# An Overview of Common Benchmarks

**Reinhold P. Weicker**

**Siemens Nixdorf Information Systems**

The main reason for using computers is to perform tasks faster. This is why performance measurement is taken so seriously by computer customers. Even though performance measurement usually compares only one aspect of computers (speed), this aspect is often dominant. Normally, a mainframe customer can run typical applications on a new machine before buying it. With microprocessor-based systems, however, original equipment manufacturers must make decisions without detailed knowledge of the end user's code, so performance measurements with standard benchmarks become more important.

Performance is a broad area, and traditional benchmarks cover only part of it. This article is restricted to benchmarks measuring hardware speed, including compiler code generation; it does not cover the more general area of system benchmarks (for example, operating system performance). Still, manufacturers use traditional benchmarks in their advertising, and customers use them in making decisions, so it is important to know as much as possible about them. This article characterizes the most often used benchmarks in detail and warns users about a number of pitfalls.

## The ubiquitous MIPS numbers

For comparisons across different instruction-set architectures, the unit MIPS, in its literal meaning of millions of instructions per second (native MIPS), has lost nearly all its significance. This became obvious when reduced instruction-set computer architectures appeared.[1] Operations that can be performed by one CISC (complex instruction-set computer) instruction sometimes require several RISC instructions. Consider the example of a high-level language statement

**"Fair benchmarking" would be less of an oxymoron if those using benchmark results knew what tasks the benchmarks really perform and what they measure.**

```
A = B + C   /* Assume mem operands */
```

With a CISC architecture, this can be compiled into one instruction:

```
add     mem (B), mem (C), mem (A)
```

On a typical RISC, this requires four instructions:

```
load    mem (B), reg (B)
load    mem (C), reg (C)
add     reg (B), reg (C), reg (A)
store   reg (A), mem (A)
```

If both machines need the same time to execute (not unrealistic in some cases), should the RISC then be rated as a 4-MIPS machine if the CISC (for example, a VAX 11) operates at 1 MIPS? The MIPS number in its literal meaning is still interesting for computer architects (together with the CPI number — the average number of cycles necessary for an instruction), but it loses its significance for the end user.

Because of these problems, "MIPS" has often been redefined, implicitly or explicitly, as "VAX MIPS." In this case MIPS is just a performance factor for 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 computer folklore saying that for typical programs a VAX 11/780 performs one million instructions per second. Although this is not true,* the belief is

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



0018-9162/90/1200-0065$01.00 © 1990 IEEE

widespread. When VAX MIPS are quoted, it is important to know what programs form the basis for the comparison and what compilers are used for the VAX 11/780. Older Berkeley Unix compilers produced code up to 30 percent slower than VMS compilers, thereby inflating the MIPS rating of other machines.

The MIPS numbers that manufacturers give for their products can be any of the following:

- MIPS numbers with no derivation. This can mean anything, and flippant interpretations such as "meaningless indication of processor speed" are justified.
- Native MIPS, or MIPS in the literal meaning. To interpret this you must know what program the computation was based on and how many instructions are generated per average high-level language statement.
- Peak MIPS. This term sometimes appears in product announcements of new microprocessors. It is largely irrelevant, since it equals the clock frequency for most processors (most can execute at least one instruction in one clock cycle).
- EDN MIPS, Dhrystone MIPS, or similar. This could mean native MIPS, when a particular program is running. More often it means VAX MIPS (see below) with a specific program as the basis for comparison.
- VAX MIPS. A factor relative to the VAX 11/780, which then raises the following questions: What language? What compiler (Unix or VMS) was used for the VAX? What programs have been measured? (Note that DEC uses the term VUP, for VAX unit of performance, in making comparisons relative to the VAX 11/780. These units are based on a set of DEC internal programs, including some floating-point programs.)

In short, Omri Serlin[2] is correct in saying, "There are no accepted industry standards for computing the value of MIPS."

## Benchmarks

Any attempt to make MIPS numbers meaningful (for example, VAX MIPS) comes down to running a representative program or set of programs. Therefore, we can drop the notion of MIPS and just compare the speed for these benchmark programs.

It has been said that the best benchmark is the user's own application. But this is often unrealistic, since it is not always possible to run the application on each machine in question. There are other considerations, too: The program may have been tailored to run optimally on an older machine; original equipment manufacturers must choose a microprocessor for a whole range of applications; journalists want to characterize machine speed independent of a particular application program. Therefore, the next best benchmark (1) is written in a high-level language, making it portable across different machines, (2) is representative for some kind of programming style (for example, systems programming, numerical programming, or commercial programming), (3) can be measured easily, and (4) has wide distribution.

Obviously, some of these requirements are contradictory. The more representative the benchmark program — in terms of similarity to real programs — the more complicated it will be. Thus, measurement becomes more difficult, and results may be available for only a few machines. This explains the popularity of certain benchmark programs that are not complete application programs but still claim to be representative for a given area.

This article concentrates on the most common "stone age" benchmarks (CPU/memory/compiler benchmarks only) — in particular the Whetstone, Dhrystone, and Linpack benchmarks. These are the benchmarks whose results are most often cited in manufacturers' publications and in the trade press. They are better than meaningless MIPS numbers, but readers should know their properties — that is, what they do and don't measure.

Whetstone and Dhrystone are synthetic benchmarks: They were written solely for benchmarking purposes and perform no useful computation. Linpack was distilled out of a real, purposeful program that is now used as a benchmark.

Tables A-D in the sidebar on pages 68-69 give detailed information about the high-level language features used by these benchmarks. Comparing these advantages with the characteristics of the user's own programs shows how meaningful the results of a particular benchmark are for the user's own applications. The tables contain comparable information for all three benchmarks, thereby revealing their differences and similarities.

All percentages in the tables are dynamic percentages, that is, 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 those normally used in the Fortran version, the language-feature-related statistics refer to the C version of the benchmarks; this was the version for which the modification was performed. However, since most features are similar in the different languages, numbers for other languages should not differ much. The profiling data has been obtained from the Fortran version (Whetstone, Linpack) or the C version (Dhrystone).

## Whetstone

The Whetstone benchmark was the first program in the literature explicitly designed for benchmarking. Its authors are H.J. Curnow and B.A. Wichmann from the National Physical Laboratory in Great Britain. It was published in 1976, with Algol 60 as the publication language. Today it is used almost exclusively in its Fortran version, with either single precision or double precision for floating-point numbers.

The 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. It consisted of several modules, each containing statements of some particular type (integer arithmetic, floating-point arithmetic, "if" statements, calls, and so forth) and ending with a statement printing the results. Weights were 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 matched the distribution observed in the program sample. The weights were chosen in such a way that the program executes a multiple of one million of these Whetstone instructions; thus, benchmark results are given as KWIPS (kilo Whetstone instructions per second) or MWIPS (mega Whetstone instructions per second). This way the familiar term "instructions per second" was retained but given a machine-independent meaning.

A problem with Whetstone is that only one officially controlled version exists — the Pascal version issued with the Pascal Evaluation Suite by the British Standards Institution — Quality Assurance (BSI-QAS). Versions in other languages can be registered with BSA-QAS to ensure

comparability.

Many Whetstone versions copied informally and used for benchmarking have the print statements removed, apparently with the intention of achieving better timing accuracy. 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 in such a way that the time spent in the extra statements becomes insignificant.

Users should know that since 1988 there has been a revised (Pascal) version of the benchmark.[3] Changes were made to modules 6 and 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,[3] performance figures for the two versions should be very similar; however, differences of up to 20 percent cannot be ruled out. The Fortran version has not undergone a similar revision, since with the separate compilation model of Fortran the danger of unintended optimization is smaller (though it certainly exists if all parts are compiled in one unit). All Whetstone data in this article is based on the old version; the language-feature statistics are almost identical for both versions.

**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* is 2,117 bytes (measurement loops only). However, because of the program's nature, 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 number 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 perhaps surprising, result is that Whetstone spends more than half its time in library subroutines rather than in the compiled user code.

The distribution of language features is shown in Tables A-D in the sidebar on pages 68-69. Some properties of Whetstone are probably typical for most numeric applications (for example, a high number of loop statements); other properties belong exclusively to Whetstone (for example, very few local variables).

**Table 1. Procedure profile for Whetstone.***

| Procedure | Percent | 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 | |
| Trigonometric functions | 21.6 | Sin, cos, atan |
| Other math functions | 31.7 | Exp, log, sqrt |
| Library functions | 53.3 | |
| Total | 100 | |

*Because of rounding, all percentages can add up to a number slightly below or above 100.

**Whetstone characteristics.** Some important characteristics should be kept in mind when using Whetstone numbers for performance comparisons.

(1) Whetstone has a high percentage of floating-point data and floating-point operations. This is intentional, since the benchmark is meant to represent numeric programs.

(2) As mentioned above, a high percentage of execution time is spent in mathematical library functions. This property is derived from the statistical data forming the basis of 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.

(3) As evident from Table D in the sidebar, Whetstone uses very few local variables. When Whetstone was written, the issue of local versus global variables was hardly being discussed in software engineering, not to mention in computer architecture. Because of this unusual lack of local variables, register windows (in the Sparc RISC, for example) or good register allocation algorithms for local variables (say, in the MIPS RISC compilers) make no difference in Whetstone execution times.

(4) Instead of local variables, Whetstone uses a handful of global data (several scalar variables and a four-element array of constant size) repeatedly. Therefore, a compiler in which the most heavily used global variables are allocated in registers (an optimization usually considered of secondary importance) will boost Whetstone performance.

(5) Because of its construction principle (nine small loops), Whetstone has an extremely high code locality. A near 100 percent hit rate can be expected even for fairly small instruction 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, reordering of the source code can boost performance up to 15 percent.

## Linpack

As explained by its author, Jack Dongarra[4] from the University of Tennessee (previously Argonne National Laboratory), Linpack didn't originate as a benchmark. When first published in 1976, it was just a collection (a package, hence the name) of linear algebra subroutines often used in Fortran programs. Dongarra, who collects and publishes Linpack results, has now distilled what was part of a "real life" program into a benchmark that is distributed in various versions.[5]

The program operates on a large matrix

*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.

(two-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 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 millions of floating-point operations per second (Mflops); the number of floating-point operations the program executes can be derived from the array size. This terminology means that the nonfloating-point operations are neglected or, stated another way, 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 some-

## Tables covering more than one benchmark

**Table A. Statement distribution in percentages. ***

| 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 (one operator) | 17.5 | 1.4 | - |
| Assignment of an expression (two operators) | 1.0 | 24.3 | 48.5 |
| Assignment of an expression (three operators) | 1.0 | 1.6 | - |
| Assignment of an expression (>three 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 |

*Because of rounding, all percentages can add up to a number slightly below or above 100.

**Table C. Operand data-type distribution in percentages.**

| 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 |

what 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 four, and the loop body contains four groups of statements, for indexes $i$, $i + 1$, $i + 2$, and $i + 3$. 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 subpackage of basic linear algebra subroutines (BLAS). Coded BLAS (as opposed to Fortran BLAS) means that these subroutines have been rewritten in assembly language. Dongarra has stopped collecting and publishing results for the coded BLAS version and requires that only the Fortran version of these subroutines be used unchanged. However, some results for coded BLAS versions are still cited elsewhere. Computing the execution-time ratio between coded BLAS and Fortran BLAS versions for the same machine offers insights about the Fortran compiler's code optimization quality: For some machines the ratio is 1.2 to 1; for others it can be as high as 2 to 1.

**Table B. Operator distribution in percentages.**

| 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 | 23.9 | 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 | 28.2 |
| <, <= (incl. loop control) | 10.1 | 10.7 | 14.5 |
| Other relational operators | 11.7 | 2.8 | 0.6 |
| Relational | 21.8 | 13.5 | 15.1 |
| Logical | 3.3 | - | 0.2 |
| Indexing (one-dimensional) | 5.9 | 24.5 | 42.3 |
| Indexing (two-dimensional) | 3.4 | - | - |
| Indexing | 9.3 | 24.5 | 42.3 |
| Record selection | 7.6 | - | - |
| Record selection via pointer | 15.1 | - | - |
| Record selection | 22.7 | - | - |
| Address operator (C) | 5.0 | 3.6 | - |
| Indirection operator (C) | 8.4 | 3.6 | - |
| C-specific operators | 13.4 | 7.2 | - |
| Total | 100 | 100 | 100 |

**Table D. Operand locality distribution in percentages.**

| 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 |

**Size, procedure profile, and language-feature distribution.** The Linpack data reported here is 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 4,537 bytes. The length for individual subprograms varies between 111 and 1,789 bytes; the most heavily used subprogram, saxpy, is 234 bytes long. Data size, in the standard version, is dominated by an array of 100 × 100 real numbers. For 32-bit machines, this means that with single precision, 40 Kbytes are used for data (80 Kbytes with double precision).

Table 2 shows the distribution of execution time in the various subroutines. The most important observation from the table is that more than 75 percent of Linpack's execution time is spent in a 15-line subroutine (called saxpy in the single-precision version and daxpy in the double-precision version). Dongarra[4] reports that on most machines the percentage is even higher (90 percent). Because of this extreme concentration of execution time in the saxpy subroutine, and because of the time-consuming instrumentation method for obtaining the measurements, language-feature distribution has been measured only for the saxpy subroutine (rolled version).

Table A in the sidebar shows that very few statement types (assignment with multiplication and addition, and "for"

**Table 2. Procedure profile for Linpack.**

| Procedure | Percent | 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 | |
| Library functions | 0.0 | |

statements) make up the bulk of the subroutine and, therefore, of Linpack itself. The data is mostly reference parameters (array values) or local variables (indexes); there are hardly any global variables.

**Linpack characteristics.** To interpret performance characterizations by Linpack Mflops, it helps to know the benchmark's main characteristics:

- As expected for a numeric benchmark, Linpack has a high percentage of floating-point operations, though only a few are actually used. For example, the program has no floating-point divisions. In striking contrast to Whetstone, no mathematical functions are used at all.
- The execution time is spent almost exclusively 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. So, in making comparisons, it is important to know whether Linpack Mflops for different machines have been computed using the same array dimensions. Also, Linpack can 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; we can assume that manufacturers report the better number.

# Dhrystone

As the name indicates, Dhrystone was developed much as Whetstone was; it is a synthetic benchmark that I published in 1984. The original language of publication is Ada, although it uses only the Pascal subset of Ada and was intended for easy translation to Pascal and C. It is used mainly in the C version.

The basis for Dhrystone is a literature survey on the distribution of source language features in nonnumeric, system-type programming (operating systems, compilers, editors, and so forth). In addition to the obvious difference in data types (integral versus floating-point), numeric and system-type programs have other differences, too: System programs contain fewer loops, simpler computational statements, and more "if" statements and procedure calls.

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

**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 1,039 bytes. Table 3 shows the distribution of execution time spent in its subprograms.

The percentage of time spent in string operations is highly language dependent; it drops to 10 percent instead of 16 percent if the Pascal (or Ada) version is used (measurement for Berkeley 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 fewer 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 but 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.

**Dhrystone characteristics.** Familiarity with the benchmark's main characteristics, described below, is important when interpreting Dhrystone performance characterizations.

- 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 percent.
- Unlike Whetstone, Dhrystone contains hardly any loops within the main measurement loop. Therefore, for microprocessors with small instruction caches (below 1,000 bytes), almost all instruction accesses are cache misses. But as soon as the cache becomes larger than the measurement loop, all instruction accesses are cache hits.
- Only a small amount of global data is manipulated, and the data size cannot be scaled as in Linpack.
- No attempt has been made to thwart optimizing compilers. The goal was for the program to reflect typical programming style; it should be just as optimizable as normal programs. An exception is the optimization of dead-code removal. Since in Version 1 the computation results were not printed or used, optimizing compilers were able to recognize many statements as

dead code and suppress code generation for these statements. In Version 2, this has been corrected.

**Ground rules for Dhrystone number comparisons.** Because of Dhrystone's peculiarities, users should be sure to observe certain ground rules when comparing Dhrystone results. First, the version used should be 2.1; the earlier version, 1.1, leaves too much room for distortion of results by dead-code elimination.

Second, the two modules must be compiled separately, and procedure merging (in-lining) is not allowed for user procedures. ANSI C, however, allows in-lining of library routines (relevant for string routines in the C version of Dhrystone).

Third, when processors are compared, the same programming language must be used on both. For compilers of equal quality, Pascal and Ada numbers can be about 10 percent 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 in-line code) for the string operations.

Therefore, for a meaningful comparison of C-version results, it helps to be able to answer certain questions:

(1) Are the string routines written in machine code?

(2) Are the string routines implemented as in-line code?

(3) Does the compiler use the fact that in the "strcpy" statement the source operand has a fixed length? If it 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 through 3 above, just as they can for programs in general. For processor comparisons, however, it is important that the compilers used apply the same amount of optimization; otherwise, optimization differences may overshadow CPU speed differences. This usually requires an inspection of the generated machine code and the C library routines.

**Table 3. Dhrystone procedure profile.**

| Procedure | Percent | What is done there |
|---|---|---|
| Main program | 18.3 | |
| User procedures | 65.7 | |
| User code | 84.0 | |
| strcpy | 8.0 | String copy (string constant) |
| strcmp | 8.1 | String comparison (string variables) |
| Library functions | 16.1 | |
| Total | 100 | |

## Other benchmarks

In addition to the most often quoted benchmarks explained above, several other programs are used as benchmarks, including

- Livermore Fortran Kernels,
- Stanford Small Programs Benchmark Set,
- EDN benchmarks,
- Sieve of Eratosthenes,
- Rhealstone, and
- SPEC benchmarks.

These range from small, randomly chosen programs such as Sieve, to elaborate benchmark suites such as Livermore Fortran Kernels and SPEC benchmarks.

**Livermore Fortran Kernels.** The Livermore Fortran Kernels, also called the Lawrence Livermore Loops, consist of 24 kernels, or inner loops, of numeric computations from different areas of the physical sciences. The author, F.H. McMahon of Lawrence Livermore National Laboratory, 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 we might expect, these kernels contain many floating-point computations and a high percentage of array accesses. Several kernels contain vectorizable code; some contain code that is vectorizable if rewritten. (Feo[6] provides a detailed discussion of the Livermore Loops.) McMahon characterizes the representativity of the Livermore Loops as follows:

The net Mflops rate of many Fortran programs and work loads will be in the subrange between the equi-weighted harmonic and arithmetic means, depending on the degree of code parallelism and optimization. The Mflops metric provides a quick measure of the average efficiency of a computer system, since its peak computing rate is well known.

**Stanford Small Programs Benchmark Set.** Concurrent with development of the first RISC systems at Stanford University and the University of California, Berkeley, John Hennessy and Peter Nye at Stanford's Computer Systems Laboratory collected a set of small programs (one page or less of source code for each program). These programs became popular mainly because they were the basis for the first comparisons of RISC and CISC processors. They have now been packed into one C program containing eight integer programs — Permutations, Towers of Hanoi, Eight Queens, Integer Matrix Multiplication, Puzzle, Quicksort, Bubble Sort, and Tree Sort — and two floating-point programs — Floating-point Matrix Multiplication and Fast Fourier Transformation.

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

There is no standardized method for 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. Perhaps a better alternative, used by Sun and MIPS, is to compute the geometric mean of the individual programs' execution times.

**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 |

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

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

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

This subset of the EDN benchmarks has been used in Bud Funk's comparison of RISC and CISC processors.[7] There seems to be no standard C version of the EDN benchmarks; the programs are disseminated informally.

**Sieve of Eratosthenes.** One of the most popular programs for benchmarking small PCs is the Sieve of Eratosthenes, sometimes called "Primes." It computes all prime numbers up to a given limit (usually 8,192). The program has some unusual characteristics. For example, 33 percent of the dynamically executed statements are assignments of a constant; only 5 percent are assignments with an expression at the right-hand side. There are no "while" statements and no procedure calls; 50 percent of the statements are loop control evaluations. All operands are integer operands, and 58 percent of them are local variables.

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

**SPEC benchmarks.** Probably the most important current benchmarking effort is SPEC — the systems performance evaluation cooperative effort. It started because benchmarking experts at various companies felt that most previously existing benchmarks (usually small programs) were 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 percent for these benchmarks. Furthermore, once a small program becomes popular as a benchmark, compiler writers are inclined (or forced) to "tweak" their compilers into optimizations particularly beneficial to this benchmark — for example, the string optimizations for Dhrystone.

SPEC's goal is to collect, standardize, and distribute large application programs that can be used as benchmarks. This is a nontrivial task, since realistic programs previously used in benchmarking (for example, 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, and Sun; subsequently, AT&T, Bull, CDC, Compaq, Data General, DEC, Dupont, Fujitsu, IBM, Intel, Intergraph, Motorola, NCR, Siemens Nixdorf, Silicon Graphics, Solbourne, Stardent, and Unisys became members.

In October 1989, SPEC released its first set of 10 benchmark programs. Table 4 contains only a rough characterization of the programs; J. Uniejewski[8] provides a more detailed discussion. Because a license must 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 performance relative to a VAX 11/780 using VMS compilers. Results for several computers of SPEC member companies are contained in the regular *SPEC Newsletter* (see Additional reading and address information). 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 that gives, in addition to the raw data, the relative performance for each benchmark program separately. Thus, users can select the subset of performance numbers for which the programming language and/or the application area best matches their applications.

## Non-CPU influences in benchmark performance

In trade journals and advertisements, manufacturers usually credit good bench-

mark numbers only to the hardware system's speed. With microprocessors, this is reduced even more to the CPU speed. However, the preceding discussion makes it clear that other factors also have an influence — for example, the programming language, the compiler, the runtime library functions, and the memory and cache size.

**Programming-language influence.** Table 5 (numbers from Levy and Clark[9] and my own collection of Dhrystone results) shows the execution time of several programs on the same machine (VAX, 1982 and 1985). Properties of the languages (calling sequence, pointer semantics, and string semantics) obviously influence execution time even if the source programs look similar and produce the same results.

**Table 5. Performance ratio for different languages (larger is better, C = 1): Stanford programs.**

| Program | 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 |

**Compiler influence.** Table 6, taken from the MIPS Performance Brief,[10] gives Dhrystone results (as of January 1990) for the MIPS M/2000 with the MIPS C compiler cc2.0. The table shows how the different levels of optimization influence execution time.

Note that optimization "O4" performs procedure in-lining, 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.

**Table 6. Compiler optimization levels in Dhrystones/sec.**

| Optimization Level | V. 1.1 | V. 2.1 |
|---|---|---|
| No opt., no "register" attribute | 30,700 | 31,000 |
| No opt., with "register" attribute | 32,600 | 32,400 |
| Optimization "O," no "register" attribute | 39,700 | 36,700 |
| Optimization "O," with "register" attribute | 39,700 | 36,700 |
| Optimization "O3" | 43,100 | 39,400 |
| Optimization "O4" | 46,700 | 43,200 |

**Runtime library system.** The role of the runtime library system is often overlooked when benchmark results are compared. As apparent from Table 1, Whetstone spends 40 to 50 percent of the execution time in functions of the mathematical subroutines library. The C version of Dhrystone spends 16 percent 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: The first is guaranteed to comply with the IEEE floating-point standard; the second is faster and may give less accurate results under some circumstances. Customers who must rely on the accuracy of floating-point computations should know which library was used for benchmark measurements.

**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 between access times for the cache and the main memory, cache size can have a considerable effect.

Table 7 summarizes the code sizes (size of the relevant procedures/inner loops) and data sizes (of the main array) for 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 differ for other architectures and compilers. Typically, RISC architectures lead to larger code sizes, whereas the data size remains the same.

If the cache is smaller than the relevant benchmark, reordering 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 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 percent in execution time. It is debatable whether the "good case" or the "bad case" better represents the system's true characteristics. In any event, customers should be aware of these effects and know when the standard order of the code has been changed.

**Table 7. Size in bytes for some popular benchmarks.**

| Program | Code | Data |
|---|---|---|
| Whetstone | ~ 256 | 16 |
| Dhrystone | 1,039 | - |
| Linpack (saxpy) (100×100 version) | 234 | 40,000 |
| Sieve (standard version) | 160 | 8,192 |
| Quicksort (standard version) | 174 | 20,000 |
| Puzzle | 1,642 | 511 |
| Ackermann | 52 | - |

## Small, synthetic benchmarks versus real-life programs

It should be apparent by now that with the advent of on-chip caches and sophisticated optimizing compilers, small benchmarks gradually lose their predictive value. This is why current efforts like SPEC's

## Obtaining benchmark sources via e-mail

Most of the benchmarks discussed in this article can be obtained via electronic mail from several mail servers established at large research institutes.[5] The major mail servers and their electronic mail addresses are shown below. Users can get information about the use of these mail servers by sending electronic mail consisting of the line "send index" to any of the mail servers.

The SPEC benchmarks are available only via magnetic tape.

| | | |
|---|---|---|
| **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. of Wollongong, NSW, Australia |

activities concentrate on collecting large, real-life programs. Why, then, should this article bother to characterize in detail these "stone age" benchmarks? There are several reasons:

(1) Manufacturers will continue to use them for some time, so the trade press will keep quoting them.

(2) Manufacturers sometimes base their MIPS rating on them. An example is IBM's (unfortunate) decision to base the published (VAX-relative) MIPS numbers for the IBM 6000 workstation on the old 1.1 version of Dhrystone. Subsequently, DEC and Motorola changed the MIPS computation rules for their competing products, also basing their MIPS numbers on Dhrystone 1.1.

(3) For investigating new architectural designs — via simulations, for example — the benchmarks can provide a useful first approximation.

(4) For embedded microprocessors with no standard system software (the SPEC suite requires Unix or an equivalent operating system), nothing else may be available.

(5) We can expect that larger benchmarks will not be completely free of distortions from unforeseen effects either. Experience gained with smaller benchmarks can help us be aware of such effects. 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.

Advice for users looking at benchmark numbers to characterize machine performance should begin with a warning not to trust MIPS numbers unless their derivation is clearly explained. Here are some other things to watch for:

- Check whether Mflops numbers relate to a standard benchmark. Does this benchmark match your applications?
- Know the properties of the benchmarks whose results are advertised.
- Be sure you know all the relevant facts about your system and the manufacturer's benchmarking system. For hardware this includes clock frequency, memory latency, and cache size; for software it includes programming language, code size, data size, compiler version, compiler options, and runtime library.
- Check benchmark code listings to make sure apples are compared with apples and that no illegal optimizations are applied.
- Ask for a well-written performance report. Good companies provide all relevant details. ■

## References


1. D.A. Patterson, "Reduced Instruction-Set Computers," *Comm. ACM*, Vol. 28, No. 1, Jan. 1985, pp. 8-21.
2. O. Serlin, "MIPS, Dhrystones, and Other Tales," *Datamation*, June 1986, pp. 112-118.
3. B.A. Wichmann, "Validation Code for the Whetstone Benchmark," Tech. Report NPL-DITC 107/88, National Physical Laboratory, Teddington, UK, Mar. 1988.
4. J.J. Dongarra, "The Linpack Benchmark: An Explanation," in *Evaluating Supercomputers*, Aad J. Van der Steen, ed., Chapman and Hall, London, 1990, pp. 1-21.
5. J. Dongarra and E. Grosse, "Distribution of Mathematical Software via Electronic Mail," *Comm. ACM*, Vol. 30, No. 5, May 1987, pp. 403-407.
6. J.T. Feo, "An Analysis of the Computational and Parallel Complexity of the Livermore Loops," *Parallel Computing*, Vol. 7, No. 2, June 1988, pp. 163-185.
7. B. Funk, "RISC and CISC Benchmarks and Insights," *Unisys World*, Jan. 1989, pp. 11-14.
8. J. Uniejewski, "Characterizing System Performance Using Application-Level Benchmarks," *Proc. Buscon*, Sept. 1989, pp. 159-167. Partial publication in *SPEC Newsletter*, Vol. 2, No. 3, Summer 1990, pp. 3-4.
9. H. Levy and D.W. Clark, "On the Use of Benchmarks for Measuring System Performance," *Computer Architecture News*, Vol. 10, No. 6, Dec. 1982, pp. 5-8.
10. MIPS Computer Systems, Inc., *Performance Brief, CPU Benchmarks*, Issue 3.9, Jan. 1990, p. 35.


## Additional reading and address information

Following are the main reference sources for each of the benchmarks discussed in this article, together with a short characterization. A contact person is identified for each of the major benchmarks so that readers can get additional information. For information about access to the benchmark sources via electronic mail, see the sidebar "Obtaining benchmark sources via e-mail."

### Whetstone

Curnow, H.J., and B.A. Wichmann, "A Synthetic Benchmark," *The Computer J.*, Vol. 19, No. 1, 1976, pp. 43-49. Original publication, explanation of the benchmark design, program (Algol 60) in the appendix.

Wichmann, B.A., "Validation Code for the Whetstone Benchmark," see Reference 3. Discussion of comments made to the original program, explanation of the revised version. 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, Internet baw@seg.npl.co.uk.

Registration of other versions: J.B. Souter, Benchmark Registration, BSI-QAS, PO Box 375, Milton Keynes, Great Britain MK14 6LL.

### Linpack

Dongarra, J.J., et al., *Linpack Users' Guide*, SIAM Publications, Philadelphia, Pa., 1976. Original publication (not yet as a benchmark), contains the benchmark program as an appendix.

Dongarra, J.J., "Performance of Various Computers Using Standard Equations Software in a Fortran Environment," *Computer Architecture News*, Vol. 18, No. 1, Mar. 1990, pp. 17-31. Latest published version of the regularly maintained list of Linpack results, rules for Linpack measurements.

Dongarra, J.J., "The Linpack Benchmark: An Explanation," see Reference 4. Explanation of Linpack, guide to interpretation of Linpack results.

Contact: Jack J. Dongarra, Computer Science Dept., Univ. of Tennessee, Knoxville, TN 37996-1301; phone (615) 974-8295, fax (615) 974-8296, Internet dongarra@cs.utk.edu.

### Dhrystone

Weicker, R.P., "Dhrystone: A Synthetic Systems Programming Benchmark," *Comm. ACM*, Vol. 27, No. 10, Oct. 1984, pp. 1,013-1,030. Original publication, literature survey on the use of programming language features, base statistics and benchmark program in Ada.

Weicker, R.P., "Dhrystone Benchmark: Rationale for Version 2 and Measurement Rules," *SIGPlan Notices*, Vol. 23, No. 8, Aug. 1988, pp. 49-62. Version 2.0 of Dhrystone (in C), measurement rules. For the Ada version, a similar article appeared in *Ada Letters*, Vol. 9, No. 5, July 1989, pp. 60-82.

Weicker, R.P., "Understanding Variations in Dhrystone Performance," *Microprocessor Report*, Vol. 3, No. 5, May 1989, pp. 16-17. What customers should know when C-version results of Dhrystone are compared; reiteration of measurement rules.

Contact: Reinhold P. Weicker, Siemens Nixdorf Information Systems, STM OS 32, Otto-Hahn-Ring 6, W-8000 München 83, Germany; phone 49 (89) 636-42436, fax 49 (89) 636-48008, Internet: weicker@ztivax.siemens.com; Eunet: weicker%ztivax.uucp@unido.uucp.

Collection of results: Rick Richardson, PC Research, Inc., 94 Apple Orchard Dr., Tinton Falls, NJ 07724; phone (201) 389-8963, e-mail (UUCP) ...!uunet!pcrat!rick.

### Livermore Fortran Kernels

Feo, J.T., "An Analysis of the Computational and Parallel Complexity of the Livermore Loops," see Reference 6. Analysis of the Livermore Fortran Kernels with respect to the achievable parallelism.

McMahon, F.H., "The Livermore Fortran Kernels: A Computer Test of the Numerical Performance Range," Tech. Report UCRL-53745, Lawrence Livermore National Laboratory, Livermore, Calif., Dec. 1986, p. 179. Original publication of the benchmark with sample results.

McMahon, F.H., "The Livermore Fortran Kernels Test of the Numerical Performance Range," in *Performance Evaluation of Supercomputers*, J.L. Martin, ed., North Holland, Amsterdam, 1988, pp. 143-186. Reprint of main part of the original publication.

Contact: Frank H. McMahon, Lawrence Livermore National Laboratory, L-35, PO Box 808, Livermore, CA 94550; phone (415) 422-1647, Internet mcmahon@ocfmail.ocf.llnl.gov.

### 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. This is the only place I have seen this benchmark in print; it is normally distributed via informal channels.

### EDN benchmarks

Grappel, R.D., and J.E. Hemenway, "A Tale of Four µPs: Benchmarks Quantify Performance," *EDN*, Apr. 1, 1981, pp. 179-265. Original publication with benchmarks described in assembler (code listings for LSI-11/23, 8086, 68000, and Z8000).

Patstone, W., "16-bit-µP Benchmarks — An Update with Explanations," *EDN*, Sept. 16, 1981, pp. 169-203. Discussion of results, updated code listings (assembler).

### Sieve

Gilbreath, J., and G. Gilbreath, "Eratosthenes Revisited," *Byte*, Jan. 1983, pp. 283-326. Program listings in Pascal, C, Forth, Fortran IV, Basic, Cobol, Ada, and Modula-2.

### SPEC benchmarks

"Benchmark Results," *SPEC Newsletter*, Vol. 1, No. 1, Fall 1989, pp. 1-15. First published list of results, in the report form required by SPEC.

Uniejewski, J., "Characterizing System Performance Using Application-Level Benchmarks," see Reference 8. This paper includes a short characterization of each SPEC benchmark program.

Contact: SPEC — System Performance Evaluation Cooperative (Kim Shanley, Secretary), c/o Waterside Associates, 39150 Paseo Padre Pkwy., Suite 350, Fremont, CA 94538; phone (415) 792-2901, fax (415) 792-4748, Internet shanley@cup.portal.com.

**Reinhold P. Weicker** is a senior staff engineer with Siemens Nixdorf Information Systems AG in Munich, Germany. His research interests include performance evaluation with benchmarks and its relation to CPU architecture and compiler code generation. He wrote the often-used Dhrystone benchmark while working on the CPU architecture team for the i80960 microprocessor. Previously, he performed research in theoretical computer science at the University of Hamburg, Germany, and was a visiting assistant professor at Pennsylvania State University.

Weicker received a diploma degree in mathematics and a PhD in computer science from the University of Erlangen-Nürnberg. He is a member of the IEEE Computer Society, the ACM, and the Gesellschaft für Informatik.

The author can be contacted at Siemens Nixdorf Information Systems AG, Otto-Hahn-Ring 6, W-8000 München 83, Germany; Internet: weicker@ztivax.siemens.com; Eunet: weicker%ztivax.uucp@unido.uucp.