Conf-831208--1

LA-UR--83-1500

DE83 012778

Los Alamos National Laboratory is operated by the University of California for the United States Department of Energy under contract W-7405-ENG-36.

MASTER

TITLE: CRAY-1 INSTRUCTION ANALYSIS: A COMPARISON OF TWO METHODS

AUTHOR(S): Joanne L. Martin
Tony Warnock

SUBMITTED TO: The Computer Measurement Group XIV Conference, Washington, DC, December 6-9, 1983

DISCLAIMER

Los Alamos

Los Alamos National Laboratory
Los Alamos, New Mexico 87545

CRAY-1 INSTRUCTION ANALYSIS: A COMPARISON OF TWO METHODS

Joanne L. Martin
Computer Research and Applications Group
Los Alamos National Laboratory
and
Tony Warnock
Cray Research, Inc.

ABSTRACT

In an effort to obtain data for workload characterization and performance evaluation studies, statistics were gathered on the frequencies of individual instructions in codes executing on the Cray-1. Two methods were used for the collection of data: (1) direct dynamic counting of executing instructions and (2) sampling of instructions at regular intervals during execution of the code. This paper describes the implementation of both methods and compares the results obtained. We conclude that because the Cray-1 is a vector processor, the sampling technique produces skewed results that misrepresent the workload being examined. Therefore, although it is easier to implement, sampling produces inaccurate results and is an invalid approach to performance evaluation studies on a vector computer. The counting of instructions provides a valuable profile of instruction frequencies and is a stable basis on which to build performance estimations.

I. INTRODUCTION

Perfomance evaluation studies on computer systems must be conducted in an appropriate context. In particular, characterizing the workload that is to execute on the system being examined is a critical step in the evaluation process. In an effort to obtain data for such workload characterization and performance evaluation studies, statistics were gathered on the frequencies of individual instructions in codes executing on the Cray-1. This paper describes the implementation of two methods used for the accumulation of these statistics and compares the results of the two techniques.

There are 128 Cray-1 instructions to be considered. By measuring the frequencies with which individual instructions are issued we are able to develop, at a very low level, a profile of the code that is being executed. Analysis of the information provided allows for the assessment of the relative importance of such machine instructions as floating-point operations, memory accesses, and jumps. Also available is information concerning the manner in which the compiler maps Fortran code to the available architecture.

II. CRAY-1 CONSIDERATIONS

The Cray-1 is a pipelined vector processor, theoretically capable of performing in excess of 100 MFLOPS (millions of floating-point operations per second). This high rate of execution is possible as a result of the combined influence of several architectural features. There are 12 tightly-coupled functional units that potentially can be activated concurrently. These units are able to receive input at every clock period and permit "chaining" of results from functional unit to functional unit.

The clock period of the Cray-1 is an extremely short 12.5 nanoseconds. Vector and scalar instructions may be executed in parallel. All units are driven from a master clock. Thus, the hardware knows exactly when an operation will complete. There are no variable time instructions on the Cray-1; all results are produced in a constant number of clock periods regardless of the data involved. A vector instruction operates on arrays of up to 64 elements, which are held in vector registers. Thus, a vector instruction may require as much as 64 times as long as the corresponding scalar instruction to execute. For these reasons, in the sequencing of instructions, it is beneficial to fit scalar operations into the time spent waiting for a vector instruction to complete.

There are conditions that must be satisfied before individual instructions may be issued. If the issue conditions are satisfied, each instruction completes in a fixed amount of time. Included in these issue conditions are the requirements that the functional unit needed for the operation must be free, the result register must be free, and the operand registers must be free. In the case of vector instructions, the last requirement can be satisfied if the issuance of the second vector instruction occurs at "chain slot time."

The chaining of vector instructions is a process through which one vector operation, which uses the result of a previous vector operation, may not be required to wait for the first operation to complete before beginning to execute. Chaining is performed by having the second vector instruction attempt to issue as soon as the first functional unit yields a result. In this process of chaining instructions, two operations may complete in only slightly more than the time required for the first operation. For chaining to occur, the second vector operation must be ready to issue at what is called the chain slot time. This time is two clock periods greater than the time for the first functional unit.

In a particular chaining situation, there is exactly one chain slot time. The failure of chaining at that time implies that the common register in use is "reserved" by the first functional unit for the duration of the entire operation.

Success of the chaining implies that the result register of the first operation is reserved for the time required to perform the succession of the two operations:

$$t = (t(\text{1st unit}) + t(\text{2nd unit}) - 1) + n,$$

where n represents the vector length [1]. Chained operations thus have a burst processing rate of more than one operation per clock period, depending on the units in the chain.

Instruction issue may imply the reservation of functional units or registers. Scalar instructions are able to reserve only the result register, and this reservation is for a time equal to the execution time of the given instruction. In contrast, vector instructions place reservations on functional units and registers for the duration of instruction execution. Reservation of a functional unit by a vector instruction may delay issuance of scalar instructions.

Finally, the Cray-1 has segmented units for all operations. That is, processing in each unit is partitioned into segments such that the work performed by one segment is completed before information proceeds to the next segment. For example, a floating-point add takes six clock periods (CP) because the floating-point adder has six segments. This segmentation of the functional units implies that operations can be initiated at every clock period. Thus, vector instructions are capable of producing one result per clock period, once the pipe has been filled.

To demonstrate the differences in the execution times for various instructions, selected instruction execution times are presented in Table I [2].

TABLE I. SELECTED INSTRUCTION EXECUTION TIMES

FUNCTIONAL UNIT	SCALAR TIME (CP)	VECTOR TIME (CP)
Logical	1	2
Shift	2	4
Integer Add	3	3
Floating Add	6*	6
Floating Multiply	7*	7
Reciprocal Approximation	14*	14

*Issue may be delayed because of a functional unit reservation by a vector instruction.

III. MEASUREMENT TECHNIQUES

Two methods were used for the collection of data: (1) direct dynamic counting of executing instructions and (2) sampling of instructions at regular intervals during execution of the code. The counting is accomplished through a preprocessor that modifies the assembly language code to incorporate counters, and the sampling is performed through a system utility that is called from the executing program.

Both methods were implemented on each of five test programs. Brief descriptions of these programs are provided in Appendix A. Also, the tests were conducted both in scalar and vector modes. Statistical tests were used to determine the degree of difference in the results obtained by the counting versus the sampling.

The preprocessor, which sets up the counting mechanism, uses as input the CAL (Cray Assembly Language) code that is generated by the compiler. For each line containing a machine instruction, the op code is analyzed and used as a pointer to a table of the 128 possible instructions. A macro is then inserted immediately before the instruction to be counted. The macro increments a counter and, in the case of a vector instruction, records the appropriate vector length. After the counting macro has been inserted for each instruction, the code is reassembled and executed.

Output from this program consists of a table of the 128 machine instructions, the number of times each instruction was issued, the total number of instructions issued, and the percentage of the total for each individual instruction. The MIPS (millions of instructions per second) rate is easily obtainable from this information when combined with the time for execution of the code without the counting mechanisms. We note that the runtime of the code is increased only by a factor of approximately 5 when the counting is instrumented.

The sampling technique uses calls to utility subroutines, in the system library, that are used for generating statistical information within executing codes. In particular, these routines enable an interrupt to stop the program every 4 ms, record from the exchange package the op code of the instruction waiting to issue and the associated vector length, and resume execution.

Output from this technique consists of the number of times each of the 128 instructions was sampled, the total number of samples taken, and the percentages of the total for the individual instructions.

To obtain enough sample points for meaningful statistics, sample files were constructed in two ways. First, 10 sampling runs were made for each of the five programs being tested, in scalar and vector modes. The results of the 10 runs were then compiled into one file that contained the total number of samples for each instruction across the 10 runs. As an alternative method for obtaining a sufficient number of samples, a loop was inserted inside the calls to the sampling routines to increase the number of iterations for the program being tested. Again, runs were made in scalar and vector modes.

IV. STATISTICAL METHODS

The $\chi 2$ - test of goodness of fit was used to examine the closeness of the results obtained on the instruction frequencies by the two measurement techniques. When used on a large sample of a multinomial population of r categories, this statistic conforms approximately to the $\chi 2$ - distribution with r-1 degrees of freedom if the hypothesis being tested is correct [3].

A set of observations that can be described by a finite number of discrete categories is a multinomial population. Suppose that the number of categories is r. The population may be defined by the relative frequencies, $\pi 1, \pi 2, \ldots, \pi r$, of the observations in the r classes. Also,

$$\Sigma \, \pi i = 1, \quad i = 1, \ldots, r.$$

The number of degrees of freedom is defined as the number of independent observations in the sample minus the number of population parameters that must be estimated from sample observations.

The 128 instructions of the Cray-1 comprise the r discrete categories of a multinomial population. The hypothetical frequency, Hi, for each category is defined as the product of the percentage of instructions measured by the counting technique for that category and the total number of samples, N, taken for the given population. That is,

$$Hi = N\pi i, \quad i = 1,\ldots,r.$$

The observed frequency, Oi, is defined as the number of instructions obtained for each category through the sampling technique. Note that

$$\Sigma Oi = N = \Sigma N\pi i, \; i = 1,\ldots,r.$$

Thus, the test of goodness of fit examines the hypothesis that the r relative frequencies, πi, of a multinomial population are equal to specified values. The $\chi 2$ - statistic is defined as

$$\chi 2 = \Sigma \, (Oi - N\pi i)**2 \; / \; N\pi i, \; i = 1,\ldots,r.$$

Observe that if $\chi 2$ is equal to 0, then the hypothetical and observed frequencies agree exactly; if it is greater than 0, they do not. In fact, the further this statistic is from 0, the greater the disparity between the values being tested.

For the comparison of the counting and sampling techniques for measuring Cray-1 instruction frequencies, the number of degrees of freedom was determined by the following definition.

$$(r\text{-}1) - (\# \text{ of } 0 - \text{cases}).$$

Initially, the number of degrees of freedom was 127, r-1. Because there were legitimately cases for which the number of instructions in a particular category was 0, these cases were eliminated from the $\chi 2$ sum.

V. RESULTS

Table II lists summary information obtained by each of the measurement techniques for the five programs under examination. Table III presents the results of the test of goodness of fit for the long sample runs, and Table IV displays the statistics obtained by making a series of 10 short sample runs and summing the results. Finally, Table V presents the acceptable values for a $\chi 2$ - distribution.

Clearly, the long sample runs provide more meaningful statistics than do the summations of the short sample tests. This is because, in the case of the short samples, the interrupts occur at approximately the same time during the successive executions of the code. Thus, if the samples are inaccurate one time, they will be equally inaccurate in the succeeding executions. This permits a cascading of the initial problem, which results in the sum of 10 short sample runs being an order of magnitude worse than the original run.

Although the long sample tests are better than the sums of the short tests, the $\chi 2$ results are still considerably outside an acceptable range for the given number of degrees of freedom. The sampling technique is clearly "seeing" different instruction frequencies than is the counting technique.

VI. CONCLUSIONS

The description of elegant lock-step operation that is presented in Section II of this paper can be marred somewhat by the presence of external interrupts. These occur at random, relative to the instruction sequence, and may destroy the possibility of chaining. This change in the chaining of the instruction sequence then has an effect on the time required to complete a particular operation.

The nonpredictability of interrupts may give misleading results if a sampling routine is used to measure instruction frequencies dynamically. In sampling, an interrupt is generated at constant intervals and the pending instruction is tallied. This should yield a sampling, in time, of the instructions being issued. There are at least two possible distortions introduced by sampling.

First, it is not clear that the sampling rate will be high enough to eliminate the "noise" caused by the various hold-issue conditions on instructions and by the distorting effect of the interrupts themselves. It is possible that the sampling results would be more comparable to the counted results if more frequent interrupts could be issued. This increased frequency of interrupts was not possible through the sampling method used for this study.

Second, because interrupts that tally a vector operation always break chaining between the instruction being tallied and the preceding one, more time will be spent in vector operations than would be the case without sampling. This fact was demonstrated and discussed by D. Wiedemann [4] from which we quote the following example:

"Suppose that a vector multiply is followed immediately by a chainable vector addition. Then, many scalar operations are performed, and this pattern is repeated many times with the vector multiply and add always using vector registers distinct from the ones just previously used. This is illustrated by the instruction sequence of Figure I.

```
v0v1*v2
v2v0+ v3
{N Clock periods of scalar instructions}
v4v5*v5
v6v4+ v7
{N Clock periods of scalar instructions}
v0v1*v1
v2v0+ v3
.
.
.
```

Fig. 1. Example of Cray-1 program with overlapping vectors and scalars."

The intent of this example is to show that the vector operations would chain and the scalar instructions would finish while waiting for the vector multiply functional unit to become free. Assuming no interrupts, the program would continue in this optimized fashion.

The issuance of an interrupt just after the issuance of the vector multiply will imply that the next vector addition will not chain, but will begin when the program resumes execution. This resumption will occur long after the multiply has completed. Subsequent scalar operations will overlap with the addition, and because the vector multiply unit is no longer busy, the next vector multiply will issue immediately afterwards. However, because the previous addition is still finishing, the issue of the next vector addition will be delayed. Having missed the opportunity to chain, the vector addition must wait for the vector multiply to complete.

The beginning of the addition returns the instruction sequence to a point similar to the return from interrupt. The execution pattern will repeat and no vector chaining will occur. This situation will persist until the next interrupt, which causes a return to the original mode. The presence of a small amount of interference causes the program to execute much slower than might have been estimated.

We conclude that because the Cray-1 is a vector processor, the sampling technique produces skewed results that misrepresent the workload being examined. Therefore, although it somewhat easier to implement, sampling produces inaccurate results and is an invalid approach to performance evaluation studies on a vector computer. The counting of instuctions provides a valuable profile of instruction frequencies and is a stable basis on which to build performance estimations.

REFERENCES

[1] Jean-Loup Baer, Computer Systems Architecture, (Computer Science Press, 1980).

[2] CAL Assembler Version 1, Reference Manual (Cray Research, Inc., 1981).

[3] Jerome C. R. Li, Statistical Inference I, (Edwards Brothers, Inc., 1964).

[4] Douglas Wiedemann, "Stability of Computer Timing," Digital Processes, 6 (1980) 297-303.

TABLE II
(PROGRAM 1)
INSTRUCTION COUNTS AND FREQUENCIES (SUMMARY INFORMATION)

	Counts		Frequencies (Long Sample Runs)		Frequencies (Sum of 10 Short Sample Runs)	
	Vector	Scalar	Vector	Scalar	Vector	Scalar
Total Jump Count	3,196,217	12,604,721	4,685	4,727	341	701
Address Computation	39,094,089	71,760,628	44,851	22,686	4,080	4,067
Integer Scalar Arithmetic	17,894,075	27,379,889	18,618	13,101	1,878	2,054
Register Fetches	5,146,267	5,146,265	6,976	4,781	565	752
Register Stores	1,740,837	1,740,837	2,307	1,395	203	241
Scalar Fetches	2,974,725	41,605,637	4,930	19,182	535	3,012
Scalar Stores	8,761,299	36,461,011	6,868	11,944	796	1,891
Scalar Flops	4,629,317	75,081,022	3,444	22,441	213	3,576
Integer Vector Arithmetic	25,609	0	302	0	33	0
Vector Flops	9,543,809	0	16,987	0	1,998	0
Register Transfers	37,140,321	177,726,236	43,242	72,783	3,515	10,049
Vector Fetches	5,504,017	0	5,028	0	517	0
Vector Stores	3,968,017	0	5,518	0	565	0

TABLE II
(PROGRAM 2)
INSTRUCTION COUNTS AND FREQUENCIES (SUMMARY INFORMATION)

	Counts		Frequencies (Long Sample Runs)		Frequencies (Sum of 10 Short Sample Runs)	
	Vector	Scalar	Vector	Scalar	Vector	Scalar
Total Jump Count	13,866,308	19,222,262	6,331	5,626	993	1,368
Address Computation	12,486,731	15,071,426	12,309	2,458	2,612	1,579
Integer Scalar Arithmetic	28,247,078	119,724,471	16,693	23,820	2,997	5,822
Register Fetches	11,210,267	11,209,517	12,916	7,948	2,141	1,959
Register Stores	782	782	7	154	50	98
Scalar Fetches	79,405,039	121,606,039	48,663	37,031	7,337	8,912
Scalar Stores	33,021,239	64,390,959	17,916	15,995	2,886	3,859
Scalar Flops	64,667,978	165,145,353	25,352	43,402	3,768	10,473
Integer Vector Arithmetic	655,305	0	2,181	0	347	0
Vector Flops	1,570,375	0	6,453	0	1,301	0
Register Transfers	45,739,275	176,104,107	22,175	36,190	4,212	11,170
Vector Fetches	605,375	0	1,034	0	251	0
Vector Stores	505,555	0	697	0	161	0

TABLE II
(PROGRAM 3)
INSTRUCTION COUNTS AND FREQUENCIES (SUMMARY INFORMATION)

	Counts		Frequencies (Long Sample Runs)		Frequencies (Sum of 10 Short Sample Runs)	
	Vector	Scalar	Vector	Scalar	Vector	Scalar
Total Jump Count	3,687,058	12,823,012	1,106	3,262	148	817
Address Computation	9,407,356	14,752,551	23,132	5,545	3,060	1,487
Integer Scalar Arithmetic	26,003,828	139,567,596	14,444	32,901	1,962	8,743
Register Fetches	2,572,767	2,571,267	646	330	82	92
Register Stores	782	782	9	168	5	38
Scalar Fetches	52,807,664	110,086,664	28,127	24,553	3,625	6,463
Scalar Stores	17,975,614	73,345,334	10,368	17,121	1,395	4,392
Scalar Flops	57,307,978	165,465,353	36,025	49,738	4,885	13,273
Integer Vector Arithmetic	896,055	0	4,554	0	602	0
Vector Flops	1,690,375	0	11,250	0	1,545	0
Register Transfers	24,839,025	123,308,482	18,160	38,874	2,399	9,877
Vector Fetches	735,750	0	1,598	0	210	0
Vector Stores	895,930	0	1,987	0	265	0

TABLE II
(PROGRAM 4)
INSTRUCTION COUNTS AND FREQUENCIES (SUMMARY INFORMATION)

	Counts		Frequencies (Long Sample Runs)		Frequencies (Sum of 10 Short Sample Runs)	
	Vector	Scalar	Vector	Scalar	Vector	Scalar
Total Jump Count	12,860,900	14,473,848	12,238	9,772	1,355	1,614
Address Computation	23,882,113	24,502,288	15,072	12,553	2,004	1,725
Integer Scalar Arithmetic	17,089,871	19,843,306	24,856	22,002	3,495	3,374
Register Fetches	4,061,253	4,061,253	3,033	2,455	339	335
Register Stores	1,343,126	1,343,126	2,776	2,754	381	354
Scalar Fetches	24,087,534	36,970,159	48,892	48,852	6,530	7,352
Scalar Stores	15,452,695	28,335,571	12,887	17,217	1,680	2,639
Scalar Flops	21,686,805	34,286,805	31,241	36,977	4,334	5,316
Integer Vector Arithmetic	120,008	0	26	0	1	0
Vector Flops	360,000	0	1,214	0	162	0
Register Transfers	14,249,592	21,836,314	19,199	20,264	2,585	2,931
Vector Fetches	389,750	0	546	0	16	0
Vector Stores	389,758	0	749	0	132	0

TABLE II
(PROGRAM 5)
INSTRUCTION COUNTS AND FREQUENCIES (SUMMARY INFORMATION)

	Counts		Frequencies (Long Sample Runs)		Frequencies (Sum of 10 Short Sample Runs)	
	Vector	Scalar	Vector	Scalar	Vector	Scalar
Total Jump Count	4,907,673	11,783,405	3,239	5,398	310	570
Address Computation	13,976,612	18,286,797	24,490	8,163	1,614	931
Integer Scalar Arithmetic	6,445,280	34,097,315	8,273	25,665	500	2,705
Register Fetches	4,591,297	4,584,429	10,050	5,960	689	705
Register Stores	219,420	219,420	517	560	24	49
Scalar Fetches	7,223,895	30,211,392	30,291	31,395	1,994	3,493
Scalar Stores	6,458,086	18,075,711	8,014	9,858	486	1,169
Scalar Flops	3,137,602	37,232,715	3,367	35,544	250	4,070
Integer Vector Arithmetic	73,917	0	662	0	49	0
Vector Flops	1,095,255	0	7,047	0	469	0
Register Transfers	15,692,090	72,719,503	20,739	48,437	1,345	5,116
Vector Fetches	1,120,883	0	1,068	0	81	0
Vector Stores	760,844	0	1,450	0	85	0

TABLE III
CHI-SQUARED VALUES FOR LONG SAMPLE RUNS

Program	1	2	3	4	5
Degrees of Freedom (Vector Mode)	51	39	40	55	55
Degrees of Freedom (Scalar Mode)	52	39	38	56	56
Chi-Squared (Vector Mode)	[illegible]62	4980	7945	784	259
Chi-Squared (Scalar Mode)	939	947	1249	909	2741

TABLE IV
CHI-SQUARED VALUES FOR SUMS OF SHORT SAMPLE RUNS

Program	1	2	3	4	5
Degrees of Freedom (Vector Mode)	43	37	32	48	41
Degrees of Freedom (Scalar Mode)	47	38	35	50	46
Chi-Squared (Vector Mode)	5252	91651	109190	10250	1665
Chi-Squared (Scalar Mode)	15945	29128	33230	13867	27573
Degrees of Freedom (Vector Mode)	48	38	33	52	49
Degrees of Freedom (Scalar Mode)	49	40	35	53	48
Chi-Squared (Vector Mode)	10949	187310	216080	20649	3603
Chi-Squared (Scalar Mode)	31429	40017	66588	27668	58192
Degrees of Freedom (Vector Mode)	48	41	33	53	50
Degrees of Freedom (Scalar Mode)	52	41	36	54	48
Chi-Squared (Vector Mode)	15720	315280	336460	30393	5063
Chi-Squared (Scalar Mode)	47197	54608	100680	41284	85058
Degrees of Freedom (Vector Mode)	49	41	36	53	50
Degrees of Freedom (Scalar Mode)	52	41	37	54	48
Chi-Squared (Vector Mode)	21040	429120	454260	40039	6690
Chi-Squared (Scalar Mode)	62262	68728	132790	54810	110500

TABLE IV
CHI-SQUARED VALUES FOR SUMS OF SHORT SAMPLE RUNS

Program	1	2	3	4	5
Degrees of Freedom (Vector Mode)	49	41	36	53	50
Degrees of Freedom (Scalar Mode)	52	41	37	54	49
Chi-Squared (Vector Mode)	26251	514690	555030	49428	8351
Chi-Squared (Scalar Mode)	77499	99174	166270	68936	136920
Degrees of Freedom (Vector Mode)	50	41	36	53	51
Degrees of Freedom (Scalar Mode)	52	41	37	54	50
Chi-Squared (Vector Mode)	31361	598480	657010	59630	10034
Chi-Squared (Scalar Mode)	92730	12[illegible]40	198300	82113	168740
Degrees of Freedom (Vector Mode)	50	41	36	53	51
Degrees of Freedom (Scalar Mode)	52	41	37	55	51
Chi-Squared (Vector Mode)	36450	[illegible]91900	765350	70058	11663
Chi-Squared (Scalar Mode)	107970	157040	232570	95879	191900
Degrees of Freedom (Vector Mode)	51	41	36	53	51
Degrees of Freedom (Scalar Mode)	52	41	37	55	51
Chi-Squared (Vector Mode)	41581	771490	876640	80338	13224
Chi-Squared (Scalar Mode)	122730	187480	[illegible]390	109670	223350

TABLE IV
CHI-SQUARED VALUES FOR SUMS OF SHORT SAMPLE RUNS

Program	1	2	3	4	5
Degrees of Freedom (Vector Mode)	51	41	36	53	51
Degrees of Freedom (Scalar Mode)	52	41	37	55	52
Chi-Squared (Vector Mode)	46775	869910	974270	90811	14634
Chi-Squared (Scalar Mode)	138690	216080	297140	123750	249030
Degrees of Freedom (Vector Mode)	51	41	36	53	51
Degrees of Freedom (Scalar Mode)	52	41	37	55	52
Chi-Squared (Vector Mode)	52434	961550	1082000	100610	16684
Chi-Squared (Scalar Mode)	154340	243860	330820	137440	274250

TABLE V
PERCENTAGE POINTS OF THE CHI-SQUARED DISTRIBUTION

ν d.f.	99.5%	97.5%	5%	2.5%	1%	0.5%
1	392704×10^{-10}	982069×10^{-9}	3.84146	5.02389	6.63490	7.87944
2	0.0100251	0.0506356	5.99147	7.37776	9.21034	10.5966
3	0.0717212	0.215795	7.81473	9.34840	11.3449	12.8381
4	0.206990	0.484419	9.48773	11.1433	13.2767	14.8602
5	0.411740	0.831211	11.0705	12.8325	15.0863	16.7496
6	0.675727	1.237347	12.5916	14.4494	16.8119	18.5476
7	0.989265	1.68987	14.0671	16.0128	18.4753	20.2777
8	1.344419	2.17973	15.5073	17.5346	20.0902	21.9550
9	1.734926	2.70039	16.9190	19.0228	21.6660	23.5893
10	2.15585	3.24697	18.3070	20.4831	23.2093	25.1882
11	2.60321	3.81575	19.6751	21.9200	24.7250	26.7569
12	3.07382	4.40379	21.0261	23.3367	26.2170	28.2995
13	3.56503	5.00874	22.3621	24.7356	27.6883	29.8194
14	4.07468	5.62872	23.6848	26.1190	29.1413	31.3193
15	4.60094	6.26214	24.9958	27.4884	30.5779	32.8013
16	5.14224	6.90766	26.2962	28.8454	31.9999	34.2672
17	5.69724	7.56418	27.5871	30.1910	33.4087	35.7185
18	6.26481	8.23075	28.8693	31.5264	34.8053	37.1564
19	6.84398	8.90655	30.1435	32.8523	36.1908	38.5822
20	7.43386	9.59083	31.4104	34.1696	37.5662	39.9968
21	8.03366	10.28293	32.6705	35.4789	38.9321	41.4010
22	8.64272	10.9823	33.9244	36.7807	40.2894	42.7956
23	9.26042	11.6885	35.1725	38.0757	41.6384	44.1813
24	9.88623	12.4011	36.4151	39.3641	42.9798	45.5585
25	10.5197	13.1197	37.6525	40.6465	44.3141	46.9278
26	11.1603	13.8439	38.8852	41.9232	45.6417	48.2899
27	11.8076	14.5733	40.1133	43.1944	46.9630	49.6449
28	12.4613	15.3079	41.3372	44.4607	48.2782	50.9933
29	13.1211	16.0471	42.5569	45.7222	49.5879	52.3356
30	13.7867	16.7908	43.7729	46.9792	50.8922	53.6720
40	20.7065	24.4331	55.7585	59.3417	63.6907	66.7659
50	27.9907	32.3574	67.5048	71.4202	76.1539	79.4900
60	35.5346	40.4817	79.0819	83.2976	88.3794	91.9517
70	43.2752	48.7576	90.5312	95.0231	100.425	104.215
80	51.1720	57.1532	101.879	106.629	112.329	116.321
90	59.1963	65.6466	113.145	118.136	124.116	128.299
100	67.3276	74.2219	124.342	129.561	135.807	140.169

This table is reproduced with the permission of Professor E. S. Pearson from *Biometrika*, vol. 32, pp. 188-189.

APPENDIX A
CHARACTERISTICS OF TEST CODES

Floating-Point Operations (in Millions), Percentage Vectorization, and Average Vector Length

	MFLOP Count	Percentage Vectorization	Average Vector Length
Program 1	68.88	99.9	7
Program 2	147.54	56.2	64
Program 3	148.82	61.5	64
Program 4	40.04	31.5	35
Program 5	41.08	92.6	31