Evaluating language efficiency

Carl Burch, CSCI 360

December 11, 2004

I was generally disappointed with the efficiency of the programs students benchmarked for the project, which made it difficult to evaluate results. After looking over programs and papers, I decided to complete the assignment myself and generate an example solution. I drew some of the ideas for test problems from students' ideas, but I ended up implementing all of the programs from scratch.

Programmers say it all the time: "I don't think language X is a good choice because it's not fast enough." Wise programmers understand that computation speed is a minor programming consideration — far more often than many computing enthusiasts would like to admit, the major issue is programmer time (i.e., cost). Even when speed is important, the major bottleneck is often an issue over which the language is nearly irrelevant, such as rendering hardware, or disk speed, or virtual memory usage.

But, still, we can't deny that sometimes the speed of a language really does matter. And for that reason, we've put together a benchmark suite comparing the performance of five languages:

C Programmers widely acknowledge that good C programming is nearly unbeatable, so this is the baseline for the benchmarking. Representing C is GNU's *gcc* compiler (version 3.3.2), using the "`-O3`" option.

Java Most usage of Java bears a significant efficiency burden in terms of the virtual machine that interprets the compiled byte code, so most programmers understand that Java is quite a bit slower. They don't typically have a good handle on how much slower, but they suspect it's a fair amount. We used Sun's JDK 5.0 to represent Java, using the "`-O`" option for compiling.

Haskell This obscure language represents a peculiar corner of the programming language world called functional languages. One of the peculiar things about such languages is that they don't have variables. Although Haskell isn't widely known among programmers, those who know of functional languages suspect that they, too, are rather slow. We used the *ghc* Haskell compiler (version 6.2.2), using the "`-O`" option, which generates machine code (using C and *gcc* as an intermediate language).

Perl One of the most widely used scripting languages out there, Perl is an interpreted, which can affect its run-time dramatically. We used Larry Wall's *perl* program (version 5.8.3).

Python This upstart interpreted language is becoming a popular rival to Perl. We used Guido van Rossum's Python 2.3.3

It's not entirely fair to use a single language system as the representative for each language, since some may be optimized more completely than others. But in each case, we selected one of the most popular systems for executing those programs; it stands to reason that the most popular system would receive the greatest work on being able to execute programs in that language efficiently.

Conventional wisdom would hold that C would be fastest among these, that Java would be second but still several times slower, and the others would fall far away behind that, probably near the same spot generally.

We used a benchmark suite of five problems, representing different categories of problems. Because language efficiency is important primarily for computational problems, all five problems are computational in nature.

Harm This simple test represents a problem involving a lot of floating-point computations. In this, we determined which is the first harmonic number that is more than 15. The nth *harmonic number* is the sum of the reciprocals of the numbers from 1 to n; for example, the second harmonic number is 1.5 ($1 + \frac{1}{2}$), while the sixth is 2.45 ($1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \frac{1}{6}$). Despite a fairly quick start, they grow very slowly, and in fact they don't reach 15 until the 1,835,422nd harmonic number.

Sieve This test represents the field of problems involving heavy integer arithmetic. For it, we determine the number of primes up to 500,000, using Eratosthenes' sieve. This algorithm involves first creating a list of the numbers in question and then, starting with the first one (which will be 2), knocking out all the multiples of that, then knocking out all the multiples of the next one remaining, then of the next one remaining, until we reach the square root of 200,000.

As a functional language, Haskell can't provide a good implementation for this algorithm, so after some experimentation with different methods, we settled on an algorithm that would first find the primes up to the square root of 200,000 (using a recursive call) and then add on all the numbers beyond the square root for which none of those primes are a factor.

Pack In this problem, we have a set of 25 numbers chosen randomly from 0 and 1, and we are to determine the set with the largest sum but not exceeding 6. This is an NP-complete problem, and the algorithm used was an exhaustive search approach. The algorithm implemented used pruning techniques to reduce the size of the search.

Just Here, we want to determine the best way to introduce line breaks into a left-justified paragraph, namely a single paragraph containing the nearly 73,000 words of *Hamlet*. Though the "best" line breaks is a subjective question, we used the traditional technique of quantifying the "badness" of a set of line breaks by first computing for each line the size of the empty gap between the right edge of that line to the right edge of the paper, and then computing the sum of the squares of these gaps' sizes. Computing the "badness" can be done efficiently via a dynamic programming algorithm. (This program, by the way, was the most input-intensive of the bunch.)

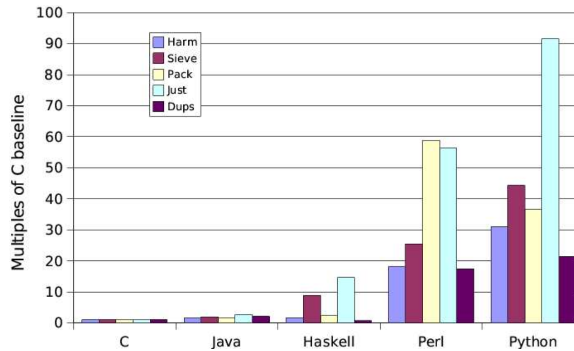
Dups This test represents programs that use string manipulation heavily. Each program took a file containing the first 4,000 words of *Hamlet* and counted how many distinct words it contained. The algorithm used to remove duplicate words began with the first word, crossing out all following duplicates; then if the second word was not already crossed out, it crossed out all the following duplicates of that one; and so on. Later, it counted how many words were left.

We implemented each of these problems in the five tested languages. We tried to implement them as efficiently as possible in each target language, but of course, we could probably trim off the times for all the languages with a more dedicated effort. The general rule here is that we tried not to spend a lot of our time on any single language, so that the total development time was spread relatively evenly.

To collect times, we ran each program seven times, discarded the slowest and fastest times, and averaged the remaining five together. All tests were performed on Linux computer (running Mandrake 10.0) containing a single low-power 1MHz VIA C3 processor. Figure 1 summarizes the results.

The conventional wisdom held in its broadest characteristics: C is indeed the fastest among those tested, with Java behind it, and the others behind both of those.

But there are also some surprises. If forced to guess how much slower Java is than C, many programmers would guess that it is several times — maybe a factor of five. For our tests, though, we found that it was only two. Of course, that's still not as good as one might hope: It takes twice as long to do the same thing.



problem	Harm	Sieve	Pack	Just	Dups	*Average*
raw C time (ms)	0.24	0.11	0.27	0.09	0.44	
C	1.00	1.00	1.00	1.00	1.00	*1.00*
Java	1.58	2.00	1.54	2.63	2.07	*1.97*
Haskell	1.71	8.91	2.43	14.67	0.68	*5.68*
Perl	18.12	25.47	58.81	56.48	17.48	*35.27*
Python	30.96	44.28	36.58	91.61	21.50	*44.99*

(All values below the line are in multiples of the C baseline.)

Figure 1: Results for all languages and problems.

Looking at the overall performance, Haskell seems to do significantly worse than Java. But when we look at the numbers harder, we can see that this was due largely to very poor performance on the the Sieve and Just problems, where the algorithm used didn't map onto the language well. On the other problems, it did quite well — and, indeed, Haskell was the only one that outperformed C on the Dups problem. It appears that Haskell is quite competitive when the algorithm can be expressed in functional terms well.

Far behind these both, we see the two interpreted languages, Perl and then Python. The slowness of these two languages derives from two factors: First, both run from an internal representation of the program, rather than compiled machine code. (The Java system tested does this, too, but its internal representation maps to machine code well, and besides the Java system may compile its internal representation into machine code before running it, removing any performance penalty for programs like those tested here.) The second source of slowness derives from the fact that both Perl and Python use dynamic typing, which adds a significant amount of overhead. In general, Perl appears to outperform Python, likely because the Perl interpreter — being both much older and much more widely used — has received more attention concerning efficiency. The only exception was the Pack problem, which incidentally was also the only program that used custom-defined functions heavily; looking at the awkward syntax for function parameters in Perl, it would not be surprising if that accounted for the difference.

So what can we conclude from this? When we have a problem that we know will be computational, we should probably stay away from interpreted languages like Perl and Python — that meshes with conventional wisdom. The choice of compiled language also makes a difference, though a much smaller one. Haskell and Java perform relatively the same, although Haskell is much more variable. Generally, since the difference between compilers of various languages isn't too large, we should choose the language that suits us best. The exception is for programs where computational performance for the programmed algorithms is critical; for these, C appears to the best of the five alternatives studied.