VIEWPOINT

The Trouble with Benchmarks, Revisited

Failure to acknowledge factors causing variability

By Milton Barber
Apogee Software, Inc., Campbell, CA

The author is the president of Apogee Software, an independent software vendor specializing in optimizing compilers for RISC microprocessors.

The RISC workstation marketplace is strongly performance-oriented, so benchmark results are very important. Unfortunately, published benchmark results are frequently misinterpreted. This article is a response to some comments that have appeared in *Microprocessor Report* and other places in the computer trade press concerning benchmarks (for example, the editorials in µPR 12/18/91, p.3, and µPR 5/27/92 p.3).

An analogy from chemistry illustrates the main problem. Suppose you are a chemist interested in measuring a certain effect, let's say the resulting concentration of a certain desirable chemical X after a certain reaction occurs. In addition, suppose that it is obvious that this particular effect is strongly influenced by at least three independent variables, let's say the starting concentration of reagent A, the starting concentration of reagent B, and the concentration of catalyst C. Now let's suppose that you do some experiments measuring the concentration of X as a function of the concentrations of A and C (making no attempt to control the concentration of B), and you submit a paper describing these results to a reputable chemical journal.

In chemistry, you would rightly expect not only rejection of this paper, but also derision from any colleague who heard about it. In computer benchmarking however, it seems to be the norm to let the same kind of analysis slide by without much comment. In fact, the situation is worse than that. Let me try a second example. Suppose you deliberately manipulate the concentration of reagent B to enhance the apparent effect of the concentration of reagent A on the resulting concentration of chemical X. Then you start up a company to manufacture and sell reagent A, citing your experiments in your marketing literature!

Again, in chemistry, this situation would result in rejection and derision and probably commercial failure. However, if the rules of the computer industry were applied, the results would be different. Most people would accept the stated results without asking too many questions, so you would sell a lot of reagent A; a few writers would ask pertinent questions, but not very many people would pay attention; and a few writers would smell something fishy but would misunderstand the situation and would call for banning the sale of reagent B.

CPU-bound benchmark numbers are relative to at least four important things: (1) the hardware, (2) the operating system, (3) the compiler used to compile the benchmark program, and (4) the benchmark program itself. For workstations, the hardware and the operating system are not independent variables, so we can conveniently lump these two together under the term "system," leaving three independent variables: system, compiler, and benchmark program.

The fact that benchmark numbers depend on all three of these factors seems to be easy to forget. As a result, many people ascribe a meaning to benchmark numbers which they don't possess. There is also some criticism of the way benchmark numbers are being used, but this criticism doesn't seem to have much effect, and some of the criticism is misplaced.

The Meaning of MIPS

Let me give an example. A system vendor recently compiled the Dhrystone 1.1 benchmark using Apogee's C compiler, and the result was divided by 1757 to yield a "MIPS," or "millions of instructions per second" number. (About 1985, a group at Intergraph applied a pcc-based compiler to Dhrystone 1.1 on a VAX 11/780 and got 1757 Dhrystones. Since the 780 is regarded as a 1 MIPS machine, division of a measured Dhrystone result by 1757 to get "MIPS" is now a widespread industry practice.) This MIPS number was then announced to the world, just as if it had some meaning attached to it. Well, let's not be too harsh. Every measured number presumably has *some* meaning attached; let's see if we can discover the meaning of this number.

Dhrystone 1.1 is a well known synthetic benchmark program about 300 lines long. It contains a small main program and 11 short subroutines. It was constructed to contain "typical" frequencies of various common operations, including a certain frequency of arithmetic operations, loop control operations, subroutine call operations, parameter references, etc. The program does not read any input data, working instead from data constants in the program.

When the system vendor ran the tests, they naturally turned the compiler optimization up. Thus,

the first thing the compiler did was inline all of the subroutines, thereby eliminating all subroutine calls and all parameter references. Then it noticed that there were many references to constants, which were propagated to their uses. Then it noticed that the loop bounds were constants, so it unrolled the loops, eliminating the loop control code. This process continued through a number of further steps, including placing many of the program variables in registers.

When Dhrystone 1.1 is compiled with this compiler on this system using only local optimization, 680 instructions/iteration are executed. When variables are placed in registers, 461 instructions/iteration are executed. When "classical" global optimizations are turned on, 407 instructions/iteration are executed. When inlining is used and full optimization is turned on, 297 instructions/iteration are executed. (And, if a compiler that went to the theoretical limit on compile-time evaluation could be found, 0 instructions/iteration would be executed).

I believe the power of the circa-1985 VAX pcc-based compiler was roughly that of the compiler mode used to produce either the 680 or 461 values, depending on whether the "non-register" or "register" version of Dhrystone was used.

Thus we might more fairly label this number obtained by the system vendor as "The ratio of the time it took the VAX 780 to execute the large number of instructions that results from applying a circa-1985 pcc-based compiler to Dhrystone 1.1, to the time it took my system to execute the small number of instructions that results from applying a modern optimizing compiler to the same program."

It should be fairly clear from this discussion that the computation and publication of a "Dhrystone MIPS" in this fashion is absurd. By turning the optimization knob on the compiler, a system vendor can dial up essentially any desired MIPS number, possibly all the way up to a MIPS number many times greater than the peak maximum issue rate of the processor! This is clearly a misuse of a benchmark, but just exactly what is the misuse involved here? I suspect most readers would respond with one or more of the following points:

- The fault lies with Dhrystone. It is unrepresentative of real applications, and should not be used as a benchmark. We should stick with SPEC.
- The fault lies with permitting those kinds of optimization to Dhrystone. We should define just what optimizations are permitted, as is done, for example, in the paper defining Dhrystone 2.
- The fault lies with system vendors for mislabeling and hyping this particular number.
- The fault lies with the compilers. These sorts of optimization are cheating, and don't have anything to do with real applications.

Understanding Optimizing Compilers

In order to sort out these various issues, I would like to develop some ideas about optimizing compilers. An optimizing compiler is a large, complicated object, and like most such objects, there are several useful ways to view it. One good viewpoint is to think in terms of individual optimizations; i.e., different individual transformations that a compiler could apply to a program that might make it run faster. Depending on how finely you wish to construct a classification scheme for optimizations, there are at least hundreds, and maybe thousands, of different optimizations that can be applied by modern optimizing compilers. These optimizations have varying properties, including:

- *Applicability*—Optimizations vary widely in the frequency with which they may be applied. The program situations permitting some optimizations occur very frequently, while others occur only rarely. For most optimizations, this frequency depends strongly on the type of application and on the programming language used.
- *Utility*—Optimizations vary widely in their utility, i.e., the degree to which they speed things up. Utility usually depends strongly on the target architecture and on the program context.
- *Cost*—Optimizations vary widely in the cost of performing them. Some optimizations are simple and easy to perform, others require time-consuming analyses and complex program changes.

Based on this viewpoint of optimizations as individual classifiable objects, we can make four observations about optimizing compilers and their use. First, if we think of optimizer design, we can say the job of designing an optimizing compiler is one of evaluating the cost-effectiveness of each optimization (depending on its applicability, utility, and cost), and designing good algorithms to perform as many cost-effective optimizations as possible.

Second, if we think of programs to be compiled, we can say that different programs will display differing degrees of sensitivity to various optimizations. Some optimizations will speed a program up dramatically, some will speed it up a little, and some will either have no impact or will possibly even slow it down. For example, Dhrystone is very sensitive to certain optimizations, since these optimizations can speed it up dramatically.

Real-life programs display a tremendous diversity and a very wide range of sensitivities to various optimizations. This creates much of the challenge in designing

an optimizing compiler. It also creates a challenge in using an optimizing compiler to get good program performance. Some people think they can feed their program to an optimizing compiler using a few simple flags to turn on all the optimizations, and thereby expect the best program performance. This idea is an illusion. Using existing technology, this approach works only with trivial programs (like Dhrystone!). With non-trivial programs, the only way to achieve the best performance is to work with the program, applying a combination of analysis and experimentation to discover which optimizations are effective on which parts of the program, and which compiler flag settings enable the appropriate optimizations.

Third, optimization technology is not static—new optimizations are constantly being discovered and old optimizations are constantly being improved. It is often said that microprocessor performance is going up at about 5% per month. But this performance is being measured with compilers that are advancing rapidly, right along with the hardware technology. Some part of that 5% per month, maybe as much as 1% per month, is due to advances in compilers.

Fourth, optimization technology is not uniform; currently available optimizing compilers display widely varying architectures. For microprocessor design, the readers of this newsletter are familiar with such important differences as RISC vs. CISC, pipelined vs. non-pipelined, and superscalar vs. superpipelined. For compiler design, there are architectural differences as deep, as complex, and as important as any of these microprocessor differences. These compiler differences mean that strongly differing sets of optimizations are likely to be applied to any given program when it is run through different optimizing compilers. Such architectural differences among compilers can readily cause performance differences of 50% or more. (If you would like some visual verification of this diversity, flip through a set of SPEC data sheets and notice how the "profile" of SPEC ratios varies from compiler to compiler.)

Putting Benchmarks in Perspective

Now, what might be concluded about the interpretation of benchmark numbers from these various observations? Most importantly, remember that benchmark numbers are relative to the compiler and that there are important architectural differences among current optimizing compilers. There are many people who would look at some kind of a performance study which compared a CISC processor to a RISC processor and would label the study as "fundamentally flawed" because of the CISC/RISC differences, and yet would look at a SPEC comparison between two RISC systems made using compilers with different architectures and blithely assume that the comparison was a good measure of relative system performance. It is no such thing. The only reasonable statement you can make in this case is that the SPEC numbers provide a comparison between one system/compiler pair and another system/compiler pair. (In fact, without evidence to the contrary, it is equally valid to regard such SPEC number comparisons as indicating relative compiler performance instead of relative system performance.)

Since benchmark numbers depend on the compiler, it is important to keep in mind that compiler technology is advancing. This means you can expect the benchmark rating of any computer system to "inflate" over time, unless you enforce a rule against recompilation (which is not very realistic, since user programs can also be recompiled).

The most striking recent example of this inflation was the application of a cache blocking optimization to the matrix300 component of SPEC89. Because of its pattern of data references, this benchmark is a "cache-buster" on workstation-level processors. Its performance is dominated by cache and TLB misses, so it is insensitive to many ordinary optimizations. Kuck & Associates developed an optimization that is able to rearrange cache-busting loops which satisfy certain conditions so that the data are referenced in "blocks" which are far more likely to be present in the cache. This is an important optimization, because it enables a wide class of numeric problems to be solved efficiently on workstations for the first time.

Unfortunately, when this optimization became available on some workstations and not on others, the resulting quantum jump in SPECmark unsettled many people whose mindset was that SPECmark measures system performance, not system/compiler performance. Thus, instead of welcoming this new optimization like they would welcome a new microprocessor with a bigger cache, many people concluded that this was "cheating" on the part of the compiler.

Consider the charge that optimizing compilers are "cheating" or "cracking" the benchmarks (see, for example, μPR 5/27/92, p.3). These accusations might result from the mistaken mindset that benchmarks measure system performance, as above, or they might result from the feeling that the optimizations used on benchmarks are not equally applicable to real programs. Certainly it is possible to invent optimizations that are very effective for Dhrystone but have little utility elsewhere. This is not the real trouble with Dhrystone however, because Dhrystone is very sensitive to optimizations such as inlining, constant propagation, loop unrolling, compile-time evaluation, etc., which all have a high util-

ity and which are all widely used.

Next, look at SPEC. The accusation that optimizations which work for SPEC don't work for real programs is hard to understand, because SPEC is largely composed of real programs. I think the trouble lies with the fact that many people are just not aware of the degree of sensitivity to powerful modern optimizations that real programs display.

To illustrate this sensitivity, consider the 20 programs of SPEC92. What we call "local optimization" today is approximately what microprocessor "optimizing" compilers delivered a few years ago. So, if I use Apogee's compilers supplemented with the KAP preprocessors from Kuck & Associates, and I compare "local optimization" performance to "best" performance for each of these 20 programs, I get improvements that range from 15% to 327%. These break down as follows: 2 are above 300%, 3 are above 200%, 11 are above 100%, and 18 are above 50%. My experience is that sensitivities of this magnitude and with this degree of variation are typical of real CPU-bound programs.

Limiting the set of optimizations that may be used when benchmark programs are compiled is possible, but this may not achieve the desired goal. One approach is to state some limitations explicitly, as was done for Dhrystone 2. Another approach is to insist that some uniform optimization mode be used for all of a set of benchmarks, such as the SPEC set.

The important question is "What is it that you are trying to measure?" During program development, a user will generally limit the set of optimizations applied by the compiler. However, once development of a performance-critical program is done, the programmer will employ analysis and experimentation to arrive at the set of optimizations that gives the best performance. If the purpose of the benchmark is to discover "development" performance levels, then some sort of optimization limits make sense. However, if the purpose of the benchmark is to discover "best" performance levels, the same tools and techniques should be available for benchmarking as are available for performance-critical programs.

Consider the four responses to the misuse of Dhrystone that were proposed above. By now, answers to these points should be much clearer. When Dhrystone is treated as a performance-critical program, it is so sensitive to optimizations that its result mostly indicates the power of the optimizer, not the power of the system. The fault lies with treating Dhrystone this way and then labeling the result as MIPS, thereby suggesting that the result is strictly a system property.

Conclusions

What should be done? Here are some recommendations.

There is less chance that some sort of a "standard" compiler will emerge than there is that a "standard" RISC system will appear. Free-market forces will continue to operate for both compilers and for systems, so we are stuck with the existing relativity of CPU-bound benchmarks to both of these things. Some people will need to change their mindset. We should not ignore the concentration of "reagent B." We should not fall into the trap of thinking that CPU-bound benchmark results are exclusively measurements of system performance.

Compiler technology is not going to stand still, just as microprocessor technology is not going to stand still. So we are stuck with the fact that benchmark results for a given system will inflate with time. We should not try to ban the sale of "reagent B." We should greet the appearance of a new or improved optimization with the same attitude we would apply to the appearance of a new or improved microprocessor.

In response to the concern expressed by the Editor (see µPR 12/18/91, p.3) that many application programs are not recompiled when new compilers appear, I have a simple proposal that should accurately expose the benefits of recompilation. Suppose that a system vendor has published SPEC numbers for machine A, and at some later time publishes SPEC numbers for machine B plus a claim that machine B is upwards binary compatible with machine A. The vendor then should also publish the SPEC numbers that result from running, on machine B, the exact same binaries that were used to get the machine A measurements. Some rules to avoid a proliferation of published SPEC numbers might be necessary, but otherwise this proposal should be easy to implement.

When a performance measurement of some sort is stated, we should always relentlessly pursue the question "Just what was measured and what is the meaning of this number?" We should not lend credence to measurements where these answers are not forthcoming. This is a deeper point than just discovering things like system configuration and compiler version. Analysis and thought are required.

The widespread *laissez faire* acceptance of mislabeled numbers such as Dhrystone MIPS or Linpack MFLOPS should make computer industry professionals ashamed. People actually spend large amounts of money based on these things! We should pursue a vigorous policy of rejection and derision.♦