Edgar H. Sibley
Panel Editor

Reflecting current data on the use of programming language constructs in systems programming, a synthetic benchmark is constructed based on the distribution appearing in the data. The benchmark executes 100 Ada statements that are balanced in terms of the distribution of statement types, data types, and data locality. Pascal and C versions of the benchmark are discussed.

DHRYSTONE: A SYNTHETIC SYSTEMS PROGRAMMING BENCHMARK

REINHOLD P. WEICKER

In recent years, there has been growing interest in the interaction between programming languages and computer architecture (cf. [25]). As a high-level language host, a computer architecture should execute efficiently those features of a programming language that are most frequently used in actual programs. This ability is often measured by a program known as a "benchmark."

In this paper, the word benchmark refers to a single program that reflects the frequency of source language constructs in real programs and, consequently, in its compiled form, the frequency of the corresponding machine language constructs. It is unavoidable that such a benchmark will measure not only machine architecture but also the ability of the compiler to generate efficient code. There is a broader meaning of the word benchmark, denoting a collection of programs that also make use of the I/O system and the Operating System in general. However, since we are chiefly interested in the interaction between programming languages and computer architecture, this area is not covered here.

A typical example of a single-program benchmark is the "Whetstone" benchmark [7], which in its original form was developed in ALGOL 60. Whetstone reflects mostly numerical computing, using a substantial amount of floating-point arithmetic; it is now used chiefly in a FORTRAN version. However, since the data for the distribution of the different statement types in this program were collected in 1970, the benchmark cannot be expected to reflect the features of more modern programming languages (e.g., record and pointer data types). Also, recent publications on the interaction between programming languages and architecture have examined more subtle aspects of program behavior (e.g., the locality of data references—local versus global) that were not explicitly considered in earlier studies.

In recent papers dealing with the performance aspects of different computer architectures, performance is usually measured using some collection of programs that happened to be available to the author (e.g., [22]). However, following the pioneering paper of Knuth [17], an increasing number of publications have been providing statistical data about the actual usage of programming language features. It therefore seemed appropriate to make another attempt at constructing a synthetic benchmark program based on these recent statistics, particularly in the area of systems programming. In two steps, this paper summarizes and compares available data on the actual use of programming languages, and presents an easy-to-implement synthetic benchmark program based on these data. The program is called "Dhrystone," an analogy to the Whetstone benchmark program. Dhrystone's intended ease of implementation, however, has consequences (e.g., cache influences) that

must be taken into account if the program is to be used to compare different computer architectures or different compilers.

As this paper concentrates on systems programming, the benchmark presented is not claimed as representative for either numeric programming or business application programming. Different classes of applications typically use different language features: Numeric-scientific programs frequently use floating-point arithmetic and often operate on arrays; business application programs are mostly dominated by I/O activities; and systems programs often use enumeration, record, and pointer data types. The programming languages applied reflect this usage (e.g., data types for single and double precision in FORTRAN; high-level I/O operations in COBOL; and pointer data types in Ada, Pascal, or C). Even when the same language is used, numeric and nonnumeric programs in some respects exhibit quite different properties (cf. [2]). In fact, Clark and Levy's measurements [5] show that application differences lead to remarkably different frequency distributions even at the machine instruction level. Therefore, if a computer system is used for different types of applications, it is better to use several different benchmarks written in different languages.

SUMMARY OF "REAL" PROGRAM STATISTICS

Several authors have published the results of data collections on the use of various programming language features; these collections have been both static and dynamic. Although, from a performance point of view, dynamic measurements of programs are generally more interesting than static measurements, they are also more difficult to collect and impose far more overhead. Consequently, there are fewer of them. In fact, there are no publications known to this author that report on dynamic data collected in an industrial, as opposed to research, environment. The languages used in developing Dhrystone cover a broad range, and the classifications used in interpreting the results vary widely. In spite of these difficulties, we have endeavored to provide a comprehensive overview of the results. A brief characterization of the 16 different data collections used is given in Figure 1.

FIGURE 1. Characteristics of the Different Data Collections (in order of publication date)

- **Knuth's FORTRAN collection** [17]

The oldest publication in this area, Knuth's FORTRAN collection contains statistics on FORTRAN programs taken from the computing centers of Stanford University (average length: 600 lines) and Lockheed Corporation (average length: 570 lines). It contains static data for about 65 programs and dynamic data for about 25 programs.

- **XPL Data** [1]

XPL is a language with some similarity to PL/1. The sample in question contains 19 programs, mostly systems programs, including compilers, that were written by students at the University of Toronto. It contains static data only on the 19 programs (average length: 1736 statements).

- **Zelkowitz's PL/1 Data** [29]

At the University of Maryland, an augmented PL/1 compiler, the PLUM compiler, developed at the University of Maryland, was used to collect dynamic data on 1294 program runs of 168 different programs. The average program length in the sample was 48.2 statements per program run. It seems likely that the longer "production" runs of programs had the data collection feature deactivated more often than the shorter programs written by programming students.

- **Elshoff's PL/1 Data** [11, 12]

Elshoff's sample [12] contains 34 PL/1 programs written in the data processing department of General Motors that installation personnel viewed as some of the better programs. The paper compares these programs with earlier programs (analyzed in [11]) written in the same department before the introduction of structured programming. Therefore, we assume that the programs included in this current sample represent better than average commercial PL/1 programs. The data are static.

- **Tanenbaum's SAL Programs** [26]

Three-hundred procedures, taken from various systems programs written in the course of a project at Vrije University in Amsterdam, were analyzed statically as well as dynamically. The programs were written in SAL, a typeless, goto-less language intended for systems programming. Emphasis was placed on good program structuring, that is, short procedures (average length of a procedure: 18.2 statements).

- **Amsterdam ALGOL 68 Programs** [14]

Fifty-three ALGOL 68 programs (average length: 153 lines) were collected at the Mathematisch Centrum Amsterdam. They are characterized by the author as "normal run-of-the-mill programs," and there seem to be more numerical programs in this sample than in the other samples covered here. In addition to static data, the paper contains numbers gained from a static evaluation of the innermost loops. Although these data are not dynamic, they do seem to closely approximate the dynamic case.

- **Pascal Compiler Statistics** [24]

Static and dynamic data were collected on several Pascal compilers. For purposes of our calculations, an average of the static data was used. Dynamic data were collected only for Pascal stack machine instructions, not for source statements.

- **Berkeley "Project for Architectural Measurement"** [21]

In this project, static and dynamic data were collected for C and Pascal and reported in unpublished papers by Earl T. Cohen and Shafi Goldwasser. The C data are the overall average for six C programs used as utilities in the UNIX

Frequency of Statements

The frequencies of occurrence of the different types of high-level language statements are given in Tables I and II.

Static Statistics. In Table I, which shows statistical results on the *static* distribution of statements, "—" indicates that the category does not apply (e.g., there is no Case statement in PL/1 or FORTRAN), and "?" means that the publication in question does not contain the information.

In the table, assignments are counted independently of the number of operands or operators. (A more detailed breakdown is given in Table III.) For subprogram calls, numbers are given (where available) for both calls to user-defined subprograms and calls to standard procedures and functions. Some collections do not have the latter since they implement, rather than use, standard subprograms. The "Return" statement is used to a significant degree only in XPL and SAL. Other languages, for example, Pascal, use an assignment for function returns and have no explicit Return statement.

In the Elshoff data [12], it is not possible to separate DO loops controlled by a loop variable from DO statements simply bracketing groups of statements. For Knuth's FORTRAN collection [17], the "Other" category includes "Continue" and "Stop," whereas in the Ada study [10], most statements in the "Other" group come from tasking ("Select," "Accept," . . . , together representing 9.4 of the total 11.6 percent). In the iMAX 432 data [28], the "Other" group includes code statements encapsulated in Ada procedures and calls to such procedures.

Dynamic Statistics. Some known results from the far fewer dynamic statistics are presented in Table II.

Further Breakdown for Assignment Statements. Table III can be considered a refinement of the "assignment statement" entry in Tables I and II. Not all sources contain numbers for all table entries. (The results shown here overlap in part with the general statistics on the data types of operands given in Table VI.)

Operator Use. The statistics collected on the use of

system, including the well-known "spell" (spelling correction) and "nroff" (text formatting) programs. The Pascal data are taken from four Pascal programs (average length: 3758 lines), a simple compiler, a pretty printer, a file comparison program, and a CAD tool reported to be representative for scientific computations.

- **University of Wisconsin Pascal Data** [6]

These data represent the largest collection for Pascal known to this author (264 programs whose average length is 455 lines). They are evaluated using various criteria but unfortunately are static data only.

- **Manchester Pascal Data** [2]

Static data were collected for five numeric-scientific programs and six other programs, all written in Pascal. In terms of formulating Dhrystone, the percentages computed from the sums of the "other" programs were used (average length: 1196 statements). The Manchester collection is heavily dominated by a compiler (accounting for 67 percent of all statements).

- **Mesa statistics** [19]

The Mesa statistics were presented at the Symposium on Architectural Support for Programming Languages and Operating Systems [25]. They are dynamic data collected at the Xerox Palo Alto Research Center from two programs, the Mesa compiler (39,000 lines of code) and a VLSI Check program (500 lines of code).

- **C Statistics of Bell Labs** [9]

These C statistics were also presented at the same symposium. They were taken dynamically from the C compiler compiling itself.

- **Pascal Statistics at the University of Colorado** [4]

The data are static and come from two sources, the University of Colorado (61 programs, average length: 508 lines) and Tektronix, Inc. (28 programs, average length: 2285 lines). Most of the programs were compilers, assemblers, editors, or other kinds of text processors; there were only three numerical programs.

- **De Prycker's Pascal and ALGOL Statistics** [8]

The side effect of a research effort on a language-independent measurement system that collects data by instrumenting the source program, the data were collected statically as well as dynamically for six programs—four in ALGOL 60 and two in Pascal. In our presentation, only the results for the Pascal programs (compiler-related tasks) are included. It should be noted that the database for these measurements is quite small.

- **Ada Application Study** [10]

Static data were collected in an "Ada Capability Study" at General Dynamics, where a major subsection of the code for a message switching facility (7500 lines) was reprogrammed in Ada.

- **iMAX 432 Data** [28]

Static data were collected from iMAX 432, the operating system for Intel's "Micromainframe" iAPX 432 (10,081 statements, 518 procedures, and 142 packages in 131 compilation units). The system is written in Ada, and the programming style reflects the "object orientation" of the iAPX 432. In Ada terms, this leads to a heavy use of packages as a means of data abstraction, and of access types that reference objects.

TABLE I. Static Statements

Source	Knuth 71 [17]	Alex. & Wortman 75 [1]	Elshoff 77 [12]	Tanenbaum 78 [26]	Grune 79 [14]	Shimasaki et al. 80 [24]	Cook & Lee 82 [6]	Brookes et al. 82 [2]	De Prycker 82 [8]	Zeigler & Weicker 82 [28]	Dobbs 83 [10]
Language	FORTRAN	XPL	PL/1	SAL	ALGOL 68	Pascal	Pascal	Pascal	Pascal	Ada	Ada
Assignment	51.0	54.0	45.7	46.5	49.7	33.8	44.0	42.0	49.3	37.1	33.7
Call	12.0	16.9	15.4	24.9	31.0	40.3	31.8	34.3	29.4	26.8	23.8
user procedures	5.0	?	11.1	24.6	27.7	36.7	16.6	17.9	9.0	26.8	23.8
standard procedures (e.g., I/O)	7.0	?	4.3	0.3	3.3	3.6	15.2	16.4	20.4	–	–
Return	4.0	4.4	0.1	4.2	–	–	–	–	–	6.9	8.2
If	10.0	16.8	21.1	17.2	8.8	18.0	14.8	14.3	9.2	9.8	10.6
with else	–	10.4	7.7	?	?	?	7.4	7.4	?	?	?
without else	10.0	6.4	13.4	?	?	?	7.4	6.9	?	?	?
Loop with Condition	–	2.2		2.1	3.2	2.2	3.3	2.2	4.7	1.3	
while	–	2.2		1.6	3.2	1.2	2.6	1.5	3.7	.09	6.4
repeat	–	–	?	0.5	–	1.0	0.7	0.7	1.0	?	
Loop with "for"	9.0	3.4		3.4	5.1	0.9	2.8	2.1	6.7	0.9	
With	–	–	–	–	–	3.7	2.1	3.8	?	?	?
Case	–	0.8	–	0.3	1.7	0.7	0.9	0.8	0.0	0.4	1.5
Exit Loop	–	–	–	1.4	0.0	–	–	–	–	1.4	2.5
Goto	9.0	1.4	3.8	–	0.0	0.3	0.3	0.5	0.3	0.0	1.6
Other	5.0	–	–	–	–	–	0.1	–	–	15.4	11.6

NOTE: In Tables I, II, and VI, all figures have been rendered with fractional parts (e.g., 51.0 instead of 51) for presentational uniformity. This should not be seen as implying a higher degree of accuracy than was found in the original source.

different operators are given in Table IV. Usually these operators occur in assignments, but they are also found in comparisons and other locations where expressions are possible.

Number of Parameters. Some of the data collections give the number of parameters in procedure calls (see Table V). In the table, the figures from [6] and [28] indicate the number of parameters in procedure *declarations*, not in procedure *calls*. Therefore, the frequency in actual calls (measured statically or dynamically) may be different.

Operand Types

Another breakdown can be made according to the data type of the operands used. In this case, it is more difficult than in the previous tables to compare the results of the different collections, since for nonscalar data types, several ways of counting are possible:

- Only the final data type on the access path is counted (e.g., in Pascal notation, Pointer ↑. CharArray [IntegerIndex] is counted as a character operand);
- only the first data type is counted (e.g., Pointer, in the above example);
- all data types are counted (Pointer, Record, Array, Character, and Integer).

The data collections listed in Table VI all seem to count the first data type on the access path only. However, this is often not stated explicitly, and misunderstandings concerning the rules for data type counting are possible. Nonetheless, Table VI tries to summarize some of the results.

There is one data collection [8] that provides only a breakdown between accesses to scalar data as opposed to accesses to array elements, the relationship being about 1:1. The collection is therefore not included in the table.

Operand Locality

In Table VII, operands are classified according to their class, that is, as local variable, global variable, parameter, etc. In most of the papers, constants are not included in the distribution of locality, although it would be justified to include them there. For those papers where constants were included, the percentages were recomputed without constants for the sake of uniformity. The numbers given for the frequency of constants are the following:

Static	Tanenbaum 78	40.0%
Percentages	Patterson 80 (C)	26.9%
	Cook and Lee 82	34.1%
	Zeigler and Weicker 83	31.6%
Dynamic	Tanenbaum 78	32.8%
Percentages	Patterson 80 (C)	23.7%

It is obvious from Table VII that the programming language exerts a great influence. With a language that supports a module concept (abstract data types, sepa-

TABLE II. Dynamic Statements

Source	Knuth 71 [17]	Zelkowitz 76 [29]	Tanenbaum 78 [26]	Grune 79 [14][1]	Patterson 80 [21]	De Prycker 82 [8]
Language	FORTRAN	PL/1	SAL	ALGOL 68	Pascal	Pascal
Assignment	67.0	33.3	41.9	64.4	44.8	64.9
Call	4.0	32.8	12.4	26.7	14.6	18.7
user procedures	3.0	6.9	12.4	25.0	?	1.6
standard procedures (e.g., I/O)	1.0	25.9	0.0	1.7	?	17.1
Return	3.0	6.0	2.6	–	–	–
If	11.0	12.1	36.0	7.7	29.4	13.1
with else	–	?	?	?	?	?
without else	11.0	?	?	?	?	?
Loop with Condition	–	3.5	2.4	[see footnote]	4.8	0.8
while	–	3.5	2.3			0.5
repeat	–	–	0.1			0.3
Loop with "for"	3.0	9.1	2.1			2.3
With	–	–	–	–	5.2	?
Case	–	0.2	1.2	1.2	1.3	0.0
Exit Loop	–	–	1.6	–	–	–
Goto	9.0	3.0	–	0.0	0.0	0.0
Other	7.0	–	–	–	–	–

[1] In the case of [14], the numbers were not obtained by actual measurements but by a static count of the innermost loops, based on the assumption that these loops would be executed most frequently. Because of the nature of this method, the "for" and "while" statements are missing from the data.

TABLE III. Assignments

Source	Elshoff 77 [12][1]	Tanenbaum 78 [26]		Grune 79 [14]		Cook & Lee 82 [6]	Carter 82 [4]
Static/Dynamic	St.	St.	Dy.	St.	Dy.	St.	St.
Language	PL/1	SAL		ALGOL 68		Pascal	Pascal
Left-hand side							
Variable				71.0	50.0	55.8	
Array Element				15.0	43.0	44.2	
Record Component				4.0	2.0		
Other				10.0	5.0		
Right-hand side							
Constant		25.8	21.0	45.0	32.0	25.3	44.3
Variable	77.6	13.6	12.0			7.2	29.7
Array Element		5.2	5.1	5.0	9.0	48.9	
Function Result		4.9	2.0	?	?		
Record Component		30.5[2]	25.2[2]	5.0	9.0		
Other				45.0	50.0		
Expression	22.2	20.0	33.5			18.7	24.2
1 Operator	20.5	15.2	20.4			14.6	20.8
2 Operators	0.7	3.0	6.9			2.4	2.0
3 Operators	0.7	1.5	5.9			1.7	0.8
4 Operators or more	0.3	0.3	0.3				0.2

[1] Elshoff's data [12] give only the number of operators in expressions.
[2] The author is unable to account for the high numbers for "Other" in [26].

TABLE IV. Operators

Source	Knuth 71 [17][1]	Alexander & Wortman 75 [1]	Elshoff 77 [12]	Tanenbaum 78 [26][2]		Grune 79 [14]		Cook & Lee 82 [6]
Static/Dynamic	St.	St.	St.	St.	Dy.	St.	Dy.	St.
Language	FORTRAN	XPL	PL/1	SAL		ALGOL 68		Pascal
Arithmetic	(100%!)	44.5	25.5	(100%!)		57.1	77.2	42.1
+	38.7	23.0	17.5	50.0	57.4	15.7	24.7	21.2
−	22.2	18.9	4.9	28.3	25.5	19.2	19.0	10.8
*	26.6	1.5	2.2	14.6	13.2	14.1	22.1	6.2
/ (Division)	10.2	1.1	0.8	–	–	7.0	8.4	1.8
div (Int. Div.)	–	?	?	7.0	3.8	–	–	0.9
**(Expon.)	2.4	?	?	–	–	1.0	2.7	–
mod	–	–	–	–	–	0.1	0.2	1.2
Comparison		27.7	62.6	(100%!)		36.9	18.1	39.3
=		12.7	41.3	48.3	50.6	27.7	11.9	19.6
/=		3.6	11.6	22.1	18.6	2.5	2.5	10.0
>		6.0	4.6	11.8	10.2	2.7	2.0	3.6
<		2.9	3.4	9.5	9.0	1.9	0.8	2.8
>=		1.4	0.7	4.5	8.4	0.8	0.5	1.2
<=		1.1	0.6	3.8	3.3	1.4	0.4	2.1
Logic		8.4	5.2	?	?	3.7	2.2	13.7
AND		4.4	1.8	?	?	1.3	0.8	4.7
OR		2.6	3.0	?	?	1.1	0.7	3.2
NOT		1.4	0.6	?	?	1.2	0.6	5.8
Other		19.4	6.5[3]	?	?	2.4	2.5	4.8[3]

[1] In [17], numbers are given for arithmetic operators only.
[2] [26] gives only the frequencies *within* each of the two respective groups "Arithmetic" and "Comparison."
[3] The large number for "Other" in [1] and [12] comes from the use of a concatenation operator, whereas in [6] the "Other" category represents the *in* operator of Pascal.

rate compilation), programmers tend to use more local variables and parameters, whereas in a language like Pascal, with no separate compilation facilities, there are many declarations on the global level.

In addition, a closer look at Table VII shows that the relation "Local:Global" is heavily influenced by the method of data collection; that is, dynamic collections tend to have a higher percentage of local variables than static collections. This should be taken into account when interpreting static statistical data.

THE SYNTHETIC BENCHMARK PROGRAM

The Dhrystone benchmark was based on the frequency distribution data shown in Tables I–VII, this, despite the fact that the statistical data often provided quite divergent results. In developing Dhrystone, we attempted not to get a "superaverage" by simply averaging over the different results but rather to ensure that

TABLE V. Distribution of the Number of Parameters in Calls

Source	Tanenbaum 78 [26]		Cook & Lee 82 [6]	De Prycker 82 [8]	Zeigler & Weicker 83 [28]
Static/Dynamic	St.	Dy.	St.	Dy.	St.
Language	SAL		Pascal	Pascal	Ada
0 Parameters	41.0	21.2	47.1		
1 Parameter	19.0	27.6	31.4		
2 Parameters	15.0	23.3	14.1		
3 Parameters	9.3	10.8	4.7		
4 Parameters	7.3	8.8	2.0		
5 Parameters	5.3	6.6	0.8		
6 Parameters and more	2.9	1.8			
Average Number of Parameters	1.5	2.0	0.9	2.1	1.3

TABLE VI. Operand Types

Source	Tanenbaum 78 [26][1]		Patterson 80 [21]	Patterson 80 [21][2]		Cook & Lee 82 [6][3]	Dobbs 83 [10]	Zeigler & Weicker 83 [28][1]
Static/Dynamic	St.	Dy.	Dy.	St.	Dy.	St.	St.	St.
Language	SAL		Pascal	C		Pascal	Ada	Ada
Simple Variables	64.5	64.0	65.0	76.2	83.8	56.8	38.5	63.2
Integer			29.0	52.4	51.9		11.9	
Character			10.0	21.0	26.9		?	
Range			9.0	–	–		?	
Enumeration			7.0	–	–		23.8	
Boolean			3.0	–	–		1.4	
Real			0.0	1.1	3.4		–	
Other			–	1.7	1.6		1.4	
Pointer	?	?	18.0	?	?	15.0	11.9	11.8
Array	16.8	14.0	11.0	0.0 [?]	?	10.3	11.9	6.0
String	–	–	–	2.6	0.0	–	7.0	0.0
Record/Structure	12.9	16.9	7.0	19.2	14.8	12.7	16.8	7.0
Union	–	–	–	2.2	1.4	–	–	–
Set	–	–	1.0	–	–	?	–	–
File	–	–	?	?	?	4.1	–	–
Other	5.8	5.0	–	–	–	1.1	14.0	12.0

[1] In [26] and [28], files do not appear since their statistics deal with operating systems software that implements files rather than uses them. In [26], the "Other" category denotes bit fields.
[2] In [21], the first data type on the access path is counted as in the other data collections. However, for arrays, the final data type (type of the array element) seems to be counted, since the types array or array element never appear. Also, pointers do not appear, even though C programs almost always use them; probably, pointers have been subsumed under "Integer."
[3] In [6], there is one operand type table with entries indicating alternatively the data type (e.g., pointer) and the locality (e.g., parameter). It is not clear from the article which category is chosen for a given operand (e.g., a parameter or a pointer type).
[4] In [10], the number of declarations, not the number of accesses, is counted. This probably gives aggregate types (arrays, records) a higher relative weight. The "Other" group consists of tasks (7.7 percent) and private types (6.3 percent) that are mostly record types.
[5] The data given here from [28] provide numbers for the first data type on the access path, if there is an access path leading to the final operand. In addition, there are 12.0 percent occurrences of an aggregate data type (record or array) as a final operand (e.g., in assignments of whole records in one statement).

the benchmark reflected good programming practice (e.g., that the number of procedure calls was not too low). Another objective was ensuring that the data types and operations used were representative of the area of systems programming rather than numerical programming or pure applications data processing (e.g., COBOL programming). This means that the data collections based on systems programs had a higher weight.

The full text of the Ada version of the Dhrystone benchmark is given in the Appendix.

Detailed Distributions for the "Dhrystone" Benchmark

In the synthetic benchmark program, 100 statements are dynamically executed between the comment lines "start timer" and "stop timer."

Statement Types. The distribution of statements is given in Tables VIII–X. Twenty-two of the 53 assignments (or 41.5 percent of the total) have a variable of a constrained (sub-)type as their destination. In general, discriminant checks will be necessary in these cases, although the compiler may recognize cases where such checks are unnecessary.

The average number of parameters in procedure or function calls is 1.8 (not counting the function values as implicit parameters).

Operators. The distribution of operators is given in Table XI.

Operand Types. In Table XII, which shows the distribution of operand types, the operand type is counted once per operand reference. When there is an access path leading to the final operand, only the final data type on the access path is counted. For example, a reference to an element of an integer array is counted as an integer reference, not as an array reference.

In Dhrystone, there are 16 references to components of a record; 9 of them go to a component in a discriminant part. In this case, Ada requires a discriminant check, but for some of these references, the compiler may recognize during optimization that discriminant checks need not be generated.

Operand Locality. There is no static nesting of blocks or procedures; therefore, all variables are either global or local (Table XIII).

Other Remarks. There may be cases where a highly optimizing compiler may recognize unnecessary statements and may not generate code for them.

No explicit effort has been made to account for the effects of a cache or to balance the use of long or short displacements for code or data. This is discussed fur-

TABLE VII. Locality

Source	Patterson 80 [21]	Patterson 80 [21]		Cook & Lee 82 [6][2]	McDaniel 82 [19][3]	Ditzel & McLellan 82 [9][4]	Carter 82 [4][5]	De Prycker 82 [8][6]	Zeigler & Weicker 83 [28][7]
Static/Dynamic	St.	St.	Dy.	St.	Dy.	Dy.	St.	Dy.	St.
Language	Pascal	C_1		Pascal	Mesa	C_1	Pascal	Pascal	Ada
Local Variables	28.0	59.8	74.9	37.9	<87.6	57.0	41.0	28.0	73.5
Global Variables	42.0	24.1	14.2	45.5	12.5	3.0	52.0	73.0	12.0
Same Package	–	–	–	–	?	–	–	–	10.3
Other Package	–	–	–	–	?	–	–	–	1.7
Intermediate Variable or Parameter	13.0	–	–	1.7	?	–	7.0	0.0	0.3
Parameter	16.0	16.1	10.9	14.6	?	19.0		?	14.2
Value	5.0	16.1	10.9	?	?	19.0	57.0	?	–
Reference	11.0	–	–	?	?	–	43.0	?	–
Function Result	?	?	?	2.2	?	?	?	?	?
Other	–	–	–	–	–	21.0	–	–	–

[1] In C, there are no intermediate-level variables and no reference parameters.
[2] In [6], the number of function results appears in a different table (i.e., their table for data types) and therefore appears here *in addition* to the other entries, making the column add up to more than 100 percent.
[3] In [19], the numbers for local variables include value parameters as well as *local indirect*, which may represent reference parameters or data accessed through local pointers.
[4] In [9], the category "Other" includes 15 percent *indirect* and 6 percent *argument pushes*.
[5] In [4], the percentages given reflect only the relationships among the variables on the one hand and among the parameters on the other; the numerical relation between variables and parameters is not given.
[6] In [8], parameters are not mentioned.
[7] In [28], the access distribution within the "Parameter" group is not available. However, there are numbers available for parameter *declarations*: 85.5 percent in, 1.6 percent in out, and 12.8 percent out.

ther in a subsequent section, "Cache Aspects."

There were many cases where a decision had to be made as to how things should be counted; for example, what exactly "execution of a loop statement" means or how many data accesses it involves. In order to keep this paper reasonably short, not all the details of these decisions are discussed here. A separate paper covering these details [27] is available from the author.

PROGRAMMING LANGUAGE ASPECTS

Although the benchmark program was written in Ada, it was designed in a way that should make it possible to develop versions for several different programming languages. However, it is difficult to make versions of a program in languages that are sufficiently unrelated and still claim that the versions are the "same" program with respect not only to the result it delivers, but also with respect to execution time. Translation into a language like FORTRAN would be difficult, since FORTRAN has no notion of a pointer type. Attempts to simulate pointer types with the language features of FORTRAN (say, with indexes) would, in fact, change the benchmark into a different program.

Pascal Version

Since the benchmark uses in its statements only the "Pascal subset" of Ada, a translation into Pascal is relatively straightforward. Procedures Main and Proc_0 are merged into the main program of the Pascal version. Within this main program, all global types and variables are defined, and all other procedures (Proc_1 – Proc_8, Func_1 – Func_3) are declared, statically in parallel, within the main program. Since the local variables of Proc_0 now become global variables, the distribution between local and global access changes.

	Ada version	Pascal version
Locals	48.5%	33.3%
Globals	7.9%	23.5%

However, this change was expected; it reflects only the different programming styles in a language with a module concept (Ada), as opposed to a language without it (Pascal). We therefore feel justified calling it the same program in both the Ada and Pascal versions.

Other minor changes that have to be made for the Pascal version are the following:

- "In out" and "out" parameters become reference parameters, whereas "in" parameters become value parameters, with the exception of the two string parameters in function Func_2, which are naturally declared in Pascal as reference parameters.
- The "rename" statement in procedure Proc_3 becomes a "with" statement; accordingly, the part "Next_Record" of the identifiers that follow has to be dropped.
- The "loop" statement in procedure Proc_2 becomes a "repeat . . until" statement.
- The "return" statements become assignments to the

TABLE VIII. Assignment Statements in "Dhrystone"

	Percentage	
$V1 := V2$ (incl. $V1 := F(..)$	10	
$V :=$ Constant	12	
Assignment, with array element	7	
Assignment, with record component	6	
	35	35
$X := Y + \mid - \mid$ and $\mid$ or Z	5	
$X := Y + \mid - \mid$ "=" Constant	6	
$X := X + \mid -1$	3	
$X := Y * \mid / Z$	2	
$X :=$ Expression, two operators	1	
$X :=$ Expression, three operators	1	
	18	18
		53

TABLE IX. Control Statements in "Dhrystone"

	Percentage	
if .... then	14	
with "else"	7	
without "else"	7	
executed	3	
not executed	4	
for I in 1 .. N loop ...	6	counted every time
while ... loop ...	4	the loop condition
loop ... exit when $A = B$ ..	1	is evaluated
case ... is	1	
return	5	
rename	1	
	32	

TABLE X. Call Statements in "Dhrystone"

	Percentage	
$P(...)$		
procedure call		10
same package	5	
other package	5	
$X := F(...)$		
function call		5
same package	2	
other package	3	
		15

TABLE XI. Distribution of Operators in "Dhrystone"

	Number	Percentage
Arithmetic	27	52.9
+	16	31.4
-	7	13.7
*	3	3
/ (int div)	2.0	1
Comparison	39.2	20
=	9	17.6
/=	4	7.8
>	1	2.0
<	3	5.9
>=	1	2.0
<=	2	3.9
Logic	4	7.8
AND	1	2.0
OR	1	2.0
NOT	2	3.9
Sum	99.9	51

TABLE XII. Distribution of Operand Types in "Dhrystone"

	Number	Percentage
Integer	131	54.4
Character	47	19.5
Enumeration	30	12.4
Boolean	11	4.6
Pointer (Access Type)	12	5.0
String_30	6	2.5
Array	2	0.8
Record	2	0.8
	241	100.0

TABLE XIII. Distribution of Locality of Operands in "Dhrystone"

	Number	Percentage
Local variables	117	48.5
Global variables	19	7.9
same package	18	7.5
other package	1	0.4
Parameters	45	18.7
in	27	11.2
in out	12	5.0
out	6	2.5
Function results	5	2.1
Constants	55	22.8
	241	100.0

function name, and the statement distribution changes accordingly. Since the function name on the left-hand side of these assignments appears as an additional local variable, the distributions of data types and locality of references change slightly; the numbers given in Tables VIII–X for the Ada version would therefore be slightly different for the Pascal version.

C Version

To translate the benchmark into the "C" language, the enumeration type is mapped into integers by a suitable "#define" declaration. (A new extension to C [23] has enumeration types similar to Pascal and Ada.) Inout and out parameters, and also the string parameters of function Func_2, are mapped into parameters of type pointer, with appropriate dereferencing inside the procedures.

C, as documented in [16], has no assignments or comparison for aggregate variables (arrays, structures). A new extension of C [23] does have assignments for structures, but no comparisons, and still no assignments for strings. Therefore, the string comparison in function Func_2 and the string assignment in procedure Proc_0 have to be replaced by a call to a suitable subroutine. The same holds true for the record assignment in procedure Proc_1, unless the C version is based on the extension mentioned above.

The C language allows the programmer to declare variables as register variables. Therefore, there could conceivably be several versions of a C Dhrystone benchmark:

- a version without register variables;
- a version that declares every local variable of a scalar type to be a register variable, as a novice programmer might;
- a version where the programmer optimizes carefully, trading off the benefit of register variables in terms of access time against the additional overhead in procedure call and return.

This latter version would vary for different machines and compilers, since the trade-off must be made carefully.

In C, it is not possible to define range constraints in either type or variable declarations, as it is with the "range" or "subrange" concepts of Pascal and Ada. Therefore, there is nothing like an "assignment with implicit check" for these cases as there is in Ada. Also, because array indexing is defined in C as equivalent to a pointer operation, C compilers cannot check array accesses against out-of-bounds conditions. Although the "union" concept of C corresponds to the "discriminant record" of Pascal and Ada, the C language definition is somewhat imprecise with regard to the question of whether discriminant checks are required, allowed, or forbidden in C. (Traditionally, C compilers don't check.) Therefore, if C results are to be compared with Pascal or Ada results, the Pascal or Ada runs should have the corresponding constraint checks turned off.

As can be seen from the discussion above, it is more difficult to translate the Dhrystone benchmark into an "equivalent" C program than it is into a Pascal version.

CACHE ASPECTS

The time necessary to execute a program depends heavily on the existence and usage of a cache, whether or not the code, data, or both happen to stay in the cache during execution of the program. (The array accesses in procedure Proc_8 are somewhat distributed in order to result in at least some cache misses.)

The most common method of execution time measurement is to include the benchmark program in a loop that is executed, say, 10,000 times and, for a very precise measurement, to subtract the execution times for 10,000 executions of an empty loop. However, this method tends to heavily overemphasize the influence of a cache. Since the Dhrystone benchmark is quite small, the code, and to a large degree also the data, will stay in the cache all the time if the cache size is above a certain minimal size. A more sophisticated program might try to balance this effect by duplicating code and data and by forcing the flow of execution to "wander around" the address space according to some statistics on working set behavior. However, for this simple benchmark program, no attempt was made to include such a mechanism.

USE OF THE BENCHMARK

Practical benchmarks are often either proprietary programs or collections of small programs that happened to be available to the user at the time. This is especially true of certain microprocessor benchmarks or benchmarks used in discussions of new architectural features. Often, a subset of the "EDN benchmarks" [13] is used. (The EDN benchmarks were originally written in assembly language at Carnegie-Mellon University for an evaluation of different computer architectures [3].) Also popular are the programs "Puzzle," "Sieve of Eratosthenes," and "Ackermann's function" (e.g., [15]). However, no studies have been conducted to determine how representative these programs are. As pointed out by Levy and Clark [18], the performance results on some machines varied up to 40 percent, depending on whether just three of the above programs were used or a fourth (Ackermann's function) was added to the sample.

The intention of this paper has been to present a better founded benchmark program for architecture or compiler discussions. The program has been used internally for comparisons of different microprocessors, for comparisons of micros with minicomputers, and for evaluation of experimental designs. In our experience, the results achieved with Dhrystone as a yardstick reflect fairly accurately the effectiveness of a particular hardware/compiler combination for systems programming applications.

Dhrystone can be easily ported and its run-time

measured on a new machine.[1] However, this intended ease of application makes mention of the following caveat important: Wherever benchmarks are used, there is an unfortunate tendency to look only at the final number, the execution time. This tendency is nicely paraphrased in the third of the "Not-So-Golden Rules of Benchmarking" [20]:

> *Conditions, cautions, relevant discussion and even actual code never make it to the bottom line when results are summarized.*

As discussed above, there are several areas where the details (language influence, compiler influence, measurement method, cache influence) have to be checked very carefully whenever a benchmark like this one is used for a comparison of different processors or different systems. (See [18] for more details.) There are inherent limitations to any single number (like a benchmark result) if it is used as the *only* criterion for the evaluation of processor architectures or compilers. However, if used judiciously, a benchmark like Dhrystone may still have value in the area of interfacing between programming language and computer architecture.

[1] For readers interested in measurement experiments, the program is available from the author in machine-readable form on floppy disk (Ada, Pascal, and C versions).

Acknowledgments. The author wants to thank Justin Rattner and Christian Ritscher for their critical reading and helpful suggestions.

APPENDIX: Program Text, Ada Version

```
--                                                                      --
--                        "DHRYSTONE" Benchmark Program                 --
--                        -----------------------------                 --
--                                                                      --
--      Version:  ADA/1                                                 --
--                                                                      --
--      Date:     04/15/84                                              --
--                                                                      --
--      Author:   Reinhold P. Weicker                                   --
--                                                                      --
------------------------------------------------------------------------
--                                                                      --
-- The following program contains statements of a high-level programming --
-- language (Ada) in a distribution considered representative:          --
--                                                                      --
--   assignments                 53%                                    --
--   control statements          32%                                    --
--   procedure, function calls   15%                                    --
--                                                                      --
-- 100 statements are dynamically executed. The program is balanced with --
-- respect to the three aspects:                                        --
--                                                                      --
--   - statement type                                                   --
--   - operand type (for simple data types)                             --
--   - operand access                                                   --
--        operand global, local, parameter, or constant.                --
--                                                                      --
-- The combination of these three aspects is balanced only approximately. --
--                                                                      --
-- The program does not compute anything meaningful, but it is syntactically --
-- and semantically correct. All variables have a value assigned to them --
-- before they are used as a source operand.                            --
------------------------------------------------------------------------
package Global_Def is
----------------------
```

```
-- Global type definitions

type Enumeration is (Ident_1, Ident_2, Indent_3, Ident_4, Ident_5);

subtype One_To_Thirty is integer range 1. .30;
subtype One_To_Fifty is integer range 1..50;
subtype Capital_Letter is character range 'A'. .'Z';

type String_30 is array (One_To_Thirty) of character;
  pragma Pack (String_30);

type Array_1_Dim_Integer is array (One_To_Fifty) of integer;
type Array_2_Dim_Integer is array (One_To_Fifty,
                                   One_To_Fifty) of integer;

type Record_Type (Discr: Enumeration := Ident_1);

type Record_Pointer is access Record_Type;

type Record_Type (Discr: Enumeration := Ident_1) is
    record
      Pointer_Comp:       Record_Pointer;
      case Discr is
      when Ident_1 =>       -- only this variant is used,
                            -- but in some cases discriminant
                            -- checks are necessary
        Enum_Comp:        Enumeration;
        Int_Comp:         One_To_Fifty;
        String_Comp:      String_30;
      when Ident_2 =>
        Enum_Comp_2:      Enumeration;
        String_Comp_2:    String_30;
      when others =>
        Char_Comp_1,
        Char_Comp_2:      character;
      end case;
    end record;
end Global_Def;

  with Global_Def;
  use Global_Def;
package Pack_1 is
----------

  procedure Proc_0;
  procedure Proc_1 (Pointer_Par_In:    in      Record_Pointer);
  procedure Proc_2 (Int_Par_In_Out:    in out  One_To_Fifty);
  procedure Proc_3 (Pointer_Par_Out:   out     Record_Pointer);

  Int_Glob:           integer;
end Pack_1;

with Global_Def;
use Global_Def;
package Pack_2 is
----------

  procedure Proc_6  (Enum_Par_In:          in      Enumeration;
```

```
                    Enum_Par_Out:          out     Enumeration);
procedure Proc_7    (Int_Par_In_1,
                     Int_Par_In_2:         in      One_To_Fifty;
                     Int_Par_Out:          out     One_To_Fifty;
procedure Proc_8    (Array_Par_In_Out_1:  in out  Array_1_Dim_Integer;
                     Array_Par_In_Out_2:  in out  Array_2_Dim_Integer;
                     Int_Par_In_1,
                     Int_Par_In_2:         in      integer);
function Func_1     (Char_Par_In_1,
                     Char_Par_In_2;        in      Capital_Letter)
                       return Enumeration;

function Func_2     (String_Par_In_1,
                     String_Par_In_2:      in      String_30)
                       return boolean;

end Pack_2;

with Global_Def, Pack_1;
use Global_Def;

procedure Main is
----------

begin
  Pack_1.Proc_0;  -- Proc_0 is actually the main program, but it is part
                  -- of a package, and a program within a package can
                  -- not be designated as the main program for execution.
                  -- Therefore Proc_0 is activated by a call from "Main".

end Main;

with Global_Def, Pack_2;
use Global_Def;

package body Pack_1 is
----------

  Bool_Glob:          boolean;
  Char_Glob_1,
  Char_Glob_2:        character;
  Array_Glob_1:       Array_1_Dim_Integer;
  Array_Glob_2:       Array_2_Dim_Integer;
  Pointer_Glob,
  Pointer_Glob_Next: Record_Pointer;

  procedure Proc_4;
  procedure Proc_5;

procedure Proc_0
is
  Int_Loc_1,
  Int_Loc_2,
  Int_Loc_3:     One_To_Fifty;
  Char_Loc:      character;
  Enum_Loc:      Enumeration;
  String_Loc_1,
  String_Loc_2: String_30;
```

```
begin
  -- Initializations
  Pack_1.Pointer_Glob_Next := new Record_Type;
  Pack_1.Pointer_Glob := new Record_Type
                         '(
                         Pointer_Comp => Pack_1.Pointer_Glob_Next,
                         Discr        => Ident_1,
                         Enum_Comp    => Ident_3,
                         Int_Comp     => 40,
                         String_Comp  => "DHRYSTONE PROGRAM, SOME STRING"
                        );
  String_Loc_1 := "DHRYSTONE PROGRAM, 1'ST STRING";
---------------
-- Start timer --
---------------
  Proc_5;
  Proc_4;
    -- Char_Glob_1 = 'A', Char_Glob_2 = 'B', Bool_Glob = false
  Int_Loc_1 := 2;
  Int_Loc_2 := 3;
  String_Loc_2 := "DHRYSTONE PROGRAM, 2'ND STRING";
  Enum_Loc := Ident_2;
  Bool_Glob := not Pack_2.Func_2 (String_Loc_1, String_Loc_2);
    -- Bool_Glob = true
  while Int_Loc_1 < Int_Loc_2 loop -- loop body executed once
    Int_Loc_3 := 5 * Int_Loc_1 - Int_Loc_2;
      -- Int_Loc_3 = 7
    Pack_2.Proc_7 (Int_Loc_1, Int_Loc_2, Int_Loc_3);
      -- Int_Loc_3 = 7
    Int_Loc_1 := Int_Loc_1 + 1;
  end loop;
    -- Int_Loc_1 = 3
  Pack_2.Proc_8 (Array_Glob_1, Array_Glob_2, Int_Loc_1, Int_Loc_3);
    -- Int_Glob = 5
  Proc_1 (Pointer_Glob);
  for Char_Index in 'A' . . Char_Glob_2 loop--loop body executed twice
    if Enum_Loc = Pack_2.Func_1 (Char_Index, 'C')
    then--not executed
      Pack_2.Proc_6 (Ident_1, Enum_Loc);
    end if;
  end loop;
    -- Enum_Loc = Ident_1
    -- Int_Loc_1 = 3, Int_Loc_2 = 3, Int_Loc_3 = 7
  Int_Loc_3 := Int_Loc_2 * Int_Loc_1;
  Int_Loc_2 := Int_Loc_3 / Int_Loc_1;
  Int_Loc_2 := 7 * (Int_Loc_3 - Int_Loc_2) - Int_Loc_1;
  Proc_2 (Int_Loc_1);
-----------------
-- Stop timer --
-----------------
end Proc_0;
procedure Proc_1 (Pointer_Par_In: in Record_Pointer)
is--executed once
  Next_Record: Record_Type
```

```
    renames Pointer_Par_In.Pointer_Comp.all;-- = Pointer_Glob_Next.all
begin
  Next_Record := Pointer_Glob.all;
  Pointer_Par_In.Int_Comp := 5;
  Next_Record.Int_Comp := Pointer_Par_In.Int_Comp;
  Next_Record.Pointer_Comp := Pointer_Par_In.Pointer_Comp;
  Proc_3 (Next_Record.Pointer_Comp);
    -- Next_Record.Pointer_Comp = Pointer_Glob.Pointer_Comp = Pointer_Glob_Next
  if Next_Record.Discr = Ident_1
  then  -- executed
    Next_Record.Int_Comp := 6;
    Pack_2.Proc_6 (Pointer_Par_In.Enum_Comp, Next_Record.Enum_Comp);
    Next_Record.Pointer_Comp := Pointer_Glob.Pointer_Comp;
    Pack_2.Proc_7 (Next_Record.Int_Comp, 10, Next_Record.Int_Comp);
  else  -- not executed
    Pointer_Par_In.all := Next_Record;
  end if;
end Proc_1;

procedure Proc_2 (Int_Par_In_Out: in out One_To_Fifty)
is  --  executed once
    --  Int_Par_In_Out = 3, becomes 7
  Int_Loc: One_To_Fifty;
  Enum_Loc: Enumeration;
begin
  Int_Loc := Int_Par_In_Out + 10;
  loop  -- executed once
    if Char_Glob_1 = 'A'
    then  -- executed
      Int_Loc := Int_Loc - 1;
      Int_Par_In_Out := Int_Loc - Int_Glob;
      Enum_Loc := Ident_1;
    end if;
  exit when Enum_Loc = Ident_1;  -- true
  end loop;
end Proc_2;

procedure Proc_3 (Pointer_Par_Out: out Record_Pointer)
is  -- executed once
    -- Pointer_Par_Out becomes Pointer_Glob
begin
  if Pointer_Glob /= null
  then  -- executed
    Pointer_Par_Out := Pointer_Glob.Pointer_Comp;
  else  -- not executed
    Int_Glob := 100;
  end if;
  Pack_2.Proc_7 (10, Int_Glob, Pointer_Glob.Int_Comp);
end Proc_3;

procedure Proc_4  -- without parameters
is  -- executed once
  Bool_Loc: boolean;
begin
  Bool_Loc := Char_Glob_1 = 'A';
  Bool_Loc := Bool_Loc or Bool_Glob;
  Char_Glob_2 := 'B';
end Proc_4;

procedure Proc_5 -- without parameters
is--executed once
```

```
begin
   Char_Glob_1 := 'A';
   Bool_Glob := false;
end Proc_5;

end Pack_1;

with Global_Def, Pack_1;
use Global_Def;

package body Pack_2 is
-----------------------

function Func_3 (Enum_Par_In: in Enumeration) return boolean;
         -- forward declaration

procedure Proc_6 (Enum_Par_In:    in  Enumeration;
                  Enum_Par_Out: out Enumeration)
is  -- executed once
    -- Enum_Par_In = Ident_3, Enum_Par_Out becomes Ident_2
begin
   Enum_Par_Out := Enum_Par_In;
   if not Func_3 (Enum_Par_In)
   then  -- not executed
     Enum_Par_Out := Ident_4;
   end if;
   case Enum_Par_In is
     when Ident_1 => Enum_Par_Out := Ident_1;
     when Ident_2 => if Pack_1.Int_Glob > 100
                       then Enum_Par_Out := Ident_1;
                       else Enum_Par_Out := Ident_4;
                       end if;
     when Ident_3 => Enum_Par_Out := Ident_2;  -- executed
     when Ident_4 => null;
     when Ident_5 => Enum_Par_Out := Ident_3;
   end case;
end Proc_6;
procedure Proc_7 (Int_Par_In_1,
                  Int_Par_In_2: in  One_To_Fifty;
                  Int_Par_Out:  out One_To_Fifty)
is  -- executed three times
    -- first call:    Int_Par_In_1 = 2, Int_Par_In_2 = 3,
    --                Int_Par_Out becomes 7
    -- second call: Int_Par_In_1 = 6, Int_Par_In_2 = 10,
    --                Int_Par_Out becomes 18
    -- third call:    Int_Par_In_1 = 10, Int_Par_In_2 = 5,
    --                Int_Par_Out becomes 17
  Int_Loc: One_To_Fifty;
begin
   Int_Loc := Int_Par_In_1 + 2;
   Int_Par_Out := Int_Par_In_2 + Int_Loc;
end Proc_7;
procedure Proc_8 (Array_Par_In_Out_1: in out Array_1_Dim_Integer;
                  Array_Par_In_Out_2: in out Array_2_Dim_Integer;
                  Int_Par_In_1,
                  Int_Par_In_2:          in      integer)
is  -- executed once
    -- Int_Par_In_1 = 3
```

```
    -- Int_Par_In_2 = 7
  Int_Loc: One_To_Fifty;
begin
  Int_Loc := Int_Par_In_1 + 5;
  Array_Par_In_Out_1 (Int_Loc) := Int_Par_In_2;
  Array_Par_In_Out_1 (Int_Loc+1) :=
                              Array_Par_In_Out_1 (Int_Loc);
  Array_Par_In_Out_1 (Int_Loc+30) := Int_Loc;
  for Int_Index in Int_Loc . . Int_Loc+1 loop -- loop body executed twice
    Array_Par_In_Out_2 (Int_Loc, Int_Index) := Int_Loc;
  end loop;
  Array_Par_In_Out_2 (Int_Loc, Int_Loc-1) :=
                              Array_Par_In_Out_2 (Int_Loc, Int_Loc-1) + 1;
  Array_Par_In_Out_2 (Int_Loc+20, Int_Loc) :=
                              Array_Par_In_Out_1 (Int_Loc);
  Pack_1.Int_Glob := 5;
end Proc_8;

function Func_1 (Char_Par_In_1,
                 Char_Par_In_2: in Capital_Letter)
                                                  return Enumeration
is  -- executed three times, returns Ident_1 each time
    -- first call:    Char_Par_In_1 = 'H', Char_Par_In_2 = 'R'
    -- second call: Char_Par_In_1 = 'A', Char_Par_In_2 = 'C'
    -- third call:    Char_Par_In_1 = 'B', Char_Par_In_2 = 'C'
  Char_Loc_1, Char_Loc_2: Capital_Letter;
begin
  Char_Loc_1 := Char_Par_In_1;
  Char_Loc_2 := Char_Loc_1;
  if Char_Loc_2 /= Char_Par_In_2
  then-- executed
    return Ident_1;
  else-- not executed
    return Ident_2;
  end if;
  end Func_1;

function Func_2 (String_Par_In_1,
                 String_Par_In_2: in String_30) return boolean
is  -- executed once, returns false
    -- String_Par_In_1 = "DHRYSTONE, 1'ST STRING"
    -- String_Par_In_2 = "DHRYSTONE, 2'ND STRING"
    Int_Loc:   One_To_Thirty;
    Char_Loc: Capital_Letter;
begin
  Int_Loc := 2;
  while Int_Loc <= 2 loop  -- loop body executed once
    if Func_1 (String_Par_In_1(Int_Loc),
               String_Par_In_2(Int_Loc+1)) = Ident_1
    then-- executed
      Char_Loc := 'A';
      Int_Loc := Int_Loc + 1;
    end if;
  end loop;
  if Char_Loc >= 'W' and Char_Loc < 'Z'
  then-- not executed
    Int_Loc := 7;
  end if;
  if Char_Loc = 'X'
  then-- not executed
```

```
    return true;
  else  -- executed
    if String_Par_In_1 > String_Par_In_2
    then  -- not executed
      Int_Loc := Int_Loc + 7;
      return true;
    else  -- executed
      return false;
    end if;
  end if;
end Func_2;

function Func_3 (Enum_Par_In: in Enumeration) return boolean
is  -- executed once, returns true
    -- Enum_Par_In = Ident_3
  Enum_Loc: Enumeration;
begin
  Enum_Loc := Enum_Par_In;
  if Enum_Loc = Ident_3
  then  -- executed
    return true;
  end if;
end Func_3;

end Pack_2;
```


REFERENCES

1. Alexander, W.G., and Wortman, D.B. Static and dynamic characteristics of XPL programs. *Computer 8*, 11 (Nov. 1975), 41–46.
2. Brookes, G.R., Wilson, I.R., and Addyman, A.M. A static analysis of Pascal program structures. *Softw. Pract. Exper. 12*, 11 (Nov. 1982), 959–963.
3. Burr, W.E., and Smith, W.R. Comparing architectures. *Datamation 23*, 2 (Feb. 1977), 48–52.
4. Carter, L.R. *An Analysis of Pascal Programs*. UMI Research Press, Ann Arbor, Mich., 1982.
5. Clark, D.W., and Levy, H.M. Measurement and analysis of instruction use in the VAX 11/780. In 9th Annual Symposium on Computer Architecture. *SIGARCH News. 10*, 3 (Apr. 1982), 9–17.
6. Cook, R.P., and Lee, I. A contextual analysis of Pascal programs. *Soft. Pract. Exper. 12*, 2 (Feb. 1982), 195–203.
7. Curnow, H.J., and Wichman, B.A. A synthetic benchmark. *Comput. J. 19*, 1 (Feb. 1976), 43–49.
8. De Prycker, M. On the development of a measurement system for high level language program statistics. *IEEE Trans. Comput. C-31*, 9 (Sept. 1982), 883–891.
9. Ditzel, D.R., and McLellan, H.R. Register allocation for free: The C machine stack cache. In Symposium on Architectural Support for Programming Languages and Operating Systems. *SIGPLAN Not. 17*, 4 (Apr. 1982), 48–56.
10. Dobbs, P. Ada experience on the Ada capability study. *Ada Letters 2*, 6 (May 1983), 59–62.
11. Elshoff, J.L. An analysis of some commercial PL/1 programs. *IEEE Trans. Softw. Eng. SE-2*, 2 (June 1976), 113–120.
12. Elshoff, J.L. The influence of structured programming on PL/1 program profiles. *IEEE Trans. Softw. Eng. SE-3*, 5 (Sept. 1977), 364–368.
13. Grappel, R.D., and Hemenway, J.E. A tale of four μPs: Benchmarks quantify performance. *EDN* (Apr. 1, 1984), 179–365.
14. Grune, D. Some statistics on ALGOL 68 programs. *SIGPLAN Not. 14*, 7 (July 1979), 38–46.
15. Hansen, P.M., Linton, M.A., Mayo, R.N., Murphy, M., and Patterson, D.A. A performance evaluation of the Intel iAPX432. *Comput. Archit. News 10*, 4 (June 1982), 17–26.
16. Kernighan, B.W., and Ritchie, D.M. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, N.J., 1978.
17. Knuth, D.E. An empirical study of FORTRAN programs. *Softw. Pract. Exper. 1* (1971), 105–133.
18. Levy, H.M., and Clark, D.W. On the use of benchmarks for measuring system performance. *Comput. Archit. News 10*, 6 (Dec. 1982), 5–8.
19. McDaniel, G. An analysis of a Mesa instruction set using dynamic instruction frequencies. In Symposium on Architectural Support for Programming Languages and Operating Systems. *SIGPLAN Not. 17*, 4 (Apr. 1982), 167–176.
20. Patstone, W. 16-bit-uP benchmarks—An update with explanations. *EDN* (Sept. 16, 1981), 169–171.
21. Patterson, D.A. (private communication).
22. Patterson, D.A., and Piepho, R.S. RISC assessment: A high-level language experiment. In 9th Annual Symposium on Computer Architecture. *SIGARCH Newsl. 10*, 3 (Apr. 1982), 3–8.
23. Ritchie, D. Recent changes to C. Appendix to The C programming language—Reference manual, Berkeley UNIX Distribution, Univ. of California, Berkeley, 1978.
24. Shimasaki, M., Fukaya, S., Ikeda, K., and Kiyono, T. An analysis of Pascal programs in compiler writing. *Softw. Pract. Exper. 10*, 2 (Feb. 1980), 149–157.
25. Symposium on Architectural Support for Programming Languages and Operating Systems. *SIGPLAN Not. 17*, 4 (Apr. 1982).
26. Tanenbaum, A.S. Implications of structured programming for machine architecture. *Commun. ACM 21*, 3 (Mar. 1978), 237–246.
27. Weicker, R.P. Details of the "Dhrystone" benchmark program. Manuscript, 1984.
28. Zeigler, S.F., and Weicker, R.P. Ada language statistics for the iMAX 432 operating system. *Ada Letters 2*, 6 (May 1983), 63–67.
29. Zelkowitz, M.V. Automatic program analysis and evaluation. In *2nd International Conference on Software Engineering* (San Francisco, Calif., Oct. 13–15). ACM, New York, 1976, pp. 158–163.

CR Categories and Subject Descriptors: C.4 [**Performance of Systems**]: *measurement techniques*; D.3.3 [**Programming Languages**]: Language Constructs—*control structures, data types and structures*; K.6.2 [**Management of Computing and Information Systems**]: Installation Management–*benchmarks*

General Terms: Languages, Performance

Additional Key Words and Phrases: statistics, benchmark, Whetstone, Dhrystone

Received 8/83; accepted 2/84

Author's Present Address: Reinhold P. Weicker, Siemens Corporate Research and Technology, Inc., 105 College Road East, Princeton, NJ 08540.