RC 21393 (96614) 27JAN99 Computer Science/Mathematics

Research Report

Efficient Support for Complex Numbers in Java

Peng Wu
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801

Samuel P. Midkiff, José E. Moreira, Manish Gupta
IBM T. J. Watson Research Center
P. O. Box 218
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - T. J. Watson - Tokyo - Zurich

This page intentionally left blank.

Efficient Support for Complex Numbers in Java

Peng Wu*
pengwu@uiuc.edu
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801

Sam Midkiff José Moreira Manish Gupta
{*smidkiff,jmoreira,mgupta*}*@us.ibm.com*
IBM T. J. Watson Research Center
P. O. Box 218
Yorktown Heights, NY 10598-0218

Abstract

One glaring weakness of Java for numerical programming is its lack of support for complex numbers. Simply creating a `Complex` number class leads to poor performance relative to Fortran. We show in this paper, however, that the combination of such a `Complex` class and a compiler that understands its semantics does indeed lead to Fortran-like performance. This performance gain is achieved while leaving the Java language completely unchanged and maintaining full compatibility with existing Java Virtual Machines. We quantify the effectiveness of our approach through experiments with linear algebra, electromagnetics, and computational fluid-dynamics kernels.

1 Introduction

The Java Grande Forum has identified several critical issues related to the role of Java(TM)[1] in numerical computing [14]. One of the key requirements is that Java must support efficient operations on complex numbers. Complex arithmetic and access to elements of complex arrays must be as efficient as the manipulation of primitive types like `float` and `double`. Whereas Fortran and PL/I directly support complex numbers as a primitive type, complex numbers are typically implemented in Java as objects of a `Complex` class. In fact, there is a proposal for a standard Java `Complex` class that would facilitate the development of numerical applications with complex arithmetic [14]. We will show in Section 2 that implementing complex numbers as objects results in a significant performance penalty in current Java environments. In one of our benchmarks, MICROSTRIP, Java performance is 120 times slower than the equivalent Fortran code.

This paper shows how the right compiler technology can deliver efficient execution of complex arithmetic codes in Java. We rely on a technique called *semantic expansion*[2] to optimize the performance of Java applications that use standard classes for complex numbers and complex arrays. Semantic expansion treats selected Java classes and methods as language primitives. The compiler recognizes the method invocations in the class file (*i.e.*, at the bytecode level) and directly translates them into an intermediate language representation of their semantics. The crux of this technique is that it allows knowledge about standard classes to be embedded within the compiler. When applied to the `Complex` class, the net effect of the semantic expansion technique is to create a virtual machine that supports complex numbers as primitive types, with *no change* to the Java source or class files. That is, complex numbers continue to appear as objects when viewed by a regular Java Virtual Machine (JVM).

The performance benefits of our approach can be enormous. When optimized with our semantic expansion techniques, the MICROSTRIP benchmark shows a 76-fold speedup, achieving 63% of the performance of equivalent Fortran code. In some other benchmarks, Java achieves 90% of the best Fortran performance. Semantic expansion

*Work performed while the author was an intern at the IBM T. J. Watson Research Center.
[1] Java is a trademark of Sun Microsystems, Inc.
[2] *Semantic expansion* is called *semantic inlining* in our earlier work.

transforms Java into a real contender for the development of scientific and engineering codes that manipulate complex numbers.

This paper is organized as follows. Section 2 describes the performance problems associated with complex numbers in Java, using the MICROSTRIP benchmark as a working example. Section 3 presents the details of our semantic expansion implementation for complex arithmetic and multidimensional arrays of complex numbers. Our experimental work and benchmarking results are reported in Section 4. Section 5 is a discussion of more general aspects of semantic expansion, in particular as a mechanism for transparently implementing lightweight objects. Related work is discussed in Section 6 and conclusions are presented in Section 7.

2 Complex numbers in Java

The Java programming language and the Java Virtual Machine do not support complex numbers as primitive data types. Therefore, the typical mechanism for implementing complex numbers in Java is through a `Complex` class. A subset of the code for such class is shown in Figure 1(a). With this approach, each complex number is represented as a `Complex` object. In most Java implementations, a `Complex` object would require 32 bytes for representation: 16 bytes for the standard object descriptor (present in all objects) and 16 bytes for the real and imaginary fields. This representation is shown in Figure 1(b).

```
public final class Complex {

    public double re, im;

    public Complex(double r, double i) { re = r; im = i; }
    public Complex assign(Complex z) { re = z.re; im = z.im; return this; }
    public Complex plus(Complex z) { return new Complex(re+z.re,im+z.im); }
    public Complex minus(Complex z) { return new Complex(re-z.re,im-z.im); }
    public Complex times(Complex z) {
        return new Complex(re*z.re-im*z.im,im*z.re+re*z.im);
    }
    public Complex times(double x) { return new Complex(re*x,im*x); }
    public Complex plusAssign(Complex z) { re += z.re; im += z.im; return this; }
}
```

(a) code for `Complex` class

```
a = new Complex(0,0);
```

(b) `Complex` object creation and representation

Figure 1: A standard Java class for complex numbers.

With the `Complex` class, arithmetic operations are performed through method invocations and intermediate computation results are represented by temporary `Complex` objects. Let `a`, `b`, and `c` be `Complex` objects. The expression $a = a \star (b + c)$ can be computed by the code:

```
a.assign(a.times(b.plus(c)));
```

Note that three method invocations and two temporary object creations occur while evaluating this simple expression. Even though the standard inlining performed by Java compilers eliminates the method overhead, we will see that the unnecessary object creation and subsequent object destruction operations place a heavy toll on program performance.

Java arrays of `Complex` objects can be easily created with conventional language constructs such as

```
Complex[] c = new Complex[n];
for (int i=0; i<n; i++) c[i] = new Complex(0,0);
```

Note that `c` is an array of *references* to `Complex` objects. The second line in the code above allocates the actual `Complex` objects. The resulting structure is shown in Figure 2. This organization is very inefficient when compared to arrays of primitive types in Java, or to arrays of complex numbers in Fortran. First, there is an extra level of indirection to get to any complex number in the array. Second, at least 50% of the storage is used for object descriptors. That is, this organization uses at least twice as much memory as a Fortran array of complex numbers.

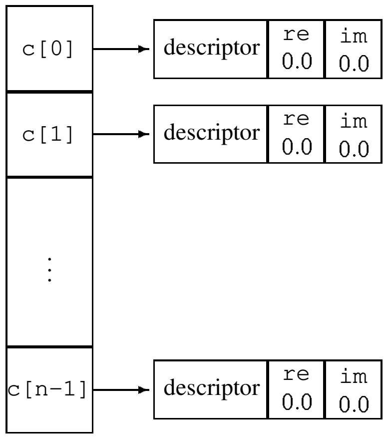


Figure 2: Structure of a Java array of `Complex` objects.

We quantify the performance impact of supporting complex numbers in Java through a working example. The MICROSTRIP benchmark [15] computes the value of the potential field $\Phi(x,y)$ in a two-dimensional microstrip structure [8]. The structure is discretized by an $(w+1)\times(h+1)$ mesh and the potential is calculated through the iterative solution of a partial differential equation. (For benchmarking we set $w = h = 999$ and we use a structure with 4 microstrips, as described in [15].) At each step of the solver, two Jacobi relaxation operations and an error reduction are performed. The key computation in the relaxation is the evaluation of

$$\Phi'_{i,j} = \frac{1}{4}(\Phi_{i+1,j} + \Phi_{i-1,j} + \Phi_{i,j+1} + \Phi_{i,j-1}) \tag{1}$$

for each point (i,j) of the mesh. MICROSTRIP has two modes of operation: DC and AC.

In the DC mode of operation, Φ is a real-valued field and the computation of Equation (1) can be performed with real arithmetic. It can be coded as

```
b(i,j) = 0.25*(a(i+1,j)+a(i-1,j)+a(i,j+1)+a(i,j-1))        (2)
```

in Fortran and

```
b[i][j] = 0.25*(a[i+1][j]+a[i-1][j]+a[i,j+1]+a[i,j-1])        (3)
```

in Java. Both constructs operate on primitive types (`real*8` for Fortran and `double` for Java) and can be executed quite efficiently, as shown by the results of the benchmark in DC mode. Java achieves 49 Mflops and Fortran achieves 97 Mflops on an RS/6000 model 590.

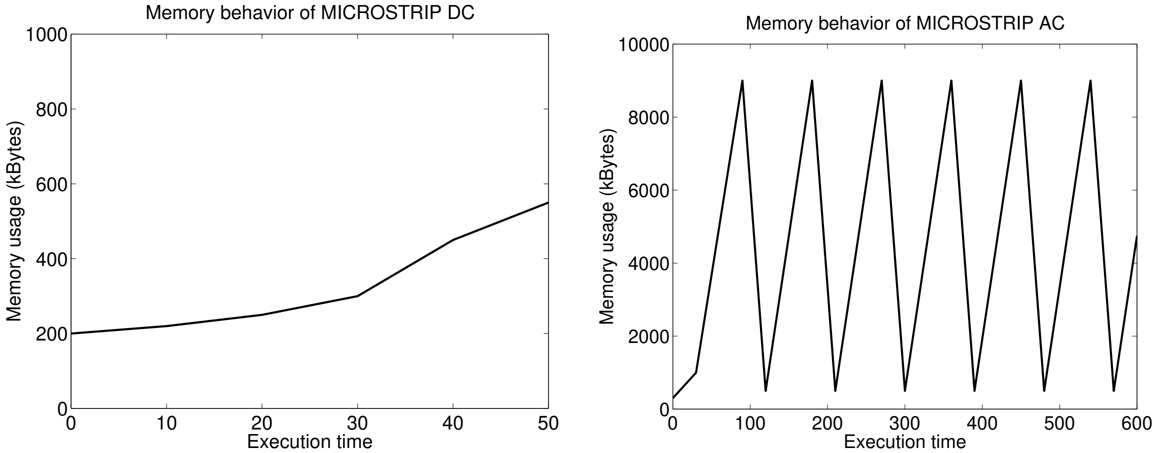
In the AC mode, Φ is a complex valued function and Equation (1) must be evaluated using complex arithmetic. Since Fortran supports complex primitive types (`complex*16` in particular), the Fortran code for Equation (1) in AC mode is the same as shown in (2). Using the `Complex` class, Equation (1) in Java for AC mode can typically be coded as

```
b[i][j].assign(a[i+1][j].plus(a[i-1][j]).plus(a[i][j+1]).plus(a[i][j-1]).times(0.25))
```

(4)

where `a` and `b` are of type `Complex[][]`.

Each complex arithmetic operation in Java creates a new complex object to represent the result of the operation. This newly created object is immediately used in the next operation and then discarded. This voracious rate of object creation and destruction can be visualized by plotting the Java MICROSTRIP memory utilization, as done in Figure 3(a) (for DC mode) and Figure 3(b) (for AC mode). The memory utilization in DC mode grows slowly, and does not require any garbage collection during the lifetime of the benchmark. In comparison, the memory utilization grows rapidly in AC mode, exhausting the total memory allocated for the application in a few iterations, thus forcing garbage collection to occur. Thus, the execution time of the Java AC mode is dominated by the cost of object creation and destruction. (Because of limitations in our monitoring tool, the plots shown in Figure 3 are for a smaller problem size, with $w = h = 99$.) In AC mode, running on the same RS/6000 model 590, the Fortran version of the benchmark achieves 120 Mflops, while the Java version achieves only 1 Mflop.



(a) Memory usage in DC mode

(a) Memory usage in AC mode

Figure 3: Memory utilization in MICROSTRIP.

An alternative to representing `a` and `b` as Java arrays of `Complex` objects is to use one of the standard multidimensional array classes also being proposed by the Java Grande Forum. Figure 4 shows some components of class `ComplexArray2D`, which implements two-dimensional arrays of complex numbers. Inside the array, values are stored in a packed form, with (*real*,*imaginary*) pairs in adjacent locations of the `data` array. (This is similar to the Fortran style for storing arrays of complex numbers.) The `get` operation returns a `Complex` object with the value of an array element, and the `set` operation assigns the value of a `Complex` object to an array element. It is easy to create arrays using the `ComplexArray2D` class:

```
ComplexArray2D a = new ComplexArray2D(w+1,h+1)
```

creates and initializes to zero a $(w+1) \times (h+1)$ array of *complex values* (not complex objects). Equation (1) can be coded as:

```
b.set(i,j,
      a.get(i+1,j).plus(a.get(i-1,j)).plus(a.get(i,j+1)).plus(a.get(i,j-1)).times(0.25)
      )
```

(5)

```
public final class ComplexArray2D {

   private double[] data;

   public ComplexArray2D(int m, int n) {
       data = new double[2*m*n];
   }

   public Complex get(int i, int j) {
       return new Complex(data[2*(i*n+j)],data[2*(i*n+j)+1]);
   }

   public void set(int i, int j, Complex z) {
       data[2*(i*n+j)] = z.re;
       data[2*(i*n+j)+1] = z.im;
   }
}
```

Figure 4: A (proposed) standard Java class for two-dimensional complex array.

The `ComplexArray2D` class offers a significant storage benefit compared to `Complex[][]`, since it only stores complex values and not `Complex` objects (a 50% reduction in storage, and associated improvements in memory bandwidth and cache behavior). The performance of numerical codes that use `ComplexArray2D`, however, is just as bad as codes that use Java arrays of `Complex` objects. Execution continues to be dominated by object creation and destruction. In fact, every `get` operation on a `ComplexArray2D` results in a new temporary object. The performance of MICROSTRIP using `ComplexArray2D` is also approximately 1 Mflop.

3 Semantic expansion of complex numbers and multidimensional arrays

Java performance on numerical codes with complex numbers is severely hampered by the overhead of object manipulation. New techniques for stack allocation of temporary objects and more sophisticated garbage collection can alleviate some of these costs [6, 17]. However, to achieve Fortran-like performance on complex arithmetic we must move beyond treating complex numbers as objects. We need to somehow directly manipulate complex values in registers, avoiding as much additional processing as possible. Machine code that directly manipulates complex values can be generated from Java using a technique called semantic expansion.

3.1 Overview of semantic expansion

Semantic expansion is a compilation strategy whereby selected classes are treated as language primitives by the compiler – their semantics are hard-wired into the compiler and optimizations based on the expanded semantics are designed accordingly. We have applied semantic expansion to the `Complex` and `ComplexArray` classes to effectively enhance our Java compiler with the knowledge of complex numbers. We are not changing the Java language at all. Complex numbers continue to be represented through objects at the Java source and bytecode levels and continue to be portable across Java implementations. When a program with complex numbers is submitted for compilation, the compiler simply takes advantage of its knowledge of the semantics of the `Complex` and `ComplexArray` classes to optimize the code aggressively.

This approach of enhancing a compiler with knowledge about the classes that implement complex numbers and multidimensional complex arrays is feasible and beneficial for several reasons. First, complex numbers are essential in many areas of science and engineering. Therefore it is reasonable to extend a compiler to treat them as primitive types (as Fortran and PL/I compilers do). Second, the semantics of complex numbers are simple and well-defined. Adding treatment for complex numbers in a compiler is inexpensive. Third, the `Complex` and `ComplexArray` classes are declared `final`. This makes it very easy for the compiler to identify the exact method that is being invoked in each use. Finally, because of their mathematical nature, treating complex numbers as primitives offers the compiler opportunities to perform very aggressive optimizations. We will quantify the enormous benefits from these optimizations in Section 4, but we discuss them qualitatively next. We describe our actual implementation of semantic expansion in Section 3.2.

The first optimization provided by semantic expansion is the elimination of method invocation overhead. The compiler replaces method invocations by code that directly implements the semantics of the method. Method invocation overhead can also be eliminated through standard inlining techniques, but semantic expansion can do more because it does not depend on the actual implementation in the class file. The compiler is free to generate any code that conforms to the specified semantics. We can keep the reference implementation simple, while the compiler uses a highly optimized implementation.

The single most important optimization provided by semantic expansion in support of complex numbers in Java is what we call *lightweight object substitution*. The basic idea is to replace `Complex` objects by more lightweight representations. These lightweight representations are semantically rich enough to serve as the object in certain program contexts, but are overall cheaper to create and manipulate. In particular, we want to use *complex values* (a pair of real and imaginary values) as much as possible. The evaluation of arithmetic expressions with complex numbers, as in Equation (4), can be performed entirely with complex values, without creating any new `Complex` objects.

Lightweight object substitution in turn makes an array class like `ComplexArray2D` very attractive. In many situations the `get` method only needs to return a complex value, thus eliminating the need to create a new object. We can then benefit from the improved layout offered by that class, which saves us at least 50% in storage and eliminates extra levels of dereferencing. Note that the array class by itself does not help much, as discussed in Section 2, because the execution time continues to be dominated by object creation and destruction. It is only when that cost is eliminated that we are able to get additional leverage from the `ComplexArray` class.

3.2 The implementation

Our prototype implementation is based on the IBM High Performance Compiler for Java (HPCJ) [5, 20], which generates native code for different IBM platforms. HPCJ compiles Java class files (bytecodes) into an intermediate stack language called W-code. W-code supports a rich variety of data types, including complex and decimal numbers. W-code and W-code based optimizers provide a comprehensive optimization and analysis infrastructure. We implemented semantic expansion in the *Toronto Portable Optimizer* (TPO), which is used by the IBM XL family of compilers for language-independent optimizations. TPO is a W-code to W-code restructurer that sits in between the HPCJ front-end and the architecture specific code generation back-end. We want to emphasize that semantic expansion occurs *after* the bytecode phase of Java compilation, before machine executable code is generated. Therefore, Java source (`.java`) and bytecode (`.class`) files continue to be portable across all Java implementations.

In performing lightweight object substitution, we make use of the W-code complex data type (denoted as *wcomplex*[3]) to serve as the lightweight representation of `Complex` objects. The *wcomplex* data type is cheaper to create, faster to access, and, as a primitive W-code data type, is subject to a wide range of optimizations already existing in TPO. During the substitution, the compiler attempts to minimize the number of temporary `Complex` objects created. This is achieved by substituting the use or creation of `Complex` objects with those of *wcomplex* data elements whenever it is safe.

In the `Complex` class (Figure 1), the `plus`, `minus`, and `times` methods are semantically expanded to directly operate on two *wcomplex* arguments and return a *wcomplex* element as its result. The `plusAssign` and `assign` methods are semantically expanded to take one `Complex` argument (the `this`) and one *wcomplex* argument, and return a `Complex` reference. Similarly, the `get` method of class `ComplexArray2D` (Figure 4) returns a *wcomplex* element and the `set` method accepts a *wcomplex* as input. With these substitutions, expressions like those in Equation (4) and Equation (5) can be evaluated without any `Complex` object creation.

Conversion between *wcomplex* elements and `Complex` objects is performed as required by the types of the formal and actual arguments of operations. If an arithmetic operation is performed on a `Complex` object, it is necessary to extract the corresponding *wcomplex* value from it. This operation is typically free, since every `Complex` object contains effectively contains a *wcomplex* element inside it. Correspondingly, if an object-oriented operation is performed on a *wcomplex* element, it has to be converted into a `Complex` object. This is a more expensive operation, since it involves object creation, but is not very frequent in numerical codes. Conversion points can be conservatively identified through a simple inspection of the abstract syntax tree (AST) representation of an expression. Figure 5(a) shows the original AST for the expression `foo(a.times(b.plus(c)))`, where `foo` is some user function that expects a `Complex` object as its parameter. The $\star$ and $+$ internal W-code operations expect *wcomplex* operands, which need to be extracted from the `Complex` objects `a`, `b`, and `c`. The transformed AST of Figure 5(b) shows the *value* operations that perform this conversion. The `foo` method expects an actual `Complex` object, and therefore the *wcomplex* result of the multiplication has to be converted with an *object* operation. This is also shown in Figure 5(b).

The simplicity of the current implementation can in some cases limit the achieved performance. The current implementation always uses a full `Complex` object wherever the formal argument of an operation is `Complex`. Thus the "=" (reference assignment) operation always causes an object to be created. For example `a = b.plus(c)` requires an object creation in our current implementation. (Note that `a.assign(b.plus(c))` *does not* require an object creation.) The current implementation also uses the full object representation for any `Complex` object passed to

[3] *wcomplex* is a W-code primitive, represented by *real* and *imaginary* parts.

(a) (b)

Figure 5: The AST for a complex arithmetic expression.

a (non-semantically expanded) method (*e.g.*, `foo`). If all uses of the argument to `foo` require only a *wcomplex* value, `foo` can be compiled to accept a *wcomplex* argument. A simple data-flow analysis can be performed intraprocedurally or interprocedurally to reduce the number of complex numbers requiring a full object representation. We stress that our experimental results to date have shown very good results without this more general analysis, and that it does not appear necessary to gain the majority of the benefits of semantic expansion of the `Complex` package.

Although the constraint that classes be `final` is acceptable for our current goal of efficiently implementing new primitive data types, future applications may require easing this. For semantic expansion of method calls, it is necessary to have an analysis that can determine that the call is to a method in the original recognized class, and not some derived class. This is also sufficient to allow data rearrangement by semantic expansion, since all accesses of the rearranged data will be via methods of the original recognized class.

4 Experimental results

In this section we analyze in more detail the behavior of MICROSTRIP and other benchmarks. We measure the performance of Java, C++, and Fortran versions for each of the benchmarks. We perform all our experiments on an IBM RS/6000 model 590 workstation. This machine has a 67 MHz POWER2 processor with a 256 kB single-level data cache and 512 MB of main memory. The peak computational speed of this machine is 266 Mflops. Fortran programs are compiled using version 4.1 of the IBM XLF compiler with the highest level of optimization (-O3 -qhot, which performs high-order loop transformations [19]). C++ programs are compiled using version 3.6 of the IBM XLC compiler with its highest level of optimization (-O3 – *i.e.*, no high-order transformations). Java programs are statically compiled with a development version of the IBM HPCJ compiler and array bounds checking is optimized using the methods described in [16]. (With this optimization, the cost of bounds checking is effectively zero.) The optimization level for Java is -O, which is the highest level available for this language (in other IBM compilers, -O3 uses associativity to optimize expressions). In accordance with current Java floating-point semantics, the `fma` (fused multiply-add) instruction of the POWER architecture is disabled for Java programs but enabled for Fortran and C++ programs. Loop nests are arranged in each of the Java and Fortran versions of a benchmark to favor the data layout of the corresponding language.

The results for MICROSTRIP AC are summarized in Figure 6. The numbers at the top of the bars indicate actual Mflops for each case. The bar labeled plain shows the performance of the standard Java implementation of this benchmark using `Complex[][]` arrays (*i.e.*, code is as shown in Equation (4)). We remind the reader that a version with `ComplexArray2D` arrays (*i.e.*, code as shown in Equation (5)) performed no better. The C++ bar shows the results for a C++ implementation of this benchmark. Complex numbers in C++ are implemented through a class with a *(real,imaginary)* pair of `doubles`. Arithmetic operations with complex numbers in C++ do not incur any object creation, and complex arrays have an efficient dense storage, but the compiler is *not aware* of the semantics of complex numbers. The complex bar shows the result after semantic expansion of the `Complex` class (with `Complex[][]` arrays). The array bar shows the result from using the `ComplexArray2D` array class *with* semantic expansion of both the `Complex` and `ComplexArray2D` classes. Finally, Fortran is the result for the Fortran version of this benchmark, with the highest level of optimization.

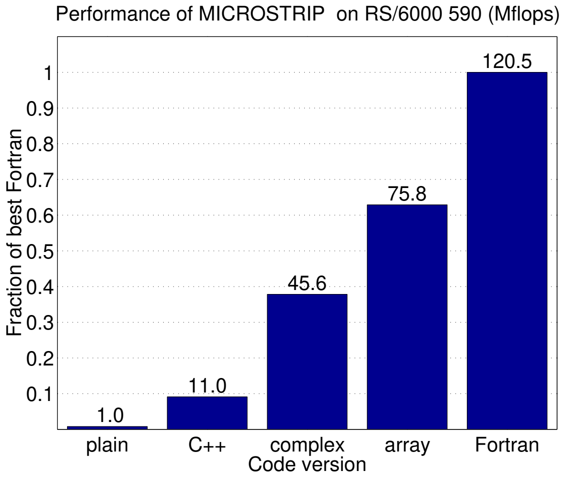


Figure 6: Summary of results for MICROSTRIP AC.

We complete the experimental evaluation of our techniques by applying semantic expansion to four additional benchmarks. All of the benchmarks are numerical codes with complex arithmetic. The benchmarks are: MATMUL, LU, FFT, and CFD. MATMUL computes $C = C + A \times B$, where C, A, and B are complex matrices of size 500×500. We use a dot-product version of matrix multiplication, with an i, j, k-loop nest. The i, j, and k loops are blocked and the i and j loops are unrolled, in all versions, to improve performance [19]. LU is a straightforward implementation of Crout's algorithm [10, 18] for performing the LU decomposition of a square matrix A, with partial pivoting. The factorization is performed in place and, in the benchmark, A is of size 500×500. FFT computes the discrete Fourier transform of a two-dimensional complex function, represented by an $n \times m$ complex array. We use the Daniel-Lanczos method described in [18] to compute the one-dimensional FFTs in the two-dimensional FFT. For our experiments we use $n = m = 256$. CFD is a kernel from a computational fluid dynamics application. It performs three convolutions between pairs of two-dimensional complex functions. Each function is represented by an $n \times m$ complex array. The computation performed in this benchmark is similar to a two-dimensional version of the NAS parallel benchmark FT. Again, for our experiments we use $n = m = 256$.

Results for these four benchmarks are summarized in Figure 7. The labels for the bars are the same as with MICROSTRIP. In all cases we observe significant performance improvements when the `Complex` class is semantically expanded. Improvements range from a factor of 13 (1.1 to 14.6 Mflops for LU) to a factor of 50 (1.1 to 55.5 Mflops for MATMUL). Further improvements can be obtained using a semantically expanded multidimensional array class. In that case we achieve Java performance that ranges from 65% (MATMUL and LU) to 90% (FFT and

CFD) of fully optimized Fortran code. The gain with array classes is particularly significant in the LU benchmark. The implementation of Crout's algorithm performs accesses both along rows and columns of matrix A in the computation of the dot-products. Accesses along columns, in Java, can be performed much more efficiently with a dense storage layout, as provided by the multidimensional array class `ComplexArray2D`. Part of the greater performance advantage of Fortran in the LU and MATMUL benchmarks comes from its ability to use the `fma` instruction. Disabling it for Fortran causes performance drops of at least 13% in those two benchmarks. (It does not affect MICROSTRIP, FFT, or CFD significantly.) A proposal for extending the floating-point semantics of Java, that would allow it to use the `fma` instruction, is currently under public review [14]. Another substantial advantage of the Fortran compiler is its ability to perform high-order loop transformations.

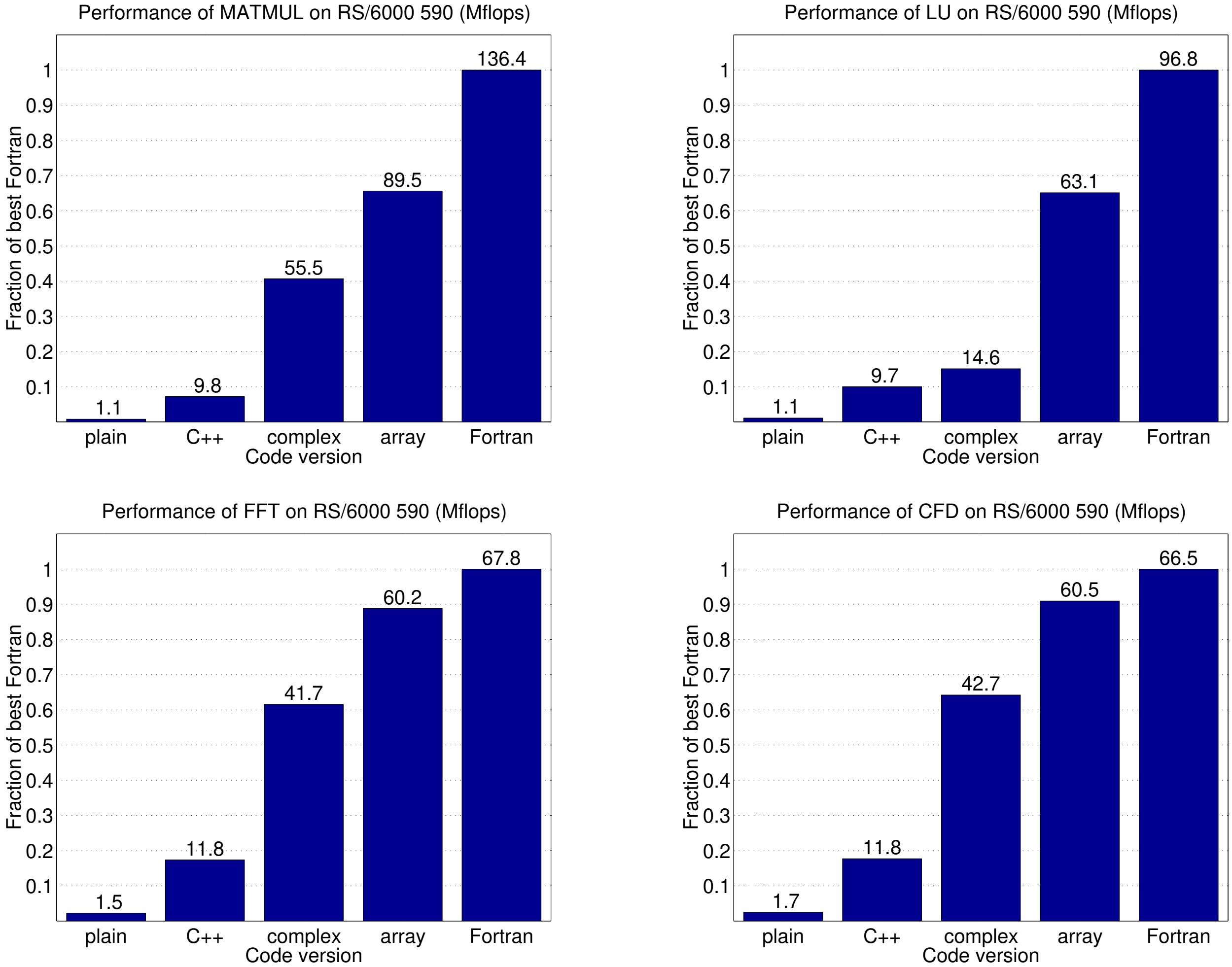


Figure 7: Summary of experimental results for MATMUL, LU, FFT, and CFD.

5 Discussion

Our methodology derives its performance benefits from treating objects of `Complex` class as simple variables holding complex values rather than true objects, when it is safe to do so. The term *lightweight object* is often used to refer to such a representation which is more efficient. Several authors [11, 9] have advocated the explicit introduction of lightweight objects (also called *value objects*) into Java. Other researchers have proposed techniques like object inlining [7] and unboxing [12], which use interprocedural data flow analysis to detect situations where objects may be replaced by lightweight objects that carry just the data values. We argue that our approach provides the performance benefits of the other approaches without altering the language or requiring extensive new compile time analyses.

The addition of lightweight objects to the language complicates its semantics considerably. Lightweight classes do not fit into the Java class hierarchy, where each class is derived from the `Object` class. It becomes necessary to enforce awkward restrictions [14] like: (i) a lightweight object cannot be assigned or compared with a `null` value, (ii) it cannot be cast to `Object` or any other reference type, (iii) `finalizer` methods are not allowed in a lightweight class, (iv) a lightweight class constructor must not explicitly invoke `super`, (v) the `instanceof` operator cannot be applied to an expression having the type of a lightweight class. In contrast, it is much cleaner to simply make the compiler recognize important class libraries which are declared as `final`. The semantic knowledge of methods in such a class makes it trivial for a compiler to infer which methods may operate on the lightweight object instead. Using straightforward local analysis, a compiler can greedily use lightweight objects, converting to a true object representation only on demand, when a method requiring the generality of a true object is encountered. Hence, no additional semantic restrictions need to be imposed on objects of these standard classes, and performance gains are obtained if these objects are manipulated mainly via the methods declared directly in these classes.

Furthermore, our approach ensures complete backward compatibility with existing JVMs. The reference implementation of the class will allow older JVMs (or any JVM that does not support semantic expansion of the particular class) to execute the user application correctly. There is merely a possible performance loss if the JVM chooses not to take advantage of the semantics of the class. As a consequence of this backwards compatibility, a JVM or static compiler can pick and choose which classes to semantically expand. It is reasonable to expect compilers for embedded systems, numerical applications or business applications to target different classes for semantic expansion. Furthermore, JVM developers can introduce new classes and semantically expand those classes, providing portability via the reference implementation and performance via semantic expansion of the new class. The standard naming conventions will prevent confusion between classes from different vendors.

The approaches relying exclusively on compiler analysis to inline [7] and unbox [12] objects have a potential advantage in that they can work with classes that are not semantically understood by the compiler. However, semantic expansion provides several important advantages over these approaches. First, large benefits are gained even with very local, simple analysis, making the technique appropriate for dynamic compilers. Second, because large benefits are gained through local knowledge, semantic expansion is not hampered by incomplete or conservative analysis. In a language with late bindings like Java, this is an important feature for static compilers. Finally, because the compiler can be endowed with a deep semantic understanding of the expanded class and its methods, transformations that are difficult (if not impossible) to infer from the Java code for the class can be performed. C++ effectively uses lightweight objects to represent complex numbers. Nevertheless, as we have seen in Section 4, a Java compiler with specific understanding of the `Complex` class semantics performs significantly better than C++.

The main cost of our approach is that additional work, in terms of compiler implementation, must be done for each class to be semantically expanded. We do not view this as a major impediment, for several reasons. First, the classes whose semantic expansion offer the most performance gains are those corresponding to often used primitive types in domain specific languages. Examples include (i) complex numbers and true multidimensional arrays in Fortran, and (ii) fixed decimal types and strings in Cobol and other commercially oriented languages. Many of these types are already present (in the form of classes) in the standard Java class libraries and just waiting for optimization. Second, the amount of work necessary to semantically expand one of these classes is small – on the order of the amount of time necessary to implement the corresponding language primitive. Third, more sophisticated analyses can co-exist with semantic expansion if unboxing of objects which are not semantically expanded is desired.

6 Related work

Our semantic expansion technique is similar to the approach taken by Fortran compilers in implementing some *intrinsic* procedures. Intrinsic procedures can be treated by the Fortran compiler as language primitives. Semantic expansion differs from Fortran's handling of intrinsics in that both new data types and operations on the new data types are treated as language primitives.

Work by Peng and Padua [21] targets standard container classes for semantic expansion. In their paper, the container semantics are not used for local optimizations or transformations inside the container. Instead, the semantics are used to help data flow analysis and detect parallelizable loops. Their work illustrates how semantic information about standard classes exposes new optimization and parallelization opportunities.

The work of Dolby [7] (targeting C++) and the work of Hall *et.al.* [12] (targeting Haskell) are related to ours in that they appear to duplicate our results with complex numbers, albeit in a different environment. In [7], aggressive interprocedural analysis identifies *all* uses of the fields of an object, and cloning creates specialized methods for each possible type of a field. This in turn allows the objects themselves to be inlined. Savings accrue from reducing the number of virtual function dispatches and from reducing the dereferencing overhead necessary to access objects fields. The techniques in [12] allows programmers to specify a base data type (consisting only of the primitive fields of a declared object), and propagate this new type through the methods operating on the declared object. We have compared the relative merits of these approaches and our work in Section 6.

In [4], Cierniak and Li describe a global analysis technique for determining that the shape of a Java array of arrays (*e.g.*, `double[][]`) is indeed rectangular and not modified. The array representation is then transformed to a dense storage and element access is performed through index arithmetic. We differ in that we do not need global analysis and in our integration of complex and true multidimensional array optimizations.

7 Conclusions and future work

We have demonstrated in this paper that high performance numerical codes using complex numbers can be developed in Java. Earlier work by our group and others [1, 2, 3, 13] has demonstrated that high performance can be obtained for codes written using the Java primitive floating point types. Our present results, combined with those earlier results, show that the answer to the question of whether Java is a suitable language for developing high performance numerical codes is an unequivocal yes.

Semantic expansion is simple to implement, has very low compile time overhead, and is compatible with existing Java compilers and Java Virtual Machines. This means that semantic expansion is usable by dynamic compilers and that Java programs developed in an environment which implements semantic expansion can be run on any Java Virtual Machine. We are currently investigating the use of semantic expansion for fixed decimal types and the Java `String` class, as well as more complex container classes, to get a better feel for the broad applicability of this technique.

Acknowledgments: The authors wish to thank Vivek Sarkar, Rick Lawrence, and Marc Snir for fruitful technical discussions, and for strongly supporting our research. We also wish to thank Ven Seshadri of IBM Canada for helping and supporting our experiments with HPCJ, and George Almasi and Albert Lee for their data on memory usage in the MICROSTRIP benchmark.

References

[1] B. Blount and S. Chatterjee. An evaluation of Java for numerical computing. In *Proceedings of ISCOPE'98*, volume 1505 of *Lecture Notes in Computer Science*, pages 35–46. Springer Verlag, 1998.

[2] R. F. Boisvert, J. J. Dongarra, R. Pozo, K. A. Remington, and G. W. Stewart. Developing numerical libraries in Java. In *ACM 1998 Workshop on Java for High-Performance Network Computing*. ACM SIGPLAN, 1998. Available at `http://www.cs.ucsb.edu/conferences/java98`.

[3] H. Casanova, J. Dongarra, and D. M. Doolin. Java access to numerical libraries. *Concurrency, Pract. Exp. (UK)*, 9(11):1279–91, November 1997. Java for Computational Science and Engineering - Simulation and Modeling II Las Vegas, NV, USA 21 June 1997.

[4] M. Cierniak and W. Li. Just-in-time optimization for high-performance Java programs. *Concurrency, Pract. Exp. (UK)*, 9(11):1063–73, November 1997. Java for Computational Science and Engineering - Simulation and Modeling II, Las Vegas, NV, June 21, 1997.

[5] IBM Corporation. IBM High Performance Compiler for Java, 1997. Information available in Web page at `http://simont01.torolab.ibm.com/hpj/hpj.html`, available for download at `http://www.alphaWorks.ibm.com/formula`.

[6] A. Deutsch. On the complexity of escape analysis. In *Proc. 24th Annual ACM Symposium on Principles of Programming Languages*, pages 358–371, San Diego, CA, January 1997.

[7] J. Dolby. Automatic inline allocation of objects. In *Proc. ACM SIGPLAN'97 Conference on Programming Language Design and Implementation*, pages 7–17, 1997.

[8] T. C. Edwards. *Foundations for Microstrip Circuit Design*. John Wiley & Sons, Chichester, NY, 1992.

[9] Java Grande Forum. Issues in numerical computing with Java. Document available at URL `http://math.nist.gov/javanumerics/issues.html`, March 1998.

[10] G. H. Golub and C. F. van Loan. *Matrix Computations*. Johns Hopkins Series in Mathematical Sciences. The Johns Hopkins University Press, 1989.

[11] James Gosling. The evolution of numerical computing in Java. Document available at Web-page `http://java.sun.com/people/jag/FP.html`. Sun Microsystems.

[12] C. Hall, S. Peyton-Jones, and P. Sansom. Unboxing using specialisation. In *Functional Programming: Workshops in Computing*, 1996.

[13] M. Jacob, M. Philippsen, and M. Karrenbach. Large-scale parallel geophysical algorithms in Java: A feasibility study. *Concurrency: Pract. Exp. (UK)*, 10(11-13):1143–1153, 1998.

[14] Java Grande Forum. Report: Making Java work for high-end computing. Java Grande Forum Panel, SC98, November 1998. URL: `http://www.javagrande.org/reports.htm`.

[15] J. E. Moreira and S. P. Midkiff. Fortran 90 in CSE: A case study. *IEEE Computational Science & Engineering*, 5(2):39–49, April-June 1998.

[16] J. E. Moreira, S. P. Midkiff, and M. Gupta. From flop to Megaflops: Java for technical computing. In *Proceedings of the 11th International Workshop on Languages and Compilers for Parallel Computing, LCPC'98*, 1998. IBM Research Report 21166.

[17] Y. G. Park and B. Goldberg. Escape analysis on lists. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 117–127, July 1992.

[18] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in FORTRAN: The Art of Scientific Computing*. Cambridge University Press, 1992.

[19] V. Sarkar. Automatic selection of high-order transformations in the IBM XL Fortran compilers. *IBM Journal of Research and Development*, 41(3):233–264, May 1997.

[20] V. Seshadri. IBM high performance compiler for Java. *AIXpert Magazine*, September 1997. Electronic publication available at URL `http://www.developer.ibm.com/library/aixpert`.

[21] Peng Wu and David Padua. Beyond arrays – a container-centric approach for paralllelization of real-world symbolic applications. In *Proceedings of the 11th International Workshop on Languages and Compilers for Parallel Computing, LCPC'98*, 1998.