DEBUGGING PL/I PROGRAMS IN THE MULTICS ENVIRONMENT Paper 225

by B. L. Wolman (617) 491-6300, Ext. 215

Honeywell Information Systems 575 Technology Square

Cambridge, Massachusetts 02139

The problems associated with debugging a program written in PL/I are simplified in the Multics system for a number of reasons. PL/I is the standard language used for programming in Multics. The Multics PL/I compiler is complete, has few restrictions, and produces efficient object programs. A variety of powerful debugging commands make use of a run-time symbol table generated by the compiler thereby allowing the user to debug his program symbolically. Statistics about the operating characteristics of a program may be easily and accurately determined.

(85 words)

INTRODUCTION

One of the popular misconceptions concerning PL/I is that programs written in PL/I are necessarily inefficient and hard to debug. Several years experience with the Multics PL/I compiler running on the Honeywell 645 has shown that in spite of the apparent complexity of the PL/I language, PL/I programs are easily debugged in the Multics environment, even by novice users who are newcomers to PL/I and are unfamiliar with the Honeywell 645. In most cases the user can debug his program symbolically without having to refer to a listing of the generated instructions or add debugging output statements to the program. This is due to a number of factors:

- the run-time environment provided by the system.
- the implementation of PL/I.
- the availability of a variety of powerful debugging facilities.

THE ENVIRONMENT

The use of PL/I as the principal tool for programming by users of Multics was envisioned at the very start of the project. Features which are required by PL/I such as a stack, pointer variables, conditions, and a recursive call/return mechanism are all provided and are directly supported by the system hardware and/or software. Consequently, the basic Multics environment is ideally suited to the needs of PL/I programs. In fact, nearly all

of Multics itself is coded in PL/I and executes in this self-maintained environment.[1-5]

The Multics system currently provides the user with a virtual address space of over 1000 segments of 65536 words each (some changes now in progress will increase the maximum size of a segment to 262144 words). Access to these segments is by means of PL/I pointer variables which contain a segment number, a word offset, and a bit offset. There is a direct correspondence between PL/I pointers and virtual addresses in Multics; PL/I pointer values may be loaded into the addressing registers of the 645 by a single machine instruction. An attempt to use a pointer whose value is the PL/I null pointer causes a condition to be signalled.

The PL/I stack is maintained for each user as a series of contiguous frames (block activation records) within a single segment. A register is dedicated by the system to point at the stack frame of the procedure being executed. Multics defines a system-wide standard call/return sequence which is relatively efficient. Stack frames can be obtained and released by executing a few instructions.

Procedure segments in Multics are normally pure and sharable. Access to procedure and data segments is set by Multics access control commands and checked by the hardware at each instruction and data reference. If a user does not have appropriate access to a segment, or if any other error such as an attempt to divide by zero happens, a machine

fault occurs. This fault is turned into a PL/I condition (e.g., "accessviolation" or "zerodivide") and is signalled by the PL/I condition mechanism. All but a few catastrophic errors are handled in this manner.

Multics provides a default error on-unit which is invoked if the user has not established an on-unit for a specific condition. In most cases, the default on-unit prints an appropriate error message (which may include information as to probable causes for the error) and calls the command processor to read a command from the user's input stream. The stack chain of calls leading up to the fault is preserved; in many cases the user's program can be restarted.

In Multics there is no real difference between a command and a program written by the user: both are PL/I procedures. Any program written in PL/I following command argument conventions may be invoked as a "command".

When the user types a command line of the form

```
edit alpha beta
```

the Multics command processor searches a specified set of directories for a procedure named "edit" and issues the equivalent of the PL/I statement

```
call edit("alpha","beta");
```

The procedures found in the system directories are the "commands" and utility procedures normally available to Multics users. Since the user can change the search rules

used by the system, he can tailor his own command set if he chooses.

THE IMPLEMENTATION

The implementation of PL/I in Multics is particularly complete and has few restrictions.[6,7] The only omission of any consequence is tasking. The Multics implementation allows:

- arbitrary pointer qualification including chains of locators and use of functions as qualifiers.
- adjustable data with no restrictions. Arrays may have any number of adjustable bounds. Structures may have any number of adjustable members.
- operations on aggregates.
- functions which return values whose length or bounds are not known at the time the call is made, e.g., returns(char(*)) or returns((*) fixed bin).
- entry variables.
- recursive procedures at no extra cost.
- full stream and record I/O.
- all data types including complex and decimal.

Since the implementation is so complete, the programmer does not have to worry about what features are or are not available to him. The ability to use the full language reduces the amount of code the user has to debug by increasing the amount of work handled by the run-time support system provided by the compiler.

The Multics PL/I compiler produces efficient object code, even when measured against the best efforts of experienced hand coders using assembly language. The availability of a compiler which generates efficient programs greatly reduces the user's desire to want to switch to assembly language for reasons of efficiency. This is particularly important in Multics because of the richness of the machine instruction set (512 instructions and 64 types of address modification) and the complexity of the system environment from the view point of an assembly language coder.

Multics PL/I makes use of a separate "operator segment" which contains assembly language coding for about 50 commonly used functions such as string moving, complex multiplication, and the index operator, as well as tables of constants for masking, shifting, storing characters, etc. This segment is shared by all PL/I programs. Communication with the operator segment is by means of a work area in a standard position in each stack frame. The operator segment is entered by a short sequence of instructions which loads certain machine registers with parameters and then jumps directly into the operator segment at a known location. The use of the operator segment reduces the cost of PL/I programs by reducing their size and by reducing paging activity.

If a begin block or internal procedure block does not declare any automatic variables with adjustable bounds or sizes and can only be entered by first entering its parent block, then the block is said to be "quick". The Multics PL/I compiler does not use a separate stack frame for such blocks. Instead, they share the stack frame of their parent block. The overhead of calling a quick block, exclusive of the cost of preparing the argument list, is only three instructions: one each at call, entry, and return. The cost of a quick procedure is also reduced because automatic storage in the parent block can be addressed directly.

The availability of a really inexpensive mechanism for internal procedures means that users can write them without having to concern themselves with efficiency. The artifice of using label variables and goto statements so that a block of code can be executed efficiently from a number of places is not necessary.

The compiler makes no restrictions on the format of structures. This is important, since programmers can choose a structure description that is appropriate for the problem they are trying to solve without having consider its acceptability to the compiler. However, it is possible for a user to specify a structure which causes the compiler to generate very expensive accessing code. There are a few "common sense" rules users can follow if they are concerned about the efficiency of their programs.

Extensive error checking is done during compilation; there are nearly 500 possible error messages. Except for a few cases of multiple, related errors within a single statement the Multics PL/I compiler normally finds most errors in a single run. It is infrequent that a user will correct a set of source errors and recompile his program only to receive another batch of error messages. Errors are reported on the user's console as they are discovered; the printed message normally includes the source for the offending statement.

The listing generated by the compiler is designed to be printed by a high-speed line printer but is formatted so that items of interest to the user can be easily located in the listing segment by inspecting it with an on-line editor. The user can control the amount and level of detail of information placed in the listing.

DEBUGGING FACILITIES

Multics provides a number of special commands which aid user debugging. There is a powerful breakpoint debug command, a facility for tracing procedure calls, and tools which help the user determine the operating characteristics of his programs. There are several options that the user can specify when he uses the PL/I compiler to cause it to generate additional information for use by debugging commands. Of these, only the "profile" option causes any change in the code generated by the compiler.

The Run-Time Symbol Table

The PL/I compiler and the system debug command cooperate to allow the user to debug his program symbolically. The compiler normally generates a run-time symbol table only if "get data" or "put data" statements are used in the source program. The compiler can be instructed, however, to generate a "full" symbol table which includes all identifiers in the source program.

Each entry in the run-time symbol table describes an identifier in the user's program giving its name, storage class, location, size, bounds and other information needed to access the identifier. Information is available about the block in which the identifier is defined as well as its relationship to other members of the structure to which it belongs.

The run-time symbol table facility is much more powerful than it needs to be to support just data directed I/O.

- Parameters, defined, and based variables can all be represented in the table. When a variable is declared based on a specific pointer, e.g., "dcl a based(p)", information is kept which allows the address of that pointer to be obtained at run-time.
- The size, offset, bounds, multipliers, or virtual origin of any identifier can be any arbitrary

expression. This is necessary for the representation of based variables.

- References to identifiers in the user's program from data directed input or from requests to the system debugger need not be fully qualified. The same algorithm used by the compiler to resolve partially qualified names is also used by the support program which searches the run-time symbol table.

The run-time symbol table is generated at the end of the object segment and is shared by all users of the segment. If it is not used during execution, there is no overhead required to support it: the pages it occupies will not be brought into core memory; no code is required to initialize it. After the program has been debugged, the run-time symbol table can be eliminated from the object segment without having to re-compile it.

The compiler will also generate a "map" of the object program when a full symbol table is requested by the user. This map is a table, placed at the end of the object segment, giving information about the location in the object segment of each source statement. The availability of this table means that the user can refer to his object program by source line number, e.g., to set a breakpoint at a specific line number. Similarly, the system debugger can tell him the line number corresponding to a given location in the

object program. In fact, as is demonstrated in Figure 1, the debug command can print the source line that corresponds to the object location.

The Debug Command

The command "debug" can be invoked at any time; for example, after an error condition has been signalled for which no on-unit exists. It may also be called directly from the user's program. It accepts requests from the user for actions such as examining some location in the virtual memory or printing a trace of the chain of calls in the user's stack. It is aware of the different PL/I data types, so variables in the object program may be displayed in the format appropriate to their type.

When a program has been compiled with a run-time symbol table, the user can refer to it symbolically, either with identifiers defined in the program or by the line number on which a statement begins. For example, if the user's program was dealing with a two-dimensional based array of integers, he could change one of the elements in the array by entering the request

```
p -> x(i+5,j-2) = 3
```

which takes the form of a PL/I style assignment. The addresses of "p", "x", "i", and "j" would be obtained from the symbol table. Any of the identifiers in this example could be part of a structure.

The debug command can also be used with PL/I programs when a run-time symbol table is not available. In this case, the user must refer to the compilation listing of his program in order to determine the location at which a variable is stored or at which a given statement starts.

The debug command has other features which let the more experienced user examine or alter the values in a machine register or display the status of the machine at the time a fault occurred. These facilities are not normally needed if a symbol table is available.

The debug command also lets the user set conditional or unconditional breakpoints in object segments. When the breakpoint instruction is executed, the debug program gains control. If the condition associated with the breakpoint is satisfied, a message is printed; at this point the user can enter requests to debug. One of the actions available is to continue execution from the point of the break. The user may associate with each break a set of debug requests which are to be automatically executed whenever the break is encounted; thus, for example, the user might use the break mechanism to "insert" a (very simple) PL/I assignment statement into his program. There is a mode of execution available with debug which lets the user run his program one PL/I statement at a time.

An object program may have more than one break set in it; similarly, more than one program may have active

breakpoints. Facilities are available in debug for listing and altering breaks. Setting a break involves changing the object program, so breakpoints remain active until explicitly removed by the user. Breakpoints should not be used when other users are sharing the segment.

There is an "escape" facility which causes debug to pass the line typed by the user to the Multics command processor instead of treating it as a request. This is a very powerful feature since it allows the user to invoke any series of Multics commands (or any of his own programs) without having to leave the debug command. He could, for example, run a special program to display the values of the static variables used by the program he is trying to debug. If he did not have such a program, he could input it, compile it, and test it while preserving the complete status of the program he was originally debugging.

The ability to "escape" back to the full Multics system to execute any series of commands is generally available in any command such as the editor that interacts with the user. As is shown in Figure 2, the "hold" command may be used to preserve the execution environment after a fault.

The Trace Command

The command "trace" lets the user monitor all calls to a specified set of external procedures. Trace modifies the standard Multics procedure linkage mechanism so that whenever control enters or leaves one of the procedures

specified by the user, a debugging procedure is invoked. The arguments given to the debugging procedure by trace enable it to obtain the values of the the arguments and return point of the procedure being called. The user can also provide his own debugging procedures instead of the one supplied as a default by the tracing package.

The action taken by the default trace debugging procedure is to print a message on the user's console whenever control enters or leaves one of the procedures being traced. There are a number of options which the user can specify to request such actions as printing the arguments (at entry, exit, or both) or stopping (at entry, exit, or both). The user can control the frequency with which the tracing message is printed, e.g., every 100 calls after the 1000th call. He can also specify the maximum recursion depth he wishes to see. The user can also request that the tracing message be printed only if the contents of some specified location in the virtual memory has changed. The default trace debugging procedure "stops" the execution of the user's program by calling the debug command; this makes all of the facilities of debug available to the user. An example of the use of trace is presented in Figure 3.

The user may start tracing a procedure at any time, even it has already been executed. Tracing may be removed at any time; subsequent calls of the procedure will execute normally. Any procedure which uses the standard Multics

calling sequence may be traced without interfering with other users who may be sharing the segment.

Determining Program Efficiency

The two debugging packages debug and trace which we have just discussed help the user find errors which prevent his program from running properly. There is another class of errors which are much harder to find. These are usually flaws in the program design (or perhaps in its implementation) which cause the program to run correctly but to take much longer to execute than it should. Simply locating the largest statement in the program or the biggest procedure is not sufficient to locate the causes of program inefficiency because that statement or procedure may be executed only once; the real offender may be some small statement which gets executed very frequently. Without detailed knowledge of program flow during execution, instruction counts alone are not much good.

The cost of executing a specified procedure, either for a single call or a total of many calls, can be determined by using the "meter" option of the trace command. This causes trace to read the system clock when control enters or leaves the specified set of procedures. The clock counts in microsecond steps, so high resolution is possible.

Once a procedure has been found to be inefficient, its operating characteristics can be examined by re-compiling it with the PL/I "profile" option.[8] This option causes the

compiler to generate in the internal static data area a table which contains an entry for each statement in the source program; the table entry contains information about the source line as well as a counter which starts out as zero. Each statement in the program is modified to start with an instruction to add one to the counter associated with the statement.

After running a program compiled with the "profile" option, the user can determine the number of times each statement in the program was executed. The table entry contains the raw cost of the statement measured in instructions, so the user can determine both the absolute total cost for the statement as well as its relative cost compared to other statements.

A number of different tools have been developed for presenting the information available in the profile table. Figure 4 shows the source for a small procedure printed by a program which computes the percentage of the total time spent in each statement. Figure 5 shows the same profile information presented in another format.

The paging characteristics of a program can be measured by using the "page trace" facility. The Multics paging mechanism maintains a buffer for each user in which the system records the segment number, page number, and time of occurrence for each of the last few hundred page faults taken by the user's process. A command is available which

formats the information kept by the system.

DIFFICULTIES

As might be expected, there are problems associated with debugging PL/I programs in Multics. Most of these problems are minor and have the effect of requiring the user to know more about the internal workings of Multics than he might otherwise have to know.

The most difficult problem occurs when a program in the user's process commits an error so severe that the system cannot continue running the process. An example of such an error is using up the entire stack segment (perhaps because of unlimited recursion). When the system detects an error of this magnitude, it prints a message such as:

Fatal Process Error. Out of bounds fault on user's stack.

and creates a new process, thereby erasing all information about the old process.

This type of error can be very difficult to find, because no information is available to the user about where it occurred. Future versions of Multics will alleviate this problem by allowing the user to retain information about the old process. The system will also be changed to detect when the user is near the end of his stack; when this occurs, a special "stack" condition will be signalled.

COMPARISON WITH OTHER WORK

PL/C [9] and the IBM Checkout Compiler [10] are approaches to the problem of debugging PL/I programs in which a special

compiler is used during the debugging phase. Extra checking is done at run-time to catch programming errors such as the use of undefined variables. No particular effort is made to generate good object code since it is assumed that the program will be re-compiled with a production compiler after having been debugged with the special compiler.

An advantage of this approach is that a great deal of information about the original source program may be preserved at run-time, thus allowing good diagnostics. A debugging compiler can often check for errors whose detection would be intolerably expensive for a production compiler, e.g. a mismatch between a based variable and the object identified by the pointer value. The Checkout Compiler allows the user to make incremental symbolic additions to his program, a very desirable feature.

A disadvantage of using a special complier is that two compilers are involved in the debugging process and therefore two sets of compiler bugs. Another disadvantage is that meaningful figures on program performance are hard to obtain.

Multics provides a single PL/I compiler which is used by all programmers, whether novice or expert. Extra checking (other than that defined as part of the PL/I language) is not done at run-time. The run-time symbol table and the map of the object program let the user refer to his program symbolically. Since a production compiler is

being used, accurate figures on program performance are available.

A "program" in Multics often consists of a number of separately compiled procedures; the Multics PL/I compiler, for example, consists of 181 procedures comprising over 137,000 instructions. Because of the poor run-time performance normally available with a special debugging compiler, it is doubtful whether such a large collection of procedures could be successfully implemented using a debugging compiler. Since a special compilation is not required for their use, the Multics debugging tools debug and trace may be successfully used in finding bugs in production software. Even if a module could be re-compiled with a debugging compiler, the resulting object program would not be the same as the one which failed.

EXDAMS[11] is a powerful debugging tool which uses a pre-processor to modify the original source program before compilation. Calls to special monitoring procedures are inserted at points of interest in the program. During execution a record is kept of the complete execution history of the program. This allows the programmer to easily determine the point at which a given variable changes, for example. This sort of debugger would be useful, even in Multics, when a program is first being debugged; its usefullness is limited by the fact that a special compilation is required.

Evans and Darley [12] discuss source language debugging of higher-level languages. They present a number of principles which they believe are important. The Multics debugging commands satisfy most of their criteria:

1. The user has flexible control over the execution of his program. The program may be run in steps which range from a single procedure call, through a single statement, down to a single instruction.
2. The data being operated on may be examined and altered at any time and this may be done in the PL/I notation.
3. The conventions of the debugging language are to a large extent designed to minimize typing. (It is only fair to point out that the Multics debug command has been accused of being overly terse.)

The area in which Multics falls short of the features desired by Evans and Darley is the lack of the facility for incremental compilation.

ACKNOWLEDGMENTS

The Multics PL/I compiler was designed and implemented by R. A. Freiburghouse, the author, G. D. Chang, and J. D. Mills; significant contributions were also made by P. A. Belmont, P. A. Green, and A. C. Franklin. The Multics debug command was written by S. H. Webber. The trace command was written by the author. Many other members of the Honeywell and M.I.T. staffs, notably M. B. Weaver, D. Bricklin, and

D. P. Reed, have made important contributions to easing the process of debugging PL/I programs in Multics.

REFERENCES

1 E I ORGANICK

The Multics System: An Examination of its Structure

MIT Press Cambridge Massachusetts 1972

2 A BENSOUSSAN C T CLINGEN R C DALEY

The Multics Virtual Memory: Concepts and Design

Comm ACM 15 5 May 1972 pp 308-318

3 R C DALEY J B DENNIS

Virtual Memory, Processes and Sharing in Multics

Comm ACM 11 5 May 1968 pp 306-312

4 F J CORBATO' J H SALTZER C T CLINGEN

Multics - The First Seven Years

AFIPS Conf Proc 40 1972 SJCC AFIPS Press 1972 pp 571-583

5 Multics Programmers' Manual

Honeywell Document AG90-93 1972

6 R A FREIBURGHOUSE

The Multics PL/I Compiler

AFIPS Conf Proc 35 1969 FJCC AFIPS Press 1969 pp 187-199

7 R A FREIBURGHOUSE

The Multics PL/I Language

Honeywell Document AG94 1972

8 D E KNUTH

An Empirical Study of FORTRAN Programs

Stanford University Computer Science Department Report CS-186

9 H L MORGAN R A WAGNER

PL/C: - The Design of a High-performance Compiler for PL/I

AFIPS Conf Proc 38 1971 SJCC AFIPS Press 1971 pp 503-510

10 IBM System/360 Operating System: PL/I Checkout Compiler

IBM form number GC33-0003 1971

11 R M BALZER

EXDAMS - EXtendable Debugging and Monitoring System

AFIPS Conf Proc 34 1969 SJCC AFIPS Press 1969 pp 567-580

12 T G EVANS D L DARLEY

On-line Debugging Techniques: A Survey

AFIPS Conf Proc 29 1966 FJCC AFIPS Press 1966 pp 37-50

Figure 1

The PL/I condition mechanism is used for most errors, including those defined by Multics. In this example, the program generates a fault by looping until it runs off the front of the stack. The default error on-unit prints the location at which the fault occurred (100 in blowup) and the location being referenced (-1 in the stack). The program was compiled with a run-time symbol table, so the Multics debug command may be used to print the source for the line in which the fault happened. The request syntax accepted by debug is designed to minimize typing: the request specifies segment blowup, location 100 in the text section, and source line output. The value of a variable may be obtained merely by typing its name; the response gives the address of the variable (450 in the static data segment) as well as its value (-1209).

Figure 2

When a fault occurs, the complete status of the executing program may be preserved. The "hold" command causes Multics to retain the chain of stack frames (block activation records) up to the current frame until the user issues an explicit "release" command. In this example, the user inputs and compiles a small procedure to fix up the loop index that caused the bounds violation in the example of Figure 1. The program blowup is reactivated by a non-local transfer of control to the external label variable and completes normally. The same change of the loop index and re-start of blowup could also be done using only the debug command.

Figure 3

The flow of control in to and out of any external procedure may be monitored with the Multics debugging procedure trace. In this example, trev is a driver program which calls procedure rev to reverse the words in a string specified by the user when trev is called. rev is coded as a recursive procedure; it contains a bug which causes infinite recursion. The "fatal error" occurs when there is no room left in the stack segment for a new frame. The reason for the infinite recursion becomes obvious when trace is used.

Figure 4

The execution profile of a Shell sort routine after having sorted the descending sequence 999, 998, ..., 0 into ascending order. Each statement is labelled with the percentage of the total execution time spent in that statement. The profile tells us that the algorithm is quite good since unnecessary interchanges were not often done.

Figure 5

Another presentation of the execution profile of the procedure shown in Figure 4. The cost is measured in number of instructions executed.

```
/*      print blowup.pl1

        blowup:   procedure;

        dcl       (j,a(10)) fixed binary;
                  loop_index fixed binary external static,
                  recovery_label label variable external static,
                  sysprint file;

                  recovery_label = thru;

                  do loop_index = -1 to -100000 by -1;
                       j = a(loop_index);
                       end;

        thru:     put skip list("loop index = ",loop_index);
                  put skip;
                  end;

        r 2127  2.205

*       pl1 blowup table
        PL/I

        WARNING 307
        The variable "a" has been referenced but has never been set.
        r 2128  5.516

*       blowup

        Error: out_bounds_err by blowup|100
        referencing stack|777777
        r 2129  2.474

*       debug
*       /blowup/100&t,s
                       j = a(loop_index);
*       loop_index
           450       -1209
*       .q
        r 2131  3.544
```

Figure 11-- A simple example of one of the debug commands.

```
blowup

Error: out_bounds_err by blowup|100
referencing stack|777777
r 2134  1.057

hold
r 2134  .211

edm fix.pl1
Segment not found.
Input.
fix:      procedure;

dcl       loop_index fixed binary external static,
          recovery_label label variable external static;

          loop_index = 12345;
          goto recovery_label;
          end;

.
Edit.
w
q
r 2135  1.789

pl1 fix
PL/I
r 2135  3.732

fix

loop index =              12345
r 2135  1.724
```

```
print (trev rev).pl1

trev:     proc(string);

dcl       string char(*) unal,
          rev entry(char(*)) returns(char(32) varying);

          put skip list(rev(string));
          put skip;

          end;

rev:      proc(string) returns(char(32) varying);

dcl       string char(*);

          i = index(string," ");
          if i = 0 then return(string);
          else return(rev(substr(string,i)) || " " || substr(string,1,i));

          end;

r 2131  4.164

trev "now is the time"

Fatal error. Process has terminated. Out of bounds fault on user's stack.
New process created.
r 2131  3.712

trace rev
r 2131  .578

trev "now is the time"
Call 1 of rev from trev|117
ARG  1 = "now is the time"
Call 2 of rev from rev|106
ARG  1 = " is the time"
Call 3 of rev from rev|106
ARG  1 = " is the time"
QUIT
r 2132  2.428
```

```
time_profile shell

Profile of shell

LINE PERCENT STATEMENT

   1       shell:    proc(x);
   2
   3       dcl       x(*) fixed bin;
   4
   5       dcl       (i,j,k,d,t) fixed binary;
   6
   7    .0           d = hbound(x,1);
   8
   9    .1 down:     d = 2*divide(d,4,17,0) + 1;
  10
  11    .1           do i = 1 to hbound(x,1) - d;
  12    .8                k = i + d;
  13
  14 12.7 up:             j = k - d;
  15
  16 63.3                 if x(j) > x(k)
  17                      then do;
  18    .2                     t = x(j);
  19    .3                     x(j) = x(k);
  20    .4                     x(k) = t;
  21                           end;
  22
  23 15.8                 if j > d
  24                      then do;
  25   3.0                    k = j;
  26   3.0                    goto up;
  27                          end;
  28    .6                end;
  29
  30    .1           if d > 1 then goto down;
  31    .0           end;
r 1047  4.596
```

```
print_profile shell

   LINE     STM        COUNT         COST          PROGRAM

                                                   shell
      7       1            1            4
      9       1            6           30
     11       1            6           66
     12       1          500         1500
     14       1         7767        23301
     16       1         7767       155340
     18       1          234          234
     19       1          234          702
     20       1          234          702
     23       1         7767        31068
     25       1         7267         7267
     26       1         7267         7267
     28       1          500         1000
     30       1            6           24
     31       1            1            1

TOTAL                                228506
r 1048   3.461
```