3.2: Timing and Profiling

Introduction

Perhaps getting your code to produce the right answers is enough. After all, if you only plan to use the program once in a while, or if it only takes a few minutes to run, execution time isn’t going to matter that much. But it might not always be that way. Typically, people start taking interest in the runtime of their programs for two reasons:

• The workload has increased.
• They are considering a new machine.

It’s clear why you might care about the performance of your program if the workload increases. Trying to cram 25 hours of computing time into a 24-hour day is an administrative nightmare. But why should people who are considering a new machine care about the runtime? After all, the new machine is presumably faster than the old one, so everything should take less time. The reason is that when people are evaluating new machines, they need a basis of comparison—a benchmark. People often use familiar programs as benchmarks. It makes sense: you want a benchmark to be representative of the kind of work you do, and nothing is more representative of the work you do than the work you do!

Benchmarking sounds easy enough, provided you have timing tools. And you already know the meaning of time.¹ You just want to be sure that what those tools are reporting is the same as what you think you’re getting; especially if you have never used the tools before. To illustrate, imagine if someone took your watch and replaced it with another that expressed time in some funny units or three overlapping sets of hands. It would be very confusing; you might have a problem reading it at all. You would also be justifiably nervous about conducting your affairs by a watch you don’t understand.

UNIX timing tools are like the six-handed watch, reporting three different kinds of time measurements. They aren’t giving
conflicting information — they just present more information than you can jam into a single number. Again, the trick is learning to read the watch. That’s what the first part of this chapter is about. We’ll investigate the different types of measurements that determine how a program is doing.

If you plan to tune a program, you need more than timing information. Where is time being spent — in a single loop, subroutine call overhead, or with memory problems? For tuners, the latter sections of this chapter discuss how to profile code at the procedural and statement levels. We also discuss what profiles mean and how they predict the approach you have to take when, and if, you decide to tweak the code for performance, and what your chances for success will be.

### Timing

We assume that your program runs correctly. It would be rather ridiculous to time a program that’s not running right, though this doesn’t mean it doesn’t happen. Depending on what you are doing, you may be interested in knowing how much time is spent overall, or you may be looking at just a portion of the program. We show you how to time the whole program first, and then talk about timing individual loops or subroutines.

#### Timing a Whole Program

Under UNIX, you can time program execution by placing the time command before everything else you normally type on the command line. When the program finishes, a timing summary is produced. For instance, if your program is called `foo`, you can time its execution by typing `time foo`. If you are using the C shell or Korn shell, time is one of the shell’s built-in commands. With a Bourne shell, `time` is a separate command executable in `/bin`. In any case, the following information appears at the end of the run:

- User time
- System time
- Elapsed time

These timing figures are easier to understand with a little background. As your program runs, it switches back and forth between two fundamentally different modes: **user mode** and **kernel mode**. The normal operating state is user mode. It is in user mode that the instructions the compiler generated on your behalf get executed, in addition to any subroutine library calls linked with your program.² It might be enough to run in user mode forever, except that programs generally need other services, such as I/O, and these require the intervention of the operating system — the kernel. A kernel service request made by your program, or perhaps an event from outside your program, causes a switch from user mode into kernel mode.

Time spent executing in the two modes is accounted for separately. The **user time** figure describes time spent in user mode. Similarly, **system time** is a measure of the time spent in kernel mode. As far as user time goes, each program on the machine is accounted for separately. That is, you won’t be charged for activity in somebody else’s application. System time accounting works the same way, for the most part; however, you can, in some instances, be charged for some system services performed on other people’s behalf, in addition to your own. Incorrect charging occurs because your program may be executing at the moment some outside activity causes an interrupt. This seems unfair, but take
consolation in the fact that it works both ways: other users may be charged for your system activity too, for the same reason.

Taken together, user time and system time are called CPU time. Generally, the user time is far greater than the system time. You would expect this because most applications only occasionally ask for system services. In fact, a disproportionately large system time probably indicates some trouble. For instance, programs that are repeatedly generating exception conditions, such as page faults, misaligned memory references, or floating-point exceptions, use an inordinate amount of system time. Time spent doing things like seeking on a disk, rewinding a tape, or waiting for characters at the terminal doesn’t show up in CPU time. That’s because these activities don’t require the CPU; the CPU is free to go off and execute other programs.

The third piece of information (corresponding to the third set of hands on the watch), elapsed time, is a measure of the actual (wall clock) time that has passed since the program was started. For programs that spend most of their time computing, the elapsed time should be close to the CPU time. Reasons why elapsed time might be greater are:

- You are timesharing the machine with other active programs.
- Your application performs a lot of I/O.
- Your application requires more memory bandwidth than is available on the machine.
- Your program was paging or swapped.

People often record the CPU time and use it as an estimate for elapsed time. Using CPU time is okay on a single CPU machine, provided you have seen the program run when the machine was quiet and noticed the two numbers were very close together. But for multiprocessors, the total CPU time can be far different from the elapsed time. Whenever there is a doubt, wait until you have the machine to your- self and time your program then, using elapsed time. It is very important to produce timing results that can be verified using another run when the results are being used to make important purchasing decisions.

If you are running on a Berkeley UNIX derivative, the C shell’s built-in time command can report a number of other useful statistics. The default form of the output is shown in [Figure 1]. Check with your csh manual page for more possibilities.

In addition to figures for CPU and elapsed time, csh time command produces information about CPU utilization, page faults, swaps, blocked I/O operations (usually disk activity), and some measures of how much physical memory our program occupied when it ran. We describe each of them in turn.

**Percent utilization**

Percent utilization corresponds to the ratio of elapsed time to CPU time. As we mentioned above, there can be a number of reasons why the CPU utilization wouldn’t be 100% or mighty close. You can often get a hint from the other fields as to whether it is a problem with your program or whether you were sharing the machine when you ran it.

**Average real memory utilization**

The two average memory utilization measurements shown in [Figure 1] characterize the program’s resource requirements as it ran.
The first measurement, *shared-memory space*, accounts for the average amount of real memory taken by your program’s text segment — the portion that holds the machine instructions. It is called “shared” because several concurrently running copies of a program can share the same text segment (to save memory). Years ago, it was possible for the text segment to consume a significant portion of the memory system, but these days, with memory sizes starting around 32 MB, you have to compile a pretty huge source program and use every bit of it to create a shared-memory usage figure big enough to cause concern. The shared-memory space requirement is usually quite low relative to the amount of memory available on your machine.

![Diagram](image)

**Figure 1**: The built-in csh time function

The second average memory utilization measurement, *unshared-memory space*, describes the average real storage dedicated to your program’s data structures as it ran. This storage includes saved local variables and COMMON for FORTRAN, and static and external variables for C. We stress the word “real” here and above because these numbers talk about physical memory usage, taken over time. It may be that you have allocated arrays with 1 trillion elements (virtual space), but if your program only crawls into a corner of that space, your runtime memory requirements will be pretty low.

What the unshared-memory space measurement doesn’t tell you, unfortunately, is your program’s demand for memory at its greediest. An application that requires 100 MB 1/10th of the time and 1 KB the rest of the time appears to need only 10 MB on average — not a revealing picture of the program’s memory requirements.

**Blocked I/O operations**

The two figures for *blocked I/O* operations primarily describe disk usage, though tape devices and some other peripherals may also be used with blocked I/O. Character I/O operations, such as terminal input and output, do not appear here. A large number of blocked I/O operations could explain a lower-than-expected CPU utilization.

**Page faults and swaps**

An unusually high number of *page faults* or any swaps probably indicates a system choked for memory, which would also explain a longer-than-expected elapsed time. It may be that other programs are competing for the same space. And don’t forget that even under optimal conditions, every program suffers some number of page faults, as explained in [Section 1.1](#). Techniques for minimizing page faults are described in [Section 2.4](#).

**Timing a Portion of the Program**

For some benchmarking or tuning efforts, measurements taken on the “outside” of the program tell you everything you
need to know. But if you are trying to isolate performance figures for individual loops or portions of the code, you may want to include timing routines on the inside too. The basic technique is simple enough:

1. Record the time before you start doing X.
2. Do X.
3. Record the time at completion of X.
4. Subtract the start time from the completion time.

If, for instance, X’s primary job is to calculate particle positions, divide by the total time to obtain a number for particle positions/second. You have to be careful though; too many calls to the timing routines, and the observer becomes part of the experiment. The timing routines take time too, and their very presence can increase instruction cache miss or paging. Furthermore, you want X to take a significant amount of time so that the measurements are meaningful. Paying attention to the time between timer calls is really important because the clock used by the timing functions has a limited resolution. An event that occurs within a fraction of a second is hard to measure with any accuracy.

### Getting Time Information

In this section, we discuss methods for getting various timer values during the execution of your program.

For FORTRAN programs, a library timing function found on many machines is called `etime`, which takes a two-element `REAL*4` array as an argument and fills the slots with the user CPU time and system CPU time, respectively. The value returned by the function is the sum of the two. Here’s how `etime` is often used:

```fortran
real*4 tarray(2), etime
real*4 start, finish

start = etime(tarray)
finish = etime(tarray)

write (*,*) 'CPU time: ', finish - start
```

Not every vendor supplies an `etime` function; in fact, one doesn’t provide a timing routine for FORTRAN at all. Try it first. If it shows up as an undefined symbol when the program is linked, you can use the following C routine. It provides the same functionality as `etime`:

```c
#include <sys/times.h>
#define TICKS 100.

float etime (parts)
struct {
    float user;
    float system;
} *parts;
{
    struct tms local;
    times (&local);
    parts->user= (float) local.tms_utime/TICKS;
    parts->system = (float) local.tms_stime/TICKS;
```
There are a couple of things you might have to tweak to make it work. First of all, linking C routines with FORTRAN routines on your computer may require you to add an underscore (_ah after the function name. This changes the entry to \texttt{float etime\_ ( parts). Furthermore, you might have to adjust the \texttt{TICKS} parameter. We assumed that the system clock had a resolution of 1/100 of a second (true for the Hewlett-Packard machines that this version of \texttt{etime} was written for). 1/60 is very common. On an RS-6000 the number would be 1000. You may find the value in a file named \texttt{/usr/include/sys/param.h} on your machine, or you can determine it empirically.

A C routine for retrieving the wall time using calling \texttt{gettimeofday} is shown below. It is suitable for use with either C or FORTRAN programs as it uses call-by-value parameter passing:

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

void hpcwall(double *retval)
{
    static long zsec = 0;
    static long zusec = 0;
    double esec;
    struct timeval tp;
    struct timezone tzp;

    gettimeofday(&tp, &tzp);
    if ( zsec == 0 ) zsec = tp.tv_sec;
    if ( zusec == 0 ) zusec = tp.tv_usec;
    *retval = (tp.tv_sec - zsec) + (tp.tv_usec - zusec ) * 0.000001 ;
}
void hpcwall_(double *retval) { hpcwall(retval); } /* Other convention */
```

Given that you will often need both CPU and wall time, and you will be continually computing the difference between successive calls to these routines, you may want to write a routine to return the elapsed wall and CPU time upon each call as follows:

```fortran
SUBROUTINE HPCTIM(WTIME,CTIME)
  IMPLICIT NONE
  *
  REAL WTIME,CTIME
  COMMON/HPCTIMC/CBEGIN,WBEGIN
  REAL*8 CBEGIN,CEND,WBEGIN,WEND
  REAL ETIME,CSCRATCH(2)
  *
  CALL HPCWALL(WEND)
  CEND=ETIME(CSCRATCH)
  *
  WTIME = WEND - WBEGIN
```

Using Timing Information

You can get a lot of information from the timing facilities on a UNIX machine. Not only can you tell how long it takes to perform a given job, but you can also get hints about whether the machine is operating efficiently, or whether there is some other problem that needs to be factored in, such as inadequate memory.

Once the program is running with all anomalies explained away, you can record the time as a baseline. If you are tuning, the baseline will be a reference with which you can tell how much (or little) tuning has improved things. If you are benchmarking, you can use the baseline to judge how much overall incremental performance a new machine will give you. But remember to watch the other figures — paging, CPU utilization, etc. These may differ from machine to machine for reasons unrelated to raw CPU performance. You want to be sure you are getting the full picture.

Subroutine Profiling

Sometimes you want more detail than the overall timing of the application. But you don’t have time to modify the code to insert several hundred `etime` calls into your code. Profiles are also very useful when you have been handed a strange 20,000-line application program and told to figure out how it works and then improve its performance.

Most compilers provide a facility to automatically insert timing calls into your code at the entry and exit of each routine at compile time. While your program runs, the entry and exit times are recorded and then dumped into a file. A separate utility summarizes the execution patterns and produces a report that shows the percentage of the time spent in each of your routines and the library routines.

The profile gives you a sense of the shape of the execution profile. That is, you can see that 10% of the time is spent in subroutine A, 5% in subroutine B, etc. Naturally, if you add all of the routines together they should account for 100% of the overall time spent. From these percentages you can construct a picture — a `profile` — of how execution is distributed when the program runs. Though not representative of any particular profiling tool, the histograms in [Figure 2] and [Figure 3] depict these percentages, sorted from left to right, with each vertical column representing a different routine. They help illustrate different profile shapes.
A *sharp profile* says that most of the time is spent in one or two procedures, and if you want to improve the program’s performance you should focus your efforts on tuning those procedures. A minor optimization in a heavily executed line of code can sometimes have a great effect on the overall runtime, given the right opportunity. A *flat profile*, on the other hand, tells you that the runtime is spread across many routines, and effort spent optimizing any one or two will have little benefit in speeding up the program. Of course, there are also programs whose execution profile falls somewhere in the middle.

Figure $\PageIndex{2}$: Sharp profile — dominated by routine 1

We cannot predict with absolute certainty what you are likely to find when you profile your programs, but there are some general trends. For instance, engineering and scientific codes built around matrix solutions often exhibit very sharp profiles. The runtime is dominated by the work performed in a handful of routines. To tune the code, you need to focus your efforts on those routines to make them more efficient. It may involve restructuring loops to expose parallelism, providing hints to the compiler, or rearranging memory references. In any case, the challenge is tangible; you can see the problems you have to fix.

There are limits to how much tuning one or two routines will improve your runtime, of course. An often quoted rule of thumb is *Amdahl’s Law*, derived from remarks made in 1967 by one of the designers of the IBM 360 series, and founder of Amdahl Computer, Gene Amdahl. Strictly speaking, his remarks were about the performance potential of parallel computers, but people have adapted Amdahl’s Law to describe other things too. For our purposes, it goes like this: Say you have a program with two parts, one that can be optimized so that it goes infinitely fast and another that can’t be optimized at all. Even if the optimizable portion makes up 50% of the initial runtime, at best you will be able to cut the total runtime in half. That is, your runtime will eventually be dominated by the portion that can’t be optimized. This puts an upper limit on your expectations when tuning.

Figure $\PageIndex{3}$: Flat profile — no routine predominates
Even given the finite return on effort suggested by Amdahl’s Law, tuning a program with a sharp profile can be rewarding. Programs with flat profiles are much more difficult to tune. These are often system codes, nonnumeric applications, and varieties of numerical codes without matrix solutions. It takes a global tuning approach to reduce, to any justifiable degree, the runtime of a program with a flat profile. For instance, you can sometimes optimize instruction cache usage, which is complicated because of the program’s equal distribution of activity among a large number of routines. It can also help to reduce subroutine call overhead by folding callees into callers. Occasionally, you can find a memory reference problem that is endemic to the whole program — and one that can be fixed all at once.

When you look at a profile, you might find an unusually large percentage of time spent in the library routines such as log, exp, or sin. Often these functions are done in software routines rather than inline. You may be able to rewrite your code to eliminate some of these operations. Another important pattern to look for is when a routine takes far longer than you expect. Unexpected execution time may indicate you are accessing memory in a pattern that is bad for performance or that some aspect of the code cannot be optimized properly.

In any case, to get a profile, you need a profiler. One or two subroutine profilers come standard with the software development environments on all UNIX machines. We discuss two of them: prof and gprof. In addition, we mention a few line-by-line profilers. Subroutine profilers can give you a general overall view of where time is being spent. You probably should start with prof, if you have it (most machines do). Otherwise, use gprof. After that, you can move to a line-by-line profiler if you need to know which statements take the most time.

---

**prof**

prof is the most common of the UNIX profiling tools. In a sense, it is an extension of the compiler, linker, and object libraries, plus a few extra utilities, so it is hard to look at any one thing and say “this profiles your code.” prof works by periodically sampling the program counter as your application runs. To enable profiling, you must recompile and relink using the –p flag. For example, if your program has two modules, stuff.c and junk.c, you need to compile and link according to the following code:

```
% cc stuff.c -p -O -c
% cc junk.c -p -O -c
% cc stuff.o junk.o -p -o stuff
```

This creates a stuff binary that is ready for profiling. You don’t need to do anything special to run it. Just treat it normally by entering stuff. Because runtime statistics are being gathered, it takes a little longer than usual to execute. At completion, there is a new file called mon.out in the directory where you ran it. This file contains the history of stuff in binary form, so you can’t look at it directly. Use the prof utility to read mon.out and create a profile of stuff. By default, the information is written to your screen on standard output, though you can easily redirect it to a file:

```
% prof stuff > stuff.prof
```

To explore how the prof command works, we have created the following ridiculous little application, loops.c. It contains a main routine and three subroutines for which you can predict the time distribution just by looking at the code.

```
main () {
    int i;
    for (i=0;i<1000;i++) {
```
```c
if (l == 2*(l/2)) foo ();
bar();
baz();
}
foo (){  
    int j;  
    for (j=0;j<200;j++)  
}
bar (){  
    int i;  
    for (i=0;i<200;i++);  
}
baz (){  
    int k;  
    for (k=0;k<300;k++);  
}
```

Again, you need to compile and link `loops` with the `-p` flag, run the program, and then run the `prof` utility to extract a profile, as follows:

```
% cc loops.c -p -o loops
% ./loops
% prof loops > loops.prof
```

The following example shows what a `loops.prof` should look like. There are six columns.

<table>
<thead>
<tr>
<th>%Time</th>
<th>Seconds</th>
<th>Cumsecs</th>
<th>#Calls</th>
<th>msec/call</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>56.8</td>
<td>0.50</td>
<td>0.50</td>
<td>1000</td>
<td>0.500</td>
<td>_baz</td>
</tr>
<tr>
<td>27.3</td>
<td>0.24</td>
<td>0.74</td>
<td>1000</td>
<td>0.240</td>
<td>_bar</td>
</tr>
<tr>
<td>15.9</td>
<td>0.14</td>
<td>0.88</td>
<td>500</td>
<td>0.28</td>
<td>_foo</td>
</tr>
<tr>
<td>0.0</td>
<td>0.00</td>
<td>0.88</td>
<td>1</td>
<td>0.</td>
<td>_creat</td>
</tr>
<tr>
<td>0.0</td>
<td>0.00</td>
<td>0.88</td>
<td>2</td>
<td>0.</td>
<td>_profil</td>
</tr>
<tr>
<td>0.0</td>
<td>0.00</td>
<td>0.88</td>
<td>1</td>
<td>0.</td>
<td>_main</td>
</tr>
<tr>
<td>0.0</td>
<td>0.00</td>
<td>0.88</td>
<td>3</td>
<td>0.</td>
<td>_getenv</td>
</tr>
<tr>
<td>0.0</td>
<td>0.00</td>
<td>0.88</td>
<td>1</td>
<td>0.</td>
<td>_strcpy</td>
</tr>
<tr>
<td>0.0</td>
<td>0.00</td>
<td>0.88</td>
<td>1</td>
<td>0.</td>
<td>_write</td>
</tr>
</tbody>
</table>

The columns can be described as follows:

- **%Time** Percentage of CPU time consumed by this routine
- **Seconds** CPU time consumed by this routine
- **Cumsecs** A running total of time consumed by this and all preceding routines in the list
- **Calls** The number of times this particular routine was called
- **msec/call** Seconds divided by number of calls giving the average length of time taken by each invocation of the routine
- **Name** The name of this routine

The top three routines listed are from `loops.c` itself. You can see an entry for the “main” routine more than halfway down the list. Depending on the vendor, the names of the routines may contain leading or trailing underscores, and there will
always be some routines listed you don’t recognize. These are contributions from the C library and possibly the FORTRAN libraries, if you are using FORTRAN. Profiling also introduces some overhead into the run, and often shows up as one or two subroutines in the prof output. In this case, the entry for _profil represents code inserted by the linker for collecting runtime profiling data.

If it was our intention to tune loops, we would consider a profile like the one in the figure above to be a fairly good sign. The lead routine takes 50% of the runtime, so at least there is a chance we could do something with it that would have a significant impact on the overall runtime. (Of course with a program as trivial as loops, there is plenty we can do, since loops does nothing.)

### gprof

Just as it’s important to know how time is distributed when your program runs, it’s also valuable to be able to tell who called who in the list of routines. Imagine, for instance, if something labeled _exp showed up high in the list in the prof output. You might say: “Hmmm, I don’t remember calling anything named exp(). I wonder where that came from.” A call tree helps you find it.

Subroutines and functions can be thought of as members of a family tree. The top of the tree, or root, is actually a routine that precedes the main routine you coded for the application. It calls your main routine, which in turn calls others, and so on, all the way down to the leaf nodes of the tree. This tree is properly known as a call graph. The relationship between routines and nodes in the graph is one of parents and children. Nodes separated by more than one hop are referred to as ancestors and descendants.

[Figure 4] graphically depicts the kind of call graph you might see in a small application. main is the parent or ancestor of most of the rest of the routines. G has two parents, E and C. Another routine, A, doesn’t appear to have any ancestors or descendants at all. This problem can happen when routines are not compiled with profiling enabled, or when they aren’t invoked with a subroutine call — such as would be the case if A were an exception handler.

The UNIX profiler that can extract this kind of information is called gprof. It replicates the abilities of prof, plus it gives a call graph profile so you can see who calls whom, and how often. The call graph profile is handy if you are trying to figure out how a piece of code works or where an unknown routine came from, or if you are looking for candidates for subroutine inlining.

To use call graph profiling you need go through the same steps as with prof, except that a -pg flag is substituted for the -p flag. Additionally, when it comes time to produce the actual profile, you use the gprof utility instead of prof. One other difference is that the name of the statistics file is gmon.out instead of mon.out:

```bash
% cc -pg stuff.c -c
% cc stuff.o -pg -o stuff
% stuff
% gprof stuff > stuff.gprof
```
The output from *gprof* is divided into three sections:

- Call graph profile
- Timing profile
- Index

The first section textually maps out the call graph. The second section lists routines, the percentage of time devoted to each, the number of calls, etc. (similar to *prof*). The third section is a cross reference so that you can locate routines by number, rather than by name. This section is especially useful for large applications because routines are sorted based on the amount of time they use, and it can be difficult to locate a particular routine by scanning for its name. Let’s invent another trivial application to illustrate how *gprof* works. [Figure 5] shows a short piece of FORTRAN code, along with a diagram of how the routines are connected together. Subroutines A and B are both called by MAIN, and, in turn, each calls C. The following example shows a section of the output from *gprof*’s call graph profile.8
Sandwiched between each set of dashed lines is information describing a given routine and its relationship to parents and children. It is easy to tell which routine the block represents because the name is shifted farther to the left than the others. Parents are listed above, children below. As with prof, underscores are tacked onto the labels. A description of each of the columns follows:

- **index** You will notice that each routine name is associated with a number in brackets ([n]). This is a cross-reference for locating the routine elsewhere in the profile. If, for example, you were looking at the block describing _MAIN_ and wanted to know more about one of its children, say _a_, you could find it by scanning down the left side of the page for its index, [5].

- **%time** The meaning of the %time field is a little different than it was for prof. In this case it describes the percentage of time spent in this routine plus the time spent in all of its children. It gives you a quick way to determine where the busiest sections of the call graph can be found.

- **self** Listed in seconds, the self column has different meanings for parents, the routine in question, and its children. Starting with the middle entry — the routine itself — the self figure shows how much overall time was dedicated to the routine. In the case _b_, for instance, this amounts to 3.23 seconds. Each self column entry shows the amount of time that can be attributed to calls from the parents. If you look at routine _c_, for example, you will see that it consumed a total time of 3.23 seconds. But note that it had two parents: 1.62 seconds of the time was attributable to calls from _a_, and 1.62 seconds to _b_. For the children, the self figure shows how much time was spent executing each child due to calls from this routine. The children may have consumed more time overall, but the only time accounted for is time-attributable to calls from this routine. For example, _c_ accumulated 3.23 seconds overall, but if you look at the block describing _b_, you see _c_ listed as a child with only 1.62 seconds. That’s the total time spent executing _c_ on behalf of _b_.

- **descendants** As with the self column, figures in the descendants column have different meanings for the routine, its parents, and children. For the routine itself, it shows the number of seconds spent in all of its descendants. For the routine’s parents, the descendants figure describes how much time spent in the routine can be traced back to calls by each parent. Looking at routine _c_, again, you can see that of its total time, 3.23 seconds, 1.62 seconds were attributable to each of its two parents, _a_ and _b_.
For the children, the descendants column shows how much of the child’s time can be attributed to calls from this routine. The child may have accumulated more time overall, but the only time displayed is time associated with calls from this routine.

- **calls** The *calls* column shows the number of times each routine was invoked, as well as the distribution of those calls associated with both parents and children. Starting with the routine itself, the figure in the *calls* column shows the total number of entries into the routine. In situations where the routine called itself, you will also see a +n immediately appended, showing that additional n calls were made recursively. Parent and child figures are expressed as ratios. For the parents, the ratio m/n says “of the n times the routine was called, m of those calls came from this parent.” For the child, it says “of the n times this child was called, m of those calls came from this routine.”

---

### gprof’s Flat Profile

As we mentioned previously, *gprof* also produces a timing profile (also called a “flat” profile, just to confuse things) similar to the one produced by *prof*. A few of the fields are different from *prof*, and there is some extra information, so it will help if we explain it briefly. The following example shows the first few lines from a *gprof* flat profile for *stuff*. You will recognize the top three routines from the original program. The others are library functions included at link-time.

<table>
<thead>
<tr>
<th>%time</th>
<th>cumulative</th>
<th>self</th>
<th>self</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>time</td>
<td>seconds</td>
<td>seconds</td>
<td>calls</td>
</tr>
<tr>
<td>39.9</td>
<td>3.23</td>
<td>3.23</td>
<td>2  1615.07 1615.07</td>
<td><em>c</em> [6]</td>
</tr>
<tr>
<td>39.9</td>
<td>6.46</td>
<td>3.23</td>
<td>1  3230.14 4845.20</td>
<td><em>b</em> [4]</td>
</tr>
<tr>
<td>20.0</td>
<td>8.08</td>
<td>1.62</td>
<td>1  1620.07 3235.14</td>
<td><em>a</em> [5]</td>
</tr>
<tr>
<td>0.1</td>
<td>8.09</td>
<td>0.01</td>
<td>3  3.33     3.33</td>
<td>_ioctl [9]</td>
</tr>
<tr>
<td>0.0</td>
<td>8.09</td>
<td>0.00</td>
<td>64 0.00     0.00</td>
<td>.rem [12]</td>
</tr>
<tr>
<td>0.0</td>
<td>8.09</td>
<td>0.00</td>
<td>64 0.00     0.00</td>
<td>_f_clos [177]</td>
</tr>
<tr>
<td>0.0</td>
<td>8.09</td>
<td>0.00</td>
<td>20 0.00     0.00</td>
<td>_sigblock [178]</td>
</tr>
<tr>
<td>...</td>
<td>....</td>
<td>....</td>
<td>....</td>
<td>....</td>
</tr>
</tbody>
</table>

Here’s what each column means:

- **%time** Again, we see a field that describes the runtime for each routine as a percentage of the overall time taken by the program. As you might expect, all the entries in this column should total 100% (nearly).

- **cumulative seconds** For any given routine, the column called “cumulative seconds” tallies a running sum of the time taken by all the preceding routines plus its own time. As you scan towards the bottom, the numbers asymptotically approach the total runtime for the program.

- **self seconds** Each routine’s individual contribution to the runtime.

- **calls** The number of times this particular routine was called.

- **self ms/call** Seconds spent inside the routine, divided by the number of calls. This gives the average length of time taken by each invocation of the routine. The figure is presented in milliseconds.

- **total ms/call** Seconds spent inside the routine plus its descendants, divided by the number of calls.

- **name** The name of the routine. Notice that the cross-reference number appears here too.
Accumulating the Results of Several gprof Runs

It is possible to accumulate statistics from multiple runs so that you can get a picture of how a program is doing with a variety of data sets. For instance, say that you wanted to profile an application — call it bar — with three different sets of input data. You could perform the runs separately, saving the gmon.out files as you go, and then combine the results into a single profile at the end:

% f77 -pg bar.f -o bar
% bar < data1.input
% mv gmon.out gmon.1
% bar < data2.input
% mv gmon.out gmon.2
% bar < data3.input
% gprof bar -s gmon.1 gmon.2 gmon.out > gprof.summary.out

In the example profile, each run along the way creates a new gmon.out file that we renamed to make room for the next one. At the end, gprof combines the information from each of the data files to produce a summary profile of bar in the file gprof.summary.out. Additionally (you don’t see it here), gprof creates a file named gmon.sum that contains the merged data from the original three data files. gmon.sum has the same format as gmon.out, so you can use it as input for other merged profiles down the road.

In form, the output from a merged profile looks exactly the same as for an individual run. There are a couple of interesting things you will note, however. For one thing, the main routine appears to have been invoked more than once — one time for each run, in fact. Furthermore, depending on the application, multiple runs tend to either smooth the contour of the profile or exaggerate its features. You can imagine how this might happen. If a single routine is consistently called while others come and go as the input data changes, it takes on increasing importance in your tuning efforts.

A Few Words About Accuracy

For processors running at 600 MHz and more, the time between 60 Hz and 100 Hz samples is a veritable eternity. Furthermore, you can experience quantization errors when the sampling frequency is fixed, as is true of steady 1/100th or 1/60th of a second samples. To take an exaggerated example, assume that the timeline in [Figure 6] shows alternating calls to two subroutines, BAR and FOO. The tick marks represent the sample points for profiling.

BAR and FOO take turns running. In fact, BAR takes more time than FOO. But because the sampling interval closely matches the frequency at which the two subroutines alternate, we get a quantizing error: most of the samples happen to be taken while FOO is running. Therefore, the profile tells us that FOO took more CPU time than BAR.

We have described the tried and true UNIX subroutine profilers that have been available for years. In many cases,
vendors have much better tools available for the asking or for a fee. If you are doing some serious tuning, ask your vendor representative to look into other tools for you.

---

**Basic Block Profilers**

There are several good reasons to desire a finer level of detail than you can see with a subroutine profiler. For humans trying to understand how a subroutine or function is used, a profiler that tells which lines of source code were actually executed, and how often, is invaluable; a few clues about where to focus your tuning efforts can save you time. Also, such a profiler saves you from discovering that a particularly clever optimization makes no difference because you put it in a section of code that never gets executed.

As part of an overall strategy, a subroutine profile can direct you to a handful of routines that account for most of the runtime, but it takes a basic block profiler to get you to the associated source code lines.

Basic block profilers can also provide compilers with information they need to perform their own optimizations. Most compilers work in the dark. They can restructure and unroll loops, but they cannot tell when it will pay off. Worse yet, misplaced optimizations often have an adverse effect of slowing down the code! This can be the result of added instruction cache burden, wasted tests introduced by the compiler, or incorrect assumptions about which way a branch would go at runtime. If the compiler can automatically interpret the results of a basic block profile, or if you can supply the compiler with hints, it often means a reduced run-time with little effort on your part.

There are several basic block profilers in the world. The closest thing to a standard, tcov, is shipped with Sun workstations; it’s standard because the installed base is so big. On MIPS-based workstations, such as those from Silicon Graphics and DEC, the profiler (packaged as an extension to prof) is called pixie. We explain briefly how to run each profiler using a reasonable set of switches. You can consult your manual pages for other options.

---

**tcov**

`tcov`, available on Sun workstations and other SPARC machines that run SunOS, gives execution statistics that describe the number of times each source statement was executed. It is very easy to use. Assume for illustration that we have a source program called `foo.c`. The following steps create a basic block profile:

```
% cc -a foo.c -o foo
% foo
% tcov foo.c
```

The `−a` option tells the compiler to include the necessary support for `tcov`.\(^{11}\) Several files are created in the process. One called `foo.d` accumulates a history of the execution frequencies within the program `foo`. That is, old data is updated with new data each time `foo` is run, so you can get an overall picture of what happens inside `foo`, given a variety of data sets. Just remember to clean out the old data if you want to start over. The profile itself goes into a file called `foo.tcov`.

Let’s look at an illustration. Below is a short C program that performs a bubble sort of 10 integers:

```c
int n[] = {23,12,43,2,98,78,2,51,77,8};
```
main ()
{
    int i, j, ktemp;
    for (i=10; i>0; i--) {
        for (j=0; j<i; j++) {
            if (n[j] < n[j+1]) {
                ktemp = n[j+1], n[j+1] = n[j], n[j] = ktemp;
            }
        }
    }
}

_tcov produces a basic block profile that contains execution counts for each source line, plus some summary
statistics (not shown):

```c
int n[] = {23,12,43,2,98,78,2,51,77,8};
main ()
1 -> {
    int i, j, ktemp;
    10 -> for (i=10; i>0; i--) {
    10, 55 -> for (j=0; j<i; j++) {
      55 -> if (n[j] < n[j+1]) {
            23 -> ktemp = n[j+1], n[j+1] = n[j], n[j] = ktemp;
        }
    }
}
1 -> }
```

The numbers to the left tell you the number of times each block was entered. For instance, you can see that the
routine was entered just once, and that the highest count occurs at the test `n[j] < n[j+1]`. _tcov_ shows more than one count on a line in places where the compiler has created more than one block.

_pixie_

_pixie_ is a little different from _tcov_. Rather than reporting the number of times each source line was executed, pixie reports the number of machine clock cycles devoted to executing each line. In theory, you could use this to calculate the amount of time spent per statement, although anomalies like cache misses are not represented.

_pixie_ works by “pixifying” an executable file that has been compiled and linked in the normal way. Below we run _pixie_ on _foo_ to create a new executable called _foo.pixie_:

```bash
% cc foo.c -o foo
% pixie foo
% foo.pixie

% prof -pixie foo
```

Also created was a file named _foo.Addr_ , which contains addresses for the basic blocks within _foo_. When the new program, _foo.pixie_ , is run, it creates a file called _foo.Count_ , containing execution counts for the basic blocks whose addresses are stored in _foo.Addr_. _pixie_ data accumulates from run to run. The statistics are retrieved using _prof_ and a special _-pixie_ flag.
Pixie’s default output comes in three sections and shows:

- Cycles per routine
- Procedure invocation counts
- Cycles per basic line

Below, we have listed the output of the third section for the bubble sort:

<table>
<thead>
<tr>
<th>procedure (file)</th>
<th>line</th>
<th>bytes</th>
<th>cycles</th>
<th>%</th>
<th>cum %</th>
</tr>
</thead>
<tbody>
<tr>
<td>main (foo.c)</td>
<td>7</td>
<td>44</td>
<td>605</td>
<td>12.11</td>
<td>12.11</td>
</tr>
<tr>
<td>_cleanup (flsbuf.c)</td>
<td>59</td>
<td>20</td>
<td>500</td>
<td>10.01</td>
<td>22.13</td>
</tr>
<tr>
<td>fclose (flsbuf.c)</td>
<td>81</td>
<td>20</td>
<td>500</td>
<td>10.01</td>
<td>32.14</td>
</tr>
<tr>
<td>fclose (flsbuf.c)</td>
<td>94</td>
<td>20</td>
<td>500</td>
<td>10.01</td>
<td>42.15</td>
</tr>
<tr>
<td>_cleanup (flsbuf.c)</td>
<td>54</td>
<td>20</td>
<td>500</td>
<td>10.01</td>
<td>52.16</td>
</tr>
<tr>
<td>fclose (flsbuf.c)</td>
<td>76</td>
<td>16</td>
<td>400</td>
<td>8.01</td>
<td>60.17</td>
</tr>
<tr>
<td>main (foo.c)</td>
<td>10</td>
<td>24</td>
<td>298</td>
<td>5.97</td>
<td>66.14</td>
</tr>
<tr>
<td>main (foo.c)</td>
<td>8</td>
<td>36</td>
<td>207</td>
<td>4.14</td>
<td>70.28</td>
</tr>
</tbody>
</table>

Here you can see three entries for the main routine from foo.c, plus a number of system library routines. The entries show the associated line number and the number of machine cycles dedicated to executing that line as the program ran. For instance, line 7 of foo.c took 605 cycles (12% of the runtime).

**Virtual Memory**

In addition to the negative performance impact due to cache misses, the virtual memory system can also slow your program down if it is too large to fit in the memory of the system or is competing with other large jobs for scarce memory resources.

Under most UNIX implementations, the operating system automatically pages pieces of a program that are too large for the available memory out to the swap area. The program won’t be tossed out completely; that only happens when memory gets extremely tight, or when your program has been inactive for a while. Rather, individual pages are placed in the swap area for later retrieval. First of all, you need to be aware that this is happening if you don’t already know about it. Second, if it is happening, the memory access patterns are critical. When references are too widely scattered, your runtime will be completely dominated by disk I/O.

If you plan in advance, you can make a virtual memory system work for you when your program is too large for the physical memory on the machine. The techniques are exactly the same as those for tuning a software-managed out-of-core solution, or loop nests. The process of “blocking” memory references so that data consumed in neighborhoods uses a bigger portion of each virtual memory page before rotating it out to disk to make room for another.12

**Gauging the Size of Your Program and the Machine’s Memory**

How can you tell if you are running out-of-core? There are ways to check for paging on the machine, but perhaps the most straightforward check is to compare the size of your program against the amount of available memory. You do this...
with the `size` command:

```
% size myprogram
```

On a System V UNIX machine, the output looks something like this:

```
53872 + 53460 + 10010772 = 10118104
```

On a Berkeley UNIX derivative you see something like this:

```
text data bss hex decimal
53872 53460 10010772 9a63d8 10118104
```

The first three fields describe the amount of memory required for three different portions of your program. The first, text, accounts for the machine instructions that make up your program. The second, data, includes initialized values in your program such as the contents of data statements, common blocks, externals, character strings, etc. The third component, bss, (block started by symbol), is usually the largest. It describes an uninitialized data area in your program. This area would be made of common blocks that are not set by a block data. The last field is a total for all three sections added together, in bytes.\(^\text{13}\)

Next, you need to know how much memory you have in your system. Unfortunately, there isn’t a standard UNIX command for this. On the RS/6000, `/etc/lscfg` tells you. On an SGI machine, `/etc/INV` does it. Many System V UNIX implementations have an `/etc/memsize` command. On any Berkeley derivative, you can type:

```
% ps aux
```

This command gives you a listing of all the processes running on the machine. Find the process with the largest value in the `%MEM`. Divide the value in the `RSS` field by the percentage of memory used to get a rough figure for how much memory your machine has:

```
memory = RSS/(%MEM/10)
```

For instance, if the largest process shows 5% memory usage and a resident set size (RSS) of 840 KB, your machine has \(840000/(5/100) = 16\) MB of memory.\(^{14}\) If the answer from the size command shows a total that is anywhere near the amount of memory you have, you stand a good chance of paging when you run — especially if you are doing other things on the machine at the same time.

---

**Checking for Page Faults**

Your system’s performance monitoring tools tell you if programs are paging. Some paging is OK; page faults and “page-ins” occur naturally as programs run. Also, be careful if you are competing for system resources along with other users. The picture you get won’t be the same as when you have the computer to yourself.

To check for paging activity on a Berkeley UNIX derivative, use the `vmstat` command. Commonly people invoke it with a time increment so that it reports paging at regular intervals:

```
% vmstat 5
```
This command produces output every five seconds.

```
procs     memory              page               disk       faults     cpu
r b w   avm   fre  re at  pi  fr  de sr s0 d2 d3 in sy cs us sy id
0 0 0   824 21568   0  0   0  0  0  0  0  0  0  0  0  0  0  20 37 13  0 198
0 0 0   840 21508   0  0   0  0  0  0  0  0  0  0  0  1  0  0  0 251 186 156  0 10 90
0 0 0   846 21460   0  0   0  0  0  0  0  0  0  0  0  2  0  0  0 248 149 152  1  9 89
0 0 0   918 21444   0  0   0  0  0  0  0  0  0  0  0  4  0  0  0 258 143 152  2 10 89
```

Lots of valuable information is produced. For our purposes, the important fields are `avm` or active virtual memory, the `fre` or free real memory, and the `pi` and `po` numbers showing paging activity. When the `fre` figure drops to near zero, and the `po` field shows a lot of activity, it's an indication that the memory system is overworked.

On a SysV machine, paging activity can be seen with the sar command:

```
% sar -r 5 5
```

This command shows you the amount of free memory and swap space presently available. If the free memory figure is low, you can assume that your program is paging:

```
Sat Apr 18 20:42:19
[r] freemem freeswap
 4032 82144
```

As we mentioned earlier, if you must run a job larger than the size of the memory on your machine, the same sort of advice that applied to conserving cache activity applies to paging activity. Try to minimize the stride in your code, and where you can't, blocking memory references helps a whole lot.

A note on memory performance monitoring tools: you should check with your workstation vendor to see what they have available beyond `vmstat` or `sar`. There may be much more sophisticated (and often graphical) tools that can help you understand how your program is using memory.

---

**Closing Notes**

We have seen some of the tools for timing and profiling. Even though it seems like we covered a lot, there are other kinds of profiles we would like to be able to cover — cache miss measurements, runtime dependency analysis, flop measurements, and so on. These profiles are good when you are looking for particular anomalies, such as cache miss or floating-point pipeline utilization. Profilers for these quantities exist for some machines, but they aren't widely distributed.

One thing to keep in mind: when you profile code you sometimes get a very limited view of the way a program is used. This is especially true if it can perform many types of analyses for many different sets of input data. Working with just one or two profiles can give you a distorted picture of how the code operates overall. Imagine the following scenario: someone invites you to take your very first ride in an automobile. You get in the passenger’s seat with a sketch pad and a pen, and record everything that happens. Your observations include some of the following:

- The radio is always on.
• The windshield wipers are never used.
• The car moves only in a forward direction.

The danger is that, given this limited view of the way a car is operated, you might want to disconnect the radio’s on/off knob, remove the windshield wipers, and eliminate the reverse gear. This would come as a real surprise to the next person who tries to back the car out on a rainy day! The point is that unless you are careful to gather data for all kinds of uses, you may not really have a picture of how the program operates. A single profile is fine for tuning a benchmark, but you may miss some important details on a multipurpose application. Worse yet, if you optimize it for one case and cripple it for another, you may do far more harm than good.

Profiling, as we saw in this chapter, is pretty mechanical. Tuning requires insight. It’s only fair to warn you that it isn’t always rewarding. Sometimes you pour your soul into a clever modification that actually increases the runtime. Argh! What went wrong? You’ll need to depend on your profiling tools to answer that.

### Exercises

#### Exercise 1
Profile the following program using gprof. Is there any way to tell how much of the time spent in routine c was due to recursive calls?

```c
main()
{
    int i, n=10;
    for (i=0; i<1000; i++) {
        c(n);
        a(n);
    }
    c(n)
    int n;
    {
        if (n > 0) {
            a(n-1);
            c(n-1);
        }
    }
    a(n)
    int n;
    {
        c(n);
    }
}
```

#### Exercise 2
Profile an engineering code (floating-point intensive) with full optimization on and off. How does the profile change? Can you explain the change?
Exercise (PageIndex(3))

Write a program to determine the overhead of the getrusage and the etime calls. Other than consuming processor time, how can making a system call to check the time too often alter the application performance?

Footnotes

1. Time is money.
2. Cache miss time is buried in here too.
3. The uptime command gives you a rough indication of the other activity on your machine. The last three fields tell the average number of processes ready to run during the last 1, 5, and 15 minutes, respectively.
4. The term “flat profile” is a little overloaded. We are using it to describe a profile that shows an even distribution of time throughout the program. You will also see the label flat profile used to draw distinction from a call graph profile, as described below.
5. Remember: code with profiling enabled takes longer to run. You should recompile and relink the whole thing without the \( -p \) flag when you have finished profiling.
6. It doesn’t have to be a tree. Any subroutine can have more than one parent. Furthermore, recursive subroutine calls introduce cycles into the graph, in which a child calls one of its parents.
7. On HP machines, the flag is \( -G \).
8. In the interest of conserving space, we clipped out the section most relevant to our discussion and included it in this example. There was a lot more to it, including calls of setup and system routines, the likes of which you will see when you run gprof.
9. You may have noticed that there are two main routines: \_MAIN\_ and \_main\_. In a FORTRAN program, \_MAIN\_ is the actual FORTRAN main routine. It’s called as a subroutine by \_main\_, provided from a system library at link time. When you’re profiling C code, you won’t see \_MAIN\_.
10. A basic block is a section of code with only one entrance and one exit. If you know how many times the block was entered, you know how many times each of the statements in the block was executed, which gives you a line-by-line profile. The concept of a basic block is explained in detail in \[Section 2.1\].
11. On Sun Solaris systems, the \( -xa \) option is used.
12. We examine the techniques for blocking in Chapter 8.
13. Warning: The size command won’t give you the full picture if your program allocates memory dynamically, or keeps data on the stack. This area is especially important for C programs and FORTRAN programs that create large arrays that are not in COMMON.
14. You could also reboot the machine! It will tell you how much memory is available when it comes up.
15. By the way, are you getting the message “Out of memory?” If you are running csh, try typing \texttt{unlimit} to see if the message goes away. Otherwise, it may mean that you don’t have enough swap space available to run the job.