The Design of Uncheatable Benchmarks Using Complexity Theory

(extended abstract)

Jin-Yi Cai Ajay Nerurkar Min-You Wu

Department of Computer Science,
State University of New York at Buffalo, Buffalo, NY 14260.
(716)645-3180x123
Email: {cai,apn,wu}@cs.buffalo.edu

1 Introduction

Benchmarks are heavily used in high performance computing to evaluate software or hardware systems. In the past, most researchers in the area of benchmark design have focused on "typical" programs and data sets for the system. The problem of making benchmarks resistant to tampering has been mostly ignored.

Verifying performance claims is of significant practical importance. However, for many existing benchmarks, it is rather easy to "improve" on benchmark results without actually improving the product, thus to "fake" superior performance. The industry is full of anecdotes of benchmark "cheating" of various degrees ranging from innocent mistakes to flagrant misuse. The main difficulty for most existing benchmarks is that its data sets are usually known in advance, thus various vendors could conceivably "optimize" their product just to turn in superior test results on these benchmarks.

Based on the tremendous progress made in recent years in theoretical computer science [ALMSS, Fre79, GGM86], especially with regard to our understanding of the power of randomization and interaction, it becomes possible to make a wide variety of existing benchmarks more accurate, resistant to tampering and more trustworthy. In making benchmarks more resistant to tampering, we use tools from complexity theory including one-way functions, trapdoor functions, and randomization.

We propose a Realistic Uncheatable Benchmark Suite (RUBS). This extends a general approach proposed by Cai, Lipton, Sedgewick and Yao in [CLSY93] (see also [AC94]). The uncheatable benchmarks satisfy the following conditions:

1. Computation performed is checked for correctness and accuracy.

2. There are no easy short-cuts on specific test instances to perform the required computation.

3. The tester of the benchmark has a computational advantage in judging the performance results made by a vendor of the system.

In order to avoid the problems with conventional benchmarks, where vendors could potentially "optimize" on specific test data, all test data in our benchmark suite are generated by some randomized process at run time. Since these test data are generated "on demand" at run time, the vendors cannot predict and thus cannot "optimize" on specific test data.

The approach is similar in spirit to the idea of public-key cryptography versus earlier approaches. For instance, in the RSA scheme [RSA78], the encryption method is public and randomized initially. The security guarantee rests on the computational intractability of factoring. In our schemes, the benchmark test routines are public and data is randomized at run time. With the help of secret hidden (trapdoor) information, correctness and performance can be independently tested and verified quickly. A related approach to verifying program correctness probabilistically has been proposed by Blum et al. [BK95].

We present two methods to obtain computational advantage in judging performance results. In the first approach, the tester gains a computational advantage via a hidden "secret," which is also randomly generated. With the secret information, the tester can know, or can easily compute, the correct answer in advance. This approach is called the Secret Key (SK) method.

Secondly, for many problems, verifying the result is easier than computing it. For example, verifying the result of a linear equation solver can be accomplished faster than any known algorithm which computes it. This asymmetry allows the tester an computational advantage over the vendor. This approach is called the Result Verification (RV) method.

Computational advantage achieved by using these methods will allow a tester to use a workstation to judge performance results of a supercomputer. Currently, our suite has benchmark programs for FFT,Gaussian elimination, matrix multiplication, and sorting. We are testing these benchmark programs for their stability and round-off errors. More performance results will be available for the final version of this paper. A web version will also be made available soon. In the remainder of this extended abstract, we will describe the first three benchmarks and also provide some test data.

2 The FFT benchmark

FFT is an $O(n \log n)$ algorithm which converts an input data set from the temporal/spatial domain to the frequency domain, and vice versa. It is an important algorithm with numerous applications in scientific, signal, and image processing. FFT is available in many benchmark sets, e.g., in NAS, SPEC-95, GENESIS, SPLASH-2, HPF benchmark, etc.

The Discrete Fourier Transform is defined as

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega^1 & \omega^2 & \cdots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)^2} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{n-1} \end{pmatrix},$$

where $\omega = e^{2\pi i/n}$. We developed an uncheatable benchmark based on the following idea. Suppose the vendor is given $x_0, x_1, \ldots, x_{n-1}$, and asked to perform FFT. The tester secretly generates a number r (say a root of unity of order higher than n). When the vendor returns the values y_j the tester computes $\sum_{j=0}^{n-1} y_j r^j$ in time $O(n)$. The trick is that, for the correctly computed values y_j,

$$\sum_{j=0}^{n-1} y_j r^j = \sum_{j=0}^{n-1}\sum_{k=0}^{n-1} \omega^{j\cdot k} x_k r^j = \sum_{k=0}^{n-1} x_k \sum_{j=0}^{n-1} (\omega^k r)^j.$$

Now $\sum_{j=0}^{n-1} (\omega^k r)^j$ is a geometric sum, and has the closed form $(r^n - 1)/(\omega^k r - 1)$ (or n if $r = w^{-k}$).

The sum $(r^n - 1)\sum_{k=0}^{n-1} x_k/(\omega^k r - 1)$ can be precomputed in $O(n)$, and the tester compares it with $\sum_{j=0}^{n-1} y_j r^j$ for the reported y_j. Using the fact that a non-zero polynomial of degree at most $n-1$ has less than n roots, we can show that this benchmark will almost certainly catch any erroneous Fourier transform y_j.

3 The Gaussian elimination benchmark

The current Gaussian elimination benchmark for linear system equation solvers is in widespread use, and is a typical example of a reasonably well crafted benchmark. Unfortunately it is also highly cheatable. We will first describe the current design of this benchmark. We will adopt the point of view of trying to deter a potentially dishonest user, who is trying to "optimize" on this test to cheat. To generate input data it uses a multiplicative congruential generator with a fixed seed and a fixed multiplier. A matrix M is generated entry by entry and a right hand side b is set to be the sum of the rows of the matrix M, to be treated as a column vector. Next the benchmark calls a subroutine to carry out the Gaussian elimination for M and then solves the system $Mx = b$. Usually, the benchmark does not check for the validity of the Gaussian elimination or vary its data set at all in its runs, thus, it is essentially computing a constant known in advance. In some versions of this benchmark, it does not check the result x, and in other versions it checks the result x by multiplying M and compares Mx with b. However, since the data set is fixed in advance, this check is really for an honest user, and it is no deterrent against tampering at all.

The intent of the pseudorandom number generator is to generate a random looking dense matrix with double precision floating point numbers. Some care is taken in the choice of the seed and the multiplier [Fis90]. However, since all the data are completely specified and known to the vendor, there is absolutely no security. After all the unique correct answer can be stored in advance. So in order to increase its viability as a general test for linear equation solvers, one should vary the seed and possibly the multiplier in each run.

Here is what we are doing to improve this benchmark. For essentially the same order of computation $O(n^2)$ at run time, the benchmark can randomly generate a "solution" x and multiply by A to form the right hand side $b = Ax$. With x hidden and later to be checked against the computed solution, we can save the current verification step of $O(n^2)$, by a simple look-up $O(n)$. The computational advantage is gained in view of the fact that solving a linear system takes atleast $O(n^{2.376})$ [CW90] using the most sophisticated method known (and $O(n^3)$ by more conventional methods.)

4 Matrix multiplication

Here, we describe the uncheatable benchmark for matrix multiplication using the SK method. We will first describe its theoretical underpinning. (See [AC94])

For simplicity, let's consider the following matrix powering problem: Given a large matrix M in floating point representation, and an integer k, compute the power M^{2^k}. A vendor makes a performance claim. The task is to verify such a claim. (Here the exponent is a power of two; thus it already takes into account the fast repeated squaring algorithm.) It is well known from Linear Algebra that every matrix M over the complex numbers has the following decomposition

$$M = T^{-1}JT,$$

where J is a Jordan matrix, and T is some non-singular matrix. A matrix is called a Jordan matrix, if it is a direct sum of the so-called Jordan blocks

$$J_\lambda = \begin{pmatrix} \lambda & 1 & 0 & \cdots & 0 \\ 0 & \lambda & 1 & \cdots & 0 \\ 0 & 0 & \lambda & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & \lambda \end{pmatrix}.$$

For any complex matrix M the Jordan normal form (JNF) J in the decomposition $M = T^{-1}JT$ is unique up to a permutation of the Jordan blocks. The matrix is diagonalizable if and only if the Jordan normal form J is a diagonal matrix, i.e., its constituent Jordan blocks are all one by one. Note that in general this is not the case, and moreover, distinct Jordan blocks may have identical eigenvalues. Such matrices are called derogatory matrices.

It is well established in numerical analysis, that finding eigenvalues and the Jordan normal form decomposition is computationally more costly than matrix multiplication. More importantly for our purposes, when the matrix is derogatory, i.e., having distinct Jordan blocks with identical eigenvalues, computing the Jordan normal form is numerically unstable, thus essentially impossible, by floating point arithmetic. (The reason behind this can be easily seen by the following example: Suppose $M = T^{-1}JT$ is 2 by 2 and has JNF $J = \begin{pmatrix} \lambda & 1 \\ 0 & \lambda \end{pmatrix}$, where λ is any complex number. Let $\tilde{M} = T^{-1}\tilde{J}T$ be a slightly perturbed matrix, where $\tilde{J} = \begin{pmatrix} \lambda' & 1 \\ 0 & \lambda'' \end{pmatrix}$, and both λ' and λ'' are close to λ, but $\lambda' \neq \lambda''$. Since now $\tilde{M}$ has unequal eigenvalues, the JNF for $\tilde{M}$ is not $\tilde{J}$ but $\begin{pmatrix} \lambda' & 0 \\ 0 & \lambda'' \end{pmatrix}$. In other words, the map from the space of matrices to its JNF is not a continuous map. Since numerical round-off errors are unavoidable, it is hopeless to compute it by floating point arithmetic.)

On the positive side we observe the following facts: Given T, T^{-1} and J, there is nothing unstable numerically to compute the product $T^{-1}JT$, even when the matrix is derogatory. Here comes one of the key observations: Once we know the Jordan normal form decomposition for a matrix $M = T^{-1}JT$, it is essentially a trivial matter to compute its high power, since as an inner automorphism of the matrix algebra, $M^i = T^{-1}J^iT$, for all $i \geq 0$, and due to its special form, the powers of a Jordan matrix can be essentially obtained by closed formulae, entry by entry.

Thus the idea of our uncheatable benchmark is the following: With some randomized process we will generate Jordan matrices, which consist of Jordan blocks of various sizes and eigenvalues, and we will purposefully assign some distinct blocks with identical eigenvalues. We then generate some random matrices T, and compute the product $M = T^{-1}JT$. We will provide the vendor with the matrix M, without revealing the Jordan normal form. For computational efficiency, we choose Householder matrices $T = I - 2\frac{uu^{\mathrm{T}}}{||u||^2}$, or a product of these Householder matrices. (Geometrically, a Householder matrix is a reflection in the hyperplane with the normal vector u, and it can be shown that every orthogonal matrix is a product of up to n reflections, thus if we choose u randomly and uniformly, the resulting product is a uniform distribution of the orthogonal matrices.) The advantage of using a Householder matrix is that the matrix is extremely simple to store and manipulate; it is its own inverse—thus we save the computation of the inverse. Security and efficiency of our benchmark derive from the fact that while $M^i = T^{-1}J^iT$ can be easily computed knowing the Jordan decomposition, it is numerically impossible to compute the Jordan decomposition efficiently given only the matrix M.

Another, perhaps conceptually simpler way, to check the result of matrix multiplication was given by Freivalds [Fre79]. He proved that if for a randomly generated vector r, $(AB)r = Cr$ then with high probability $AB = C$. Since $(AB)r = A(Br)$ this checking process can be carried out using matrix-vector multiplications alone and so can be done in time $O(n^2)$. This gives the tester a large computational advantage over the vendor. We will describe in the next subsection some test data from the implementation of this matrix multiplication benchmark. Due to space limitations we will not describe the sorting benchmark which also uses the SK method.

4.1 Test data

The following table shows the timing (in seconds) reported on a SPARCstation by a program written in C that multiplied two $N \times N$ matrices using the standard $O(n^3)$ algorithm and then checked the result using Freivald's method.

Operation	N=200	N=300	N=400	N=500
Multiplication	8.89	31.64	81.47	167.21
Checking	0.09	0.19	0.35	0.52

We have also implemented this benchmark in Java since our ultimate objective is to make this benchmark suite available on the Web. These were the timings (in seconds) reported by a typical run of our Java implementation.

Operation	N=100	N=200	N=300
Multiplication	9.53	74.23	249.46
Checking	0.31	0.88	1.90

References

[AC94] Sigal Ar and Jin-Yi Cai. Reliable Benchmarks Using Numerical Instability. In the Proceedings of the Fifth Annual *ACM-SIAM Symposium on Discrete Algorithms* (SODA), 34–43. Arlington, VA 1994.

[ALMSS] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy. Proof verification and hardness of approximation problems. In *Proc. 33rd IEEE Symposium on Foundations of Computer Science (FOCS)*, 1992, 14–23.

[BK95] M. Blum and S. Kannan. Designing programs that check their work. *Journal of the ACM*, 42, 1995.

[CLSY93] Jin-Yi Cai, Richard J. Lipton, Robert Sedgewick and Andrew Chi-Chih Yao. Towards Uncheatable Benchmarks. In the Proceedings of *The Structure in Complexity Theory Conference*, 1993, 2–11.

[CW90] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9:251–280, 1990.

[Fis90] G. S. Fishman. Multiplicative congruential random number generators with modulus 2**b: an exhaustive analysis for $b = 32$ and a partial analysis for $b = 48$. *Math. Comp.*, 189:331–344, 1990.

[Fre79] R. Freivalds. Fast probabilistic algorithms. *Lecture Notes in Computer Science*, 74:57–69, Springer-Verlag, 1979.

[GGM86] O. Goldreich, S. Goldwasser, and S. Micali. How to construct random functions. *Journal of the ACM*, 33:792–807, 1986.

[LA92] LAPACK User's Guide. SIAM, 1992.

[RSA78] R. Rivest, A. Shamir and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21 (2):120–126, 1978.

[Yao82] A. Yao. Theory and Applications of Trapdoor Functions. In *Proc. 23rd IEEE Symposium on Foundations of Computer Science (FOCS)*, 1982, 80–91.