Translating Java to C++

Krishna Kunchithapadam
(`krishna@cs.wisc.edu`)
CS 838 Project Report (Fall 1996)

Abstract

Java is seen as an important applications-programming language of the future. However the performance of Java programs on the Java Virtual Machine (JVM) leaves much to be desired. In this report, I describe the implementation and performance of a Java-to-C++ translator for stand-alone Java application programs. My experiments show that it is possible to get between one to two orders of magnitude improvement in performance by translating Java sources to native machine code using C++ as an intermediate language. In some cases, it is also possible to achieve performance (via this translation) that is comparable to that of C++ code developed from scratch and carefully optimized by hand. I also comment on those features of Java that preclude certain kinds of performance optimizations that are possible in a language like C++.

1 Introduction

The Java programming language is, for both technical and non-technical reasons, an extremely popular language with scope for widespread use over the next few years. However, it is also an interpreted language (for reasons having to do with its security model and the network orientation of the language during its early development). Interpretation greatly slows down the execution of Java programs, even in situations where the advantages provided by the interpreter (the Java Virtual Machine [JVM]) are not needed. Further complicating the problem is the fact that the JVM interprets Java bytecodes, and security and portability concerns once again restrict the kinds of optimizations that can be specified purely in Java bytecode. In the battle between complete portability and security on the one hand, and high performance on the other, there are at least some instances where performance is the overriding concern.

In order to produce high-performance Java applications, it is necessary to come up with solutions that may be classified into two broad categories. The first category of solutions focuses on building hardware that directly implements the semantics of the JVM. The second category of solutions focuses on translating either Java source or Java bytecode into other languages (which can be either directly executed or further translated into native machine code). Solutions in the first category do not sacrifice either the portability or the security of Java, while solutions in the second category usually ignore any portability concerns (and may sometimes also relax Java's security model). In this project, I chose to translate Java source into C++ (and compile the C++ code into native machine code) as a means of improving the performance of Java applications.

The rest of the report is structured as follows. Section 2 provides some motivation for why translation of Java programs is needed at all in the first place, and why C++ is a good target language. Section 3 describes the structure of the translation tool and shows, by means of some small examples, the kinds of transformations that are performed in translating Java to C++. Section 4 describes the performance improvements that were obtained from the translation on both micro-benchmarks and large application programs. Section 5 discusses the lessons that were learned from the exercise, and particularly focuses on why it may not be possible in every instance to optimize Java code as well as semantically equivalent C++ code and the implications this has for programmers and software design. Section 6 closes with a summary of the project and a comparison of this work with other related work.

2 Why translate Java? Why target C++?

One of the much-touted advantages of Java is its simplicity and thus, its support for quick program development. However, unless programs are written many times and run once (as is often the case in academia), the performance of the application matters. It is also not a widely appreciated fact that the JVM causes excessive slowdown of programs (part of the reason why this is not so apparent is that until now, Java's primary use has been in the development of network applets where computation speed is rarely a concern).

As part of this study, I wrote Java versions of three C++ programs. The C++ code was carefully optimized both in terms of the algorithms and data-structures that were used. The Java code was a more or less direct translation of the C++ syntax into Java syntax. Given the straight-forward nature of the C++ applications and the translation process, it seems unlikely that the Java versions of these applications have any systematic negative performance bias.

The following table shows the performance of the C++ and Java versions of the three applications (the platform for all performance studies is a 200Mhz PentiumPro running the Solaris 2.5.1 operating system). The C++ code was compiled with GNU g++ (version 2.7.2, at the highest level of optimization). The Java programs were compiled (with `javac` and optimization enabled) and interpreted (with `java`) using the Java Developers Kit (JDK 1.0.2) from Sun Microsystems (and this will be the system referred to henceforth when speaking about the performance of Java).

TABLE 1. usertimes in seconds, 200Mhz PentiumPro

program	C++	Java	Ratio (Java/C++)
anagram	3.01s	324s	108
mandelbrot	1.42s	80.9s	57.0
dilemma	13.7s	322s	23.5

Anagram is a file IO and string-manipulation intensive application, *Mandelbrot* is a numerically intensive application, and *Dilemma* is an object-oriented application. As can be seen, the slowdowns of one to two orders of magnitude in the Java versions are serious enough to point to the need for translating Java programs into native code for faster execution.

The choice of C++ as the target language for the translation of Java programs is somewhat more subtle. The reasons are summarized in the following list:

- C++ is a very powerful language and Java is, for the most part, a proper subset of C++, especially in terms of its Object-Oriented semantics. Since the translation can be clearly specified, it is possible to avoid the pitfalls and complications (real or imaginary) of C++, and yet use the power of the language in suitable idioms.
- The translation of Java syntax to C++ is also fairly straight-forward (in fact, the similarity of Java's syntax to that of C and C++ is seen as a major advantage for Java in attracting the C/C++ community) and the additional features of Java syntax are easily implemented in C++.
- C++ compilers implement more than a decade worth of compiler research. Therefore, native machine code generated from C++ sources, in addition to directly executing on hardware, also have the potential of being highly optimized. And because of the widespread use and popularity of C++, few other languages are backed by a commensurate level of compiler support.

  Until Java compilation technology catches up with C++ compiler technology, C++ appears to be a good choice as a "target" intermediate language for translating Java programs into native machine executables.

3 Structure of the Translator

The Java-to-C++ translator (henceforth referred to simply as the "translator") is implemented in C++ and Perl5. The scanner/parser for Java sources is based on C++ code generated by the `flex`/`bison` tools. The parser produces a text-based representation of the *Abstract Syntax Tree* of the Java sources. The AST is then read in by a Perl5 program, subjected to analysis, and used to produce the "target" C++ code.

Each Java `class` in the original set of sources is translated into a pair of C++ class header and class implementation files. In some cases, for example Java `interface` classes with no `static` data, the implementation files need not be generated.

The resulting C++ sources and headers are compiled (with the GNU g++ compiler) and linked with a Java runtime library (which essentially implements the special semantics of Java pointers and arrays, and provides implementations of the native methods in the JVM) to produce a Unix executable `a.out`.

3.1 Limitations of the Translator

In order to simplify design and implementation, the translator makes the assumption that the Java source programs it translates are legal (and this restriction is easy to check and enforce by first attempting to compile all sources with a standard Java compiler). Since the translated C++ code cannot guarantee the safety of non-legal Java programs, this restriction is an important one.

The Java runtime library does not currently support the AWT and Applet libraries. The primary reason for this is that the AWT and the Applet libraries are almost completely implemented by native methods, and are subject to change in subsequent Java versions. The Applet interface is also associated with certain network security restrictions that are not relevant for Java applications. There is however no fundamental restriction in the translator that prevents support for these libraries.

The runtime library also does not support all the semantics of Java threads (partly because the semantics are not completely specified in version 1.0.2 of the JVM and are also subject to change in future). Once again, there is no fundamental restriction in the translator itself that prevents support for full-blown Java thread semantics.

3.2 Translation Examples

In this section, I will give a few examples of the translation of Java source to C++. These examples are not meant to be a complete or comprehensive description of the translator, but are meant to provide a feel for how Java source maps to C++. In particular, the examples should indicate that the translation of Java into C++ is fairly straight-forward and idiomatic.

3.2.1 The "Hello world" program

```
// Java source

class Hello {
    public static void main(String args[]) {
        System.out.println("hello world");
    }
}


// C++ source

#if !defined(__Hello_h_)
#define __Hello_h_

#include <java-base.h>
#include <java/lang/all>

class Hello : public Object {
    public: static void main(java_arr< java_ptr< String > > args) {
        System::out->println("hello world");
    }

    public: virtual ~Hello() {
    }
};
```

```
#endif /* !defined(__Hello_h_) */
```

In spite of its simplicity, the above example shows many of the important features of the translation process.

- The header file `<java-base.h>` provides the declarations and definitions for the C++ template classes `java_ptr<T>` and `java_arr<T>`, which respectively, implement the semantics (including garbage collection and safety) of Java pointers and Java arrays.
- The Java component selection operator '`.`' is translated into either a C++ scoping operator '`::`' or a C++ object dereference operator '`->`' as appropriate. Package and type name analysis in the translator indicates which C++ operator is required in each case.
- Every variable of a Java type (other than those of the primitive types) is encapsulated in a `java_ptr<T>`. Likewise, every array is encapsulated in a `java_arr<T>`.
- Every user-defined Java type inherits (in the C++ sense) from the Java type `Object`. This is not quite true of Java interfaces since they specify pure types (and thus avoids any possible multiple-inheritance problems in C++).
- The translator synthesizes various functions. For example, the above code has a synthesized virtual class destructor (which would be a placeholder for the invocation of `finalize` methods in Java types which define them).

3.2.2 Explicit Constructor Invocation

Java allows constructors to make explicit calls to other constructors (of the same class or of the `super` class) as the first statement in the constructor body. However, the constructors of C++ classes are not callable functions (except in the context of placement-`new`). In order to support explicit constructor invocation, the translator generates callable functions that implement the body of each constructor. The constructor itself then becomes a placeholder for invoking these constructor functions. Since C++ supports function name overloading, the Java semantics map in a clean manner into C++.

```
// Java source

class Foo {
    private int x_, y_;

    public Foo(int x, int y) {
        x_ = x;
        y_ = y;
    }

    public Foo() {
        this(0, 0); // explicit constructor invocation
    }
}


// C++ source
```

```
/* code elided */
class Foo : public Object {
    private: int x_, y_;

    public: void ctor_Foo(int x, int y) {
        x_ = x;
        y_ = y;
    }

    public: Foo(int x, int y) {
        ctor_Foo(x, y);
    }

    public: void ctor_Foo() {
        ctor_Foo(0, 0);
    }

    public: Foo() {
        ctor_Foo();
    }

    /* code elided */
};

/* code elided */
```

3.2.3 Java Pointer semantics

As mentioned in the "Hello World" example, the C++ template class `java_ptr<T>`, implements the semantics of Java pointers. This includes safe dereferencing and garbage collection. The following code snippet shows how the encapsulation of Java classes in C++ template classes can be used to implement garbage collection.

```
// Java source

class Complex {
    /* code elided */

    public Complex add(Complex z) {
        return new Complex(/* computation */);
    }

    public Complex multiply(Complex z) {
        return new Complex(/* computation */);
    }

    /* code elided */
}

    Complex z0 = new Complex(x, y);
    Complex z  = new Complex(0, 0);

    do {
```

```
        z = z.multiply(z).add(z0);
    } while (/* condition */);


// C++ source

class Complex : public Object {
    public: java_ptr< Complex > add(java_ptr< Complex > z) {
        return new Complex(/* computation */);
    }

    public: java_ptr< Complex > multiply(java_ptr< Complex > z) {
        return new Complex(/* computation */);
    }
};

    java_ptr< Complex > z0 = new Complex(x, y);
    java_ptr< Complex > z  = new Complex(0, 0);

    do {
        z = z->multiply(z)->add(z0);
    } while (/* condition */);
```

In the above example, the C++ code does not contain any explicit memory deallocation, although `Complex` values are being allocated each time a `multiply` or an `add` method is invoked. The overloaded assignment operator (`operator=`) for the `java_ptr<T>` class implements the deallocation based on reference counting. The overloaded dereference operator (`operator->`) for the `java_ptr<T>` class implements the safety semantics of Java pointers. The above code also demonstrates the power of C++ and that its support for operator overloading can be used in a clean manner to implement interesting semantics.

3.2.4 Translation summary

As the above examples indicate, the translation of Java source to C++ involves a straightforward and regular mapping of Java constructs into C++ constructs. The Java syntax for method bodies is almost directly translatable into C++. The use of template-based encapsulation classes allows the translator to implement the semantics of Java pointers and arrays in C++.

4 Performance Results

This section describes the performance improvements that were obtained by the translator. I will discuss both micro-benchmarks and large application programs.

4.1 Micro-benchmarks

To a good first approximation, Java code can be classified into two types: numerical (which make heavy use of integer and floating-point operations) and object-oriented (which make heavy use of memory allocation and method dispatch). The following table shows the performance of Java code and the translated C++ code on micro-benchmarks which exercise each of these two code categories; and the improvement factors that were obtained from translation. The names of the micro-benchmark are self-explanatory.

TABLE 2. usertimes in seconds, 200Mhz PentiumPro

micro-benchmark	Java	Translated C++	Improvement (Java/C++)
integer ops	67.3s	2.37s	28.4
floating-point ops	60.6s	2.00s	30.3
memory allocation	147s	88.5s	1.66
method dispatch	102s	29.9s	3.41
fast dispatch	102s	24.6s	4.15

As can be seen, the performance improvements are most dramatic for computation intensive code (whether integer or floating-point). Memory allocation (and associated garbage collection) seems to be the feature that is least amenable to performance improvement (this will be discussed further in the next section).

Depending on the mix of these operations in larger programs, one can expect performance improvements of anywhere between one and two orders of magnitude for the translated C++ code over the original Java code.

4.2 Application programs

The study of the performance improvement in large Java applications is complicated by two factors: the relative dearth of Object-Oriented applications and the impact of the runtime system on the performance of Java programs.

The following set of application programs are all CPU intensive in order to account for the differences between the performance of the I/O native methods in the JVM and those that I had implemented in the Java-to-C++ runtime library.

TABLE 3. usertimes in seconds, 200Mhz PentiumPro

program	C++	Java	Translated C++ (Xlate)	Improvement (Java/Xlate)	Gap (Xlate/C++)
mandelbrot	1.42s	81.0s	36.7s	2.20	25.8
mut-mandel	1.42s	38.6s	5.15s	7.49	3.63
dilemma	13.7s	322s	18.2s	17.7	1.33
genetic alg.	n/a	506s	33.7s	15.0	n/a
nqueens	n/a	2.88s	0.10s	28.8	n/a

The table indicates the performance of native C++ code (if it is available), the performance of the Java version of the application, and the performance of the translated C++ code. It also indicates the improvement factors for the translated C++ code over the original Java code and (where applicable) the gap factors remaining between the original C++ version of the code and the translated C++ version.

Mandelbrot is a floating-point and memory-allocation intensive mandelbrot iterator, *Mutmandel* is a version of the mandelbrot iterator that is not memory-allocation intensive (it performs about 3 orders of magnitude fewer allocations than *Mandelbrot*). *Dilemma* is a simulation of the evolution of populations of various strategies for playing the Prisoner's Dilemma game. *Genetic Alg.* is a genetic algorithm testbed. *Nqueens* finds solutions to arbitrary N-queens chess configurations.

The above table highlights the following features of the code produced by the Java-to-C++ translator.

- As indicated by the micro-benchmarks, between one and two orders of magnitude of performance improvement can be obtained for applications depending on how computation and memory-allocation intensive they are.
- Java applications that are not memory-allocation intensive can be translated into C++ code that can be executed to with a factor of 5 of semantically equivalent C++ code that was developed from scratch and carefully optimized. In the case of *Dilemma*, it is particularly encouraging to note the very comparable performance of the native C++ version of the application and the translated C++ version, since this application is highly object-oriented and relies quite heavily on dynamic dispatch.

  Therefore, one can expect to see comparable performance between "native" C++ versions and Java-to-C++ translated versions of OO-intensive programs.
- Memory-allocation seems to be the primary performance killer. None of these applications is garbage-collection intensive and a conservative collector based on reference-counting provides all the necessary semantics.

5 Bridging the Performance Gap

We saw in the previous section that the Java-to-C++ translator can provide substantial improvements in the performance of CPU-intensive Java applications. However, there is still a gap (and in some cases, a fairly large one) between the performance of the translated code and semantically equivalent C++ code developed from scratch. While it is not altogether *fair* to compare canonical C++ code against the idiomatic (but non-canonical) C++ code that results from the translation of Java programs, it is actually instructive to make such a comparison since it tells us where we must focus our efforts in order to get additional gains in performance (and equally importantly, where we should not be spending much effort).

I have classified the factors contributing to the performance gap into two groups:

- The *Evitable* factors: These are features of my specific implementation of the translator and the Java-to-C++ runtime system. These are also places where additional performance improvements seem possible and additional effort useful.

- The *Inevitable* factors: These are features of the Java programming language and its idioms that actually prevent optimization by compilers. These are also places where additional performance improvements seem unlikely and additional effort useless.

5.1 The Evitable factors

Both the Java-to-C++ translator and the supporting runtime library were written without any performance concerns in mind. The translator does produce “good” C++ code and the runtime library does perform fairly well, but there is scope for improvement in both places.

5.1.1 Runtime library

Perhaps the most important lesson to be learned from the exercise of translating Java to C++ is that the performance of the runtime system is very critical to the performance of Java applications. Almost all interesting methods in the standard Java packages are native; the code that is written in Java source is of a fairly trivial nature. This is particularly true of the AWT, Applet, IO, and Net packages.

A highly optimized runtime system seems essential to good performance of Java programs. This is true independent of the specific translation process used; for example, even if a Just-In-Time (JIT) compiler were used inside the JVM, it would still need to make calls into the JVM’s runtime library.

It is also interesting to note that the various implementations of the JVM differ primarily in the performance of their runtime systems.

5.1.2 Code inlining

It is generally accepted that code-inlining can buy upto an order of magnitude improvement in the performance of programs. The improvements come from a combination of improved code locality and the optimizations (particularly register allocation) that are enabled by procedure integration.

My current implementation of the Java-to-C++ translator does not allow cross-class method inlining since the class declaration and implementation are written to separate files (inlining between methods of the same class is still possible, however).

Increasing the opportunities for method inlining (particularly the inlining of constructors and accessor methods for classes) would require the inclusion of method bodies inside header files. This is difficult to do in general since the Java language allows the mutual dependence of class types and the mutual nesting of the corresponding class references. Still, in cases where there are no circular dependences, it is possible for the C++ compiler to inline method bodies across class boundaries and thus gain a substantial improvement in performance.

The primary problem with method inlining is that it is difficult to control with current C++ compilers and its effects on the performance of programs are highly system dependent (particularly w.r.t. cache sizes and cache configurations).

5.2 The Inevitable factors

There are a number of features of Java itself, stemming primarily from the reference-oriented semantics of the language, that actually preclude various kinds of optimizations that are possible in value-oriented languages like Standard ML/NJ or in value-based C++ idioms. With current compiler research, it seems unlikely that the performance gap resulting from the following factors can be easily bridged.

5.2.1 Memory allocation and lack of stack locality

Although it is difficult to quantify precisely, studies in pedagogy (and personal experience) indicate that programmers find it easier to reason in terms of values rather than in terms of references. However, a value-oriented approach to programming conflicts quite directly with Java's reference-oriented approach. Every object in Java that is not of one of the primitive types is a reference into the heap. Merely making these references opaque, dereference-safe, and garbage-collected does not solve the conceptual problem of dealing with the mutability inherent in references.

A reference-oriented approach has two related consequences:

- The programmer must explicitly allocate memory if a value-oriented idiom is to be implemented. The overhead of memory allocation (due to both the allocator itself and the lack of locality in the data cache) causes performance degradation of programs.
- In order to avoid the problems associated with memory allocation, programmers will feel the needless pressure to violate value-oriented semantics and implement, for example, mutable objects where they had earlier provided non-mutable ones. The *Mut-mandel* application program is a good example of this. The C++ code was written to use a `Complex` number class that provided non-mutable versions of addition and multiplication (i.e. adding or multiplying two `Complex` numbers produced a new `Complex` number and the values of the original numbers were unchanged). However, the code that made use of these `Complex` numbers could equally well be written with a mutable semantics for addition and multiplication, because `Complex` numbers were both read and written only once inside the loop.

  Such an identification and transformation should not be required of the programmer (whose burden is already sufficiently great). But the performance figures show us that these are precisely the examples where both the Java code and the translated C++ code perform very poorly, and hence are cases where the performance-conscious programmer will feel the need to adopt bad programming practices simply to improve the execution speed of their code.

  If at all possible, the compiler should identify and optimize such cases where new allocation is not needed. And the reference-oriented semantics of Java make this sort of identification either too difficult or even impossible.

C++, by not mandating either a value-oriented or a reference-oriented approach, gives programmers the freedom to choose the idiom most suited to the problem being solved. For example, C++ allows values to be allocated on the stack. This completely avoids any allocator related overheads since there is no allocation *per se*. The stack also has much better cache locality than does the heap and stack-allocated objects can be accessed with much less overhead than can heap-allocated objects. This allows programmers to both design value-oriented code and use it without serious concerns about performance degradation.

Conversely, it is unlikely that Java compilers or translators will ever be able to completely get around the memory allocation problem and make use of the stack for storing values. This will always result in a serious loss of performance for programs that depend on memory-allocation.

5.2.2 Safety restrictions

Java requires that scalar references and arrays be safe, i.e., each use of the reference must potentially be checked for both type-safety and value-legality (a null pointer cannot be dereferenced, and array bounds must be checked before use). While it may be possible in some cases to hoist these checks outside loops, such instances are rare. It is also not possible to even express such kinds of optimizations in Java bytecode, and probably equally difficult to do so in translated C++ without very aggressive analysis.

The safety restrictions of Java, combined with the use of references rather than values, preclude optimizations of both Java code and any translated version of it. It seems quite likely that performance gains can be obtained here only if the safety requirements are relaxed somewhat.

5.3 Garbage collection

Before the start of this project, I had expected garbage-collection to be a major issue in the translated C++ code. It was therefore something of a surprise to find out that a conservative collector based on reference-counting was sufficient for all of the application programs and for most of the Java language libraries (in fact, circular data-structures seem to be used only in the AWT and the Thread libraries; and even then, most data-structures in the Thread library cannot be safely garbage-collected).

The almost complete irrelevance of garbage-collection to the performance of many Java programs seems to indicate that the efficiency of garbage-collection algorithms (for example, *mark-and-sweep* vs. *generation-tracking*) will buy little in terms of the execution time of application programs.

5.4 Summary of the Performance of translated Java code

The areas most amenable to quick improvements in the performance of Java code (whatever the translation strategy or intermediate language) are the runtime system of the JVM and aggressive method-inlining (and register allocation). Optimization techniques specific to OO-languages, e.g. method specialization, dispatch caching, etc., may also be good aspects to investigate in extracting maximum performance from Java programs.

The areas least amenable to any sort of performance improvements are those dealing with memory allocation (it is necessarily slow), garbage-collection (it seems irrelevant given the problems with memory allocation), safety semantics (which are necessarily slow), and poor data locality (which is a consequence of reference-semantics and heap allocation).

6 Conclusions

In this project, I implemented a fairly comprehensive translator from Java source to C++. Combined with a suitable runtime support library, the generated C++ code can be compiled into native machine code that runs anywhere from an order to two orders of magnitude faster than the original Java program, for CPU-intensive applications. In some cases, particularly programs that are Object-Oriented, the performance of the generated C++ code is comparable (within 50%) to the performance of canonical C++ code developed from scratch and carefully optimized by hand.

My implementation of the translator compares favorably with *Toba*, a Java bytecode to C translator being developed at the University of Arizona. The Toba researchers report performance improvements of a factor of 30 on numerical micro-benchmarks, and anywhere between a factor of 1 to 5 for memory-allocation and OO-dispatch related micro-benchmarks. They also report a factor of 5 to 10 improvement in application performance. Other Java to C translators perform comparably.

The runtime support system and method inlining seem to be the primary areas in which additional performance gains are possible in such a translation scheme. Java's reference-oriented approach, the need for memory-allocation, and the safety requirements in the language introduce overheads that are difficult to optimize away without very aggressive and global analysis.

7 References

[1] Sun Microsystems, *Online Java language documentation*, `http://java.sun.com/`, 1996.

[2] ANSI/ISO Committee, *Online ANSI/ISO C++ public draft*. `http://www.cygnus.com/misc/wp/index.html`, October 1995.

[3] Larry Wall, Tom Christiansen, Randal Schwartz, *Programming Perl*, O'Reilly and Associates Inc., 1996.

[4] Free Software Foundation, *Online GNU documentation*, (`g++`, `bison`, `flex`).

[5] David Ungar, *Generation Scavenging: A Non-disruptive, High-performance Storage Reclamation Algorithm*, ACM Software Engineering Notes, 1984.

[6] Hans Boehm, *Space-Efficient Conservative Garbage Collection*, PLDI 1993.

[7] Andrew W. Appel, *Simple Generational Garbage Collection with Fast Allocation*, Software Practice and Experience, Vol. 19, No. 2, Feb. 1989.

[8] Todd Proebsting et. al, *Online Toba and Sumatra documentation*, `http://www.cs.arizona.edu/sumatra/toba/`. 1996.