Supporting Multidimensional Arrays in Java

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

Abstract

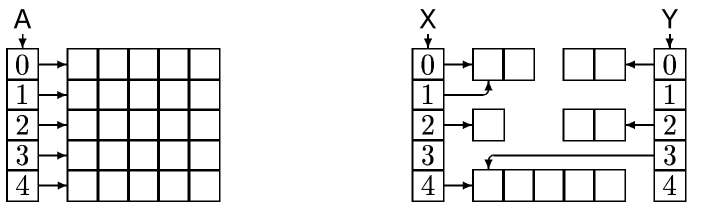
The lack of direct support for multidimensional arrays in Java$^{\text{TM}}$ has been recognized as a major deficiency in the language's applicability to numerical computing. It has been shown that, when augmented with multidimensional arrays, Java can achieve very high-performance for numerical computing through the use of compiler techniques and efficient implementations of aggregate array operations. Three approaches have been discussed in the literature for extending Java with support for multidimensional arrays: class libraries that implement these structures; relying on the JVM to recognize those arrays of arrays that are being used to simulate multidimensional arrays; and extending the Java language with new syntactic constructs for multidimensional arrays and directly compiling those constructs to bytecode. This paper presents a balanced presentation of the pros and cons of each technique in the areas of functionality, language and virtual machine impact, implementation effort, and effect on performance. We show that the best choice depends on the relative importance attached to the different metrics, and thereby provide a common ground for a rational discussion of the technique that is in the best interests of the Java community at large, and the growing community of numerical Java programmers.

1 Introduction

The multidimensional array (or *multiarray* for short) is an intuitive concept for numerical programmers in Fortran and C. Multiarrays are d-dimensional rectangular collections of elements. A multiarray is characterized by its *rank* (number of dimensions or axes), its elemental data *type* (all elements of a multiarray are of the same type), and its *shape* (the extents along its axes). Elements of a multiarray are identified by their indices along each axis. Let a d-dimensional multiarray A of elemental type T have extent n_j along its j-th axis, j = 0,...,d-1. Then, a valid index i_j along the j-th axis must be greater than or equal to zero and less than n_j. Our definition of multiarrays also includes the following: The type, rank, and shape of a multiarray are immutable during its lifetime. We will see that this immutability property has important performance implications.

The Java Programming Language$^{\text{TM}}$ does not support true multidimensional arrays. This has been recognized as a major deficiency in Java's applicability to numerical computing. Whereas the more recent versions of Fortran have significantly enhanced support for multidimensional arrays, Java provides only single-dimensional arrays. To some extent, it is possible to simulate multidimensional arrays with arrays of arrays. For example, `double[][]` is an array of one-dimensional arrays of `double`s, which can be used to simulate two-dimensional arrays. Figure 1(a) illustrates the concept of arrays of arrays, simulating a two-dimensional multiarray. This approach, however, leaves much to be desired.

Arrays of arrays are not necessarily rectangular, and can lead to both inter- and intra-array aliasing, as shown in Figure 1(b). Furthermore, the structure of arrays of arrays can change during a computation. These



(a) regular array of arrays (b) irregular arrays of arrays

Figure 1: The structure of an array of arrays. It can be used to simulate multiarrays (a), and also to build more complicated structures (b).

characteristics make the job of automatically optimizing Java array code almost impossible for existing compilers. In contrast, the technology to analyze and optimize code that manipulates rectangular, fixed shape multidimensional arrays has been mature for many years, as demonstrated by the success of commercially available Fortran compilers. Another limitation of arrays of arrays has to do with parameter passing. It is very difficult to pass general regular sections of an array of arrays between caller and callee. Referring to Figure 1(a), whereas it is trivial to pass row 0 of array `A` to a method (just pass `A[0]`), it is not possible to pass column 0 of array `A` without first copying it to another one-dimensional array.

The performance and functional benefits of augmenting Java with true multidimensional arrays have already been demonstrated [1]. The immutable and rectangular shapes of multiarrays enable the application of various compiler techniques to Java, including loop transformations and automatic parallelization. It also becomes much simpler to pass multiarray sections between caller and callee. In the face of these benefits, there have been several proposals to support true multidimensional arrays in Java.

Most of the previous proposals for supporting multidimensional arrays in Java have focused on class libraries that provide array language functionality (such as found in Fortran 90) in Java. A proposal for standardization of a multidimensional array class library is officially under way as JSR-083 [12]. It has been demonstrated that compilers can optimize Java code using these multidimensional array libraries, achieving performance similar to state-of-the-art Fortran compilers. One deficiency with the class library approach is that the syntax for the application programmer is cumbersome. This has led to some proposals to extend the syntax of the language to support multidimensional arrays, possibly through operator overloading. The new constructs simply add to existing Java syntax. All currently valid Java programs remain valid with their current semantics. The Java source code can then be translated to Java bytecode that invokes the appropriate methods of the class library.

Another approach that has been discussed is translating an extended multiarray syntax directly to Java bytecode. We show how multidimensional array declarations and operations can be directly translated to Java bytecode, without requiring any additional class libraries. Our approach supports arrays of arbitrary type and rank, and leads to bytecode that can be executed on existing virtual machines.

Finally, a third approach that has been discussed in the literature is that of relying exclusively on the JVM to recognize those arrays of arrays that are being used to simulate multidimensional arrays. In this approach, no new language constructs are necessary, as it is up to the JVM to choose a more efficient implementation for those arrays. This approach can also deliver good performance when the JVM analysis is precise enough, but it does not improve the existing interfaces for numerical computing.

The major goal of this paper is to deliver a comparative discussion of the three approaches to adding multidimensional arrays to Java presented above. We start in Section 2 with an illustration of the performance and interface problems associated with Java arrays of arrays. In Section 3 we present a possible syntax for multiarrays in Java, and discuss the performance impact of supporting true multidimensional arrays. In Sections 4, 5, and 6 we introduce the three approaches discussed above: class library, direct translation, and transparently by the JVM, respectively. We present a description of the three approaches, listing the pros and cons of each. In Section 7, we compare each of the approaches with regards to functionality, impact on the language specification, implementation efforts, and typical achievable performance. We show that all approaches have their merits and problems. This comparison leads to an objective classification of these three approaches. It forms the basis for a rational decision regarding their implementation and deployment. Finally, in Section 8 we conclude this paper and discuss future activities.

2 Java arrays of arrays

As a running example throughout this paper, we consider the implementation of two methods from the basic linear algebra subroutines (BLAS): `dgemv` and `dgemm`. Figure 2 shows the code for an implementation of `dgemv` using regular Java arrays. This method computes

$$y \leftarrow \alpha \tilde{A} x + \beta y,$$

where A is a matrix of double precision floating-point values and x and y are vectors of floating-point values. $\tilde{A}$ denotes either A^T (when the `trans` flag is **true**) or A (when the `trans` flag is **false**). We compute the matrix-vector multiply in two different ways, depending on the value of the `trans` flag. If `trans` is **true**, then element i of y is the dot-product of column i of A by vector x, scaled by α and added to βy_i. Otherwise, element i of y is the dot-product of row i of A by vector x, scaled by α and added to βy_i.

```
public static void dgemv(boolean trans, int m, int n, double alpha,
                         double[][] A, double[] x, double beta, double[] y) {

       if (trans) {
          for (int i=0; i<m; i++) {
              double s = 0;
              for (int j=0; j<n; j++) s += A[j][i]*x[j];
              y[i] = alpha*s + beta*y[i];
          }
       } else {
          for (int i=0; i<m; i++) {
              double s = 0;
              for (int j=0; j<n; j++) s += A[i][j]*x[j];
              y[i] = alpha*s + beta*y[i];
          }
       }

       return;
}
```

Figure 2: The `dgemv` method with Java arrays.

Current successful Fortran compilers for high performance numerical applications include optimizers capable of applying high order transformations [18] to the code. These transformations include, but are not limited to, loop fusion, unimodular transformations, loop tiling, unroll-and-jam, and loop parallelization [2, 16, 21]. Now, consider the job of a Java just-in-time compiler as it optimizes execution of the bytecode generated for the loop nests in the above `dgemv` example. Before it can apply any transformations to a loop nest, the compiler must prove that (i) the loop nest executes without exceptions, (ii) the arrays involved are not aliased to each other or to themselves, and (iii) the shapes of the arrays do not change during execution. In general, these properties cannot be verified at compile time.

One solution is to apply versioning to dynamically select between two instances of a loop nest: a *safe* instance that is guaranteed to not generate exceptions, and an *unsafe* instance that may generate exceptions.

```
public static void dgemv(boolean trans, int m, int n, double alpha,
                         double[][] A, double[] x, double beta, double[] y) {

       if (trans) {
          for (int i=0; i<m; i++) {
              double s = 0;
              for (int j=0; j<n; j++) s += A[j][i]*x[j];
              y[i] = alpha*s + beta*y[i];
          }
       } else {
          if ((m <= y.length) && (n <= x.length) && (∀ i, n <= A[i].length) && (m <= A.length)) {
             // safe version
             for (int i=0; i<m; i++) {
                 double s = 0;
                 for (int j=0; j<n; j++) s += A[i][j]*x[j];
                 y[i] = alpha*s + beta*y[i];
             }
          } else {
             // unsafe version
             for (int i=0; i<m; i++) {
                 double s = 0;
                 for (int j=0; j<n; j++) s += A[i][j]*x[j];
                 y[i] = alpha*s + beta*y[i];
             }
          }
       }

       return;
}
```

Figure 3: The `dgemv` method with Java arrays and a versioning test for exception safety.

Figure 3 shows the versioning transformation applied to the "no transpose" loop nest of `dgemv`. We note that the versioning test can be expensive, as it requires testing the length of all rows of array `A`.

The versioning in Figure 3 is not good enough to enable compiler transformations, as there may be aliasing among the arrays `A`, `x`, and `y`. Therefore, we augment the versioning test to check for possible aliasing, as shown in Figure 4. Again, the test is made expensive by the need to verify the aliasing between array `y` and each row of array `A`.

One more step needs to be taken to ensure that the compiler can optimize the loop nest. Since another thread in the same execution context could be modifying the shape of array `A`, we need to make a private copy of that array. We accomplish that by creating a new array of arrays `A'`, and copying into each element of `A'` a pointer to the corresponding row of array `A`. Similar to the versioning test, this operation has complexity proportional to the number of rows of array `A`. This privatization operation is illustrated in Figure 5. The safe and alias-free loop nest in that figure can now be optimized with traditional loop transformations.

Figure 6 shows the code for an implementation of `dgemm` using regular Java arrays. The `dgemm` method computes

$$C \leftarrow \beta C + \alpha\tilde{A} \times \tilde{B},$$

where C, A, and B are matrices of double precision floating-point numbers. $\tilde{A}$ denotes either A^T (if the `transa` flag is **true**) or A (if the `transa` flag is **false**). The same holds for $\tilde{B}$ and the `transb` flag. The computation of the resulting C matrix is done one column at a time. If `transb` is false, then the i-th column of the resulting matrix is computed with a `dgemv` call for matrix A and the i-th column of B. Otherwise, the i-th column of the resulting matrix is computed with a `dgemv` call for matrix A and the i-th row of B. Note that for each value of i, we have to extract the i-th column of C into a `double[]` array, in order to pass it to the `dgemv` method. We then have to copy the resulting vector back into column i of C. We also need to perform a copy to pass the i-th column of B to `dgemv`. This example illustrates the inconveniences

```
public static void dgemv(boolean trans, int m, int n, double alpha,
                         double[][] A, double[] x, double beta, double[] y) {

       if (trans) {
          for (int i=0; i<m; i++) {
              double s = 0;
              for (int j=0; j<n; j++) s += A[j][i]*x[j];
              y[i] = alpha*s + beta*y[i];
          }
       } else {
          if ((m <= y.length) && (n <= x.length) && (n <= A[i].length ∀ i) && (m <= A.length)
              && (y != x) && (y != A[i] ∀ i)) {
              // safe and alias-free version
              for (int i=0; i<m; i++) {
                  double s = 0;
                  for (int j=0; j<n; j++) s += A[i][j]*x[j];
                  y[i] = alpha*s + beta*y[i];
              }
          } else {
              // safe version
              for (int i=0; i<m; i++) {
                  double s = 0;
                  for (int j=0; j<n; j++) s += A[i][j]*x[j];
                  y[i] = alpha*s + beta*y[i];
              }
          }
       }

       return;
}
```

Figure 4: The `dgemv` method with Java arrays and a versioning test for exception safety and alias disambiguation.

of parameter passing with Java arrays of arrays.

3 Multiarray constructs

In this section, we describe syntactic additions, first proposed by Michael Philippsen and Roldan Pozo of the Java Grande Numerics Working Group, to represent multiarrays in Java. By implementing a parser for this new syntax with JavaCUP (`http://www.cs.princeton.edu/~appel/modern/java/CUP`), we have verified that they do not to cause any parsing conflicts with the existing Java syntax. We also discuss how the properties of multiarrays facilitate compiler optimizations and support better library interfaces. In the next sections we will discuss how to actually implement these multiarray constructs in Java.

3.1 Syntax

We need a syntax that differentiates multidimensional arrays from Java arrays of arrays. A reference to a d-dimensional multiarray of type $\langle type \rangle$ is declared as $\langle type \rangle[[*\{,*\}^{d-1}]]$, where $\{\}^{d-1}$ denotes repetition of a pattern $d-1$ times. For example, `double[[*]] x` declares `x` as a reference to a one-dimensional multiarray of `double`s, and `float[[*,*]] Y` declares `Y` as a reference to a two-dimensional multiarray of `float`s.

A d-dimensional multiarray of shape $(n_0, n_1, \ldots, n_{d-1})$ and type $\langle type \rangle$ is created with the construct

`new` $\langle type \rangle[[n_0, n_1, \ldots, n_{d-1}]]$.

```
public static void dgemv(boolean trans, int m, int n, double alpha,
                         double[][] A, double[] x, double beta, double[] y) {

       if (trans) {
          for (int i=0; i<m; i++) {
              double s = 0;
              for (int j=0; j<n; j++) s += A[j][i]*x[j];
              y[i] = alpha*s + beta*y[i];
          }
       } else {
          double[][] A' = new double[m][]; for (int i=0; i<m; i++) A'[i] = A[i];
          if ((m <= y.length) && (n <= x.length) && (n <= A'[i].length ∀ i) && (m <= A'.length)
              && (y != x) && (y != A'[i] ∀ i)) {
              // safe and alias free version
              for (int i=0; i<m; i++) {
                  double s = 0;
                  for (int j=0; j<n; j++) s += A'[i][j]*x[j];
                  y[i] = alpha*s + beta*y[i];
              }
          } else {
              // safe version
              for (int i=0; i<m; i++) {
                  double s = 0;
                  for (int j=0; j<n; j++) s += A[i][j]*x[j];
                  y[i] = alpha*s + beta*y[i];
              }
          }
       }

       return;
}
```

Figure 5: The `dgemv` method with Java arrays, array privatization for thread safety, and a versioning test for exception safety and alias disambiguation.

```
public static void dgemm(boolean transa, boolean transb, int m, int n, int p,
                         double alpha, double[][] A, double[][] B,
                         double beta, double[][] C) {

    double[] c = new double[m];
    double[] b = new double[n];
    for (int i=0; i<p; i++) {
        for (int j=0; j<m; j++) c[j] = C[j][i];
        if (transb) {
            dgemv(transa,alpha,A,B[i],beta,c);
         } else {
            for (int j=0; j<n; j++) b[j] = B[j][i];
            dgemv(transa,alpha,A,b,beta,c);
        }
        for (int j=0; j<m; j++) C[j][i] = c[j];
    }

    return;
}
```

Figure 6: The `dgemm` method with Java arrays.

For example, `x = new double[[n]]` and `Y = new float[[m,n]]` are valid assignments, since the type and rank of the created array matches the type and rank of the reference variable. We note that `new double[[n]]` is different from `new double[n]`. The former creates a one-dimensional multiarray of `double`s, whereas the later creates a (conventional) Java array.

Individual elements are referenced using conventional subscript notation, with single brackets and commas (“,”) separating the indices for the different axes. Array elements can be used either as a value in an expression, or as the target of an assignment. Figure 7 shows the code for the `dgemv` method using the proposed language constructs. The only differences between this code and that of Figure 2 are in (i) the declaration of the multiarrays, (ii) the use of two-dimensional indexing for multiarray A, and (iii) m and n no longer need to be explicitly passed in as parameters.

```
public static void dgemv(boolean trans, double alpha,
                         double[[*,*]] A, double[[*]] x, double beta, double[[*]] y) {

    int m = y.size(0);
    int n = x.size(0);

    if (trans) {
        for (int i=0; i<m; i++) {
            double s = 0;
            for (int j=0; j<n; j++) s += A[j,i]*x[j];
            y[i] = alpha*s + beta*y[i];
        }
    } else {
        for (int i=0; i<m; i++) {
            double s = 0;
            for (int j=0; j<n; j++) s += A[i,j]*x[j];
            y[i] = alpha*s + beta*y[i];
        }
    }

    return;
}
```

Figure 7: The `dgemv` method using multiarray constructs.

The syntax also supports array sections. Whenever a multiarray is expected, it is legal to use a regular section of a multiarray. A regular section is defined by specifying a range of indices or a single index for each axis of a multiarray. A range of indices is an arithmetic progression of index values specified by the form $f : l$ or $f : l : s$, where f represents the first index in the range, l is the last index in the range, and s is the stride (or increment) between consecutive indices. If s is omitted, then it is treated as 1. It is also possible to use just a colon (":") to denote a range, which represents a range going from 0 to largest index for that axis, in steps of 1. The rank of a multiarray section is equal to the rank of the original array minus the number of index constructs that are single values (not ranges). The code for `dgemm` is shown in Figure 8. This example shows more significant differences when compared with Figure 6. We note that the computation of the matrix-multiply is much simplified by our ability to extract sections of multiarrays. We directly pass the i-th column of C to `dgemv`. We also pass either the i-th row or the i-th column of B, depending on the value of `transb`.

Each d-dimensional multiarray A has associated with it a data vector D^A and a d-element *shape vector* n^A (of integers). The data vector D^A is a one-dimensional Java array of the same elemental type as A, and it represents the storage area for the elements of A. The shape vector describes the extents of A along each axis. A reference to D^A can be obtained through method `data()` on multiarray A. The values of n_i^A can be obtained by invoking method `size(`i`)` on multiarray A.

3.2 Performance impact

Java code with multiarrays can be optimized through the same versioning techniques discussed in Section 2. However, as shown in Figure 9, the versioning test is much simpler and faster. Exactly two parameters define the number of rows and columns of array `A`. Alias disambiguation between `y` and `x` and between `y` and `A` requires just a simple comparison between the corresponding data storage pointers. Finally, since the shapes of multiarrays are immutable, privatization is not necessary.

```
public static void dgemm(boolean transa, boolean transb,
                         double alpha, double[[*,*]] A, double[[*,*]] B,
                         double beta, double[[*,*]] C) {

    int p = C.size(1);

    for (int i=0; i<p; i++) {
        if (transb) {
            dgemv(transa,alpha,A,B[[i,:]],beta,C[[:,i]]);
        } else {
            dgemv(transa,alpha,A,B[[:,i]],beta,C[[:,i]]);
        }
    }

    return;
}
```

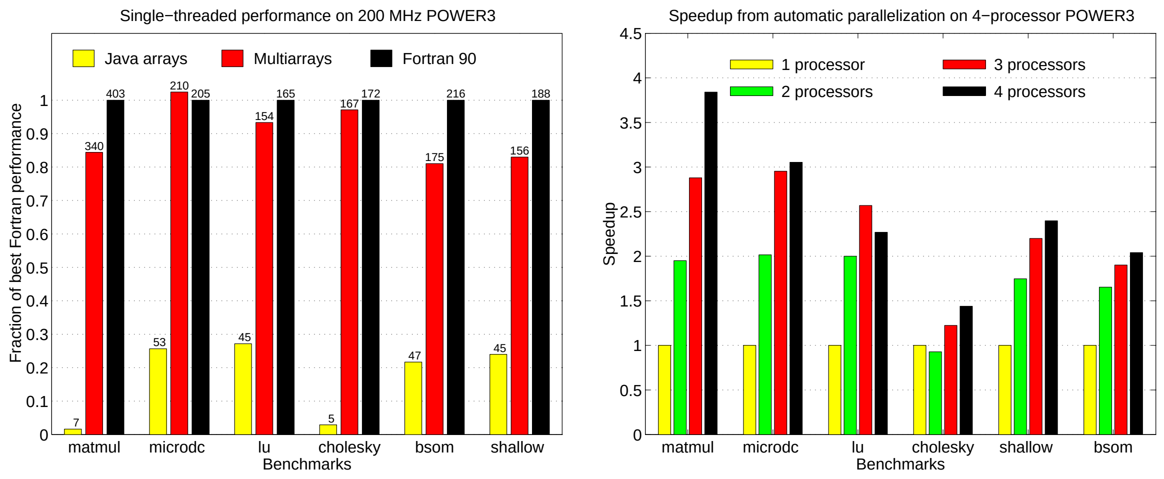
Figure 8: The dgemm method using multiarrays.

```
public static void dgemv(boolean trans, int m, int n, double alpha,
                         double[[*,*]] A, double[[*]] x, double beta, double[[*]] y) {

       if (trans) {
          for (int i=0; i<m; i++) {
              double s = 0;
              for (int j=0; j<n; j++) s += A[j][i]*x[j];
              y[i] = alpha*s + beta*y[i];
          }
       } else {
          if ((m <= y.length) && (n <= x.length) && (n <= A.size(1)) && (m <= A.size(0))
              && (y.data() != x.data()) && (y.data() != A.data())) {
              // safe and alias-free version
              for (int i=0; i<m; i++) {
                  double s = 0;
                  for (int j=0; j<n; j++) s += A[i][j]*x[j];
                  y[i] = alpha*s + beta*y[i];
              }
          } else {
              // unsafe version
              for (int i=0; i<m; i++) {
                  double s = 0;
                  for (int j=0; j<n; j++) s += A[i][j]*x[j];
                  y[i] = alpha*s + beta*y[i];
              }
          }
       }

       return;
}
```

Figure 9: The dgemv method with multiarrays and a versioning test for exception safety and alias disambiguation.

To demonstrate the bottom-line impact of multiarrays in Java, we implemented NINJA [15], a prototype Java compiler that supports multiarrays. NINJA automatically performs versioning of loop nests, and applies high-order loop transformations and parallelization to the safe and alias-free versions of those loops. We applied NINJA to a suite of six numerically-intensive Java benchmarks: matmul (a matrix multiply), microdc (an iterative job solver), lu (an LU factorization), cholesky (a Cholesky factorization), bsom (a neural network data mining kernel), and shallow (a shallow-water simulation). Results for experiments on a 200 MHz POWER3 machine are shown in Figure 10.

Figure 10(a) compares the performance obtained with single-threaded versions of each benchmarks:



(a) serial execution performance (b) parallel execution performance

Figure 10: Performance impact of multiarrays on various numerical benchmarks. Numbers at the top of bars in plot (a) represent measured Mflops.

(i) a Fortran 90 version compiled with a state-of-the-art Fortran compiler, (ii) a version with Java arrays executed with the best commercial Java environment available in the machine at the time of experiment, and (iii) a version with multiarrays, compiled and executed by NINJA. It is clear that the multiarray NINJA version can be several times faster than the Java arrays version, and can achieve from 80 to 100% of the Fortran 90 performance.

Figure 10(b) shows the speedup obtained though automatic parallelization of loops performed by NINJA, when Java code with multiarrays is executed on 1, 2, 3, and 4 processors. Speedups are relative to single-processor performance of each parallel code. We observe that high efficiency is achieved in most benchmarks, with better than 2 speedup on 4 processors for all but cholesky.

4 A multiarray package

The Array package for Java consists of a group of classes that implement true multidimensional arrays. The Array package was designed with several goals in mind: (i) to provide the functionality expected from multiarrays by programmers of numerically intensive applications and (ii) to enable high-performance optimizations by leveraging existing compiler techniques. This section will present a high level description of the Array package and its multiarrays (which will be denoted “Arrays” to distinguish them from generic multiarrays and Java arrays). It also discusses the benefits and costs of incorporating the Array package into Java. A more detailed description of the Array package can be found in [14].

4.1 A description of the Array package

The Array package for Java is a library based implementation of multiarrays. The dense and rectangular shape of multiarrays facilitate the application of automatic compiler optimizations.

The class hierarchy of the Array package is straightforward. The leaves of the hierarchy correspond to final concrete classes, each implementing a multiarray of specific type and rank. For example, `doubleArray2D` is a two-dimensional Array of double precision floating-point numbers. The shape of an Array is defined at object creation time. For example,

```
intArray3D A = new intArray3D(m,n,p);
```

creates an $m \times n \times p$ three-dimensional Array of integer numbers. Defining a specific concrete final class for each Array type and rank effectively binds the semantics to the syntax of a program, enabling the use of mature compiler technology that has been developed for languages like Fortran and C.

A version of `dgemv` with the Array package is shown in Figure 11. We note that it is a straightforward translation of the multiarray code in Figure 7. A version of `dgemm` with the Array package is shown in Figure 12.

```
public static void dgemv(boolean trans, double alpha,
                         doubleArray2D A, doubleArray1D x,
                         double beta, doubleArray1D y) {

  int m = y.size(0);
  int n = x.size(0);

  if (trans) {
    for (int i=0; i<m; i++) {
      double s = 0;
      for (int j=0;j<n;j++) s += A.get(j,i)*x.get(j);
      y.set(i,alpha*s + beta*y.get(i));
    }
  } else {
    for (int i=0; i<m; i++) {
      double s = 0;
      for (int j=0;j<n;j++) s += A.get(i,j)*x.get(j);
      y.set(i,alpha*s + beta*y.get(i));
    }
  }

  return;
}
```

Figure 11: The `dgemv` method with the Array package.

```
public static void dgemm(boolean transa, boolean transb,
                         double alpha, doubleArray2D A, doubleArray2D B,
                         double beta, doubleArray2D C) {

   int p = C.size(1);

   for (int i=0; i<p; i++) {
      if (transb) {
         dgemv(transa, alpha, A, B.section(i, new Range(0,n-1)),
               beta, C.section(new Range(0,m-1), i));
      } else {
         dgemv(transa, alpha, A, B.section(new Range(0,n-1), i),
               beta, C.section(new Range(0,m-1), i));
      }
   }

   return;
}
```

Figure 12: The `dgemm` method with the Array package.

The Array package supports element-wise and aggregate operations on Arrays. For example, computing a two-dimensional Array C of shape $m \times n$, where each element is the sum of the corresponding elements of Arrays A and B (also of shape $m \times n$), can be written as shown in Figure 13. A proposed syntax for aggregate Array operations is shown in Figure 13(c).

```
for (int i=0; i<m; i++)
  for (int j=0; j<n; j++)
    C.set(i,j,A.get(i,j)+B.get(i,j));
```

(a) code using elemental operations

```
C = A.plus(B);
```

(b) code using aggregate operations

```
C = A + B;
```

(c) proposed syntax for aggregate operations

Figure 13: Examples of Array operations

There are subtle differences between the elemental and aggregate forms. The aggregate form enforces *array semantics*: all elements of A and B are first read, the addition is performed, and only then are the resulting values written to the elements of C. The first (element-wise) version computes one element of C at a time. If C happens to share storage with either A or B, the resulting values of elements of C may differ from the aggregate form. A second difference between the aggregate and elemental forms is in the reporting of out-of-bounds or null pointer accesses. The aggregate operations determine if any access implied by the computation of the operation can cause an access exception. If an exception causing accesses is implied, the exception is thrown immediately. This means that aggregate operations either execute completely, or leave the result Array unchanged. In contrast, the elemental form of the operation may change some elements of C before the exception is thrown. Both element-wise and aggregate forms have their merits, and the Array package is designed so that the two forms can be aggressively optimized as with state-of-the-art Fortran compilers.

4.2 A critique of the Array package

To support full Array functionality, accessor methods and special classes for objects to be used as subscripts (`Index` and `Range`) are defined to allow sections of Arrays to be specified. Because the semantic specification of Array operations is contained within the classes of the Array package, no JVM changes are necessary to support these programs, thus insuring portability across existing JVM's.

The Array package specification requires certain classes with well-defined semantics to be supported. Although a reference implementation exists, should the Array package become a standard it is likely, and desirable, that third party implementations will be developed. Because the Array package does not specify how the elements of an Array are laid out (in the spirit of Java arrays), it is possible to implement layouts based on space filling curves and recursive blocking [5, 10, 11]. Some researchers advocate a specific layout for the elements of a multiarray, with the argument that such specification would facilitate the development of performance-portable code. We note that exposing internal object layouts is against Java's philosophy. It also prevents future optimizations arising from advances in data structures and algorithms.

The implementations of the BLAS and other aggregate operations, and various elemental operations can be very sophisticated (and built upon existing libraries using JNI [4, 8, 19]) for applications requiring high performance. Alternatively, a very simple implementation is also possible, leading to a small Array package implementation and allowing it to be downloaded more easily and to fit on clients with much less memory than high performance compute servers.

To obtain maximal performance from programs written with the Array package, some JVM support is necessary. A simple, local optimization called *semantic expansion* [22] allows elemental accesses to be as efficient as in C and Fortran, and allows efficient support of complex numbers and multiarrays of complex numbers.

The design of the Array package makes it very easy for a JVM to optimize programs that use Arrays. Because Arrays are rectangular it is easy to automatically generate efficient multi-version code to create program regions that are free of access violations and amenable to other optimizations.

Array properties allow for a simple and effective run-time aliasing test to be developed. The inclusion of tests at the beginning of aggregate operations to ensure that no access violations occur during the computation of the operation allow low overhead bounds optimizations already present in some dynamic compilers (*e.g.*, the ABCD test [3]) to be very effective.

Without syntactic support, however, the Array package can be cumbersome to use, as seen in Figure 13(a). That code computes `C[i,j] = A[i,j]+B[i,j]`, which is much clearer than the representation using method calls seen in the figure. Syntax support can be included in two ways: by adding operator overloading to Java [7], or by adding syntax to the Java core language. The former is strongly opposed by many because it requires a large change to the basic language, and complicates the use of the language. It has the advantage that the Array package only needs to be hosted on machines making use of the Array package. In the latter approach, a multiarray syntax, as presented in Section 3, would be supported by the Java to bytecode translator, *javac*. When translating multiarrays, *javac* would make the appropriate references to the Array package. This last approach has a smaller effect on the language definition, but, it makes Array operations part of the standard Java language. This in turn requires that the Array package for Java be part of the core Java libraries, and places a high premium on the development of a compact standard implementation. Three promising approaches for developing such a compact representation are: (i) only support a restricted range of array dimensions (*i.e.*, up to three); (ii) within the Array package (and transparently to the user) map all array dimensionalities to the highest supported dimensionality; and (iii) exploit the regular nature of the Array package to have the install package contain a program to generate the Array package classes rather than contain the classes themselves. We note that the latter only minimizes the size of the install package and not the *installed* package, but even this approach will mitigate most of the impact the the Array package size for the Standard and Enterprise editions of Java.

5 Direct translation of multiarrays to bytecode

In this section, we describe another approach to implementing multiarrays in Java. It consists of translating the multiarray syntactic constructs of Section 3 directly to JVM bytecode, without underlying library support. Java-like languages with multiarray constructs have been proposed in [20, 23]. The idea of adding new language constructs to Java that translate directly to bytecode was demonstrated to be effective for complex numbers in [9, 17].

5.1 Translation rules

This section presents translation rules for the new multiarray language constructs directly to bytecode. In the interest of clarity, we illustrate those rules by showing not the bytecode generated, but the Java code equivalent to that bytecode.

Some translation rules are illustrated in Figure 14, which shows the translated code for the `dgemv` code of Figure 7. A d-dimensional multiarray is represented by $2d+2$ values: a data storage pointer, a d-element shape vector, and a $(d+1)$-element weights vector. The weights vector describes the placement of elements in the data vector. Let ω_i^A represent the i-th elements of the weights vector ω^A. Each element of array A

corresponds to one and only one element of the data array D^A according to the following relation:

$$A\left[i_0, \ldots, i_{d-1}\right] \equiv D^A\left[i_0 w_0 + \ldots + i_{d-1} w_{d-1} + w_d\right]. \tag{1}$$

Formal parameter A (a `double[[*,*]]`) is expanded into six parameters (line 2): (i) the storage area `A`; (ii) the shape parameters `A$n0` and `A$n1`; and (iii) the weights parameter `A$w0`, `A$w1`, and `A$w2`. Formal parameters x and y are expanded into four variables, as necessary to represent a one-dimensional multiarray (lines 3 and 5, respectively). In general, each d-dimensional multiarray in the formal parameter list of a method actually counts as $2d+2$ parameters toward the limit of parameters per method that bytecode supports.

```
 1   public static void dgemv(boolean trans, double alpha,
 2                            double[] A, int A$n0, int A$n1, int A$w0, int A$w1, int A$w2,
 3                            double[] x, int x$n0, int x$w0, int x$w1,
 4                            double beta,
 5                            double[] y, int y$n0, int y$w0, int y$w1) {
 6
 7       int m = y$n0;
 8       int n = x$n0;
 9
10       if (trans) {
11           for (int i=0; i<m; i++) {
12                double s = 0;
13                for (int j=0; j<n; j++) s += A[j*A$w0 + i*A$w1 + A$w2]*x[j*x$w0 + x$w1];
14                y[i*y$w0 + y$w1] = alpha*s + beta*y[i*y$w0 + y$w1];
15           }
16       } else {
17           for (int i=0; i<m; i++) {
18                double s = 0;
19                for (int j=0; j<n; j++) s += A[i*A$w0 + j*A$w1 + A$w2]*x[j*x$w0 + x$w1];
20                y[i*y$w0 + y$w1] = alpha*s + beta*y[i*y$w0 + y$w1];
21           }
22       }
23
24       return;
25   }
```

Figure 14: Translation of multiarray code for `dgemv`.

Invocation of the `size` method on a multiarray is replaced by direct access to the shape variables of that multiarray. This is illustrated in lines 7 and 8 of Figure 14. Lines 13, 14, 19, and 20 of Figure 14 illustrate how multiarray element accesses are translated to an indexing operation into the data storage area of the multiarray, according to Equation 1.

It is not possible for a method to return a multiarray, since this would require the return of multiple values (data storage, shape vector, and weights vector). Although this restriction creates an asymmetry in the language, result arrays can be passed as an extra argument to be filled in by the method. From a performance perspective, this is the preferred approach anyway, since it avoids the expensive operation of creating a new array to return the result.

Additional translation rules are illustrated in Figure 15, which shows the translated code for the `dgemm` in Figure 8. Extracting a column or a row of a two-dimensional multiarray is accomplished by computing the appropriate shape and weight vectors for a one-dimensional multiarray. This is illustrated in the calls to `dgemv` in lines 11-15 and 18-21.

5.2 A critique of using direct translation to implement multiarrays

The primary accomplishment of this approach is that it allows true multiarrays to be expressed in Java by modifying the grammar and language specification, leaving untouched the virtual machine specification,

```
    public static void dgemm(
         boolean transa, boolean transb, double alpha,
         double[] A, int A$n0, int A$n1, int A$w0, int A$w1, int A$w2,
         double[] B, int B$n0, int B$n1, int B$w0, int B$w1, int B$w2,
         double beta, double[] C, int C$n0, int C$n1, int C$w0, int C$w1, int C$w2) {

        int p = C$n1;

        for (int i=0; i<p; i++) {
        if (transb) {
            dgemv(transa, alpha,
                  A, A$n0, A$n1, A$w0, A$w1, A$w2,
                  B, B$n1, B$w1, B$w2+i*B$w0,
                  beta,
                  C, C$n0, C$w0, C$w2+i*C$w1);
            }
        } else {
            dgemv(transa, alpha,
                  A, A$n0, A$n1, A$w0, A$w1, A$w2,
                  B, B$n0, B$w0, B$w2+i*B$w1,
                  beta,
                  C, C$n0, C$w0, C$w2+i*C$w1);
            }
        }

        return;
     }
}
```

Figure 15: Translation of multiarray code for `dgemm`.

that is, the semantics of bytecode. Therefore, even though this technique requires fairly significant changes to *javac*, it allows classes to be written and compiled in an environment that has the changed *javac*, with bytecode being produced that can execute on any JVM.

The multiarrays supported allow a limited form of array sections, those which can be represented by triplet notation, to be passed as parameters. More general subsets of rows and columns are not supported, and it is not clear how to support them without greatly complicating both the generation of bytecode and the Java grammar. Moreover, because the supported multiarrays are a collection of objects rather than a single object, they cannot be returned by methods. Other complications include the behavior of `instanceof` and the reflection services.

More positively, multiarrays of any rank are supported by this technique since the generation of bytecode for allocation and access of multiarrays is driven by parsing, and not by the preexistence of classes corresponding to multiarray of the desired rank. For similar reasons, the elements of the multiarray can be any legal Java type.

6 Using an Enhanced JVM to Support Multiarrays

In this section, we discuss an alternate approach that does not require any changes to the language (at the source or bytecode level) or any new classes in the class library. Instead, it relies on the JVM to use dense multidimensional arrays when it is safe to do so. The JVM can internally use a multiarray in place of an array of arrays based on compiler analysis or runtime information, or both.

6.1 Basic approach

The compiler-based approach involves performing a shape analysis of arrays and using a dense representation when the analysis shows that the array is rectangular [6]. This approach can be quite effective on simple cases and enables generation of efficient code. However, it has the drawback of requiring *whole program analysis*, which is not only expensive in compilation time but also impractical in Java due to dynamic language features like dynamic class loading.

A simpler approach is for the JVM to use runtime information, as proposed in [13]. In this approach, a flag is included in the internal representation of an array to indicate if it is dense. The *dense* flag is set when the array is created initially (the JVM bytecode `multianewarray` is used to create a multidimensional array). For instance, consider the following statement:

```
double[][] A = new double[m+1][n+1]
```

The JVM allocates contiguous space to hold all the elements of the two-dimensional array, but also creates an array of row pointers, as shown in Figure 16, allowing the representation to be consistent with the Java language specification. Any time the structure of the array is modified with a direct assignment to a row of the array (such as with statements of the form `A[i] = new double[2*n]` or `A[i] = B[i]`), the dense flag is reset.

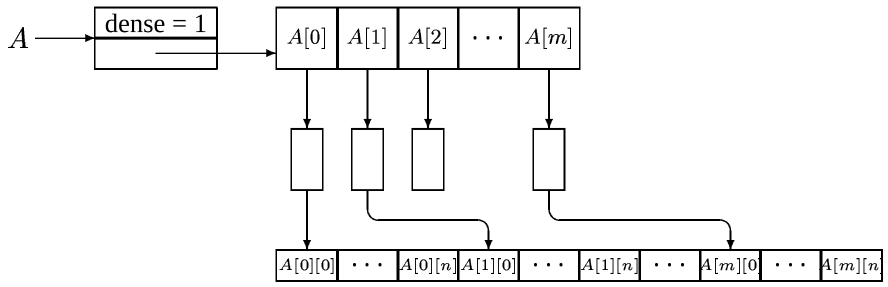


Figure 16: A dense representation for Java arrays.

In general, code handling arrays has to be prepared for the possibility that a multidimensional array is not always dense. This problem may be overcome by applying code versioning, where one version of the code is optimized for the case where the arrays being referenced are dense, while another version handles the case where the arrays are not dense.

6.2 A critique of the enhanced JVM approach

An obvious advantage of this approach is that it requires no changes to the Java language or the JVM specification. The programmer can enjoy the performance benefits of multiarray representation for existing Java programs in a largely transparent manner.

However, the JVMs do need to become more sophisticated (than current JVMs) to detect the *denseness* property of arrays automatically, and there are likely to be many situations when even a sophisticated JVM would fail to detect that it is safe to use a multiarray representation. For instance, in the presence of a call to a native method, a JVM may be forced to assume conservatively that the shape of an array may be changed by the native code. Furthermore, there are programming styles that can cause a given analysis technique to fail. For example, while it is easy to recognize a two-dimensional array created in a single operation:

```
    double[][] a = new double[m][n];
```

it may be more difficult to detect one whose creation is spread over many steps:

```
    double[][] a = new double[m][];
    for (int i=0; i<m; i++)
      a[i] = new double[n];
```

In particular, the simple runtime analysis with the *dense* flag, described in Section 6.1, would fail on the latter code (*i.e.*, it would indicate a nondense array).

A second drawback of this approach is that it offers no new features that make it more convenient to write array-intensive codes. First, high-level aggregate operations like copying or addition of arrays have to be necessarily programmed in terms of loops with element-wise operations. Second, there is no support for creating sections of arrays, and hence, no support for passing array sections as parameters. As an example, for identical computations over a row and a column of an array, it is not possible to directly pass that row or column as an argument to a generic method operating over a vector.

In summary, the approach of relying exclusively on JVM analysis to detect multiarrays has no barriers to acceptance: there are no issues with regard to standardization of any new constructs and no code migration issues. However, this transparency comes at a price of requiring significant sophistication in the JVM to realize good performance, and the likelihood of frequent failures in obtaining the performance benefits, due to factors like calls to non-analyzed code fragments and unexpected programming styles. Furthermore, it supports a much reduced functionality for writing array-based codes in a convenient manner.

7 Comparison

In the previous sections, we discussed three different approaches to “adding” multidimensional arrays (multiarrays) to Java. For each of those approaches, we discussed its intrinsic pros and cons. We now turn to a more comparative analysis, with the goal of establishing an objective classification of those three approaches. We compare them with regards to their impact on (i) language and virtual machine specification (in which we consider less impact to be better), (ii) functionality (more functionality is better), (iii) implementation efforts (less effort is better), and (iv) typical achievable performance (higher performance is better). We summarize our findings for each of those criteria in Table 1, discussing them in more detail below. For purpose of this discussion, we name the proposals **A** (Array package with new syntax), **VM** (better JVM that recognizes implicit multiarrays), and **D** (direct translation to bytecode).

7.1 Impact on specifications

Proposal **VM** (a better virtual machine) is clearly the one with the least impact on current Java and JVM specifications. More precisely, there is no impact, since it is entirely up to the virtual machine internals to recognize and support multiarrays. No new language constructs, class libraries, and/or bytecodes are necessary. Equally important, there are no standardization issues and no compatibility issues. A better virtual machine that, as discussed in Section 6, recognizes and supports true multidimensional arrays will simply execute some numerical codes faster. In principle, application programmers do not have to be aware of what is going on behind the curtains, inside the virtual machine. This total transparency, however, does have a drawback, that we shall discuss later in Section 7.4.

Proposal **D** (direct translation to bytecode) is next in terms of impact to existing specifications. It requires new syntactic constructs to be added to the Java Programming Language. These constructs are strictly additive, since there is no change to the syntax and semantics of existing constructs, a positive feature. The impact is limited to the Java Programming Language, since the bytecode generated by translating the new

Table 1: A summary of the compared characteristics of each approach.

Criteria	better		worse
Impact on specifications	**VM**	**D**	**A**
Functionality	**A**	**D**	**VM**
Implementation efforts (minimal)	**VM**	**D**	**A**
Implementation efforts (performance)	**A**	**D**	**VM**
Achievable performance	**A**	**D**	**VM**

A: Array package with new syntax
VM: Virtual machine that recognizes multiarrays
D: Direct translation to bytecode

constructs is entirely conventional. The Java to bytecode compiler (*javac*) converts multiarrays to the single-dimensional arrays directly supported by bytecode. There are no class libraries that need to be invoked in support of multiarrays with proposal **D**.

Proposal **A** (Array package with new syntax) is the one with most impact to existing specifications. It requires new language constructs, either for general support of operator overloading or dedicated to multiarrays as in proposal **D**. In addition, it requires a fairly sophisticated standard class library to implement the required functionality. If dedicated support to multiarrays is added to the Java Programming Language, then this class library must become part of the core run-time system.

7.2 Functionality

Not surprisingly, the classification of the approaches with respect to functionality is opposite to that with respect to impact on specifications. Proposal **VM** requires no specification changes but it also adds no new features. In particular, proposal **VM** does not support the passing of an array section to a method that takes an array as parameter. This functionality, albeit in a limited form, has been available to numerical programmers since the early days of Fortran. Both proposals **A** and **D** allow the passing of multiarray sections as method parameters.

Proposal **D** does not go much beyond its support of multiarray sections as parameters. Moreover, the multiarrays in proposal **D** exhibit some undesirable characteristics, resulting from the way they are implemented. Multiarrays in proposal **D** are not implemented as individual objects, but as a combination of objects (the storage array) and primitive types (the shape and weights descriptors). What that means is that a multiarray is not derived from `Object`, and therefore it does not behave as an `Object`. We have seen that we cannot return a multiarray from a method, a peculiar asymmetry with respect to other data types. Also, passing a multiarray to a method actually requires passing several parameters at the bytecode level.

Proposal **A** offers the richest and most consistent set of functionalities. First of all, each multiarray is implemented as a single object, which does behave like a Java `Object`. In proposal **A**, multiarrays can be returned from methods like any other data type. Furthermore, aggregate multiarray operations can be trivially translated to the appropriate method invocations. Proposal **A** is the only one that implements comprehensive array language functionality, as one can find in Fortran 90, MATLAB, and APL.

We note that all of the above proposals allow multidimensional arrays to be supported *without* any changes to the JVM specifications on bytecode, thus ensuring backward compatibility with existing JVMs.

7.3 Implementation efforts

At this point, it is important to distinguish between the effort needed to create a minimally compliant implementation and the effort necessary to create a high-performance implementation. After all, the first motivation for introducing multidimensional arrays in Java was to improve the performance of numerical codes. Regarding a minimal compliant implementation, approach **VM** requires no effort, since existing virtual machines already support current Java arrays. Approach **D** requires a new Java to bytecode translator that understands the new multiarray constructs. Approach **A**, in addition to a new translator, requires an accompanying class library.

The classification is quite different when we take into account the effort to produce a high-performance implementation. Independent of the approach, it has been demonstrated that versioning for bounds checking and alias disambiguation, followed by loop transformations, are important techniques for achieving high-performance on Java numerical codes [1]. Approach **A** actually facilitates the application of those techniques, since the method invocations that appear in the bytecode convey all the semantic information of the operations being performed. For both approaches **VM** and **D**, the optimizer must first "recover" the multiarray operations from bytecode that are manipulating either arrays of arrays (approach **VM**) or single-dimensional arrays (approach **D**). In addition, for approach **VM**, the optimizer has to determine that an array of arrays actually implements a true multidimensional array before proceeding with optimizations.

7.4 Typical achievable performance

In some situations, all three approaches can deliver the same performance when equipped with comparable optimization engines. More important, however, is to consider how each approach behaves with typical user code.

Let us first consider approach **VM**. We expect this approach to be the least robust in terms of its ability to use the (dense) multiarray representation and obtain the attendant performance benefits. As discussed in Section 6.2, simple factors like the presence of native methods and unexpected programming styles can prevent even a sophisticated JVM from employing the multiarray representation. The inability to recognize a multiarray may jeopardize many important optimizations. The feature of transparency to the programmer, supported by approach **VM**, also leads to a lack of guidelines to the application programmer in the development of efficient code.

Both approaches **A** and **D** encourage the application programmer to write code with arrays that are known to be rectangular and dense. Approach **A** has additional advantages. Aggregate multiarray operations, supported by approach **A**, are translated to method invocations. Those method invocations can be implemented efficiently by a smart virtual machine, through semantic expansion, or by native code accessed through JNI. In approach **D**, only individual array element operations appear in the bytecode. Furthermore, the implementation of multiarrays in approach **A** is completely hidden from the programmer, even at the bytecode level. Hiding those details opens the possibility to new optimizations for Java. In particular, we have discussed the benefits of using a block recursive organization of the array elements.

8 Conclusions

It is not a goal of this paper to pick a winner among the three approaches. Nevertheless, we can draw some objective conclusions that help identify and develop good multidimensional array solutions for Java.

First, it is clear that one of the approaches discussed in this paper, direct translation to bytecode, represents a compromise. It is not as transparent as the "enhanced virtual machine" but it is not as intrusive to existing specifications as the Array package approach. It provides some features for numerical programmers, but not as many as the Array package approach. The effort to implement a high-performance system with the direct translation approach is halfway between the other two approaches, as is the typical performance numerical programmers can expect to achieve.

Nothing prevents someone from writing a compiler that translates Java augmented with operator overloading and/or the new language constructs to regular bytecode that can be executed by a Java virtual machine. Correspondingly, anyone can release a multiarray class library in Java. However, for new features to gain popularity and acceptance, it is important to have standardization. The asymmetry in the multiarray constructs (not being returnable, not behaving as `Object`) with the direct translation approach may seriously discourage their adoption as part of the official Java language. The typically large sizes of Array package implementations may also discourage their inclusion in the core run-time system of virtual machines, thus posing difficulties to extending the language with Array package-specific syntax.

Enhancing a virtual machine with the capability to identify implicit multidimensional arrays will result in a performance improvement for some programs. Our claim, however, is that this approach is not enough in terms of achieving the highest level of performance on most codes and delivering the functionality that numerical programmers expect.

Finally, we have argued that the choice of the best approach for supporting multiarrays depends on the relative importance attached to various metrics. If greater functionality for array-based computations and high levels of performance are considered to be the most important, the Array package based approach is clearly the best approach. If having the smallest impact on the language and virtual machine specification is deemed much more important than improved functionality and high performance for a wider range of array codes, the enhanced JVM approach works best. If it is considered important to take the middle ground on all of these metrics, the direct translation approach turns out to be the most attractive approach.

Acknowledgments: The authors with to thank Ron Boisvert, John Caron, Sid Chatterjee, Joe Darcy, Mikel Lujan, Michael Philippsen, and Roldan Pozo, of the Java Grande Numerics Working Group, for valuable contributions to the development of multidimensional arrays in Java.

References

[1] P. V. Artigas, M. Gupta, S. P. Midkiff, and J. E. Moreira. Automatic loop transformations and parallelization for Java. In *Proceedings of the 14th ACM International Conference on Supercomputing*, pages 1–10, Santa Fe, NM, May 2000.

[2] U. Banerjee. Unimodular transformations of double loops. In *Proc. Third Workshop on Programming Languages and Compilers for Parallel Computing*, Irvine, California, August 1990.

[3] Rastislav Bodik, Rajiv Gupta, and Vivek Sarkar. ABCD: Eliminating array bounds checks on demand. *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, June 2000.

[4] R. F. Boisvert, J. J. Dongarra, R. Pozo, K. A. Remington, and G. W. Stewart. Developing numerical libraries in Java. *Concurrency, Pract. Exp. (UK)*, 10(11-13):1117–29, September-November 1998. ACM 1998 Workshop on Java for High-Performance Network Computing. URL: `http://www.cs.ucsb.edu/conferences/java98`.

[5] S. Chatterjee, A. R. Lebeck, P. K. Patnala, and M. Thottenthodi. Recursive array layouts and fast parallel matrix multiplication. In *Proceedings of the 11th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 222–231, Saint-Malo, France, June 1999.

[6] M. Cierniak and W. Li. Just-in-time optimization for high-performance Java programs. *Concurrency, Pract. Exp. (UK)*, 9(11):1063–73, November 1997.

[7] J. D. Darcy. Borneo: Adding IEEE 754 floating point support to Java. Master's thesis, Computer Science Division, University of California, Berkeley, May 1998.

[8] 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. URL: `http://www.cs.ucsb.edu/conferences/java98`.

[9] Edwin Günthner and Michael Philippsen. Complex numbers for Java. *Concurrency: Practice and Experience*, 12(6):477–491, May 2000.

[10] 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.

[11] I. Kodukula, N. Ahmed, and K. Pingali. Data-centric multi-level blocking. *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 346–357, 1997.

[12] J. E. Moreira et al. JSR-083, Java^{TM} Multiarray Package. URL: `http://java.sun.com /aboutJava/communityprocess/jsr/jsr_083_multiarray.html`.

[13] J. E. Moreira, S. P. Midkiff, and M. Gupta. From flop to megaflops: Java for technical computing. *ACM Transactions on Programming Languages and Systems*, 22(2):265–295, March 2000. IBM Research Report RC 21166.

[14] J. E. Moreira, S. P. Midkiff, M. Gupta, P. V. Artigas, M. Snir, and R. D. Lawrence. Java programming for high performance numerical computing. *IBM Systems Journal*, 39(1):21–56, 2000. IBM Research Report RC21481.

[15] J.E. Moreira, S.P. Midkiff, M. Gupta, P. Artigas, P. Wu, and G. Almasi. The NINJA project: Making Java work for high performance computing. *Communications of the ACM*, 2000. submitted.

[16] D. A. Padua and M. J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, December 1986.

[17] M. Philippsen and E. Günthner. cj: A new approach for the efficient use of complex numbers in Java. URL: `http://wwwipd.ira.uka.de/~gunthner/`.

[18] 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.

[19] 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. URL: `http://www.cs.ucsb.edu/conferences/java98`.

[20] C. van Reeuwijk, F. Kuijlman, H.J. Sips, and S.V. Niemeijer. Data-parallel programming in Spar/Java. In *Proceedings of the Second Annual Workshop on Java for High-Performance Computing, Santa Fe, New Mexico*, pages 51–66, May 2000.

[21] M. E. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471, October 1991.

[22] P. Wu, S. P. Midkiff, J. E. Moreira, and M. Gupta. Efficient support for complex numbers in Java. In *Proceedings of the 1999 ACM Java Grande Conference*, 1999. IBM Research Report RC21393.

[23] Yelick, Semenzato, Pike, Miyamoto, Liblit, Krishnamurthy, Hilfinger, Graham, Gay, Colella, and Aiken. Titanium: A high-performance Java dialect. In *ACM 1998 Workshop on Java for High-Performance Network Computing*. ACM SIGPLAN, 1998. URL: `http://www.cs.ucsb.edu/conferences/java98`.