AT&T Bell Laboratories Technical Journal
Vol. 63, No. 8, October 1984
Printed in U.S.A.

The UNIX *System:*

Cheap Dynamic Instruction Counting

By P. J. WEINBERGER*

(Manuscript received October 18, 1983)

There are two ways to profile the behavior of a program: timing and counting. Timing is traditional in *UNIX*™ operating systems. This paper describes an easy implementation of count profiling, and gives several examples and applications. It has been implemented on the Motorola 68000, VAX™, and AT&T 3B20 computers.

I. INTRODUCTION

Measurement and testing form the bridge between the algorithms of the theoreticians and efficient working programs. In all but the simplest and shortest-running programs, the implementer makes assumptions about the form and quantity of the input, and about which parts of the program do or do not need to be fast. Unless these assumptions are based on careful measurement, they are usually inaccurate, and so the program is unexpectedly slow. Likewise, it is a common observation that testing a large program does not find all the bugs, and that it is hard even to execute all parts of the program.

This paper presents a technique for ameliorating both of these difficulties. If a programmer is told how often each instruction is executed, then it is easy to tell whether a set of tests has executed all the instructions, and the parts of the program executed most fre-

* AT&T Bell Laboratories.

quently stand out clearly. This is not a new idea; several compilers have generated counting code (see Ref. 1.) Strangely enough, counting facilities are rare to nonexistent in production environments. (See Ref. 2, Section 3.1, for more comments on testing and profiling.)

The next section contains a brief discussion of time-based profiling. Following that is a description of an implementation of counting-based profiling. Then follow some examples and applications.

II. TIME PROFILING

The usual way of measuring performance is by timing. At best this gives fairly crude data, unless the machine has an accurate clock.[3] The *UNIX* operating system includes a profiler based on timing. A program that requests profiling tells the system the location of an array of counters, one for each n bytes of its executable test. Then every time the hardware clock ticks (50, 60, or 100 times a second) when the program is running, the kernel increments the word in the counting array corresponding to the program counter. When the program finishes, the user can see how much time is spent in each routine. This is an immensely valuable but flawed tool. First, it requires quite a large table to record exactly which instruction was being executed when the clock ticked, and this is not the default. Typically, the values are compressed corresponding to a single counter with each range of $n = 8$ bytes. Occasionally, counts from one routine are attributed to a neighboring one, when the section of the program corresponding to a counter spans two routines. Second, even on a slow machine, 10,000 instructions are executed for every one that is profiled, so it is impossible to get reliable counts for any but a few subroutines except on long-running programs. For instance, a program that runs 40 seconds is sampled 2400 times. If a subroutine accounts for 20 percent of the time, it should have been counted 480 times, with a standard deviation of about 22 counts. Thus, the expected inaccuracy even for an 8-second routine is about 10 percent, and for less time, even less accuracy is expected. Correspondingly, there is little chance of estimating test coverage with sampling. Finally, if the behavior of the program is at all correlated with the clock, then the sampling is not random. Communications programs and those that do a lot of input/output (I/O) are at least partially synchronized with the clock, and their timings are unreliable.

III. COUNTING

An alternative is to count every execution of every instruction. For the moment, think of the program as being in assembly language. If you insert a counting statement at the beginning of each basic block of the program, you know how often each instruction is executed. For

the purposes of this paper, a basic block is a contiguous set of instructions, all of which have to be executed exactly once if the first is executed, and conversely. If the program terminates abnormally, then the last basic block will have been started, and so counted, but instructions after the failure are not counted. In that case a few instructions in one basic block may have counts that are one too large.

What does it take to carry this out? First, detect the beginning of basic blocks. Second, insert some counting code that does not affect the correctness of the program. Third, retrieve the counts when the program terminates. (It is also useful to get counts from programs that do not terminate, like the operating system.) Fourth, find some way of correlating the data with the original source of the program. Thus the implementation consists of two parts. The first scans through the program, inserting counting code and allocating storage for the counts. The second takes the count output and produces various sorts of reports. In between, run the program being profiled.

IV. C, FORTRAN, AND PASCAL

Above I maintained the fiction that counting was for assembly language programs. Assembly language is produced by the compilers, so that the counting code is inserted by a separate pass after the compiler and before the assembler. The association between basic blocks and lines of the source program is made by compiling with an option that produces line numbers in the symbol table for the debugger. The program that inserts counting code also interprets line number and file name assembler directives, and leaves a file containing the correspondence between basic blocks and line numbers. The following diagram shows the normal flow of events for C programs (see Fig. 1).

A program named `bb` inserts counting code in the assembly language (see Fig. 2).

The file `x.sL` contains each machine instruction in the original program with the number of the basic block it is in, together with lines noting line numbers and function names.

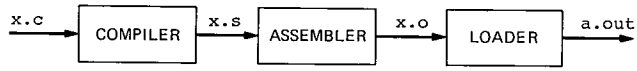

Fig. 1—Normal compilation flow for C programs.

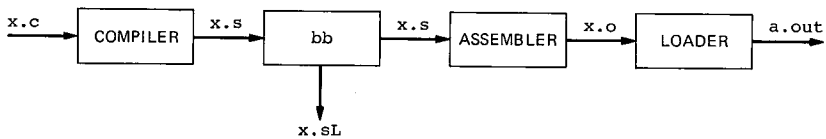

Fig. 2—Inserting counting code.

This file, the source file, and the output file containing counts are combined to give program listings containing the number of times each line was executed.

V. BASIC BLOCKS

It should be easy to find the beginning of a basic block. Any instruction that is the target of a branch and any instruction following a branch starts a basic block. Fortunately, *UNIX* system assemblers have the property that all branches lead to labels, so one can take all labeled instructions to begin basic blocks, rather than doing flow analysis or address arithmetic (except that the label the compiler generates just after a `case` or `switch` instruction must be ignored, since inserting code would spread out the jump table and make the program incorrect).

It is clear that one need not count all basic blocks. The counts associated with some are implied by counts associated with others. For instance, in

if *cond* then *true-piece* else *false-piece*

the sum of the counts for the *true* and *false* pieces must equal that of the *cond* piece (see Ref. 4). A compiler could take advantage of this information, but a program processing assembly language would have to do flow analysis. Also, the program that prints the counts would need the source, the count data, and the rules for deriving the implied counts. I do not take advantage of this opportunity.

VI. TRANSPARENT COUNTING CODE

What kind of code should be inserted? With each compilation unit (a file), allocate an array of integers, one for each basic block. Then the counting code for a basic block should add "one" to the array element for the basic block.

Although an array of integers is specified above, it requires a moment's consideration to show that integers are satisfactory. A 32-bit integer can hold counts up to 2^{32}(=4,294,967,296). If a basic block executes in a microsecond, then it would take more than an hour in that basic block before the count overflows. Therefore, integer counts are pretty safe, but for programs that summarize the data it is best to use double precision.

6.1. Counting instructions

The ideal counting instruction increments an arbitrary location in memory, changing nothing else. Few, if any, machines have such an instruction. Either the machine has condition codes, which are affected by adding 1, or some address arithmetic is needed, or the number to be incremented must be in a register, or some combination of all of these.

For the Motorola 68000 and the VAX* processors, two of the machines with counting implementations, there are instructions that increment an arbitrary integer using an address contained in the instruction stream. The only drawback is that these instructions affect the condition codes.

6.2. Condition codes

If the counting instructions affect the condition codes, then it may not be safe to insert an instruction at the beginning of a basic block. I use a simple test: if the first instruction of the basic block kills (in the charming language of flow analysis) the condition codes, then the increment instruction is inserted. Otherwise the program inserts a more complicated sequence, which preserves the condition codes around an increment. It is not always easy to find such a sequence, being somewhat tricky on the VAX machine (Kirk McKusick provided relatively simple code). One would think that a subroutine call would always suffice but on some machines subroutine calls change the condition codes.

The required trick is a consequence of processing assembly language. If the code were being inserted by compilers, the generated instructions could be chosen by mechanisms otherwise present in the compiler, rather than requiring special consideration.

6.3. Addressing

The counting code needs to add 1 to some location in memory. This requires that the inserted code be able to generate the address of the location without affecting the execution of the program. Fortunately, many machines can address all of memory from the instruction stream. If yours cannot, you may view this as an amusing challenge.

6.4. Storage for counts

The arrays for counts could be allocated globally, or for each source file, or for each procedure. The middle choice is the natural one, since files are compilation units. The program that processes the assembly language generates the space for the arrays at the end of the file when it knows how many basic blocks there are.

The counting arrays are linked together at run time (following a suggestion of Channing Brown). Special code is generated after the entry point of a procedure to check to see if the file's counting array has been linked into the list of active arrays, and to link it in if necessary.

---

* Trademark of Digital Equipment Corporation.

6.5. Span-dependent instructions

Many machines have several forms of branches varying in how far the target is from the branch instruction. Inserting counting code between a branch and its target moves them apart, so short branches may no longer reach their targets. The command `bb` changes all short branches into long ones. Of course this slows the program down, but not much.

There is a similar problem with certain special loop instructions (e.g., `aoblss` on the VAX processor) which implicitly contain short branches. These are replaced by equivalent code that includes a long branch. In both these cases the reported counts are those for the original program.

VII. GETTING THE COUNTS OUT

Before the program terminates, it must write the counts out, lest they be lost. To this end the library's standard exit routine, which flushes buffers, is replaced by a routine that flushes buffers and then appends the counts to a file named `prof.out` in the current directory. It produces the counts by scanning the linked list of counting arrays. Each array contains the full name of the file it corresponds to, its length, and the actual counts. The first two were provided by `bb` and the counts come from executing the program. The name and the counts are written on `prof.out`.

It is useful to be able to extract counts from programs that never call the system exit routine, such as the operating system kernel and various network servers. In the case of the operating system, it is easy to read the counting arrays out of the system's memory using `dev/mem`. Also, it is easy to recover the information from a system dump. On most versions of the system it is not generally possible for one program to read the memory of another, so getting counts out of a running program requires prearrangement: the program must write out the counts itself, and any way of telling it to do so is reasonable. I usually use some signal. When the program gets the signal it writes out the counts, using the algorithm described above, and then continues. If a program aborts it is not hard to extract the count arrays from the `core` file.

VIII. A SHORT EXAMPLE

Here is the program `max.c`, the interesting part of which finds the location of the maximum element of an array of length 100,000 of random integers. After looking at the code, but before looking at the statement counts, the reader might like to guess how often a new maximum is found.

```
#define N 100000
int x[N];
main()
{  int i;
   srand(getpid());
   for(i = 0; i<N;i++)
     x[i] = lrand();
   max(x, N);
}
max(v, n)
int v[];
{  int i, j;
   j = 0;
   for(i = 1; i<n;i++){
     if(v[i]>v[j])
       j = i;
   }
   return(j);
}
```

The user gets an executable program by typing `lcomp max.c`. After executing the program, the user types `lprint`, and gets the following output (the italic line numbers are not part of the output).

```
 1.      1    #define   N 100000
 2.      1    int x[N];
 3.      1
 4.      1    main()
 5.      1    {  int i;
 6.      1       srand(getpid());
 7.      1       for(i = 0; i<N;i++)
 8.  100000        x[i] = lrand();
 9.      1       max(x, N);
10.      1    }
11.      1
12.      1    max(v, n)
13.      1    int v[];
14.      1    {  int i, j;
15.      1       j = 0;
16.      1       for(i = 1;i<n;i++){
17.  99999         if(v[i]>v[j])
18.     10          j = i;
19.  99999              }
20.      1       return(j);
21.      0    }
```

The 10 new maxima are approximately what the theory predicts. The counts of 1 on the declarations and blank lines come from the next executable basic block (see Section 10.1). Thus, a blank line after line 17 would have a count of 99,999 in the output.

IX. PRINTING THE RESULTS

The program, `lprint`, prints counts. It produces output broken down by instructions, source line, function, or file. At its most verbose it will print each assembly language instruction with the number of times it was executed. By default it prints each line of the source with the number of times it was executed, as above. Because the correspondence between basic blocks, which is what are being counted, and source lines for compiled languages is inexact, these line counts need to be viewed with a modicum of understanding (see below). For intermediate amounts of detail, `lprint` summarizes by functions, or prints each line with the number of machine instructions executed. Later there are some examples of line counts. Here is an example of summary by function:

```
16779455ie     524353calls     38i     0ine      _naput
99211687ie     524353calls     80i    17ine      _naget
       0ie           0calls    31i    31ine     _nafree
    3686ie          67calls    60i     2ine     _naupdat
   91478ie        1434calls    73i     3ine      _naread
    4779ie          81calls    69i    10ine     _nawrite
     420ie          14calls    31i     1ine     _natrunc
30344908ie     523189calls     61i     1ine      _nastat
  457595ie        1333calls   368i    80ine      _nanami
72571404ie     1576004calls    107i    11ine       _send
```

The first column shows how many instructions were executed in that function. The second column gives the number of times the function was executed. The third gives the number of instructions in the compiled function, the fourth gives the number of those that were never executed, and the last column is the name of the function. The same data summarized by file are

```
219465412ie 918i 156ine 60352487bbe 255bb 59bbne neta.c
```

The new information is in columns four, five, and six. These are the number of executions of basic blocks, the number of basic blocks in the file, and the number of those never entered during execution, respectively.

X. USING THE OUTPUT

10.1. But what does it really mean?

Here is an example. The italic numbers are not part of the program's output. The code is a piece of the operating system, and the data are real.

```
1.  2204448  loop:
2.  2204448          slot = INOHASH(dev, ino, fstyp);
3.  2204448          ip = &inode[inohash[slot]];
4.  4378850          while (ip != &inode[-1]){
5.  2919642            if(ino ==ip->i_number&&dev==
                         ip->i_dev
6.  2919642              &&fstyp==ip->i_fstyp){
7.  745240               if((ip->i_flag&ILOCK)!=0){
8.  513                    ip->i_flag|=IWANT;
9.  513                    sleep((caddr_t)ip,PINOD);
10. 513                    goto loop;
11. 744727               }
12. 744727               if((ip->i_flag&IMOUNT)!=0){
13. 418411                 for(mp=&mount[0];mp<&mount
                             [NMOUNT];mp++)
14. 4509270                if(mp->m_inodp==ip){
15. 418411                 dev=mp->m_dev;
16. 418411                 ino=ROOTINO;
17. 418411                 fstyp=mp->m_fstyp;
18. 418411                 goto loop;
19. 4090859                }
20. 4090859                panic("no imt");
21. 326316               }
22. 326316               ip->i_count++;
23. 326316               ip->i_flag|=ILOCK;
24. 326316               return(ip);
25. 2174402            }
```

Note that there are some peculiarities in the output. This is the case for the `for` statement at line 13, where the first basic block, the initialization, is executed 418,411 times, while the test is executed at least 4,509,270 times, as can be seen from the next line. Also, the C compiler (at least the one used for the example) has a slightly inaccurate count of line numbers, as we can see from the large numbers on statement 20, which actually was never executed. The problem here is that the C compiler did not recognize the end of the loop until it got to that line, so the loop increment code was associated with that line. Finally, the large count on line 25 is from the first line not shown, and represents the false branch of the test at line 5.

The problem with the compiler is that there is no exact correspondence between basic blocks and statements in C (or Fortran or Pascal). While this is regrettable, the data are not randomly weird, but systematically weird, and thus are usually interpreted unambiguously. Adding curly braces frequently helps with compound statements. Also, the profiler's idea of lines is the same as the debugger's idea, so it would appear to the user, for instance, that the line after the loop is being executed each time the debugger single steps through the loop.

This part of the kernel is profiled on purpose, not just for this paper. The loop at line 13 searches a linked list, and the question is whether the ordering of the items in the list should be changed, or whether some other data structure should be used. Since the list was searched 418,411 times using 4,509,270 comparisons, and since I know that the list is usually about 16 items long, it appears that some rearranging might make a slight difference. As a side effect of the profiling, note that of the 745,240 times the test at line 7 succeeded, 513 times the resource found was locked.

10.2. Bottlenecks

Time profiling determines which routines are taking lots of time. Then count profiling, by highlighting the busy parts, gives information that explains why the routines are taking so much time. Reference 4 gives examples in which count profiling led to a speedup by factors of 2 to 4.

10.3. Testing

The next example is the body of a routine to find the square root of the number `a` modulo a prime `p`. It was run several times on random data in the hope that all the code would be covered.

```
 1.  8    extern short primetab[];
 2.  8    modsqrt(a, p)
 3.  8    {  short *x;
 4.  8       int i, j, s, t, e, u;
 5.  8       a %= p;
 6.  8       if(a<0)
 7.  0         a += p;
 8.  8       if(a == 0)
 9.  0         return(0);
10.  8       if(p % 4 == 3)
11.  5         return(mpow(a, (p + 1)/4,p));
12.  3       u = p - 1;
13.  3       for(e = 0;(u&1) == 0;e++)
14. 10         u>>= 1;
15.  3       s = mpow(a, u, p);
16.  3       if(s == 1)
```

```
17. 0         return(mpow(a, (u+1)/2, p));
18. 3       for(x=primetab+1; legendre(*x,p)!=-1;
19. 5         x++);
20. 3       for(j=0; j<e; j++)
21. 5         if (s==p-1)
22. 3           break;
23. 2         else
24. 2           s=(s*s)%p;
25. 3       s=mpow(*x, u, p);
26. 3       for(i=0; i<e-j-2;i++)
27. 2         s=(s*s)%p;
28. 3       i=(1-u)/2;
29. 3       i%=p-1;
30. 3       if(i<0)
31. 3         i+=p-1;
32. 3       t=mpow(a, i, p);
33. 3       t=(s*t)%p;
34. 3       s=(s*s)%p;
35. 3       while((t*t)%p!=a)
36. 0         t=(t*s)%p;
37. 3       return(t);
```

Unfortunately, the return at line 17 and the loop at line 36 were never tested. The trivial tests at lines 7 and 9 seem safe enough. Before I used this subroutine in a program I managed to find tests that covered all the statements. (There is still no guarantee that the program is correct, but at least all the parts have been executed.)

10.4. An application to microcomputer architecture

Dynamic instruction counting can be used to compare alternative architectures for new machines. The simplest case is that of a microprocessor with fixed-length instructions and no cache. In this case one expects that the memory bus is the limiting factor, so that the processor is either retrieving instructions, retrieving data, or storing data. Instruction counts transform into program timing directly. Of course we can't get counts from executing the program on nonexistent hardware. Instead, we write the compiler so it will produce code for an existing machine but preserve the basic block structure it would produce on the new machine. Execution counts on the existing machine then give execution counts for the new machine. In more realistic cases, of course, it requires elaborations of the basic counting technique to get all the data needed to compare architectures. Other ways of getting this information, such as simulation or instruction traces, require much more computer time.

XI. BUT WHAT DOES IT COST?

Not much. Each basic block contains one extra counting instruction, one that involves both a fetch and a store, and so is relatively expensive. Hence, the cost depends on how long basic blocks are, and they are typically short (2.54 VAX instructions for a set of several common C programs). Usually profiling costs between 50 percent and a factor of 2 in CPU time.

XII. SOCIOLOGY

This work raises an obvious question. Why not modify the compilers to insert counting code? The problems with condition codes and addressing just would not come up. Alternately, wouldn't a preprocessor, which inserts counting statements into the source be a better idea? This latter idea was implemented by Mike Lesk in a preprocessor named `vcc`, which only checked for test coverage. It is hard to insert legal statements in some contexts without doing a careful job of parsing, and if one is going to parse, why not change the compiler?

I have at least four reasons for processing the assembly language, the last of which turns out to be the most important. First, it is easy. The programs stand alone, rather than having to be inserted in the complicated compiler. The whole package, including a table of machine instructions for the VAX machine, is 993 lines of C, and the first version took about three days to write. Second, it is not restricted to C, as all the compilers put out assembly language. Third, profiling can include library routines, some of which are in assembly language. Last, it can be distributed and installed by unprivileged users, which is not true of a modified compiler. Thus, the programs have spread widely inside AT&T Bell Laboratories without any official support, or even recognition, from system administrators.

REFERENCES

1. E. H. Satterthwaite, "Debugging Tools for High Level Languages," Software Pract. Exper., 2, No. 3 (July 1973), pp. 197–217.
2. J. L. Bentley, *Writing Efficient Programs*, Englewood Cliffs, NJ: Prentice Hall, 1982.
3. Cray-1 Computer System Hardware Reference Manual, Cray Research Inc., 1976.
4. J. Fitch, "Profiling a Large Program," Software Pract. Exper., 7, No. 4 (July 1977), pp. 511–8.
5. D. E. Knuth and F. R. Stevenson, "Optimal Measurement Points for Program Frequency Counts," BIT, *13* (1973), pp. 313–22.

AUTHOR

Peter J. Weinberger, B.S. (Mathematics), 1964, Swarthmore College; Ph.D. (Mathematics), 1969, University of California at Berkeley; Bellcomm, Inc., 1969–1970; Instructor and Assistant Professor of Mathematics, University of Michigan, 1970–1976, AT&T Bell Laboratories, 1976—. Mr. Weinberger is Head of the Computer Systems Research Department. Since coming to AT&T Bell Laboratories, he has worked on databases, operating systems, networking, and compilers.