A Comparison of Three Approaches to Language, Compiler, and Library Support for Multidimensional Arrays in Java

José E. Moreira      Samuel P. Midkiff      Manish Gupta
{jmmoreira, 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(TM) has been recognized as a major deficiency in the language’s applicability to numerical computing. The typical approach to adding multidimensional arrays to Java has been through class libraries that implement these structures. It has been shown that the class library approach can achieve very high-performance for numerical computing, through the use of compiler techniques and efficient implementations of aggregate array operations. Because of the inconvenience of accessing array elements through method invocations, it is advocated by many that class libraries for multidimensional arrays should be combined with new language syntax to facilitate manipulation of those multidimensional arrays. Another 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. This approach can also deliver good performance, but it does not improve the existing interfaces for numerical computing. There is yet a third approach: extending the Java language with new syntactic constructs for multidimensional arrays and directly compiling those constructs to bytecode. The new constructs provide a more convenient interface for numerical computing, without requiring a matching class library. This paper is a comparative discussion of the three approaches to adding multidimensional arrays to Java mentioned above. We present a description of the three approaches, listing the pros and cons of each. We give a more detailed description of the third approach – language constructs translated to bytecode – as it is a new contribution. We compare each of the approaches with regards to functionality, impact on the language and virtual machine specification, implementation efforts, and typical achievable performance. We show that the best choice depends on the relative importance attached to the above metrics.

1. INTRODUCTION
The Java Programming Language(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 the support for multidimensional arrays, Java provides only single-dimensional arrays. To a certain extent, it is possible to simulate multidimensional arrays with arrays of arrays. For example, double[][] is an array of one-dimensional arrays of doubles, which can be used to simulate two-dimensional arrays. Figure 1 illustrates the concept of arrays of arrays. This approach, however, leaves much to be desired.

Arrays of arrays are not necessarily rectangular. Arrays of arrays can lead to both inter- and intra-array aliasing. The structure of arrays of arrays can change during computation. All those 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.

The performance and functional benefits of augmenting Java with true multidimensional arrays have already been demonstrated [2]. 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 [1]. 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 defi-
ciency 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 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 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.

This paper introduces a third approach to supporting true multidimensional arrays in Java. We propose a set of language constructs for the declaration and manipulation of multidimensional arrays. We also 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. The new constructs simply add to existing Java syntax. All currently valid Java programs remain valid with their current semantics. The new constructs have been verified not to cause parsing conflicts, and can be translated to verifiable Java bytecode.

Given the current efforts for standardization of multidimensional arrays in Java, including JSR-083, we feel it is important to have a discussion within the community of the relevant issues. The major goal of this paper is to deliver a comparative discussion of the three approaches to adding multidimensional arrays to Java discussed above. We present a description of the three approaches, listing the pros and cons of each. We give a more detailed description of the third approach – language constructs translated to bytecode – as it is a new contribution. 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. Although it is not our goal in this paper to pick a winner, this comparison leads to an objective classification of these three approaches. It forms the basis for a rational decision regarding their implementation and deployment.

To avoid confusion, from now on we refer to multidimensional arrays as multiarrays. Multiarrays are n-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\(_j\).

As a running example, we consider the implementation of two methods from BLAS: \texttt{dgemv} and \texttt{dgemm}. Figure 2 shows the code for an implementation of \texttt{dgemv} using regular Java arrays. This method computes

\[ y \leftarrow \alpha A^T x + \beta y, \]

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

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

    if (trans) {
        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 {
        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 \texttt{dgemv} method with Java arrays.

Figure 3 shows the code for an implementation of \texttt{dgemm} using regular Java arrays. The \texttt{dgemm} method computes

\[ C \leftarrow \beta C + \alpha A^T \times B', \]

where C, A, and B are matrices of double precision floating-point numbers. A\(^T\) denotes either A\(^T\) (if the \texttt{transa} flag is \texttt{true}) or A (if the \texttt{transa} flag is \texttt{false}). The same holds for \texttt{B'} and the \texttt{transb} flag. The computation of the resulting C matrix is done one column at a time. If \texttt{transb} is false, then the i-th column of the resulting matrix is computed with a \texttt{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 \texttt{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 \texttt{dgemv} method. We then have to copy the resulting vector back into column i of C.

The rest of this paper is organized as follows. Section 2 discusses how to implement multiarrays in Java with a class library. Section 3 discusses the approach of relying exclusively on the virtual machine to identify implicit multiarrays. Section 4 presents the new proposal with language constructs for the declaration and manipulation of multidimensional arrays. Section 5 is a comparative discussion of those three approaches. Finally, Section 6 presents our conclusions.
public static void dgemm(boolean transa,
        boolean transb, double alpha, double[][] A,
        double[][] B, double beta, double[][] C) {
    int m = C.length;
    int p = C[0].length;
    int n = transb ? B[0].length : B.length;
    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) {
            for (int j=0; j<n; j++) b[j] = B[i][j];
        } 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];
}

Figure 3: The dgemm method with Java arrays.

2. USING THE ARRAY PACKAGE AND LANGUAGE EXTENSIONS TO SUPPORT MULTIAARRAYS

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].

2.1 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. Concrete classes for Arrays with elements of type type derive from the class typeArray, which in turn derives from the base class Array. Thus doubleArray2D is a two-dimensional Array of double precision floating-point numbers that derives from doubleArray. 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 dgemm with the Array package is shown in Figure 4(a). It is clear that some syntactic support is necessary to make programming with the Array package more appealing. At the very minimum, some support to represent element access with the more conventional “[]” notation is necessary. The same dgemm code with the proposed multiarray syntax is shown in Figure 4(b). We note that both codes get translated to the same bytecode, with the get and set method invocations. The multiarray syntax can be extended to support multiarray sections as described in Section 4.1.

    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;
    }

(a) explicit accessor methods

Figure 4: The dgemv 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 5. A proposed syntax for aggregate Array operations is shown in Figure 5(c).

    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[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.get(i,j)*x.get(j);
                y[i] = alpha*s + beta*y[i];
            }
        }
        return;
    }

(b) proposed syntax for element access

Figure 5: The dgemv method with the Array package.
gate 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.

2.2 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 [6, 11, 12]. Some researchers advocate a specific layout for the elements of a multiarray, with the argument that such specification would allow efficient support of complex numbers and multiarrays of complex numbers.

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 [4]) to be very effective.

Without syntactic support, however, the Array package can be cumbersome to use, as seen in Figure 5(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 [8], 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. The latter 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. It is our feeling that the development of a very compact implementation of the Array package is possible, and will mitigate the impact of needing to include the Array package in the core Java libraries.

3. 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.

3.1 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 [7]. 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:

```java
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 6, 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 6: A dense representation for Java arrays.](image)

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.

### 3.2 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:

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

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

```java
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 3.1, 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 as well 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 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.

### 4. LANGUAGE CONSTRUCTS FOR MULTIARRAYS

In this section, we describe our new approach to multiarrays in Java. It consists of adding new language constructs that represent multiarrays and operations on those multiarrays. Those constructs are then translated directly to JVM bytecode. Java-like languages with multiarray constructs have been proposed in [17, 19]. The idea of adding new language constructs to Java that translate directly to bytecode was demonstrated to be effective for complex numbers in [10, 15].

#### 4.1 Syntax

We need a syntax that differentiates multidimensional arrays from Java arrays of arrays. A reference to a `d`-dimensional multiarray of type `(type)` is declared as `(type) [type] d` i.e. `Y[x, *]` declares `Y` as a reference to a one-dimensional multiarray of doubles, and `float[*] Y` declares `Y` as a reference to a two-dimensional multiarray of floats.

A `d`-dimensional multiarray of shape `(n0, n1, ..., nd-1)` and type `(type)` is created with the construct...
new {type}[n0, n1, ..., nd-1].

For example, x = new double[n] and Y = new float[m,n] are all valid assignments, since the type and rank of the created array matches the type and rank of the reference variable. We note that x = new double[n] actually creates a Java one-dimensional array of doubles. A reference to a Java one-dimensional array can be converted to a one-dimensional multiarray of the same type. (The reverse is not true.)

Individual elements are referenced using conventional subscript notation, with 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 difference between this code and that of Figure 2 are in (i) the declaration of the multiarrays, (ii) the use of the size method to determine the shape of the multiarrays, and (iii) the use of two-dimensional indexing for multiarray A.

```
public static void dgemv(boolean trans, double alpha, double[*,*] A, double[*,*] B, double beta, double[*,*] C) {
    int m = C.size(0);
    int n = C.size(1);

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

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

We also provide limited support of array sections. Whenever a formal parameter is a multiarray, it is possible to pass as an actual parameter a regular section of a multiarray. The regular section must be of the right rank and type. A regular section is defined by specifying a range of indices or a single index for each axis of a multiarray. (In the limiting case, all indices are single values, and the construct no longer specifies a multiarray section but a multiarray element.) 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 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 dgemv is shown in Figure 8. This example shows more significant differences when compared with Figure 3. 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.

```
public static void dgemv(boolean trans, double alpha, double[*,*] A, double[*,*] B, double beta, double[*,*] C) {
    int m = C.size(0);
    int n = C.size(1);

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

**Figure 8: The dgemv method using multiarrays.**

Each d-dimensional multiarray $A$ has associated with it a data vector $D^A$, a d-element shape vector $n^A$ (of integers), and a $(d+1)$-element weights vector $w^A$ (also of integers). The data vector $D^A$ is implemented as 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. The weights vector describes the placement of elements in the data vector. Let $w^A_i$ represent the i-th element of the weights vector $w^A$. Each element of array $A$ corresponds to one and only one element of the data array $D^A$ according to the following relation:

$$A[i_0, ..., i_{d-1}] \equiv D^A[i_0w_0 + \ldots + i_{d-1}w_{d-1} + w_d].$$

(1)

There is a set of query functions that let the programmer determine the shape and organization of a multiarray. A reference to $D^A$ can be obtained through field data of multiarray $A$. The values of $n^A_i$ and $w^A_i$ can be obtained by invoking methods size(i) and weight(i) respectively, where i is an integer literal.

### 4.2 Translation rules

This section presents the translation rules for the new multiarray language constructs. 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 9, which shows the translated code for the dgemm code of Figure 7. As discussed in Section 4.1, 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. 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.

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 9. Correspondingly, invocation of the weight method (not shown in the example) is replaced by direct access to the weight variables of the multiarray. Lines 13, 14, 19, and 20 of Figure 9 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 10, 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 17-21 and 24-28.

```
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$w2,
    double[] B, int B$w0, int B$w1,
    int B$w2, int B$w3, int B$w4,
    double beta,
    double[] C, int C$n0, int C$n1,
    int C$w0, int C$w1, int C$w2) {
    int m = C$n0;
    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,$B$w1,B$w2+1*$B$w0,
                beta,
                C,$C$w0,$C$w2+1*$C$w1);
        } else {
            dgemv(transa, alpha,
                A,$A$w0,$A$w1,$A$w2,
                B,$B$w0,$B$w2+1*$B$w1,
                beta,
                C,$C$w0,$C$w2+1*$C$w1);
        }
    }
    return;
}
```

**Figure 10: Translation of multiarray code for dgemm.**

### 4.3 A critique of using language changes to implement multiarrays

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

### 5. DISCUSSION

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 L (new language constructs).

#### 5.1 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 3, 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 5.4.
public static void dgemv(boolean trans, double alpha,
    double[] A, int A$n0, int A$n1, int A$w0, int A$w1, int A$w2,
    double[] x, int x$n0, int x$w0, int x$w1,
    double beta,
    double[] y, int y$n0, int y$w0, int y$w1) {

    int m = y$n0;
    int n = x$n0;

    if (trans) {
        for (int i=0; i<m; i++) {
            double s = 0;
            for (int j=0; j<n; j++) s += A[j*A$w0 + i*A$w1 + A$w2]*x[j*x$w0 + x$w1];
            y[i*y$w0 + y$w1] = alpha*s + beta*y[i*y$w0 + y$w1];
        }
    } else {
        for (int i=0; i<m; i++) {
            double s = 0;
            for (int j=0; j<n; j++) s += A[i*A$w0 + j*A$w1 + A$w2]*x[j*x$w0 + x$w1];
            y[i*y$w0 + y$w1] = alpha*s + beta*y[i*y$w0 + y$w1];
        }
    }

    return;
}

Figure 9: Translation of multiarray code for dgemv.

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

<table>
<thead>
<tr>
<th>Criteria</th>
<th>better</th>
<th>worse</th>
</tr>
</thead>
<tbody>
<tr>
<td>Impact on specifications</td>
<td>VM</td>
<td>L</td>
</tr>
<tr>
<td>Functionality</td>
<td>A</td>
<td>L</td>
</tr>
<tr>
<td>Implementation efforts (minimal)</td>
<td>VM</td>
<td>L</td>
</tr>
<tr>
<td>Implementation efforts (performance)</td>
<td>A</td>
<td>L</td>
</tr>
<tr>
<td>Achievable performance</td>
<td>A</td>
<td>L</td>
</tr>
</tbody>
</table>

A: Array package with new syntax
VM: Virtual machine that recognizes multiarrays
L: New language constructs

Proposal L (new language constructs) 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 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 L.

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 L. 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.

5.2 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 L allow the passing of multiarray sections as method parameters.

Proposal L does not go much beyond its support of multiarray sections as parameters. Moreover, the multiarrays in proposal L exhibit some undesirable characteristics, resulting from the way they are implemented. Multiarrays in proposal L 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.
5.3 Implementation efforts

At this point, it is important to distinguish between the efforts necessary for creating a minimally compliant implementation and the effort necessary to create a high-performance implementation. After all, the first motivation for introducing multi-dimensional 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 L 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 [3]. 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 L, the optimizer must first “recover” the multiarray operations from bytecode that are manipulating either arrays of arrays (approach VM) or single-dimensional arrays (approach L). 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.

5.4 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 3.2, 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, followed by approach VM, also leads to a lack of guidelines to the application programmer in the development of efficient code.

Both approaches A and L 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 L, 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.

6. 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 the new approach introduced in this paper, multiarray language constructs that translate directly 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 new language constructs 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. 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) 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 newly proposed approach based on language extensions turns out to be the most attractive approach.

7. REFERENCES


