# Why Java Will Always Be Slower than C++

*by Dejan Jelovic*

*"Java is high performance. By high performance we mean adequate. By adequate we mean slow."* - Mr. Bunny

Anybody that has ever used a non-trivial Java program or has programmed in Java knows that Java is slower than native programs written in C++. This is a fact of life, something that we accept when we use Java.

However, many folks would like to convince us that this is just a temporary condition. Java is not slow by design, they say. Instead, it is slow because today's JIT implementations are relatively young and don't do all the optimizations they could.

This is incorrect. No matter how good the JITs get, Java will *always* be slower than C++.

## The Idea

People who claim that Java can be as fast as C++ or even faster often base their opinion on the idea that more disciplined languages give the compiler more room for optimization. So, unless you are going to hand-optimize the whole program, the compiler will do a better job overall.

This is true. Fortran still kicks C++'s ass in numeric computing because it is more disciplined. With no fear of pointer aliasing the compiler can optimize better. The only way that C++ can rival the speed of Fortran is with a cleverly designed active library like Blitz++.

However, in order to achieve overall results like that, the language must be designed to give the compiler *room for optimization*. Unfortunately, Java was not designed that way. So no matter how smart the compilers get, Java will never approach the speed of C++.

## The Benchmarks

Perversely, the only area in which Java can be as fast as C++ is a typical benchmark. If you need to calculate Nth Fibonacci number or run Linpack, there is no reason why Java cannot be as fast as C++. As long as all the computation stays in one class and uses only primitive data types like int and double, the Java compiler is on equal footing with the C++ compiler.

## The Real World

The moment you start using objects in your program, Java looses the potential for optimization. This section lists some of the reasons why.

### 1. All Objects are Allocated on the Heap

Java only allocates primitive data types like int and double and object references on the stack. All objects are allocated on the heap.

For large objects which usually have identity semantics, this is not a handicap. C++ programmers will also allocate these objects on the heap. However, for small objects with value semantics, this is a major performance killer.

What small objects? For me these are iterators. I use a lot of them in my designs. Someone else may use complex numbers. A 3D programmer may use a vector or a point class. People dealing with time series data will use a time class. Anybody using these will definitely hate trading a zero-time stack allocation for a constant-time heap allocation. Put that in a loop and that becomes O (n) vs. zero. Add another loop and you get O (n^2) vs. again, zero.

### 2. Lots of Casts

With the advent of templates, good C++ programmers have been able to avoid casts almost completely in high-level programs. Unfortunately, Java doesn't have templates, so Java code is typically full of casts.

What does that mean for performance? Well, all casts in Java are dynamic casts, which are expensive. How expensive? Consider how you would implement a dynamic cast:

The fastest thing you could do is assign a number to each class and then have a matrix that tells if any two classes are related, and if they are, what is the offset that needs to be added to the pointer in order to make the cast. In that case, the pseudo-code for the cast would look something like this:

```
DestinationClass makeCast (Object o, Class destinationClass) {
    Class sourceClass = o.getClass (); // JIT compile-time
    int sourceClassId = sourceClass.getId (); // JIT compile-time

    int destinationId = destinationClass.getId ();

    int offset = ourTable [sourceClassId][destinationClassId];

    if (offset != ILLEGAL_OFFSET_VALUE) {
        return <object o adjusted for offset>;
    }
    else {
        throw new IllegalCastException ();
    }
}
```

Quite a lot of code, this little cast! And this here is a rosy picture - using a matrix to represent class relationships takes up a lot of memory and no sane compiler out there would do that. Instead, they will either use a map or walk the inheritance hierarchy - both of which will slow things down even further.

### 3. Increased Memory Use

Java programs use about double the memory of comparable C++ programs to store the data. There are three reasons for this:

1. Programs that utilize automatic garbage collection typically use about 50% more memory that programs that do manual memory management.
2. Many of the objects that would be allocated on stack in C++ will be allocated on the heap in Java.
3. Java objects will be larger, due to all objects having a virtual table plus support for synchronization primitives.

A larger memory footprint increases the probability that parts of the program will be swapped out to the disk. And swap file usage kills the speed like nothing else.

## 4. Lack of Control over Details

Java was intentionally designed to be a simple language. Many of the features available in C++ that give the programmer control over details were intentionally stripped away.

For example, in C++ one can implement schemes that improve the locality of reference. Or allocate and free many objects at once. Or play pointer tricks to make member access faster. Etc.

None of these schemes are available in Java.

## 5. No High-Level Optimizations

Programmers deal with high-level concepts. Unlike them, compilers deal exclusively with low-level ones. To a programmer, a class named Matrix represents a different high-level concept from a class named Vector. To a compiler, those names are only entries in the symbol table. What it cares about are the functions that those classes contain, and the statements inside those functions.

Now think about this: say you implement the function *exp (double x, double y)* that raises *x* to the exponent *y*. Can a compiler, just by looking at the statements in that function, figure out that *exp (exp (x, 2), 0.5)* can be optimized by simply replacing it with *x*? Of course not!

All the optimizations that a compiler can do are done at the statement level, and they are built into the compiler. So although the programmer might know that two functions are symmetric and cancel each other now, or that the order of some function calls is irrelevant in some place, unless the compiler can figure it out by looking at the statements, the optimization will not be done.

So, if a high-level optimization is to be done, there has to be a way for the programmer to specify the high-level optimization rules for the compiler.

No popular programming language/system does this today. At least not in the totally open sense, like what the Microsoft's Intentional Programming project promises. However, in C++ you can do template metaprogramming to implement optimizations that deal with high-level objects. Temporary elimination, partial evaluation, symmetric function call removal and other optimizations can be implemented using templates. Of course, not all high-level optimizations can be done this way. And implementing some of these things can be cumbersome. But a lot can be done, and people have implemented some snazzy libraries using these techniques.

Unfortunately, Java doesn't have any metaprogramming facilities, and thus high-level optimizations are not possible in Java.

## So...

Java, with the current language features, will never be as fast as C++. This pretty much means that it's not a sensible choice for high-performance software and the highly competitive COTS arena. But its small learning curve, its forgiveness, and its large standard library make it a good choice for some small and medium-sized in-house and custom-built software.

## Notes

**1.** James Gosling has proposed a number of language features that would help improve Java performance. You can find the text here. Unfortunately, the Java language has not changed for four years, so it doesn't seem like these will be implemented any time soon.

**2.** The most promising effort to bring generic types to Java is Generic Java. Unfortunately, GJ works by removing all type information when it compiles the program, so what the execution environment sees is the end is again the slow casts.

**3.** The Garbage Collection FAQ contains the information that garbage collections *is* slower than customized allocator (point 4 in the above text).

**4.** There is a paper that claims that Garbage Collection Can Be Faster than Stack Allocation. But the requirement is that there is seven times more physical memory than what the program actually uses. Plus, it describes a stop-and-copy collector and doesn't take concurrency into account. [Peter Drayton: *FWIW, this is an over-simplification of the paper, which provides a means of calculating what the cross-over point is, but doesn't claim that 7 is a universal cross-over point: it is merely the crossover point he derives using the sample inputs in the paper*.]

## Feedback

I received a lot of feedback about this article. Here are the typical comments, together with my answers:

> *"You forgot to mention that all methods in Java are virtual, because nobody is using the final keyword."*

The fact that people are not using the final keyword is not a problem with the language, but with the programmers using it. Also, virtual functions calls in general are not problematic because of the call overhead, but because of lost optimization opportunities. But since JITs know how to inline across virtual function boundaries, this is not a big deal.

> *Java can be faster than C++ because JITs can inline over virtual function boundaries.*

C++ can also be compiled using JITs. Check out the C++ compiler in .NET.

> *In the end, speed doesn't matter. Computers spend most of their time waiting on our input.*

Speed still maters. I still wait for my laptop to boot up. I wait for my compiler. I wait on Word when I have a long document.

I work in the financial markets industry. Sometimes I have to run a simulation over a huge data set. Speed matters in those cases.

> *It is possible for a JIT to allocate some objects on a stack.*

Sure. Some.

> *Your casting pseudo-code is naive. For classes a check can be made based on inheritance depth.*

First, that's only a tad faster than the matrix lookup.

Second, that works only for classes, which make up what percentage of casts? Low-level details are usually implemented through interfaces.

> *So we should all use assembly, ha!?*

No. We should all use languages that make sense for a given project. Java is great because it has a large standard library that makes many common tasks easy. It's more portable than any other popular language (but not 100% portable - different platforms fire events at different times and in different order). It has garbage collection that makes memory management simpler and some constructs like closures possible.

But, at the same time, Java, just like any other language, has some deficiencies. It has no support for types with value semantics. Its synchronization constructs are not efficient enough. Its standard library relies on checked exceptions which are evil because they push implementation details into interfaces. Its performance could be better. The math library has some annoying problems. Etc.

Are these deficiencies a big deal? It depends on what you are building. So know a few languages and pick the one that, together with the compiler and available libraries, makes sense for a given project.

Read more...

---