RC 21313 (96030) 14OCT98 Computer Science/Mathematics

Research Report

Improving Java Performance Through Semantic Inlining

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.

Improving Java Performance Through Semantic Inlining

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

This paper describes a new compilation technique called *semantic inlining*. Unlike traditional lexical inlining, which only incorporates a lexical representation of a method into the calling context, semantic inlining incorporates into the calling context semantic information about the method and its class. This is accomplished by treating selected classes and methods, which are crucial to performance, as primitive data types and primitive operations on those data types. We show that, in our numerical benchmarks which use complex numbers and multidimensional arrays, semantic inlining yields speedups of between 35 and 81, delivering performance between 65% to 90% of Fortran. The speed and effectiveness of semantic inlining make it practical for both static and dynamic compilers, and make it an effective technique for extending the primitive data types and operations of the Java programming language without altering the Java Virtual Machine specification.

1 Introduction

The Java(TM)[1] programming language, with its portability, extensibility, and expanding programmer base, has drawn serious attention from the technical and commercial computing community, with important applications in these domains already being developed in Java [4, 13]. However, for Java to become a major programming language in these fields, its implementations must deliver performance comparable to that from more traditional languages, such as Fortran and C.

The modern trend in programming language design is to move the complexity from the language (*e.g.*, as in PL/I [1]) into class libraries. The benefit of this approach is a cleaner, simpler language, and an extensive collection of (often complicated) standard data types and methods. The downside is that data types and operations that are important in some problem domains are no longer primitives, which adversely affects performance. This paper describes a technique called *semantic inlining* that exploits these trends in language design. Semantic inlining (applied to Java) allows standard Java classes and methods to be treated by the compiler as language primitives. A compiler performing semantic inlining in Java operates at the bytecode level (*i.e.*, the class file). Calls to these selected methods are converted, by the compiler, directly into its intermediate language (IL) representation of the program. This conversion may involve one or more of (i) annotating the IL (*e.g.*, with aliasing information), (ii) generation of specialized code in the IL for the method, or (iii) calls to specialized run-time libraries. The crux of this technique is that it allows knowledge about standard classes to be encoded within the compiler. This approach raises the abstraction level of the compiler and contrasts with the more traditional *lexical inlining* performed by many compilers. With lexical inlining, semantics of inlined methods have to be rediscovered by expensive, and often conservative, global analysis. As detailed further is Section 2, semantic inlining provides the following advantages: (i) optimizations can be performed with only local data flow information; (ii) both data and operations on the data can be efficiently optimized; (iii) the method semantics are exposed to the compiler, enabling new optimizations and analyses as well as improving

*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.

traditional optimizations and analyses; and (iv) the engineering of reference implementations of standard classes and additional Java language primitives is improved.

As an example of the performance improvements that can be achieved using semantic inlining, consider the MICROSTRIP benchmark [16] (see Section 4 for a detailed analysis of this benchmark). In the DC mode of operation, with computations performed in the real domain, Java performance is approximately 50% of Fortran. In the AC mode of operation, with computations in the complex domain, Java performance is less than 1% of Fortran. (More precisely, it is 0.85%.) Complex arithmetic in Java is performed with a `Complex` class. Using the semantic inlining techniques discussed in this paper, we were able to achieve a Java performance in AC mode of 63% of that of Fortran. This represents a speedup by a factor of 76 in this benchmark.

The major contributions of this paper are: (i) a detailed description of the concept of semantic inlining, and (ii) the demonstration through an implementation and experiments of the effectiveness of semantic inlining. This paper is organized as follows. Section 2 describes semantic inlining and its benefits in more detail, with particular emphasis on the creation of lightweight objects. It discusses semantic inlining of important standard classes. Section 3 presents the details of our semantic inlining implementation for complex arithmetic and multidimensional arrays. Our experimental work and benchmarking results are reported in Section 4. Related work is discussed in Section 5 and conclusions are presented in Section 6.

2 Semantic inlining

Semantic inlining is a compilation strategy whereby selected standard classes and methods in a language are treated as language primitives by the compiler. This enables the compiler to encode into its algorithms knowledge about the data structures and operations on that data. It is typically very difficult, or even impossible, for a compiler to reconstruct the same knowledge through global analysis of a lower-level representation. Some of the benefits that accrue from this built-in knowledge are:

- Through its understanding of the semantics of the class data and operations, the compiler can perform many optimizations locally, without global analysis. Local optimizations are typically inexpensive, and thus quite appropriate for a dynamic compiler.

- The compiler can apply transformations to both the data representation and operations on that data. (It understands both.) This allows performance difficulties caused by limitations in the language primitives used to build new data types to be overcome. The ability of the compiler to apply these transformations is not hampered by incomplete or conservative global analysis results.

- The method semantics, as expressed by the compiler in the IL, enable new optimizations and are exposed for further exploitation by standard global analysis and optimization techniques. There is no information loss from conservative inter-procedural analysis techniques or the presence of *Java Native Interface* (JNI) calls.

- The reference implementation of semantically inlined classes can focus on clarity, conciseness, and correctness. The actual inlining performed by the compiler is independent of the reference implementation, as long as the correct semantics is provided.

- Semantic inlining effectively extends the instruction set of a *Java Virtual Machine* with the semantically inlined methods. Older JVMs (or any JVM that does not support semantic inlining) can still execute the user application, since there is always a reference implementation of the methods. Note that this mechanism for extending JVMs does not use any of the small number of remaining bytecodes.

Figure 1 shows graphically the difference between semantic inlining and the more traditional, or lexical, inlining. A method call that is lexically inlined has its bytecode (or other) description inlined by the the compiler, which then represents the description as a collection of primitive IL operations. The semantic understanding of the inlined method by the compiler is, without further analysis, the disjoint understanding of each primitive. With further analysis, the semantic understanding of the compiler is a function of its understanding of these primitives and other primitives in the program. With semantic inlining, the compiler *recognizes* the method call, and directly generates an efficient IL representation of the method. The operations performed by the semantically inlined representation may be computationally simpler than those implied by the bytecode description of the method. The structure of the data, as implemented by the compiler, can also be different from the implied one. (The data structure and code generated by the compiler have to be consistent and must preserve the desired semantics). Furthermore, from its knowledge of the class and method semantics, the compiler can generate annotations in the IL that represent this knowledge. These annotations can give the compiler a comprehensive view of entire methods and of the relationships between methods in a class. This additional information available to the compiler can both directly enable other optimizations and, when properly integrated, sharpen other traditional analyses.

In the remainder of this section, we discuss how semantic inlining can be used to create *lightweight objects* in Java. We then show examples of how lightweight objects can be applied to several standard (or proposed to be standard) Java classes. The classes we discuss are a complex number class, a true multidimensional array class, and the Java `String` class.

2.1 Lightweight objects

We use the term *lightweight object* to refer to any representation of an object that is semantically rich enough to serve as the object in some program context, but is overall cheaper to create, use, and destroy. Although semantic inlining enables many different optimizations and analyses, we will discuss lightweight objects in some detail. Many of the performance deficiencies of object oriented languages relative to procedural languages follow from the treatment of objects. Therefore, the use of lightweight objects is an important class of optimizations enabled by semantic inlining. Several authors [9, 7] have advocated the explicit introduction of lightweight objects (also called *value objects*) in Java. We create lightweight objects only at the IL representation of the program, using them as replacements for regular objects when possible. Although the exact form of a lightweight object depends, in general, on both the form of the original object and the particular methods invoked on that object, we can make some generalizations. These generalizations will be made more concrete in the discussion of specific classes, below.

Let **OBJ** be some Java object. Let $obj_{\mathbf{P}}$ be a lightweight version of **OBJ** with specific properties **P**. In general, **P** is a function of the context in which the object is being used, as different lightweight representations can be more useful for different purposes. We define a function $\mathcal{L}_{\mathbf{P}}(\mathbf{OBJ})$ that returns **true** if analysis shows that all live references to object **OBJ** can use the representation $obj_{\mathbf{P}}$. $\mathcal{L}_{\mathbf{P}}(\mathbf{OBJ})$ returns **false** otherwise. We also define two conversion functions, $obj_{\mathbf{P}} = \bar{\mathcal{O}}_{\mathbf{P}}(\mathbf{OBJ})$ (which produces a lightweight object given an object) and $\mathbf{OBJ} = \mathcal{O}_{\mathbf{P}}(obj_{\mathbf{P}})$ (which yields an object given a lightweight object). For conciseness, this paper only discusses objects with a single lightweight representation. In general, different lightweight representations might exist for a given object. The set of lightweight representations for an object can be modeled by a lattice: The bottom is a lightweight object with no fields, the top is the full object, and the meet operator forms the representation that is semantically rich enough to represent the union of the properties of its arguments.

Semantic inlining can be viewed as doing a bottom up traversal of the expression trees in the intermediate representation of the program. The object methods are first converted to inlined method calls. The inlined method can replace some or all of the formal object parameters in the original method by corresponding lightweight representations. If an actual argument to an inlined method is an **OBJ**, and the corresponding formal argument is a lightweight object, then we must apply the conversion $\bar{\mathcal{O}}_{\mathbf{P}}(\mathbf{OBJ})$. Conversely, if the actual argument is an $obj_{\mathbf{P}}$ and the corresponding formal argument is a regular object, we must apply the conversion $\mathcal{O}_{\mathbf{P}}(obj_{\mathbf{P}})$.

2.2 Semantic inlining of `Complex`

In this section we discuss semantic inlining of a `Complex` class that is part of a proposal by the Java Grande Forum for consideration as a Java standard [7]. The `Complex` class represents a complex number as two `doubles`, corresponding to the *real* and *imaginary* parts. Selected methods from this class are shown in Figure 2.

The lightweight representation of an object **OBJ** of the `Complex` class is the *value complex* object $obj_{\mathtt{vcomplex}}$ of type `vcomplex`. Figure 3 shows the layout of a value complex object, which only contains the real and imaginary fields of a `Complex` object. (Note that these value objects are only visible in the IL of the compiler.) Also shown in that figure is the layout of a `Complex` object **OBJ**. Every `Complex` object contains a value object inside it. It also contains all the object description data, present in every Java object, that we refer to as the *junk* field. Conversion of a `Complex` object to a `vcomplex` object via the $\bar{\mathcal{O}}_{\mathtt{vcomplex}}()$ function requires only extracting the address of the data field portion of the object. Conversion of a `vcomplex` object to a `Complex` object, through the $\mathcal{O}_{\mathtt{vcomplex}}()$ function, is trivial if the lightweight object is part of a regular object. If that is not the case, then the target object has to be created first and the data fields have to be copied from the `vcomplex` object to the `Complex` object.

Figure 4 shows the corresponding lightweight methods for each of the Complex methods in Figure 2. The lightweight methods take parameters of type `vcomplex` (the actual value of the object) or `vcomplex&` (a reference, or pointer, to the object). Return parameters can also be values (`vcomplex`) or references (`vcomplex&`). From the semantics of the methods in Figure 2, we know that no aliases are created for input parameters in any of those methods. Any `Complex` object **OBJ** that is only used as parameters to those methods has $\mathcal{L}_{\mathtt{vcomplex}}(\mathbf{OBJ}) =$ **true**, and can be replaced by a `vcomplex` object. By construction, intermediate results in arithmetic expressions are always `vcomplex` objects. The practical implication is that just by analyzing statements within a single method, and without any global information, many `Complex` object creations can be changed to to the (cheaper) creation of `vcomplex` objects. This is verified by the results shown in Section 4.

We show a detailed example in Figure 5. A Java code fragment and its original expression trees are shown in Figure 5(a) and Figure 5(b), respectively. Let those be the only references to `P` and `Q` in the method being compiled. Semantic inlining can replace all method calls by their lightweight equivalents, as shown in Figure 5(c), except for the `display` method. That method has no lightweight equivalent and expects an `Object` as a parameter. Because of the unknown (to the compiler) behavior of `display(Object)`, $\mathcal{L}_{\mathtt{vcomplex}}(\mathtt{P})$ is **false**. Therefore `P` remains a `Complex` object and is created with the regular `Complex` constructor. $\mathcal{L}_{\mathtt{vcomplex}}(\mathtt{Q})$ is **true**, since `Q` is only used in `Complex` methods. `Q` can be transformed into a lightweight object and the `Complex` constructor for `Q` is replaced by a `vcomplex` constructor. The call to `times` takes two arguments, a `vcomplex` and a `double`. Since `P` is a `Complex` object, it has to be converted to `vcomplex` by the application of $\bar{\mathcal{O}}_{\mathtt{vcomplex}}(\mathtt{P})$. (This operation is just

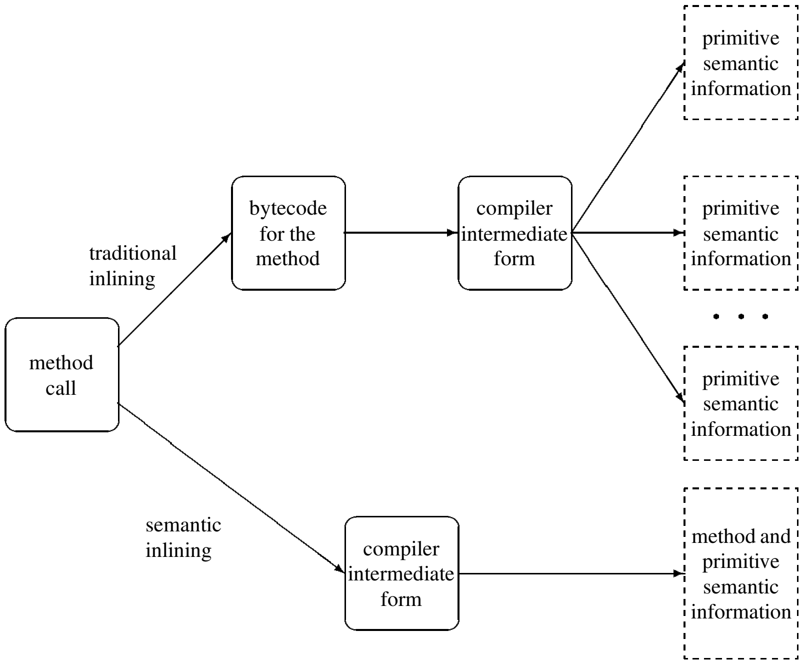


Figure 1: Overview of semantic inlining.

```
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 static Complex plus(Complex x, Complex y) {
        return new Complex(x.re+y.re,x.im+y.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 plusEquals(Complex z) { re += z.re; im += z.im; return this; }
}
```

Figure 2: A standard Java class for complex arithmetic.

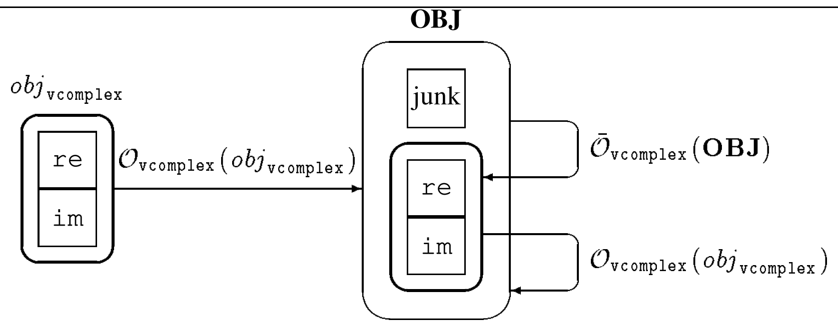


Figure 3: `Complex` value object description.

a pointer manipulation.) The vcomplexes returned by times and Q can be directly used as parameters to plus. Finally, the vcomplex returned by plus and P, once again converted by $\bar{\mathcal{O}}_{\texttt{vcomplex}}(\texttt{P})$, are used in assign. Note that we have optimized from four object creations in the original code of Figure 5(a) to a single object creation (of P in Figure 5(b).

2.3 Semantic inlining of true multidimensional arrays

The *multidimensional array package* is also part of the Java Grande Forum proposal. Classes in this package implement rectangular arrays of various ranks and types. For example, doubleArray3D implements a three-dimensional array of doubles and ComplexArray2D implements a two-dimensional array of Complex. The array classes support Fortran 90-like functionality, such as efficiently extracting subsections of arrays. Figure 6 shows the conceptual difference between a two-dimensional rectangular array $A(1 : m, 1 : n)$ and a Java array of arrays $A[1 : m][\,]$. (We take the liberty of starting indexes at 1 for convenience.) Note that each row of $A[1 : m][\,]$ can be be of a different length and that, although not shown in the figure, rows can be aliased to one another.

For compilers that understand their semantics, multidimensional array classes offer more opportunities for optimization than regular Java arrays. Because the shape of an array is rectangular and immutable after its creation, bounds checking optimization and aliasing analysis is greatly facilitated. With regards to bounds checking, the transformations described in [15, 17] can be applied to a two-dimensional rectangular array without the need to pre-compute the minimum length of rows. Also, the privatization of arrays to a thread's *working memory* [10] (a crucial step for thread safety of those transformations with Java arrays) is unnecessary. Aliasing information is easier to extract with multidimensional rectangular arrays. There can be no aliasing between two rows of the same array or between a row from an array A and a row from a different array B.

Figure 7 shows selected methods in the ComplexArray2D class. Each complex value is stored as a pair of doubles, in corresponding real and imaginary arrays of doubles. Access to elements is exclusively through set and get methods, which hide this implementation from the user. The set and get methods accept and return regular Complex objects. (In reality, the set and get methods are more complicated than that, as they operate on any array section.)

The structure for a ComplexArray2D object A, as defined by Figure 7, is shown in Figure 8(a). (Only the real array is shown.) Note that this layout is not appropriate for traversal along columns, and that storing or retrieving an element requires access to two distinct arrays of doubles, resulting in poor memory locality. Furthermore, the get operation is expensive, as it requires an object creation. In the lightweight version of the ComplexArray2D object, the set and get methods are modified to retrieve and store vcomplex objects. Also, the storage layout is changed to a row-major and dense collection of vcomplex objects, as shown in Figure 8(b). Each vcomplex object is stored as a pair of consecutive double values. The set and get methods can be implemented as Fortran-like indexing operations into the dense storage. (Actually, it is more C-like, since it is row major.)

By combining semantic inlining of Complex and ComplexArray2D objects we get a synergetic effect: Computations with array elements never require the creation of a Complex object, since intermediate results are always vcomplex as well. The only Complex objects that must be left in a Java code after semantic inlining are Complex scalars that cannot not be transformed because they are treated as an object in some context the compiler does not understand (*e.g.*, passed to an unknown method).

The ComplexArray2D class is an example where semantic inlining allows the library designer to focus on

`Complex` methods	`vcomplex` methods
`Complex(double r, double i)`	`vcomplex& vcomplex(double r,double i)`
`Complex assign(Complex z)`	`vcomplex& assign(vcomplex& this, vcomplex z)`
`Complex plus(Complex z)`	`vcomplex plus(vcomplex this, vcomplex z)`
`static Complex plus(Complex x, Complex y)`	`vcomplex plus(vcomplex x, vcomplex y)`
`Complex times(Complex z)`	`vcomplex times(vcomplex this, vcomplex z)`
`Complex times(double x)`	`vcomplex times(vcomplex this, double x)`
`Complex plusEquals(Complex z)`	`vcomplex& plusEquals(vcomplex& this, vcomplex z)`

Figure 4: Equivalent `Complex` and `vcomplex` methods.

```
S1  Complex P = new Complex(0,0);
S2  Complex Q = new Complex(1,1);
S3  P.assign(Q.plus(P.times(4.0)))
S4  display(P);
```

(a) A Java program fragment.

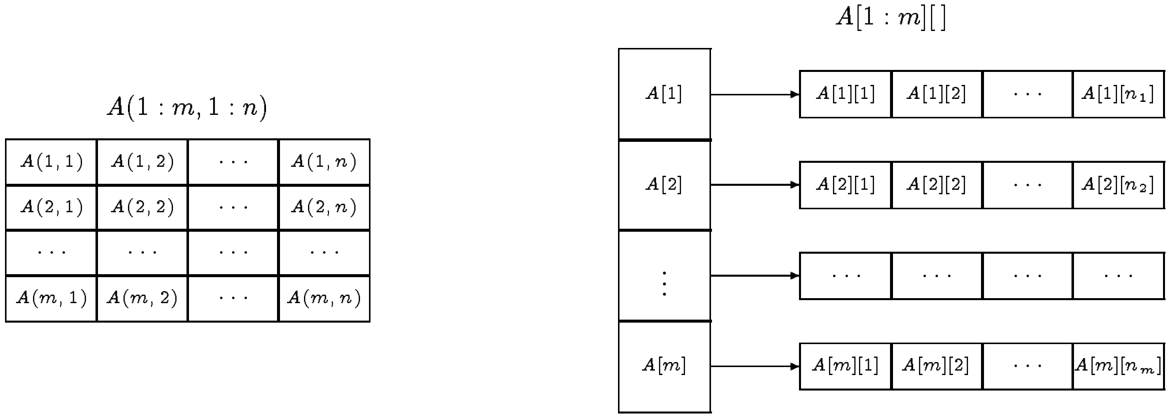
```
                =                                        =
             ↗     ↖                                  ↗     ↖
          P           Complex(0,0)                Q           Complex(1,1)

P.assign(Complex)
           ↑
         Complex Q.plus(Complex)
                         ↑
                       Complex P.times(double)
                                        ↑
                                       4.0

display(Object)
          ↑
          P
```

(b) Expression trees before inlining.

```
                =                                        =
             ↗     ↖                                  ↗     ↖
          P           Complex(0,0)                Q           vcomplex(1,1)

assign(vcomplex&,vcomplex)
 Ō_vcomplex(P) ↑          ↑
          P      vcomplex plus(vcomplex,vcomplex)
                                    ↑        ↑
                                    Q     vcomplex times(vcomplex,double)
                                                 Ō_vcomplex(P) ↑        ↑
                                                               P       4.0

display(Object)
          ↑
          P
```

(c) Expression trees after inlining.

Figure 5: Example of semantic inlining for `Complex` class.



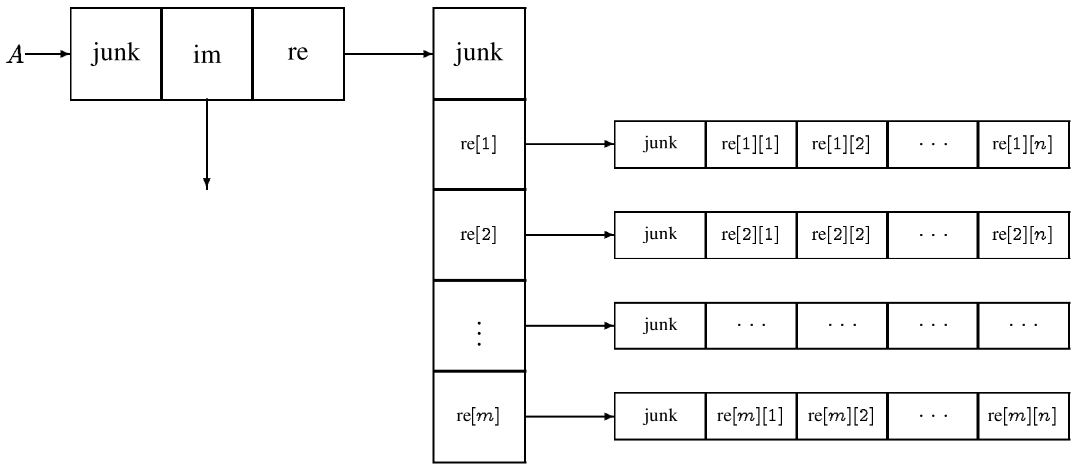
(a) $m \times n$ array

(b) m array of arrays

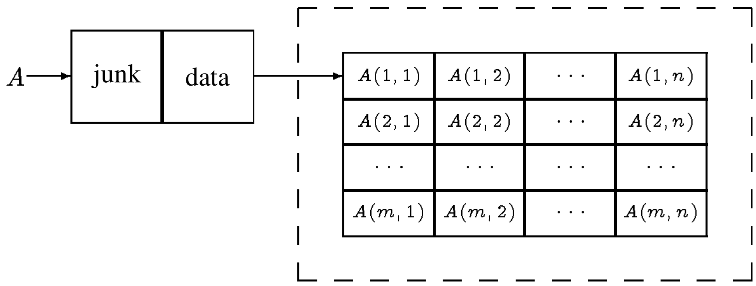
Figure 6: An $m \times n$ rectangular array *vs.* an array of arrays.

```
public final class ComplexArray2D {

   double[][] re;
   double[][] im;

   ComplexArray2D(int m, int n) {
       re = new double[m][n];
       im = new double[m][n];
   }
   public Complex get(int i, int j) { return new Complex(re[i][j],im[i][j]); }
   public void set(int i, int j, Complex z) { re[i][j] = z.re; im[i][j] = z.im; }
}
```

Figure 7: A (proposed) standard Java class for two-dimensional complex array.



(a) Reference implementation form of the object.



(b) Semantically inlined form of the object.

Figure 8: Multidimensional array objects.

clarity and conciseness of a definition. It would have been more efficient to use a single array of x of $2mn$ doubles to represent all complex values, with $x[2(im+j)]$ and $x[2(im+j)+1]$ storing the real and imaginary fields of complex element (i,j). This would simplify memory reference and facilitate access along columns. (The `get` method would still require an object creation.) However, the semantics of such an implementation are not as clean. The maximum number of elements would have been $\sim$ 1 billion (2^{31} `double` fields). Also, the maximum allowable extent along one axis would be a function of the extent along the other axis. The bounds checking semantics would also be undesirable, since invalid i and j indices (along their respective axes) can still result in a valid index into x. The reference implementation of Figure 7 allows the maximum extent of any axis to be the same ($2^{31}-1$) and independent of the other axis. Each index also has to be valid along its corresponding axis to result in a valid reference. The discussion presented here for two-dimensional arrays of `Complex` also applies to arrays of other ranks and types.

2.4 Semantic inlining of *String* functions

In the interest of conciseness, our description of the semantic inlining of this class is briefer than the previous two. String manipulation in Java is provided by a standard class, but is not efficient when compared to hand-tuned C code. Much of the inefficiency derives from the compiler blindly applying the "+" operator to strings without any knowledge of the underlying implementation. Thus the operation $\mathtt{S} = \mathtt{s}_1 + \mathtt{s}_2 + \mathtt{s}_3$, where $\mathtt{S}$, $\mathtt{s}_1$, $\mathtt{s}_2$ and $\mathtt{s}_3$ are strings, is naively implemented in some JVM's as:

$$\mathtt{s}_{\mathtt{tmp}} = \mathtt{s}_1.\mathtt{concat}(\mathtt{s}_2)$$
$$\mathtt{S} = \mathtt{s}_3.\mathtt{concat}(\mathtt{s}_{\mathtt{tmp}})$$

which copies the characters in $\mathtt{s}_1$ and $\mathtt{s}_2$ twice. Similarly, $\mathtt{s}_1.\mathtt{concat}(\mathtt{s2}).\mathtt{length}$ need not form the concatenation of $\mathtt{s}_1$ and $\mathtt{s}_2$ to return its length. We are currently working on the implementation of lightweight objects for `String`. By integrating known semantic information about the `String` class into our compiler, optimizations such as the ones we discussed will be possible [20].

3 An implementation of semantic inlining

In this section, we describe a prototype implementation of semantic inlining for the `Complex` and multidimensional array classes. We picked these classes as our first targets for optimization for several reasons, including: (i) `Complex` allows us to test the concept of semantic inlining on a numeric type, with the experience gained being applicable to the *decimal* types that are ubiquitous in commercial and financial codes; (ii) the multidimensional array type is fundamental to numeric computing, and success in this area should improve almost all computationally intensive programs. It also highlights the freedom the compiler has with semantic inlining in altering both the data representation and operator implementation.

Our prototype implementation is based on the IBM High Performance Compiler for Java (HPCJ) [3, 21], 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. The actual semantic inlining was implemented 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 seats in between the HPCJ front-end and the, architecture specific, code generation back-end.

3.1 Implementation of semantic inlining for `Complex` class

Our implementation of semantic inlining for the `Complex` class differs from the general discussion in Section 2.2 in two ways. First, the computation of the $\mathcal{L}_{\mathtt{vcomplex}}$ function, that determines if a lightweight object can be used to represent an object, is very conservative. It uses *no* data-flow information about objects other than that implied by individual `Complex` method semantics. As an effect, $\mathcal{L}_{\mathtt{vcomplex}}(\mathbf{OBJ}) = \mathbf{true}$ only for temporary objects used strictly as inputs (not modified) in inlined methods. In particular, when the `vcomplex` result of a method call is assigned to an object, the lightweight object will always be converted to a `Complex` object through the $\mathcal{O}_{\mathtt{vcomplex}}()$ function. In the example of Figure 5, this conservative analysis would prevent `Q` from being allocated as a `vcomplex` object, and would require an additional conversion from `Complex` to `vcomplex` when `Q` is passed to `add`. Our performance results, presented in Section 4 correspond, therefore, to a lower bound on the performance that can be obtained by integrating semantic inlining with better data flow analysis.

The second difference in the implementation is that the function $\bar{\mathcal{O}}_{\mathtt{vcomplex}}()$ is *not* implemented by passing in the base of the data field, but rather by copying the object field values into a temporary `vcomplex` object. (This is legal since we only use lightweight objects as strictly input parameters.) This again makes our experimental results conservative.

Finally, as an additional benefit of semantic inlining, we are able to map the `vcomplex` object directly into the W-code *complex* data type, thereby exposing the `vcomplex` objects to the full range of the TPO and back-end optimizations on complex arithmetic. Most of the arithmetic operations can be represented as a single W-code instruction on complex data, and most `vcomplex` objects can be allocated in registers.

3.2 Implementation of semantic inlining for multidimensional array classes

The implementation of the array class is straightforward, and creates a true dense multidimensional array. The semantically inlined array constructor allocates a contiguous chunk of storage large enough to hold all of the elements of the array. The inlined constructor for `ComplexArray2D` returns a lightweight version of `ComplexArray2D` that behaves (at the interface level) like the reference implementation. (Only the internal organization is different.) Therefore, the conversion from array objects to their lightweight counterpart is always valid. The `get` and `set` methods are semantically inlined to Fortran-like indexing operations into the data storage allocated by the constructor. These methods operate on W-code data elements of type complex, which in turn can be used in the semantically inlined version of the `Complex` arithmetic operators. (See Section 3.1.)

4 Experimental results

In the Introduction, we briefly mentioned results from the MICROSTRIP benchmark. We saw that a Java implementation of the DC mode of that benchmark performed quite well in comparison with the Fortran version. However, in AC mode the Java version executed 120 times slower. In this section we analyze in more detail the behavior of MICROSTRIP and other 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 with a production version of the IBM XLF compiler with the highest level of optimization (-O3 -qhot). 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 [17]. In accordance to current Java floating-point semantics, the `fma` (fused multiply-add) instruction of the POWER architecture is disabled for Java programs but enabled for Fortran 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.

4.1 MICROSTRIP

The MICROSTRIP benchmark [16] computes the value of the potential field $\Phi(x, y)$ in a two-dimensional microstrip structure [6]. 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 [16].) 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, an 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.) 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. Java does not provide a complex primitive type. The `Complex` class is used to implement complex arithmetic and Equation (1) 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)

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 9(a) (for DC mode) and Figure 9(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 show in Figure 9 are for a smaller problem size, with $w = h = 99$.)

By semantic inlining the methods of the `Complex` class, we eliminate the need for creation and destruction of temporary objects. All arithmetic is performed on complex values rather than `Complex` objects. The semantic inlining raises the performance of the Java version of MICROSTRIP AC from 1 to 46 Mflops. Further enhancements can be obtained by replacing Java arrays of `Complex` objects by `ComplexArray2D` objects that are also inlined. In that case, the code for Equation (1) in Java becomes

```
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)
      )
```

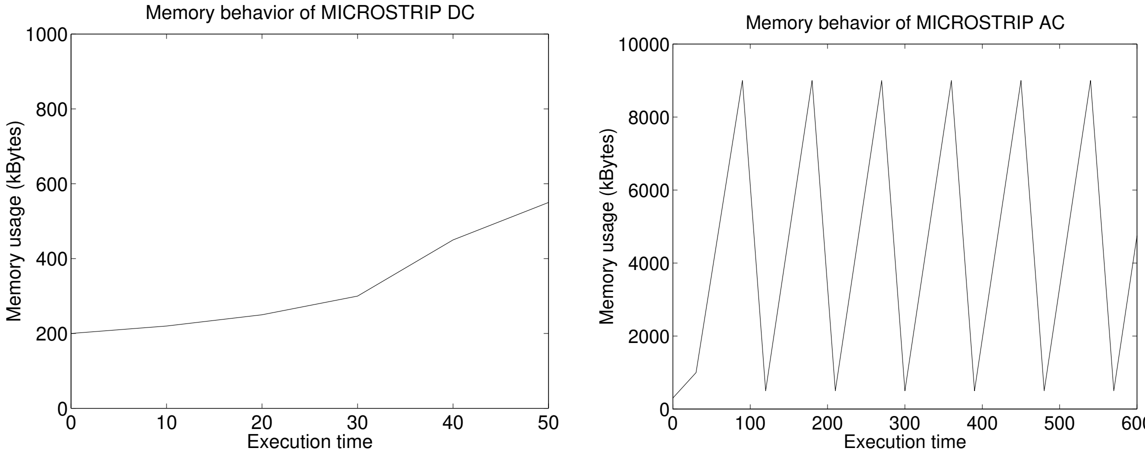
(5)

Multidimensional arrays of complex values offer several benefits over conventional Java arrays of `Complex` objects, which have been discussed in Section 2. In the particular case of MICROSTRIP AC, these benefits translate into a performance improvement to 76 Mflops for the Java version, representing a 76-fold performance improvement over the original Java code. The results for MICROSTRIP AC are summarized in Figure 10. The bar labeled plain shows the performance of the standard Java implementation of this benchmark. The complex bar shows the result after semantic inlining of the `Complex` class. The array bar shows the result from using the semantically inlined array class. Finally, Fortran is the result for the Fortran version of this benchmark, with the highest level of optimization.

4.2 Other benchmarks

We complete the experimental evaluation of our techniques by applying semantic inlining to four additional benchmarks. All of the benchmarks are numerical codes with intensive 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, to improve performance. LU is a straightforward implementation of Crout's algorithm for performing the LU decomposition of a square matrix A [8], 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 [19] 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 11. 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 inlined. 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 inlined 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. 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



(a) Memory usage in DC mode

(a) Memory usage in AC mode

Figure 9: Memory utilization in MICROSTRIP.

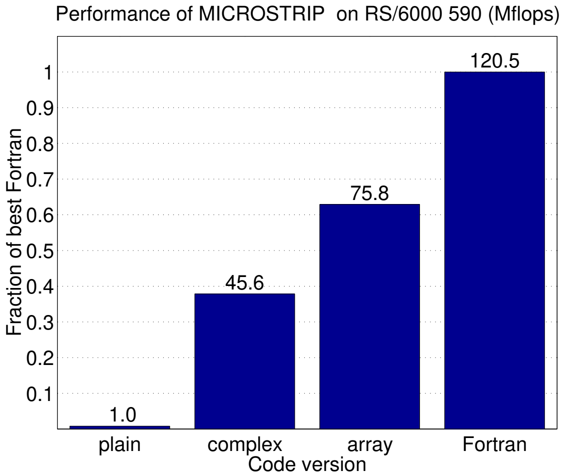


Figure 10: Summary of results for MICROSTRIP AC.

proposal for extending the floating-point semantics of Java, that would allow it to use the `fma` instruction, is currently under public review [14].

5 Related work

Most of the early work on inlining [18] concentrated on lexical inlining of functions and subroutines. Section 2 discusses in some detail the differences between our techniques for semantic inlining and this more traditional lexical inlining.

Our semantic inlining 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 inlining 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[22] targets standard container classes for semantic inlining. In their paper, the inlined 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 [5] (targeting C++) and the work of Hall, *et.al.* [11] (targeting Haskell) is of interest to us in that it appears to duplicate our results with complex numbers, albeit in a different environment. In [5], 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 [11] 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.

The strength of these techniques relative to ours is that they operate on user defined data types as well as standard types. Their disadvantage is that they require global analysis, which our methods do not, making them relatively more expensive in compile time. Their methods will also fail if the source code for the method is not available at compile time, if operations on the objects are implemented in another language (*e.g.*, via JNI calls), or if the analysis is conservative. Our techniques also have the advantage of allowing types to be converted, so that even if the optimization is globally illegal, it can still be performed locally. Given that Fortran represents close to a practical upper bound on the gain from these optimizations, we have achieved most of the potential gain with little compile time complexity.

Object representation in the Emerald system is described in [12]. Emerald supports distributed objects, and with type inferencing analysis they can determine that objects are *local* (*i.e.*, not distributed) and can therefore be represented more efficiently. They recognize *primitive* objects, which are "small and immutable". It is unlikely their handling of primitives could do the sort of transformations we perform on multidimensional arrays. They do not do method substitutions to enhance the possibility of optimizations, and do not incorporate semantics from more complicated operations directly into their IL.

In [2], Cierniak and Li describe a global analysis technique for determining that the shape of a Java array of arrays 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.

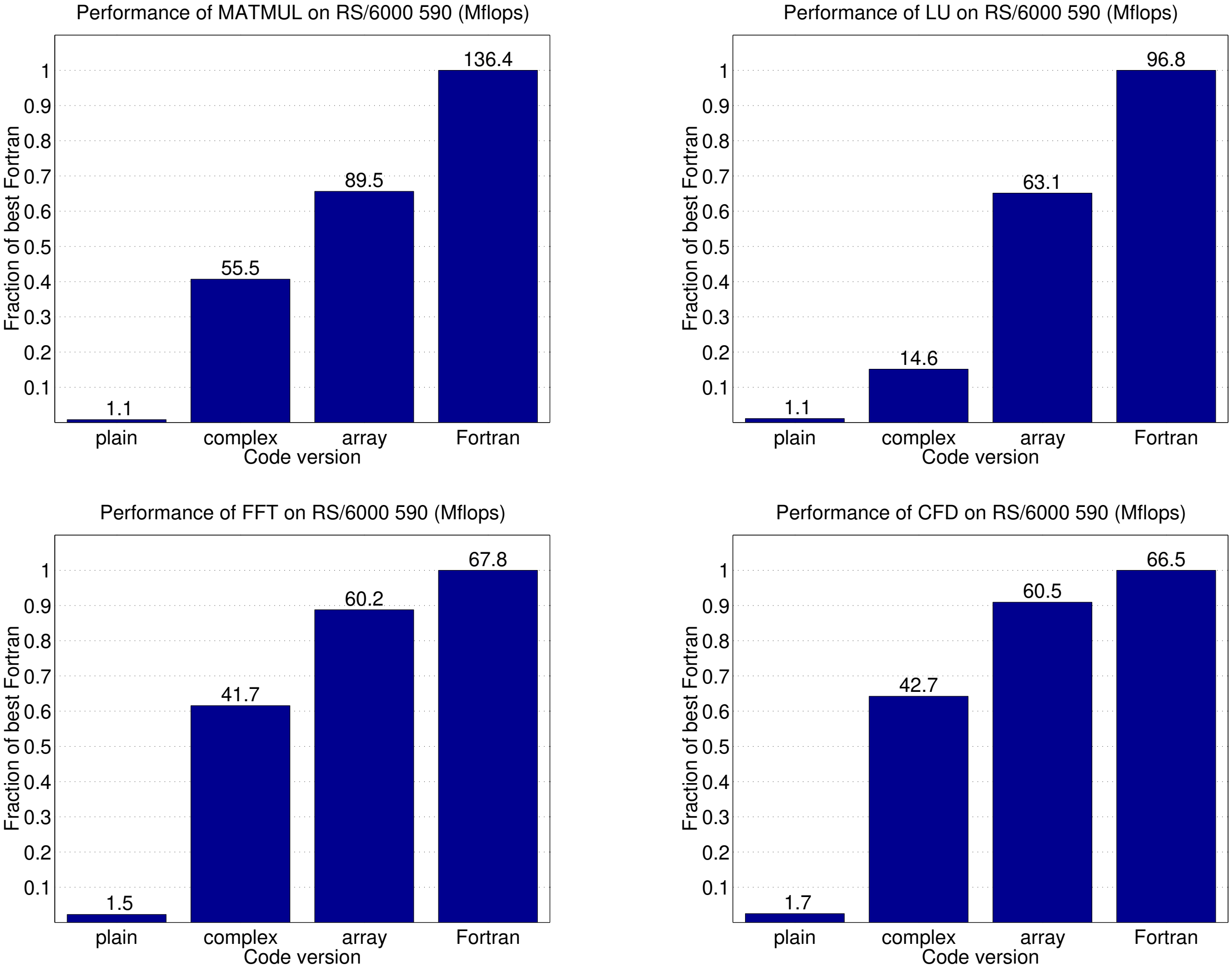


Figure 11: Summary of experimental results.

6 Conclusions and future work

We have described *semantic inlining*, a technique that allows dynamic or static compilers to dramatically improve the performance of Java programs with almost no overhead for analysis. We achieve this result by extending the compiler with knowledge of the semantics of important standard classes and methods. This allows the compiler to generate intermediate code for the methods that represent this (possibly global) knowledge. The method semantics can also be integrated into the compiler's analysis algorithms. If the compiler did not treat the standard method calls as primitives, it would be necessary to recover this information via more expensive global analysis – global analysis that might not even be possible, depending on the implementation and compile-time availability of these methods.

Semantic inlining is, in practice, very fast at compile time, making it applicable to both static and dynamic compilation systems. We measured the effectiveness of semantic inlining through implementations for proposed standard `Complex` and multidimensional array classes. In numerical benchmarks with complex arithmetic we achieve up to 90% of the performance of fully optimized Fortran, and speedups of up to 80 relative to compiled Java code without semantic inlining (but fully optimized otherwise).

We are starting to diversify our research from just the realm of technical computing. We are currently designing and implementing semantic inlining for the Java `String` class, input/output operations, and for a *decimal* numeric class. String and I/O operations are pervasive in various types of applications, and decimal arithmetic is important for commercial applications.

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] American National Standards Institute, New York, New York. *American National Standards Programming Language PL/I, X3.53*, 1976.

[2] 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.

[3] 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`.

[4] IBM Corporation. IBM San Francisco, 1998. Information available in Web page at `http://http://www.ibm.com/Java/Sanfrancisco`.

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

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

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

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

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

[10] James Gosling, Bill Joy, and Guy Steele. *The Java(TM) Language Specification*. Addison-Wesley, 1996.

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

[12] N.C. Hutchinson. *Emerald: An Object-Based Language for Distributed Programming*. PhD thesis, University of Washington, 1987.

[13] M. Jacob, M. Philipsen, and M. Karrenbach. Large-scale parallel geophysical algorithms in Java: A feasibility study. *Concurrency: Practice & Experience*, 10, 1998.

[14] Sun Microsystems. Sun proposes modification to Java programming languages's floating point specification. `http://www.sun.com/smi/Press/sunflash/9803/sunflash.980324.17.html`, 1998.

[15] S.P. Midkiff, J.E. Moreira, and M. Snir. Optimizing bounds checking in java programs. *IBM Systems Journal*, 37(3):409–453, August 1998.

[16] 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.

[17] 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.

[18] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufman, 1997.

[19] 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.

[20] Vivek Sarkar. Private conversation, October, 1998.

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

[22] 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.