U.S. DEPARTMENT OF COMMERCE • National Bureau of Standards

NBSIR 86-3416

Simple Multiprocessor Performance Measurement Techniques and Examples of Their Use

Alan Mink, John W. Roberts,
Jesse M. Draper, Robert J. Carpenter

U. S. DEPARTMENT OF COMMERCE
National Bureau of Standards
Center for Computer Systems Engineering
Institute for Computer Sciences and Technology
Gaithersburg, MD 20899

Institute for Computer Sciences and Technology

CMRF

COMPUTER MEASUREMENT
RESEARCH FACILITY
FOR HIGH PERFORMANCE
PARALLEL COMPUTATION

July 1986

Partially sponsored by the
Defense Advanced Research Projects Agency
1400 Wilson Boulevard
Arlington, VA 22209
ARPA Order No. 5520, July 23, 1985

Simple Multiprocessor Performance Measurement Techniques and Examples of Their Use

---

Alan Mink
John W. Roberts
Jesse M. Draper
Robert J. Carpenter

Center for Computer Systems Engineering
Institute for Computer Sciences and Technology
National Bureau of Standards
Gaithersburg, MD 20899

Partially sponsored by the
Defense Advanced Research Projects Agency
1400 Wilson Boulevard
Arlington, Virginia 22209
ARPA Order No. 5520, July 23, 1985

U.S. Department of Commerce, Malcolm Baldrige, Secretary

National Bureau of Standards, Ernest Ambler, Director

July 1986

TABLE OF CONTENTS

Simple Multiprocessor Performance Measurement Techniques and Examples of Their Use

Alan Mink
John W. Roberts
Jesse M. Draper
Robert J. Carpenter

This report describes simple hardware techniques for the measurement of the performance of multiprocessor computers. A number of examples of data obtained using these techniques are reported, as well as a discussion of the timing accuracy obtainable with this approach.

Key words: Multiprocessor computers; parallel computers; performance measurement; hardware.

INTRODUCTION

A full expression of the performance of multiprocessor parallel computers will require the measurement of many parameters. These include the utilization of such resources as the individual processors, interconnection network, cache memory, etc. Special hardware to gather this information without perturbation of the computer under test is under development at the National Bureau of Standards. Our long-range goal is to develop a measurement device which will be connected at many (100-200) points to the system under test. The signal lines will be divided into groups, and each group fed into a "preprocessor", which performs extensive reduction and interpretation of the incoming data. The preprocessors generally run independently from one another, but when any one activates a trigger mechanism, they all cooperate to place a record of the system state into a large memory from which further analysis can be performed. Such a measurement device can be made sufficiently general that it may be used with a substantial range of multiprocessor architectures.

Since it will take many months to design and build the long-term measurement device, we decided to begin by setting up the Interim Measurement System, in order to begin taking measurements earlier. We will be able to gain first-hand experience in the problems of measuring multiprocessors, and can use the knowledge gained to guide the development of the long-term system. Furthermore, there is a large amount of needed design information common to the two measurement systems, and we will be able to establish the suitable solutions on the interim system before it will be possible to do so on the long-term system. This will help to catch major design weaknesses early, so they can be avoided in the later system.

LIMITATIONS OF TRADITIONAL APPROACHES

The traditional software approach to instrument a program for measuring performance is to insert supervisory (system interrupt) calls (to the operating system) to obtain a timestamp at the beginning and end of an event, to store this information in a buffer area in memory, and later write it to a file at some convenient time. A major problem with this approach is that it perturbs the original program with extra overhead, particularly additional execution time. This additional execution time is required for the operating system to provide the time and for the code added to the program being tested to store the time with the data describing the event. This time overhead is on the order of tens of milliseconds, or worse. In a multiprocessor system, this overhead is only suffered by the processor asking for the timing; all others proceed without delay. This can greatly distort execution in interprocessor synchronization situations. Of course this distortion would not exist in a uniprocessor situation.

INTERIM MEASUREMENT SYSTEM

The Interim Measurement System, unlike the traditional software approach, minimally perturbs the program by requiring very little overhead to capture a timestamp. The Interim Measurement System code imbedded in the (users) software of the system under test is very simple; as little as a single assignment statement per timestamp. The overall configuration of the measurement system is illustrated in Figure 1. At selected test points in the programs, the processors write *events* to a board called the Event Data Card, which has been installed in the normal I/O space of the system. These are *edc* assignment statements. The testbed computer being measured has a Multibus card cage for peripheral devices, so the Event Data Card was implemented on a Multibus board and designed to respond as a Multibus slave device. Only a few microseconds are required for the computer being measured to execute each *edc* assignment statement, since no supervisory calls are made to the operating system. As reported below, the time overhead for a measurement is two to fifteen microseconds (NOT milliseconds), so there is much less effect on parallel programming operation.

Overall View

As each *edc* statement is encountered in the execution of the test program on the system being measured, an *event* is said to occur. The *events* contain data specified by the user at the time they are inserted in programs used to test the system, and may include process ID, values of variables, or other state information. The Event Data Card then adds hardware signals including a microsecond timestamp, processor identification, and user/supervisor mode status, and sends a trigger signal to a logic analyzer to enter the data into its 48-bit-wide buffer. The use of hardware probes to determine the identity of the processor initiating the measurement *event* is complicated by extensive use of pipelining in the hardware of the system under test. By the time the Event Data Card becomes aware of the need to capture data, many of the signals of interest have vanished. The solution is to use flip-flops on the Event Data Card to always capture these signals while they are valid, then hold them until they need to be entered into the data latch. A logical combination of signals provides the needed processor identification. Many other useful signals on this and other systems could be derived using this technique, which is a simple form of the preprocessing to be used in the long-term measurement system. Note that in shared-memory multiprocessors, all processors can write to at least some common range of addresses, one of which can be used as the address for an Event Data Card. Thus only a single Event

Data Card is needed in most shared memory systems.

Data Buffer (Logic Analyzer)

As mentioned above, a logic analyzer is used as a data buffer to store the event data. One complete item of data, including the timestamp, is captured every time it is triggered. The logic analyzer is controlled through an IEEE-488 interface by a Sun computer, which performs statistical analysis and display of the data, as described below. The logic analyzer we are using takes 48-bit samples. The 48 input lines are allocated among the signals to be captured as follows: 32 bits for timestamp, 11 bits for event data written by the user program, four bits for processor ID, and one bit to indicate user/supervisor mode.

Limitations caused by the buffer. The logic analyzer link in the measurement chain suffers from significant limitations in capacity and speed. The 512 word buffer size of the logic analyzer limits the number of continuous samples that can be captured. The effective transmission speed of the event data from the logic analyzer to the analysis computer via the IEEE 448 interface is on the order of seconds for the entire buffer contents. The logic analyzer cannot capture samples while it is transmitting its buffer contents via the IEEE 488 interface. The logic analyzer cannot report the number of samples it has captured or lost; therefore, it is important to plan each experiment carefully around this limitation. Finally, the number of "independent" devices in the data chain creates at best annoyance, and at times problems, in the access and transmission of data.

Other buffering techniques. Had this not been an *interim* system, we would have gone to the trouble of providing a substantially larger buffer for the event data on the Event Data Card itself. In this case the data could either be analyzed later on the system being tested, or on an external machine as in the present case.

Program Perturbation

The Interim Measurement System has some limitations. The Interim Measurement System still perturbs the the execution of the test program, though a great deal less than the traditional approach. As in the traditional approach, a programmer must add code to the program to delineate the events to be measured. With the Interim Measurement System this is just a simple assignment statement for each timing event. Each time a new set of events is to be measured, a programmer must again change the locations of the added commands. Since activation of the Event Data Card requires execution of code placed in the test programs, it is inevitable that there will be some perturbation of the results caused by the measurement process itself. The long-term measurement system will avoid this problem; in the meantime, we can produce useful results by taking the perturbations into account, and keeping them as small as possible. Experiments to measure this perturbation are reported below, and show a cost of from two to ten microseconds per write to the Event Data Card with the "guinea pig" computer we are testing.

Summary of System Issues

Although the interim measurement system greatly improves on the traditional measurement approach, it still perturbs the system to some extent and has some other limitations. The full-fledged measurement system, currently under construction, does not suffer from these disadvantages and will not perturb the system being measured.

MEASUREMENT METHODOLOGY

The coordination of a programmer familiar with the application program being measured and an analyst is required to obtain measurements using the interim measurement system.

Measurement Events

The programmer and the analyst determine the events to be measured and the data that the programmer will output to the Event Data Card to identify each event. There are two major categories of metrics that are usually measured, intervals and frequencies. Intervals require paired events, the start of the interval and the end. Frequencies may require one or two events, depending on the variation of the metric. In the simplest form, only one event is necessary to keep a simple count. In more complex variants, two events are necessary to determine ratios. The programmer must insert special *edc* measurement statements in the source code of the application program. These write the event data to the Event Data Card identifying the measurement events. The analyst must write (or modify) an analysis program that will take the timestamped application-specific data, match up interval boundaries of each event, and output a statistical summary for each event.

Running the Experiment

The programmer and the analyst then begin execution of the application program on the multiprocessor computer under test. At times previously coordinated with the programmer, the analyst initiates a capture program on the analysis computer which acquires a copy of the logic analyzer's buffer containing the timestamped event data. This data is placed in a file to be analyzed by a program written by the analyst. If the test program has been so designed, this process may be repeated a few times for each experiment, to determine the stability of the statistical results. The logic analyzer cannot sample the Event Data Card information while the analysis computer is reading out the data buffer. Therefore, either the experiment must be designed to limit the amount of event data written to the Event Data Card or the data captured in any arbitrary window must be sufficient for analysis.

INITIAL MEASUREMENTS

A series of experiments were planned to test the Interim Measurement System and to determine the program and execution overhead involved in its use. In the following discussions, the macro

"edc(<data>)" represents the *edc* measurement statement, which actually is coded (in C) as

*<address of Event Data Card> = <event data to write>.

Determination of Measurement Overhead

The frist program did nothing but write to the Event Data Card to determine the overhead of the *edc()* measurement statement. In this program there is only one kind of event, the *edc* statement. Therefore no encoding of the data is necessary and, in fact, the data written is irrelevant. Only the timestamp is of interest to determine the length of each successive interval. Also the amount of data is not significant since whatever window of data was captured is as valid as those lost. Each experiment consisted of three variations to the program structure. The first variation used a constant in the *edc* statement.

```
edc(0xFA0);
edc(0xFA0);
edc(0xFA0);
 etc.
```

The second variation used a variable in the *edc* statement.

```
edc(q);
edc(q);
edc(q);
 etc.
```

The third variation used a constant ORed with a variable in the *edc* statement.

```
edc(proc|0xFA);
edc(proc|0xFA);
edc(proc|0xFA);
 etc.
```

This last variation was indicative of the type of data expected in other experiments, where the variable may represent the process number and the constant may represent the specific event.

Measurement of Loop Structures

Data were taken of the intervals between successive *edc* assignment statements imbedded in typical programming looping structures. The second experiment consisted of a single *edc* assignment statement imbedded in a DO-WHILE loop construct.

```
do
{   edc(<one of the above three variations here>);
}while(--q);
```

The interval between successive executions of the *edc* instruction measured the speed of the DO-WHILE construct plus one *edc* execution. The third experiment consisted of a single *edc* assignment statement imbedded in a WHILE loop construct.

```
while(q--)
{   edc(<one of the above three variations here>);
};
```

The fourth experiment consisted of a single *edc* assignment statement imbedded in a FOR loop construct.

```
for(i=1; i<=q; i++)
{   edc(<one of the above three variations here>);
}
```

This resulted in a measure of the speed of the FOR construct.

Data Analysis

The analysis program took the raw timestamp data and converted it to intervals. The first pass of the analysis program computed statistics for all interval data. The second pass of the analysis program used the smallest interval as a baseline and discarded all intervals greater than four times as large. Since the intervals between events were small and regular, this culling operation filtered out time intervals not attributable to the application test code. These longer intervals are from sources such as interrupts,wherein the processor serviced some other task in the middle of the execution of the test loop structure. The statistics for this reduced set of intervals was computed along with its distribution. The statistics for each experiment include the number of sample points (events), their range and mean. If any values are more than four times as large as the minimum value, they are excluded. A new range and mean are computed for the reduced data set. The distribution of the data values for the culled data are provided for later plotting (Figures 2 - 13). This includes the percentage of data in each one-microsecond interval.

Results

Each experiment shows a narrow range of time for the event being measured, which is consistent with the expected results. One may initially expect a single time value for the event interval for such a repetitious series, but after some thought a narrow range seems more reasonable. First, the resolution of the Event Data Card timestamp is one microsecond. Since the events are asynchronous to the Event Data Card clock, a plus-or-minus one microsecond variation is possible due to the phase difference of the two clocks. Second, a few microsecond variation may be expected due to the instruction prefetch of the multiprocessor computer under test. Due to the structure of the code, no time variation is expected based on the operation of the cache, and translation look-aside buffer misses in the memory management unit are not expected to be frequent in this test. The environment in which these initial measurements were made was that of an unloaded system -- no other active users and only a single (non-parallel) active process. The measurement results are summarized in Table 1, but are more graphically presented in the figures mentioned below.

Simple measurement statements. The first experiment consisted of nothing but in-line sequential *edc* statements, with no intervening code between each statement. This resulted in a measure of the speed of the *edc* statement. This experiment was conducted with three variations in the data written at each event. Two runs are plotted from each. In the first case, a simple constant is written (See Figures 2a and 2b). In this, as in all other histograms in this paper, only the **culled** data is plotted. Each plot shows the percentage of the unculled data points which were discarded. In the second case a simple variable is written (See Figures 3a and 3b). In the third case, a variable ORed with a constant is written (See Figures 4a and 4b).

Loop structures. Data were taken of the intervals between successive *edc* statements imbedded in typical programming looping structures. The second experiment consisted of a single *edc* statement imbedded in a DO-WHILE loop construct. This resulted in a measure of the speed of the DO-WHILE construct. Two such runs are plotted for all three types of data written by the *edc* statement (See Figures 5a and 5b, 6a and 6b, and 7a and 7b). The third experiment consisted of a single *edc* statement imbedded in a WHILE loop construct. This resulted in a measure of the speed of the WHILE construct. Two such runs are plotted for all three types of data written by the *edc* statement (See Figures 11a and 11b, 12a and 12b, and 13a and 13b). The fourth experiment consisted of a single *edc*

statement imbedded in a FOR loop construct. This resulted in a measure of the speed of the FOR construct. Two such runs are plotted for all three types of data written by the *edc* statement (See Figures 8a and 8b, 9a and 9b, and 10a and 10b). Subtracting the basic measurement overhead (Figures 2 through 4) from the corresponding times in Figures 5 through 13 allows evaluation of the execution time of the loop mechanisms in the presence of different types of imbedded instructions. The appropriate columns in Table 1 show the substantial consistency.

CONCLUSIONS

The results shown here demonstrate that a relatively simple hardware attachment makes it possible to measure execution times of programs, *and small segments of programs,* with few-microsecond accuracy and without substantially perturbing the execution of the program.

LOOP TYPE	DATA TYPE	RAW DATA				NUMBER OF OUTLYING SAMPLES	CULLED DATA			
		NO. OF SAMPLES	MICROSECONDS				MICROSECONDS			
			MIN	MAX	MEAN		MIN	MAX	MEAN	LOOP - DATA (MEANS)
None	Constant	513	2	1121	8.52	3	2	5	3.28	
None	Variable	513	2	789	8.80	4	2	6	3.45	
None	Constant | Variable	513	5	1104	17.6	8	5	10	6.28	
Do While	Constant	513	7	393	8.45	1	7	8	7.70	4.42
Do While	Variable	513	7	8	7.50	0	7	8	7.50	4.05
Do While	Constant | Variable	513	9	365	10.2	1	9	11	9.50	3.22
While	Constant	513	9	10	9.30	0	9	10	9.30	6.02
While	Variable	513	8	9	8.80	0	8	9	8.80	5.35
While	Constant | Variable	513	11	335	11.7	1	11	12	11.1	4.82
For	Constant	513	8	354	8.97	1	8	9	8.30	5.02
For	Variable	513	8	9	8.30	0	8	9	8.30	4.85
For	Constant | Variable	513	11	379	11.8	1	11	12	11.1	4.82

TABLE 1 - Execution time measurements

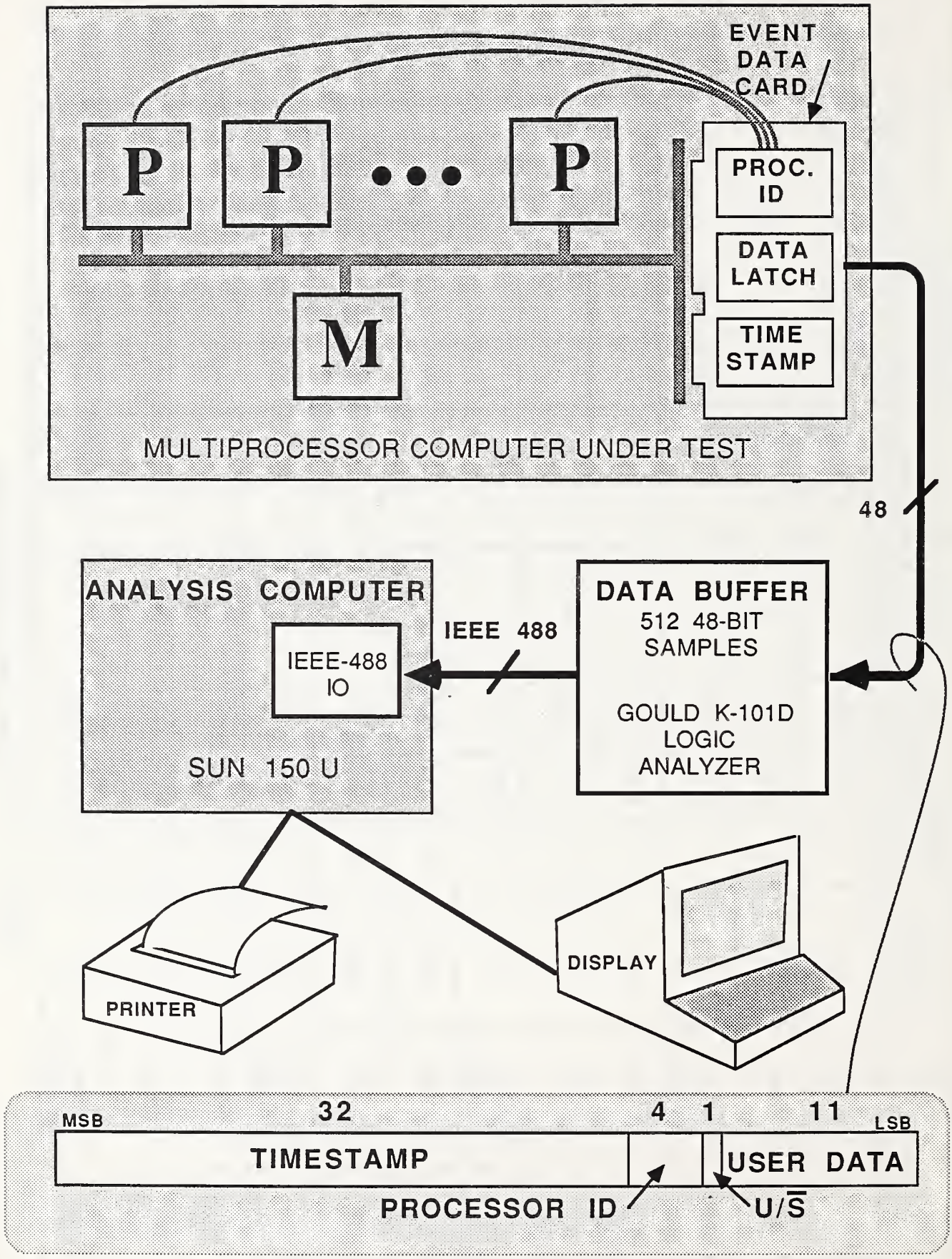

FIG 1 - INTERIM MEASUREMENT SYSTEM

NBS652
860505
860715

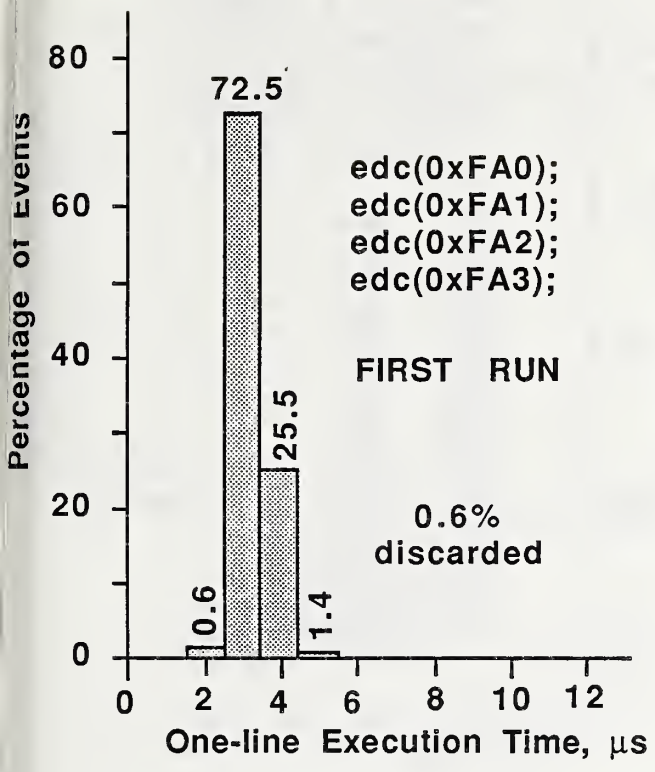

Fig. 2a

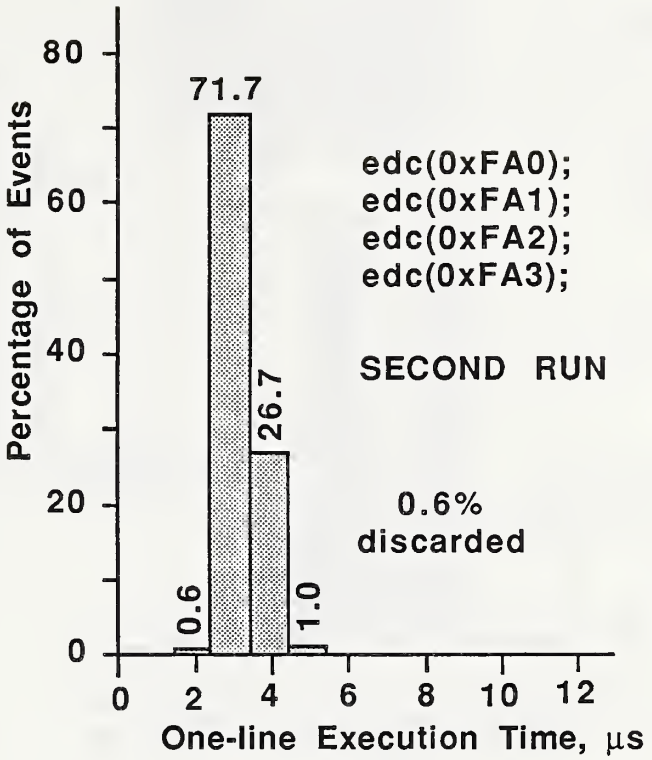

Fig. 2b

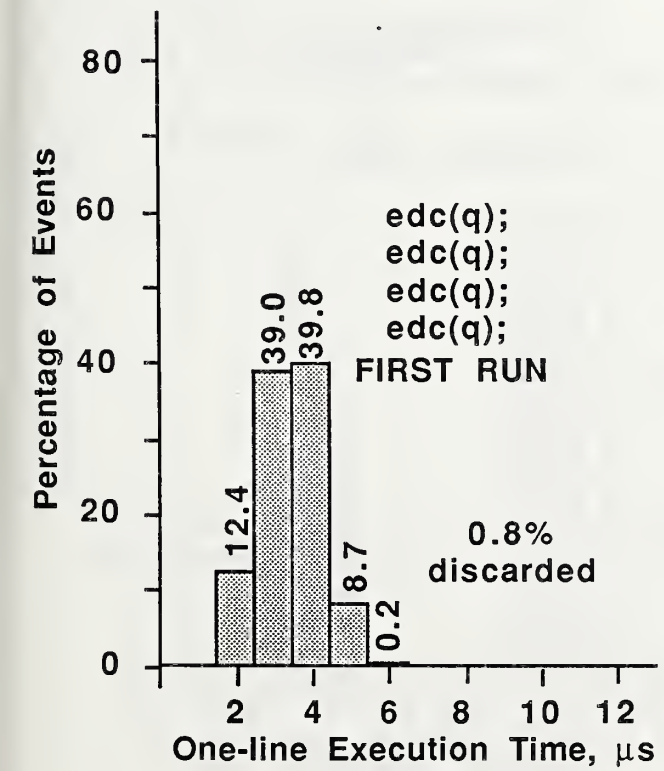

Fig. 3a

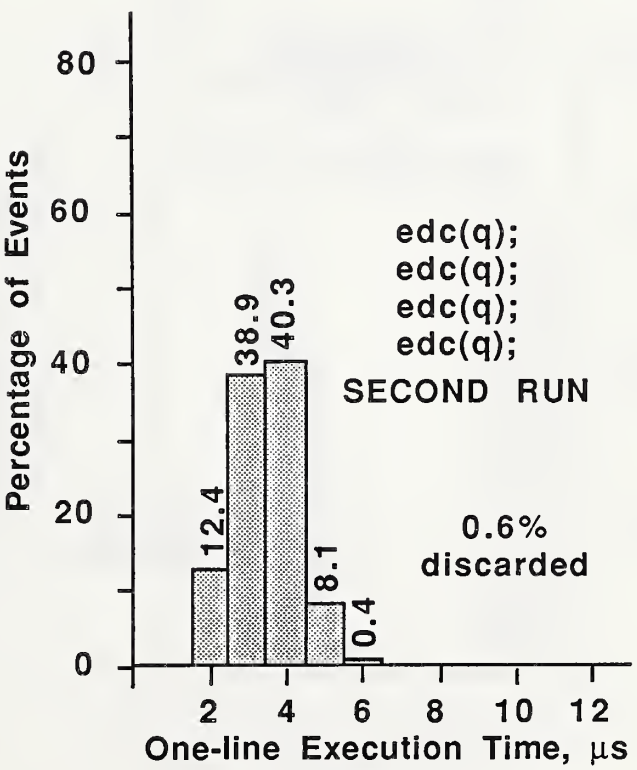

Fig. 3b

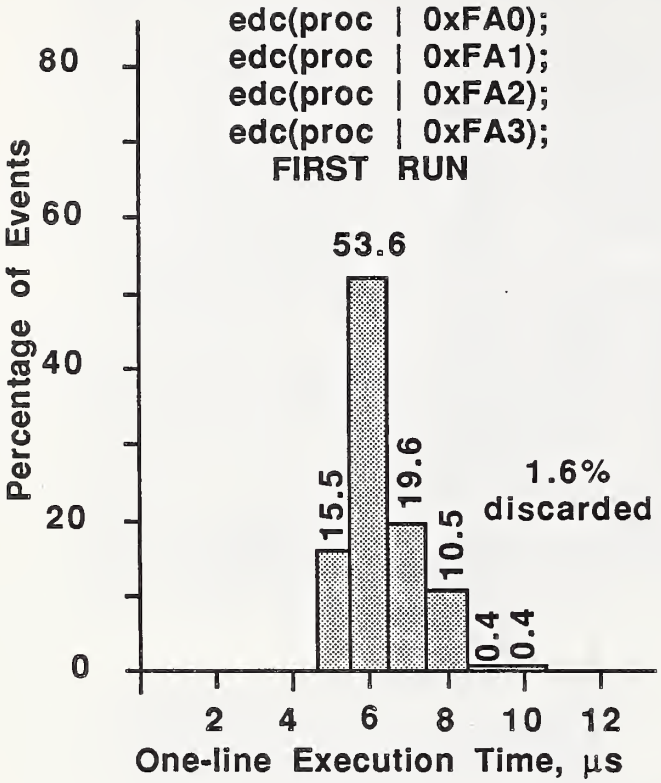

Fig. 4a

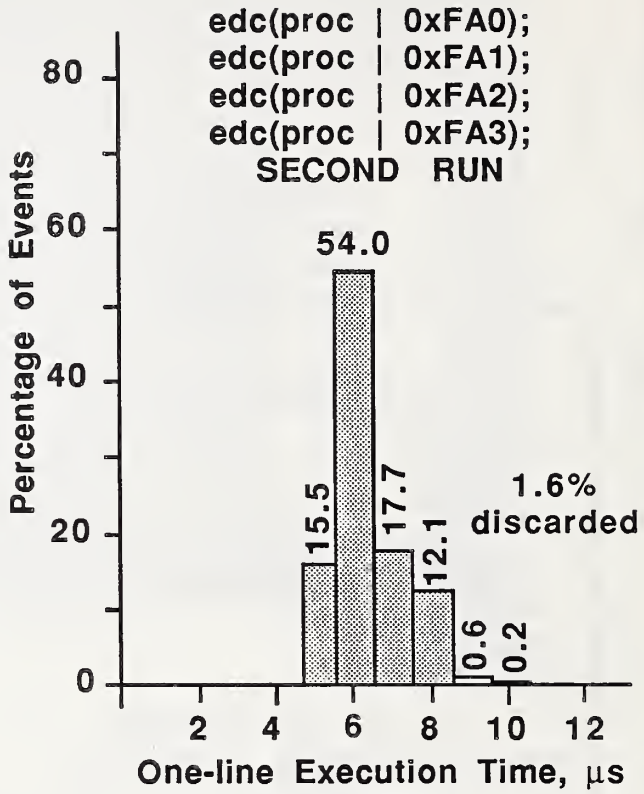

Fig. 4b

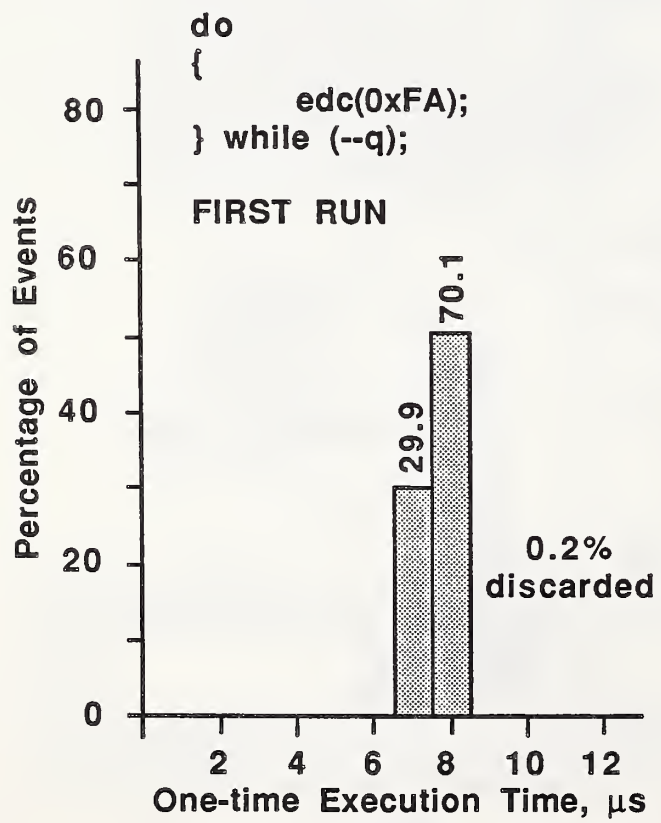

Fig. 5a

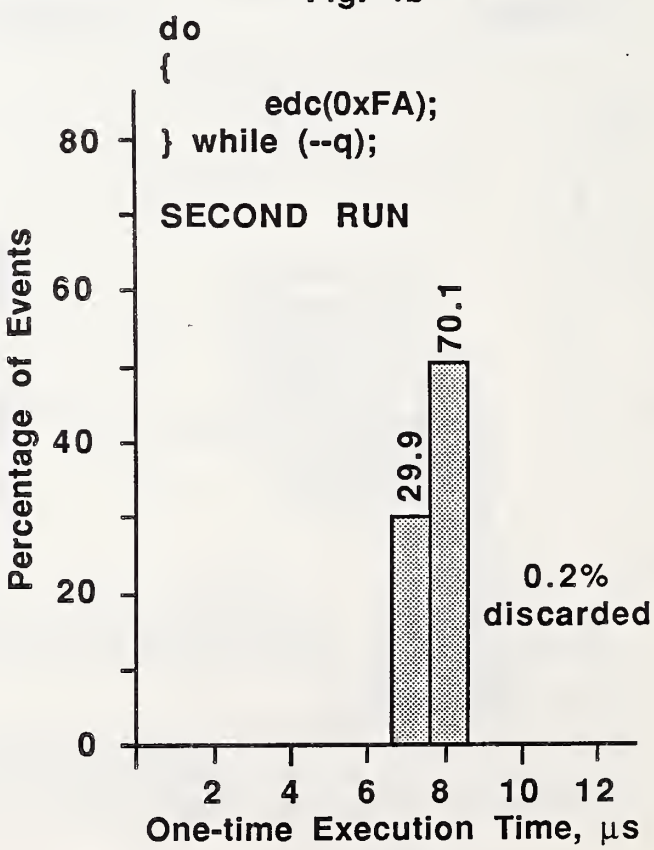

Fig. 5b

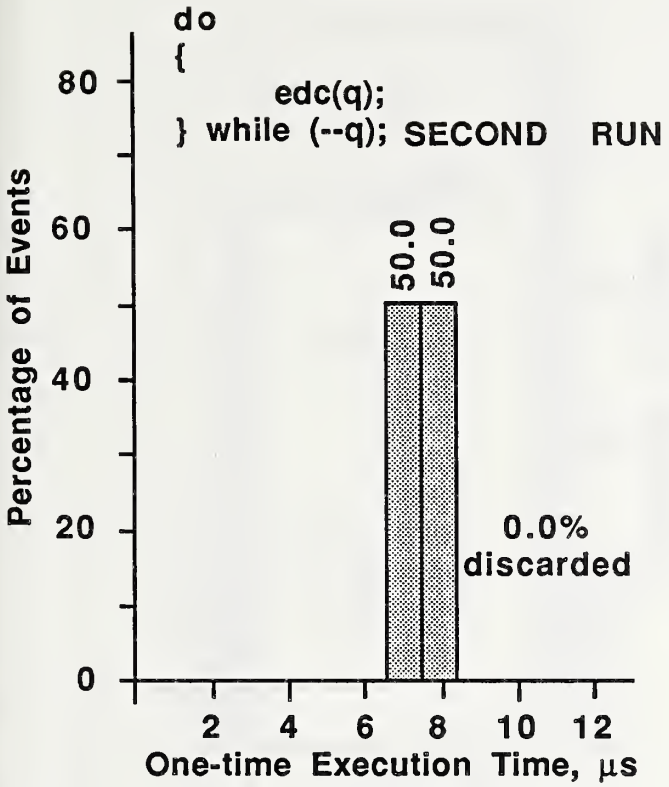

Fig. 6a

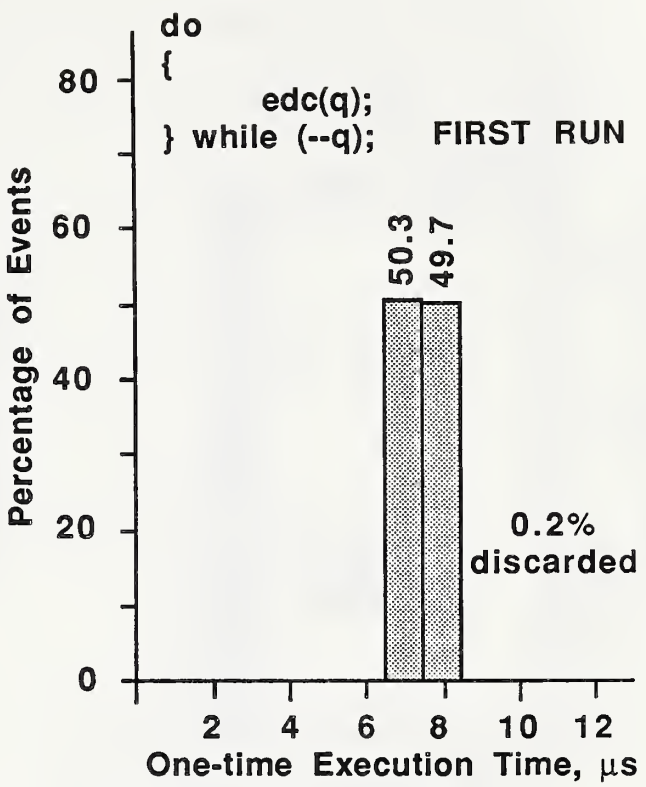

Fig. 6b

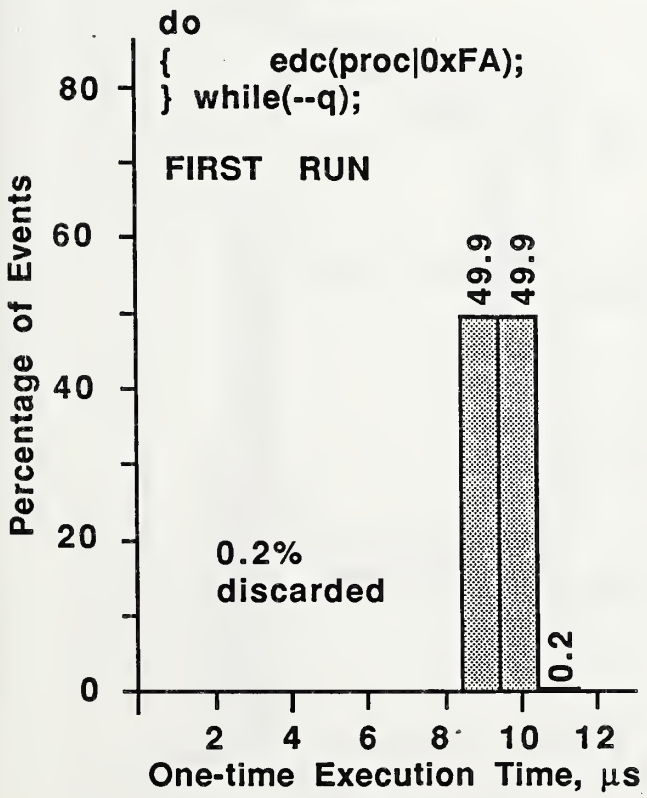

Fig. 7a

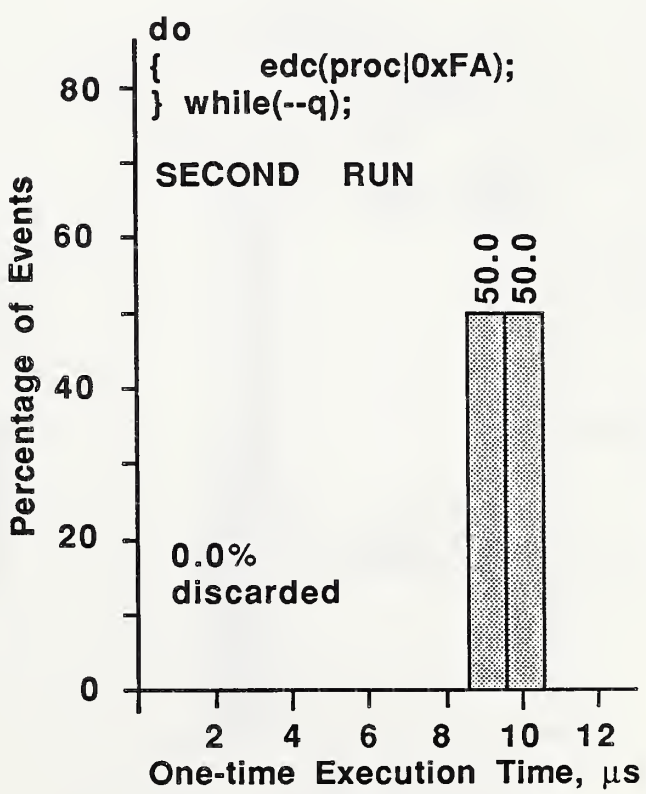

Fig. 7b

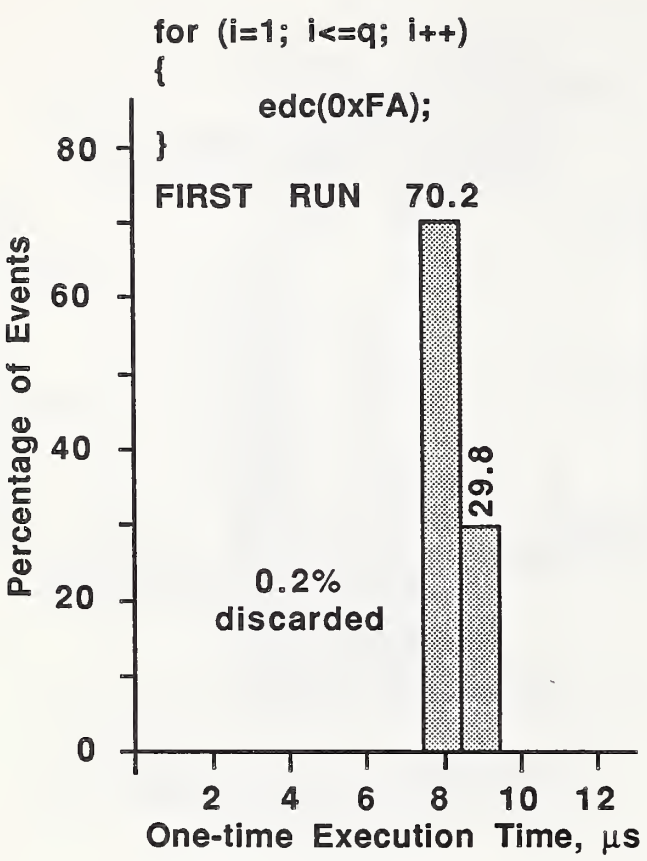

Fig. 8a

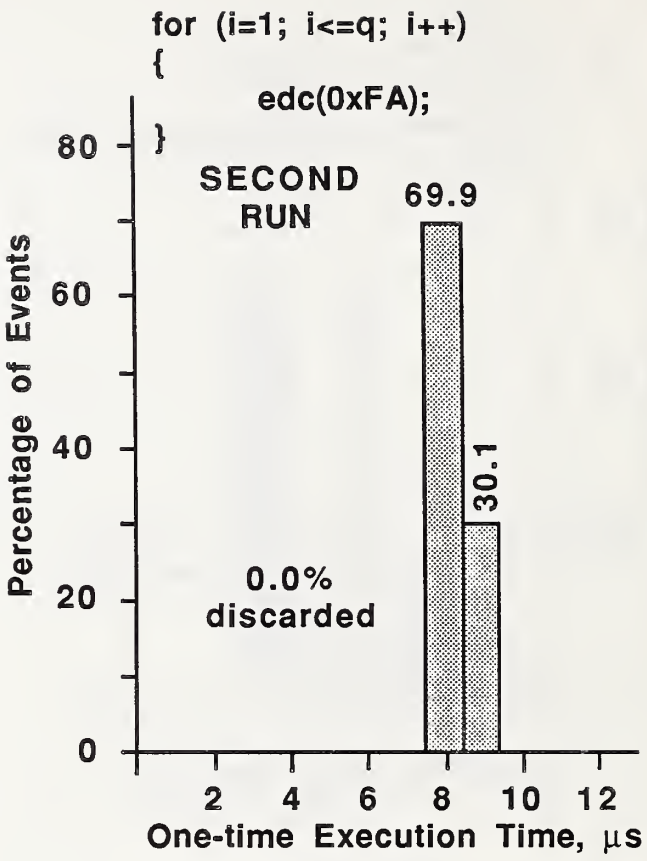

Fig. 8b

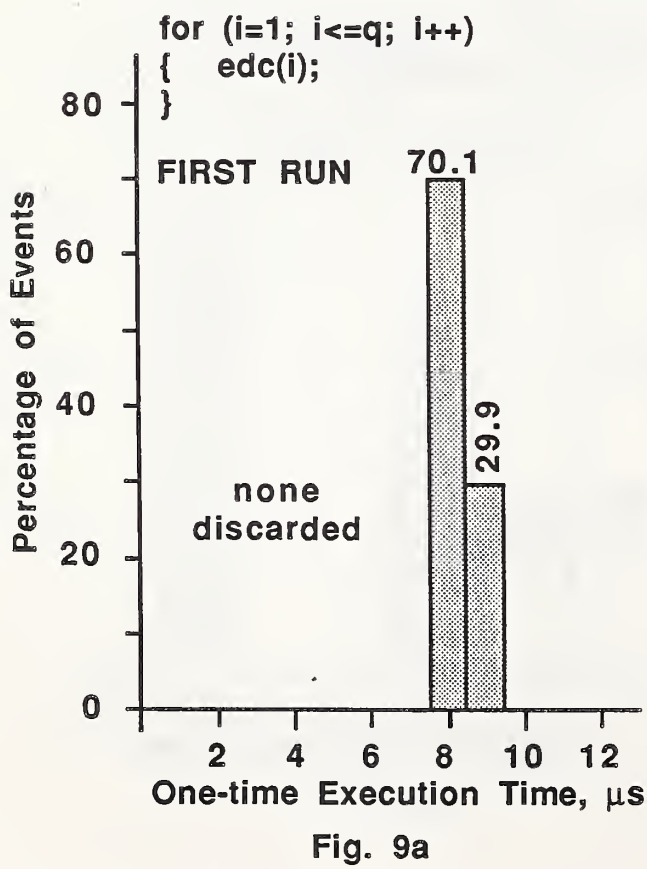

Fig. 9a

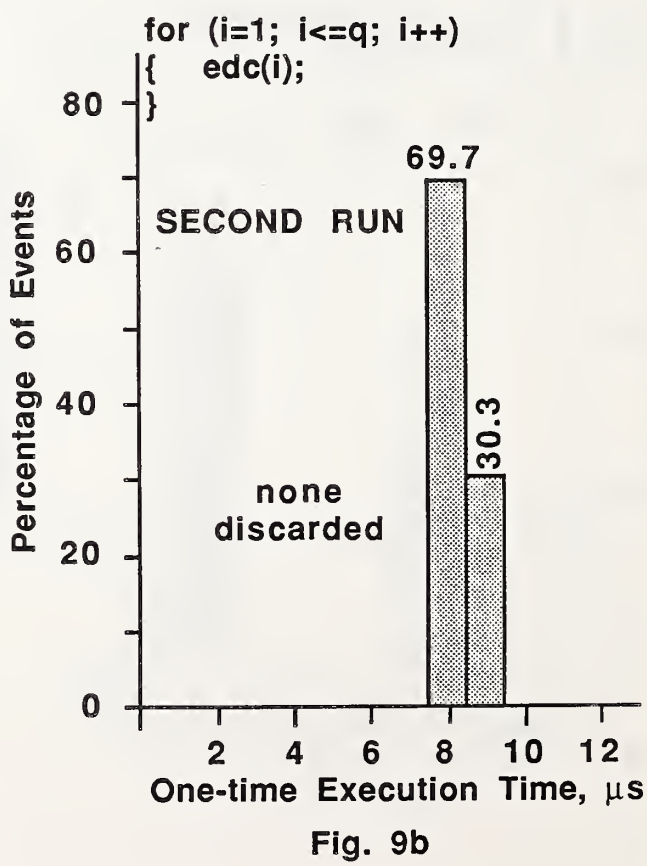

Fig. 9b

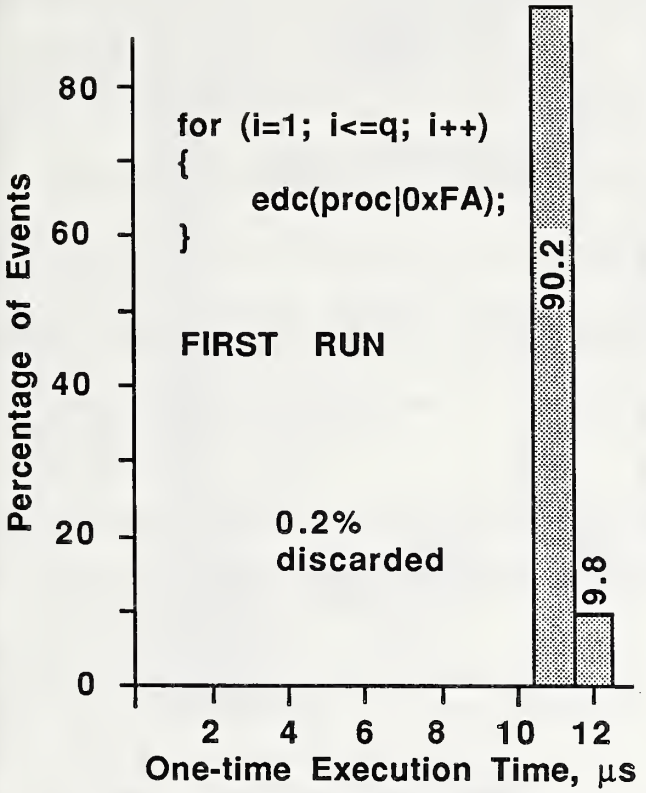

Fig. 10a

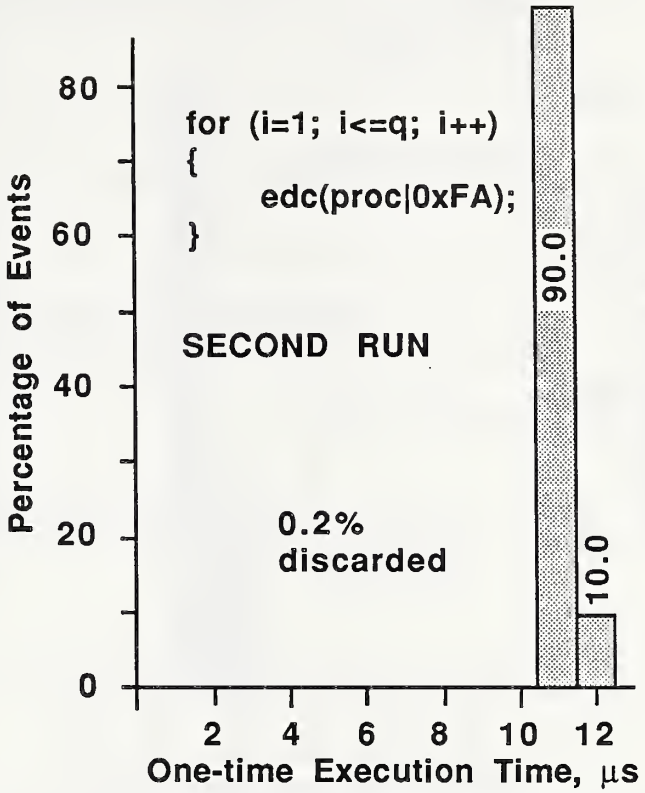

Fig. 10b

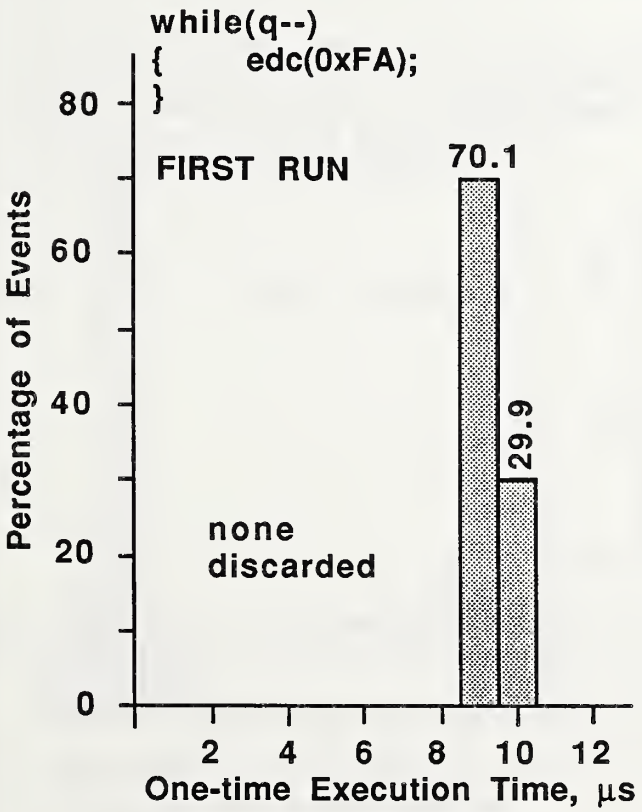

Fig. 11a

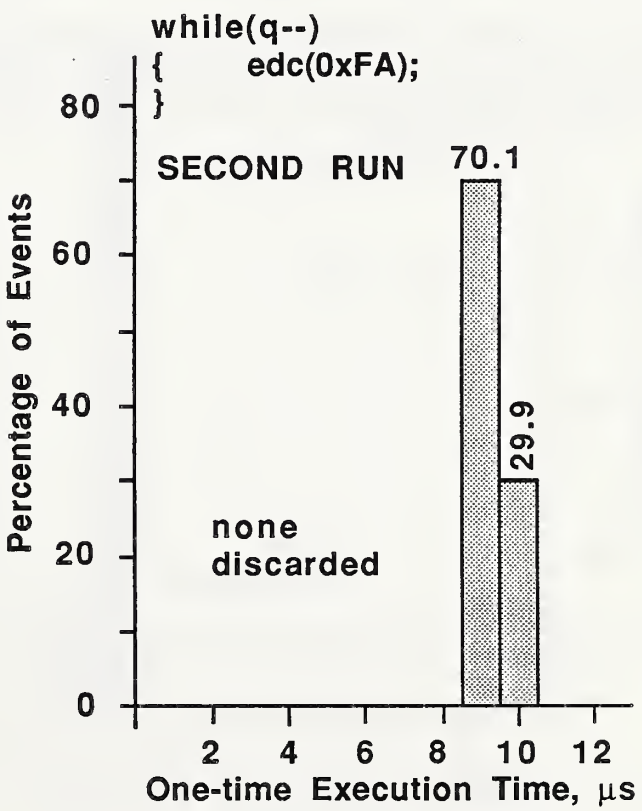

Fig. 11b

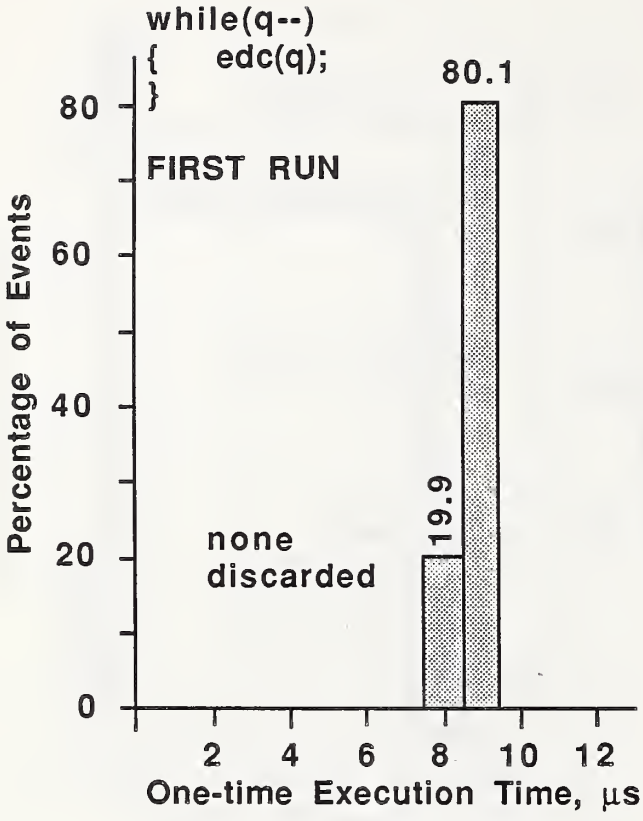

Fig. 12a

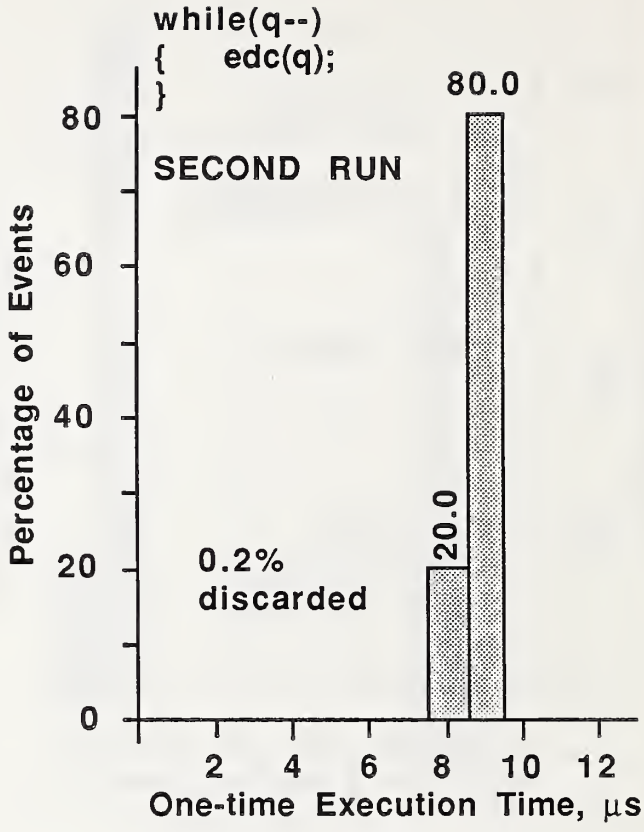

Fig. 12b

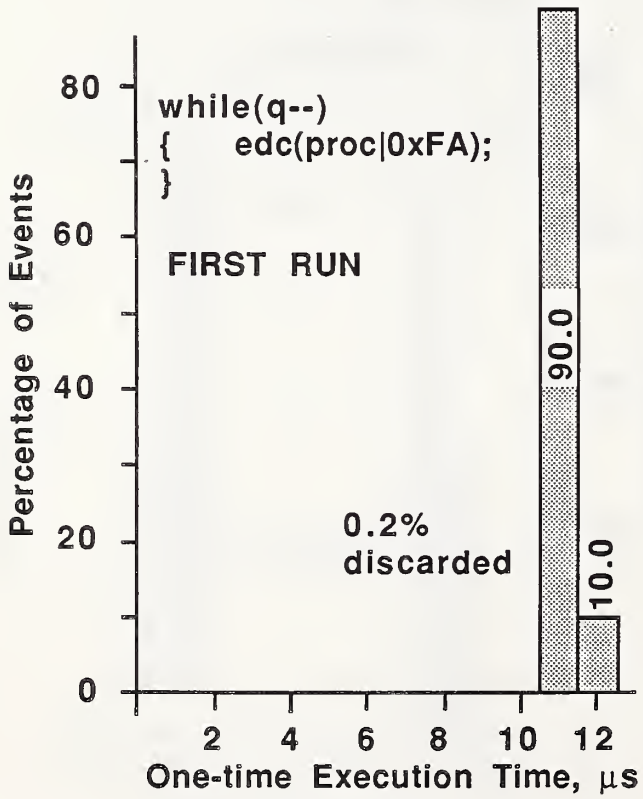

Fig. 13a

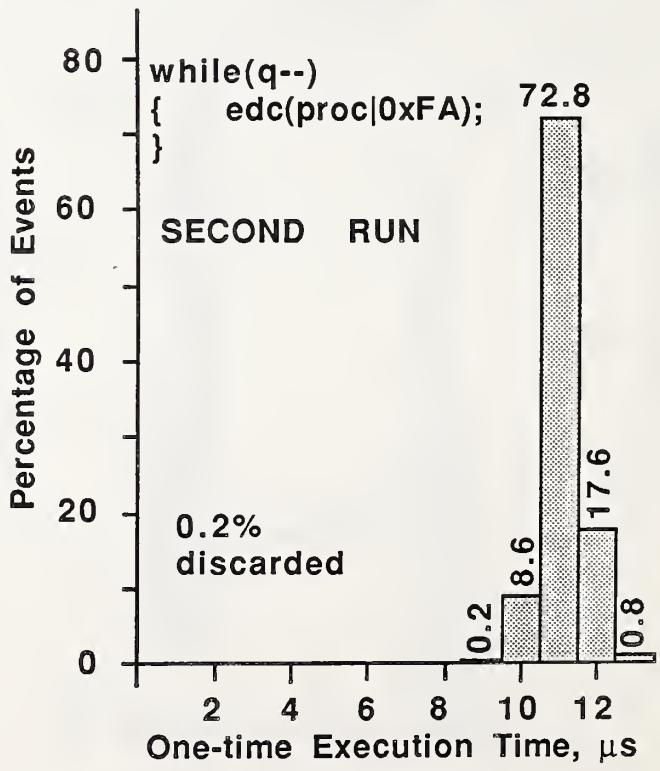

Fig. 13b

NBS-114A (REV. 2-8C)

U.S. DEPT. OF COMM. BIBLIOGRAPHIC DATA SHEET (See instructions)	1. PUBLICATION OR REPORT NO. NBSIR 86-3416	2. Performing Organ. Report No.	3. Publication Date JULY 1986

4. TITLE AND SUBTITLE

Simple Multiprocessor Performance Measurement Techniques and Examples of Their Use

5. AUTHOR(S)

Alan Mink, John W. Roberts, Jesse M. Draper, Robert J. Carpenter

6. PERFORMING ORGANIZATION (If joint or other than NBS, see instructions)

NATIONAL BUREAU OF STANDARDS
DEPARTMENT OF COMMERCE
WASHINGTON, D.C. 20234

7. Contract/Grant No.

8. Type of Report & Period Covered

9. SPONSORING ORGANIZATION NAME AND COMPLETE ADDRESS (Street, City, State, ZIP)

Defense Advanced Research Projects Agency
1400 Wilson Boulevard
Arlington, Virginia 22209

10. SUPPLEMENTARY NOTES

☐ Document describes a computer program; SF-185, FIPS Software Summary, is attached.

11. ABSTRACT (A 200-word or less factual summary of most significant information. If document includes a significant bibliography or literature survey, mention it here)

This report describes simple hardware measurement techniques for the measurement of the performance of multiprocessor computers. A number of examples of data obtained using these techniques are reported, as well as an indication of the timing accuracy obtainable with this approach.

12. KEY WORDS (Six to twelve entries; alphabetical order; capitalize only proper names; and separate key words by semicolons)

hardware; multiprocessor computers; parallel computers; performance measurement

13. AVAILABILITY

☒ Unlimited

☐ For Official Distribution. Do Not Release to NTIS

☐ Order From Superintendent of Documents, U.S. Government Printing Office, Washington, D.C. 20402.

☒ Order From National Technical Information Service (NTIS), Springfield, VA. 22161

14. NO. OF PRINTED PAGES

19

15. Price

$9.95

USCOMM-DC 6043-P80