root`) were compiled to record the number of times they were called (it is encouraging to see the same counts once again); the other procedures are (unprofiled) library routines that perform miscellaneous input/output and housekeeping functions. The fourth column tells the average number of milliseconds per call for all functions with statement counts.

The procedure-time profile shows that `sqrt` uses the lion's share of CPU time.[2] It was called 5456 times, once for each test of the `for` loop. Program P3 calls that expensive routine just once per call of `prime` by moving the call out of the loop. Program P3 is about 4 times faster than P2 when $n = 1000$ and over 10 times faster when $n = 100{,}000$. At $n = 100{,}000$, the procedure-time profile shows that `sqrt` takes 88 percent of the time of P2, but just 48 percent of the time of P3; it is a lot better, but still the cycle hog.

[2] The December 1986 column discussed faster ways of computing square roots. For computing primes and for many other programs, though, the way to efficiency is not a faster square root routine but avoiding roots entirely.

```
        prime(n)
        int n;
        {   int i, bound;
999         bound = root(n);
999         for (i = 2; i <= bound; i++)
5288            if (n % i == 0)
831                 return 0;
168         return 1;
        }
```

PROGRAM P3 (fragment). Compute Root Once

Program P4 incorporates two additional speedups. First, it avoids almost three-quarters of the square roots by special checks for divisibility by 2, 3, and 5. (The statement counts show that divisibility by two identifies roughly half the inputs as composites, divisibility by three gets a third of the remainder, and divisibility by five catches a fifth of those numbers still surviving.) Second, it avoids about half the remaining divisibility tests by considering only odd numbers as potential factors. It is faster than P3 by a factor of about three, but it is also buggier than its predecessor. Can you spot the problem by examining the statement counts?

```
        root(n)
        int n;
265     { return (int) sqrt((float) n); }

        prime(n)
        int n;
        {   int i, bound;
999         if (n % 2 == 0)
500             return 0;
499         if (n % 3 == 0)
167             return 0;
332         if (n % 5 == 0)
67              return 0;
265         bound = root(n);
265         for (i = 7; i <= bound; i = i+2)
1530            if (n % i == 0)
100                 return 0;
165         return 1;
        }

        main()
        {   int i, n;
1           n = 1000;
1           for (i = 2; i <= n; i++)
999             if (prime(i))
165                 printf("%d\n", i);
        }
```

PROGRAM P4 (buggy). Special Case for 2, 3, and 5

The previous programs found 168 primes, while P4 found just 165. Where are the three missing primes? Sure enough, I treated three numbers as special cases, and introduced one bug with each: `prime` reports that 2 is not a prime because it is divisible by 2, and similarly botches 3 and 5. The tests are correctly written as

```
if (n % 2 == 0)
    return (n == 2);
```

If n is divisible by 2, it returns 1 if n is 2, and 0 otherwise. The procedure-time profiles of Program P4 are summarized in Table I.

Program P5 is faster than P4 and also (and more importantly) correct. It replaces the expensive square root operation with a multiplication, as shown in this fragment:

TABLE I. The Procedure-Time Profiles of Program P4

	Percent of time in		
n	sqrt	prime	other
1000	45	19	36
10,000	39	42	19
100,000	31	56	13

```
265         for (i = 7; i*i <= n; i = i+2)
1530            if (n % i == 0)
100                 return 0;
165         return 1;
```

It also incorporates the correct tests for divisibility by 2, 3, and 5. The total speedup is about twenty percent over P5.

The final program tests for divisibility by only integers that have previously been identified as primes; Program P6 is on page 591, coded in the Awk language. The procedure-time profile of the C implementation shows that at $n = 1000$, 49 percent of the run time is in `prime` and `main` (the rest is in input/output), while at $n = 100{,}000$, 88 percent of the run time is spent in those two procedures.

Table II summarizes the programs we've seen. It includes two other programs as benchmarks: Program Q1 computes primes using the Sieve of Eratothenes sketched in Problem 2. Problem Q2 measures input/output cost. For $n = 1000$, it prints the integers 1, 2, . . . , 168; for general n, it prints the appropriate number of integers.

TABLE II. Summary of Primality Programs

	Run time in seconds		
Program	$n =$ 1000	$n =$ 10,000	$n =$ 100,000
P1. Simple version	2.4	169	?
P2. Test only up to root	5.8	124	2850
P3. Compute root once	1.4	15	192
P4. Special case 2, 3, 5	0.5	5.7	78
P5. Replace root by *	0.3	3.5	64
P6. Test only primes	0.3	3.3	47
Q1. Simple sieve	0.2	1.2	10.4
Q2. Print 1.. $P(n)$	0.1	0.7	5.3

This exercise has concentrated on one use of profiling: as you're tuning the performance of a single subroutine or function, profilers can show you where the run time is spent. In his literate program in this *Communications* (see "Further Reading"), Hanson uses a profiler to tune a program that is a couple of pages long.

Using Profilers

Profilers are handy for small programs, and indispensable for working on large software. Brian Kernighan used a line-count profiler on a 4000-line program that had been widely used for several years. Scanning the 75-page listing showed that most counts were in the hundreds and thousands, while a few were in the tens of thousands. An obscure piece of initialization code, though, had a count near a million. Kernighan changed a few parts of the six-line loop, and thereby doubled the speed of the program. He never would have guessed the hot spot of the program, but the profiler led him right to it.

Kernighan's experience is quite typical. In a paper cited under "Further Reading," Don Knuth presents an empirical study of many aspects of Fortran programs, including their profiles. That paper is the source of the often quoted (and more often misquoted) statement that "less than 4 per cent of a program generally accounts for more than half of its running time." Numerous studies on many languages and systems have shown that for most programs that aren't input/output bound, a large fraction of the run time is spent in a small fraction of the code. This pattern is the basis of testimonials like the following:

> In his paper, Knuth describes how the line-count profiler was applied to itself. The profile showed that half of the run time was spent in two loops; changing a few lines of code doubled the speed of the profiler in less than an hour's work.
>
> The February 1984 column describes in detail how Chris Van Wyk sped up a 3000-line program by 55 percent in a few hours. The profile showed that 70 percent of the run time was spent in the storage allocator; 20 lines of code fixed the problem.
>
> In 1984 Tom Szymanski of Bell Labs put an (intended) speedup into a large system, only to see it run ten percent slower. He started to remove the modification, but then enabled a few more profiling options to see why it had failed. He found that the space had increased by a factor of twenty; line counts showed that storage was allocated many more times than it was freed. A single instruction fixed that bug. The correct implementation sped up the system by a factor of two.
>
> Profiling showed that half of an operating system's time was spent in a loop of just a few instructions. Rewriting it in microcode made it an order of magnitude faster but didn't change the system's throughput: the performance group had optimized the system's idle loop! (I have heard several versions of this charming story, but I can't find a precise reference. I would appreciate more any information from readers; anonymity is guaranteed.)

These experiences raise a problem that we only glimpsed in the last section: on what inputs should one profile a program? The primality programs had the single input n, which nonetheless strongly affects the time profile (input/output dominates for small n, while computation dominates for large n). Some programs have profiles quite insensitive to the input data; I'd guess that most payroll programs have pretty consistent profiles, at least from February to November. The profiles of other programs vary dramatically with the input; haven't you ever suspected that your system was tuned to run like the wind on the manufacturer's benchmark, while it crawls like a snail on your important jobs? Take care in selecting your input mix.

Profilers are useful for tasks beyond performance. In the primality exercise, for instance, they pointed out a bug in Program P4. Line counts are invaluable for evaluating test coverage; zero counts, for instance, show

untested code. Dick Sites of Digital Equipment Corporation describes other uses of profiling: "(1) Deciding what microcode to put on chip in a two-level microstore implementation. (2) A friend at Bell Northern Research implemented statement counts one weekend in a real-time phone switching software system with multiple asynchronous tasks. By looking at the unusual counts, he found six bugs in the field-installed code, all of which involved interactions between different tasks. One of the six they had been trying (unsuccessfully) to track down via conventional debugging techniques, and the others were not yet identified as problems (i.e., they may have occurred, but nobody could attribute the error syndrome to a specific software bug)."

A Specialized Profiler

The principles of profiling we've seen so far apply to languages ranging from assemblers and Fortran to Ada. But many programmers now work in more powerful notations; how should we profile a computation in Lisp or APL, or in a network or database language?

We'll take UNIX® pipelines as an example of a more interesting computational model. Pipelines are a sequence of filters; data is transformed as it flows through each filter. In the June 1986 column, for example, Doug McIlroy described this pipeline for printing the 25 most common words in a document, in decreasing frequency.[3]

```
cat $* |
tr -cs A-Za-z '\012' |
tr A-Z a-z |
sort |
uniq -c |
sort -rn |
sed 25q
```

I profiled McIlroy's pipeline as it found the 25 most common words in thirteen Programming Pearls columns. The first six lines in the output were:

```
3463 the
1855 a
1556 of
1374 to
1166 in
1104 and
  ...
```

The profile of the computation is presented in Table III. The left parts describe the data at each stage: the number of lines, words, and characters. The right parts describe the filters between the data stages: user, system, and real times (all in seconds) are followed by the command itself.

This profile provides much information of interest to programmers. The pipeline is fast; 3.5 minutes of real time for 150 book pages is moving right along on a VAX-11/750. The first sort consumes 57 percent of the run time of the pipeline; that finely tuned utility will be hard to speed up further. The second sort takes only 14 percent of the pipeline's time, but is ripe for tuning.[4] The profile also identifies a little bug lurking in the pipeline; UNIX gurus may enjoy finding where the null line was introduced.

The profile also teaches us about the words in the document. There were 57,651 total words, but only 4731 distinct words. After the first transliteration program, there are 4.3 letters per word. The output showed that the most common word was "the"; it accounts for 6 percent of the words in the files. The six most common words account for 18 percent of the words in the file; special-casing the 100 most common words in English might be an effective way to speed up a word-count program. Try finding other interesting factoids in the counts.

Like many UNIX users, I had previously profiled pipelines by hand, using the word count (`wc`) command to measure files and the `time` command to measure processes. The "pipeline profiler" that produced Table III automates that task: its takes as input the names of a pipeline and several input files, and produces the profile as output. Two hours and fifty lines of code sufficed to build the profiler. The next section elaborates on this topic.

Building Profilers

Building a real profiler is hard work; a colleague recently spent several weeks of effort spread over several

UNIX is a registered trademark of AT&T Bell Laboratories.

[3] This version is changed just slightly from McIlroy's. The seven filters have the following tasks: (1) Concatenate all input files. (2) Make one-word lines by transliterating the complement (-c) of the alphabet into newlines (ASCII octal 12) and squeezing out (-s) multiple newlines. (3) Transliterate uppercase to lowercase. (4) Sort to bring identical words together. (5) Replace each run of duplicate words with a single representative and its count (-c). (6) Sort in reverse (-r) numeric (-n) order. (7) Pass through a stream editor; quit (q) after printing 25 lines.

[4] The second sort takes 25 percent of the run time of the first sort on just 8 percent of the number of input lines—the numeric flag is very expensive. When I profiled this pipeline on a single column, the second sort was almost as expensive as the first; the profile is quite sensitive to the input data.

TABLE III. Profile of McIlroy's Pipeline

lines	words	chars	times	
10717	59701	342233	14.4u 2.3s 18r	tr -cs A-Za-z \012
57652	57651	304894	11.9u 2.2s 15r	tr A-Z a-z
57652	57651	304894	104.9u 7.5s 123r	sort
57652	57651	304894	24.5u 1.6s 27r	uniq -c
4731	9461	61830	27.0u 1.6s 31r	sort -rn
4731	9461	61830	0.0u 0.2s 0r	sed 25q
25	50	209		

months building one of production quality. This section describes how a simple version can be built much more easily.

Dick Sites claimed that his friend "implemented statement counts one weekend." I found that pretty hard to believe, so I decided I'd try to build a profiler for Awk, an unprofiled language that I use frequently. A couple of hours later, my profiler produced the output shown in Program P6. The number in angle brackets after a left curly brace tells how many times the block was executed; fortunately, the counts match those produced by the C line counter.

```
function prime(n,  i) { <<<998>>>
  for (i=0; x[i]*x[i]<=n; i++) { <<<2801>>>
      if (n % x[i] == 0) { <<<831>>>
          return 0
      }
  }
  { <<<167>>> }
  x[xc++] = n
  return 1
}

BEGIN { <<<1>>>
  n = 1000
  x[0] = 2; xc = 1
  print 2
  for (i = 3; i <= n; i++) { <<<998>>>
      if (prime(i)) { <<<167>>>
          print i
      }
  }
  exit
}
```

PROGRAM P6. Check Only Primes, Coded in Awk

My profiler consists of two five-line Awk programs. The first program reads the Awk source program and writes a new program in which a distinct counter is incremented at the start of each block; all counters are written to a file at the end of execution. When the resulting program runs, it produces a file of counters. The second program reads those counters and merges them back into the source text. The profiled program is about 25 percent slower than the original, and not all Awk programs are handled correctly (I had to make one-line changes to profile several programs). But for all its flaws, a couple of hours was a small investment to get a prototype profiler up and running. (Details on a similar Awk profiler can be found in Section 7.2 of *The Awk Programming Language* by Aho, Kernighan, and Weinberger, published by Addison-Wesley in 1987.)

Quick profilers are more commonly written than written about; here are a few examples:

- In the August 1983 *BYTE*, Leas and Wintz describe a profiler implemented as a 20-line program in 6800 assembly language.
- Howard Trickey of Bell Labs implemented function counts in Lisp in an hour by changing `defun` to increment a counter as each function is entered.
- In 1978, Rob Pike implemented a time profiler in 20 lines of Fortran. After `CALL PROFIL(10)`, subsequent CPU time is charged to counter 10.

On these and many other systems, it is possible to write a profiler in an evening. The resulting profiler could easily save you more than an evening's work the first time you use it.

Principles

This column has only scratched the surface of profiling. I've stuck to the basics, and ignored exotic ways of collecting data (such as hardware monitors) and exotic displays (such as animation systems). The message of the column is equally basic:

Use a profiler. July is Profiler Month; please profile at least one piece of code in the next few weeks. Remember, a programmer never stands as tall as when stooping to help a small program.

Build a profiler. If you don't have a profiler handy, fake it. Most systems provide basic profiling operations; programmers who had to read console lights 25 years ago can get the same information today from a graphics window on a personal workstation. A little program is often sufficient to package a system's instrumentation operations into a convenient tool.

Problems

1. Suppose the array X[1 .. 1000] is sprinkled with random real numbers. This routine computes the minimum and maximum values:

```
Max := Min := X[1]
for I := 2 to 1000 do
    if X[I] > Max then Max := X[I]
    if X[I] < Min then Min := X[I]
```

   Mr. B. C. Dull observed that if an element is a new minimum, then it cannot be a maximum. He therefore rewrote the two comparisons as

```
if      X[I] > Max then Max := X[I]
else if X[I] < Min then Min := X[I]
```

   How many comparisons will this save, on the average? First guess the answer, then implement and profile the program to find out. How was your guess?

2. The following problems deal with computing prime numbers:

   a. Programs P1 through P6 squeezed two orders of magnitude out of the run time. Can you wring any more performance out of this approach to the problem?

b. Implement a simple Sieve of Eratosthenes for computing all primes less than n. The primary data structure for the program is an array of n bits, all initially true. As each prime is discovered, all of its multiples in the array are set to false. The next prime is the next true bit in the array.

c. What is the run time as a function of n of the sieve in part b? Find an algorithm with running time of $O(n)$.

d. Given a very large integer (say, several hundred bits), how would you test it for primality?

3. A simple statement-count profiler increments a counter at each statement. Describe how to decrease memory and run time by making do with fewer counters. (I once used a Pascal system in which profiling a program slowed it down by a factor of 100; the line-count profiler I used in this column slows down a program by a few percent.)

4. A simple procedure-time profiler estimates the time spent in each procedure by observing the program counter at a regular interval (60 times a second on my system). This information tells the time spent in each part of the program text, but it does not tell which procedures called the time hogs. In his program cited under "Further Reading," Hanson uses a profiler that gives the cost of each function and its dynamic descendants. Show how to gather more information from the run-time stack to allocate time among callers and callees. Given this data, how can you display it in a useful form?

5. Precise numbers are useful for interpreting profiles of a program on a single data set. When there is a lot of data, though, the volume of digits can hide the message in the numbers. How would you display the line-count profile of a program (or a pipeline) on 100 data sets? Consider especially graphical displays of the data.

6. Program P6 is a correct program that is hard to prove correct. What is the problem, and how can you solve it?

Further Reading

Don Knuth's "Empirical Study of FORTRAN Programs" appeared in *Software—Practice and Experience 1* in 1971 (pp. 105–133). Section 3 on "dynamic statistics" discusses both line-count and procedure-time profilers, and the statistics they were used to gather. Section 4 tunes seventeen critical inner loops, for speedup factors ranging from 1.5 to 13.1. I have read this classic paper at least once a year for the past decade, and it gets better every time; I can't recommend it too highly.

A new section on "Literate Programming" debuts in this issue of *Communications*. I expect that most readers of this column will enjoy most literate programs and their reviews. This month's offering is particularly juicy: David Hanson uses a profiler to produce a rock-solid and efficient program for computing common words in a file, and John Gilbert eloquently and insightfully compares Hanson's approach to Knuth's and McIlroy's solutions in the June 1986 "Programming Pearls."

Solutions to June's Problems

2. In Section 3.9 of their *Unix Programming Environment* (Prentice-Hall, 1984), Kernighan and Pike present a program named `bundle`. The command

```
bundle file1 file2 file3
```

produces a UNIX shell file. When executed, it writes copies of all the files in the bundle.

3. The problem asked for a self-reproducing program: one that prints exactly its source text when executed. Such a program exists in any universal model of computation; the proof of that fact uses the Recursion Theorem and the *s-m-n* Theorem of recursive function theory. Hackers have long delighted in writing self-reproducing programs in real languages; Fortran and C seem to be particularly popular. (I recently marveled at a 356-character self-reproducing *palindromic* C program written by Dan Hoey.) The problem becomes much easier if you allow the program to self-reproduce on the error output. If you start with a small file (say, the single word "junk"), and then iteratively feed the error messages produced as input back to the compiler, the process usually converges quickly.

4. The UNIX file system does not classify files by type, but several programs use the contents of files as an implicit self-description. The `file` command, for instance, examines a file and guesses whether the comments represent ASCII text, program text, shell commands, etc. In their book cited above, Kernighan and Pike present a program called `doctype` that reads a text file and deduces what language processors need to be run on it.

5. Examples of name-value pairs include PL/1's `GET DATA` statement and Fortran's `NAMELIST` input. Arrays and functions both map names to values.

7. A general principle states that the output of a program should be suitable for input to the program. This is especially important for programs that are pipes and in window systems that allow output to be selected and entered as input with a few mouse motions.

For Correspondence: Jon Bentley, AT&T Bell Laboratories, Room 2C-317, 600 Mountain Avenue, Murray Hill, NJ 07974.