Parallel Computing 17 (1991) 1153–1172
North-Holland

A detailed look at some popular benchmarks *

Reinhold P. Weicker

Siemens Nixdorf Informationssysteme, STM OS 32, München, Germany

Received March 1991
Revised April 1991

Abstract

Weicker, R.P., A detailed look at some popular benchmarks, Parallel Computing 17 (1991) 1153–1172.

'Fair benchmarking' can be less of an oxymoron – if the people using benchmark results know what tasks the benchmarks really perform and what they measure. This paper characterizes in detail the most commonly used industry standard benchmark programs (Whetstone, Dhrystone, Linpack) and names a number of pitfalls that users have to be aware of. Other benchmarks are mentioned, and the relative merits of small vs. large benchmarks are discussed. The often neglected non-CPU influences on benchmark results are discussed, and 'consumer advice' is offered.

Keywords. Benchmark computer performance; MIPS; Whetstone; Linpack; Dhrystone; SPEC.

1. Introduction: MIPS and MFLOPS

Traditionally, computer speed is given in MIPS or MFLOPS. However, these numbers often generate more confusion than enlightment:

- MIPS in the literal sense of the word means Million Instructions Per Second. But what program, written in which language, is the basis for the computation? What compiler has been used?
- In the commercial world, where the instruction set of the IBM mainframes (370 series, Siemens Nixdorf 7500 series, and compatibles) is dominating, MIPS numbers are mostly based on a specific COBOL program resulting in a 'composite mix' of instructions.
- Microprocessor manufacturers sometimes give 'Peak MIPS' numbers that are considerably higher than MIPS numbers based on any real program. For this peak MIPS, only the fastest available instruction is considered. In the extreme case, peak MIPS just count how many NOOP instructions can be executed per second.
- In the world of supercomputers, the traditional unit is MFLOPS. Again, this often means 'Peak MFLOPS ', the highest rate of floating-point operations per second, obtainable only in ideal cases, including maximum parallelism achieved with a vector-processing unit. 'Linpack MFLOPS ' (discussed below) are a totally different unit.
- The advent of RISC architectures has additionally contributed to a loss of significance of the word 'MIPS' in its literal meaning. For a given high-level language program, RISC machines execute more instructions, but these instructions are simpler, easier to decode, and permit a

* An earlier version of this manuscript was published unter the title 'An Overview of Common Benchmarks' in IEEE Computer 23 (12) (1990) 65-75.

more efficient pipelining. These factors typically contribute to an overall gain in speed. However, this gain would be exaggerated if MIPS in the literal sense of the word (also called 'native MIPS') are used for performance measurements. For the end-user, only the resulting execution time is important, not the number of machine-level instructions executed for a given high-level language program.

Because of these problems, the word MIPS has often been redefined, implicitly or explicitly ('VAX MIPS'): People often denote by MIPS just a performance factor of a given machine relative to the performance of a VAX 11/780: If a machine runs some program or set of programs x times faster than a VAX 11/780, it is called an x MIPS machine. This is based on a 'computer folklore' statement saying that a VAX 11/780 performs, for typical programs, one million of instructions per second. Although this folklore is not true, [1] it is widespread. When VAX MIPS's are quoted, one should know what programs are the basis of the comparison, and what compilers are used for the VAX 11/780. Older Berkeley UNIX compilers produced up to 30% slower code than VMS compilers, thereby inflating the other machine's MIPS rating.

In short, one cannot help but agree with Omri Serlin's [9] statement: "There are no accepted industry standards for computing the value of MIPS".

2. Benchmarks

Any attempt to give MIPS numbers some useful meaning (e.g. VAX MIPS) boils down to running a representative program or set of programs. Therefore, it is better to drop the notion of MIPS and just measure the speed of these 'benchmark' programs.

It has been said several times that the best benchmark is one's own application. However, this is often unrealistic: It is not always possible to run the application on each machine in question; the program may have been tailored to run optimally on an older machine; OME manufacturers must chose a microprocessor for a whole range of applications; journals want to characterize machine speed independent of a particular application program. Therefore, the next best benchmark

- is written in a high-level language, making it portable across different machines,
- is representative for some kind of programming style (e.g. system programming, numerical programming, commercial programming),
- can be measured easily,
- has a wide distribution.

Obviously, some of these requirements contradict each other: The more representative the program is for some real programs, the more complicated it will be. It becomes more difficult to measure, and chances are that results are available for only a few machines.

This explains the popularity of some benchmark programs that are not complete application programs but still have reason to claim a certain representativity for some area. This article covers mainly the most common 'stone-age' benchmarks (CPU/memory/compiler benchmarks only), in particular the Whetstone, Dhrystone, and Linpack benchmarks. Results obtained with these benchmarks are the ones most often cited in manufacturer's publications and in the trade press; they are better than meaningless MIPS numbers, but readers should know their properties, what they measure and what they don't measure.

[1] Some time ago, I ran the Dhrystone benchmark program on VAX 11/780's with different compilers. With Berkeley UNIX (4.2) Pascal, the benchmark was translated into 483 instructions, executed in 700 μsec, yielding 0.69 (native) MIPS. With DEC VMS Pascal (V, 2.4), 226 instructions were executed in 543 μsec, yielding 0.42 (native) MIPS. Interestingly, the version with the lower MIPS rating executed the program faster.

Table A
Statement distribution in %

Statement	Dhrystone	Whetstone	Linpack/saxpy
Assignment of a variable	20.4	14.4	–
Assignment of a constant	11.7	8.2	–
Assignment of an expression (1 operator)	17.5	1.4	–
Assignment of an expression (2 operators)	1.0	24.3	48.5
Assignment of an expression (3 operators)	1.0	1.6	–
Assignment of an expression (> 3 operators)	–	6.8	–
One-sided if statement, 'then' part executed	2.9	0.5	–
One-sided if statement, 'then' part not executed	3.9	0.1	2.2
Two-sided if statement, 'then' part executed	4.9	4.0	–
Two-sided if statement, 'else' part executed	1.9	4.0	–
For statement (evaluation)	6.8	17.3	49.3
Goto statement	–	0.5	–
While/Repeat statement (evaluation)	4.9	–	–
Switch statement	1.0	–	–
Break statement	1.0	–	–
Return statement (with expression)	4.9	–	–
Call statement (user procedure)	9.7	11.9	–
Call statement (user function)	4.9	–	
Call statement (system procedure)	1.0	–	
Call statement (system function)	1.0	4.7	–
	100	100	100

Table B
Operator distribution in %

Operator	Dhrystone		Whetstone		Linpack/saxpy	
+(int/char)	21.0		11.9		14.1	
−(int)	5.0		6.0		–	
*(int)	2.5		6.0		–	
/(int)	0.8		–		–	
Integer arithmetic	29.3	29.3	23.9	23.9	14.1	14.1
+(float/double)	–		14.9		14.1	
−(float/double)	–		2.1		–	
*(float/double)	–		9.3		14.1	
/(float/double)	–		4.6		–	
Floating-point arithmetic	–	–	30.9	30.9	28.2	28.2
<, < = (incl. loop control)	10.1		10.7		14.5	
Other relational operators	11.7		2.8		0.6	
Relational	21.8	21.8	13.5	13.5	15.1	15.1
Logical	3.3	3.3	–	–	0.2	0.2
Indexing (1-dimensional)	5.9		24.5		42.3	
Indexing (2-dimensional)	3.4		–		–	
Indexing	9.3	9.3	24.5	24.5	42.3	42.3
Record selection	7.6		–		–	
Record selection via pointer	15.1		–		–	
Record selection	22.7	22.7	–	–		–
Address operator (C)	5.0		3.6		–	
Indirection operator (C)	8.4		3.6		–	
C-specific operators	13.4	13.4	7.2	7.2	–	–
		100		100		100

Table C
Operand data type distribution in %

Operand data type	Dhrystone	Whetstone	Linpack/saxpy
integer	57.0	55.7	67.2
char	19.6	–	–
float/double	–	44.3	32.8
enumeration	10.9	–	–
boolean	4.2	–	–
array	0.8	–	–
string	2.3	–	–
pointer	5.3	–	–
	100	100	100

Table D
Operand locality distribution in %

Operand locality	Dhrystone	Whetstone	Linpack/saxpy
local	48.7	0.4	49.5
global	8.3	56.3	–
parameter (value)	10.6	18.6	17.0
parameter (reference)	6.8	1.9	24.6
function result	2.3	1.3	–
constant	23.4	21.6	8.8
	100	100	100

Whetstone and Dhrystone are synthetic benchmarks: They have been written solely for benchmarking purposes, they perform no useful computation. Linpack has been distilled out of a real program, a program with a 'normal' purpose that is now used as a benchmark.

Tables A–D give detailed information [2] about the high-level language features used by these benchmarks. By comparing these numbers with his of her own programs, the reader can appreciate how meaningful the results produced with a particular benchmark are for his or her applications. The tables contain comparable information for all three benchmarks, therefore enabling the reader to see immediately the differences and the similarities of the benchmarks.

All percentages are dynamic percentages, i.e. percentages obtained by profiling or, for the language feature distribution, by adding appropriate counters on the source level and executing the program with counters. Note that for all programs, even for the programs that are usually used in the FORTRAN version, the language-feature related statistics refer to the C versions of the benchmarks; this was the version for which the modification was performed. However, since most features are similar for other language versions, numbers for other languages should not be too different. The profiling data have been obtained from the FORTRAN version (Whetstone, Linpack) or the C version (Dhrystone).

3. Whetstone

The Whetstone benchmark was the first program in the literature that was explicitly designed for benchmarking purposes, its authors are H.J. Curnow and B.A. Wichmann from

[2] Due to rounding, all percentages can add up to a number slightly below or above 100.

the National Physical Laboratory in Great Britain. It was published in 1976, with ALGOL 60 as the publication language. Today it is almost exclusively used in its FORTRAN version, with either single precision or double precision for floating-point numbers.

The Whetstone benchmark owes its name to the Whetstone ALGOL compiler system. This system was used to collect statistics about the distribution of 'Whetstone instructions', instructions of the intermediate language used by this compiler, for a large number of numerical programs. A synthetic program was then designed, consisting of several modules where each module contains statements of some particular type (e.g. integer arithmetic, floating-point arithmetic, if statements, calls), ending with a statement printing the results. Weights are attached to the different modules (realized as loop bounds for loops around the individual modules' statements) such that the distribution of Whetstone instructions for the synthetic benchmark matches the distribution observed in the program sample. The weights have been chosen in a way that the program executes a multiple of one million of these Whetstone instructions; benchmark results are given accordingly as KWIPS (Kilo Whetstone Instructions Per Second) or MWIPS (Mega Whetstone Instructions Per Second). In this way, the familiar term 'instructions per second' was kept, but given a machine-independent meaning.

A problem with Whetstone is that there is only one officially controlled version, the Pascal version issued with the Pascal Evaluation Suite by the British Standard Institute – Quality Assurance (BSI-QAS). Versions in languages other than Pascal can be registered with BSA-QAS to ensure comparability.

Many Whetstone versions copied informally and used for benchmarking are versions with the print statements removed, apparently with the intention to achieve better timing accuracy. However, this is contrary to the authors' intentions since optimizing compilers may then eliminate significant parts of the program. If timing accuracy is a problem, the loop bounds should be increased such that the time spent in the extra statements becomes insignificant.

Users should know that since 1988, there is a revised (Pascal) version of the benchmark [10]. Changes have been made to module 6 and module 8 to adjust the weights and to preclude unintended optimization by compilers; the print statements have been replaced by statements checking the values of the variables used in the computation. According to Wichmann [10], performance figures for the two versions 'should be very similar'; however, differences up to 20% cannot be excluded. The FORTRAN version has not undergone a similar revision since with the separate compilation model of FORTRAN, the danger of unintended optimization is smaller – although it certainly exists if all parts are compiled in one unit. All Whetstone data in this article are based on the old version; the language feature statistics are almost identical for both versions.

3.1. Size, procedure profile and language feature distribution

The static length of the Whetstone benchmark (C version) as compiled by the VAX UNIX 4.3 bsd C compiler [3], is 2117 bytes (measurement loops only). However, because of the nature of the program, the length of the individual modules is more important. They are between 40 and 527 bytes long; all except one are less than 256 bytes long. The weights (upper loop bounds) of the individual modules are between 12 and 899.

Table 1 shows the distribution of execution time spent in the subprograms of Whetstone (VAX 11/785, bsd 4.3 FORTRAN, single precision).

The most important (and to many, surprising) result is that Whetstone spends more than half of its time not in the compiled user code but in library subroutines.

[3] With the UNIX 4.3 bsd language systems, it was easier to determine the code size for the C version. The numbers for the FORTRAN version should be similar.

Table 1
Procedure profile for Whetstone

Procedure	%	%	What is done there
Main program	18.9		
p3	14.4		FP arithmetic
p0	11.6		indexing
pa	1.9		FP arithmetic
User code	46.8	46.8	
Trigonometric functions	21.6		sin, cos, atan
Other math. functions	31.7		exp, log, sqrt
Library functions	53.3	53.3	
		100	

The distribution of language features is shown in Tables A–D. Some properties of Whetstone are probably typical for most numeric applications (e.g. high amount of loop statements), others are a property of Whetstone only (very few local variables).

3.2. Whetstone characteristics

The following enumeration lists some of the most important characteristics of Whetstone that one should keep in mind when Whetstone numbers are used for performance comparisons:

- Whetstone has a high percentage of floating-point data and floating-point operations. This was intended since the benchmark is meant to represent numeric programs.
- As shown in the preceding section, a high percentage of execution time is spent in mathematical library functions. This property is derived from the statistical data that formed the basis for Whetstone; however, it may not be representative for most of today's numerical application programs. Since the speed of these functions (realized as software subroutines or microcode) dominates Whetstone performance to a high degree, manufacturers can be tempted to manipulate the runtime library for Whetstone performance.
- As evident from *Table D*, Whetstone uses hardly any local variables. At the time when Whetstone was written, the issue of local vs. global variables was hardly discussed in software engineering, let alone in computer architecture. Because of this unusual lack of local variables, register windows (e.g. in the SPARC RISC architecture) or good register allocation algorithms for local variables (e.g. in the MIPS RISC compilers) don't make a difference in Whetstone execution times.
- Instead of local variables, Whetstone uses a handful of global data (several scalar variables and a four-element array of constant size) over and over again. Therefore, a compiler where the most heavily used global variables are allocated in registers (an optimization that is usually considered of secondary importance) will boost Whetstone performance.
- Due to its construction principle (9 small loops), Whetstone has an extremely high code locality. A near 100% hit rate can be expected even for fairly small instructions caches. For the same reason, a simple reordering of the source code can significantly alter the execution time in some cases. For example, it has been reported that for the MC68020 with its 256-byte instruction cache, re-ordering of the source code can boost performance up to 15%.

4. Linpack

As described by its author [2], Linpack didn't originate as a benchmark, it was first just a collection (a package, hence the name) of linear algebra subroutines often used in FORTRAN

programs. It was first published in 1976; the author is Jack Dongarra from the University of Tennessee (previously: Argonne National Laboratory). He has now distilled what was part of a 'real-life' program into a benchmark that is distributed in various versions [3]. Linpack results are collected and published by Jack Dongarra.

The program operates on a large matrix (2-dimensional array); however, the inner subroutines manipulate the matrix as a one-dimensional array, an optimization customary for sophisticated FORTRAN programming. The matrix size in the version distributed by the standard mail servers is 100×100 (within a two-dimensional array declared with bounds 200), but versions for larger arrays also exist.

The results are usually reported in terms of MFLOPS (Million Floating-Point Operations Per Second), the number of floating-point operations executed by the program can be derived from the array size. Note that this terminology means that the non-floating point operations are neglected or, stated otherwise, that their execution time is included in that of the floating-point operations. When floating-point operations become increasingly faster relative to integer operations, this terminology becomes more and more misleading.

For Linpack, it is important to know what version is measured with respect to the following attribute pairs:

- *Single / double*:
  FORTRAN Single Precision or Double Precision for the floating-point data.
- *Rolled / unrolled*:
  In the 'unrolled' version, loops are optimized at the source level by 'loop unrolling': The loop index (say, i) is incremented in steps of 4, and the loop body contains 4 groups of statements, for indexes i, $i+1$, $i+2$, $i+3$. This technique saves execution time for most machines and compilers; however, more sophisticated vector machines, where loop unrolling is done by the compiler generating code for vector hardware, usually execute the standard (rolled) version faster.
- Coded BLAS/FORTRAN BLAS:
  Linpack relies heavily on a sub-package of 'Basic Linear Algebra Subroutines' (BLAS). 'Coded BLAS' (as opposed to 'FORTRAN BLAS') means that these subroutines have been rewritten in assembly language. Linpack's author, Jack Dongarra, has discontinued to collect and publish results for 'Coded BLAS' versions and requires that only the FORTRAN version of these subroutines is used unchanged; however, some results of 'Coded BLAS' versions are still cited elsewhere. Computation of the execution time ratio between the coded BLAS and the FORTRAN BLAS versions for the same machine can reveal some insights about the FORTRAN compiler's code optimization quality: For some machines, the ratio is 1.2, for others, it can be as high as 2.

4.1. Size, procedure profile and language feature distribution

The Linpack data reported in this section are for the 'rolled' version, single precision, with FORTRAN BLAS; code sizes have been measured with VAX UNIX bsd 4.3 FORTRAN.

The static code length for all subprograms is 4537 bytes. For the individual subprograms, the length varies between 111 and 1789 bytes; the most heavily used subprogram, 'saxpy', has a length of 234 bytes. Data size is – in the standard version – dominated by an array of 100×100 real numbers. For 32-Bit machines, this means that with single precision, 40KBytes are used for data (double precision: 80 KBytes).

The procedure profile in *Table 2* shows the distribution of execution time in the various subroutines. The most important observation from *Table 2* is that more than 75% of Linpack's execution time is spent in a 15-line subroutine (called 'saxpy' in the single precision version, 'daxpy' in the double precision version). Dongarra [2] reports that on most machines, the percentage is even higher (90%).

Table 2
Procedure profile for Linpack

Procedure	%	What is done there
Main program	0.0	
matgen	13.8	
sgefa	6.2	
saxpy	77.1	$y[i] = y[i] + a*x[i]$
isamax	1.6	
miscellaneous	1.2	
User code	100.0	
Library functions	0	

Because of this extreme concentration of the execution time in the 'saxpy' subroutine, and because of the time-consuming instrumentation method for the measurements, language feature distribution has been measured only for the subroutine 'saxpy' (rolled version).

It can be seen from *Table A* that very few statement types (assignment with multiplication and addition, 'for' statement) make up the bulk of the subroutine and, therefore, of Linpack in total. The data are mostly reference parameters (the array values) or local variables (indexes), there are hardly any global variables.

4.2. Linpack characteristics

For performance characterizations by Linpack MFLOPS, one should know the following main characteristics:

- As one would expect for a numeric benchmark, Linpack has a high percentage of floating-point operations. However, only a few floating-point operations are actually used. For example, there are no floating-point divisions in the program. In striking contrast to Whetstone, no mathematical functions are used at all.
- The execution time is almost exclusively spent in one small function. This means that even a small instruction cache will show a very high hit rate.
- Contrary to the high locality for code, Linpack has a low locality for data. A larger size for the main matrix leads – depending on the cache size – to significantly more cache misses and therefore to a lower MFLOPS rate. Therefore, it is important to know in comparisons whether Linpack MFLOPS for different machines have been computed using the same array

Table 3
Dhrystone procedure profile

Procedure	%	%	What is done there
Main program	18.3		
User procedures	65.7		
User code	84.0	84.0	
strcpy	8.0		string copy (string constant)
strcmp	8.1		string comparison (string variables)
library functions	16.1	16.1	
		100	

dimensions. In addition, Linpack can also be highly sensitive to the cache configuration: A different array alignment (201 × 200 instead of 200 × 200 for the global array declaration) can lead to a different mapping of data to cache lines and therefore to a considerably different execution time. The program as distributed by the standard mail servers delivers MFLOPS numbers for two choices of leading dimension, 200 and 201; one can assume that manufacturers report the better number.

5. Dhrystone

As the name already indicates, Dhrystone was developed in a similar way as Whetstone, it is a synthetic benchmark published by the author in 1984. The original language of publication is Ada, although it uses only the 'Pascal subset' of Ada and was intended to be easily translatable to Pascal and C; presently it is mainly used in the C version.

Dhrystone is based on a literature survey on the distribution of source language features in non-numeric, system-type programming (operating systems, compilers, editors, etc.). It has been observed that in addition to the obvious difference in data types (integral types vs. floating-point types), numeric and system-type programs have other differences also: System programs contain less loops, simpler computational statements, more 'if' statements and procedure calls.

Dhrystone consists of 12 procedures, they are included in one measurement loop with 94 statements. During one loop ('one Dhrystone'), 101 (C Version: 103) statements are executed dynamically. The results are usually given in 'Dhrystones per second'. The program (current version: 2.1) has been mainly distributed through the UNIX network Usenet; it is also available from the author on floppy disk. Results for the Dhrystone benchmark have been collected and posted regularly to the UNIX network Usenet by Rick Richardson (latest list of results: April 29, 1990).

5.1. Size, procedure profile and language feature distribution

The static length of the Dhrystone measurement loop, as compiled by the VAX UNIX (bsd 4.3) C compiler, is 1039 bytes. *Table 3* shows the distribution of execution time spent in its subprograms.

The percentage of time spent in the string operations is highly language-dependent; it drops to 10% instead of 16%, if the Pascal (or Ada) version is used (measurement for Berkely Unix 4.3 Pascal). On the other hand, the number is higher for newer RISC machines with optimizing compilers, mainly because they spend much less time in procedure calls than the VAX.

Consistent with usage in system-type programming, arithmetic expressions are simpler than in the other benchmarks, there are more 'if' statements and less loops.

Dhrystone was the first benchmark to explicitly consider the locality of operands: Local variables and parameters are used more often than global variables. This is not only consistent with good software engineering practices, it is also important for modern CPU architectures (RISC architectures): On older machines with few registers, local variables and parameters are allocated in memory in the same way as global variables; on RISC machines, they typically reside in registers. The resulting difference in access time is one of the most important advantages of RISC architectures.

5.2. Dhrystone characteristics

For performance characterizations by Dhrystone, one should know the following main characteristics:

Table 4
SPEC benchmark programs

Acronym	Short characterization	Language	Main data types
gcc	GNU C compiler	C	integer
espresso	PLA simulator	C	integer
spice 2g6	Analog circuit simulation	Fortran	floating-point
doduc	Monte Carlo simulation	Fortran	floating-point
nasa7	Collection of several numerical 'kernels'	Fortran	floating-point
li	LISP interpreter	C	integer
eqntott	Switching function minimization, mostly sorting	C	integer
matrix300	Various matrix multiplication algorithms	Fortran	floating-point
fpppp	Maxwell equations	Fortran	floating-point
tomcatv	Mesh generation, highly vectorizable	Fortran	floating-point

Table 5
Different languages: Standard programs

	Performance ratio (larger is better, C = 1)			
	Bliss	C	Pascal	Ada
Search	1.24	1.0	0.70	
Sieve	0.63	1.0	0.80	
Puzzle	0.77	1.0	0.73	
Ackermann	1.20	1.0	0.80	
Dhrystone (1.1)		1.0	1.32	1.02

Table 6
Compiler optimization levels: Dhrystone

	Dhrystones/sec	
	V.1.1	V.2.1
No opt., no registers	30700	31000
No opt., registers	32600	32400
Optim. 'O', no registers	39700	36700
Optim. 'O', registers	39700	36700
Optimization 'O3'	43100	39400
Optimization 'O4'	46700	43200

Table 7
Benchmark sizes for some popular benchmarks

	Code bytes	Data bytes
Whetstone	~256	16
Dhrystone	1039	–
Linpack (saxpy) 100×100 version	234	40000
Sieve standard version	160	8192
Quicksort standard version	174	20000
Puzzle	1642	511
Ackermann	52	–

Table 8
VAX-relative performance, measured on the basis of different benchmarks (as of March 1991)

	Intel	Motorola	Intel	MIPS	IBM	Intel
Processor	80386/80387	68030/68881	80486	R3000/3010	RS6000	80860
MHz	25	33	33	33	25	33
Dhrystone	7.6	6.6	25.7	41.5	34.9	38.6
Whetstone	2.7	6.9	(8.7)	21.7	28.8	23.5
Linpack	3.4	4.4	(9.4)	35.7	77.9	32.1
Int. SPEC	6.0	5.2	18.2	27.1	20.2	16.7
Float. SPEC	n.a.	3.1	9.2	26.1	36.7	23.6
SPECmark	4.1	3.8	12.1	26.5	28.9	20.5

Note: These numbers may be outdated at the time when this article appears, due to improvements in compilers, variations in the cache size and cache design in the systems used for measurements. Readers should get the latest information from the manufacturers. Numbers in parentheses are scaled from numbers given for other clock frequencies and may not be accurate.

- As intended, Dhrystone contains no floating-point operations in its measurement loop.
- A considerable percentage of execution time is spent in string functions; this number should have been lower. In extreme cases (MIPS architecture and C compiler), this number goes up to 40%.
- Unlike Whetstone, Dhrystone contains hardly any loops within the main measurement loop. Therefore, for microprocessors with small instruction caches (below 1000 bytes), almost all instruction accesses are cache misses. But as soon as the cache size gets larger than the measurement loop, all instruction accesses are cache hits.
- Only a few global data are manipulated, and the data size cannot be scaled as in Linpack.
- No attempt has been made to thwart optimizing compilers. The goal was that the program should reflect typical programming style; it should be similarly optimizable as normal programs. An exception is the optimization of 'dead code removal': Since in version 1 the results of the computation were not printed or used otherwise, optimizing compilers were able to recognize too many statements as 'dead code' and to suppress code generation for these statements. In version 2, this has been corrected.

5.3. Ground rules for Dhrystone number comparisons

Because of the peculiarities of Dhrystone, users should make sure that the following 'ground rules' are observed when Dhrystone results are compared:

- The version should be 2.1; the earlier version 1.1 leaves too much room for distortion of results by dead code elimination.
- The two modules must be compiled separately, and procedure merging (inlining) is not allowed for user procedures. ANSI C, however, allows the inlining of library routines (relevant for the C version of Dhrystone: String routines).
- When processors are compared, the same programming language must be used on both processors. For equal quality compilers, Pascal and Ada numbers can be about 10% better because of the string semantics: In C, the length of a string is normally not known at compile time, and the compiler needs – at least for the string comparison statement in Dhrystone – to generate code that checks each byte for the string terminator byte (null byte). With Pascal and Ada the compiler can generate word instructions (usually inline code) for the string operations.
- For a meaningful comparison of C version results, one should know:
  (1) Are the string routines written in machine code?

(2) Are the string routines implemented as inline code?
(3) Does the compiler use the fact that in the 'strcpy' statement, the source operand has a fixed length? If is does (legal according to ANSI C), this statement can be compiled in the same way as a record assignment, which can result in considerable savings.
(4) Is a word alignment assumed for the string routines? This is acceptable for the 'strcpy' statement only, not for the 'strcmp' statement.

Language systems are allowed to optimize for cases (1)–(3) similarly as they can optimize programs in general. However, for processor comparisons, one should make certain that the compilers used apply the same amount of optimization; otherwise the optimization differences may overshadow the CPU speed differences. This usually requires an inspection of the generated machine code and of the C library routines.

6. Other benchmarks

In addition to the most-quoted benchmarks explained above, there are several other programs used as benchmarks, among them:

- Stanford Small Programs Benckmark Set
- EDN Benchmarks
- Sieve of Eratosthenes
- Livermore Fortran Kernels
- Perfect Club Benchmarks
- SPEC Benchmarks

These programs range from small, randomly chosen programs (Sieve) to elaborate benchmark suites (Livermore Fortran Kernels, SPEC Benchmarks). Those benchmarks that have become more popular in the area of numerical computing are covered elsewhere in this issue.

6.1. Livermore Fortran Kernels

The 'Livermore Fortran Kernels', also called the 'Lawrence Livermore Loops', consist of 24 'kernels', i.e. inner loops of numeric computations from different areas of the physical sciences. The author, F.H. McMahon (Lawrence Livermore National Laboratory, Livermore, CA), has collected them into a benchmark suite and has added statements for time measurement. The individual loops range from a few lines to about one page of source code. The program is self-measuring and computes MFLOPS rates for each kernel, for three different vector lengths.

As can be expected, these kernels contain a high amount of floating-point computations and a high percentage of array accesses. Several kernels contain vectorizable code; some contain code that is vectorizable if rewritten. For a detailed discussion of the Livermore Loops, see Feo [4]. McMahon characterizes the representativity of the Livermore Loops as follows: "The net Mflops rate of many Fortran programs and workloads will be in the sub-range between the equi-weighted harmonic and arithmetic means depending on the degree of code parallelism and optimization. The Mflop metric provides a quick measure of the average efficiency of a computer system since its peak computing rate is well known."

6.2. Stanford small programs benchmark set

At about the same time when the first RISC systems were developed at the Universities of Berkeley and Stanford, a set of small programs (one page of source code or less for each program) was collected at Stanford University (Computer Systems Laboratory) by John Hennessy and Peter Nye. They became popular mainly because they were the basis of the first

comparisons of RISC and CISC processors. They have now been packed into one C program containing

- 8 integer programs: Permutations, Towers of Hanoi, 8 Queens, Integer Matrix Multiplication, Puzzle, Quicksort, Bubble Sort, Tree Sort,
- 2 floating-point programs: Floating-point Matrix Multiplication, Fast Fourier Transformation.

The characteristics of the individual programs vary, most of them contain a high percentage of array accesses. There seems to be no 'official' publication of the source code, the only place where I have seen the C code in print is a manufacturer's performance report.

There is no standardized method of generating an overall 'figure of merit' from the individual execution times. In one version, a driver program assigns weights between 0.5 and 4.44 to the individual execution times. A probably better alternative, used by Sun and MIPS, is to compute the geometric mean of the individual programs' execution times.

6.3. EDN benchmarks

The program collection now known as the 'EDN benchmarks' was developed by a group at Carnegie-Mellon University for the project 'Military Computer Family', it was published by EDN in 1981. Originally, the programs were written in several assembly languages (LSI-11/23, 8086, 68000, Z8000); the intention was to measure the speed of microprocessors without also measuring the compiler's quality.

A subset of the original benchmarks is now often used in a C version:

- Benchmark E: String search
- Benckmark F: Bit test/set/reset
- Benchmark H: Linked list insertion
- Benchmark I: Quicksort
- Benchmark K: Bit matrix transformation.

It was this subset of the EDN benchmarks that has been used in Bud Funk's comparison of RISC and CISC processors [5]. There seems to be no standard C version of the EDN benchmarks; the programs are disseminated in an informal way only.

6.4. Sieve of Eratosthenes

One of the most popular programs for benchmarking in the world of small PC's is the 'Sieve of Eratosthenes', sometimes also called 'Primes'. It computes all prime numbers up to a given limit (usually 8192). The program has some unusual characteristics: 33% of the dynamically executed statements are assignments of a constant, only 5% are assignments with an expression at the right hand side. There ar no 'while' statements and no procedure calls, 50% of the statements are loop control evaluations. All operands are integer operands, 58% of them are local variables.

The program is mentioned here not because Sieve can be considered a good benchmark but because, as one author has put it, "Sieve performance of one compiler over another has probably sold more compilers for some companies than any other benchmark in history".

7. SPEC benchmarks

The SPEC (System Performance Evaluation Cooperative) effort grew out of a feeling of the benchmarking experts at various companies that most previously existing benchmarks (usually small programs) are inadequate:

- Small benchmarks can no longer be representative for real programs when it comes to testing the memory system, because with the growing size of cache memories and the introduction of on-chip caches for high-end microprocessors, the cache hit ratio comes close to 100% for these benchmarks.
- Once a small program gets too popular as a benchmark, compiler writers are inclined (or forced) to 'tweak' their compilers into optimizations that are particularly beneficial to this benchmark. An example are the string optimizations for Dhrystone.

SPEC has the goal to collect, standardize and distribute large-size application programs that can be used as benchmarks. This is a nontrivial task since realistic programs that have been used before in benchmarking (e.g. the UNIX utilities 'yacc' or 'nroff') often require a license and are therefore not freely distributable.

The founding members of SPEC were Apollo, Hewlett-Packard, MIPS, Sun; in the meantime, AT&T, Bull, CDC, Compaq, Data General, DEC, Fujitsu, IBM, Intel, Intergraph, Motorola, NCR, Prime, Siemens, Silicon Graphics, Solbourne, Stardent, and Unisys have joined as members. It can be expected that the SPEC benchmark suite will soon become more and more important.

In October 1989, SPEC has released a first set of 10 benchmark programs (*Table 4*).

Table 4 contains only a very rough characterization of the programs; a more detailed discussion of the present SPEC benchmark suite is given by Dixit [1]. Because a license has to be signed, and because of its size (150_000 lines of source code), the SPEC benchmark suite is distributed via magnetic tape only.

Results are given as the performance relative to a VAX 11/780 (using VMS compilers for the VAX). Results for several computers of SPEC member companies are contained in the regular SPEC Newsletter. A comprehensive number, the 'SPECmark' is defined as the geometric mean of the relative performance of the 10 programs. However, SPEC requires a reporting form where, in addition to the raw data, the relative performance is given for each benchmark program separately. In this way, users can select the subset of performance numbers where the programming language and/or the application area matches their applications most closely.

The SPEC benchmarking effort is covered in more detail in the article "SPECulations" by Kaivalya Dixit (this issue [1]).

8. Non-CPU influences in benchmark performance

In trade journals and advertisements, manufacturers usually credit good benchmark numbers only to the speed of the hardware system. In the case of microprocessors, this is even more reduced to the speed of the CPU. However, it has become obvious from the preceding discussion that there are other influencing factors as well:

- Programming language
- Compiler
- Runtime library functions
- Memory and cache size.

8.1. Programming language influence

Table 5 (numbers from Levy and Clark [6] and the author's collection of Dhrystone results) shows the execution time of several programs on the same machine (VAX, 1982 and 1985).

There are properties in the languages (calling sequence, pointer semantics, string semantics) that have an obvious influence on execution time even if the source programs look similar and produce the same results.

8.2. Compiler influence

Table 6, taken from the MIPS Performance Brief [7], gives Dhrystone results (as of January 1990) for the MIPS M/2000 with the MIPS C compiler cc2.0; it shows how the different levels of optimization influence execution time.

Not that optimization O4 performs procedure inlining, an optimization not consistent with the ground rules and included in the report for comparison only. On the other hand, the 'strcpy' optimization for Dhrystone is not included in any of the optimization levels for the MIPS C compiler. If it is used, the Dhrystone rate increases considerably.

8.3. Runtime library system

The runtime library system plays a role that is often overlooked when benchmark results are compared. As apparent from *Table 1*, Whetstone spends 40–50% of the execution time in functions of the mathematical subroutines library. The C version of Dhrystone spends 16% of the execution time in the string functions (VAX, Berkeley UNIX 4.3 C); with other systems, the percentage can be higher.

Some systems have two flavors of the mathematical floating-point library, one that is guaranteed to comply with the IEEE floating-point standard, and a faster one that may give less accurate results under some circumstances. For customers that have to rely on the accuracy of floating-point computations, it is important to know which library has been used for benchmark measurements.

8.4. Cache size

It is important to look for the built-in performance boost when the cache size reaches the relevant benchmark size. Depending on the difference of access time for the cache and for main memory, the effect of the cache size can be considerable.

Table 7 summarizes the code sizes (size of the relevant procedures/inner loops) and data sizes (of the main array) of some popular benchmarks. All sizes have been measured for the VAX 11 with the UNIX bsd 4.3 C compiler, with optimization '−O' (code compaction). Of course, the sizes will be different for other architectures and compilers. Typically, RISC architectures will lead to larger code sizes whereas the data size will remain the same.

If the size of the cache is smaller than the relevant benchmark size, a reordering of the code can, for some benchmarks and cache configurations, lead to considerable savings in execution time. Such savings have been reported for Whetstone on MC 68020 systems (reordering of the source program) as well as for Dhrystone on NS 32532, where just a different linkage order can lead to a difference of up to 5% in execution time. One can argue whether the 'good case' or the 'bad case' better represents the system's true characteristics; in any case, customers should be aware of these effects and should know when the standard order of the code has been changed.

9. Small synthetic benchmarks versus real-life programs

From the preceding sections, it has become apparent that with the advent of on-chip caches and of sophisticated optimizing compilers, small benchmarks gradually loose their predictive value. This has been the reason why current efforts like SPEC's activities concentrate on collecting large, real-life programs. One might therefore ask why this article's detailed charcterization of the 'stone-age' benchmarks has been written at all. However, there are reasons why it still makes sense to know as much as possible about these benchmarks:

- The trade press will continue to quote them for some time, and manufacturers will still use them.
- Manufacturers sometimes base their MIPS rating on them.
- For investigations of new architectural designs, e.g. via simulations, the benchmarks can give a useful first approximation.
- For embedded microprocessors with no standard system software (the SPEC suite requires UNIX or an equivalent operating system), nothing else may be available.
- It can be expected that larger benchmarks are also not completely free from distortions by unforeseen effects; the experience gained with smaller benchmarks can help us to be aware of them. For example, it won't be as easy to tweak compilers for the SPEC benchmarks as it is for the small benchmarks – but if it happens, it also will be harder to detect.
- In addition, even though large benchmarks like the SPEC benchmarks are derived from real applications, there can be an inherent problem with these benchmarks: The programs are sometimes quite old, they have to be frozen for benchmarking purposes at an early time. Some of them reflect the programming style of older times, when for example procedure calls were considered expensive and to be avoided – contrary to modern software technology that encourages modularization.
- Since the synthetic benchmarks have well-understood properties, they are good candidates for controlled experiments, e.g. for simluations with varying memory access times, or for compilers. In such usage, there is usually no pressure from marketing departments for the highest possible number, and then these benchmarks can be still quite useful.

10. Performance quotations by manufacturers

Even though, as stated above, small synthetic benchmarks can be still useful for controlled experiments, their use for overall system performance indication is highly questionable. Manufacturers use them in this way often, and the reason is obvious: Whenever for a new machine results are quoted relative to another machine, in particular to the VAX 11/780, the small benchmarks make the new machine look better, for the reasons given above (cache size vs. program size, targeted compiler optimizations). *Table 8* illustrates this point. It gives performance data quoted by the manufacturers for various microprocessors, all converted to numbers relative to a VAX 11/780.

Almost always, small benchmarks make the machine look better, hence their popularity with marketing departments.

11. Consumer advice

As a summary, the following advice can be given to users looking at benchmark numbers for a characterization of machine performance:

- Don't trust MIPS numbers unless there is a clear derivation how they have been obtained.
- Check whether MFLOPS numbers relate to a standard benchmark. Does this benchmark match your application?
- Know the properties of the benchmarks whose results are advertized.
- Be sure that you know all the relevant facts about your system and the manufacturer's benchmarking system:
  Hardware: Clock frequency, memory latency, cache size
  Software: Programming language, code size, data size, compiler version, compiler options, runtime library

- Check code listings of the benchmarks to make sure that apples are compared with apples, and that no illegal optimizations are applied.
- Ask for a well-written 'performance brief' / 'performance report'. Good companies inform you about all the relevant details.

References

[1] K. Dixit, The SPEC benchmarks, *Parallel Comput.*, this issue.
[2] J.J. Dongarra, The LINPACK benchmark: An explanation, in: A.J. Van der Steen, ed. Evaluation of Supercomputers (Chapman and Hall, London, 1990) 1–21.
[3] J.J. Dongarra and E. Grosse, Distribution of mathematical software via electronic mail, *Comm. ACM* 30 (5) (May 1987) 403–407.
[4] J.T. Feo, An analysis of the computational and parallel complexity of the Livermore Loops, *Parallel Comput.* 7 (2) (June 1988) 163–185.
[5] B. Funk, RISC and CISC benchmarks and insights, *Unisys World* (Jan. 1989) 11–12 and 14.
[6] H. Levy and D.W. Clark: On the use of benchmarks for measuring system performance, *Comput. Architecture News* 10 (6) (Dec. 1982) 5–8.
[7] MIPS Computer Systems, Inc; Performance Brief, CPU Benchmarks. Issue 3.9, January 1990, 35 p.
[8] D.A. Patterson, Reduced instruction set computers, *Comm. ACM* 28 (1) (Jan. 1985) 8–21.
[9] O. Serlin, MIPS, Dhrystone and other tales, *Datamation.* (June 1986) 112–118.
[10] B.A. Wichmann, Validation code for the Whetstone Benchmark, Report NPL-DITC 107/88 (National Physical Laboratory, Teddington, UK), March 1988, 16 p.

Appendix 1: How to obtain benchmark sources via electronic mail

Most of the benchmarks discussed in this article can be obtained via electronic mail from several 'mail servers' established at large research institutes [3]. The major mail servers and their electronic mail addresses are shown in *Table E.* Users can get information about the use of these mail servers by sending electronic mail consisting of the line 'send index' to one of the mail servers.

The SPEC benchmarks are available only via magnetic tape from SPEC.

Appendix 2: Additional readings and address information

The following list contains the main reference sources for each of the benchmarks discussed in this article, ordered by benchmark, together with a short characterization. It also names a

Table E
Major mail servers and their addresses

North America:		
uucp:	uunet!research!netlib	Murray Hill, New Jersey
Internet:	netlib@research.att.com	
Internet:	netlib@ornl.gov	Oak Ridge, Tennessee
Europe:		
EUNET/uucp:	nac!netlib	Oslo, Norway
Internet:	netlib@nac.no	
EARN/BITNET:	netlib%nac.no@norunix.bitnet	
X.400:	s = netlib; o = nac; c = no;	
Pacific:		
Internet:	netlib@draci.cs.uow.edu.au	Univ. Wollongong, NSW, Australia

contact person for each of the major benchmarks where readers can get additional information. For information about access to the benchmark sources via electronic mail, see Appendix 1, 'How to obtain benchmark sources via electronic mail'.

1. Whetstone

- H.J. Curnow, and B.A. Wichmann: A synthetic benchmark, *Comput. J.* 19 (1) (1976) 43-49.
  Original publication, explanation of the benchmark design, program (ALGOL 60) in the appendix.
- B.A. Wichmann: Validation code for the Whetstone Benchmark. See [10].
  Discussion of comments made to the original program, explanation of the revised version. The paper contains a program listing of the revised version, in Pascal, including checks for correct execution.
- Contact: Brian A. Wichmann, National Physical Laboratory, Teddington, Middlesex, England TW11 OLW. Phone: +44-81-943-6976; Fax: +44-81-977-7091; Electronic Mail (Internet): baw@seg.npl.co.uk
- Registration of other versions: J.B. Souter, Benchmark Registration, BSI-QAS, P.O. Box 375, Milton Keynes, Great Britain MK146LL.

2. Linpack

- J.J. Dongarra, et al., *LINPACK Users' Guide* (SIAM Publications, Philadelphia, PA, 1976).
  Original publication (not yet as a benchmark), containing the benchmark program as an appendix.
- J.J. Dongarra: Performance of various computers using standard equations software in a FORTRAN environment, *ACM Comput. Architecture News* 18 (1) (Mar. 1990) 17–31,
  Latest published version of the regularly maintained list of LINPACK results, rules for LINPACK measurements.
- J.J. Dongarra: The LINPACK benchmark: An Explanation. See [2].
  Explanation of LINPACK, guide to the interpretation of LINPACK results.
- Contact: Jack J. Dongarra, Computer Science Department, University of Tennessee, Knoxville, TN 37996-1301, USA. Phone: +1-615-974-8295; Fax: +1-615-974-8296; Electronic Mail (Internet): dongarra@cs.utk.edu

3. Dhrystone

- R.P. Weicker, R. Dhrystone: A synthetic systems programming benchmark, *Comm. ACM* 27 (10) (Oct. 1984) 1013–1030.
  Original publication, literature survey on the use of programming languages features, base statistics and benchmark program in Ada.
- R.P. Weicker, Dhrystone benchmark: Rationale for Version 2 and measurement rules, *SIGPLAN Notices* 23 (8) (Aug. 1988) 49-62.
  Version 2.0 of Dhrystone (in C), measurement rules. For the Ada version, a similar article has appeared in Ada Letters 9 (5)(Jul. 1989) 60-82.
- R.P. Weicker, Understanding variations in Dhrystone performance, *Microprocessor Rep.* 3 (5)(May 1989) 16-17.
  What customers should know when C version results of Dhrystone are compared; reiteration of the measurement rules.

- Contact: Reinhold P. Weicker, Siemens Nixdorf Information Systems, STM OS 32, Otto-Hahn-Ring 6, W-8000 München, Germany. Phone: +49-89-636-42436; Fax: +49-89-636-48008; Electronic Mail (Eunet): weicker%ztivax.uucp@unido.uucp; Electronic Mail (Internet): weicker@ztivax.siemens.com.
- Collection of results: Rick Richardson, PC Research, Inc., 94 Apple Orchard Drive, Tinton Falls, NJ 07724, USA; Phone: +1-201-389-8963; Electronic Mail (UUCP): ..!uunet!pcrat!rick

4. Livermore Fortran Kernels

- F.H. McMahon, The Livermore Fortran Kernels: A computer test of the numerical performance range, Lawrence Livermore National Laboratory, Livermore (CA), Technical Report UCRL-53745, Dec. 1986. 179 p.
  Original publication of the benchmark, together with sample results.
- F.H. McMahon, The Livermore Fortran Kernels Test of the numerical performance range, in: J.L. Martin, ed. *Performance Evaluation of Supercomputers* (North-Holland, Amsterdam, (1988) 143–186.
  Reprint of the main part of the original publication.
- J.T. Feo, An analysis of the computational and parallel complexity of the Livermore Loops. See [4].
  Analysis of the Livermore Fortran Kernels with respect to the achievable parallelism.
- Contact: Frank H. McMahon, Lawrence Livermore National Laboratory, L-35, P.O. Box 808, Livermore, CA 94550, USA. Phone: +1-415-422-1647; Electronic Mail (Internet): mcmahon@ocfmail.ocf.llnl.gov

5. Stanford Small Programs Benchmark Set

- Appendix 2 – Stanford Composite Source Code.
  Appendix to : Performance Report 68020 / 68030 32-Bit Microprocessors, Motorola Inc., BR705/D, 1988, pp. A2-1-A2-15.
  The only place where I have seen this benchmark in print; it is normally distributed via informal channels.

6. EDN Benchmarks

- R.D. Grappel, J.E. Hemenway, A tale of four μPs: Benchmarks quantify performance, *EDN*. (Apr. 1, 1981) 179–265.
  Original publication, with benchmarks described in assembler (Code listings for LSI-11/23, 8086, 68000, Z8000).
- W. Patstone, 16-bit-μP benchmarks – An update with explanations, *EDN* (Sep. 16, 1981) 169–203.
  Discussion of results, updated code listings (assembler).

7. Sieve

- J. and G. Gilbreath, Eratosthenes revisited, *Byte* (Jan. 1983) 283-326.
  Program listings in Pascal, C, Forth, FORTRAN IV, BASIC, COBOL, Ada, Modula-2.

8. SPEC Benchmarks

- Benchmark results, *SPEC Newsletter* 1 (1)(Fall 1989) 1–15.
  First published list of results, in the report form required by SPEC.

- K. Dixit, The SPEC benchmarks.
  See [1].
  The paper includes a short characterization of each of the SPEC benchmark programs.
- Contact: SPEC – System Performance Evaluation Cooperative, c/o NCGA, 2722 Merilee Drive, Suite 200, Fairfax, VA 22301, USA. Phone: +1-703-698-9600, Ext. 318; Fax: +1-703-560-2752; Electronic Mail (Internet): spec-ncga@cup.portal.com.