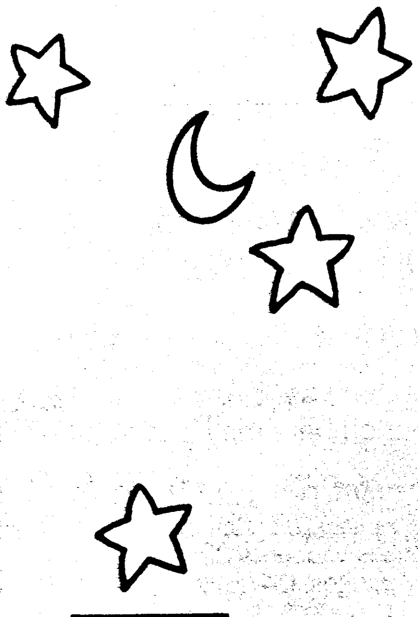
SPECIAL FEATURE

Accurate Unix Benchmarking: Art, Science, or Black Magic?

Confused about benchmarks? You'll feel more confident if you follow these guidelines.



David F. Hinnant

Benchmarking, though now more popular than ever before, remains one of the more tumultuous areas in the field of performance metrics. Accurate benchmarks help both users and system managers make crucial decisions about the systems they consider buying or upgrading. A PC user may want to find out how long it will take to run a certain type of program. A mainframe system manager may want to know how many users can be placed on a given machine without having those users banging on the door, complaining that the response time is terrible. Benchmarks, if properly designed, written, and performed can help both parties reach the right decision.

Frequently, however, users accept benchmark results without examining the raw data, a condition that may leave them wondering what to believe. These users may not be experienced enough in benchmarking technique to comfortably analyze the way the data was generated.

Benchmarks are only useful to users if they fully understand *what* is being measured. Users all too often assume that the person who ran the benchmark:

- *fully understood what was being benchmarked,* (As Eugene Miya commented in a 1985 Usenet message, even if the benchmark is run under ideal circumstances, veteran benchmarkers admit they are not always sure what the machine is doing in response to their benchmark.)
- *maintained adequate control of the environment under which the benchmark was run,* and
- *performed a "sanity check" to see that the generated benchmark data is within the boundaries of reason.*

These are not always good assumptions. It is quite possible that a benchmark designed to evaluate a particular function, if performed under the wrong conditions, can generate raw data that is totally meaningless. In essence, the principles behind the scientific method—both empirical and analytical—must be applied if benchmark results are to have any merit at all. When the scientific method is not followed, or when the benchmark data is misinterpreted, problems and confusion result.

When the data reported cannot be replicated, or when the logic behind the benchmark is unclear, users who have trouble with the benchmark tend to think they must be doing something wrong. They conclude without hesitation that benchmarking is an art and not a science. (In fact, some computer vendors engage in the "art" of creative benchmarking. They

0272-1732/88/1000-0064$01.00 © 1988 IEEE

carefully craft benchmark data so that the vendor never comes in second. In this context, the term "art" is, perhaps, more appropriate.)

Here, I offer a simple mathematical model of the interrelation of hardware and software to illustrate the necessity for knowledge of, and control over, the system under evaluation. In addition, I present a list of hardware and software variables in Unix systems that should be considered when running benchmarks and a set of guidelines to benchmarking technique. While I use Unix as the case study, most (if not all) of the unwritten rules generally apply to other operating system environments. Finally, I apply these guidelines to various benchmarks to illustrate the need for proper technique.

Benchmarks serve a number of purposes including identifying system bottlenecks, system tuning, application performance tuning, system configuration planning, and outright system comparisons. Although benchmarks can be loosely categorized into synthetic, natural, and hybrid types, I do not discuss the distinctions between these types. For the purposes of this article, a benchmark is any test—synthetic, natural, or hybrid—that attempts to measure some performance value of a computer.

Admittedly, this is a broad definition of benchmarking, but it serves the purpose here. Performance is an ill-defined term when used in the context of benchmarking. It can mean any one of a number of things, but it is commonly interpreted as interactive response time or task throughput rate, depending on the context in which it is used.

Background

Unfortunately, benchmarking has historically been poorly understood and often improperly performed. This is as true in the Unix community as any other. Most people take a stab at benchmarking by writing a pet benchmark program and running it on various machines when they get the chance. More often than not, the program doesn't measure what was intended because the benchmark is run under the wrong conditions, or because it is improperly designed. Worse yet, these benchmarks frequently get included in reports and purchasing recommendations. As a result, benchmarking receives undue reverence, with unquestioning acceptance of results.

Benchmarking in one form or another has been around for literally decades.[1] Only recently, however, has it received the stature it now commands when comparing everything from buses to disk drives to complete computer systems. And, nowhere is this reverence more apparent than in the Unix community. Vendors of microprocessors and entire systems alike produce a relentless barrage of conflicting benchmarking data to persuade customers that their Unix product outperforms the competition. Users are usually caught in the middle of this squabble and are left wondering how to determine which vendor's argument is more sound. This predicament is unfortunate because Unix is often implemented on state-of-the-art hardware for which little performance data is available and which needs unbiased, in-depth performance measurement the most.

The state of performance metrics

The basic problem with the current state of benchmarking is the lack of widespread knowledge on the subject; this fundamental problem is the seed for all others. Since benchmarking has been around for such a long time, why is the collective knowledge on benchmarking techniques not more widely disseminated? One reasonable theory is that the field of performance metrics, of which benchmarking is a major element, has been and still is isolated from other disciplines in computer science.[2]

Academic and theoretical interest in the performance metrics area is quite small in comparison with other areas in computer science. Few, if any, regularly published journals—nor columns in other journals—print performance metrics. No concise guide on running benchmarks exists to which users can refer. Few academic curricula include performance metrics (particularly in an undergraduate program). Special-interest groups seem to be the primary method of exchanging that information.

Whether this separation is evolutionary, intentionally self-imposed, or relegated by the mainstream of computer science, its effect remains detrimental to the field in general and in particular to the layman who is trying to use benchmarks to assist in a purchasing decision. Unfortunately, technical disciplines are no more immune to condescension than any other. More theoretical computer scientists consider performance modeling to be an implementation detail. This perspective appears to be extremely limited.

Many goals and benchmarking philosophies developed over 20 years ago are still important today, even though the computing systems that provoked this interest in benchmarking have long since been replaced. Not surprisingly, the most important question to the end user is still, How long will it take to process my job?[3] As much now as then, computer scientists try to determine the best way to benchmark a computer.[4] Why has such little progress been made over the years? Perhaps the isolation of those interested in benchmark theory is to blame here too. This insularity then seemingly hampers progress in the performance metrics community as well.[2] Indeed, it is fairly difficult to find and obtain good, current reference material on the subject.

The work that has been done (in both the theoretical and applied areas of performance metrics) has by and large been performed on older mainframes in which squeezing the most out of a costly machine was an important endeavor. Little work has been done on newer architectures, especially those in microprocessor-based Unix systems.

The forefront of benchmark technology continues to reside with a few special-interest groups and engineering groups of large corporations. Examples include the Association for Computing Machinery SIGMETRICS group and Computer Measurement Group, Inc. In the Unix community, the /usr/group (which proclaims itself "a commercially oriented Unix system users organization,"[5]) began a performance metrics working group. Ironically, no such counterpart exists in the more technically oriented Usenix Association. Perhaps commercial vendors have a vested interest in evaluating customer benchmarks for flaws and developing their own for marketing propaganda.

A primer in technique

It is difficult to arrive at an all-encompassing definition of benchmarking technique. I describe it here as an a priori logic. The technique centers on the premise that to predict accurately the performance of any given system (in the broader noncomputer sense of the word), one must first be able to describe what constitutes a system and which of the component parts of the system effectively affect each other. Users do not have to know in detail how the individual components of the system operate, nor how they relate *quantitatively*, but just that they affect each other *qualitatively*.

Unwritten Rules of Benchmarking

When running benchmarks or evaluating the results of benchmarks run by others, you need a list of benchmarking guidelines to which to refer. Here is a basic one.

1) Are apples and oranges being compared? Only machines of similar classification should be compared. It is of little value to compare a Cray to a PC. They are *expected* to be different. One doesn't normally compare an eagle to an emu, though they both are birds.

2) Are the configurations of the machines close? Only machines of similar configuration should be compared. Don't compare a PC with a 256-Kbyte RAM to another one with one megabyte if the benchmark to be run is memory intensive, unless you are investigating the effect of insufficient memory. For example, if the performance of microprocessors under Unix is being evaluated, it is invalid to compare an 80286 Xenix machine to a 68010 4.2 BSD machine because the operating system versions are not being held constant.

3) Is the machine doing other things besides running the benchmark? Benchmarks should only be run when other processes cannot affect them. Don't run a CPU-intensive benchmark while some other user is running one of the Unix text formatters *troff*(1) or *nroff*(1) (both quite CPU intensive). The emphasis here is on controlling the number of variables. Ideally, the machine should be idle except for the benchmarks being run.

4) Is the benchmark useful? Does the benchmark data generated have any real significance?

5) Is the benchmark being used in the proper context? A benchmark designed to evaluate one particular area should not be used to evaluate another, perhaps unrelated, area. Don't use the Dhrystone benchmark[6] to evaluate floating-point performance. A benchmark meant to compare compiler optimization should not be used to compare operating systems.

6) Is the system environment controlled? The environment surrounding the benchmark should be examined to make certain it is not affecting the benchmark results. For example, a benchmark that measures a disk's read throughput and is run several times (for averaging results) sequentially should flush the disk buffer cache between each run. A more common environmental consideration is normal Unix file system fragmentation. The list of free blocks in a virgin file system is very sequential. This means that writes to a new file will cause the drive heads typically to step in one direction and thus dramatically decrease the seek time and enhance benchmark performance. On a normal, scrambled

Once users realize that a set of components in a system can change its performance, they can vary one component while holding the others constant to see how that component quantitatively affects the system. This approach—solving for the unknown in the system, whatever it is—permeates benchmarking technique. Thus, we have a base rule: To generate valid benchmark data, understand the computer system. (See the accompanying box for other rules.)

When comparing an element of two different computer systems, users must keep the systems identical except for the component in question. Literally interpreted, this is obviously impractical and is just the theoretical best case as two different systems rarely differ in only one aspect. In fact, computer systems start with two basic elements, hardware and software, with an active system composed of the operating system software and the application load (possibly a benchmark program). However, the systems under investigation should be kept as close as reasonably possible.

This requirement is particularly true for the traditional, single-threaded, single-task benchmarks most users run. The importance of identical system configuration decreases somewhat as the sophistication of the benchmark increases. More sophisticated benchmarks measure a "user"—whatever that is. Then, the configuration becomes a component that is measured as a whole.

As an example, a compilation of a sample C program on a IBM PC AT under quiescent conditions may be faster than a compilation of the same program on a DEC VAX 11/750 under similar conditions. Does this necessarily mean the PC is faster than the VAX? Absolutely not. Not only are two entirely different classes of systems being (incorrectly!) compared (the VAX 11/750 is a multiuser minicomputer and the AT is a

file system the free list is not very sequential at all. This condition results in, for example, a write of one block of data from a file to reside in block 1234 and the next in block 5678.

Benchmarks that perform disk input and output should be run under similar conditions—either all on a virgin file system, or all on a well-used file system. The problem in using only pristine file systems is that the measured performance is unrepresentatively faster than that of a well-used file system.

7) Can enough data be gathered to be confident of the results? Can what the benchmark is attempting to measure really be measured? In other words there should be a good signal-to-noise ratio. For example, running a benchmark that completes in less than a second may not give representative data since the granularity of user time is usually given in tenths of a second. Here, a change in one tenth generates a 10-percent difference in the result. The real (elapsed) time reported by *time*(1) on many older Unix systems (versions 7 and 4.1 BSD) has a possible error of two seconds due to the one-second granularity of the Unix system clock.

Consider the following: *time*(1) reads the system time just before the next system clock tick and then activates *fork*(2)s and *exec*(2)s on the benchmark. Here, at most one second of error has been introduced. Next, *time*(1) reads the system clock just after a tick upon completion of its child process. Now, another second of error is introduced.

8) Is the benchmark properly designed and coded? The benchmark should be tested to make sure it is indeed doing what is desired. The integrity of the benchmark should be verified by some other means than supposing it works for all cases because it seems to work in this one. The return codes from system calls and library routines should always be checked.

9) Are the benchmark results reproducible? Can similar raw data be reproduced reliably and accurately?

10) Are the results from the benchmark reasonable? A little bit of common sense goes a long way. If the disk throughput is measured at 500 Kbits per second, and the disk subsystem transfer rate is only 50 Kbps, something is wrong somewhere. With systems using different software clock rates, timing routines that do not reconcile this fact in their calculations will emit surprising results.

11) Are all other variables held constant? As demonstrated earlier, when a specific component is being evaluated, all other variables must either be held constant or guaranteed not to affect the component under test.

12) Are statistical tools being properly applied? As we all know, raw data can be manipulated to prove almost anything if statistical tools are improperly used. The geometric mean should be used when generating a composite ranking of the results from normalized benchmark data.[7]

13) Is the conclusion reasonable? Both the raw data and the statistical analysis should support any conclusions drawn.

By varying only one component in the system, the performance of several others may be affected.

single-user super-microcomputer), they are being compared without regard for other elements in the systems. Moreover, the "constant" element used in the comparison (the compiler—here the application) differs dramatically between the VAX and the AT. The VAX C compiler (depending upon the operating system—another unknown) may perform superior optimization and emit higher quality code. The compiler probably performs more passes, may need more memory to run, and probably includes substantial debugging code for use with *adb*(1), *sdb*(1), or *dbx*(1) (if run in a Unix environment). Computer systems simply cannot be compared without regard for variables; we achieve more of a contrast than a comparison.

To further illustrate this point, I describe mathematically the possible relationships between the individual components of the three basic elements by developing a simple model. Assume there are *n* software components ($S_i = {}_{1,\ldots,n}$) and *m* hardware components ($H_j = {}_{1,\ldots,m}$). Each software component *may* be affected by other software components, and by each hardware component. Each hardware component *may* be affected by other hardware components. This is not an unreasonable model. I say *may affect* because in some cases the interaction between components is negligible, and in other cases no interaction occurs at all.

For example, an arithmetic coprocessor will favorably affect CPU performance (a hardware term in the equation) by offloading work to the coprocessor but will do little to affect disk drive throughput (another hardware term). (CPU refers to both central processing units and microprocessing units.) Hard-disk throughput is a function of both software (the driver in the kernel, buffering algorithms, sizes—all software terms) and hardware (controller transfer rate, track seek time, track caching).

In summary, the performance of an application on a system is a function of the performance of the components of the system in which hardware components may affect software components and other hardware components, and software components may affect other software components. In some cases software may actually affect hardware performance. An intelligent disk controller may actually work against the software driver, if the driver is improperly written, to decrease overall performance. This effect can be disregarded here, however, because the model is crude.

Thus, the first-order approximation on component interrelation is one in which software performance is a function of the hardware and of other software components, and hardware performance can be a function of other hardware components. These hardware and software performance parameters ultimately affect the performance of the application load. Mathematically, we express this as:

$$P_{ap} = S_{system}(S,H) + H_{system}(H)$$

where P_{ap} is the performance of the application, S_{system} is the performance of the system software, and H_{system} is the performance of the system hardware.

Since both the hardware and software functions are actually comprised of many component terms, we express this more exactly as:

$$P_{ap} = S_1(\sum_{i=2}^{i=n} a_{1_i} S_i, \sum_{j=1}^{j=m} b_{1_j} H_j) + S_2(S_1 + \sum_{i=3}^{i=n} a_{2_i} S_i, \sum_{j=1}^{j=m} b_{2_j} H_j)$$

$$+ S_3(\sum_{i=1}^{i=2} a_{3_i} S_i + \sum_{i=4}^{i=n} a_{3_i} S_i, \sum_{j=1}^{j=m} b_{3_j} H_j) + \cdots$$

$$+ H_1(\sum_{j=2}^{j=m} b_{1_j} H_j) + H_2(H_1 + \sum_{j=3}^{j=m} b_{2_j} H_j)$$

$$+ H_3(\sum_{j=1}^{j=2} b_{3_j} H_j + \sum_{j=4}^{j=m} b_{3_j} H_j) + \cdots$$

We can reduce this to:

$$P_{ap} = \sum_{k=1}^{k=n} a_k S_k(\sum_{i=1}^{i=k-1} a_i S_i + \sum_{i=k+1}^{i=n} a_i S_i, \sum_{j=1}^{j=m} b_j H_j)$$

$$+ \sum_{l=1}^{l=m} b_l H_l(\sum_{j=1}^{j=l-1} b_j H_j + \sum_{j=l+1}^{j=m} b_j H_j)$$

where P_{ap} is the performance of the application, S_i is the performance of a software component, H_j is the performance of a hardware component, and *a* and *b* are coefficients of interaction.

Granted this is a gross generalization, but the equation is sophisticated enough for the purposes of this discussion. (For a more accurate and exhaustive treatise, see Febish,[8] Grenlander,[9] or Silverman.[10]) However, this simple mathematical relation illustrates an important concept: By varying only one component in the system, the performance of several others may be affected. Therefore, the impact of a given component on the system as a whole cannot be measured accurately unless all other unknowns are either held constant or are known not to affect the performance of the other components in the system.

In any given benchmark, this interdependency relation simply cannot be ignored. For example, if an application software product is under evaluation, the hardware configuration and operating system configuration must be held as constant as possible so that the performance of the application is the only unknown. Thus, we determine performance without having to account for other changing conditions. (Other methods can be used to solve systems of equations with several unknowns. However, since the model is admittedly very crude, the value of this approach is dubious.) The converse is also true. When comparing a hardware element, benchmarkers should keep the remaining hardware and software terms (including the application, which in this case is probably a benchmark program) as similar as possible from run to run.

We referred to several distinct components in computer systems in passing. These and other components may have a direct impact on benchmark results. Benchmarkers must be able to determine what effect (if any) a component will have on a given benchmark. Here is a list of some of these basic hardware and software elements and their functions in a generic Unix system.

Hardware components. A nonexclusive list of variables in the hardware factor of the last equation include:

CPU type. Architectural differences between CPUs obviously affect benchmark performance. Some CPUs inherently perform certain operations better than others. Some CPUs have more general-purpose registers than others. Some have an internal 32-bit data bus, and some have a 16-bit bus. The external address and data bus width also varies widely. For example, the Intel 8088 has an 8-bit, external multiplexed data and address bus. Internally it has a 16-bit bus. Thus, the CPU requires two bus cycles to bring each operand for an ADD instruction. Because the internal and external buses differ in size, some people call the 8088 a 16-bit microprocessor, and some call it an 8-bit. Both are correct, but in different contexts. These and other architectural considerations can weigh heavily on benchmark differences. RISC, or reduced instruction-set computing, architectures, for example, inherently have different characteristics than traditional microprocessors.

Coprocessor support. When a floating-point coprocessor is available, benchmarks containing floating-point arithmetic behave differently—hopefully much faster. The performance of various arithmetic coprocessors also varies widely. In some cases floating-point hardware generates different results from those generated with floating-point software. A related area concerns the way and the degree of accuracy with which floating-point hardware operations perform.[11] Some coprocessors have 64-bit precision, and some have 80-bit. Some chips conform to the IEEE-754 floating-point standard.[12] Some floating-point chips may also assist in long integer arithmetic.

Multiprocessor support. Systems with more than one CPU are becoming more and more abundant. These systems typically use either peer-to-peer or master/slave CPU relationships. Any benchmark that attempts to measure a single processor must assure that the others are not available. However, in real-life situations since all processors are available, it makes more sense to evaluate the efficiency of the entire multiprocessor system. The performance of an ideal system with n processors is the sum of the performance of each processor from 1 to n. In practice, each CPU spends some cycles on the overhead associated with communicating with the other system CPUs, thereby reducing the aggregate number of cycles available for useful work. Multiprocessor systems are usually compared by the percentage increase each additional CPU provides, and the point at which adding additional CPUs ceases to improve performance.

Memory wait states. Machines that claim to run with zero states sometimes don't. Some marketing literature will mention the 64-Kbyte, zero-wait-state cache but not the 16 Mbytes of two-wait-state RAM.

Quantity of memory. Memory is very important if a multitasking benchmark is being run. Machines in which all the tasks run without being swapped to disk will outperform those that don't. In nonpaging systems, thrashing can occur if there is insufficient memory for the number of active processes in the system because the processes are constantly being swapped to make way for other processes. In paging systems, thrashing can occur when a very high page fault rate occurs due to a process faulting and having to free pages that will be needed again right away to satisfy the current page fault.

Clock rate. This rate varies widely with hardware, and numerically is misleading. Higher clock rates do not necessarily mean higher performance when two differing architectures are compared. CPUs with the slower clock rates may actually perform more useful work per clock or bus cycle than faster CPUs. Moreover, even fast CPUs may have to wait on memory due to wait states.

CPU cache. Cache size, construction, and availability vary from CPU to CPU. Small, tightly looping benchmarks may remain entirely in cache memory throughout the life of the benchmark. This aspect can make a wait-state-laden system (or a system with slow input and output) seem very fast. In general, small benchmarks should be avoided.

Bus structure. Bus address width, data width, bandwidth, transfer rate, and other variables vary widely from machine to machine. These, of course, ultimately affect the transfer capacity of everything from disks to memory.

Swap or page device. When too little swap space is provided in systems that only can swap, the system panics. In paging systems when too little paging area is provided, the system "thrashes" much like it does when it is out of memory. Swap space may also use a different soft interleaving factor, and if this is not optimal, performance suffers. If the system has several disk drives of dissimilar performance characteristics, is the swapping or paging device the faster device?

Software components. Let's consider the operating system and the application program. Just as important as the operating system itself is its configuration. Identical operating systems and identical underlying hardware may perform benchmarks differently if not tuned properly. In other words, there are several ways to influence the outcome of a benchmark since tuning can also be used to favor certain types of benchmarks.

Operating system version. Different versions of Unix implemented on identical hardware perform differently. For example, ports of Berkeley Unix (4.*x* BSD) support demand paging. Ports of System V may or may not do the same depending on the version (or release) implemented on the particular CPU. The file system also differs dramatically between System V, Releases 3 and 4.3 BSD, which are the two most important basic releases of Unix today. The differences between System V and 4.*x* BSD do not end here.

Some differences between operating systems transcend Unix versions. One of the more significant that affects benchmarks is the software clock rate. Many systems have the software clock interrupt the CPU 60 times a second; others interrupt 100 times a second. Some (to decrease overhead) use much lower values. Many time-oriented system calls and library routines base their concept of time on these clock "ticks," and the programs that use these routines must take this into account. For example, utilities use the *times*(2) system call return values in *n*th's of a second where *n* is the software clock rate.

Operating system utilities. Separate from kernel considerations, the performance of the so-called standard Unix utilities varies between versions—just as the syntax does. Commands such as *grep*(1), *ls*(1), *sort*(1), and even *cat*(1) are good examples of this variance.

Operating system configuration. There are a number of tunable parameters in the Unix kernel. Most are specified in configuration files at kernel compile time and are thus compiled into the kernel itself. The average benchmarker finds it difficult (though not impossible) to determine what these values really mean. The easiest way to be sure what the values are is to recompile the kernel. This compilation is possible even with binary-only systems as usually one file is compiled and then linked with the rest of the kernel, which is already in binary form. Here are some of the more important tunable parameters from a benchmarking perspective:

- The Unix file system's block-oriented interface buffers disk blocks. The number of disk blocks can dramatically affect benchmark performance. The more disk-intensive the benchmark, the more impact this factor has on performance (provided the benchmark isn't so large as to saturate memory so swapping or paging occurs). However, the more blocks in the cache, the less memory is available for user programs.

- The size of the *i*-node table determines how many files can be active at any one time in the system. The larger the table, the less memory is available. Likewise for the open file table in which one table entry exists for every open call: More entries imply less memory is available for user programs. Both of these values are secondary considerations compared with the number of disk buffers mentioned above. However, on a microcomputer with a limited quantity of memory this can become an important consideration. The tables must be of reasonable size to enable multiuser benchmarks to run as expected.

- Each process occupies a slot in the system process table. Some benchmarks use lots of processes and when this value is too low, the benchmarks report erroneous results because no more processes can be created. For example, the Unix C compiler, cc, prints "Try again" when this occurs, because it cannot fork or create a child process to continue the compilation. If too many benchmarks are run concurrently under the same login, the per-user process limit is reached and again forks fail. Because the per-user process limit is usually much lower than the total number of process table slots in the system, this limit is normally reached first. In general, benchmark programs that use system calls (most do) should check the return code of every call and report an abnormal return as a fatal error. System calls that seldom fail under normal operating conditions often fail when benchmarks are run.

- Different Unix versions cause different things to happen when a system runs out of swapping/paging space. Many systems panic and cease operation entirely. Some try to avoid system death by terminating the process needing to swap. Swap space and paging area should be guaranteed to be adequate for the benchmarks to be run.

- Other important tunable parameters include the number of *clists* (small buffers used mainly in terminal I/O buffering), and in System V systems, the number of semaphores, message queues, and shared-memory segments.

• When floating-point software is available, Unix usually supports it by one of two methods, kernel traps or libraries. Kernel traps are usually used when floating-point hardware is possible, but may or may not be available in a particular system. Kernel traps to floating-point routines may take somewhat longer to perform because of the overhead in taking the kernel trap. This overhead is avoided by using a library that is loaded in at link time by the Unix linker *ld*(1). The trade-off is that the code size of the resulting binary is substantially larger. Although not really a tunable kernel parameter, some Unix implementations do allow the system administrator to choose the method of software floating-point support.

Also of major concern is the accuracy of the floating-point routines.[11] Many systems that support floating-point hardware conforming to the IEEE standard also provide identical software floating-point routines so systems with either hardware or software floating-point operations generate indistinguishable results. Various Intel 80*X*86 implementations, for example, often accomplish this by emulating the 8087, 80287, or 80387 in software.

Some floating-point software routines generate results that conform to the IEEE standard format but do not always follow all IEEE rules when performing the operation. Here, the result is obtained faster, but at the cost of accuracy.

Use of operating system. Do all software packages being compared utilize Unix resources in the same way? The answer to this question may not be as obvious as it seems. Database management systems in particular may operate differently depending upon their internal design. For example, they may want their own disk partition on which to operate, bypassing the Unix file system altogether. They may use their own database functions or use the routines provided by the system, for example, the 4.*x* BSD *dbm*(3) routines.

Operating system environment. The operating system environment under which the benchmarks are run must be well known and tightly controlled. For example, depending on the component being evaluated, the benchmark may need to be run in either a normal fragmented file system or a well-organized virgin file system.

The operating system should also be in a steady-state condition. Just after booting, kernel data structures and buffer pools are in their most pristine state. Under normal system use, the linked lists controlling these structures and buffer pools become disordered and complex. Also, the disk buffer cache may still have unused blocks, and if a small disk I/O benchmark is run, the operating system may not have to flush any buffers at all. Therefore, benchmarks run just after a

The operating system environment must be well known and tightly controlled.

system is booted may not give accurate results, depending upon what is being evaluated.

Application elements. The application element can be any normal application program, a combination of programs, or a benchmark that is intended to evaluate some function of either the hardware or the software. Often this combination is deemed to represent a user load.

In Unix one special application element is the C compiler. Since most Unix benchmarks are written in C, and since most of the Unix kernel is as well, the efficiency of the code generated by the C compiler can affect not only benchmark performance but total operating system performance as well.

Implementations of the portable C compiler often sacrifice efficiency for portability. In other words, the portable C compiler may give misleading results if the results are interpreted as a system metric. For example, the portable C compiler, which is standard on nearly every Unix system, may compile a database benchmark so that it runs poorly. A highly optimized Pascal compiler may compile the equivalent code so that the benchmark runs much more efficiently. If a benchmark is being developed to evaluate various systems for a given application, the benchmark should be written in the same language and compiled by the same compiler that the application uses.

Now that the potential effect of varying one component of the system on the application has been shown, we examine other portions of benchmarking technique.

Observing the obvious. Because benchmarks are widely used to substantiate claims of product superiority, one would hope that vendors would have in-depth experience with benchmarking. This is not always the case. Vendors of software products often claim that their Unix product outperforms others running under Unix (a DBMS for lack of a better example). Given the hardware and software variable components and the previous examples, we now see that this claim can easily be made vague and meaningless.

The vendor may have benchmarked a package on the best performing machine it could find in the range of similar machines, and a competitor's on the worst. That is, a DBMS running on a "Megawonga 5" with a 16-MHz Motorola 68020 CPU, a 68881 math coprocessor, 4 Mbytes of zero-wait-state RAM, and using demand-paging virtual memory undoubtedly has a natural architectural advantage over a "Blitz 9000" with a 6-MHz Intel 80286 with 0.5 Mbytes of two-wait-state RAM with floating-point libraries. This outcome is true even though the "systems" may both be classified as super-microcomputers. The vendor may have referred to the machines only as Megawonga 5 and Blitz 9000. What would happen if the Blitz 9000 used a zero-wait-state, 16-Kbyte cache, an 80287 math coprocessor, and 80186 I/O processors?

The results could have very well been the exact opposite. Moreover, what if the normal Megawonga 5 came with an 8-MHz 68000 and 1 Mbyte of RAM? The results would very likely have been different here as well. To reiterate, the configuration of the machine must be taken into account when running the benchmark, and should be clearly reported in the results of the benchmark. It is essentially invalid to compare competing components—whether disk drives or application programs—unless they are run in similar hardware and software environments *or* unless you can be sure that the remainder of the environment has no effect on the component under test.

For example, comparing a totally CPU-bound, nonfloating-point application on a DEC VAX 11/780 to a microcomputer has some merit if one is contemplating purchasing a microcomputer to run this task. If the application uses floating-point operations and the VAX has floating-point hardware but the microcomputer doesn't, this is still a valid comparison because one may be interested in just how slow the application will be if run on the microcomputer. Comparing differing components beyond this is unreasonable because the expected performance differences are substantial. Application-oriented benchmarks are meant to compare similar machines. Using them to compare unlike machines will likely lead to invalid conclusions.

Personal computers are frequently compared in magazines, and often the benchmark results are quite meaningless. Perhaps it's nice to know that an IBM PC clone can format a disk faster than an Apple II, but of what practical use is this information? It's not surprising that the PC clone outperforms the Apple II, and not many people will rush out and purchase a clone because of this. How often are disks formatted with respect to total system usage? Here the invalidity of the comparison is obvious. In other cases it may not be quite so obvious. What *is* of practical use is comparing an IBM PC clone to a real IBM PC and perhaps other clones as well. Here it makes much more sense to say that "copying files on *X* is twice as fast as on *Y*."

To simulate multiple users, some benchmarks invoke multiple copies of the same test. These tests would more or less run concurrently. While this approach is reasonable at first glance, real users rarely do the same things at the same times. The results would be misleading since some resources would become saturated prematurely while others would be underutilized.

As a first check on validity when reviewing benchmarks performed by others, one need only ask if data describing the operating environment under which the benchmarks were run are presented in a concise and readable form. If information about the way the benchmarks were run is not present, it may indicate that sufficient care was not taken to control the environment when performing the benchmarks.

Even if proper benchmarking procedure is followed, mistakes can still occur. These mistakes often include human experimental error and statistical error. The two make a deadly combination. In any field, compiling statistical data can be a source of possible error. Benchmarking is no exception.[7] However, if proper scientific method is applied here, the problems can be avoided or at least minimized.

Examples

Since we've discussed proper benchmarking technique, it is now in order to offer examples of poor benchmarking technique.

Vendor benchmarks. As a general rule, benchmark results touted by vendors should be viewed with skepticism—particularly those results not produced by an impartial third party. Unfortunately, "bench-marketing" often replaces objective benchmarking. Here are several examples.

Two well-known microprocessor vendors have long been comparing their microprocessor families, and both companies have long proclaimed architectural and performance superiority. In 1985, both companies published reports using several independently developed benchmarks[6,13,14] to compare their microprocessors in a Unix environment. Using the same benchmarks, each company seemed to prove its product was superior in this environment. Clearly, both companies cannot be correct, and actually neither claim can be sustained because both reports violate many of the benchmarking rules just discussed.

Recently some vendors, of both operating system software alone and complete computer systems, have elevated the level of deception in benchmarks run on their systems by changing the way certain system functions behave. For example, Figure 1 shows a simple, popular benchmark that exercises system calls.

This benchmark can be, and has been, easily compromised by replacing the *getpid*(2) call with a library call by the same name that performs the actual system call only once ($i = 1$) and simply returns the same value for all subsequent calls (i = **2 to** i = LOOPS).

Such efforts at deception have become even more elegant. Because the Dhrystone is such a well-known and well-distributed user-mode, CPU-intensive benchmark, some compilers have been "hacked" to recognize the Dhrystone and to optimize for it. This is perhaps done by optimizing code fragments when certain variable names or constructs are encountered.

```
#define     LOOPS 25000

main()
{
    register int i, j;

    for (i = 0; i < LOOPS; i++)
        j = getpid();
}
```

Figure 1. Getpid benchmark.

These efforts defeat the objective of benchmarks entirely.

User benchmarks. Because of the state of performance metrics in the user community, it's just as easy for users with good intentions to generate invalid benchmarks or benchmark data.

Figure 2 shows the C source for a simple benchmark that reads the first 100 blocks of a file to evaluate sequential disk access. Although simple enough at first glance, there are several serious problems with this benchmark.

• Is it really measuring sequential disk access or is it measuring sequential *file* access? Because of normal file system disorder, the physical blocks for a_file are probably not consecutive. Therefore the disk heads may be moving back and forth as the benchmark reads (logical) blocks 1 through *n* sequentially. **Guidelines 4 and 5 were violated.** (Refer to the earlier box.)

• In Unix the kernel may attempt disk "read ahead" to anticipate the requests of the user program and bring the next few (logical) blocks of the file into the disk buffer cache. Under other operating systems this may or may not be the case. **Guideline 8 was violated.**

• If the benchmark is run in a pristine file system, the physical blocks associated with a_file may actually be very sequential so that the entire 100-block file may be read from disk in only a few disk accesses. **Guideline 8 was violated.**

• Most microcomputer Unix implementations have over 100 disk buffers. Thus, unless the buffer cache is flushed between invocations, benchmark data collected after the initial run will be misleading. **Guidelines 9 and 10 were violated.**

• Bufsiz is a relative quantity. On some systems it may be 512 bytes and on others it may be 1,024. Depending on other variables, including internal disk sector size, more work may occur when a 1,024-byte block is read than when a 512-byte block is read. **Guideline 6 was violated.**

• Some file systems use a logical disk block size of up to 8,192 bytes, thus satisfying many subsequent Bufsiz requests (usually 512 or 1,024 bytes) from the disk buffer cache after the driver has performed only one read operation. **Guideline 6 was violated.**

```
#include <stdio.h>
main()
{
    char buffer[BUFSIZ];
    register int i;
    int fildes;

    if ((fildes = open("a_file", 0)) < 0) {
        fprintf(stderr, "Cannot open the file.\n");
        exit(1);
    }

    for (i = 0; i < 100; i++)
        if (read(fildes, buffer, BUFSIZ) <= 0) {
            fprintf(stderr, "Error reading block %d.\n", i);
            exit(1);
        }
    exit(0);
}
```

Figure 2. Disk read benchmark.

```
/* define the granularity of your times(2) function (when used) */
/*#define HZ    50       /* times(2) returns 1/50 second (europe?) */
#define HZ     60       /* times(2) returns 1/60 second (most) */
/*#define HZ    100      /* times(2) returns 1/100 second (WECo) */
```

Figure 3. times(2) defines for the Dhrystone benchmark.

```
#define TIMES 50000

main()
/* The first way of doing things -- use a function call */
#ifdef EMPTY
{
    register unsigned int i, j;
    for (i=0; i < TIMES; i++)
      j = empty(i);
}

/* the empty function */
empty(k)
register unsigned int k;

{
    return(k);
}
#endif
#ifdef ASSIGN
/*  The second way of doing things -- without a function call */
{
    register unsigned int i, j;

    for (i = 0; i < TIMES; i++)
      j = i;
}
#endif
```

Figure 4. scall.c code fragment.

Benchmark developers are not immune from problems either. In particular, they need to take sufficient precautions to ensure that the users running the benchmark understand the benchmark and how to configure it to be run on various systems. An example appears in Figure 3, which displays a code fragment from a widely circulated C-language version of the popular Dhrystone benchmark.

Defining Hz to be 100 on a 60-Hz system would make the results substantially faster. Users must take care to include the correct defining construct, but how do they determine the proper Hertz value for a given system? A statement to reference users to the *times*(3) manual page for the system in question would enable them to readily determine the correct value. **Guideline 10 was violated.**

Lastly, Figure 4 shows a code fragment from *Byte* magazine's Unix benchmark suite that purports to measure function call overhead.[14] The benchmark is compiled and run with Assign defined and then with Empty defined. User times from these runs are subtracted, and the difference is supposed to be the function call overhead.

The function call overhead for most systems measured is less than two seconds with several systems having less than 0.5 seconds of overhead. User time reported by *time*(1) has a granularity of 0.1 seconds. In other words, a maximum possible random error of 0.1 seconds exists due to the output formatting of the Time command alone. Therefore a data point of 0.5 seconds has an uncertainty of 20 percent.

There will be other error factors as well that are not as easily calculated; in particular, process scheduling variability. Moreover, the code fragment provides no information as to the variance and standard deviation of the data. Therefore, these samples are hardly statistically significant. **Guidelines 7 and 12 were violated.**

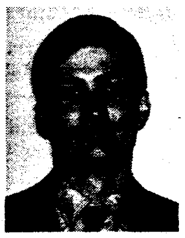
Accurate benchmarking seems to fall midway between science and art. As a science it involves the precise monitoring and measurement of computer systems. Yet, it is as much an art as is programming, and the goal of the artist is to create the perfect benchmark. Unfortunately, the perfect benchmark remains elusive. Whether developing benchmarks, running benchmarks, or examining the data produced by others, benchmarkers should ask themselves the questions posed by the unwritten rules of benchmarking.

While Unix was used as a case study here, most (if not all) of the unwritten rules generally apply to other operating system environments.

I hope that the perspective of benchmarking technique presented here helps dispel the aura of mystique that surrounds accurate benchmarking. I also hope that this perspective will stimulate further interest in the Unix user community and will enable all of us to take vendor claims with a grain of salt. ▩

References

1. O. Serlin, "MIPS Dhrystones and Other Tales," *Datamation,* June 1, 1986, p. 112.
2. D. Ferrari, "Considerations on the Insularity of Performance Evaluation," *IEEE Trans. Software Eng.*, Vol. SE-12, No. 6, CS Press, Los Alamitos, Calif., June 1986, pp. 678-683.
3. E.O. Joslin, "Application Benchmarks: The Key to Meaningful Computer Evaluations," *Proc. 20th ACM Nat'l Conf.*, 1965, pp. 27-37.
4. D. Ferrari, "Work-load Characterization and Selection in Computer Performance Measurement," *Computer,* Aug. 1972, pp. 18-24.
5. *CommUNIXations,* Vol. VI, No. 1, /usr/group, Santa Clara, Calif., p. 1.
6. R.P. Weicker, "Dhrystone: A Synthetic Systems Programming Benchmark," *Comm. ACM,* Vol. 27, No. 10, Oct., 1984, pp. 1013-1030.
7. P.J. Fleming and J.J. Wallace, "How Not to Lie with Statistics: The Correct Way to Summarize Benchmark Results," *Comm. ACM,* Vol. 29, No. 3, Mar. 1986, pp. 218-221.
8. G.J. Febish, "Experimental Software Physics," in *Experimental Computer Performance Evaluation,* D. Ferrari and M. Spadoni, eds., North-Holland, Amsterdam, 1981, pp. 33-55.
9. U. Grenlander, "An Introduction to Compumetrics," Brown University Report (NTIS AD-766 470), US Government Printing Office, Washington, D.C., July 1973.
10. H.F. Silverman, "Response Time Characterization of an Information Retrieval System," *IBM J. Research and Development,* Vol. 17, No. 5, Sept. 1973, pp. 394-403.
11. E.H. Spaffod and J.C. Flaspohler, "A Report on the Accuracy of Some Floating Point Math Functions on Selected Computers," Tech. Report GIT-ICS 85/06, Georgia Institute of Technology, Atlanta, Jan. 1986.
12. *ANSI/IEEE Standard 754-1985 for Binary Floating-Point Arithmetic,* CS Press, 1985.
13. H.J. Curnow and B.A. Wichmann, "A Synthetic Benchmark," *Computer J.,* Vol. 19, No. 1, Feb. 1976, pp. 43-49.
14. D.F. Hinnant, "Benchmarking Unix Systems," *Byte,* Vol. 9, No. 8, Aug. 1984, p. 132.



David F. Hinnant currently serves in a technical management position at Bell Northern Research. Previously, he was manager of Unix Standards and Performance for Northern Telecom, Inc., and served as manager of SCI System's Software Development Center. His research interests include performance metrics and cache design of multiprocessor systems.

Hinnant received his BS in physics from East Carolina University. He is a member of the ACM and the IEEE Computer Society and is co-chairperson of the /usr/group Technical Committee working group on performance metrics.

Questions about this article can be directed to the author at 2017 Hunterfield Lane, Raleigh, NC 27609; uucp {decvax, akgua}!mcnc!ecsvax!dfh.

Reader Interest Survey

Indicate your interest in this article by circling the appropriate number on the Reader Interest Card.

Low 162 **Medium** 163 **High** 164