How can We Use a "slow" Computer Comfortably?

Kozo Itano*

In order to establish a more comfortable user interface, a system should inform the user the behavior of his program during execution. For this purpose, a mechanism for monitoring a program during execution is implemented, and examples of the monitoring are given for several practical cases in an experimental system.

1. Introduction

Interactive use of time sharing systems has been widespread among contemporary users, and a rapid response of the system is indispensble for establishing a good user interface. For example, if the system responses within 0.1 second, a user would not be concern about the response time. However, when the response time exceeds several seconds, the user would be irritated against the "slow" computer, even if he knows that the excution of the program needs long computation time. The essential reason why he is irritated is that he is forced to be kept waiting for an indefinite time and he is never given any information about when the execution of the program is finished. When the turn around time exceeds several seconds, the user has to keep his nerve at some tension until the system responses to him. This kind of situation may disturbe his creative thinking especially in case he engages in intelligent work. In such an enviornment, if the system could inform the user of the current execution state of his programs, a more comfortable user interface would be established.

There are two kind of turn around times: waiting time and execution time. A successful example of reporting system of the waiting time is the Cambridge 370 system [1] which reports to the user when his job will be executed. In case of execution time, however, there is no attempt to predict the whole execution time in advance, except that only the used cpu time is given when requested by the user [2].

The new idea devised can monitor the behavior of the program during execution and can predict when it is finished. In this paper the author gives the detailed mechanism, implementation of the system, and examples of observation of the actual execution of the programs.

2. How an Interactive System Should be

In this section, we would analyze a mechanism of interaction between a user and a computer. A typical example is a time sharing operating system, where a user usually types commands through a terminal and a response is printed or displayed at his terminal. In the case of a program which needs strong interaction such as a text editor, a rather rapid response is indispensable to make a good user interface. However, we cannot expect such a short response time in all the cases; some may need long computation time. In most cases, the computation power of the machine is limited and the system cannot execute all jobs in an instant. In this situation, a user as a human being is forced to wait for some time, and he usually cannot accept it; hence his creative thinking is disturbed.

One of the successful solutions to these situations is given in the UNIX operating system [14], which supports the two major capabilities: full duplex terminal input/output and multi-process. In this system, a user can input his commands independently from the meassage outputs or responses to them. He need not wait for the response of the system unless necessary. Further, the multi-process capability permits a single user to execute several programs in parallel. For example, he can edit a text during compilation of other programs. This means that the user can get to the next work and need not wait for the completion of the programs which has a long computation time.

In many cases, however, we must wait for the completion of the current program. For example, we cannot execute or test a program until the compilation and likage are finished. Unfortunately he might not have a text to edit while waiting for the completion of the computation. Hence, "how to know the current state of the programs" remains as an important problem in order to use a "slow" computer comfortably, even if the system supports full duplex input/output and multi-process capabilities.

3. Basic Algorithm

A basic strategy to understand the behavior of programs during execution is to monitor periodically an execution probe (E-probe is used) which indicates how

*Institute of Information Sciences and Electronics, University of Tsukuba, Sakura-mura, Niihari-gun, Ibaraki 305, Japan.

the execution is proceeding [3]. Usually, an E-probe is a function of several variables in the program or data to be monitored. In order to eliminate the overhead due to the observation of programs, we evaluate the E-probe only when it is observed, for example, each one second. The E-probe is 0 when the execution begins, and 1 when the execution is finished. If this E-probe is completely proportional to the time from the beginning of the execution, it would be known precisely how the execution of the program is proceeding by monitoring the E-probe. Further, we can know how much time is necessary to finish the execution of the program. For example, when the E-probe is observed as 0.3, it indicates that 30 percents of the total execution has been finished.

However, an ideal E-probe which is completely proportional to the time cannot be implemented in practical programs, therefore we have to use some approximation. As an approximation of the E-probes, we present two mechanisms below.

(1) Mechanism 1

In this mechanism, as an E-probe we use the size of data which are used during input or output. For example, let Q be the size of total input data to be processes, and q be the size of input data which was been processed already. Then, the E-probe E is defined as:

$$E=q/Q.$$

Q and q are commonly used in most read routines, and the use of these variables would not produce any overhead during execution.

This mechanism can be used in the case of translators such as compilers and assemblers. In the case of a multi-pass compiler, the E-probe becomes somewhat complex, because the relation between passes should be considered.

(2) Mechanism 2

In this mechanism, we should analyze the behavior of the program during execution and define the E-probe. As a simple example of this case, a matrix multiplication program P1 is shown below.

```
DO 10 J=1, N
DO 10 I=1, N
C(I, J)=0
DO 10 K=1, N
10 C(I, J)=C(I, J)+A(I, K)*B(K, J)
```

P1. Matrix multiplication

In this program P1, the statement 10 is executed N^3 times, then the E-probe E is defined as shown below.

$$E=((J-1)N^2+(I-1)N+K-1)/N^3.$$

As an E-probe, also we can use E0 and E1 as an approximation of this E.

$$E0=(J-1)/N$$
$$E1=((J-1)N+I-1)/N^2.$$

By the use of this E-probe, we can predict the total execution time. Let t be the time from the beginning of the execution, E(t) be the E-probe at time t, and T be the total execution time to be predicted, then T is

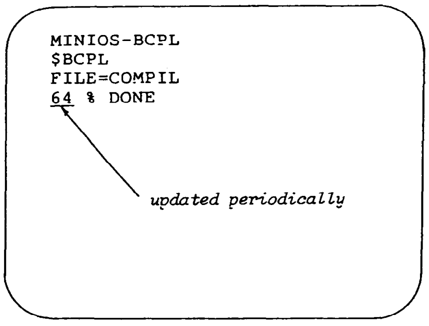

Fig. 1 Display Format.

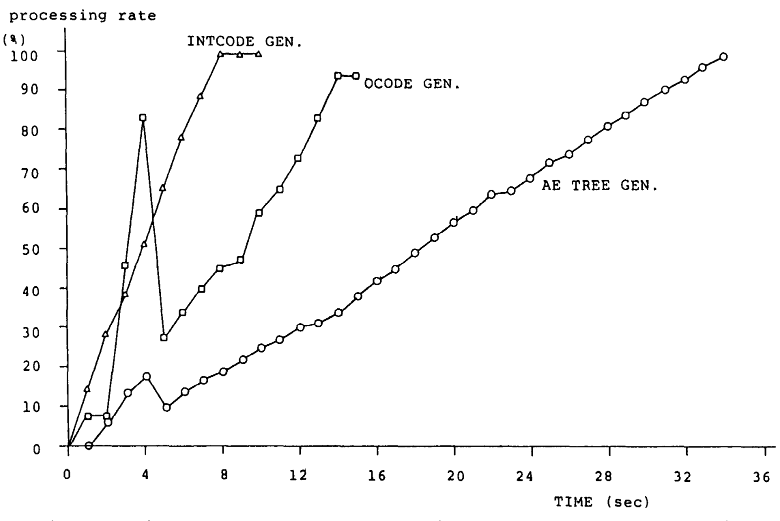

Fig. 2 Relation between the E-probe and execution time in case of the BCPL compiler.

estimated as follows:

$$T = t/E(t).$$

And how much the execution has been proceeding is indicated by the E-probe E(t) itself.

4. Implementation

An experimental system to monitor the execution of programs is implemented on a small computer system the TOSBAC-40C which is equipped with 64K bytes of main memory, five mega bytes of magnetic disks, two magnetic tape drives, a character display console, a real time clock, and a printer. For easiness of the modification of the operating system, MINIOS [4] was used.

For the implementation of the mechanism 1, we used a BCPL compiler [5] and runoff program [6] written in BCPL. Since the data size of the file should be definite before the execution, the byte size of files was installed in the file system of MINIOS. For the implementation of the mechanism 2, a Gaussian elimination program [7] was chosen. The results of the monitoring of the execution is displayed on the character display console in real time. This console is connected to the computer through a high speed communication line interface which allows quick updating of the screen. An example of display format is shown in Fig. 1. In order to avoid a noisy message, the message is updated in the same position of the screen each one second.

5. Precision of the Monitoring of Execution

Measurements of program behavior are made for the

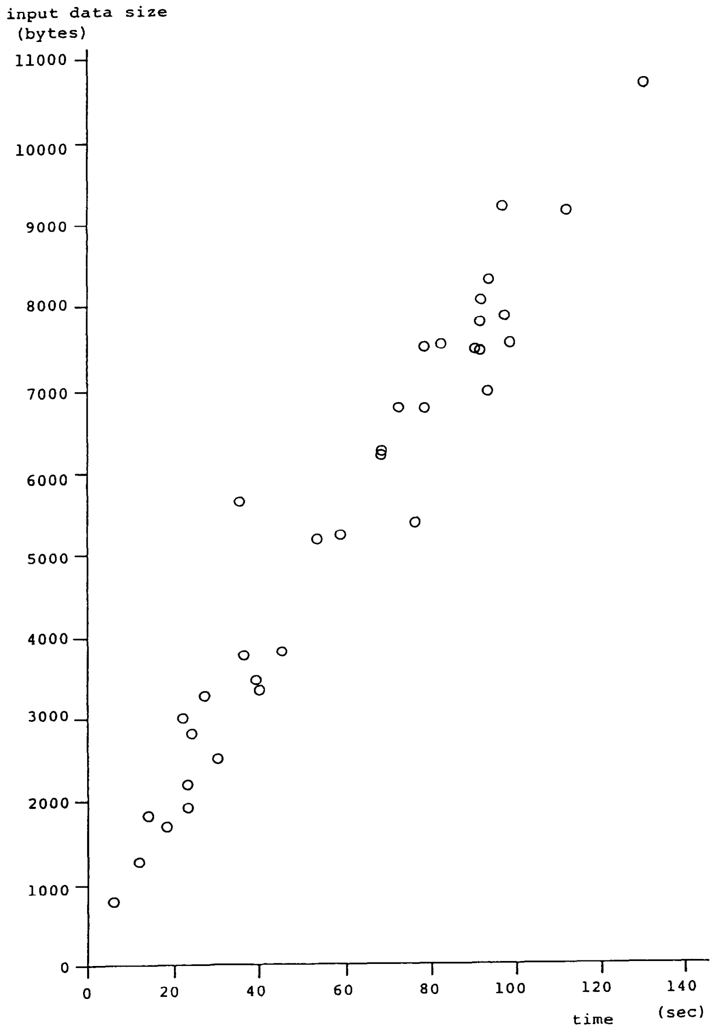

Fig. 3 Relation between execution time and input data size in case of BCPL compiler.

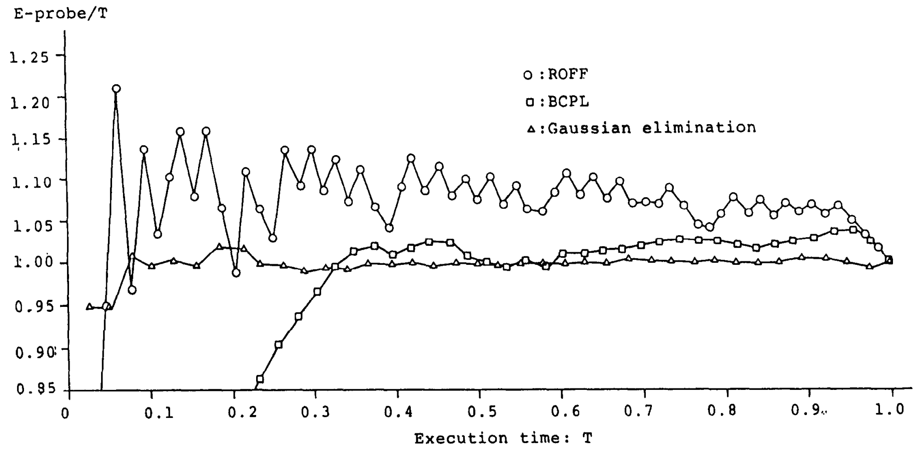

Fig. 4 Relation between the E-probe and execution time in case of the BCPL compiler, the runoff, and the Gaussian elimination.

BCPL compiler, the runoff program, and the Gaussian elimination program. The BCPL compiler has three phases: (1) AE tree generation, (2) OCODE generation, and (3) INTCLDE generation. Although the first and third phases of the compiler gave good results, the second phase gave rather poor results. This is because the input of the second phase is an AE tree whose data structure is not a linear one.

In the actual estimation of the total computation time, the E-probe is not so precise in the beginning of the program execution. However, it becomes more precise as the execution proceeds. We show the measured relation of the E-probe and actual execution time as in Fig. 2, and the relation between execution time and size of input data in Fig. 3. The results measured for the Gaussian elimination are also shown in Fig. 4.

6. Concluding Remarks

Quality and adaptability of the monitoring is mostly dependent upon the E-probe. Therefore, the most important problem is to make a good E-probe systematically in the programs to be observed. Many analyses have been done concerning the behaviors of the programs during execution [8-13] and there is a good possibility to make up a good E-probe by the use of this kind of analysis.

The applications discussed above are restricted in the limited class of programs related to the man-machine interface. However, the E-probe could be applied to many other situations of computer system. For example, a more optimum job scheduling algorithm based on the prediction using the E-probe would enable rearranging the execution of jobs. Also, the E-probe might be useful in debugging facilities, because the users can monitor their programs under test implicitly or explicitly.

Acknowledgement

The author would like to express his thanks to Dr. Tetsuo Ida at Institute of Physical and Chemical Research and Mr. Kiyoshi Ishihata at University of Tokyo for their helpful discussions.

References

1. Steward, P. and Stibbs, R. J. Cambridge 370/165 user's reference manual, University of Cambridge Computing Service (1976).
2. DEC SYSTEM-20 User's guide, DEC (1976).
3. Itano, K. Prediction of the actual execution time of programs and its application (in Japanese), Programming Symposium, Hakone Japan, 21 (January 1980), 185-194.
4. MINIOS Reference Manual (in Japanese), University of Tokyo (1974).
5. Richard, M. BCPL: A tool for compiler writing and system programming, SJCC (1976), 557-566.
6. Ida, T., Itano, K. and Ishihata, K. Implementation of the alphanumeric text formatter: ROFF (in Japanese), Annual Report of Computer Centre, University of Tokyo, 7 (1977).
7. Forsythe, G. and Moler, C. B. Computer solution of linear algebraic systems, Prentice-Hall (1976).
8. Usijima, K. and Harada, K. Tools for analysis and evaluation of software (in Japanese), Johoshori, 20, 8 (1979), 703-711.
9. Ingalls, D. The execution time profile as a programming tool, In design and optimization of compilers edited by Rustin, R., Prentice-Hall (1972), 107-128.
10. Knuth, D. E. An empirical study of FORTRAN programs, Software Practice and Experience, 1 (1971), 105-1433.
11. Knuth, D. E. and Stevenson, F. R. Optimal measurement points for program frequency counts, BIT, 13 (1973), 313-322.
12. Ramamoorsey, C. V., Kim, K. H. and Chen, W. T. Optimal placement of software monitoring aiding systematic testing, IEEE Trans. SE., SE-1, 4 (1975), 403-411.
13. Fosdick, L. D. and Osterweil, L. J. Data flow analysis in software reliability, Computing Surveys, ACM, 8, 3 (1976), 305-330.
14. Ritchie, D. M. and Thompson, K. The UNIX time sharing system, CACM, 17, 7 (1974), 365-375.

(Received April 7, 1980: revised July 25, 1980)