Is Java Fast Enough?

James W. Cooper

Toc: There is a whole body of work on high performance Java. See what we can learn from it.

Deck: some subversive thoughts on why we use OO programming techniques.

Everyone knows that OO programming is Good and procedural programming is Bad, right? It's like Batman vs. the Joker or Mr. Freeze. But what are the costs and advantages of OO programming? Do we know that it helps? Intellectually, OO programming seems cleaner and more elegant. In many cases it makes the main program flow much easier to read and modify. But is it really to everyone's advantage?

For example, why is it that Java has the reputation of being so slow? Is it because it is so OO that it is PC? Is it because the compilers and JVMs are still not that sophisticated? Well, one way to examine this question is to find out what people have been doing in the area of high performance Java programming. Most of these efforts have been in trying to write numerically intensive computation programs in Java.

Large, scientific computations have traditionally been written in FORTRAN and more recently in C, but Java for the most part has been ignored as too slow to be useful. But is it really that slow? I recently talked with Zoran Budimlic of the Center for High Performance Software at Rice University about his work in converting the LINPACK (Linear Algebra Package) library into Java and optimizing its performance[2, 3]. The LINPACK library is primarily a set of FORTRAN modules for various kinds matrix manipulation: multiplication, diagonalization, inversion and so forth for whole, triangular and packed matrices. Computational scientists in physics, chemistry and engineering have used these FORTRAN libraries for years and have invested a great deal of effort in optimizing the code and the compilers that produce the object modules.

There has not been much effort expended in optimizing Java compilers themselves for scientific computations because of some of the restrictions of Java: every array index must be checked and an exception thrown if the index is out of bounds, and because Java does not directly support multidimensional arrays. In fact, since Java represents matrices as arrays of arrays, there is no guarantee that every array will even be of the same length. You can see a discussion of ways of circumventing these issues in an article by IBM Research's Ninja Project group [1]. There is in fact a whole mailing list and conference, called JavaGrande, dealing with the problem of high performance Java.

The approach both Budimlic and the IBM group have taken is to write Java programs that transform the original Java into a new Java program that handles arrays more efficiently. Their programs optimize the access to the array elements and guarantee that for most cases, no exceptions can be thrown. The question is, how well do they do?

What I found interesting was the way Budimlic carried out these experiments. They started by coding a dozen or so of the major LINPACK routines in FORTRAN-like Java.

This means that everything was a static method: there were no objects instantiated, and the methods were called from a static main method. Then, they rewrote these routines in what they called "OO-Lite," where a Matrix class was created, but the manipulations within that class were much like you would write in FORTRAN. These included breaking out of loops and passing direct references to arrays between routines.

Finally, they wrote a completely object-oriented set of LINPACK classes which they called OwlPack (Objects Within Linear algebra PACKage). The performance differences between these three approaches is rather striking. I show the statistics for three such routines in Table1. Proportional results were also found on a Sun Ultra 5.

Table 1 – Times for matrix computations on a Pentium Pro

Matrix routine	**FORTRAN style**	**"Lite" OO**	**OO style**
dgesl solves matrix equation	0.865	0.967	4.488
dgedi computes determinant and inverse	1.428	1.677	33.288
dgefa Performs LU factorization	0.698	0.838	18.496

They also compared these results with those for a pure FORTRAN-90 program on the Sun, and found that in general the FORTRAN-style Java programs were only about 70-130% slower than pure, compiled FORTRAN. This is really a rather encouraging bit of news, since popular impressions are that Java is unconscionably slow.

However, the striking thing about Table 1 is how slow the complete OO programming solution is. While FORTRAN-style and Lite OO show only a few percent difference in performance, a really well-structured OO program can be on average, 10-20 times slower than that. This is a bit depressing.

Why does this occur? The "Lite" OO version was somewhat slower than the Fortran-style version because of extra indirections in the innermost loops of their routines, caused by having created Matrix objects. Here are some of Budimlic 's reasons for the slow performance of the well-written OO programs,

- Every number that is part of a computation is allocated on the heap as a separate object, with additional overhead for instantiation and garbage collection.
- Numbers that are elements of a matrix are scattered all over the heap, eliminating cache performance benefits that standard matrices can utilize.
- All numeric operations were done through method calls on corresponding objects, which incurred additional overhead, and dynamic dispatch to determine which method is being invoked.
- Each number takes up more memory as an object.

- Objects and method calls prevent or limit some common compiler optimizations, such as code motion, that can be performed in the FORTRAN and OO-Lite versions.

On the other hand, the OO versions of the code were smaller, easier to read and much easier to maintain, and this can be very important in a production environment.

So Do We Give Up on OO?

If it makes things10-20 times slower, why in the world would we write OO programs? Well, I have a number of answers to that. First, it is possible that making every number an object is a bit extreme and we may not normally do that except in the case of complex numbers, that are not part of the Java language in any other way.

But I did ask Budimlic what he thought, and his answers were really most reassuring. If you write in a FORTRAN-like style, you have the overhead of writing and maintaining code in this much more difficult style. You also will give up the fact that future optimizing compilers may do a much better job on good, OO programs than on ones you try to write to be fast yourself. Just to take a simple example, do you really believe that

```
x += 5;
```

is faster than

```
x = x + 5;
```

Do you believe that

```
z = (a > b) ? a : b;
```

is more efficient than

```
if (a > b )
  z = a;
else
  z = b;
```

You know that both of these are just syntactic conveniences (or obfuscations) and that any decent compiler will generate the same code either way.

Well, compilers really are a lot smarter than we are, and we can be sure that over time, Java compilers will benefit from the technology that we know has gone into the last 30 years of FORTRAN compiler development. Eventually Java compilers will catch up and there is no reason to abandon good OO coding styles because there are still more hurdles to jump over.

Optimizing Java

There are some optimizing tricks that Budimlic and his colleague Ken Kennedy came up with, however. The first of these is *class specialization*. If a class might handle one of several numeric types based on a number class (here they called it LNumber) that would lead to dynamic dispatching, you clone the class and make several simplified versions for each of the object types, such as LFloat, LDouble, and so forth.

The second major optimization is called *object inlining.* Objects that are contained within a particular class are replaced with their data, which can then be accessed directly. In doing this, they have to perform analysis to make sure that these objects are not changed within the class, so that this is a safe transformation to make. Of course, this has to be applied selectively, or the code would expand tremendously, so some analysis needs to be performed to choose which objects are most useful to inline.

By performing these optimizations by hand, while investigating building automatic Java translation tools, they achieved the very encouraging results in Table 2.

Table 2 – Performance of Optimized OO Java Programs

	FORTRAN style	OO style	Optimized OO	F90 Solaris native
Dgefa	0.698	18.496	0.796	0.194
Dgesl	0.865	4.488	1.162	0.307
Dgedi	1.428	33.288	1.693	0.364

In other words, the possible optimization of real OO code performs nearly the same as FORTRAN style procedural coding. In addition, it is now much closer to being competitive with the results of actual compiled FORTRAN. So again, compilers continue to improve and the simplicity and clarity of Java continues to be a persuasive reason for using it in all kinds of programming.

Almost Whole Program Compilation

Following the above experiment, they wrote code to implement these changes automatically in a system they called JaMake, and achieved nearly the same speedup. The methods they implemented to transform Java into a form that would remain pure Java but which would execute much more rapidly amounted in large part to identifying classes that were public (and therefore nominally extensible) and then making most of them private so they couldn't be extended and transforming them to allow interprocedural optimization which might undo the original class and method structure. For example, once the classes were private, they could perform object inlining and class specialization, making several versions of a class for different kind of variables.

If they carried out this transformation on the entire program, it would become less useful to the users, since they could no longer extend any of the classes at all. So they took the approach of allowing users to specify which classes they might want to extend and excluded them from the optimization. This approach also greatly reduced the optimization computation time, since they could also decide automatically that some program classes would be skipped.

There were also some classes in program code they worked on optimizing which were themselves derived classes. This made it difficult to make the base classes private and final so they could not be extended. In these cases, they replaced the derived classes with an auxiliary public *interface* to the base class. The details are a little complex to go into in this column, but the work is very clever and is described in a paper on Budimlic 's web site.

Final Subversive Thoughts

Another subversive thought. A few years ago, I was trying to persuade my academic friends that Java is a far more elegant teaching language than Pascal or C, and it ought to be introduced to students in introductory programming courses. Well, we won that one big time. Java is now the most widely taught computer language because it is OO, elegant and harder to screw up. On the other hand, as Budimlic noted, more students are now gaining a computer science education where the only language they want to use is Java. So it isn't just that we thought we'd try to optimize Java: we have no choice. Our new programmer's won't (or can't) program in any other language. We're a victim of Java's success!

One final note: I don't actually ever find that Java is the real bottleneck in any of my work. It is far more likely to by disk I/O, database access or network delays. The number of places where Java performance really needs to be tuned is quite small. In addition, as a general rule, you write the program in the best OO style you can and then look for tuning opportunities if you find that something is slow, rather than writing code in peculiar and unreadable ways to start with.

References

1. J.E. Moreira, *et. al.,*"The Ninja Project," *Communications of the ACM,* **44**(10), 102, (2001).
2. Z. Budimlic, K. Kennedy and J. Piper The Cost of Being Object-Oriented: A Preliminary Study, *Scientific Computing*, **7**(2), 87, 1999.
3. See also http://www.cs.rice.edu/~zoran/ for a complete bibliography.