# High Performance Numerical Computing in Java: Language and Compiler Issues

P. V. Artigas M. Gupta S. P. Midkiff J. E. Moreira
{*artigas,mgupta,smidkiff,jmoreira*}*@us.ibm.com*

IBM Thomas J. Watson Research Center
Yorktown Heights NY 10598-0218

**Abstract**

Poor performance on numerical codes has slowed the adoption of Java within the technical computing community. In this paper we describe a prototype array library and a research prototype compiler that support standard Java and deliver near-Fortran performance on numerically intensive codes. We discuss in detail our implementation of: (i) an efficient Java package for true multidimensional arrays; (ii) compiler techniques to generate efficient access to these arrays; and (iii) compiler optimizations that create safe, exception free regions of code that can be aggressively optimized. These techniques work together synergistically to make Java an efficient language for technical computing. In a set of four benchmarks, we achieve between 50 and 90% of the performance of highly optimized Fortran code. This represents a several-fold improvement compared to what can be achieved by the next best Java environment.

## 1 Introduction

Despite the advantages of Java$^{TM}$[1] as a simple, object oriented programming language, it has not been widely adopted within the technical computing community. The primary reason for this is obvious: the performance of technical computing programs written in Java, and executed with currently available commercial environments, trails far behind the equivalent Fortran programs. In this paper we discuss a prototype array library and a prototype research compiler developed by the Numerically Intensive Java group at IBM Research that allows standard Java programs to achieve Fortran-like performance. These techniques deliver several-fold speedups on uniprocessor codes. They make Java a practical language for numerical computing.

Why has Java performance in the domain of technical computing, up to now, been so poor? There are several reasons which work together to degrade Java performance. First, three very useful Java features – (i) array reference and `null`-pointer checks; (ii) using objects to represent all but the most widely used numeric types; and (iii) the structure of Java arrays – are extremely detrimental to performance in the domain of numerical computing [20]. Second, much compiler research on Java has focused on optimizations for more general applications, leading to naive (from the point of view of technical computing) implementations of the features mentioned above. Third, these naive implementations lead to poor performance, reinforcing the perception that Java is not appropriate for technical computing and reducing the emphasis on improving Java performance for numerical applications.

We will now give an overview of the detrimental effects of these language features. The rest of the paper will discuss in detail our library and compiler approach and how it overcomes these detrimental effects. Our

[1] Java is a trademark of Sun Microsystems Inc.

approach is an example of language and compiler codesign. Although we make absolutely no modifications to the Java language *per se*, we introduce a class library (a package) for multidimensional arrays and advocate its use in developing technical codes. (An implementation of the Array package is available for free download from `http://www.alphaworks.ibm.com`. More information on our research can be found at `http://www.research.ibm.com/ninja`.) In effect, we *grow* the language without having to modify any part of it. The Array package has been designed to enable powerful compiler optimization techniques that significantly improve the performance of technical computing. A key enabling compiler technique is *semantic expansion*, which involves substituting calls to methods with known semantics by equivalent code that implements the same semantics (*i.e.*, treating standard methods as language primitives) [31]. Semantic expansion is discussed further in Section 3.2.

**Array references and `null`-pointer checks:** The Java language specification requires that references through a pointer first check that the pointer is not `null`. If it is `null`, a run-time exception is thrown. Java also requires that each array access be checked to ensure that the element being referenced is within the declared bounds of the array. Both of the features help enforce security and reliability of Java programs. On many architectures, the direct cost of the checks is quite low for valid references. The real problem arises from the combined effects of the required checks and Java precise exception model [24]. Because Java strictly specifies the order of evaluation of expressions – including accesses of expression operands – an exception must appear to have been thrown exactly after previous accesses and operations have been performed in strict program order. This prohibits the reordering of accesses and operations that are the foundation of the aggressive optimizations that make Fortran a high performance language [28]. Our general solution is to create different static instances of the loop body, with and without bounds and `null`-pointer exceptions. The proper static instance is dynamically selected to execute the iteration space of the loop. The static instance of the loop body which has no exceptions can be aggressively optimized. Details of our implementation are discussed further in Section 3.3.

**Efficient array structures:** Java native arrays (which will be referred to as *Java arrays*, with a lower case first *a*) allow the construction of very general random access data structures. Unfortunately, this same generality makes the the array reference checking optimizations described above more expensive, and dataflow analysis (pointer alias analysis and dependence analysis) necessary for aggressive optimization nearly impossible. Consider the example in Figure 1, which illustrates several problems with Java arrays. First, general optimization of bounds checking requires finding out the extent of all rows of a Java two-dimensional array. This is an $O(n)$ operation, where $n$ is the number of rows. If the array was known to be rectangular, it would be an $O(1)$ operation – the reading of a value in the array descriptor. Second, different rows of the array can be aliased (*e.g.*, rows 2 and $n-2$ of $A$) even though their indices have different values. This is an example of intra-array aliasing. It is also possible to have inter-array aliasing as shown in Figure 1 for rows 0 of $B$ and $A$. Thus, pointer alias analysis, not the cheaper dependence analysis, is necessary for optimization. Because of the dynamic nature of Java arrays, the alias analysis is very difficult. Our solution to the problem with Java arrays is the creation of an Array package for Java. This Array package implements multidimensional rectangular *Arrays* (with a capital first *A*) which are necessary for technical computing. The Array package is described in more detail in Section 2. When combined with the two compiler optimizations described in Section 3, semantic expansion and bounds checking elimination, this package overcomes both the dataflow analysis problems and the array reference bounds checking problems.

The Array package for Java and its associate compiler optimizations create an environment that delivers both the functionality and performance of Fortran 90 programming in Java. (This approach is similar to other successful object-oriented numerical computing environments such as POOMA and A++/P++ [27,

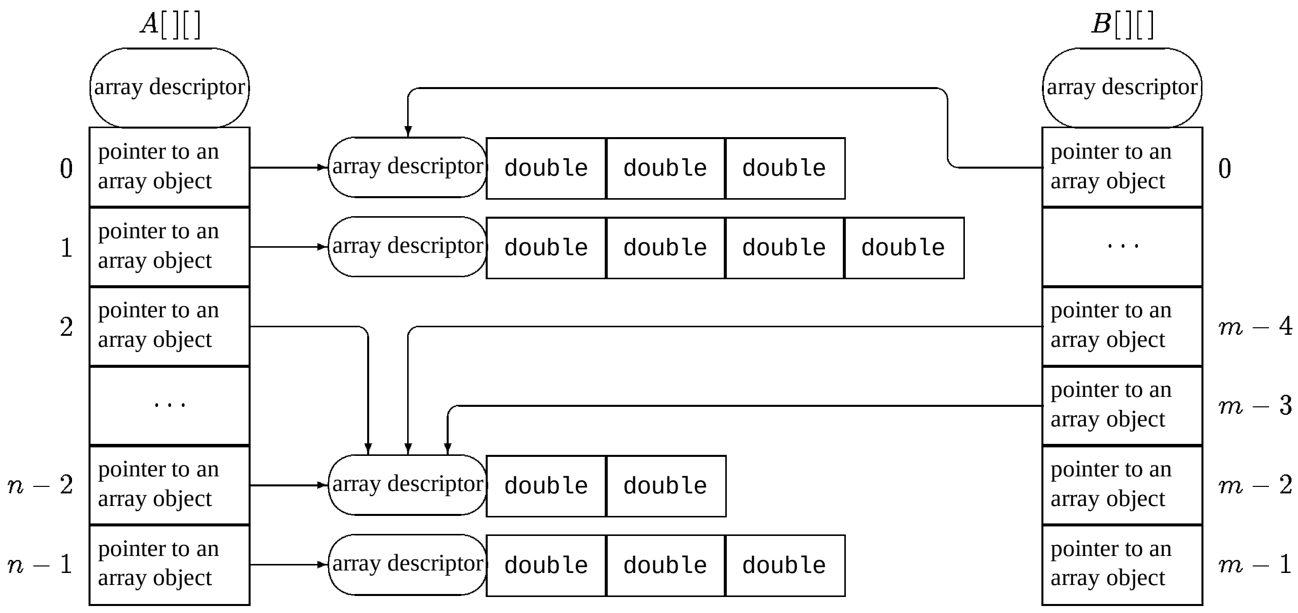

Figure 1: Structure of two-dimensional Java arrays.

25, 30].) We show in Section 4 that we can, using completely automatic compiler optimizations, raise the performance of 100% pure Java from a mere 1-2% to 65-90% of highly optimized Fortran. Even when compared to the highest performing Java environment (JDK 1.1.6 with the IBM JIT) currently available for our platform, which already incorporates advanced optimizations such as bounds checking elimination, we still show a several-fold improvement in performance. We note that the Fortran codes we compare Java against are representative of truly optimized computations, that achieve between 35-90% of the hardware peak performance.

The rest of this paper is organized as follows. Section 2 describes our Array package for Java, discussing its main components and presenting an example of its usage. Section 3 describes the compiler optimizations that we implemented in conjunction with the Array package. Section 4 reports our experimental results from using the Array package with four numerical benchmarks. Finally, Section 5 discusses related work and Section 6 presents our conclusions.

## 2 The Array package for Java

A major design goal for the Array package was to overcome the limitations inherent in Java arrays in terms of performance and ease of use, without losing any key benefits of Java arrays or requiring any changes to the language semantics. The Array package provides the following benefits:

- *more expressivity*: the package allows programmers to use high-level Array operations (like Array addition, transpose), which are similar to those provided by languages like Fortran 90.
- *high performance*: many design features allow programs using the Array package to achieve high performance: (i) the high-level array operations are written in a transactional style, discussed below, that enables aggressive compiler optimizations in spite of various exception checks; (ii) a high-performance BLAS (Basic Linear Algebra Subprograms) library is available as part of the Array

package for common linear algebraic computations; and (iii) the basic Array access operations are amenable to efficient implementation and accurate compiler analysis, when used in conjunction with the semantic expansion technique; for instance, the compiler would know that two rows of an Array cannot be aliased with each other.

- *safety and security features of Java preserved*: the requirements, mandated by Java semantics, for bounds checking and `null`-pointer checking are preserved; in fact, the Array package requires extensive checks for other exceptional conditions, like nonconforming Arrays for high-level Array operations, that arise in the context of the operations introduced in the package.

- *flexibility of data layout*: the actual data layout for the Arrays is not exposed to the programmer (as it is not specified). While this may prevent the programmer from doing certain optimizations, we believe this is beneficial in the longer term because it facilitates data layout optimizations for Arrays [10, 18, 21, 22, 14]. The compiler has fewer constraints on ensuring the correctness of the program in the presence of data layout transformations, and can avoid copy operations otherwise needed to restore the data layout to a fixed format.

The class hierarchy of the Array package is shown in Figure 2. It has been defined to enable aggressive compiler optimizations. There is a separate class for each Array *elemental data type* and *rank* (currently, we support ranks 0 through 3). There are also separate classes for Arrays of complex numbers and Arrays of `Objects`. This approach enables easy static type checking of Arrays defined by the Array package. We make extensive use of *final* classes, since most Java compilers (and, in particular, *javac*) are able to apply method inlining to methods of final classes. Since the type and rank of an Array are defined by a final class, the semantics of any Array operation (and in particular element access) are defined statically. This is a key feature exploited in our optimization techniques. Obviously, by making the Array classes final we prevent application programmers from extending those classes directly. A *decorator* design pattern [15] can be used if programmers want to "extend" Array classes.

In contrast to Java arrays discussed in the previous section, an Array defined by the Array package has a nonragged rectangular shape and constant bounds over its lifetime. An axis of this Array can be indexed by either an integer (specifying a single element), a `Range` (specifying a triplet), or an `Index` (specifying an enumerated list). Indexing operations are used in *accessor methods* that read/write data from/to an Array, and in *sectioning methods* that create views of an Array. A valid index for an Array axis must be greater than or equal to zero and less than the extent of that axis.

All operations (methods and constructors) of the Array package have, in the absence of JVM failure, transactional behavior. That is, there are only two possible outcomes of an Array operation: (i) the operation completes without an exception being thrown, or (ii) an exception is thrown before any data is changed. This allows aggressive program transformations in the main computational part of the method, while honoring the Java exception semantics.

## 2.1 Defined methods

The Array package defines four groups of methods that are always present in any Array package class: Array operation, Array manipulation, Array accessor, and Array inquiry. In addition, there is a collection of BLAS routines for operations on floating-point data.

- *Array operations*: these are scalar operations applied element-wise to a whole Array. The methods in this category include assignment, arithmetic and arithmetic-assign operations (analogous, for example, to `+=` and `*=` operators in Java), comparison operations, and logic operations (where appropriate for the elemental data type). These operations are further subdivided as *Scalar-Array*

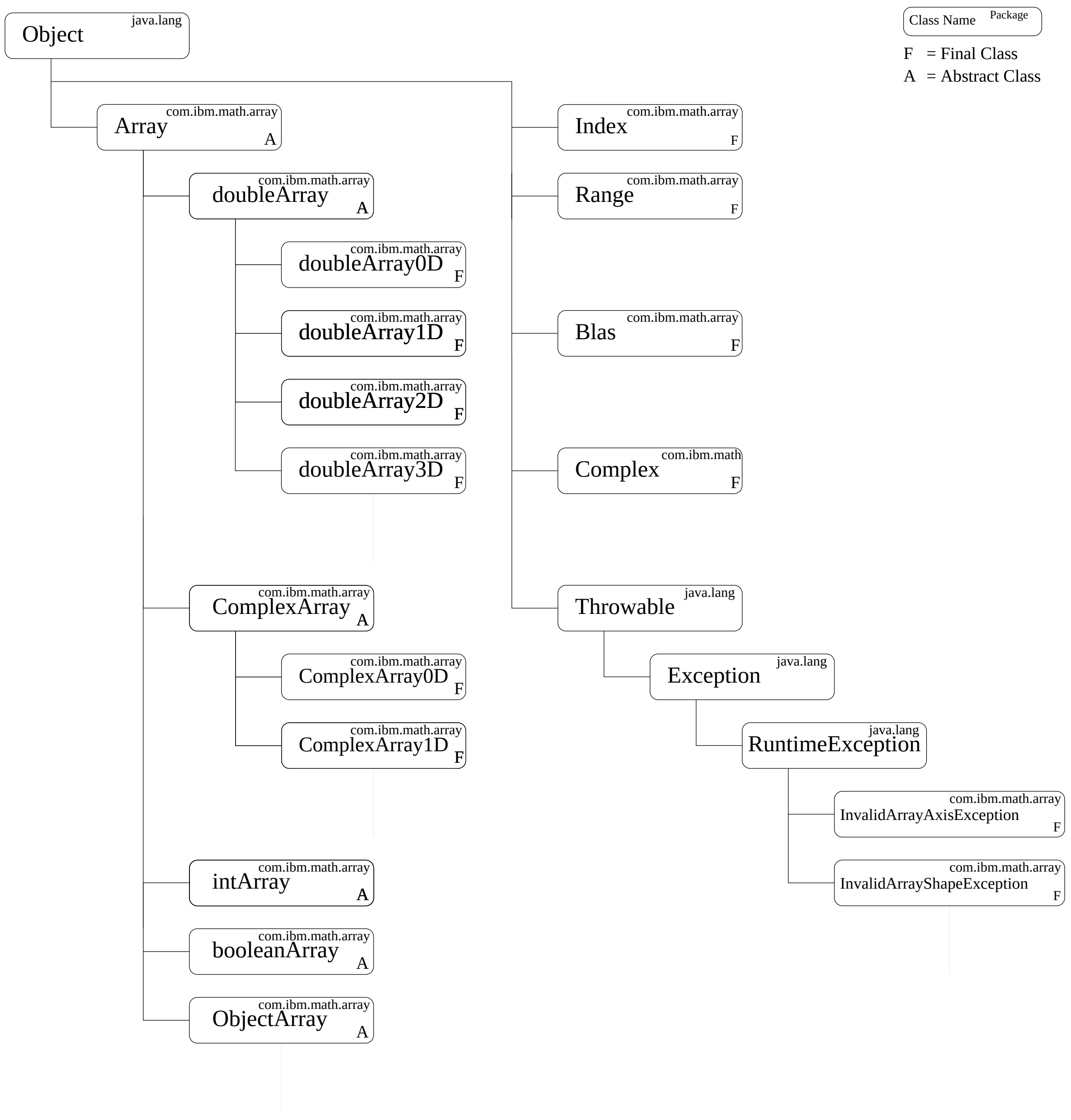
Object java.lang
Class Name Package
F = Final Class
A = Abstract Class
Array com.ibm.math.array A
Index com.ibm.math.array F
doubleArray com.ibm.math.array A
Range com.ibm.math.array F
doubleArray0D com.ibm.math.array F
doubleArray1D com.ibm.math.array F
Blas com.ibm.math.array F
doubleArray2D com.ibm.math.array F
doubleArray3D com.ibm.math.array F
Complex com.ibm.math F
ComplexArray com.ibm.math.array A
Throwable java.lang
ComplexArray0D com.ibm.math.array F
Exception java.lang
ComplexArray1D com.ibm.math.array F
RuntimeException java.lang
InvalidArrayAxisException com.ibm.math.array F
InvalidArrayShapeException com.ibm.math.array F
intArray com.ibm.math.array A
booleanArray com.ibm.math.array A
ObjectArray com.ibm.math.array A

Figure 2: Simplified class hierarchy chart.

and *Array*-*Array* operations. Scalar-Array operations apply the operation to a scalar and each element of the Array. Array-Array operations apply the operation to the equivalently (scalar) elements of two Arrays. If both Arrays in an Array-Array operation do not have the same shape, a `NonconformingArrayException` is thrown. These operations are highly optimizable and parallel in nature.

These operations implement *array semantics*: In evaluating an expression (*e.g.*, $A$ += $C$), all data is first fetched (*i.e.*, $A$ and $C$ are read from memory), the computation is performed (*i.e.*, $A + C$ is evaluated), and only then the result is stored (*i.e.*, $A$ is modified). The Array package uses a form of dynamic dependence analysis [2] to determine when to use a temporary array to hold intermediate data and when it is safe to avoid using temporaries.

- *Array manipulation*: methods in this group include `section`, `permuteAxes` and `reshape`. These methods operate on the Array as a whole, creating a new view of the data and returning a new Array object expressing this view. This new Array object shares its data with the old object when possible to avoid the overhead of data copying.

- *Array accessor methods*: the methods of this group are `get`, `set` and `toJava`. The `get` and `set` are basic accessor methods which use the indexing schemes described earlier. The method `toJava` allows a user to extract a Java array from an Array package Array. This feature, together with constructors that convert Java arrays into Array package Arrays, allows pieces of code that work with Java arrays to co-exist with (*i.e.*, exchange data with) pieces of code that use the Array package.

- *Array inquiry methods*: this group contains methods `last`, `size`, `rank` and `shape`. They are fast, descriptor-only operations which return information about the (invariant) properties of the Array.

- *BLAS routines*: the Basic Linear Algebra Subprograms (BLAS) [12] are the building blocks of efficient linear algebra codes. BLAS implements a variety of elementary vector-vector (level 1), matrix-vector (level 2), and matrix-matrix (level 3) operations. We have designed a BLAS class as part of the Array package. It provides the functionality of the BLAS operations for the multidimensional Arrays in that package. Note that this is a 100% Java implementation of BLAS, and not an interface to already existing native libraries. With this approach, combined with our optimizing compiler, we achieve near native code performance while still benefiting from the portability and reproducibility offered by Java.

## 2.2 An example

Figure 3 illustrates several features of the Array package by comparing two straightforward implementations of the basic BLAS operation `dgemm`. The `dgemm` operation computes $C = \alpha A^* \times B^* + \beta C$, where $A$, $B$, and $C$ are matrices and $\alpha$ and $\beta$ are scalars. $A^*$ can be either $A$ or $A^T$. The same holds for $B^*$. In Figure 3(a) the matrices are represented as `doubleArray2D` objects from the Array package. In Figure 3(b) the matrices are represented as `double[][]`.

The first difference is apparent in the interfaces of the two versions. The `dgemm` version for the Array package transparently handles operations on sections of a matrix. Section are extracted by the caller and passed on to `dgemm` as `doubleArray2D` objects. Section descriptors have to be explicitly passed to the Java arrays version, using the `m`, `n`, `p`, `i0`, `j0`, and `k0` parameters. The calling format for computing $C(0:9, 10:19) = A(0:9, 20:39) \times B(20:39, 10:19)$ using the Array package is

```
dgemm(NoTranspose,NoTranspose,1.0,
      A.section(new Range(0,9),new Range(20,39)),
      B.section(new Range(20,39),new Range(10,19)),0.0,
      C.section(new Range(0,9),new Range(10,19)))
```

and using Java arrays is

```
dgemm(NoTranspose,NoTranspose,10,20,10,0,10,20,
      1.0,A,B,0.0,C)
```

Next, we note that the computational code in the Array package version is independent of the orientation of $A$ or $B$. We just perform a (very cheap) transposition if necessary, by creating a new descriptor for the data

```
public static void dgemm(int transa,
                         int transb,
                         double alpha,
                         doubleArray2D a,
                         doubleArray2D b,
                         double beta,
                         doubleArray2D c)
  throws NonconformingArrayException {

  if (transa == Transpose)
    a = a.permuteAxes(1,0);
  if (transb == Transpose)
    b = b.permuteAxes(1,0);

  int m = a.size(0);
  int n = a.size(1);
  int p = b.size(1);

  if (n != b.size(0))
    throw new NonconformingArrayException();
  if (p != c.size(1))
    throw new NonconformingArrayException();
  if (m != c.size(0))
    throw new NonconformingArrayException();

  for (int i=0; i<m; i++) {
    for (int j=0; j<p; j++) {
      double s = 0;
      for (int k=0; k<n; k++) {
        s += a.get(i,k)*b.get(k,j);
      }
      c.set(i,j,alpha*s+beta*c.get(i,j));
    }
  }
}
```

(a)

```
public static void dgemm(int transa,
                         int transb,
                         int m,
                         int n,
                         int p,
                         int i0, j0, k0,
                         double alpha,
                         double[][] a,
                         double[][] b,
                         double beta,
                         double[][] c) {

  if (transa != Transpose) {
    if (transb != Transpose) {
      for (int i=i0; i<i0+m; i++) {
        for (int j=j0; j<j0+p; j++) {
          double s = 0;
          for (int k=k0; k<k0+n; k++) {
            s += a[i][k]*b[k][j];
          }
          c[i][j] = alpha*s+beta*c[i][j];
        }
      }
    } else {
      for (int i=i0; i<i0+m; i++) {
        for (int j=j0; j<j0+p; j++) {
          double s = 0;
          for (int k=k0; k<k0+n; k++) {
            s += a[i][k]*b[j][k];
          }
          c[i][j] = alpha*s+beta*c[i][j];
        }
      }
    }
  } else {
       .
       .
       .
  }
}
```

(b)

Figure 3: An implementation of `dgemm` using (a) the Array package and (b) Java arrays.

using the `permuteAxes` method. In comparison, the code in the Java arrays version has to be specialized for each combination of orientation of $A$ and $B$. (We only show the two cases in which $A$ is not transposed.)

Finally, in the Array package version we can easily perform some shape consistency verifications before entering the computational loop. If we pass that verification, we know we will execute the entire loop iteration space without exceptions. Such verifications would be more expensive for the Java arrays, as we would have to traverse each row of the array to make sure they are all of the appropriate length. Furthermore, at least the row-pointer part of each array would have to be privatized inside the method, to guarantee that no other thread changes the shape of the array.

This example illustrates the benefits, both for the programmer and the compiler, of operating with the true multidimensional rectangular Arrays of the Array package.

## 3 Compiler optimizations

The previous sections outlined the benefits of the Array package for numerical computing and explained why Java arrays are not a good match for numerical intensive code. In this section we discuss the compiler support required to extract high performance of numerically intensive Java code written with the Array package. First we present an overview of our compiler infrastructure. Then we discuss the two major compiler optimizations that we implemented: semantic expansion and bounds checking optimization.

### 3.1 The IBM XL family of compilers

Our compiler infrastructure is based on the IBM XL family of compilers. Figure 4 shows the high-level architecture of these compilers. The compilers from the XL family are able to compile source code from various source languages and generate code for various target architectures. The compilation process consists of three major steps:

1. The language-specific front-end translates the source code into an intermediate representation called W-Code. For Java, this step is implemented by a combination of *javac*, which translates Java into bytecode, and the High Performance Compiler for Java (HPCJ), which translates bytecode into W-Code.

2. The portable optimizer implements language and architecture independent optimizations. It consumes and produces intermediate representations in W-Code. The portable optimizer step is implemented by TPO (Toronto Portable Optimizer).

3. The back-end implements architecture and implementation specific optimizations and generates the actual executable code. It translates W-Code to machine code. In our case, the back-end step is implemented by TOBEY, which generates POWER/PowerPC executable code.

The second step (the portable optimizer) may be bypassed if desired (for faster compilation). Our optimizations were implemented in two of the modules described above. Semantic expansion was implemented in HPCJ, because it is a language-dependent optimization. The bounds checking optimization was implemented in TPO because it is, in principle, independent of language and architecture (even though it is particularly useful for Java). We note that semantic expansion must, in any case, precede the bounds checking optimization to allow effective analysis of code using the Array package, which is necessary for applying bounds checking optimization on that code.

### 3.2 Semantic expansion of Array references

As discussed earlier, to avoid changing the Java language itself, we added support for true multidimensional arrays in the form of a set of classes comprising the Array package. Because of encapsulation, all data access in the Array package is through accessor methods. In the absence of aggressive compiler optimizations, these method invocations are expensive, both in terms of their overhead and in making analysis for the containing program regions very conservative. In order to obtain high performance, we must perform the array accesses in a more transparent, compiler-optimizable, way. We use the semantic expansion technique [31] to achieve this.

Our Java front-end (HPCJ) implements semantic expansion of the `get` and `set` accessor methods for Arrays. A call to a `get` or `set` method that accesses a single array element is replaced by inline code that performs the same array access, including the checks for `null`-pointer exception and out-of-bounds array exception that might be thrown due to that access. The compiler exploits information about the dense layout of Array data to generate code using the multidimensional array indexing operations in the

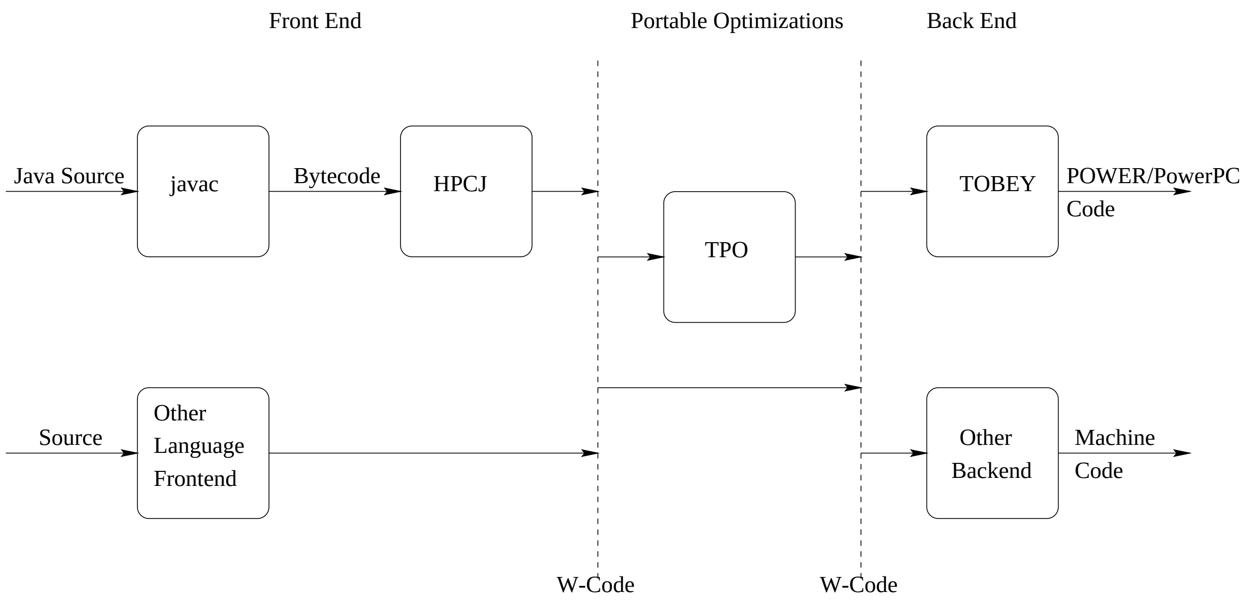

Figure 4: Architecture of the XL family of compilers.

W-Code intermediate representation. We note that this information could not have been exposed to the later compilation phases by applying regular method inlining instead of semantic expansion. The resulting code can be optimized by the W-Code based optimizer (TPO) using standard techniques (in conjunction with the bounds checking optimization). Thus, the Java Array package, combined with semantic expansion in the compiler, effectively provides the advantages of true multidimensional arrays in Java without any changes to the JVM or to the language itself and without violating the safety benefits provided by the Java language.

## 3.3 Bounds checking optimization

The Java requirement that every array access be checked for `null`-pointer and out-of-bounds exceptions, coupled with its precise exceptions model, provides the benefits of safety and security, and also helps in the debugging of programs. On the other hand, it has a severe impact on the performance of numerical codes. The goal of the bounds checking optimization is to enable the generation of high performance machine code without eliminating the benefits provided by the exception checks (we use the term *bounds checking* loosely to cover `null`-pointer checks as well).

Our approach is to create *safe regions* by program transformation so that all array accesses in the safe regions are guaranteed not to generate a `null`-pointer or out-of-bounds exception [24]. This has both direct and indirect benefits. First, the exception checks on all array references inside the safe regions can be removed, reducing their direct overhead. Second, the safe region code can now be freely transformed (for example, using loop transformations like tiling) without violating the precise exceptions model of Java. In this work, we use the *versioning* technique to create safe and unsafe regions, with a run-time test to check whether the safe or the unsafe region should be executed [24]. The optimization is done in two phases: the *array range analysis phase* and the *safe region creation phase*. We note that this optimization can be applied to arrays in any language (including the Arrays from the Array package).

**Array range analysis:** In the analysis phase, the compiler obtains summary access information for each array with respect to the *intervals* (*i.e.*, the loops and the overall procedure) in a method. This information is represented using a bounded *regular section descriptor* (RSD) [8, 19], which provides information about the lower bound and the upper bound on (read and write) accesses in each array axis. The procedure for computing the bounded RSDs uses a combination of standard array analysis techniques and sophisticated symbolic analysis, described further in [17]. However, an important difference in the context of applying this analysis for the bounds checking optimization is that we can ignore certain aliasing information while identifying the section of an array that is read or written. For example, in Figure 5, we can summarize the access information for the array $A$ accurately (for the purpose of the bounds checking optimization), even if $A$ may potentially overlap with any part of the array $C$. The bounds checks for $A$ can be safely eliminated based only on the summary of accesses for $A$. Of course, a different transformation, such as loop reordering or parallelization, would have to consider the aliasing between $A$ and $C$.

For cases where the compiler is unable to obtain, even symbolically, information about the range of array accesses, the corresponding range information is set to $\perp$, indicating an unknown value. For example, if the base address of an array $A$ changes inside a loop in an unknown manner, the lower and upper bounds information for each axis of $A$ is set to $\perp$.

**Safe region creation:** In this phase, the information from array range analysis is used to drive the versioning transformation which creates safe and unsafe regions. Our current implementation processes all loops in the method, and then selects each outermost loop for which complete array range information is symbolically available for every array reference (*i.e.*, there are no $\perp$ expressions in the array range information, although there may be symbolic variables appearing in the expressions which are invariant with respect to that loop). A duplicate code copy is created for each selected loop. From the copy which is to be made the safe region, all `null`-pointer checks and out-of-bounds checks for the arrays are eliminated. An expression for the run-time test to check whether the safe or the unsafe region should be executed, is created in two parts: the first part verifies that none of the arrays is `null`, and the second part verifies for each axis of every array that the possible lower and upper bounds of access computed by the array range analysis are within the actual bounds of the array at run-time. This expression is simplified by expression-folding techniques. When the complete expression guarding the execution of the safe region can be evaluated at compile time, the generated code is much simpler because one of the regions is recognized as unreachable code and eliminated.

**An example of Array range analysis and safe region creation:** As an example of our bounds checking optimization technique, Figure 5 shows what happens with a simple code for matrix multiplication. For the purpose of clarity in this example, we use the notation `A[i,j]` to represent either `A.get(i,j)` or `A.set(i,j,...)`. The original code, provided as input to the compiler, is shown in Figure 5(a). The $\mathsf{CHK}_b()$ operation denotes explicit bounds checks for each index, while $\mathsf{CHK}_n()$ denote the `null`-pointer checks. The code optimized through versioning is shown in Figure 5(b). The safe region contains no explicit checks and can be aggressively optimized.

The original code in Figure 5(a) contains three potential safe regions, one for each loop. For the innermost (`k`) region/loop we can prove that the range of accesses for the three Arrays are:

$$\begin{array}{l} C[i:i, j:j] \\ A[i:i, 0:n-1] \\ B[0:n-1, j:j] \end{array} \tag{1}$$

For the middle (`j`) loop we compute the following range information:

$$\begin{array}{l} C[i:i, 0:p-1] \\ A[i:i, 0:n-1] \\ B[0:n-1, 0:p-1] \end{array} \tag{2}$$

```
for (i=0;i<m;i++)
  for (j=0;j<p;j++)
    for (k=0;k<n;k++)
      CHK_n(C)[CHK_b(i),CHK_b(j)] =
            CHK_n(C)[CHK_b(i),CHK_b(j)] +
            CHK_n(A)[CHK_b(i),CHK_b(k)] *
            CHK_n(B)[CHK_b(k),CHK_b(j)];
```

(a)

```
if (A!=null)&&(B!=null)&&(C!=null)&&
   (m-1 < A.size(0))&&(n-1 < A.size(1))&&
   (n-1 < B.size(0))&&(p-1 < B.size(1))&&
   (m-1 < C.size(0))&&(p-1 < C.size(1)) {

   for (i=0;i<m;i++)
     for (j=0;j<p;j++)
       for (k=0;k<n;k++)
         C[i,j] = C[i,j] + A[i,k]*B[k,j];

} else {

   for (i=0;i<m;i++)
     for (j=0;j<p;j++)
       for (k=0;k<n;k++)
         CHK_n(C)[CHK_b(i),CHK_b(j)] =
               CHK_n(C)[CHK_b(i),CHK_b(j)] +
               CHK_n(A)[CHK_b(i),CHK_b(k)] *
               CHK_n(B)[CHK_b(k),CHK_b(j)];

}
```

(b)

Figure 5: An example of safe region creation through range analysis and versioning: (a) the original code, (b) optimized code.

And finally, for the outermost (`i`) loop we compute:

$$\begin{array}{l} C[0:m-1, 0:p-1] \\ A[0:m-1, 0:n-1] \\ B[0:n-1, 0:p-1] \end{array} \tag{3}$$

Since complete information is available for the outermost loop, the compiler generates the versioning code of Figure 5(b). Note that the tests to verify in-bounds access with respect to the lower bounds of the arrays are folded away, as they evaluate to *true* at compile time.

## 3.4 Array package advantages for bounds checking optimization

We now discuss some additional advantages of using Arrays from the Array package over Java arrays, in terms of the complexity of the bounds checking optimization. First, since Java multidimensional Arrays are actually arrays of arrays, eliminating bounds checks completely on those arrays requires a limited form of pointer chasing analysis, to correlate references to different rows of the same base multidimensional array, while summarizing accesses to that array. Alternatively, an implementation could view two different rows of a multidimensional array as separate arrays, in which case, the optimization would only be effective for the last axis of the array. Second, in a multithreaded environment, we need an array privatization scheme to prevent other threads from asynchronously changing the shape of the multidimensional array being subjected to the optimization [24]. For an $(n_1 \times n_2 \times \ldots \times n_d)$ $d$-dimensional array, this is an $O(n_1 \times n_2 \times \ldots \times n_{d-1})$ operation, which can be potentially expensive. These costs can be avoided for Array package Arrays, as the shapes of these Arrays are invariant during their lifetime. At most, the base address of a multidimensional Array would need to be privatized for an Array package implementation that may change the base address of an Array. Our current Array package implementation does not change the base address, so no such privatization is needed.

## 4 Experimental results

We performed a series of experiments to measure the impact of the Array package and our compiler optimizations on the performance of Java numerical codes. In this section we present a summary of results for four benchmarks. We performed 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.

We compare the performance of three different versions of each benchmark. The first version is implemented in Fortran and serves as a performance reference. The second version is implemented in Java using only Java arrays. The third version is also implemented in Java, but using the Array package. Fortran programs are compiled using version 6.1 of the IBM XLF compiler with the highest level of optimization (-O3 -qhot, which performs high-order loop transformations [28]). For the Java arrays version, we report performance obtained using JDK 1.1.6 with the IBM JIT, which delivers the best performance for this version from all the Java environments available for our platform, including HPCJ. (The IBM JIT incorporates many advanced optimizations such as bounds checking and `null`-pointer checking optimization.) For the Array package version, we report the performance using our HPCJ-based compiler.

**The benchmarks:** The four benchmarks used in our experiments are: MATMUL, BSOM, MICRO DC, and TOMCATV. MATMUL computes $C = C + A \times B$, where $C$, $A$, and $B$ are matrices of size $500 \times 500$. 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. The reader should note that this benchmark *does not* use the BLAS routines in the Array package. BSOM (Batch Self-Organizing Map) benchmark is a data-mining kernel representative of technologies incorporated into Version 2 of the IBM Intelligent Miner. It implements a neural-network-based algorithm to determine clusters of input records that exhibit similar attributes of behavior. We time the execution of the *training* phase of this algorithm, which actually builds the neural network. This phase consists of 25 *epochs*. Each epoch updates a neural network with 16 nodes using 256 records of 256 fields each. MICRO DC computes the potential field in a microstrip structure. It solves the equation $\nabla^2\Phi = -\frac{\rho}{\epsilon}$ on a discretized domain using Jacobi relaxation [26]. For this benchmark, we use the problem configuration described in [23], with a $1000 \times 1000$ grid and four parallel microstrips of cross section $100 \times 10$. TOMCATV is part of the SPECfp95 suite (`www.spec.org`). It is a vectorized mesh generation with Thompson solver code. The benchmark consists of a main loop, which iterates until convergence or until a maximum number of iterations is executed. At each iteration of the main loop, the major computation consist of a series of stencil operations on two (coupled) grids, $X$ and $Y$, of size $n \times n$. In addition, a set of tridiagonal systems are solved through LU decomposition. For our experiments, we use a problem size $n = 513$.

**Results:** Results for the four benchmarks are summarized in Table 1 and Figure 6. Table 1 summarizes the efficacy of our bounds checking optimization. For each benchmark, column "loops" lists the total number of loop constructs in the program. (The nest of Figure 5(a) counts as three loops.) Column "optimized" lists the number of loop constructs that can be optimized with our compiler. That is, the number of loops for which the compiler was able to compute complete array range information. Column "coverage" is the ratio, in percentage, of loops optimized to total loops. Finally, column "safe regions" list the actual number of safe regions created for the benchmark. Since safe regions are created per loop nest, the number of safe regions can be much less than the number of optimized loops. For example, the safe region of Figure 5(b) optimizes three loops.

In the plots of Figure 6, the height of each bar is normalized with respect to the performance of the Fortran version. The numbers on the top of the bars indicate actual Mflops achieved for that version. The

| benchmark | loops | optimized | coverage | safe regions |
|---|---|---|---|---|
| MATMUL | 17 | 14 | 82.4% | 6 |
| BSOM | 25 | 21 | 84.0% | 13 |
| MICRO DC | 12 | 11 | 91.7% | 5 |
| TOMCATV | 20 | 20 | 100.0% | 5 |

Table 1: Summary of loops optimized in each benchmark.

JDK116 bar shows the result for the Java array version, with JDK 1.1.6 + JIT. The HPCJ bar shows the result for the Array package version, as compiled by plain HPCJ (*i.e.*, without neither the semantic expansion nor bounds checking optimization). The HPCJ+SE bar shows the result for the Array package version with semantic expansion. The BOUNDS bar shows the best result that can be accomplished with 100% pure Java: it uses the Array package with semantic expansion and bounds checking optimization. The FMA bar shows the extra boost in performance that we can get by allowing Java to use the fused-multiply add (`fma`) instruction of the POWER architecture. This instruction is currently illegal in Java, but proposals that allow its use are under consideration [20]. (Our FMA results were obtained by enabling the TOBEY backend to generate `fma` instructions. TOBEY then looks for occurrences of chained multiplication and addition, implementing both operations with a single `fma` instruction.) Finally, the FORTRAN bar shows the result for the Fortran version of the benchmark. The Fortran version uses the `fma` instruction. (For the purpose of completeness, we report here that JDK 1.1.6 + JIT achieves 3.2, 1.7, 1.3, and 1.4 Mflops, or approximately the same as HPCJ, for the Array package versions of MATMUL, BSOM, MICRO DC, and TOMCATV, respectively.)

**Discussion:** The performance of numerical codes using Java arrays (the JDK116 bars) is clearly unsatisfactory. It is typically between 10% (MATMUL,MICRO DC) and 25% (BSOM) of Fortran performance. The performance of Array package code is terrible with standard Java environments (the HPCJ bars). The execution cost in this case is dominated by the overhead of method invocation. Even when semantic expansion is used (the HPCJ+SE bars), the performance of the Array package versions are almost always worse than the performance with Java arrays. Although semantic expansion eliminates the method overhead, the explicit bounds and `null`-pointer checks are expensive and prevent optimizations. It is only when bounds checking (and `null`-pointer checking) optimization is performed (the BOUNDS bars) that we start to see Java performance that is much better than with Java arrays and comparable with Fortran. Speedups over Java arrays vary from 2.5 (BSOM) to 6.3 (MICRO DC). Performance with semantic expansion and bounds optimization is between 50% (MATMUL) and 80% (MICRO DC) of Fortran. When `fma`s are allowed, the performance can be as high as 90% (MATMUL) of Fortran. We note that for three benchmarks (MATMUL, BSOM, and TOMCATV), `fma`s were generated by the compiler when enabled. The benefits of `fma`s are substantial for MATMUL and BSOM, less so for TOMCATV. For MICRO DC, no `fma`s were generated by the compilers (not even by the Fortran compiler).

## 5 Related work

The importance of having high-performance libraries for numerical computing in Java has been recognized by many authors. In particular, [4, 5, 29] describe projects to develop such libraries entirely in Java. In addition, there are approaches in which access to existing numerical libraries (typically coded in C and Fortran) is provided to Java applications [3, 9, 16]. All of these approaches shift the burden of delivering high performance entirely to the libraries.

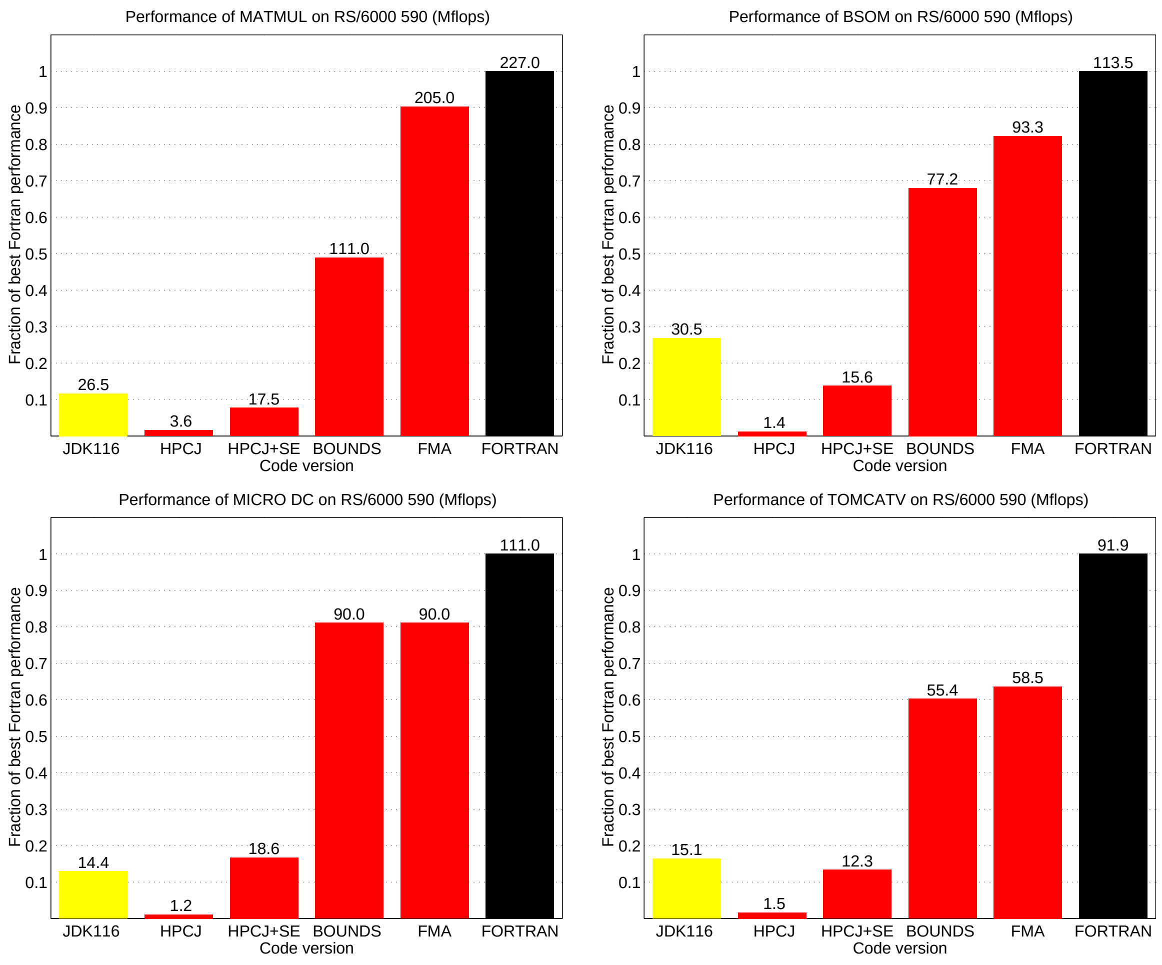

Figure 6: Experimental results for all benchmarks.

It is well know that programming with libraries has its limitations. Certain classes of applications can be effectively expressed in terms of linear algebra operations. These applications can be efficiently implemented on top of (very successful) linear algebra libraries such as LAPACK and ESSL. Other classes of applications are not so well structured. In those cases, the programmer has to develop the application from scratch, counting with minimal library support. For this reason, the Array package is more than just a library: It is a mechanism to represent array operations in a manner that can be highly optimized by compilers. Linear algebra code can be developed with the Array package using the BLAS primitives. Less structured code can use the array operations directly. We have shown that with semantic expansion and other compiler optimizations, Java code built on top of the most basic `set` and `get` operations of the Array package can achieve high performance.

In [11], 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. This approach can lead to results that are as good as from using multidimensional Arrays in the Array package. We differ in that we do not need global analysis: Arrays from the Array package are intrinsically rectangular and can always be stored in a dense form.

In [6], a project begun at the JavaSoft division of Sun Microsystems and its follow-on at Rice University are outlined. The work was aimed at a *dynamic* (or *just-in-time*) compilation system, and focused on local scalar optimizations. Hand compilations using a technique called *object inlining* achieved up to a 7-fold speedup on an *Oopack* routine (an object oriented version of Linpack). Continuing work by the same group, described in [7], discusses additional compiler optimizations to enhance the performance of polymorphic, object oriented numerical code in Java. These projects are complementary to our work, and show the gains that can be made in Java performance.

The technique of semantic expansion we use is similar to the approach taken by Fortran compilers in implementing some *intrinsic* procedures [1]. Intrinsic procedures can be treated by the Fortran compiler as language primitives, just as we treated the `get` and `set` methods in various classes of the Array package. Semantic expansion differs from the handling of intrinsics in Fortran in that both new data types and operations on the new data types are treated as language primitives. Semantic expansion was used to reduce the overhead of handling *complex* numbers in Java (in particular, the overhead associated with creating a large number of temporary objects) in [31].

Marmot [13] is an optimizing static Java compiler being developed at Microsoft Research. It implements both standard scalar (Fortran and C style) optimizations and basic object oriented optimizations. It also performs bounds checking optimization by determining, through a form of range analysis, which checks are redundant. There is no discussion of versioning in [13]. Marmot shows significant speedups relative to other Java environments in several benchmarks, but its performance is only average for the numerical benchmarks listed in [13].

The bounds-checking optimization used in this work was first presented by us in [24]. The optimization was applied by hand, and only to Java arrays, in [24]. In contrast, this paper advocates the use of an Array package (which simplifies the bounds checking optimization and leads to many other benefits) and describes complementary compiler optimizations implemented by us, that deliver high performance on a suite of numerical benchmarks. The optimizations described in this paper were applied automatically by the compiler, without manual intervention.

## 6 Conclusion

We have demonstrated that high-performance numerical codes can be developed in Java. The benefits of Java from a productivity and demographics point of view have been known for a while. Many people have advocated the development of large, computationally intensive applications in Java based on those benefits. However, Java performance has consistently been considered an impediment to its applicability to numerical codes. This paper has presented a set of techniques that lead to Java performance that is comparable with the best Fortran implementations available today.

Adopting a language and compiler codesign approach, we have used a class library to introduce in Java features of Fortran 90 that are of great importance to numerical computing. These features include complex numbers, multidimensional arrays, and libraries for linear algebra. These features have a strong synergy with the compiler optimizations that we are developing, particularly bounds checking optimization and semantic expansion. Semantic expansion eliminates the unacceptable overhead of accessing individual Array elements through accessor methods and provides the compiler with knowledge of the semantics of those Arrays. Bounds (and `null`-pointer) checking optimization eliminates the overhead of run-time checks and enables other existing compiler optimizations. All components of the Array package, including the BLAS library, are implemented entirely in Java. This guarantees portability of code and reproducibility of results in various platforms. Of course, one can develop conforming implementations using native code, but that requires special care in each platform to guarantee that corner cases behave appropriately and the right exceptions are generated. (For instance, using the native BLAS of each vendor practically guarantees that each implementation will behave differently.) The beauty of our approach is that we can achieve near native code performance with a 100% Java implementation.

We demonstrated that we can deliver Java performance in the 65-90% range of the best Fortran performance for a variety of benchmarks. This is a very important step in making Java the preferred environment for developing large-scale computationally intensive applications. However, there is more than just performance to our approach. The Array package and associated libraries create a Java environment with many of the features that experienced Fortran programmers have grown accustomed to. This combination of delivering Fortran array manipulation features and performance, while maintaining the safety and elegance of Java, is the core of our approach to making Java the environment of choice for new numerical computing. Our next logical step is to exploit the automatic parallelization features of TPO. The safe regions of code, produced by our techniques, are amenable to automatic parallelization techniques developed for C and Fortran code. The main issue we have to address is providing proper aliasing information so that parallelization can be accomplished. We note that versioning can also be helpful in dealing with aliasing issues. Looking back at the code of Figure 5(b), we can insert additional conditions to the runtime test to verify that $A$, $B$, and $C$ are not aliased. The safe region would then also be free of aliasing and more aggressive optimization and parallelization could be performed.

**Acknowledgements:** The authors wish to thank Rick Lawrence and Marc Snir for fruitful technical discussions, and for strongly supporting our research. We also wish to thank the members of the HPCJ and TPO teams at the IBM Toronto Lab for their work on the compiler infrastructure.

## References

[1] J. C. Adams, W. S. Brainerd, J. T. Martin, B. T. Smith, and J. L. Wagener. *Fortran 90 Handbook: Complete ANSI/ISO Reference*. McGraw-Hill, 1992.

[2] Utpal Banerjee. *Dependence Analysis*. Loop Transformations for Restructuring Compilers. Kluwer Academic Publishers, Boston, MA, 1997.

[3] A. Bik and D. B. Gannon. A note on level 1 BLAS in Java. In *Proceedings of the Workshop on Java for Computational Science and Engineering - Simulation and Modeling II*, June 1997. available at `http://www.npac.syr.edu/users/gcf/03/javaforcse/acmspecissue/latestpapers.html`.

[4] 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.

[5] 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`.

[6] Z. Budimlic and K. Kennedy. Optimizing Java: Theory and practice. *Concurrency, Pract. Exp. (UK)*, 9(11):445–63, November 1997. Java for Computational Science and Engineering - Simulation and Modeling II Las Vegas, NV, USA 21 June 1997.

[7] Z. Budimlic and K. Kennedy. Prospects for scientific computing in polymorphic, object-oriented style. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, San Antonio, March 1999.

[8] D. Callahan and K. Kennedy. Analysis of interprocedural side effects in a parallel programming environment. *Journal of Parallel and Distributed Computing*, 5:517–550, 1988.

[9] 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.

[10] M. Cierniak and W. Li. Unifying data and control transformations for distributed shared memory machines. In *Proc. ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, La Jolla, CA, June 1995.

[11] 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.

[12] J. J. Dongarra, I. S. Duff, D. C. Sorensen, and H. A. van der Vorst. *Solving Linear Systems on Vector and Shared Memory Computers*. Society for Industrial and Applied Mathematics, 1991.

[13] R. Fitzgerald, T. B. Knoblock, E. Ruf, Bjarne Steensgaard, and D. Tarditi. Marmot: an optimizing compiler for Java. Technical report, Microsoft Research, October 1998. URL: `http://research.microsoft.com/apl% discretionary//default.htm`.

[14] C.-W. Tseng G. Rivera. Data transformations for eliminating conflict misses. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation*, pages 38–49, Montreal, Canada, June 1998. ACM Press.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, 1995.

[16] V. Getov, S. Flynn-Hummel, and S. Mintchev. High-performance parallel programming in Java: Exploiting native libraries. In *ACM 1998 Workshop on Java for High-Performance Network Computing*. ACM SIGPLAN, 1998. Available at `http://www.cs.ucsb.edu/conferences/java98`.

[17] M. Gupta, S. Mukhopadhyay, and N. Sinha. Automatic parallelization of recursive procedures. Technical Report RC 21333(96110)4NOV1998, IBM Research, November 1998.

[18] F. G. Gustavson. Recursion leads to automatic variable blocking for dense linear algebra algorithms. *IBM Journal of Research and Development*, 41(6):737–755, November 1997.

[19] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

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

[21] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. A matrix-based approach to the global locality optimization problem. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques (PACT'98), Paris, France*, October 1998.

[22] M. Kandemir, A. Choudhary, J. Ramanujam, N. Shenoy, and P. Banerjee. Enhancing spatial locality via data layout optimizations. In *Proceedings of Euro-Par'98, Southampton, UK*, volume 1470 of *Lecture Notes in Computer Science*, pages 422–434. Springer Verlag, September 1998.

[23] 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.

[24] 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.

[25] R. Parsons and D. Quinlan. Run time recognition of task parallelism within the P++ parallel array class library. In *Proceedings of the Scalable Parallel Libraries Conference*, pages 77–86. IEEE Comput. Soc. Press, October 1993.

[26] 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.

[27] J. V. W. Reynders, J. C. Cummings, M. Tholburn, P. J. Hinker, S. R. Atlas, S. Banerjee, M. Srikant, W. F. Humphrey, S. R. Karmesin, and K. Keahey. POOMA: A framework for scientific simulation on parallel architectures. In *Proceedings of First International Workshop on High Level Programming Models and Supportive Environments, Honolulu, HI*, pages 41–49, April 16 1996. Technical report available at `http://www.acl.lanl.gov/PoomaFramework/papers/papers.html`.

[28] 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.

[29] M. Schwab and J. Schroeder. Algebraic Java classes for numerical optimization. In *ACM 1998 Workshop on Java for High-Performance Network Computing*. ACM SIGPLAN, 1998. Available at `http://www.cs.ucsb.edu/conferences/java98`.

[30] G. V. Wilson and P. Lu, editors. *Parallel Programming using C++*. MIT Press, 1996.

[31] P. Wu, S. P. Midkiff, J. E. Moreira, and M. Gupta. Efficient support for complex numbers in Java. In *To appear, Proceedings of the 1999 ACM Java Grande Conference*, 1999. available as IBM Research Division technical report no. 21393.