MEASURING TRANSACTION RESPONSE TIMES

Chuck Storla
Hewlett Packard
Rolling Meadows, IL

ABSTRACT: One of the major problems confronting system managers and programmers trying to improve the performance of an application is their lack of knowledge of exactly where the application is spending its time. This paper discusses a method of "instrumenting" the user's application so that precise timings are available to the development staff. One of the major benefits of this method is that it may not require any modification or even recompilation of existing code.

I. Introduction.

An on-line application program typically contains subsystem calls and code which performs the following functions: terminal I/O, file handling and logic/computation. All of the specific functions of a user's program such as data entry, inquiry, and error handling, will fall into these three categories. Many current users of the HP3000 have existing applications which perform these respective functions with V/3000, IMAGE/3000 and a high-level language such as COBOL. Although the method discussed in this paper has some general applicability, it will be specifically aimed at the users in this environment.

Should a user experience performance problems with a particular application, the programming staff needs to determine where the bottlenecks are. This can be accomplished through intelligent guessing, through consultation with a Hewlett Packard Performance Specialist or by adding timing code to the program(s) under suspicion. We will add this extra code to the program to determine which phase is taking an unusually large amount of time to execute. If we can narrow the scope of our investigation to that portion of the application which takes the most time to perform or at least takes longer than we feel it should, then we are much closer to knowing how to solve our performance problem. We might find, for example, that for a transaction which takes 20 seconds, the program uses five seconds to retrieve all of the records we need, 2 seconds to display the data, but thirteen seconds to format it. We might then choose to spend our time improving the performance of this

formatting code, rather than on a redesign of the database.

This is not meant to imply that the portion of a program which requires the most time to execute is necessarily the least efficient or even the cause of the problem. However, any area in which a program spends a great deal of time will certainly be of interest to anyone wishing to optimize its execution.

II. DETERMINING WHAT TO TIME.

If we examine a typical on-line application we might find that the program had the following structure:

```
BEGIN
Initialize data, Open files;                    << 1 >>
Display formatted screen;                       << 2 >>
WHILE NOT DONE DO
   BEGIN
   Read screen;  << for input data >>           << 3 >>
   IF end THEN
      done:=TRUE
   ELSE
      BEGIN
      Retrieve data from files;                 << 4 >>
      Format for output to screen;              << 5 >>
      Display data;                             << 6 >>
      END;
   END;
Clean-up, Close files;                          << 7 >>
END.
```

This program is a simplified version of many now in use on HP3000's. This description does not include any error handling and is limited to a single screen, but will serve for the purposes of this discussion. Our interest in the performance of this program would probably center on steps << 3 >> through << 6 >>. The additional steps (1,2 and 7) would only be of interest if this program was run many times and for brief periods. Unless this is true, the start-up and shut-down functions are probably not as crucial to us as the intermediate processing. However, since file opens can be very high overhead operations, we might need to consider these steps in other programs.

To the terminal operator, the function of this program would appear as follows:

```
(a)   :run program
                << wait for 1 & 2 to complete >>
(b)    enter data, hit ENTER
                << wait for 3 through 6 to complete >>
(c)    look at data and either EXIT or return to step (b)
                << if EXIT wait for 3 & 7 to complete >>
```

Obviously, from the terminal user's perspective, the important timing consideration here is the time he or she must sit and wait between hitting ENTER and receiving a full response. Time from the operator's perspective will from now on be referred to as "wall time" and will be distinguished from the amount of time the computer system spends executing instructions on the behalf of this user, which is known as "cpu time". We are interested in cpu time only as it pertains to improving the wall time responses we can provide our terminal user.

It is very important to note that there are many instances in which our program can require significant amounts of wall time for a step that requires little cpu time. This will be especially true when our program must compete for a resource such as a file or data-base lock or must wait for the operating system to grant a buffer, sufficient room in memory or a disc I/O. During the time we are waiting for a particular resource, our process is said to be impeded. While in this state, no cpu time will be used by our process, but many seconds or even minutes of wall time may go by. This being the case, we are interested in timing both wall and cpu times.

Returning to our hypothetical program, we see that we might like to add our timing code just before and just after each of the major steps, especially << 3 >> through << 6 >>. Given the nature of these steps we might assume that step three, reading the screen, will not take much cpu time but due to the operator's typing speed and delays caused by his or her decision making, our measured wall time will be significant. The wall time attributed to thinking about what to enter and actually typing the data is usually referred to as "think time". The wall time from the point at which ENTER is hit until the screen is ready to accept new input will be referred to as "transaction time".

III. HOW TO TIME.

There are several intrinsics available to programmers on the HP3000 to determine both cpu time and wall time. The first of these intrinsics are PROCTIME and TIMER, respectively.* The PROCTIME intrinsic will return the number of milliseconds that have been "charged" to a process for cpu usage. The TIMER intrinsic returns the number of milliseconds from the last time the system was started, or since the last automatic reset to zero. This reset occurs on twenty-four day intervals at midnight.

It should be noted that these intrinsics each have a resolution of one millisecond and therefore should not be used to make single measurements in that range. At the application level in which we are currently interested, most measurements will result in times greater than a tenth of a second and often into seconds. Should an investigation require timings in the millisecond range or below, special methods must be used.

IV. A METHOD OF IMBEDDING TIMING CALLS.

Part of our method then will be to insert into our application program the calls to PROCTIME and TIMER so that we can arrive at elapsed times for cpu and wall time. In many cases however, it may be difficult or undesirable to modify an existing program to add the necessary timing calls. If the calls were imbedded within the source code itself we would as well need a means of disabling the timing during normal operation. In looking at our example program and considering it to be written so that the screen handling is done through calls to V/3000 intrinsics and the file handling is done through calls to IMAGE/3000 intrinsics, we might modify our pseudo-code as follows:

*These intrinsics are documented in the "MPE Intrinsics Reference Manual" (30000-90010).

```
BEGIN
Initialize data, Open files;                    << 1 >>
VSHOWFORM; << Display screen >>                 << 2 >>
WHILE NOT DONE DO
   BEGIN
   VREADFIELDS; << Read screen >>               << 3 >>
   VFIELDEDITS;
   IF end THEN
      done:=TRUE
   ELSE
      BEGIN
      VGETBUFFER;
      << Retrieve data from files >>            << 4 >>
      DBFIND's & DBGET's;
      Format for output to screen;              << 5 >>
      VPUTBUFFER;
      VSHOWFORM; << Display data >>             << 6 >>
      END;
   END;
Clean-up, Close files;                          << 7 >>
END.
```

We can now see that several of the major steps which we wish to time become one or more intrinsic calls. If we could time each of the indivdual intrinsic calls and as well capture the time between several of the calls we could get the bulk of the data we need.

The method used by MPE to link a program to external procedures provides the mechanism we need. At the time a program is executed with the :RUN command** the MPE LOADER will try to resolve any unresolved externals by searching one or more Segmented Libraries. The default is to search only the SL.PUB.SYS file, while at most three SLs will be searched. Assuming a program file resides in a group other than PUB of an account other than SYS, the command ":RUN program;LIB=G" will cause the search to proceed as follows:

**or is allocated with the :ALLOCATE command or has a process created with the CREATE intrinsic.

SEGMENTED LIBRARY SEARCH ORDER

1) SL.group.account is searched

any externals not resolved or any "new" externals will cause a search of

2) SL.PUB.account

any externals not resolved or any "new" externals will cause a search of

3) SL.PUB.SYS

any unresolved externals at this point will cause the load to abort

The reference to any "new" externals after 1) and 2) refers to the following situation: Assume a program which has two unresolved externals, procedures "A" and "B". If we run the program and specify LIB=G, then SL.group.account will be searched. Assuming that this SL contains procedure "A", it will be resolved. When procedure "A", in turn calls "X" which is not contained in SL.group.account, then "X" is a new external and it must be resolved at the account or system SL level.

If the program we wish to time exists in a non-PUB group of a non-SYS account then it might search for the V/3000 and IMAGE/3000 intrinsics in the first two groups and, not finding them, will finally link to those routines in SL.PUB.SYS. If we wish to "intercept" the calls to a particular intrinsic, we can accomplish this at the group SL level. To intercept each call to DBGET, for example, to determine how much time our data-base reads were taking we could write our own version of DBGET and place it in SL.group.account. Now all calls to DBGET would execute our own DBGET routine. This solves the problem of capturing the calls to DBGET, but to allow the program to operate correctly we need to somehow get from our routine to the "real" DBGET in SL.PUB.SYS. If our local DBGET makes a call to DBGET it will simply be recursive on itself.

The solution involves calling another user written routine which we will call DBGET'. This routine will simply accept

all of the standard DBGET parameters and use them to call DBGET. It will be placed into the account SL so that its call to DBGET will be resolved to the "real" DBGET in the system SL. Thus, when our program calls DBGET, it executes our group SL version of DBGET which logs some timing information and then calls DBGET'. This routine in SL.PUB.account in turn, calls DBGET in SL.PUB.SYS which executes the actual function.

This solution solves several of the problems we discussed previously. SInce it requires no modification of user or system code, it is easy to implement and can be used in some situations where original source code is not available for modification and recompilation. It also allows us to enable and disable the timing code easily. To enable the timing simply run the program(s) with LIB=G. The default case is disabled (LIB=S).

V. CONSIDERATIONS.

The following must be considered when evaluating this technique:

- Programs must reside in non-PUB group of non-SYS account.
- If the programs currently use the group SL, then two versions must be created. One including timing calls, one without.
- Performance may be minimally affected by the additional procedure calls (two per call to a timed intrinsic) and the additional timing and logging code.
- If timing information is being written to a file, provisions must be made to allow it to be shared between multiple users.
- Timing of some portions of the program must be inferred, since only certain procedure calls are captured.
- No indication is given as to the cause of poor performance in the case of impedes.

The method discussed here provides a tool for application programmers to use in learning more about the behavior of their systems. The use of dummy intrinsics in group and account SLs has been used successfully by various Hewlett Packard personnel as a method of timing the execution of some routines and as a method of logging all terminal interaction to a disc file for later manipulation. Within the limitations of the data gathered and depending upon the analysis of that data it provides a mechanism that is both easy to implement and is specific to the area in which data is desired.