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

NBSIR 87-3585

Institute for Computer Sciences and Technology

Hardware-Assisted Multiprocessor Performance Measurements

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

Advanced Systems Division

June 1987

CMRF

COMPUTER MEASUREMENT
RESEARCH FACILITY
FOR HIGH PERFORMANCE
PARALLEL COMPUTATION

Sponsored by the
Defense Advanced Research Projects Agency
under ARPA order number 5520,
July 23, 1985, and July 28, 1986.

Hardware-Assisted Multiprocessor Performance Measurements

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

Advanced Systems Division
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

June 1987

Hardware-Assisted Multiprocessor Performance Measurements

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

This report describes the implementation and use of a hardware-assisted trace measurement system (TRAMS) used to obtain performance measurements of parallel cooperating processes executing on a multiprocessor computer. The benefit of TRAMS is that the overhead required to obtain timing information is approximately two orders of magnitude better than the standard system timing call, thus providing more accurate results with little perturbation to the measured processes. This level of accuracy allows measurement of fine-grain portions of these parallel processes which cannot be reasonably measured using standard techniques, and are therefore usually presented as negligible. Some measurements that have been obtained using TRAMS on a tightly-coupled, shared-memory parallel processor are reported here and include basic programming constructs, process creation, process synchronization, and shared memory allocation.

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

1. INTRODUCTION

Ferrari [1] has classified measurement tools as either hardware or software/firmware based. In his taxonomy non-interference and high resolution are the main advantages of hardware tools, since they use probes to capture information from the system under test without any time or space interference. While software/firmware tools do cause interference to the system under test, they provide more flexibility than the hardware tools since the measurements obtained can be tailored to the specific application. Ferrari also states [1,pg 46] that these two classes of tools are complementary, and suggests that an integrated tool could provided the most convenient solution to a measurement

---

* Present address of Jesse M. Draper is Supercomputing Research Center, Lanham, Maryland.

This work was partially sponsored by the Defense Advanced Research Projects Agency.

technique.

The traditional software approach to instrument a program for measuring performance is to insert supervisory 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. In particular, additional execution time is required for the operating system to provide the timestamp and for the code added to the program being tested to store the timestamp with the data describing the event. This time overhead is on the order of hundreds of microseconds, or worse. In a uniprocessor running a non-time critical application such time perturbations merely increases execution time, but does not otherwise effect the execution path of the program. In a multiprocessor and in a uniprocessor running real-time applications it can greatly distort the execution path and, therefore, dramatically change the performance.

Schrott [2] in an attempt to measure the performance of a real-time system, consisting of a single processor, has used an integrated hardware and software approach to measurement. In his approach a second processor is used to capture event data generated by the real-time processor. The time ordered sequence of entrances and exits to the operating system are the events of interest. The operating system is instrumented at each entrance and exit point with code (one or two instructions) that writes to the second processor via an interprocessor link. The purpose of the second processor is to minimize the interference to the real-time system and not alter its timing behavior, while capturing the timestamped data for on the fly or later reduction and processing.

A similar problem exists for multiprocessor systems where the timing behavior of the activities on each of the processors is critical to performance measurement of such systems. In a multiprocessor environment a software measurement approach can greatly distort synchronization of cooperating parallel processes. Timing overhead is suffered only by the processor asking the operating system for the timestamp information; all other processors proceed without delay. The higher the measurement overhead that is introduced, the more pronounced the effect. In the worst case these synchronization delays can produce a domino effect that can make the instrumented program behave completely differently from the uninstrumented version. Hardware tools could be used to measure "microscopic" events such as cache hit ratios, bus delays, bus duty cycles, etc., just as they can for uniprocessor systems. When the activities of the application or system processes on multiprocessor systems are of interest, then these "macroscopic" events cannot be effectively measured by traditional software measurement approaches. Therefore, the problem is to obtain timestamped event data with minimum interference to the timing behavior of each of the processes.

An integrated hardware and software measurement tool developed for the CONCERT multiprocessor system [3] is described by Mitchell [4]. The CONCERT multiprocessor is comprised of a set of clusters, each cluster consists of a group of tightly coupled processors with shared memory. For each cluster a hardware device is accessible via the shared memory bus. This device is designed to accumulate event timing information during the execution of a program and the results can be read back after the program terminates. The user must embed software commands in the program indicating to the device the beginning and end of an event along with the event number. The device preprocesses the time information yielding a fix set of metrics. As in Schrott's approach, the embedded software minimally perturbs the original program.

Again, an integrated hardware and software tools approach seems the most promising, and our specific approach is discussed in the remainder of this paper. Unlike Schrott, we have provided a general software interface to the hardware device that captures and timestamps the event data. Rather than embedding the software interface only in the operating system, we have provided the applications programmer as well as the systems programmer access to this measurement tool, similar to Mitchell's approach. Unlike Mitchell's approach we have left the information describing the event to be defined by the user, rather than predefined, and in addition we capture each individual timestamped event rather than preprocess them into fixed performance metrics. In addition to the timestamp, each event is also processor stamped to indicate the processor it is currently executing on, since precesses can migrate among the various processors. As a result we can also reconstruct the specific sequence of events, in addition to computing performance metrics. The TRAce Measurement System (TRAMS) is designed as a memory-mapped device which maintains its own clock for timestamping. From the programmer's point of view TRAMS is treated as a memory location which, when written to, will store and timestamp the information designating an event. The interference incurred by the additional TRAMS measurement software added to the program is as little as a single memory assignment statement per timestamp.

The TRAMS concept is quite general and can be applied to various multiprocessor architectures. In this paper we describe our implementation of TRAMS for a tightly-coupled, shared-memory multiprocessor system. We also present a sample of the various performance measurements made on our multiprocessor system using TRAMS. We made fine-grain measurements on the overhead introduced by TRAMS itself and on standard looping language constructs, followed by measurements of larger-grain system activities such as process creation, synchronization locking, and software timing. The final measurements show the overhead involved in dynamically initializing and allocating shared memory for a large-grain application program.

2. TRAMS IMPLEMENTATION

The overall configuration of TRAMS is illustrated in Figure 1. At selected test points in the programs, *events* are written to a specially designed board called the Event Data Card, which has been installed in the normal I/O space of the system under test. These writes are called *edc* measurement statements, and contain data specified by the user to identify an event such as process ID, values of variables, or other state information. Only a few microseconds are required for the computer being measured to execute each *edc* statement, since no supervisory calls are made to the operating system. (The system being tested contains eight commercial 32-bit processors with separate floating point and memory management units, running at 10 MHz.) In our current implementation the event data is fixed at 11 bits. Used as one field, this is sufficient to identify 2048 unique events; used as two fields of 3 and 8 bits each, it can identify 8 separate processes and 256 unique events. The Event Data Card then adds hardware signals including a 32-bit microsecond timestamp, a 4-bit processor identification, and a 1-bit user/supervisor mode status, and sends a trigger signal to a FIFO to enter the data into its 48-bit-wide buffer. Thus, in addition to being plugged into the IEEE 796 bus of the system under test, the EDC has a number of direct connections (probes) to each of the processors. These probes are necessary to obtain state information that is not available on the bus (e.g., processor identification). While the computer under test could read the data out of the FIFO at a later time, that task is currently performed by a separate analysis computer which can either store the raw data for later processing or process it immediately.

Obtaining measurements with TRAMS requires familiarity with the application program being measured. The programmer determines the events to be measured and the data that will be output to the Event Data Card to identify each event. Intervals and frequencies are two major categories of metrics. 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 (e.g., the number of times per second that a loop is executed. In more complex variants, two or more events are necessary to determine ratios (e.g., that part of a loop accounts for 90% of the loop's execution time). The programmer must insert special *edc* measurement statements in the source code of the application program. These statements write data to the Event Data Card identifying each measurement event. An analysis program must be written that will take the timestamped application-specific data, match interval boundaries of each event, and output a statistical summary for each event. We have primarily been using a fixed format for the event data (e.g., process ID and event number) and, therefore, have developed a table-driven analysis program which can be used by only modifying the table describing the events without modifying the program.

The major design goal of the Event Data Card was to produce a simple, straightforward interface between the multiprocessor system being tested and the measurement storage and analysis equipment, and to develop this interface with a minimum of design effort. This goal was accomplished by use of commercially available assemblies. The Event Data Card logic consists of an IEEE 796 interface, an address recognizer, a data latch, a timestamp counter, a synchronization circuit, an interface for hardware probes, and logic for intermediate processing of signals from the hardware probes. The availability of a reliable 10MHz clock signal on the backplane makes it unnecessary to include a clock circuit. The interfaces between these modules are reasonably simple. To minimize the implementation effort, we chose a logic analyzer as the FIFO in our TRAMS, which had the added benefit of simplifying the EDC output function. Since the output signals are tapped directly from the appropriate locations on the EDC board by the logic analyzer, output drivers are not needed.

The data latch has sixteen outputs, which are currently wired to take in eleven bits of software-determined data from the processor initiating the write, plus five bits of data from the hardware probe logic. In the current implementation of the Event Data Card, the hardware probes are used to identify the processor initiating any particular write to the measurement system, and to determine whether that processor is operating in user or supervisor mode. Because of pipelining in the system under test, the hardware probe logic must use a register to retain the state information until it is written.

In our current implementation a logic analyzer with a 512-word buffer is used as the FIFO to store event data. One complete item of data, including the timestamp, is captured every time it is triggered. The logic analyzer link in the measurement chain suffers from limitations in capacity, speed, and functionality. Its 512-word buffer size is a serious limitation; a much larger on-board memory would be considerably better. In our current implementation the effective transmission speed of event data from the logic analyzer to the analysis computer via the IEEE 488 interface is on the order of seconds for the entire buffer contents. Moreover, the logic analyzer cannot capture samples while it is transmitting its buffer contents via the IEEE 488 interface. Nor can it report the number of samples it has captured or lost; therefore, it is important to plan each experiment carefully around this limitation.

3. LOW LEVEL MEASUREMENTS

A series of experiments were planned to test TRAMS 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;
```

3.1 Determination of Measurement Overhead

The first test 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* measurement statement.

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

The second variation used a variable.

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

The third variation used a constant ORed with a variable.

```
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 a process number and the constant may represent a specific event.

3.2 Measurement of Loop Structures

We next measured the intervals between successive *edc* measurement statements embedded in typical programming looping structures. The second experiment consisted of a single *edc* statement embedded 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* statement measured the speed of the DO-WHILE construct plus one *edc* execution. The third experiment consisted of a single *edc* statement embedded in a WHILE loop construct.

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

The fourth experiment consisted of a single *edc* statement embedded 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.

3.3 Data Analysis

Our analysis program converted the raw timestamp data to intervals. On the first pass it computed statistics for all interval data. On its second pass it 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, in which the processor serviced some other task in the middle of the execution of the test loop structure. For each experiment the statistics include the number of culled sample points (events), their range and mean. The distribution of the values for the culled data is provided for later plotting (See figures 2 - 13). This includes the percentage of data in each one-microsecond interval.

3.4 Results

Each experiment shows a narrow range of time for the event being measured, which is consistent with 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 clock on the Event Data Card and the 10-MHz clock on the system under test. Second, a variation of a few microseconds 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 figures 2 - 13.

Figures 2 - 4 show the results from the first experiment, which measured the speed of the *edc* statement for three different types of arguments. In the first case, writing a simple constant (See figure 2) took approximately 3 microseconds. 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. Writing a simple variable (See figure 3) took about 3 - 4 microseconds, and writing a variable ORed with a constant (See figure 4) took an average of 6 microseconds.

We next measured the intervals between successive *edc* measurement statements embedded in typical programming loop structures. The histograms in figures 5 - 13 show results ranging from 7 to 11 microseconds for various loop structures. 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 embedded instructions. The appropriate columns in Table 1 show the substantial consistency of the results.

4. OVERHEAD MEASUREMENTS

Parallel programming on a multiprocessor requires some knowledge about the costs of parallelism. Creating and synchronizing parallel processes incurs overhead that must be weighed carefully against the benefits of concurrency. Shared memory itself can be expensive to implement, even without the effects of contention. In the following discussion we first describe the costs of software timing and then give details about the overhead for some of the most important functions involved in setting up and running a divide-and-conquer parser consisting of six parallel cooperating processes.

4.1 Software Timing

A software timing call requires some overhead (obviated by TRAMS), which we measured using TRAMS. Table 2 shows maximum, minimum, average, and median times for the two basic Berkeley UNIX[1] timing calls, *gettimeofday* and *getrusage*. Each of the system calls is at least two orders of magnitude slower than the six-microsecond writes to TRAMS, although *getrusage* provides considerably more information.

Table 2a

Run	Software Timing Overhead (microseconds) Calls to gettimeofday()				
	Number of Samples	Maximum	Minimum	Average	Median
1	170	999	659	682.08	660.00
2	170	989	647	669.47	648.00
3	170	985	647	668.40	648.00
4	169	1050	648	670.80	648.00
5	169	985	647	670.53	648.00
TOTAL	848	1050	647	672.26	649.00

1. UNIX is a trademark of Bell Laboratories.

Table 2b

Run	Software Timing Overhead (microseconds) Calls to getrusage()				
	Number of Samples	Maximum	Minimum	Average	Median
1	170	1218	857	885.01	857.00
2	170	1209	839	868.65	839.00
3	170	1186	841	866.92	842.00
4	170	1191	841	866.79	841.00
5	170	1220	842	869.87	843.00
TOTAL	850	1220	839	871.45	843.00

These measurements show that a hardware-assisted trace measurement system can be dramatically less perturbing than software measurements.

4.2 Synchronization

We measured locking and unlocking overhead during the period when one process was doing useful work and the other five identical processes were in constant contention for a single lock guarding a critical section. Because the locking call spins until it succeeds, the locking times reported in Table 3a directly reflect the contention. Each of the populations included a significant number of long times that probably reflect device interrupts. As a consequence, we have reported both the number of anomalies (values at least an order of magnitude greater than the minimum) and the median, which in this case may give a more reliable indication of contention times than the mean. Calls to unlock the lock generally take between 18 and 22 microseconds (see Table 3b), with a median of 19 microseconds for all seven programs. If we subtract six microseconds (the average overhead of one write to TRAMS) from the median, we get 13 microseconds as the real cost of unlocking a lock. Direct measurements of locking and unlocking a lock without contention give mean values of 41 and 16 microseconds, respectively. Subtracting the measurement overhead from each of these produces 35 and 11 microseconds, respectively; hence, the minimum synchronization overhead for this multiprocessor is 46 microseconds plus the time to execute the critical section.

Table 3a

Longest Parsable String	Locking Overhead (microseconds)					
	Number of Samples	Number of Anomalies	Maximum	Minimum	Average	Median
50	500	31	2477	43	197.85	130.00
100	500	29	1562	44	180.76	123.00
200	500	17	948	44	172.29	123.00
300	500	22	945	44	181.21	123.00
400	500	31	816	44	181.51	123.00
500	501	32	1122	47	184.92	128.00
600	500	29	2211	43	192.61	125.00

Table 3b

Longest Parsable String	Unlocking Overhead (microseconds)					
	Number of Samples	Number of Anomalies	Maximum	Minimum	Average	Median
50	516	2	2267	18	23.97	19.00
100	516	3	455	18	20.97	19.00
200	516	0	22	15	19.00	19.00
300	516	3	726	17	21.66	19.00
400	516	0	22	18	19.04	19.00
500	516	0	24	18	19.02	19.00
600	516	2	1993	18	23.53	19.00

4.3 Process Creation

Some strategically placed print statements had earlier shown that process creation was taking considerably longer for programs that could parse long strings than for those with less capability. We therefore preceded each "fork ()" system call by a write to TRAMS. For each child the first executable instruction is another write to TRAMS to identify the return from the fork call. Since the size of the program's parsing tables varies dramatically with the size of the longest input string that can be parsed, we measured seven different versions. We ran each version four times, keeping the input string constant and small enough that even the smallest program could parse it. Table 4 shows the maximum, minimum, and average fork times for each program.

Table 4

Longest Parsable String	Shared Memory Size (Kbytes)	Number of Samples	Overhead for Process Creation (milliseconds)		
			Maximum	Minimum	Average
50	33	24	166	121	130
100	125	24	162	127	137
200	490	24	178	149	161
300	1096	24	203	191	198
400	1941	24	288	197	254
500	3026	24	360	317	329
600	4351	24	426	401	410

For comparison, we measured a simple kernel with similar data requirements but no shared memory. Process creation took approximately 90 milliseconds and did not increase with data size, even though for each fork both program and data would presumably be copied in their entirety rather than shared. Whatever the cause of the high overhead for creating processes that share memory, it appears that, on this system, programs with large shared data structures will require coarse-grain rather than fine-grain parallel tasks. Otherwise, the overhead of process creation will be too costly to recoup from process concurrency.

4.4 Dynamic Initialization and Allocation of Shared Memory

The computer under test has both static and dynamic shared memory. We could not measure the overhead of static initialization and allocation of shared memory without instrumenting the operating system since those events occur before the beginning of the "main" program. We could, however, measure the times for the calls to library routines that dynamically initialize and allocate shared memory. As Tables 5a and 5b show, those times can be quite long for programs with large requirements for shared memory. As a result we were able to establish that static shared memory is *considerably* faster in this system.

Table 5a

Longest Parsable String	Shared Memory Size (Kbytes)	Number of Samples	Overhead for Shared Memory Initialization (milliseconds)			
			Maximum	Minimum	Average	Median
50	33	41	389	184	209	189
100	125	42	450	283	337	337
200	490	40	1773	431	923	1002
300	1096	42	5830	701	2113	965
400	1941	40	4753	1083	1949	1323
500	3026	40	3837	1568	1870	1602
600	4351	43	6965	2128	2779	2330

Table 5b

Longest Parsable String	Shared Memory Size (Kbytes)	Number of Samples	Overhead for Shared Memory Allocation (milliseconds)			
			Maximum	Minimum	Average	Median
50	33	41	83	71	72	72
100	125	42	271	265	267	267
200	490	40	1080	1032	1050	1047
300	1096	41	2862	2597	2652	2621
400	1941	40	12891	5054	5421	5225
500	3026	40	16960	9025	9552	9354
600	4351	40	14888	13359	13827	13753

These values, like those for process creation, suggest that on the system under test programs with large requirements for shared memory should use coarse-grain rather than fine-grain parallelism.

5. LIMITATIONS AND FURTHER APPLICATIONS

The current TRAMS implementation is useful for measurements on uniprocessors or multiprocessors using a globally-available IEEE 796 bus. It is possible, however, to enhance the system in several straightforward ways. Providing substantial memory on the Event Data Card itself would make it possible to collect more samples per run and hence remove some of the current constraints on how much we can measure at a single time. Users who need different measurements could change both the counting rate of the timestamp counter and the allocation of bits for processor data and hardware signals.

To expand the system substantially or to apply it to a non-global-memory multiprocessor architecture would require redesign. The principles of the design, however, can be applied to a wide range of multiprocessor architectures. The basic philosophy is to build a general hardware assist to the standard software measurement approach that will minimize both the requirements for hardware design and the perturbation of the system under test. Similar memory-mapped devices can be installed in many systems. Since the TRAMS approach is reasonably simple, it could be applied to each processor in a distributed system.

The TRAMS concept allows the user to identify the process number and event in the data field of the *edc* measurement statement, while the Event Data Card appends a timestamp, a processor ID, and a bit to tell whether the processor is in user or supervisor state. The Event Data Card could be further simplified if multiprocessor manufacturers would provide both the processor ID and the user/supervisor state for the current bus cycle on the bus itself. This addition would eliminate the need for hardware probes and their associated logic.

6. SUMMARY

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. The measurement examples of system overhead for parallel programs on the testbed computer show that this multiprocessor being measured is suitable primarily for coarse-grain rather than fine-grain parallelism. While synchronization is not costly, process creation can take almost half a second for programs with large requirements for shared memory. In addition, the costs of dynamically initializing and allocating shared memory make static shared memory considerably more attractive for programming with cooperating parallel processes. Overall, these results make it clear that a simple trace measurement system can provide invaluable information for parallel programming on a multiprocessor.

7. REFERENCES

[1] Ferrari, D. "Computer System Performance Evaluation", Prentice-Hall, Inc., Englewood Cliffs, N.J., 1978.

[2] Schrott, G. and Tempelmeier, T., "Monitoring of Real Time Systems By a Separate processor", Proc. of the 12th Intern'l Federation of Automatic Controls / IFIP Workshop: Real Time Programming 1983, Hatfield, UK, Mar. 1983, pp 69-79.

[3] Anderson, T. L., "The Design of a Multiprocessor Development System", Masters Thesis, MIT, Dept of Electrical Engineering and Computer Science, Sept. 1982.

[4] Mitchell, S., "SySM Functional Requirements Description", Harris Corp., P.O. Box 98000, Melbourne, Fl. 32902, Feb. 1986.

USCOMM-NBS-DC

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

U.S. DEPT. OF COMM. **BIBLIOGRAPHIC DATA SHEET** *(See instructions)*	1. PUBLICATION OR REPORT NO. NBSIR 87-3585	2. Performing Organ. Report No.	3. Publication Date JUNE 1987

TITLE AND SUBTITLE

Hardware-Assisted Multiprocessor Performance Measurements

AUTHOR(S)

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

PERFORMING ORGANIZATION *(If joint or other than NBS, see instructions)*	7. Contract/Grant No.
NATIONAL BUREAU OF STANDARDS DEPARTMENT OF COMMERCE WASHINGTON, D.C. 20234	8. Type of Report & Period Covered

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

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

. SUPPLEMENTARY NOTES

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

. 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 the implementation and use of a hardware-assisted trace measurement system (TRAMS) used to obtain performance measurements of parallel cooperating processes executing on a multiprocessor computer. The benefit of TRAMS is that the overhead required to obtain timing information is approximately two orders of magnitude better than the standard system call, thus providing more accurate results with minimum perturbation to the measured processes. This level of accuracy allows measurement of fine-grain portions of these parallel processes which cannot be reasonably measured using standard techniques, and are therefore usually presented as negligible. Some measurements that have been obtained using TRAMS on a tightly-coupled, shared-memory parallel processor are reported here and include basic programming constructs, process creation, process synchronization, and shared memory allocation.

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

AVAILABILITY	14. NO. OF PRINTED PAGES
[✓ Unlimited]	15
☐ For Official Distribution. Do Not Release to NTIS	
☐ Order From Superintendent of Documents, U.S. Government Printing Office, Washington, D.C. 20402.	15. Price
[✓ Order From National Technical Information Service (NTIS), Springfield, VA. 22161]	$9.95