9/2?

LBL-11290 c.2

Lawrence Berkeley Laboratory
UNIVERSITY OF CALIFORNIA

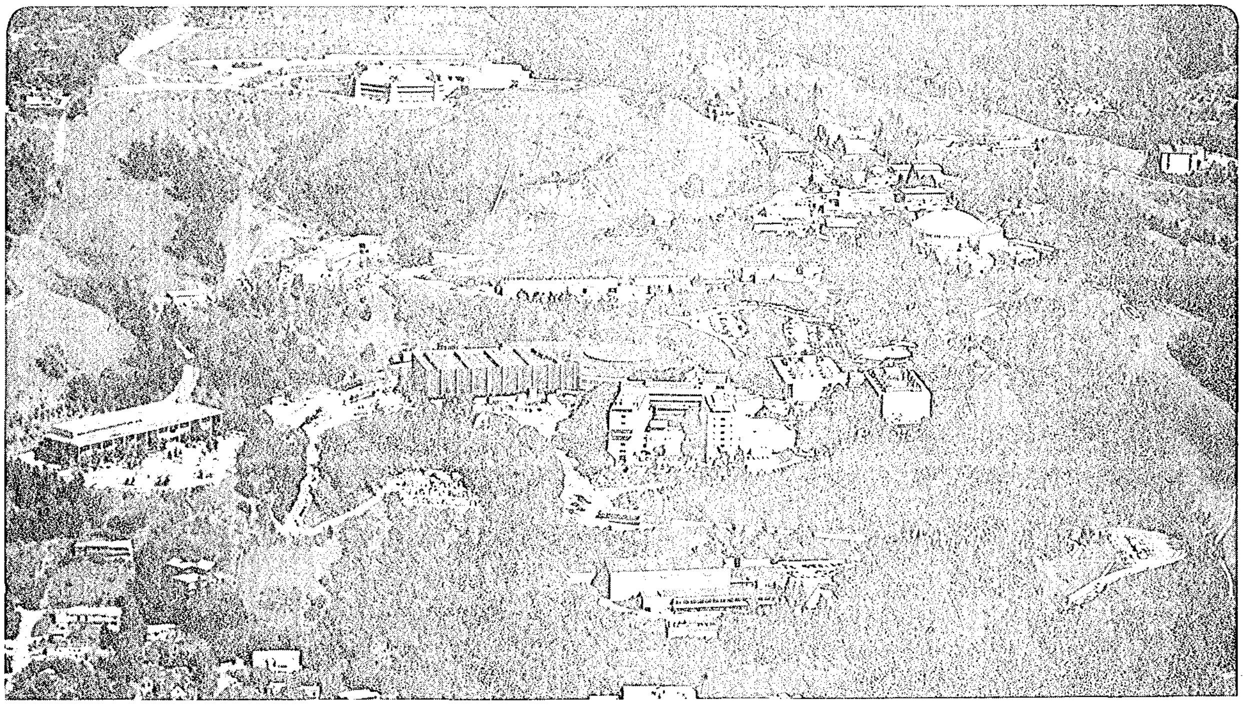
Engineering & Technical Services Division

Presented at the VIM-33/ECODU-30 Control Data User Group Joint Conference, University of Manchester, Manchester, England, September 22-25, 1980

AUTOMATIC PROGRAM TIMING PROFILES WITH FTN4

Richard Friedman

September 1980



Prepared for the U.S. Department of Energy under Contract W-7405-ENG-48

LBL-11290 c.2

LBL-11290

Automatic Program Timing Profiles With FTN4

Richard Friedman

Computer Center
Lawrence Berkeley Laboratory
University of California

Presented at the VIM-33/ECODU-30
CONTROL DATA User Group Joint Conference

September 22-25, 1980

University of Manchester
Manchester, England

ABSTRACT

Design of a scheme for producing execution timing profiles of FORTRAN programs automatically is proposed with a recommendation to implement it as an option to the compiler. An experimental implementation on the LBL 7600 is also described.

Prepared for the U.S. Department of Energy
under contract W-7405-ENG-48

LBL-11290

Automatic Program Timing Profiles With FTN4

Richard Friedman

Lawrence Berkeley Laboratory
University of California
Berkeley, California

September, 1980

Finding where a program is spending most of its time is the very first step in any attempt at optimization. Unfortunatly, this information is not always available or even predictable. Rather than waste programmer time optimizing parts of a program that are relatively less time consuming than others, FORTRAN source-level optimization is better directed when an accurate execution timing profile of the program is available. This paper speculates on how such timing profiles could be produced by optional compiler-generated code, and describes an experimental implementation using the FTN4 compiler on the BKY 7600. It concludes with a suggestion to CDC to consider implementation of a program timing option for their current compilers.

1. Profile Definition and Requirements:

By "automatic program timing profile" we are referring to a process whereby a program in execution is clocked in such a manner that on program termination a report can be generated that gives information about the time spent in various parts of the program. A requirement is that this be done automatically, without any changes to the source program; e.g. by compiler option.

In this paper we will be considering timing profile methods that measure the time spent in subprogram units, possibly including library routines. In a later section, we will discuss an extension to blocks of statements within subprograms.

Currently, the only method available to the CDC FORTRAN programmer using FTN4 or FTN5 to obtain timing information about particular program has been manual insertion of explicit calls to the library routine SECOND. In many cases, especially with very large source codes, this is too problematic, and already assumes knowlege of the behavior of the program. An automatic method, where the compiler inserts the necessary timing apparatus directly into the object code, is much more reasonable than manual, selective, program modification.

The basic scheme were are considering here, then, is one where the program is compiled with a special "profile" option selected, the resulting object binary executed, and, on program termination, a report generated automatically by a library routine giving, for each subprogram called:

- o number of times called
- o total time accumulated in that subprogram, expressed both in seconds, and as a percent of the total program execution time
- o average time per call in that subprogram

In addition, there might be some estimation of the overhead involved in doing the timing, and other statistics. Figure 1 shows the output from our experimental profile procedure currently running with the FTN 4.8 compiler on the 7600 at the Lawrence Berkeley Laboratory (BKY).

2. Design Considerations:

Some of the basic design considerations are:

- o No changes to the source code (e.g. calling a special library routine) are needed.
- o The overhead involved in the timing process is minimized.
- o Running with the "profile" option selected does not impose any serious restrictions and works with OVERLAY as well as with segmented programs.

We will now speculate on the mechanics of the program execution profile.

Basic Scheme:

With the profile option selected, the compiler generates a bit more elaborate code for each subroutine CALL, FUNCTION reference, and library call made via the RJT -- Return Jump with Traceback -- FTNMAC macro. Before a subprogram is called, the new code must "turn off" the "clock" for the calling routine, and "turn on" the "clock" for the called routine. Then the usual RJT is issued. On return the code then turns off the called routine′s clock, and turns back on the calling routine′s clock.

By "clocking" external references in this way we can get an accurate picture of program′s timing profile. The timing results are disjoint since at any one moment only one clock can be running. Also, by clocking at the RJT rather than at the ENTRY and RETURN points, we can time calls to routines not compiled with the profile option selected -- e.g. library routines, in particular I/O library routines. (There are some sticky problems here, to be discussed later.)

First we must define what a "clock" is, and how they get turned "on" and "off".

Clocks and Clocking:

For every routine that gets called in this manner there is an entry in a clock table. The clock table resides in the main program and is fixed length, though an option should be available to set its size. The clock table origin is an entry point in the main program, to make access to the table direct. Each clock entry is four words:

```
    word 1 - subprogram name
    word 2 - call count
    word 3 - accumulated time
    word 4 - start time  -or-  zero
```

The first time a routine is called an entry is established for that routine in the clock table. Turning on a clock consists of setting the start time and incrementing the call count. Turning off a clock consists of computing elapsed time (time now minus start time) and adding this to the accumulated time, and clearing the start time. A clock is considered "running" if word 4 is non-zero.

On program termination, the FORTRAN library resident (FORSYS) calls a report generator that reads the clock table and produces the profile table. Routines that are not called during the execution of the program will not have entries in the clock table, and therefore will not appear in the profile. Likewise, routines not called via the RJT macro (such as the math routines SIN, COS, etc., which are called by an RJ without traceback and have very fragile contexts) will not be timed separately. Their execution timings will appear as part of the timing for the calling routine.

The CALL Statement

Now we can fill in some of the details on how a CALL statement would be compiled with the profile option selected. (Note that although we will be discussing the CALL statement throughout, what follows also applies to FUNCTION references, and to implicit library references such as READ, BUFFERIN, etc. that utilize the RJT macro.)

The CALL statement breaks down into the following steps:

1. Get the current elapsed time.
2. Get pointer to caller's clock.
3. Turn off caller's clock.
4. Get pointer to called's clock.
5. Turn on called's clock.
6. Now issue RJT to called routine.
7. (Return) Get the current elapsed time.
8. Get pointer to called's clock.
9. Turn off called's clock.
10. Get pointer to caller's clock.
11. Turn on caller's clock.

The pointers to the calling and called routines' clocks can be allocated by the compiler. For example, CALL SUBR would define (once) a symbol >SUBR in a special USE block in the calling routine reserved for clock pointers. Initially these pointers would be zero. There would also be a pointer for the calling routine's clock. The "get pointer to clock" operation would then consist of:

1. Fetch >name
2. If zero, call clock-table manager with name. Return with address of clock for name and store in pointer >name

The table manager would be called with the name of the routine whose clock address we need. This library routine would search the clock table for a match on the name. If found, the address of the entry is returned. If the entry is not found an entry is created at the next available space in the table. This initial entry has the name of the routine in word 1, and words 2,3,4 zero. The address of this entry is then returned. (The table manager also checks against running out of table space.)

Using this scheme, the table is searched only once per unique call from each routine. (For overlay and segmented programs, the table will have to be searched again each time a new copy of the routine is loaded.)

The main program starts this process by establishing its clock entry and turning its own clock on.

Of course, there are some significant special cases to worry about. These have to do with overlays, alternate entry points, alternate returns, calling routines that are formal parameters, and call-chains that intermingle profile-compiled and not profile-compiled routines:

Alternate Entry Points:

An interesting problem arises with subprograms with more than one entry point. A unique clock will be defined for each of the entry points called during execution. If nothing is done about this, the profile report generated at program termination will be misleading. Also, should this routine subsequently call another routine, it must know which entry point was called so that it can turn off the appropriate clock before leaving the routine, and turn it back on again upon return.

One solution is to compile code at each additional entry point that asks the table manager routine to mark the clock for the alternate entry point name with a pointer to the main entry's clock. The report generator can then take care of the multiple-entry clocks later. The profile report would have the correct total time listed under the main entry point, but each multi-entry point would be marked as being a part of the main routine.

The problem with subsequent calls from this routine can be solved by forcing each entry point in multi-entry point routines to always redefine the pointer to the routine's clock with the name of each entry point.

```
SUBROUTINE SOUR
        ....get pointer to clock SOUR
        ....and store in >SOUR
ENTRY GRAPES
        ....mark GRAPES as alternate for SOUR
        ....get pointer to clock GRAPES
        ....and store in >SOUR
CALL SORRY
        ....turn off clock pointed at
        ....by >SOUR; turn on clock SORRY
```

It seems necessary that the total time spent in the routine be listed with the main entry point name, and information be given showing how many times each alternate entry point was called, and the amount of time spent in the routine when entered this way.

Alternate Returns:

A basic assumption in this timing scheme is that all calls return to the instruction following the calling RJ. This, unfortunatly, may not always be the case. Subprograms defined and called with (non-standard) alternate returns, non-standard use of ASSIGN'ed GOTO, and certain I/O library routines may return to arbitrary locations and foul the timing sequence surrounding the CALL. This will leave the called routine's clock running. The calling routine's clock will not be turned back on. A subsequent attempt to turn off the calling routine's clock will find it already off. Likewise, a subsequent call to the previously called routine will find that clock still on.

Under these circumstances assistance from the compiler is needed. To insure the proper sequence of turning on and off clocks, the compiler will have to insert return code (steps 7 thru 11 above) following each possible alternate return point in the calling routine. This is particularly important with FTN5's END=sn and ERR=sn.

We note that this problem would not occur if the profiling scheme was organized around subprogram entry and exit points rather than CALL statements. In the entry/exit scheme, clocks are turned off and on upon entry to the routine, and then switched back upon return. The CALL statement then does nothing new except pass to the called routine the pointer to the caller's clock. The only problem, as was noted earlier, is that with this scheme only those routines compiled with the profiling option will be timed. Timing CALL statements provides more information than entry/exit timing, and is probably more useful as a result.

Overlays:

The problem with overlays is that they are entered via the library subprogram OVERLAY. By itself, the time spent in an overlay would then appear in the profile as time spent in the subprogram OVERLAY, without any further differentiation. However, the compiler can help here by issuing conditional entry/exit code for overlay main programs.

What is needed is a way to detect the main program of an overlay higher than the (0,0). This can be accomplished by requiring that all (0,0) main programs compiled with the profile option have at least one file (OUTPUT) defined on the PROGRAM header card. This seems reasonable since we would expect the profile report generator to write directly to the file OUTPUT.

The only possible main program having no files defined would then be an OVERLAY. This situation can now be detected by the entry/exit sequence for the main program. The conditional code would turn off the clock for the library routine OVERLAY and turn on the clock for the overlay's main program. On return back to the calling overlay, the exit code must remember to turn off the clock and turn back on the clock for OVERLAY.

<u>Formal Parameter CALLs:</u>

When a subprogram is called with the name of another external subprogram as an argument, it is not possible to properly time the subsequent call to this routine. For example, if routine ONE calls TWO with THREE as an argument to TWO, TWO's subsequent call to THREE cannot be timed:

```
SUBROUTINE ONE(..)
EXTERNAL THREE
CALL TWO(THREE)
   ...
END
SUBROUTINE TWO(EXT)
EXTERNAL EXT
CALL EXT                ..cannot time call
```

The call cannot be timed because subroutine TWO is presented with an address of the entry point of THREE and not its name. The clock table would have to be expanded to include entry addresses for this to work, which leads to real problems in overlay and segmentation environments. Another possibility would be to assume that word entry.address-2 contains the traceback word for the routine, and hence the subprogram name, but this is not always true (especially for library routines or user-written COMPASS routines).

The CALL in routine TWO above would be able to detect that the call to EXT is a call to a formal parameter since this information is passed on to the

RJT macro already. In this case, the RJT code should assemble as the basic, un-profiled RJT. In the profile, the time spent in "EXT" would be accumulated as time spent in the calling routine, in this case TWO.

Intermingled Executions:

Typical applications of the profiling option would probably involve compilation of a set of source level FORTRAN routines which will call FORTRAN library routines and possibly user library routines that were not compiled with profiling. In the simple case where a profiling routine calls a non-profiling library routine there is no trouble, as we have seen. Since we are timing CALL statements, the time spent in the library routine is properly timed. Any routines subsequently called by the library routine are not specifically timed and their execution is included as time spent in the first routine.

As long as the "CALL-chain" from the profiling-compiled routine to the non-profiling set of routines is consistent, there is no problem. However, should the chain be broken by a call to a profiling-compiled routine which subsequently calls another routine (of any flavor), we have a problem with clocks left running:

```
A ==> b ==> C ==> d         (A,C profiling; b,d not)
  :     :     :--- Turn off C, on d clocks
  :     :          RJ to d
  :     :          off d, on C
  :     :
  :     :--- no timing code generated
  :
  :--- Turn off A, on b
       RJ to b
       off b, on A
              -- clock C left running...
```

When C calls d in the above example, the clock for C is already off. Turning off a clock that is already off will corrupt the accumulated time count (acc.time = acc.time + now.time - start.time , but start.time = 0). Therefore, the "turn off clock" procedure must detect this situation. An error diagnostic could be issued identifying the clock and call in trouble. Subsequently, the programmer could elect to compile the routine separately, putting the call in un-profiling mode.

Alternatively, the off-clock procedure could just ignore turning off clocks that are already off. A subsequent call to a routine with a clock still running leads to the comparable state of turning on a clock that is already on. By the time this situation is detected it is probably too late to issue a diagnostic.

Timing Accuracy and Estimating Overhead:

How accurate a timing is really needed? If the primary purpose of the profile report is to identify which routines absorb the most execution time, perhaps the only important information is the relative execution times expressed in percent of total program time. However, if the timing error is not uniformly distributed, even the relative timings could be inaccurate.

Two factors can introduce serious errors: 1) the resolution of the system clock used by the timing code, and 2) the overhead involved in the timing process.

On the 7600 we have found it necessary to utilize the monitor function that returns the job's CPU elapsed time in cycles rather than the standard function that returns milliseconds. With a machine cycle time of 27.5 nanoseconds the millisecond clock is far too vague. (In fact, we have redefined the FORTRAN library function SECOND to use this cycle-time clock, returning cycles*27.5E-09) Words 3 & 4 in the clock table are integer cycle counts in our 7600 FTN4 implementation.

Clearly then, the times now obtained by the CALL statement code outlined in our basic scheme above includes the time spent fetching clock pointers, turning on and off clocks, and requesting the current time from the system. It is really not an accurate picture of the actual execution time of the program in the non-profiling mode. This overhead is a function of the number of times the program is CALLed. In fact, the overhead time spent before the actual RJ to the called routine (steps 1 thru 5) appears as part of the execution time of the called routine, while the overhead time spent upon return from the called routine (steps 7 thru 11) appears as part of the execution time of the calling routine.

There seem to be two ways to try to minimize or eliminate this overhead in the timings expressed in the profile report. The first would be to estimate the overhead involved in the two parts of the timing operation and subtract it from the appropriate timings. There is some randomness involved due to the table referencing that occasionally must be done. A statistical estimation could be computed beforehand and this "fudge factor" multiplied by the number of times a routine is called, subtracted from the gross elapsed time. Also, another fudge factor is needed to reflect the time spent making CALLs. Consequently, the number of CALLs issued by a routine must be tallied and multiplied by some average time per call factor, and this subtracted from the gross elapsed time for each routine as well.

```
net.xeq.time =
        gross.xeq.time - entries * f1 - calls * f2

   where: entries = number of times routine is called
          f1 = "entry phase fudge factor"
          calls = number of calls made from routine
          f2 = "exit phase fudge factor"
```

The other approach might be to bracket out the overhead sequences by doubling the number of time requests made to the system. The CALL sequence would now look like:

```
1.  Mark TIME1
2.  Turn off CALLer's clock using TIME1
3.  Find CALLed's clock
4.  Mark TIME2
5.  Turn on CALLed's clock using TIME2
6.  Issue RJ to CALLed routine
7.  (Return) Mark TIME3
8.  Turn off CALLed's clock using TIME3
9.  Find CALLer's clock
10. Mark TIME4
11. Turn on CALLer's clock using TIME4
```

Here the only unaccounted for time is the few cycles involved in turning on a clock with a new time value (one store instruction). The disadvantage is the doubling of the number of system calls per subprogram CALL, which may seriously degrade the throughput (and cost) of the running program. This could be particularly disastrous in the case of programs that typically make 100,000 or more subprogram calls.

Estimating the overhead seems to be the least painful way of taking care of the problem, but how accurately can this be estimated, and how good will the numbers in the profile be? This seems to be a very open question.

3. FTN4TRA -- An Experimental Implementation:

In response to a request from one of our users to provide a profile timing feature with FTN4 similar to the "flowtrace" option available with the CFT compiler on the CRAY-1 computer, we came up with FTN4TRA.

FTN4TRA is a callable procedure that compiles source programs using FTN4's E option, generating COMPASS, which is subsequently assembled using a modified FTNMAC macro text file, TRAMAC. TRAMAC redefines the RJT macro (CALL statement) to do the profile timing described above. Also, there are expanded definitions of the macros used to defined program/subprogram entry points.

Not all the special cases and problem situation described in this paper are fully handled (this is, after all, just an experiment). However, even with its limitations the package has already proven itself useful as a tool in attempts at optimizing large and complex programs.

FTN4TRA has some interesting practical features. A procedure call to FTN4TRA is used in place of the control card call to FTN. The options available are:

```
   CALL,FTN4TRA,key=param,....
```

T= Timing option. If T=0 is specified, subroutine calls are only counted. No timing is done. This is intended for the initial use of FTN4TRA to determine which routines are called too many times to make timing practical. If not specified, full timing is done on all subprogram calls.

N= Size of clock table. Default is 200 clocks. Word size of table is 4*N (2*N if T=0 spcified). Small programs can save memory space by setting N to an appropriate size. If N is exceeded during execution, the program is terminated and the profile produced of the execution up to that point.

X= "IGNORE" directives file (see below). If not specified, no IGNORE directives are expected. If a file name is given, one logical record is read. IGNORE directives provide a means to indicate which routines are not to be timed when called.

Other strictly FTN4 options (I=, L=, OPT=, LCM=, etc.) are passed directly to the compiler.

IGNORE Directives:

Typically, a program is first compiled using FTN4TRA with T=0 to discover if any routine is called an excessive number of times (100,000 or more). The overhead with T=0 is minimal. Excessively called routines can be eliminated from the timing process by inclusion on IGNORE directives read via the X=file option. The IGNORE directive has the form:

```
IGNORE  (sub1,sub2,...,subk)
```

When a subprogram appears on an IGNORE directive, any call to that routine is not timed. That is, the time spent in an IGNORE'd routine is accumulated as part of the calling routine. No timing code is generated for any calls made by the IGNORE'd routine either. It is as if the IGNORE'd routine were compiled separately without profiling.

IGNORE directives are actually macro calls that are read by the FTN4TRA procedure and inserted into the source file for TRAMAC. TRAMAC is then assembled with these inserted IGNORE statements, which generate conditional assembly code within TRAMAC to check subprogram names. The RJT and program entry/exit macros revert in assembly to their un-profiling state if the subprogram involved appears on the list generated by the IGNORE.

Timing Accuracy:

The subprogram times printed by the FTN4TRA profile (see Figure 1) are gross times, uncorrected for overhead. In the summary at the end of the profile, an attempt is made to estimate the approximate amount of time wasted to overhead, using a very inaccurate fudge factor of .0001 sec/call. This is probably of not very much use, but we found that our users of FTN4TRA were more interested in the relative times (the PERCENT column) than the absolute times. Also we tally "unaccounted time", or the difference between the actual total time and the sum of the times spent in all the routines.

Note the warning INCOMPLETE in the Figure 1 profile. This indicates that the clock for this routine was found running at program termination. This would be expected for the main program and the subroutine EXIT, which never return.

The major difficulty with FTN4TRA has been the increase in memory requirements and execution time inherent in the profiling process. Although every attempt was made to be as optimal as possible, programs that experience a great number of subprogram calls will suffer from the overhead in space and execution time. The IGNORE directive does help some of these situations. It also allows the programmer to profile sections of his code individually when a full profile execution would be too costly, or not fit in core.

4. Possible Extensions:

Timing Statement Blocks:

The obvious extension to the profiling scheme is to provide some way to get down to the statement level. Once a subprogram is identified to be ripe for optimization, how do you find out where within the subprogram it is spending most of its time? And can you find out automatically, without having to pepper the routine with calls to SECOND?

With help from the compiler, it should be possible to provide an option (compiler directive in-line?) that breaks the routine into blocks of statements and times each entry/exit through the block. A report generator could produce a profile on program termination something like:

```
    lines 23-56  17 times  .000231 sec.  1.19 %
```

Obviously, one would have to be very selective about using such a block profiling option. Typically, one would expect to use it on a single routine in execution, once identified by a run with the subprogram call profile.

Cross-Referencing Calls:

The profile could be extended to include a list for each routine of all the routines that call it. This would add to the overhead and the size of the clock table, but could be useful in determining whether a subprogram would be more efficiently coded into the calling routines. A subprogram called from only one routine might be more efficient using COMMON blocks to pass arguments.

Because of the overhead requirements, cross-referencing might be better implemented as part of the no-timing profile (FTN4TRA's T=0 option).

5. A Recommendation:

Using the FTN4TRA procedure at BKY, one user was able to reduce a program's execution time by a significant amount. The profile revealed that the program was spending 40% of its time in a trivial pack/unpack routine. By re-writing this routine in a more intelligent manner, the new version ran 30% faster.

Another user discovered that as a result of a logic error in his program that had been hidden for years, the library routine REWIND. was being called 50 times unnecessarily for every 1 necessary call. Fixing the bug improved execution significantly.

Users of the CRAY-1S computer system, where optimization is critical, have found the automatic "flowtrace" feature of the CFT compiler an invaluable aid.

We recommend to CDC that this feature be considered an important enhancement to existing compilers, especially FTN5. As outlined above, it should be automatic, requiring little or no change to the source program.

The T=0 no-timing option of FTN4TRA, along with the N=n option to set clock size, and the IGNORE directives, have proved sufficiently useful to be considered in any profiling design. A means to direct the profile to a specific file should also be provided. Finally, the profiling option should not be chained to any particular OPT level as DEBUG was to OPT=0.

-*-

```
  --   7600 PROGRAM FLOW PROFILE   --
 ROUTINE      CALLS          TIME(SEC.)              PERCENT    AV.SEC/CALL


 ALIGN            0          .00229306                  .193     0.           INCOMPLETE
 SETFLS          11          .00059730                  .050      .00005
 OUTCI.          51          .01217582                 1.027      .00024
 INPCI.          80          .02186632                 1.844      .00027
 OVERLAY          5          .00082624                  .070      .00017
 R12              1          .00331282                  .279      .00331
 INIT0            1          .00072185                  .061      .00072
 ORBGEN           1          .00053562                  .045      .00054
 INIT1            1          .00576672                  .486      .00577
 READ1            1          .00112365                  .095      .00112
 DATE             1          .00011124                  .009      .00011
 RING1            1          .00628386                  .530      .00628
 CRBOUT           1          .00012152                  .010      .00012
 ERROR1           1          .00085965                  .073      .00086
 SEX1             6          .00239982                  .202      .00040
 RRAN           834          .04828706                 4.073      .00006
 RNORM          828          .13510008                11.395      .00016
 RGEN           828          .04640339                 3.914      .00006
 SEX2             6          .00232196                  .196      .00039
 MAJOR            1          .00132003                  .111      .00132
 SCAT             1          .02501177                 2.110      .02501
 SUBTRA           1          .01145686                  .966      .01146
 LINLSQ           2          .00950560                  .802      .00475
 HECOMP           2          .09350553                 7.887      .04675
 HOLVE            2          .00729889                  .616      .00365
 RESID            2          .00631164                  .532      .00316
 BSET             1          .01234951                 1.042      .01235
 DEORB1           1          .66808940                56.349      .66809
 HARMN            2          .00015309                  .013      .00008
 STAT1          456          .02715644                 2.290      .00006
 STAT2            1          .00009303                  .008      .00009
 FIN1             1          .00014509                  .012      .00015
 CRBMEA           2          .00034958                  .029      .00017
 READ2            2          .00165061                  .139      .00083
 MEA1             2          .00852478                  .719      .00426
 HARM1            1          .00044226                  .037      .00044
 INIT4            1          .00027866                  .024      .00028
 READ41           1          .00123519                  .104      .00124
 SAMPTS           1          .00684404                  .577      .00684
 SAMOUT           1          .00014061                  .012      .00014
 ANAL4            1          .01254864                 1.058      .01255


TOTAL TIME (SEC.)=          1.18562601
TIME UNACCOUNTED (SEC.)=         .00010684  OR     .009 PERCENT OF TOTAL TIME
      3143 SUBPROGRAM CALLS TIMED.  TIMING OVERHEAD (SEC).=          .31430000 APPROX.  (   26.509 PERCENT)

     7600 FTN4 FLOWTRACE COMPLETE
```

Figure 1