# Profiling under UNIX by Patching

*Matt Bishop*

Research Institute for Advanced Computer Science
NASA Ames Research Center
Moffett Field, CA 94035

*ABSTRACT*

Profiling under UNIX® is done by inserting counters into programs either before compiling, during compiling, or during assembly. A fourth type of profiling involves monitoring the execution of a program, and gathering relevant statistics during the run. This paper looks at this method and an implementation of it, and discusses its advantages and disadvantages.

October 1, 1987

# Profiling under UNIX by Patching

*Matt Bishop*

Research Institute for Advanced Computer Science
NASA Ames Research Center
Moffett Field, CA 94035

Keywords: profiling, patching, UNIX, execution monitor

## Introduction

There is a saying among programmers, ''[m]ake it right before you make it faster''[1]. This involves testing the program, usually by running it on some test data. But how can a programmer be sure that the test data really exercises all paths of control, so every statement is executed at least once? And once the programmer is satisfied the program is right, how can he tell in what sections of code the program spends most of its time?

Obtaining the answers to these questions require the use of a tool called a *profiler*. This tool will monitor the execution of a program, gather statistics on the program execution, and print the results in an understandable form. Among the units of a program that can be profiled are functions and source lines; one can think of these as being large-grained and fine-grained units, the idea being that one profiles the function calls to determine in which function the program spends most of its time, and then look at a source line profile of the function to determine what parts should be rewritten. Several books describe methods for using this information to improve program performance[2,3].

There are also two types of statistics that are gathered from profiling; each has its own uses. The first is timings, which give the number of seconds (or clock ticks) spent in each unit. These statistics must be read with an understanding of factors that corrupt the timings. Since instructions are usually executed far faster than one clock tick per instruction, timings are rarely exact; for example, if

Work reported here was supported by the National Aeronautics and Space Administration under contract NCC 2-387.

a subroutine is called and returns between clock ticks, the subroutine would not show up in timings. Timings also depend on things not related to the program, such as the speed of paging and what parts need to be paged in. So, while timings are a useful guide, they are not ideal. The second statistic is counts, which give the number of times the relevant unit has been executed. Counts have the advantage that they are entirely precise; but since the units being counted may vary wildly in complexity, they lack the weighting that timings provide.

Timing and counting statistics are both generated in the same way. Special instructions are placed between the units being monitored, such as function or block entry points. When the program runs, this special code increments timers or counters, and when the program ends, the information is saved somewhere. The programmer can then analyze this information to see the timings and counts that interest him.

There are four basic ways to implement profiling programs. The first is to modify the compiler to generate the special code; the second is to use a preprocessor to insert special code in the source program; the third is to use a postprocessor to insert special code in the assembly language program produced by the compiler; and the fourth is to use an execution monitor. Traditionally, UNIX profiling has been done using the first method[4]. This method has the disadvantage that one needs access to the compiler sources to implement it, and system administrators are as a rule reluctant to replace a working compiler with a locally modified one. It has the advantage that no preprocessing or postprocessing is needed to add the instructions, and issues such as handling the state of the process do not arise since the compiler will deal with them. Of late, the second method has also been used[5]. Its problems are that the postprocessor must preserve condition codes across the inserted special code, and in order to work correctly, the postprocessor must have an intimate knowledge of the target computer's assembly language. The problems with preprocessors are different; basically, preprocessors require that the program be parsed and (where necessary) rewritten to prevent the special code being inserted from causing syntax errors. These methods have the advantage that one need not modify the compiler to use them, since they are not a part of the compiler itself.

Very little attention has been paid to using execution monitors with UNIX thus far. This paper will examine the design, implementation, and experiences using such a tool. First, we shall discuss how an execution monitor works, and then describe the implementation of this tool, and some experiences with its use.

## How an Execution Monitor Works

Use of an execution monitor involves a technique called *patching*[6]. When the execution monitor runs, it starts the program to be profiled and immediately suspends it. The monitor then saves instructions at the beginning of each unit of the program to be profiled, and replaces them with instructions that will cause a fault when executed (called breakpoints.). When this is done, the execution monitor restarts the program to be profiled. Whenever a unit is reached, a fault occurs, and control is returned to the execution monitor; the execution monitor determines if the fault was caused by entry into a unit, and if so increments the counters and timers associated with that unit. It then puts back the original instruction, and single steps through the program being profiled until some other instruction is executed. The execution monitor then replaces the instruction with a breakpoint and the execution monitor restarts the program being profiled.

The technique of modifying the process space of the process being profiled is called patching. It is a very powerful technique, and is used by dynamic debuggers to enable a programmer to manipulate both data and instructions in a program being debugged. Like dynamic debugging, use of a patching technique requires the operating system to allow one process to change the instruction stream of another process in order to allow the placement and replacement of breakpoints in the process being monitored. Compilers must provide some means of associating the units being profiled or debugged with addresses in the process instruction or data space, so that the execution monitor can determine where to place the breakpoints. In essence, profiling by patching is a very simple form of dynamic debugging; so if a computer system supports any kind of dynamic debugging, execution monitors for profiling can be written. And just as with dynamic debuggers, the granularity of the counts that the profiler can provide depends on the amount of information in the symbol table of the object code of the program that is to be profiled. For example, if line numbers were not present but function names and addresses were, source lines could not be profiled but function calls could be.

Two questions about this patching procedure immediately come to mind. When the illegal instruction and the instruction it replaced are exchanged, and the traced program is single stepped, the instruction might be re-executed. If this happened, the count associated with the line would be incorrect. To avoid this error, the execution monitor must check the program counter after the single stepping. If the replaced instruction were re-executed, it increments the counter for that instruction and repeats the single stepping. When the program counter shows that some other instruction has been executed, the illegal instruction is restored.

The second question is related. Implicit in this method is the assumption that the instruction causing the fault does not change the state of the traced process, and in particular the condition codes. Usually, this is no problem since illegal instructions cause faults not reflected in the condition codes; if there is no such instruction, however, matters become far more complicated. The execution monitor should substitute three instructions rather than one:

| | |
|---|---|
| $n$ | copy condition code register to location $n + k_2$ |
| $n + k_1$ | execute illegal instruction |
| $n + k_2$ | store the former condition codes here |

Then, before allowing the program to continue, the execution monitor would have to replace the contents of locations $n$ through $n + k_2$ with what was originally there, and then restore the condition codes from location $n + k_2$. This process would have to continue until the instruction at location $n + k_2$ is passed, at which point everything can be restored as it was before the instruction at $n$ was executed.

Once the program has finished execution, the execution monitor must print the results. There are two ways to do this. The traditional method of other profilers running under UNIX has been to dump the results in an intermediate file (called *mon.out* or something similar) and provide another program to print the data there in an intelligible format. The second is to add the code to print the results to the program being profiled. The first approach provides more flexibility, because users can examine the raw data directly and combine the data produced by several runs; no doubt this is why UNIX profilers tend to use it. However, UNIX profilers work with a fairly small amount of data (namely, counts and timings of function calls) rather than with large amounts of data such as counts for each line. Moreover, for an execution monitor, adding code to make an intelligible printout adds nothing to the program being traced, since this code resides in the monitor itself. So the situation is not so clear-cut here, and in fact either method could be used with equal ease.

## An Implementation of an Execution Monitor

The execution monitor described above is being implemented in two steps, the first of which has been completed and the second of which is in progress.

The first version, described in this section, counts the number of times each source line is executed; the second version allows functions to be counted as well. The basic structure of both versions is the same; the next section describes the differences in detail. The first version runs on both VAX† and MC 68000 versions of 4.2 BSD. The second version is being implemented on a VAX running 4.3 BSD.

The first step in instrumenting the profiled program is to locate the beginnings of units to be counted within the traced program. This is done by looking at the symbol table. When a special debugging option is given, the 4.2 BSD C compiler creates symbol table entries for both source file names and line numbers, and with each line number provides the address of the first machine instruction in that line. One complication is that several line number entries may have the same address, for example if a multiline comment is present. These are loaded into an array of structures of the form

```
struct {
        union {
                unsigned t_val;         /* value in symbol table */
                ADDRESS t_tadd;         /* same, treated as an address */
        } t_lpos;                       /* where the line occurs */
        WORD t_word;                    /* the word that´s there */
        WORD t_ill;                     /* the word with illegal inst. */
        UNIT t_unit;                    /* unit being profiled */
        unsigned int t_count;           /* count from execution monitor */
};
```

The type *UNIT* contains information used to print the profile; since this version profiles line numbers only, this is defined as:

```
typedef union {
        struct {                        /* structure to hold line number */
                unsigned int tus_lno;           /* line number */
                char *tus_fnm;                  /* pointer to file name */
        } tu_lno;
} UNIT;
```

The types *ADDRESS* and *WORD* are defined to be the types of an address and a

† VAX is a Trademark of Digital Equipment Corporation.

word on the current machine; for example, on a VAX, these are

```
typedef unsigned int WORD;          /* what a machine word is */
typedef WORD *ADDRESS;              /* what a machine address is */
```

The field *t_word* will hold the word at that location, and the field *t_ill* will hold the same word but with the instruction being replaced by an illegal instruction. All lines are found in one pass over the symbol table.

The next step is to replace the instructions at the beginning of each line with the illegal opcode. In the implementation, this opcode is the opcode LDPCTX ("LoaD Process ConTeXt"[7]), which is a privileged operation (and when executed by a user's program will cause a fault) but which does not alter the condition codes after the fault. First, the process to be profiled is started after marking that it is to be traced; on the VAX, this causes a fault after the first instruction of that process is executed. At this time, words are copied from the child process' memory into the array of structures described above, and replaced with words modified with the illegal instruction at the address indicated by the line number. (Use of words rather than bytes is necessary, even on a byte-addressed machine like the VAX, because the *ptrace* call[8] reads and writes only words.)

Now, the profiled process is ready to run. It is signaled to continue, and the execution monitor waits for a fault or termination. If the child terminated, the program analyzes the results. If it faults, the execution monitor determines what signal caused the fault and where the program faulted. If the fault was not an illegal instruction, or the address is not that of a line, the execution monitor will attempt to force the child process to continue as though it had received that fault. (This usually results in that process terminating, possibly with a core dump.) Otherwise, the execution monitor adds 1 to the *t_count* fields of all lines with that address in *t_lpos*. It copies the *t_word* field of the appropriate entry in the array into the traced process' text space, and then single steps, checking each step until the instruction has been passed. The appropriate *t_ill* field is copied into the profiled program's instruction space. Now, the new program counter value must be compared to the addresses of the line numbers, lest two lines occupy less than one machine word. If this is true, the entire procedure is repeated using the new instruction and line number. If not, program execution continues.

Printing in the first version is done by the execution monitor; the user can request line counts, a full histogram, or a scaled histogram. The basic scheme is the same for all formats -- simply traverse the array of line numbers and print the counts. In all cases, the usual format is to print the counts followed by the source file lines. Here is a sample of output from this program; the program simply generates an array of 1000 numbers and sorts them using a Shell sort[9]:

```
CTRACE Version 1.3 (July 25, 1983)

FILE     LINE  COUNT
x.c         1     0:  #define MAX 100
x.c         2     0:
x.c         3     0:  main()
x.c         4     1:  {
x.c         5     1:          register int i;
x.c         6     1:          int list[MAX];
x.c         7     1:          long random();
x.c         8     1:
x.c         9     1:          for(i = 0; i < MAX; i++)
x.c        10   100:                  list[i] = random();
x.c        11     1:
x.c        12     1:          shell(list, MAX);
x.c        13     1:  }
x.c        14     0:
x.c        15     0:  shell(v, n)
x.c        16     0:  int v[], n;
x.c        17     1:  {
x.c        18     1:          register int i, j, gap, temp;
x.c        19     1:
x.c        20     1:          for(gap = n/2; gap > 0; gap /= 2)
x.c        21     6:                  for(i = gap; i < n; i++)
x.c        22   503:                          for(j=i-gap; j>=0 && v[j]>v[j+gap]; j-=gap) {
x.c        23   386:                                  temp = v[j];
x.c        24   386:                                  v[j] = v[j+gap];
x.c        25   386:                                  v[j+gap] = temp;
x.c        26   386:                          }
x.c        27     1:
x.c        28     1:  }
```

Note that the counts must be interpreted properly. For example, look at the “for” loop in lines 9−10. Even though the count is 1, the test in the "for" statement is executed 100 times; the problem is that the 4.2 BSD C compiler puts the symbol for the line number at the machine instruction generated for the initialization, and the next line number is for that of the loop. Unfortunately, fixing this would require the compiler to be changed.

## The Next Version

This version works on principles similar to the first version, but will permit functions and basic blocks to be profiled. Profiling functions is of more use than the profiling of lines and blocks, since one need not have compiled the program with debugging information, and need not have the source available. However, it requires information about how different machines handle function calls. Some, such as the MC 68000, begin at the address stored in the symbol table. In this case, the illegal instruction can be placed at the address of the function. Others, such as the VAX, begin execution at the word after the address of the function (the word at the address is used to indicate what registers should be saved, among other things). In these cases, the illegal instruction must be placed at the first word executed upon entry into the function. Profiling blocks requires information about each block to be placed in the symbol table; compilers that can do this generally only do so when debugging information is requested.

The definition of the structure used to hold profiling information and of *UNIT* in this version are a bit more complex:

```
struct {
        union {
                unsigned t_val;         /* value in symbol table */
                ADDRESS t_tadd;         /* same, treated as an address */
        } t_lpos;                       /* where the line occurs */
        WORD t_word;                    /* the word that's there */
        WORD t_ill;                     /* the word with illegal inst. */
        unsigned int t_type;            /* type of unit in this structure */
        UNIT t_unit;                    /* unit being profiled */
        unsigned int t_count;           /* count from execution monitor */
};
```

The field *t_type* contains one of three values:

```
#define     U_LINE    1         /* this structure contains a line */
#define     U_FUNC    2         /* this structure contains a function */
#define     U_BBLK    3         /* this structure contains a basic block */
```

The type *UNIT* is defined as:

```
typedef union {
        struct {                                /* structure to hold line number */
                unsigned int tus_lno;                   /* line number */
                char *tus_fnm;                          /* pointer to file name */
        } tu_lno;
        char *tu_func;                          /* pointer to function name */
        unsigned int tu_lvl;                    /* level of basic block */
} UNIT;
```

The second difference is that the user will be able to specify what lines, source files, blocks, and functions are to be profiled. One of the main problems with the first version is that a signal trap occurred on every line (this will be dramatically illustrated in the next section, when timings of the sample program are shown.) In the second version, this will only be true with the specific parts that the user wants to trace.

## Comparison of Profiling Methods

The discussion in the introduction pointed out some problems with various methods of profiling: having the compiler generate counters and timers, preprocessing programs and inserting profiling code; postprocessing assembly language output from the compiler and inserting profiling code; and using an execution monitor. The question of which method is best cannot be answered simply; to a large degree, it depends on what tools are available and what information is desired.

First, if the user wants to generate counts for each source line, using compiler-generated code is probably not an option, since most UNIX compilers do not provide such statistics. (The Berkeley pascal compiler *pxp*(1) comes close, producing statistics for blocks.) Preprocessing programs solves the problems posed by condition codes, since the compiler takes care of them; but such programs require at minimum a parser (to ensure adding the profiling statements does not produce a syntax error.) Postprocessing has the problem with condition codes, and requires a knowledge of the machine's assembly language instructions as well as the code generated by the assembler; for example, the type of branch instruction used on many machines (such as the VAX) depends on how far a branch may occur. Patching requires that one be able to read and write the address space of the process being traced, and be able to scan the symbol table of the associated program. It does not depend on any knowledge of the language or the compiler being used; in fact, the implementation described above was written to analyze programs in C, but it correctly analyzed a pascal program without even being recompiled! So from the programming point of view, patching is easier to program.

Patching allows programs to be profiled even if their source files are not available and the program has been optimized as long as the symbol table is intact; this can be useful. For example, suppose one is dealing with a program with many source files. This program is already used in production. To find the function called most often, methods for profiling would generally require recompilation; in a production environment this might pose severe problems (such as putting an unacceptably heavy load on a computer.) But by using patching one could determine the most commonly-called routine without recompiling. At that point, if a faster version of that subroutine were available, someone could decide whether to recompile the program with the new version of the subroutine. (Most likely this would be done at non-peak hours, such as in the evening.)

From the user's point of view, patching is the most flexible method but the slowest. Using patching, one can profile one section of the program, and then profile a completely different section without having to recompile the program. None of the other three methods of profiling allow this; all would require recompilation. Only patching allows any profiling without compiling special code; all other methods add code before assembly; as a result, to profile using these methods, previously compiled programs must be recompiled. While patching will only allow you to profile those units saved in the symbol table, in most cases this includes functions, which are very often the main units of interest.

Because the other three methods all add code to the program, they require additional data space (for the counters) and instruction space (for the routines or instructions that increment the counters.) This increases the size of the process image and may produce unintended side effects. Patching does not add any new code, and all data is stored in another process' image; so there is no change in size to the profiled process' image. In fact, that process cannot even detect it is being monitored without scanning and analyzing its own instruction stream!

Finally, should the profiled program terminate abnormally (say, with a bus error), other UNIX profiling packages will not allow the user to obtain a profile because the intermediate file is either not written out or corrupt. (*Gprof* generated an intermediate file, but core-dumped; *prof* did not generate any intermediate file.) Correcting this problem would not always be possible, since some events causing abnormal termination cannot be trapped (for example, the signal *SIGKILL*). An execution monitor, however, can easily determine why the profiled process stopped, and since the statistics gathered are in the process space of the monitor rather than the profiled program, the requisite statistics can be generated.

## Timings

The disadvantage of patching is that it exacts a high price in time. In addition to incrementing a counter, the monitored process faults once, has a breakpoint and another instruction replaced, and faults again, at which point the breakpoint and instruction are again replaced. Throughout all this, the monitor is executing. This results in a large increase in run time. In this section, three statistics quantify the increase in time; the first is the time spent executing user instructions, the second is the time spent executing system instructions (as in response to a system call), and the final is the total time executing the program. In the following tables, both absolute times and factors (using unprofiled programs as a factor of 1) will be given.

Table I displays the impact of profiling lines on the time; the numbers displayed are the average of ten runs of the sample program presented in an earlier section. Runs were made on a 11/730 running 4.3 BSD without profiling and using patching. Because no other utility allows this type of profiling, this table lists only unprofiled code and patched code.

Table I. Timings of Methods of Profiling Lines

| method | | times (sec) | | | factors | | |
|---|---|---|---|---|---|---|---|
| | | user | system | total | user | system | total |
| no profiling | | 0.06 | 0.14 | 0.19 | 1.00 | 1.00 | 1.00 |
| | monitor | 6.51 | 38.76 | 45.27 | 108.50 | 242.25 | 205.79 |
| patching | process | 3.98 | 36.58 | 40.56 | 66.33 | 228.62 | 184.36 |
| | total | 10.49 | 75.34 | 85.83 | 174.83 | 470.89 | 390.15 |

Notice that the time spent executing within the system (which is the time needed for system calls to complete) dominates the time spent executing the user level code in the execution monitor. The system time includes the time required to fault; that is why the process being monitored has so much system time. Thus, since there were 4336 breakpoint traps encountered (two for each number in the counts), this means that the process being profiled takes 0.01 seconds to trap on a breakpoint and the execution monitor spends 0.01 seconds servicing a trap (that is, replacing the breakpoint and instruction in the monitored process, and finding and updating the appropriate counter if necessary.)

The reader might wonder why we did not use a Pascal version of the program, and obtain timings for the profiling produced by *pxp*(1). In fact, this was done, but one of the counts provided by *pxp* was wrong! The explanation is of course simple: the pascal interpreter does some flow analysis and instruments only the beginning of basic blocks, so *pxp* profiles basic blocks and not lines. Thus, the timings would not be representative of line-by-line profiling.

Table II displays the impact of profiling functions on time spent executing user and system instructions. These timings were obtained by using ,*prof*(1) a standard UNIX profiling utility, and a primitive version of the second version of the execution monitor. As before, these numbers are the average of ten runs of the program (not the sample program above; this program contains more function calls.) Runs were made on a 11/730 running 4.3 BSD.

Table II. Timings of Methods of Profiling Functions

| method | | times (sec) | | | factors | | |
|---|---|---|---|---|---|---|---|
| | | user | system | total | user | system | total |
| no profiling | | 0.045 | 0.303 | 0.348 | 1.000 | 1.000 | 1.000 |
| profiling | | 0.092 | 0.552 | 0.644 | 2.044 | 1.821 | 1.850 |
| | monitor | 1.038 | 4.304 | 5.342 | 23.066 | 14.204 | 15.350 |
| patching | process | 0.471 | 4.204 | 4.675 | 10.466 | 13.874 | 13.433 |
| | total | 1.509 | 8.508 | 10.017 | 33.533 | 28.079 | 28.784 |

Notice these times are not so high, because there are fewer functions, and hence fewer traps, than there were lines in the previous example.

## A Wish List

Certain characteristics of the kernel impose limits on what an execution monitor can do. The major bottleneck is the system call *ptrace*, which is the mechanism used to control the execution of the profiled program. Its main problem is that only children may be controlled, and only children started up after the execution monitor has begun can be profiled. This poses several problems. First, only the parent part of a process that forks can be monitored; children are on their own. Second, it is not possible to monitor a program that is already running (such as the kernel.) Third, every signal will cause a trap to the execution monitor; it should be possible to instruct the process being profiled to treat certain signals normally rather than having the profiled program return control to the monitor. Finally, the *ptrace* mechanism is itself cumbersome and slow, and should be replaced with something more elegant and faster. Not being able to obtain timing information from a child process that has not terminated is also a problem. Were this not so, the execution monitor would be able to provide timing statistics as well as counts, by obtaining timings at each unit and subtracting. (In some cases, extra illegal instructions would need to be inserted; for example, at the end of functions as well as at the beginning.) A third useful feature would be automatically preserving condition codes when a fault occurs, and restoring them when execution resumes. This problem can usually be circumvented by choosing the instructions to place in the profiled process' text space appropriately, but it would be better not to have to worry about this at all.

Many of these features would be useful in contexts other than profiling; for example, in debugging[10]. Some manufacturers of multiprocessing machines have already made some of these changes.†

## Conclusion

Patching is a very powerful method of profiling. It allows any executable program with a symbol table to be profiled, and the more functions and source line numbers in the symbol table, the more that can be profiled. It does not rely on the existence of either assembly language source files or higher level language source files; indeed, even if the source is unavailable, the program can be profiled! Its drawback, that it causes the profiled program to run very slowly, can be

† For example, the *ptrace* system call for Dynix 2.0, by Sequent Computer Systems, Inc., will allow decendants of children to be monitored, as well as allowing running programs to be monitored[11].

ameliorated by judiciously choosing the units, and sections of code, to be profiled.

*Acknowledgements:* Thanks to the two anonymous referees whose suggestions greatly improved the first version of this paper.

**References**

1. Kernighan, B. W., and Plauger, P. J., *The Elements of Programming Style*, McGraw-Hill Book Company, New York, NY ©1974.

2. Bentley, J. L., *Writing Efficient Programs*, Prentice-Hall, Inc., Englewood Cliffs, NJ ©1982.

3. Plum, T., and Brodie, J., *Efficient C*, Plum Hall, Inc., Cardiff, NJ ©1985.

4. Graham, S. L., Kessler, P. B., and McKusick, M. K., "An Execution Profiler for Modular Programs", *Software – Practice and Experience* **13**(8), pp. 671 – 685 (Aug. 1983).

5. Weinberger, P. J., "Cheap Dynamic Instruction Counting", *AT&T Bell Laboratories Technical Journal* **63**(8), pp. 1815 – 1826 (Oct. 1984).

6. Plattner, B., and Nievergelt, J., "Monitoring Program Execution: A Survey", *Computer* **14**(11), pp. 76 – 93 (Nov. 1981).

7. –, *VAX Architecture Handbook*, Digital Equipment Corporation, Maynard, MA ©1981.

8. –, *UNIX Programmer's Manual Reference Guide, 4.2 Berkeley Software Distribution, Virtual VAX-11 Version*, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA (Mar. 1984), as reprinted by the USENIX Association.

9. Kernighan, B. W., and Ritchie, D. M., *The C Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ ©1978.

10. Himelstein, M., and Rowell, P., ‘‘Multi-process Debugging’’, *USENIX Summer 1985 Conference Proceedings*, Portland, OR (June 1985).

11. Vander Borght, John, *private communication* (September 1986).