Computer benchmarking: paths and pitfalls

The most popular way of rating computer performance can confuse as well as inform; avoid misunderstanding by asking just what the benchmark is measuring

SYSTEMS
COMPUTERS

Computing power is now crucial in many areas of research and engineering, from theoretical mathematics to circuit design to bioengineering.

But measuring that power is an imprecise art at best. The performance of a computer is a function of many interrelated considerations, including the application at hand, the size of the problem, the algorithm to solve it, the level of human effort to optimize the program, the compiler's ability to optimize, the age of the compiler, the operating system, and the architecture of the computer, to name only a few. No single approach to evaluation addresses the requirements of everyone who needs to measure performance. There is no universal metric of value.

One of the most widely used techniques is benchmarking—running a set of well-known programs on a machine to compare its performance with that of others. The origins of the word benchmark lie in topographic surveying, in which a stationary object whose latitude, longitude, and elevation have been measured is identified by a benchmark to serve as a reference point in tidal observations and surveys.

Benchmarking and other evaluations measure new computer systems either in absolute terms (will the proposed system do the job?) or relative terms (which of several products will perform best in a given context?). These evaluations are important at all phases of the life cycle of a computer [see "Evaluation goes beyond benchmarking, p. 41]. Designers use benchmark tests to help them choose between alternative architectures and implementations; buyers rely on them when deciding which system to purchase; and users can infer from them which coding styles will lead to optimal program execution in their computing environment.

Although benchmarks are essential in performance evaluation, simple-minded application of them can produce misleading results. In fact, bad benchmarking can be worse than no benchmarking at all. If a performance evaluation is to be effective, it will include:

- Accurate characterization of the workload.
- Initial tests using simple programs.
- Further tests with programs that approximate ever more closely the jobs that are part of the workday.

Scope of the benchmarking problem

Good performance evaluation becomes more critical for the larger, more powerful, costlier computers. Benchmarks and additional evaluations tend to be most painstaking in the [illegible]puter arena, where millions of dollars are spent on each machine purchased and users demand maximum performance. Evaluation with relatively simple benchmarks such as Whetstones may suffice for smaller computers, both because their price does not justify the cost of full-scale evaluation and because they have relatively simple architectures. The performance of a typical small computer generally will not vary much from application to application. Larger computers, with virtual memory, special attachments to accelerate floating-point performance, and other costly architectural enhancements, are more sensitive to the characteristics of their applications.

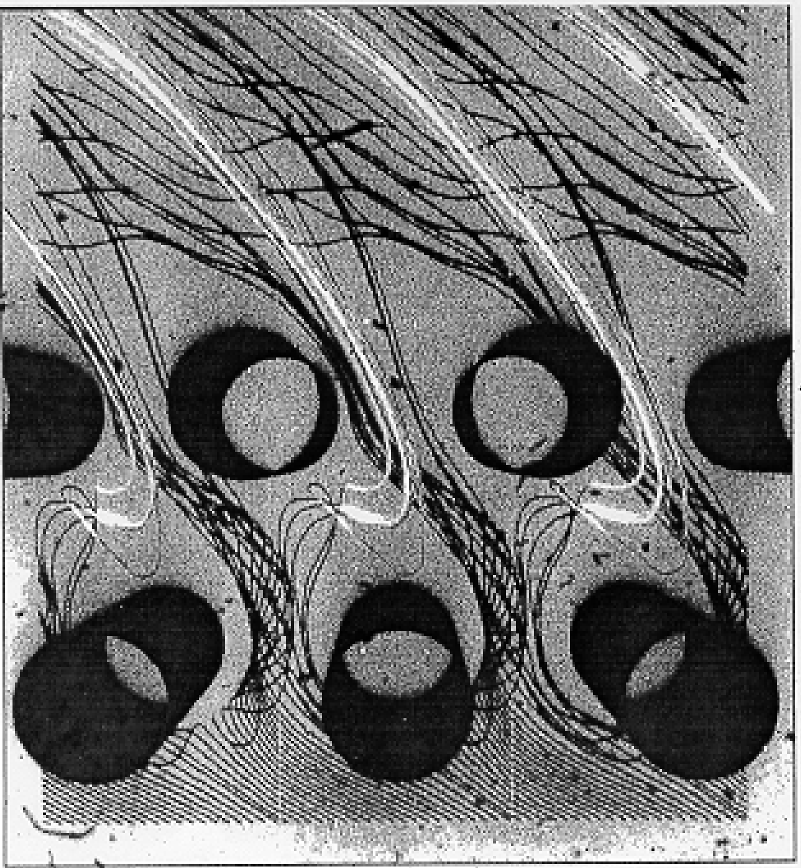
Evaluation of supercomputers is the most complex, because the range of architectural types available highlights the hardware and software issues that distort benchmark results. In terms of central processing unit speed, the CDC 7600 (the last scalar supercomputer design) typically executed about 2 to 6 million floating point operations per second (megaflops). The Cray-1, a vector design, executes anywhere from 6 to 140 megaflops, depending on how well a program exploits its advanced design; for other vector designs, the vector-to-scalar performance ratio may exceed 100 to 1.

Parallel computers—collections of cooperating processors—add another dimension to the task of computer evaluation. At best, the performance of a vector multiprocessor is the sum of the vector performance of all of its processing units; at worst, it runs only as fast as a single scalar processor. The total performance range is equal to the number of processors multiplied by the range of performance of its individual processing elements. A 32-processor machine with a 90-to-1 ratio of vector and scalar processing rates could execute some programs 2880 times as fast as others. Because the margin of error in benchmarking is proportional to the performance range of the system being studied,

Jack Dongarra Argonne National Laboratory
Joanne L. Martin IBM Thomas J. Watson Research Center
Jack Worlton Los Alamos National Laboratory

Defining terms

Gaussian elimination: a method for solving systems of linear equations based on manipulation of the matrix representing those equations.
Kernel: the central portion of a program, containing the bulk of its calculations, which consumes the most execution time.
Pipelining: a computer design technique in which an operation is broken down into subsidiary tasks, each carried out in overlapped fashion.
Scalar processing: calculations performed one at a time.
Tuning: making changes in a program to improve its perfor[illegible] without altering its results.
[illegible]interface: the collection of screen formats, editing tools, commands, and software tools by which a user interacts with a computer.
Vector processing: a method for carrying out many repetitive mathematical operations with a single computer instruction.
Workload: the mix of different types of program typically run at a given worksite; major characteristics include I/O requirements, amount and kinds of computation, and degree of vectorization possible.



This simulation of fluid flow over an array of cylinders is an example of the kind of computing for which performance evaluation is vital. Depending on how well the structure of scientific software matches the computer hardware, execution rates can vary by factors of 100 or more.

accurate benchmarking of supercomputers obviously becomes ever more difficult.

How to avoid the pitfalls

The point of constructing a benchmark is not to produce a single number representing a computer's performance. Instead, the intent is to understand how overall performance is affected by system components such as architecture, compiler, operating system, and peripherals, and to create a set of realistic expectations for users. For designers, the results of benchmarking will assist in optimizing hardware and software; for end users, good benchmarks will teach programming styles that take advantage of architectural features.

Evaluators who do benchmarks in pursuit of a single,, all-encompassing number can end up with meaningless results if they commit these errors:

- Neglecting to characterize the workload.
- Selecting kernels that are too simplistic.
- Using programs or algorithms adapted to a specific computer system.
- Running benchmarks under inconsistent conditions.
- Selecting inappropriate workload measures.
- Neglecting the special needs of the users.
- Ignoring the difference between the frequency and the duration of execution.

Although neglecting to characterize the workload is the most obvious mistake, it is also one of the most frequent. The tendency is to believe that intuition serves well when considering the workload that is to execute on a given system. For example, researchers at Los Alamos National Laboratory in New Mexico once thought that the compiler for a Cray supercomputer would vectorize most of a large numerical program. They based their performance estimates on that assumption. However, the program turned out to be only 30 percent vectorized, and thus ran at only one-tenth the speed they had estimated. In contrast, Frank McMahon of the Lawrence Livermore National Laboratory, Livermore, Calif., has reported results from studies conducted on the workload there that support a strong correlation between performance results on the Livermore Fortran Kernels (or Livermore Loops) and on the overall workload [see "Program kernels for benchmarking, p. 43].

Simple test programs are certainly the easiest to construct, port, and analyze, but they may also be the least useful, especially if they are overinterpreted. Matrix multiplication is used all too often for measurements of system performance. If a vector or parallel system performs poorly for matrix multiplication, it will probably perform worse still for a workload consisting of more general computations—but the converse is not true. Good performance for matrix multiplication does not promise the same for a general workload, because matrix multiplication can be implemented very efficiently on parallel and vector architectures, whereas many other computations cannot.

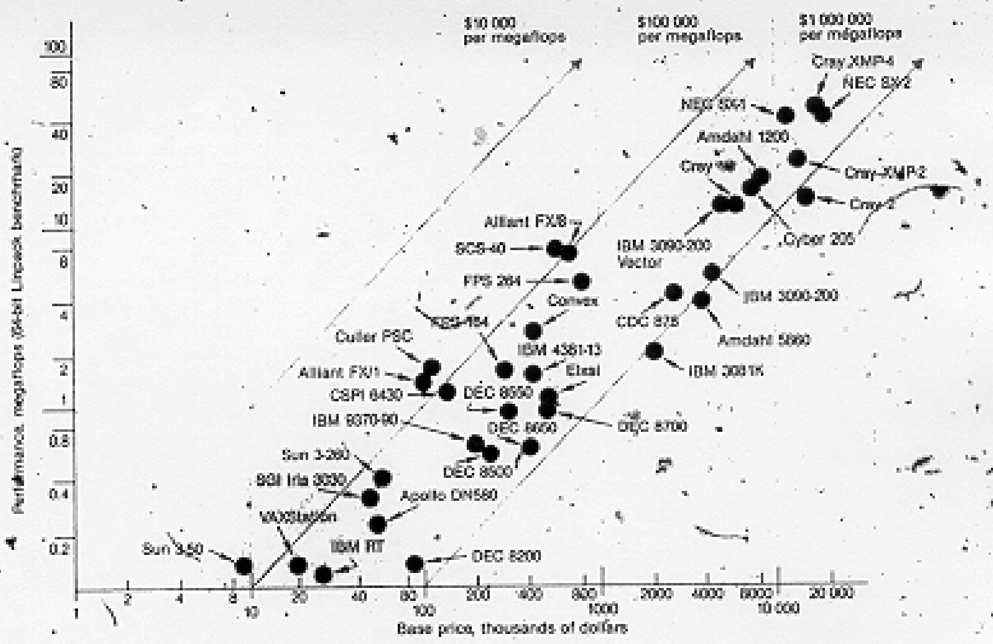
Simple tests are valuable mainly in helping to evaluate a system's attainable peak processing rate, as well as perhaps some of the barriers to attaining that peak. In 1986, Walid Abu-Sufah and Allen Malony at the University of Illinois in Champaign-Urbana used a set of basic operations to test the performance of the Alliant FX/8, a new minisupercomputer. The simplicity of their test programs helped in determining the performance implications of variables like vector length, cache size, partitioning of vectors across multiple processors, and the multiprocessing overhead. It was possible to consider the impact of stride (the spacing of stored elements in memory) and locality of reference. They were also able to infer the programming structures that are well suited to the FX/8.

A more subtle mistake than relying on basic operations is the failure to recognize that all programs are adapted to some extent to the specific computer system for which they were written. Even if an application is coded in a high-level language, the choices of mathematical model, numerical method, algorithm, and program definition that transform the physical problem into computer program result in a specific implementation of a general problem. "Plain vanilla" code does not exist. The question is, how drastically should benchmarks be modified during the evaluation of a system? The answer depends on the context in which the experiments are made, as well as on the perspective of the investigator. Modified benchmarks will not predict the performance of unmodified programs.

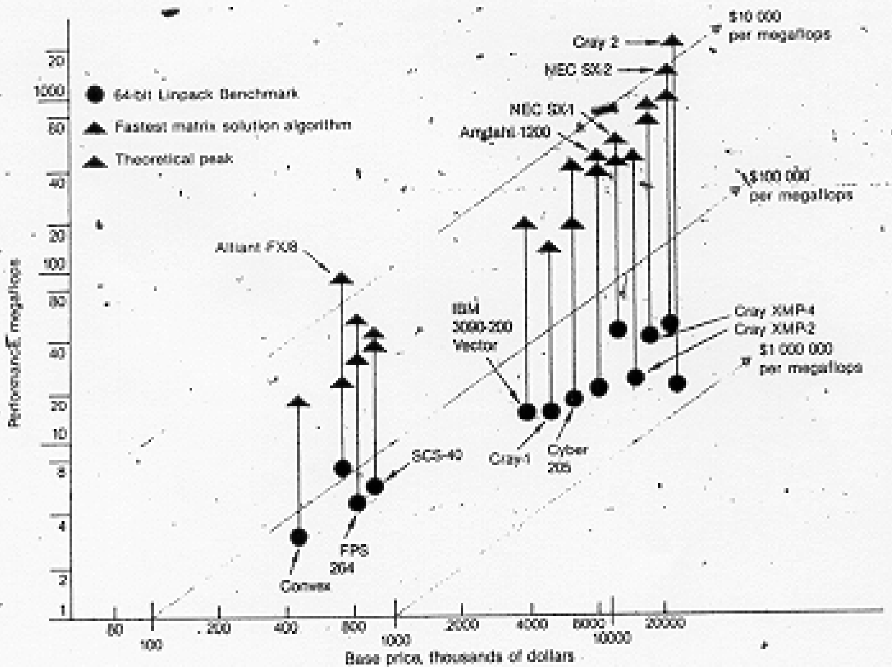
A variation in the conditions for running benchmarks can

Widely used benchmarks

Livermore Loops: computational routines extracted from programs at Lawrence Livermore National Laboratory; used to test scalar and vector floating-point performance.
Linpack: linear algebra software library; used to test scalar and vector floating-point performance.
Whetstones: synthesized benchmark of basic arithmetic; used to test performance of midsize and small computers.
Dhrystones: synthesized benchmark for system-programming operations; used to test nonnumeric performance of midsize and smaller computers.



When ranked by their performance on the Linpack benchmark, various computers span a price-performance range from less than $100 000 per megaflops to well over $1 million. (Another benchmark, such as the Livermore Loops, will give slightly different rankings.) The best price-performance machines appear to be Sun-3 workstations, the Culler PSC, and the Alliant FX/1, but this price-performance benefit will be of little use to scientists who require the large memory or full power of a supercomputer. Nevertheless it raises a question about the tradeoffs between a case in which many users share the largest and most powerful machine available, and the alternative of sharing several less powerful systems. The machines fall into four broad categories: supercomputer, minisupercomputer, superminicomputer, and workstation. The supercomputers cost between $1 million and $10 million each and have benchmark performance between 10 and 100 megaflops. The minisupercomputers cost between $100 000 and $1 million and execute 1 to 10 megaflops. The superminis, like the minisupercomputers, cost between $100 000 and $1 million, but they run less than 1 megaflops. The workstations cost between $10 000 and $100 000 and are in the same range as the superminis for performance.



Computers typically fall far short of their peak performance when executing real programs, such as the Linpack benchmark. However, they usually perform better if a program is hand-coded to take maximum advantage of machine resources. Here, hand-coded programs for solving a system of linear equations—the problem tackled by Linpack—can run more than 10 times as fast as straightforward compiled code.

change results dramatically. In one case, a Hitachi S810-20 computer showed a performance variation of 25 percent on the original 14 Livermore Loops because of variations in the computing environments in which the benchmarks were run (compiler, operating system, and the like). Although it is important to measure the entire system—hardware, system software, and user software—it is most important to separate system components and identify the ones that are being measured by any given test. A benchmarking error of this nature appeared in a recent paper by Kirk Jordan of Exxon Research and Engineering in Florham Park, N.J. The paper combined estimated extrapolations and measured results; compared measurements on mature systems with studies using an experimental compiler; and compared optimized codes with unoptimized. These deficiencies in approach were noted in the paper; their presence should have precluded making conclusions. [See "Performance Comparison of Large-Scale Scientific Computers," *IEEE Computer*, March 1987.]

Another complication in benchmarking is the selection of the workload measure. Benchmarking studies often use the unweighted arithmetic mean to compare computer systems, but this practice tends to overemphasize the fastest parts of a series of computations. A better measure is the harmonic mean, which weights each piece of the series according to its frequency but inversely according to its execution rate. The harmonic mean acknowledges that the faster a problem executes, the smaller the portion of total time it will consume. The performances of the Cray X-MP, the Cray-2, and the IBM 3090 on Livermore Loops for vectors of length 90 (a medium length), as reported by McMahon at Livermore, furnish a simple example of the magnitude of difference between the arithmetic and harmonic means. The Cray X-MP-1 had an arithmetic mean of 52.6 megaflops and a harmonic mean of 13.9. For the Cray-2, the figures were 32.7 and 5.5, respectively; for the IBM 3090-180, they were 13.9 and 7.7.

Computers must be friendly

The raison d'etre of supercomputers is to provide users with the speed and capacity for timely execution of otherwise intractable problems. Users should not be required to perform computational gymnastics to achieve the turnaround time of which a system is deemed capable. Adequate supporting tools such as editors, profilers, debuggers, optimizing compilers, and user-friendly operating systems play an important role as intermediaries between the architecture and the user. Programming puzzles that must be solved to enhance performance may be of interest to investigators experimenting with a computer system, but they are only a hindrance to those whose research depends on the computer as a tool.

When benchmark results are being analyzed, evaluators must keep in mind the difference between frequency of execution of various operations and the time taken to execute them. Test programs are frequently assessed in terms of both the number of instructions executed and the total time spent executing in the vector (or parallel) mode. It is tempting to regard each of these measurements as being the percentage of vectorization (or parallelization). However, the percentage of total instructions performed in high-speed mode is different from the percentage of total time spent in high-speed mode. Ignoring this difference can result in disparate predictions of the speedup to be gained by vectorization for a given system.

Consider an analogous exercise: predicting the degree of speedup that will be achieved on a 100-kilometer journey (the analog of a given computing task) by driving some part of the way at a very fast speed (executing in high-speed mode). Should the speedup be defined in terms of the distance traveled at high speed or in terms of the time traveled at high speed? It is at this point that confusion may set in, although the total time for the trip will not vary.

To address a specific example, a car that travels 80 km at 80 kilometers per hour and 20 km at 20 km/h will spend 50 percent of the time at high speed and 50 percent at low speed. (In terms of a computation, it might be tempting to say that it is 50 percent vectorized.) However, if the car is driven even faster on the 80-km stretch— at 160 km/h, say—then it will spend only 33⅓ percent of the time at the high speed (the apparent percentage of vectorization goes down). Another way of looking at the problem gives more intuitive results: consider only the distance traveled at the high and low speeds. In these terms, the car travels 80 percent of the distance at high speed and 20 percent at low speed (80 percent of the operations are vectorized). Traveling faster for

Evaluation goes beyond benchmarking

The value of a computer depends on the context in which it is used, and that context varies by application, by workload, and in terms of time. An evaluation that is valid for one site may not be good for another, and an evaluation that is valid at one time may not hold true just a short time later. Benchmarking is only one part of the computer performance evaluation process, which includes qualitative as well as quantitative considerations.

Qualitative evaluation focuses on design characteristics or potential flaws. The most basic consideration is whether the computer in question exists. New computers are often marketed while hardware is still in development—the so-called "paper tiger" phase. Developers may make impressive claims about performance that has been "simulated with conservative assumptions," but a new machine should be regarded with skepticism until it has been delivered to customers in significant quantities. Next comes the so-called white elephant phase, when a machine has been brought to market but its acceptance is uncertain. A computer in this juvenile phase usually lacks good software, and if it is withdrawn from the market purchasers may be left without support.

Compatibility is perhaps the strongest qualitative issue to be examined. The scope of compatibility can determine what software and peripherals are available, as well as the ease of sharing programs and files among organizations. Compatibility is not an all-or-nothing question; while a few computers may be bit-compatible—each capable of running the others' programs in machine-code form—almost the same capability may be achieved if the programs can be recompiled from source code. Any two given computers may or may not run identical operating systems; in some cases, different operating system software may run on identical computer hardware at two worksites. In that case, programs might require modifications even though the underlying hardware is compatible.

While compatibility is desirable to ease the transition to new hardware, it can also throttle innovation. On the other hand, without at least compatibility of language, compiler, and operating system, users may have to develop much of their own software from scratch, thus delaying the time when they start getting real results.

Even without looking at performance figures, evaluators can rule out certain classes of computer on the basis of their architectures, which may become obsolete as design advances are made. For example, in the high-performance arena, scalar computers, which process data items individually, have been made largely obsolete by vector machines, which can process sequences of data very efficiently, and single-processor designs are now yielding to multiprocessors.

Only after qualitative issues have been settled does quantitative evaluation play a part. Issues here include cost, productivity, reliability, and performance. Cost comprises not only the cost of acquisition but also installation, operation, training, application development, and maintenance. Productivity is usually defined in terms of cost-performance ratios, but productivity evaluation should also include "quality of life": interactive access, high-speed graphics, powerful editors and debuggers, and on-line documentation. Without those tools, it may not be possible to exploit the full power of a computer. Reliability is of concern when applications may take several days to execute in a batch processing system. The mean times to failure for current supercomputer hardware range from hundreds to thousands of hours, so the weight of this issue is somewhat less than in the early 1960s, when computers might fail every 30 minutes.

The last of the quantitative evaluations is computing speed. This performance evaluation, if done well, can:

- Aid designers of future architectures and systems by pointing out the strengths and weaknesses of current systems.
- Assist in the reasonable characterization of system capabilities for site-specific procurement needs.
- Promote development of software—applications programs and system programs—for the effective utilization of existing architectures.

—J.D., J.M., and J.W.

the 80-km portion of the trip would not change the percentage of vectorization, although at the higher vector speed the entire task would be expected to take less time.

A hierarchical approach to evaluation

Understanding the workload is the foundation for performance studies, at all stages of the computer life cycle—design, procurement, and operation. Once the workload is understood, evaluators can design a sequence of increasingly complex tests to investigate specific components of performance, culminating in tests that correlate directly with a given workload. In such a hierarchical approach, information gained at each stage can contribute to analysis at the next level. The effort expended to progress through the entire hierarchy is very rigorous, so evaluators may decide to stop at an intermediate level, on the grounds that the added information is not worth the expense. However, effective benchmarking must include certain basic steps [see "A minimalist approach using program kernels," p. 42].

At the lowest level of the hierarchy are programs that test the basic operations of a computer—addition, multiplication, and so forth. They should be simple enough to be analyzed completely. From measurements of the basic operations, evaluators can determine the costs of filling and emptying pipelines, using or bypassing cache memory, accessing standard memory, making conflicting accesses to the same memory bank, and accessing disks. For vector and parallel computers, these programs also let evaluators determine the vector lengths at which half the peak performance is attained, the lengths at which the break-even point for vector-scalar tradeoffs is reached, the asymptotic peak performance for vector operation, and the amount of work to be done between synchronization points so as to overcome the cost of various parallel processing overheads. Because programs that test basic operations tend to indicate peaks in performance rather than norms, they should be used only as an aid to understanding subsequent tests, not as an end in themselves.

At the second level of the benchmark hierarchy are program kernels—short excerpts from actual programs. Probably the most famous are the Livermore Loops and the Whetstones, the latter being used more frequently on minicomputers and superminis than on supercomputers. Since the Livermore Loops represent the computationally intense portions of actual programs at Lawrence Livermore National Laboratory, a machine's performance on the loops gives some indication of what can be expected of a supercomputer under working conditions. In general, such program kernels measure only the performance of the central processing unit. Rarely are they designed to measure an integrated system, and their extreme simplicity can create some misconceptions about performance on the entire workload. Another drawback of such relatively simple programs is that their individual performance statistics may vary across the entire range of potential system performance and will depend heavily on the sophistication of the system compiler.

Basic routines, the third level, are major building blocks extracted from a spectrum of applications. They should exercise both the central processing unit and I/O systems. An evaluation using basic routines would not, however, cover the interactions between routines that occur in actual programs. Moreover, if other routines consume any significant amount of time in a typical program, the basic routines may not paint a good picture of total system performance.

At this stage of the hierarchy, benchmark results may begin to have an impact on the end user, because they provide information not only on performance but also on program structure and other optimization considerations. The Linpack routines are a good example of code modification based on performance evaluation. Because the same routines have been executed on a wide spectrum of systems architectures, the original program code has been modified over the years to reflect advances in computer organization and design. In particular, techniques have progressed from vector solutions of systems of linear equations to solutions that mix matrix and vector calculations to block-based methods that use matrix calculations throughout—yielding up to a threefold improvement on the same machine.

At the fourth level of the evaluation hierarchy are stripped-down versions of major programs in the routine workload. They measure some of the interactions required between basic routines. On the other hand, they contain little of the pre- and postprocessing required by complete versions of programs, and they may or may not include interactions between the central processing unit requirements and the I/O requirements. Stripped-down programs are easier to manipulate and port from one system to another than are full-scale programs; if the full-scale programs are confidential or classified, reduced versions permit testing where it would otherwise be impossible.

One danger of benchmarking with abbreviated versions of programs is that some requirements for memory capacity and I/O performance are relaxed, and the stripped-down version may lose some of the original structure. This elimination can create havoc when testing a parallel processing system. For example, if a program is scaled to fit on a system with limited memory, its performance might differ completely from the performance to be expected if the original memory sizes had been maintained.

Testing full-scale programs—the fifth level of the evaluation hierarchy—gives the most accurate picture of system performance for typical problems, yet it too has drawbacks. Full-scale programs are difficult to port from system to system, and evaluations of them tend to reflect the difficulty of measuring both programming style and the effort expended to fit a program to a given architecture. Measurements of full programs should be made only after the results of simpler benchmarks are fully understood and the computing environment is carefully defined. Furthermore, benchmarking full applications must include full-size data sets

Checklist for benchmark comparisons

1. Are any changes to be made to the mathematical models or algorithms for any of the machines being compared? If so, are the changes equivalent for all of the systems tested?
2. If kernels are to be used, are any of them optimized for any of the machines tested?
3. If so, are the levels of optimization comparable?
4. Are the configurations of the different systems comparable?
5. Do the systems to be tested have the identical memory capacity, and does it correspond to the capacity of a working system?
6. Are the peripherals comparable?
7. Are the compilers comparable? Are compiler directives inserted in some programs to promote compiler vectorization and/or parallelization, but not in others?

A minimalist approach using program kernels

1. Conduct a workload characterization study; determine the problem classes that comprise the total workload and their corresponding workload fractions.
2. Select a subset of the programs in the total workload to represent classes of applications fairly; include programs from the various classes, and assign a weight to each program proportional to its representation in the total workload.
3. Designate portions (kernels) of the selected programs to represent the entire programs—the critical step in successful benchmarking.
4. Time the kernels on the system under test.
5. Compute the weighted harmonic mean of kernel execution rates.

Program kernels for benchmarking

Two collections of small programs often used for benchmarking are the Linpack benchmark and the Livermore Fortran Kernels, also known as the Livermore Loops. Both are drawn from real program code, rather than being synthetic routines designed for benchmarking only.

The Livermore Loops are 24 samples of floating-point computations, taken from diverse scientific applications. They were developed by researchers at Lawrence Livermore National Laboratory in Livermore, Calif., where it was infeasible to do benchmarks with programs that were part of the routine workload. Large-scale applications programs at Livermore may exceed 100 000 lines of Fortran, and have many hardware-specific routines, such as I/O, memory management, or graphics routines that complicate efforts to port them to new systems. In addition, most of the major Livermore software is classified material. Because the predictive validity of the benchmarks at Livermore is very important, researchers have made every effort to devise a collection that is representative of their workload. Good predictions of performance on the kinds of computing done at Livermore, however, do not imply that the loops will predict performance on other workloads.

The kernels are derived from real computations, but few are complete calculations; rather, they are segments of code that typify often-used Fortran programming constructs or that challenge the ability of a Fortran compiler to produce optimized machine code. The kernels represent the best and worst cases of common Fortran practice, to give a realistic performance range. All of the Livermore kernels can characterize performance as a function of vector length—small, medium, or large. Small vectors contain well under 100 elements, or fewer than the number of registers in a vector processor; medium vectors contain about 100 elements, roughly the same number as the registers of a vector processor; and large vectors contain far more elements—well over 100.

Unlike the Livermore Loops, the use of the Linpack benchmark for rating computers in terms of overall speed was, in some sense, an accident. The benchmark was originally designed to give users of the Linpack software package information on execution times for solving a sample system of linear equations. It first appeared as an appendix to the Linpack Users' Guide, containing data on the performance of 23 computers. Over the years other data have been added, and today the list includes 200 machines, ranging from personal computers to supercomputers.

The Linpack benchmark features two routines: the first for executing the decomposition of a matrix, and the second for solving the system of linear equations represented by the matrix, using the decomposition performed by the first routine. Most of the time is spent in the decomposition routine (on the order of n^3 floating-point operations for an n-by-n matrix). Once the matrix has been decomposed, finding the solution requires on the order of n^2 additional floating-point operations.

Most of the computing is done by a subroutine called Saxpy, which multiplies a scalar, α, by a vector, x, and adds the results to another vector, y. It is called approximately $n^2/2$ times by the decomposition routine and n times by the solver routine with vectors of varying length. Saxpy performs a few one-dimensional index operations and storage references in addition to the floating-point addition and multiplication.

The two higher-level Linpack routines use two-dimensional arrays, but the lower-level routines like Saxpy deal with only one dimension. The higher-level routines access the two-dimensional arrays by column, then pass the address of each column to Saxpy. The Saxpy routine treats the column by itself as a one-dimensional array. Since the indexing goes down a column of the two-dimensional array, the references to the one-dimensional array are sequential and have unit stride. The performance for stepping down a column is much better than when addressing across the columns of a two-dimensional array, because Fortran dictates that two-dimensional arrays be stored by column in memory. Accesses to consecutive elements of a column refer to consecutive memory locations, while accesses to consecutive elements of a row are separated by as many elements as there are in a column, requiring more arithmetic to locate them. *—J.D., J.M., and J.W.*

as well as complete program code; otherwise results will still be misleading.

At the sixth and highest level of evaluation, tests should include algorithmic techniques under development, to take into account future workload directions and novel approaches that may be found in algorithm and architecture design. It would be unwise to rule out a promising architectural design simply because current programs are tied closely to an existing system. Although compatibility is an important consideration when advancing from one computing generation to the next, substantial performance gains may at some point require the replacement of existing software. Results from testing developmental programs may be difficult to generalize; their value lies instead in indicating the potential of new supercomputer systems for which new programs may be created and new workloads developed.

To probe further

Listings of computer performance on the Livermore Loops and Linpack benchmarks are available in two technical reports: *The Livermore Fortran Kernels: a Computer Test of Numerical Performance Range*, by Frank McMahon, published in October 1986 by the Lawrence Livermore National Laboratory in Livermore, Calif., as technical report UCRL-53745; and *Performance of Various Computers Using Standard Linear Equations Software in a Fortran Environment*, by Jack Dongarra, published in April 1987 by Argonne National Laboratory, Argonne, Ill., as technical report MCS-TM-23.

Two papers that discuss the current status of supercomputer evaluation are: *An Agenda for Improved Evaluation of Supercomputer Performance*, by E.F. Infante et al., published in 1986 by the National Academy Press in Washington, D.C.; and "Supercomputer Performance Evaluation: Status and Directions," by Joanne Martin and Dieter Muller-Wichards, *Journal of Supercomputing*, May 1987.

About the authors

Jack Dongarra (A) is a computer scientist specializing in numerical algorithms in the math and computer science division of Argonne National Laboratories. He was involved in the development of the Linpack and Eispack matrix-solution packages. Dongarra received a B.S. in mathematics from Chicago State University in 1972, an M.S. in computer science from the Illinois Institute of Techology in 1973, and a Ph.D. in applied mathematics from the University of New Mexico in 1980.

Joanne L. Martin (A) is a member of the research staff at the IBM Thomas J. Watson Research Center in Yorktown Heights, N.Y. Her current interests include supercomputer performance evaluation and analysis of large-scale scientific programs. Martin is a member of the National Bureau of Standards ad hoc Committee on Benchmarking and is editor in chief of the *International Journal of Supercomputer Applications*. She received a Ph.D. in theoretical mathematics from Johns Hopkins University in 1981.

Jack Worlton (M), a Laboratory Fellow at Los Alamos National Laboratory, is also president of Worlton & Associates, a consulting firm in Los Alamos, N.M. ◆