Timing and Profiling - Basic Block Profilers*

Charles Severance
Kevin Dowd

This work is produced by OpenStax-CNX and licensed under the
Creative Commons Attribution License 3.0†

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*[1] 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.

1 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:

*Version 1.3: Aug 25, 2010 11:16 am -0500

†http://creativecommons.org/licenses/by/3.0/

[1] 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 (<http://cnx.org/content/m33690/latest/>)

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

The `-a` option tells the compiler to include the necessary support for *tcov*.[2] Several files are created in the process. One called *foo.d* accumulates a history of the exe- cution 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:

```
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):

```
        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.

[2] On Sun Solaris systems, the `-xa` option is used.

2 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*:

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

Also created was a file named *foo.Addrs*, which contains addresses for the basic blocks within *foo*. When the new program, *foo.pixie* , is run, it creates a file called *foo.Counts* , containing execution counts for the basic blocks whose addresses are stored in *foo.Addrs*. *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:

```
procedure (file)          line  bytes      cycles      %  cum %

main (foo.c)                 7     44         605  12.11  12.11
_cleanup (flsbuf.c)         59     20         500  10.01  22.13
fclose (flsbuf.c)           81     20         500  10.01  32.14
fclose (flsbuf.c)           94     20         500  10.01  42.15
_cleanup (flsbuf.c)         54     20         500  10.01  52.16
fclose (flsbuf.c)           76     16         400   8.01  60.17
main (foo.c)                10     24         298   5.97  66.14
main (foo.c)                 8     36         207   4.14  70.28
....                        ..     ..          ..    ...    ...
```

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).