Software

Knuth's Profiler Adapted to the VMS Operating System

R.M.Damerell

Abstract

This article describes a Pascal profiler originally written by D.E. Knuth. In principle, this should be portable to any machine. In practice it required a lot of work to adapt it to VMS. We believe that the modified profiler can now support the whole of Standard Pascal and many non-Standard parts of VMS Pascal; and that it should be more easily portable than the original. We also provide a companion utility for generating execution count files.

Introduction

This article is about a Pascal profiler which was written by D.E. Knuth and distributed with the Stanford TEX software. As there seems to be no published description, we begin by explaining how it works.

Suppose you have a program — let's call it `Snail` — that runs unbearably slowly. A *profiler* is a supplementary utility that determines how much time the `Snail` is spending in executing different portions of its code. What usually happens is that a typical `Snail` will spend nearly all of its time executing a small subset of itself. Such a subset is usually stigmatised by such names as "bottleneck", "critical section", "innermost loop", etc. Any serious attempt to speed up a `Snail` must concentrate on this "critical" section, either by actually rewriting it to run faster or by rewriting the higher-level code to make it run less often, or maybe adopting an entirely new algorithm. Nothing else is likely to make any significant difference. Thus a profiler is an essential tool for any programmer who is concerned about the execution speed of his or her programs.

Most profilers work by making some special calls to the operating system, asking it to monitor the behaviour of the `Snail` in some way as it crawls. Some typical examples are given in [1,2,6]. Knuth's profiler (called "`Profile`") works on an entirely different principle. It reads the source code of `Snail`, making a table of the time consumed by each statement. For each statement in the code, `Profile` estimates the time to be $w * f$ where:

- w, the *weight*, is the time taken to execute the statement once,
- f, the *frequency*, is the number of times the statement was executed in a run of the `Snail` program.

`Profile` then prints a listing of the `Snail` program with weight and frequency data added. The weight of each statement is estimated by parsing the statement, making reasonable assumptions about how it might be executed on a typical machine. If `m` and `n` are integers, the Pascal statement: `x:=2.1*(m+n);` would probably be executed as: fetch `m` and `n`; add; convert to real; multiply; deposit result in `x`. The costs of all these primitive operations are stored as constants in `Profile`. `Profile` adds them all together to get the weight, then multiplies by the frequency to get the total cost. The frequency is read in from a supplementary file called a *count* file. This contains a long list of numbers; essentially it lists the number of times every statement in `Snail` was executed in a trial run.

Thus it appears that `Profile` does not require any special help from the local operating system; so in principle it should be runnable on any machine. In practice it is a very different story. The main obstructions to running `Profile` on a new machine are:

1. No mechanism is provided for generating the count file.
2. All Pascal compilers implement different languages — of the same name!

We have been working on the problem of installing `Profile` on the VMS operating system, with the ulterior aim of eventually producing a portable version of `Profile`. This article describes the progress made so far. In order to avoid confusion, we call the altered program "VMS-`Profile`", reserving "`Profile`" for Knuth's original.

Generating the Count File

`Profile` was originally written for the KL-10 machine at Stanford, on which D.R. Fuchs altered the system debugger to make it generate a count file. This is obviously not a practical option for other users. Hardly any manufacturers provide the source of their software and few site managers would allow ordinary users to alter it. Even if we could alter the VMS debugger, it could not be distributed as this would be a breach of copyright.

We have therefore written a completely separate utility called Preprofile for generating count files. Preprofile reads the source code of the Snail program and generates a new program file with a name like SNAIL_COUNT.PAS. If all goes well, this will be a valid Pascal program, which does everything that Snail does and also writes a count file, called SNAIL.COU. This can then be fed into VMS-Profile along with the original SNAIL.PAS file.

So in order to profile a program on VMS, you need to do the following: compile and link VMS-Profile and Preprofile; define commands to run them; run Preprofile on Snail; compile and link Snail_count; define further commands to run Snail_count instead of Snail. Then put the SNAIL.COU file into the same directory as the source of Snail and run VMS_Profile on Snail and (with luck) you get a profiled file called SNAIL.PRO.

The basic algorithm of Preprofile is fairly obvious. At each place in the Snail program file where Profile will need to see a count, Preprofile inserts a piece of code to advance a counter. Roughly speaking:

```
while ⟨condition⟩ do ⟨statement⟩
```

becomes

```
while ⟨condition⟩ do begin
    count[i] := count[i]+1 ;
    ⟨statement⟩ end;
```

In the outermost block of Snail, Preprofile must declare all the extra variables. At the start of the statement part of the Snail program, Preprofile inserts code to set all the counters to zero. At the end, it inserts code to open the count file, write all the accumulated counts, then close it.

This mechanism now seems to be working, on all the Snail programs that we have tried. The most obvious disadvantage is that the Snail_count program will clearly run even more slowly than the original Snail did. The extra time is not itself all that important, because with luck you never need to run Snail_count more than once. The real disadvantage of the extra time is that Snail_count will never produce any useful information unless it can be run to completion. Another problem is that all the extra variables that Preprofile inserts into the Snail program must have names different from all the variables that were there previously. We have not managed to solve this problem; the best we can do is to give the extra variables unpronounceable names like "ZQRWHZ3XX" which do not figure prominently in most programmers' code.

Preprofile is a much simpler program than Profile. Profile has to parse the Snail program in great detail, but Preprofile is interested only in those syntax words of Pascal that affect the flow of control in Snail. It turned out that many of the most complicated parts of Profile could be replaced by a routine that merely copies parts of the text to the output file.

Improved Output

We have made several changes to VMS-Profile to try to improve the usefulness of its output. First consider the index of module names. Profile is designed to work with the Stanford WEB system. (We assume that everybody is familiar with WEB; see [4] if not.) As TANGLE assembles a WEB program, it inserts markers into its output like {123:}...{:123}, indicating the start and end of the replacement text of each module. If Profile sees these markers, it assumes that Snail was originally a WEB program and generates an index. For each module that contains executable code, Profile calculates the total cost of all the statements in that module. It also calculates the cost of each module as a percentage of the total cost of the whole program.

This index of modules is essential. The output of Profile is inevitably bulky, and without an index it would be a hopeless task to wade through an enormous listing in search of the critical sections. But Profile only makes an index if it sees WEB-style module markers. Therefore we have altered VMS-Profile to make it build an index of functions, in addition to Profile's index of modules. (From now on "function" will include "procedure".) VMS-Profile calculates the cost of each function both as an absolute amount and as a percentage of the total cost.

We have made minor changes to the format of the index, to improve its signal-to-noise ratio. Since the percentage costs calculated by VMS-Profile are inevitably inaccurate, we see no point in giving them to 6 decimal places. Also we list only those modules or functions that score at least 2% of the total cost.

We have also altered the way VMS-Profile lays out the Snail program. The main output of Profile is the whole of the Snail program, with weight and frequency data attached. This is arranged in columns like this:

⟨*statement*⟩ ..... ⟨*weight*⟩ ⟨*frequency*⟩

In VMS-Profile, we moved the weight and frequency columns to the left hand side. This change seems ridiculously trivial, but is actually important.

The original layout has the disadvantage that the statement has to fit into a fixed width. When the statement is indented, the width is further reduced. The effect is that `Profile` imposes a limit of $62 - k$ on the length of quoted strings in the `Snail` program, where k is the current amount of indentation. This is illogical because `Profile` is supposed to work with `TANGLE` and `TANGLE`'s limit is 69. If this limit is violated, `Profile` stops with a fatal error. VMS-`Profile` allows a much larger limit; if it sees an over-long string it merely splits it, so the output is essentially undamaged. The new layout means that a statement can now spread out to any width; the output file is much shorter because it does not need so much padding; we can add a column for $weight * frequency$ (which is the data the user actually needs).

One of the methods that Knuth used for debugging TEX was the TRIP test [5,3]. This is a special input file containing many unusual constructions, intended to exercise the entire TEX program. He found that this is a powerful device for revealing obscure bugs in a program, after the obvious bugs have been fixed and the program seems to be working. In order to help with this method of debugging, VMS-`Profile` prints a list of the line numbers of executable statements that did not get executed in the trial run.

The Many-Languages Problem

This problem is compounded by the fact that `Profile` uses a rather simple-minded top-down parsing algorithm. It is well known that such parsing methods do not work well on programs that have syntax errors. In theory this should not matter because `Snail` has to be working before it makes any sense to try to profile it. In practice, `Profile` runs into trouble as soon as you try to move it to another machine, say from Machine A to Machine B. Every construction in B–Pascal that is not in A–Pascal is seen by `Profile` as a syntax error. The usual result is that after a little while, `Profile` becomes totally confused and loses track of the boundaries between statements in the `Snail` program. It is therefore essential to adapt `Profile` to read B–Pascal before it can be used on the new machine.

Of course we can always try to make ad-hoc changes to `Profile` to support this or that feature of the new language, but this approach produces masses of bugs. Even with a `debug_help` procedure (based on the one in TEX) it is a difficult business to adapt `Profile` to a new system. We believe that we have managed to make VMS-`Profile` support the whole of Standard Pascal and many of the more accessible features of VMS Pascal. But we can never be sure that we have succeeded. There is always the danger that some unexpected (but perfectly valid) combination of Pascal syntax will reveal another bug. We believe that it will require a great deal of effort to produce a satisfactory solution of the many-languages problem. Meanwhile, neither `Profile` nor VMS-`Profile` can be regarded as portable.

The following examples will show some of the difficulty. Consider what happens when `Profile` reads a variable declaration, say

```
var horse, dog, goat: real;
```

`Profile` scans the list of names, then the type, then it sets up structures in its memory so that it will in future recognise a "`horse`" when it sees one. Now suppose that `horse` was previously declared in an outer block. Then `Profile` again does the obvious thing: it saves the previous definition of `horse` on a stack. When the current block is exited the previous definition will be un-saved. Now suppose the definition came in a procedure header, say:

```
procedure hunt(horse, fox: integer; yak:
        real);
```

Then `Profile` again knows what to do: it first defines the parameters `horse`, `fox`, and `yak`: then it defines the procedure itself.

All this is quite straightforward in principle; the details are not necessary here. Now consider: what must `Profile` do when it reads the word "`forward`"? If `horse` was defined in an outer block, that definition must be recovered from the stack. But also the new definition of `horse` as a parameter of `hunt` must be saved somewhere so that `Profile` will know what to do with `horse` when scanning the definition of `hunt`. This definition cannot be saved on the stack because the current stack frame will be erased by the time we reach the definition of `hunt`. It follows that we have to assemble an entirely new structure to represent a procedure header in order to handle forward declared procedures before they have been defined.

In VMS Pascal the formal parameters of functions can have default values. In the previous example, suppose that `horse` and `fox` had been declared with default values. Then when the function is called you can omit any parameters with defaults and pass the others by explicit assignment, as in: `hunt(yak:=4)`. The library procedures of VMS Pascal make extensive use of this feature.

The VMS versions of TEX and METAFONT use parts of the VMS system library. In order to handle these programs VMS-Profile must read the library header file (called "starlet.pas"). This file is a monster, nearly three times as large as TEX.PAS. We had to increase the size of all the arrays in VMS-Profile to accommodate all the data; in turn this forced us to use long numbers to address these arrays because 16-bit numbers are not large enough.

It is clear that adapting Profile to another machine is not just a simple matter of adapting its system-dependent procedures to the eccentricities of a new compiler. Many of the internal structures have to be redesigned. These structures are represented by linked lists. It is terribly easy for list-processing programs to become messy, and messy programming is utterly abhorrent to the spirit of the WEB language. It is an accepted convention that any respectable WEB program must contain a clear explanation of how it is supposed to work. There seem to be two main difficulties that must be overcome in order to write a clean program that does list-processing. First, it is impossible to specify the structure of a complicated list in words. We need an easily-readable notation. We have therefore included in VMS-Profile the beginnings of a suite of TEX macros for this purpose. These macros are no use for complicated lists; even so, they make a valuable contribution to the clarity of VMS-Profile. The Appendix at the end shows some examples.

The second main difficulty of list-processing is that Pascal has no suitable primitives; so every operation needs half-a-dozen statements. We have therefore written a set of WEB macros for simple list operations.

Although the many-languages problem is unsolved, we have managed to solve a small part of it. VMS Pascal provides a great many non-Standard predeclared functions. Some of these have weird syntax. The most extreme example is the open procedure, which links a disk file to a Pascal file variable. This procedure is both complicated and important; it is difficult to imagine how any serious programmer in VMS Pascal could avoid using it. Its declaration is something like this:

```
procedure open(file_variable:file;
  file_name:S_typ:='';
  history:H_typ:=new;
  record_length:integer:=132;
  access_method:A_typ:=sequential;
  record_type:R_typ:=variable;
  carriage_control:C_typ:=list;
  organization:O_typ:=sequential;
  disposition:D_typ:=save;
  file_sharing:W_typ:=none;
  function user_action:integer:=none;
  default:S_typ:='';
  error:E_typ:=message); extern ;
```

where S_typ can be any character string type and H_typ, etc., are enumerated types whose values are here immaterial. (The true definition of open is even more complicated than this simplified paraphrase suggests; it seems to be impossible to express this in Pascal.) In order to handle these predeclared functions, VMS-Profile must assemble suitable structures in memory to represent their headers. It would be an unbearably long and error-prone task to do this by hand. The only tolerable method is to write the declarations of these functions into a file and make VMS-Profile read them before it reads the Snail program itself.

For this purpose, we use the pool file mechanism of the WEB language. This is a most valuable feature of WEB which deserves to be far more widely used than it is at present. We have yet to see any large Pascal program that could not be improved by judicious use of this mechanism. It was invented by Knuth to circumvent the difficulty that Standard Pascal has no satisfactory mechanism for handling character strings. When TANGLE is assembling a WEB program, if it reads a string in double quotes, it writes that string into a supplementary file called a *pool* file. The idea is that the Tangled program can then read all these strings from its pool file into its memory. So we insert all the predeclarations into the VMS_PROFILE.WEB file. When VMS-Profile starts up it reads the pool file before it reads the Snail file. Here are some sample definitions:

```
declare("const true=1;false=0;",
  "maxint=2147483647;minint=-maxint;",
  "minchar=0;maxchar=255;",
  "type boolean=false..true;",
  "integer=minint..maxint;",
  "char=minchar..maxchar;",
  "text=file of char;")
```

The declarations are written in the usual Pascal form; they may extend over several lines and each line must be enclosed in double quotes. Then declare must be called on these lines. Several lines may be declared at once; then they must be separated by commas to keep TANGLE happy. Procedures and functions must have just the header, followed by "extern". For comparison, here is part of the equivalent code from Profile:

```
char_loc:=get_avail;
info(char_loc):=char_type;
int_loc:=get_avail;
info(int_loc):=int_type;
p:=get_avail; link(int_loc):=p;
q:=get_avail; val(q):=-max_int;
info(p):=q; q:=get_avail;
val(q):=max_int; link(p):=q;
bool_loc:=get_avail;
info(bool_loc):=int_type;
p:=get_avail; link(bool_loc):=p;
zero_loc:=get_avail; val(zero_loc):=0;
info(p):=zero_loc; one_loc:=get_avail;
val(one_loc):=1; link(p):=one_loc;
id5("f")("a")("l")("s")("e")
   (bool_const)(0);
id4("t")("r")("u")("e")(bool_const)(1);
p:=get_avail; val(p):=max_int;
id6("m")("a")("x")("i")("n")("t")
   (int_const)(p);
id7("i")("n")("t")("e")("g")("e")("r")
   (defined_type)(int_loc);
id7("b")("o")("o")("l")("e")("a")("n")
   (defined_type)(bool_loc);
```

And here is how it all works. `Declare` is a `WEB` macro with no replacement text. When `TANGLE` reads a `declare`, it first evaluates the argument. As this is a string in double quotes, it copies the string into the pool file. Then it evaluates the `declare` and solemnly puts nothing into the Pascal file. Then VMS-`Profile` reads the pool file and parses the declarations as if they were part of the `Snail` file itself. Where functions use non-standard syntax (like `write` and `open` above) we have made some ad-hoc changes to VMS-`Profile`'s parsing routines to support these.

We believe that this mechanism is much cleaner than the previous one, as you can actually read the declarations. It does have one unfortunate consequence; in order for VMS-`Profile` to allow for the execution time of these predeclared functions, we must specify these times. This is done using `Profile`'s "change-weight" mechanism. Recall that the weight of a statement is the estimated time to run it once. If `Profile` gets this wrong, you can rectify this by adding a so called "change-weight" comment, which looks like this: `{+100}`. This means "add 100 units to the cost of the current statement". In VMS-`Profile` you can add change-weight comments to external declarations, thus:

```
declare("function sin(x:real):real;",
          "extern{+100};")
```

If this comment is seen, then the number is assumed to be the cost of the function. In this respect VMS-`Profile` is inferior to `Profile`, because in `Profile` all the costs are tidily collected together in one place. We think the improvement in clarity outweighs the price.

Future Developments

The current version of VMS-`Profile` contains several problems besides those mentioned above. The first concerns the accuracy of the calculated profile. Ideally, when moving `Profile` to a new machine, one ought to calibrate it by measuring the time taken to do all the primitive operations and writing these times into the table of costs in the program. This would be a tremendously long and messy job, and probably not worth doing. Given that VMS-`Profile` works by examining the source of `Snail`, it cannot possibly have as close a contact with reality as a profiler that actually monitors the crawling `Snail`. On the other hand an accurate profile is neither needed nor possible. Any modern operating system is doing several jobs at once; so the time taken for a given task will vary according to the burden of other tasks. If a profiler gives a useful result, that result will be that "function `X` is using 10 times as much time as everything else". Since the truth is inevitably fuzzy, we believe that any calculated profile within a factor of 2 is probably good enough for practical purposes.

When discussing `Profile`'s index, we said that `Profile` produces a list of all the modules in `Snail` and their total costs, and slurred over the question of how this is actually done. There are two ways of doing this. If `M` is a module, then its *explicit* cost is defined as the total cost of the statements actually contained in `M`. But `WEB` modules may be nested to any depth. So we can also define the *implicit* cost of a module. The implicit cost of module `M` is the total cost of the statements in `M` and also all modules directly or indirectly included in `M`. Roughly speaking, the explicit cost of a module is the amount of time you might save by rewriting its code to run infinitely fast; the implicit cost is what you might save if you could bypass that module altogether. `Profile` lists all the modules in `Snail`, giving both their explicit and implicit costs.

When VMS-`Profile` calculates its index of functions, it can only calculate explicit costs. The explicit cost of a function `F` is the (estimated) amount of time used by the code that is actually part of `F`, ignoring any time used by functions called by `F`. The only way we could estimate an

implicit cost of F would be by assuming that every invocation of every function uses the same time. (This assumption is clearly false, but VMS-Profile has no way to get more accurate information.)

Suppose that function F calls function G q times. Then we must add $q * c/n$ to the implicit cost of F, where c is the cost of G and n is the total number of times G has been called. This simple-minded approach fails when functions call one another recursively. In order to find implicit costs, VMS-Profile would have to solve a set of linear equations. It is easy to prove that the matrix of coefficients is nonsingular but ill conditioned. We have not tackled the problems of assembling these equations or of finding a suitable method for solving them.

In conclusion, we believe that Knuth's profiler is potentially a useful program, but it cannot realise its full potential until it is made portable. Copies of VMS-Profile and Preprofile have been submitted to the archives at Aston, with a suggested directory name "[tex-archive.utils.vms-profile]". They may be freely copied, "as is", on condition that no warranty is expressed or implied.

References

1. M. Bishop, *Profiling under UNIX by patching*, Softw. Pract. Exp., **17**, 729–739, (1987).

2. T. Cargill and B. Locanthi, *Cheap hardware support for software debugging and profiling*, Computing Architecture News, **15 (5)**, 82–83, (1987).

3. D.E. Knuth, *The Errors of TEX*, Softw. Pract. Exp., **19**, 607–686, (1989).

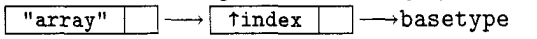
4. D.E. Knuth, *Literate Programming*, Comput. J., **27**, 97–111, (1984).

5. D.E. Knuth, *A torture test for TEX, Stanford Comput. Sci. Report STAN-CS-1027*, Nov. 1984.

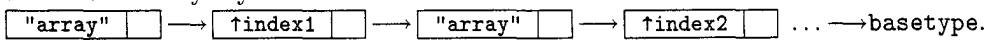
6. B. Plattner and J. Nievergelt, *Monitoring program execution: a survey*, Computer, **14**, 76–93, (1981).

Appendix 1

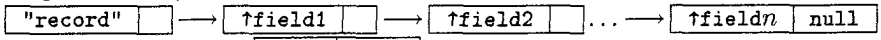
Here we give some examples of the linked-list macros mentioned earlier. There are some errors of mis-alignment, which we regard as not worth fixing. A Pascal array type is represented by the structure:

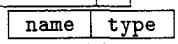


and multi-dimensional arrays by:

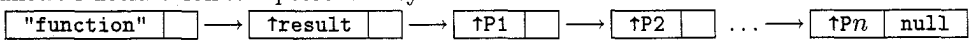


A record type is represented by

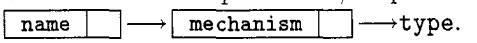


where each pointer fieldi points to 

Finally, a function declaration is represented by



where P1, etc., correspond to the parameters. For each parameter, P1 points to:



While a list is being built, it looks like this:

This structure is non-intuitive, but it works. The chief booby-trap is that you must remember to remove or bypass the leading cell before starting to extract data from the list.

Appendix 2

This is the source for Appendix 1, with most of the plain text deleted. First, the underlying macros:

```
% This one puts a box around its argument; based on the
% 'control sequence token' macro in TeXbook

\def\cstok#1{\leavevmode\thinspace\hbox{\vrule\vtop{\vbox{\hrule
        \hbox{\vphantom{\char124}\thinspace{\ninett#1}\thinspace}}
\hrule}\vrule}\thinspace}
```

```
% Partitioned boxes for linked lists

\def\abox#1{\cstok{\hskip0.5em#1\hskip0.5em}}
\def\dbox#1#2{\cstok{\hskip0.5em#1\hskip0.5em$\char124$\hskip0.5em#2\hskip0.5em}}
\def\leftbox#1{\dbox{#1}{}}

\def\TO{$\longrightarrow$}
\def\m{\hskip0.5em&\hskip-0.5em}
\def\^{\char'013}
\def\d{$\downarrow$}
\def\u{$\uparrow$}

% Pascal arrays:

\centerline{\leftbox{"array"}\TO\leftbox{\^index}\TO {\tt basetype}}

\centerline{\leftbox {"array"}\TO\leftbox {\^index1}\TO
\leftbox {"array"}\TO \leftbox {\^index2} \dots\TO {\tt basetype}.}

% Record type:

\centerline{\leftbox {"record"}\TO\leftbox {\^field1}\TO
\leftbox {\^field2}\dots\TO\dbox {\^field$n$}{null}}
\noindent where each pointer {\tt field$i$} points to \dbox {name}{type}.

\noindent Finally, a function declaration is represented by

\centerline{\leftbox{"function"}\TO\leftbox{\^result}\TO
\leftbox{\^P1}\TO\leftbox{\^P2} \dots\TO\dbox{\^P$n$}{null}}

For each parameter, {\tt P1} points to:

\centerline{{\tt P1}\TO  \leftbox{name}\TO \leftbox{mechanism}\TO {\tt type}.}

% List structure:

$$\vbox{ \baselineskip=12pt
\+\TO\m\leftbox{}\TO\leftbox {A}\TO\leftbox {B}\TO\leftbox {C}\dots\TO
 \m\dbox{W}{null}\cr \+&\d&\u\cr \vskip-9pt \+&\hrulefill&\cr }$$
```

◇ R.M.Damerell
Maths Dept,
Royal Holloway & Bedford New College
Egham, Surrey, U.K.
Janet: uhah208@uk.ac.ulcc.pluto