The NINJA Project: Making Java Work for High Performance Numerical Computing

José E. Moreira* Samuel P. Midkiff Manish Gupta
IBM Thomas J. Watson Research Center

Pedro Artigas
School of Computer Science, Carnegie Mellon University

Peng Wu George Almasi
Department of Computer Science, University of Illinois

1 Introduction

When Java(TM) was first introduced, there was a perception (properly founded at the time) that its many benefits came at a significant performance cost. In few areas were the performance deficiencies of Java so blatant as in numerical computing. Our own measurements, with second-generation Java virtual machines, showed differences in performance of up to one hundred-fold relative to C or Fortran. The initial experiences with such poor performance caused many developers of high performance numerical applications to reject Java out-of-hand as a platform for their applications.

Despite the rapid progress that has been made in the past few years, the performance of commercially available Java platforms is not yet on par with state-of-the-art Fortran and C compilers. Programs using complex arithmetic exhibit particularly bad performance. Furthermore, current Java platforms are incapable of automatically applying important optimizations for numerical code, such as loop transformations and automatic parallelization [12]. Nevertheless, our thesis is that there are no technical barriers to high performance computing in Java. To prove this thesis, we have developed a prototype Java environment, called Numerically INtensive JAva (NINJA), which has demonstrated Fortran-like performance can be obtained by Java on a variety of problems. We have successfully addressed issues such as dense and irregular matrix computations, calculations with complex numbers, automatic loop transformations, and automatic parallelization. Moreover, our techniques are straightforward to implement, and allow reuse of existing optimization components already deployed by software vendors for other languages [9], lowering the economic barriers to Java's acceptance.

The next challenge for numerically intensive computing in Java is convincing developers and managers in this domain that Java's benefits can be obtained with performance comparable with highly tuned Fortran and C. This treatise contributes to overcoming that challenge. Once it is accepted that Java performance is only an artifact of particular implementations of Java, and that there are no technical barriers to Java achieving excellent numerical performance, our techniques will allow vendors and researchers to quickly deliver high performance Java platforms to program developers.

*The corresponding author may be reached at `jmoreira@us.ibm.com`, or P.O. Box 218, Yorktown Heights, NY 10598.

2 Java Performance Difficulties

Among the many difficulties associated with optimizing numerical code in Java, we identify three characteristics of the language that are, in a way, unique: (i) exception checks for `null`-pointer and out-of-bounds array accesses, combined with a precise exception model, (ii) the lack of regular-shaped arrays, and (iii) weak support of complex numbers and other arithmetic systems. We discuss each of these in more detail.

The Java exception model: Java requires all array accesses to be checked for dereferencing via `null`-pointer and out-of-bounds indices. An exception must be thrown if either violation happens. Furthermore, the precise exception model of Java states that when the execution of a piece of code throws an exception, all the effects of those instructions prior to the exception must be visible, and no effect of instructions after the exception should be visible [6]. This has a negative impact on performance in two ways: (i) checking the validity of array references contributes to runtime overhead, and (ii) code reordering in general, and loop iteration reordering in particular, is prohibited, thus preventing almost all optimizations for numerical codes. The first of these problems can be alleviated by aggressive hardware support that masks the direct cost of the tests. The second problem is more serious and requires compiler support.

Arrays in Java: Unlike Fortran and C, Java has no direct support for truly rectangular multidimensional arrays. Java allows some simulation of multidimensional arrays through arrays of arrays, but that is not an ideal solution. Arrays of arrays have two major problems.

First, arrays of arrays are not necessarily rectangular. Determining the shape of an array of arrays is, in general, an expensive runtime operation. Even worse, the shape of an array of arrays can change during computation. Figure 1(a) shows an array of arrays being used to simulate a rectangular two-dimensional array. In this case, all rows have the same length. However, arrays of arrays can be used to construct far more complicated structures, as shown in Figure 1(b). We note that such structures, even if unusual for numerical codes, may be natural for other kinds of applications. When a compiler is processing a Java program, it must assume the most general case for an array of arrays unless it can prove that a simpler structure exists. Determining rectangularity of an array of arrays is a difficult compiler analysis problem, bound to fail in many cases. One could advocate the use of pragmas to help identify rectangular arrays. However, to maintain the overall safety of Java, a virtual machine must not rely on pragmas that it cannot independently verify, and we are back to the compiler analysis problem. It would be much simpler to have data structures that make this property explicit, such as the rectangular two-dimensional arrays of Figure 1(c).

Knowing the shape of a multidimensional array is necessary to enable some key optimizations that we discuss below. As can be seen in Figure 1(b), the only way to determine the minimum length of a row is to examine all rows. In contrast, determining the size of a true rectangular array, as shown in Figure 1(c), only requires looking at a single length.

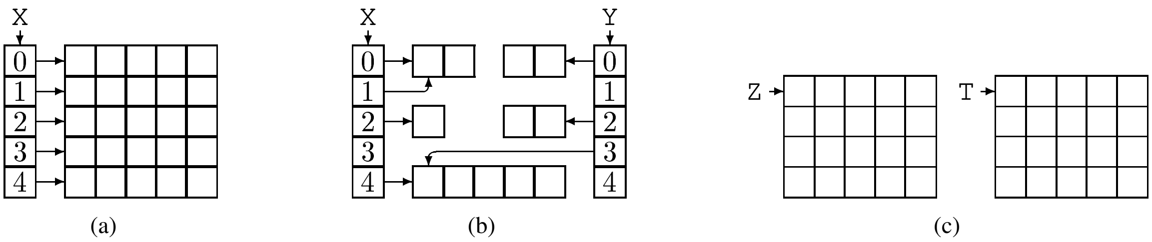


Figure 1: Examples of (a) array of arrays simulating a two-dimensional array, (b) array of arrays in a more irregular structure, and (c) rectangular two-dimensional array.

Second, arrays of arrays may have complicated aliasing patterns, with both intra- and inter-array aliasing. Again, alias disambiguation – that is, determining when storage locations are not aliased – is a key enabler of various optimization techniques, such as loop transformations and loop parallelization, which are so important for numerical codes. The aliasing problem is illustrated in Figure 1. For the arrays of arrays shown in Figure 1(b), two different arrays can share rows, leading to *inter-array* aliasing. In particular, row 4 of array `X` and row 3 of array `Y` refer to the same storage, but with two different names. Furthermore, *intra-array* aliasing is possible, as demonstrated by rows 0 and 1 of array `X`. For the true multidimensional arrays shown in Figure 1(c) (`Z` and `T`), alias analysis is easier. There can be no intra-array aliasing for true multidimensional arrays, and inter-array aliasing can be determined with simpler tests [12].

Complex numbers in Java: From a numerical perspective, Java only has direct support for real numbers. Fortran has direct support for complex numbers. Both Fortran and C++ provide a means for efficiently supporting other arithmetic systems. Support for complex numbers in Fortran and C++ comes from the ability to represent low-cost data structures that can be efficiently allocated on the stack or in registers. Java, in contrast, represents any non-primitive data type as a full fledged object. Complex numbers are typically implemented as objects of a class `Complex`, and every time an arithmetic operation generates a new complex value, a new `Complex` object has to be allocated. That is true even if the value is just a temporary, intermediate result. An array of n complex numbers requires the creation of n objects of type `Complex`. We have observed the largest differences in performance between Java and Fortran when executing code that manipulates arrays of complex numbers. Because `Complex` objects are created at each arithmetic operation, almost all of the execution time of an application with complex numbers is spent creating and garbage collecting `Complex` objects used to hold intermediate values.

The three difficulties described above are at the core of the performance deficiencies of Java. They prevent the application of mature compiler optimization technology to Java and, thus, prevent it from being truly competitive with more established languages such as Fortran and C. We next describe our approach to eliminating these difficulties, and we will show that, with the proper technology, the performance of Java numerical code can be as good as with any other language.

3 Java Performance Solutions

Our research showed that the performance difficulties of Java could be solved by a careful combination of language and compiler techniques. We developed new class libraries that "enrich" the language with some important constructs for numerical computing. Our compiler techniques take advantage of these new constructs to perform automatic optimizations. Above all, we were able to overcome the Java performance problems mentioned earlier while maintaining full portability of Java across all virtual machines.

The Array package and semantic expansion: To attack the absence of truly multidimensional arrays in Java, we have defined an Array package with multidimensional arrays (denoted in this text as *Arrays*, with a capital *A*) of various types and rank (*e.g.*, `doubleArray2D`, `ComplexArray3D`, `ObjectArray1D`). See the sidebar *The Array package for Java* for further discussion. Element accessor methods, sectioning operations, gather and scatter operations, and basic linear algebra subroutines (BLAS) are some of the operations defined for the Array data types. The Arrays have an immutable rectangular and dense shape, which simplifies testing for aliases and facilitates the optimization of runtime checks. The Array classes are written in fully compliant Java code, and can be run on any JVM. This ensures that programs written using the Array package are portable.

When Array elements are accessed via the get and set element operations, each element access will be encumbered by the overhead of a method invocation, which is unacceptable for high performance computing. This problem is avoided by a compiler technique known as *semantic expansion*. In semantic expansion, the compiler looks for specific method calls, and substitutes efficient code for the call. In the case of the Array get and set operations, code identical to that generated for C or Fortran subscripting operations is substituted for the call to the get or set accessor. This allows programs using the Array package to have high performance when executed on JVM's that recognize the Array package methods.

The Complex class and semantic expansion: A complex number class is also defined as part of the Array package, along with methods implementing arithmetic operations on complex numbers. Again, semantic expansion is used to convert calls to these methods into code that uses a *value-object* version of `Complex` objects (containing only the primitive values, not the object headers). Any computation involving methods that can be semantically expanded in this manner can now use complex values, with conversion to `Complex` objects done in a lazy manner upon encountering a method or primitive operation that truly requires object-oriented functionality. Thus, the programmer continues to treat complex numbers as objects (maintaining the clean semantics of the original language), while our compiler transparently transforms them into value-objects for efficiency.

Versioning for safe and alias-free regions: For Java programs written with the Array package, the compiler can perform simple transformations that eliminate the performance problems caused by Java's precise exception model. The idea is to create regions of code that are guaranteed to be free of exceptions. Once these exception-free (also called *safe*) regions have been created, the compiler can apply traditional core-reordering optimizations, constrained only by data and control dependences [12]. The safe regions are created by *versioning* of loop nests. For each optimized loop nest, the compiler creates two versions – safe and unsafe – guarded by a runtime test. This runtime test establishes whether all Arrays in the loop nest are valid (not `null`), and whether all the indexing operations inside the loop will generate in-bound accesses. If the tests passes, the safe version of the loop is executed. If not, the unsafe version is executed. Since the safe version cannot throw an exception, explicit runtime checks can be omitted from the code.

We take the versioning approach a step further. Application of automatic loop transformation (and parallelization) techniques by a compiler requires, in general, alias disambiguation among the various arrays referenced in a loop nest. We rely on a key property of Java that two object references (the only kinds of pointers allowed in Java) must either point to identical or completely non-overlapping objects. Use of the Array package facilitates checking for aliasing by representing a multidimensional array as a single object. Therefore, we can further specialize the safe version of a loop nest into two variants: (i) one in which all multidimensional arrays are guaranteed to be distinct (no aliasing), and (ii) one in which there may be aliasing between arrays. The safe and alias-free version is the perfect target for compiler optimizations. The mature loop optimization techniques, including loop parallelization, that have been developed for Fortran and C programs can be easily applied to the safe and alias-free region.

An example of the versioning transformations to create safe and alias-free regions is shown in Figure 2. Figure 2(a) illustrates the original code, explicitly showing all `null` pointer and array bounds runtime checks that are performed. Figure 2(b) illustrates the versioned code. A simple test for the values of the A and B pointers and a comparison between loop bounds and array extents can determine if the loop will be free of exceptions or not. If the test passes, then the safe region is executed. Note that the array references in the safe region do not need any explicit checks. The array references in the unsafe region, executed if the test fails, still need all the runtime checks. One more comparison is used to disambiguate between the storage areas for arrays A and B. A successful disambiguation will cause execution of the alias-free version. Otherwise, the version with potential aliases must be executed. At first, there seems to be no difference between the alias-free version and the version with potential aliases. However, the compiler internally annotates the symbols in the alias-free region as not being aliased with each other. This information is later used to enable the various loop transformations.

```
for (i = 0; i < n; i++) {
  /* code for A[i] = F(B[i + 1]) with explicit checks */
  chknull(A)[chkbounds(i)] =
        F(chknull(B)[chkbounds(i + 1)])
}
```

(a) original code

```
if (  (A ≠ null) ∧ (B ≠ null)∧
      (n − 1 < A.length) ∧ (n < B.length)) {
  /* This region is free of exceptions */
  if (A ≠ B) {
    /* This region is free of aliases */
    for (i = 0; i < n; i++) {  A[i] = F(B[i + 1]) }
  } else {
    /* This region may have aliases */
    for (i = 0; i < n; i++) {  A[i] = F(B[i + 1]) }
  }
} else {
  /* This region may have exceptions and aliases */
  for (i = 0; i < n; i++) {
    chknull(A)[chkbounds(i)] =
          F(chknull(B)[chkbounds(i + 1)])
  }
}
```

(b) code after safe and alias-free region creation

Figure 2: Creation of safe and alias-free regions.

Libraries for numerical computing: Optimized libraries are an important vehicle for achieving high-performance in numerical applications. In particular, libraries provide the means for delivering parallelism transparently to the application programmer.

There are two main trends in the development of high-performance numerical libraries for Java. In one approach, existing native libraries are made available to Java programmers through the *Java Native Interface* (JNI) [3]. In the other approach, new libraries are developed entirely in Java [2]. Both approaches have their merits, with the right choice depending on the specific goals and constraints of an application. For further information on a particular library for numerical computing in Java, see the sidebar *Numerical linear algebra in Java*.

4 Implementation and Results

We have implemented our ideas in the NINJA prototype Java system based on the IBM *xl* family of compilers. Figure 3 shows the high-level organization of these compilers. The front-ends for different languages transform programs to a common intermediate representation called W-Code. The *Toronto Portable Optimizer* (TPO) is a W-Code to W-Code transformer which performs classical optimizations, like constant propagation and dead code elimination, and also high level loop transformations based on aggressive dataflow analysis. Finally, the transformed W-Code is converted into optimized machine code by an architecture-specific back-end. Semantic expansion of the Array package methods [1] is implemented within the IBM *High Performance Compiler for Java* [11] (HPCJ). Safe region creation and alias versioning have been implemented in the TPO.

We note that the use of a static compiler – HPCJ – represents a particular implementation choice. In principle, nothing prevents the techniques described in this article from being used in a dynamic compiler. Moreover, by using the quasi-static dynamic compilation model [10], the more expensive optimization and analysis techniques employed by TPO can be done off-line, sharply reducing the impact of compilation overhead.

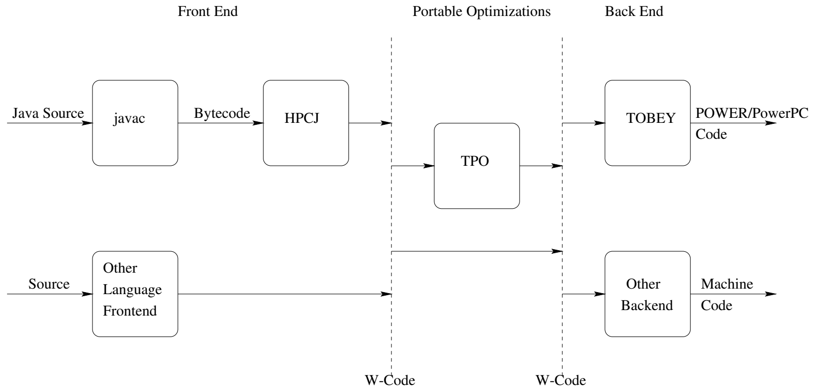


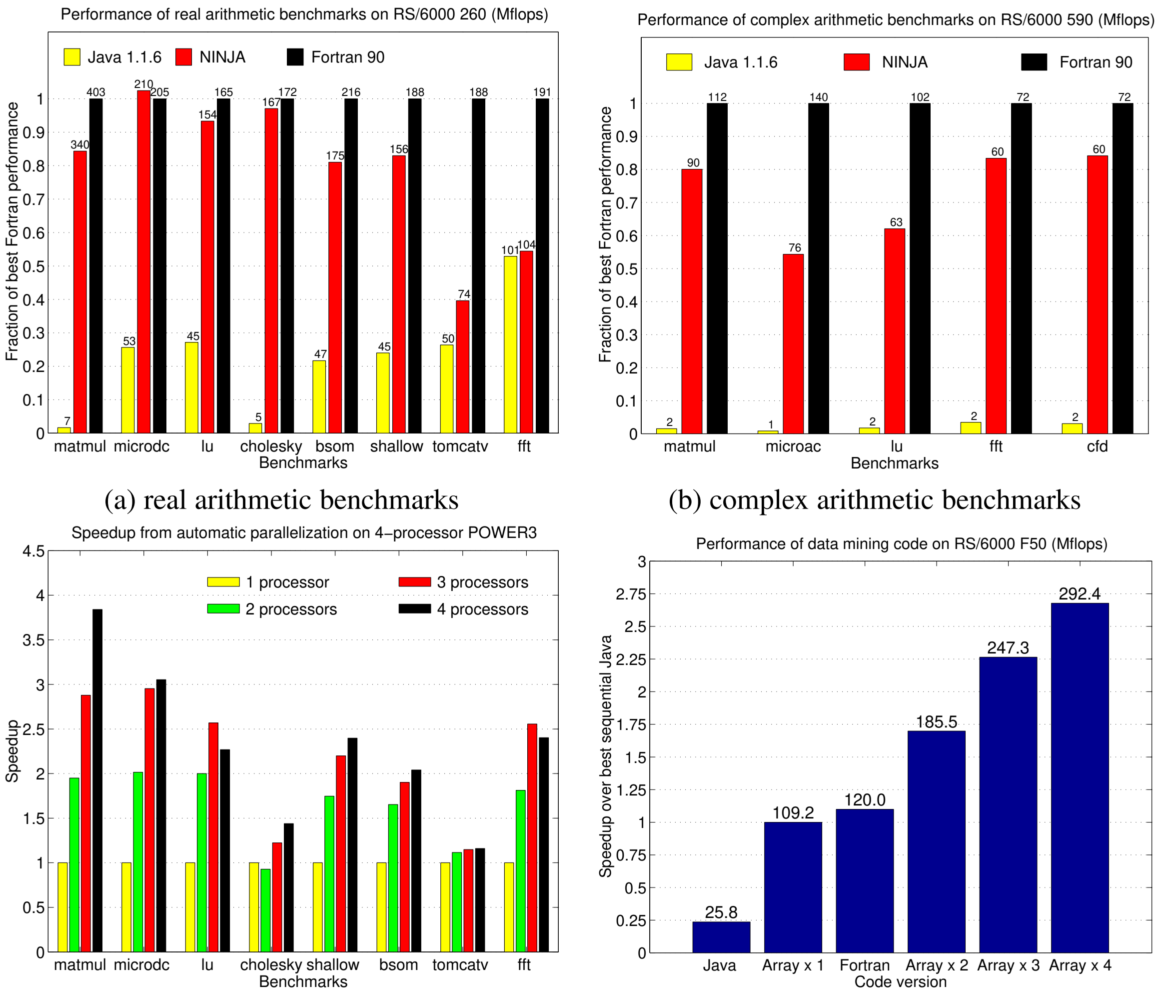
Figure 3: Architecture of the IBM *xl* compilers.

We used a suite of eight real and five complex arithmetic benchmarks to evaluate the performance impact of our techniques. We also applied our techniques to a production data mining application. These benchmarks and the data mining application are described further in [1] and [8]. The effectiveness of our techniques was assessed by comparing the performance produced by the NINJA compiler with that of the IBM Development Kit for Java version 1.1.6 and the IBM *xlf* Fortran compiler on a variety of platforms.

Sequential execution results: Results for the eight real arithmetic benchmarks, when running in strictly sequential (single-threaded) mode, are summarized in Figure 4(a). The height of each bar is proportional to the best Fortran performance achieved in the corresponding benchmark. The numbers at the top of the bars indicate actual Mflops. For the Java 1.1.6 version, arrays are implemented as `double[][]`. The NINJA version uses `doubleArray2D` Arrays from the Array package and semantic expansion. For six of the benchmarks (matmul, microdc, lu, cholesky, bsom, and shallow) the performance of the Java version (with the Array package and our compiler) is 80% or more of the performance of the Fortran version. This high performance is due to well-known loop transformations, enabled by our techniques, which enhance data locality. The Java version of tomcatv performs poorly because one of the outer loops in the program is not covered by a safe region. Therefore, no further loop transformations can be applied to this particular loop. The performance of fft is significantly lower than its Fortran counterpart because our Java implementation does not use interprocedural analysis.

Results for complex arithmetic benchmarks: Results for the five complex arithmetic benchmarks (fft, matmul, lu, cfd, and microac) are summarized in Figure 4(b). Again, the height of each bar is proportional to the best Fortran performance achieved in the corresponding benchmark, and the numbers at the top of the bars indicate actual Mflops. For the Java 1.1.6 version, complex arrays are represented using a `Complex[][]` array of `Complex` objects. No semantic expansion was applied. The NINJA version uses `ComplexArray2D` Arrays from the Array package and semantic expansion. In all cases we observe significant performance improvements between the Java 1.1.6 and NINJA versions. Improvements range from a factor of 35 (1.7 to 60.5 Mflops for cfd) to a factor of 75 (1.2 to 89.5 Mflops for matmul). We achieve Java performance that ranges from 55% (microac) to 85% (fft and cfd) of fully optimized Fortran code.

Parallel execution results: Loop parallelization is another important transformation enabled by safe region creation and alias versioning. We report speedup results from applying loop parallelization to our



(a) real arithmetic benchmarks

(b) complex arithmetic benchmarks

(c) speedup from automatic parallelization

(d) applying the Array package to data mining

Figure 4: Performance results of applying our Java optimization techniques to various cases.

eight real arithmetic Java benchmarks. All experiments were conducted using the Array package version of the benchmarks, compiled with our prototype compiler with automatic parallelization enabled. Speedup results, relative to the single processor performance of the parallel code optimized with NINJA, are shown in Figure 4(c). The compiler was able to parallelize some loops in each of the eight benchmarks. Significant speedups were obtained (better than 50% efficiency on 4 processors) in six of those benchmarks (matmul, microdc, lu, shallow, bsom, and fft).

Results for parallel libraries: We further demonstrate the effectiveness of our solutions by applying NINJA to a production data mining code. In this case, we use a parallel version of the Array package which uses multithreading to exploit parallelism within the Array operations. We note that the user application is a strictly sequential code, and that all parallelism is exploited transparently to the application programmer. Results are shown in Figure 4(d). The conventional (Java arrays) version of the application achieves only 26 Mflops, compared to 120 Mflops for the Fortran version. The single-processor Java version with the Array package (bar Array x 1) achieves 109 Mflops. Furthermore, when run on a multiprocessor, the performance of the Array package version scales with the number of processors (bars Array x 2, Array x

3, and Array x 4 for execution on 2, 3, and 4 processors, respectively), achieving almost 300 Mflops on 4 processors.

5 Conclusions

Our results show that there are no serious technical impediments to the adoption of Java as a major language for numerically intensive computing. The techniques we have presented are simple to implement and allow existing compiler optimizers to be exploited. Moreover, Java has many features like simpler pointers and flexibility in choosing object layouts, which facilitate the optimization techniques we have developed. The impediments instead are economic and social – an unwillingness on the part of vendors of Java compilers to commit the resources to develop product-quality compilers for technical computing; the reluctance of application developers to make the transition to new languages for developing new codes; and finally, the widespread belief that Java is simply not suited for technical computing. The consequences of this situation are severe: a large pool of programmers is being underutilized, and millions of lines of code are being developed using programming languages that are inherently more difficult and less safe to use than Java. The maintenance of these programs will be a burden on scientists and application developers for decades. It is our hope that the concepts and results presented in this paper will help overcome these impediments, and accelerate the acceptance of Java to the benefit of the technical computing community.

References

[1] P. V. Artigas, M. Gupta, S. P. Midkiff, and J. E. Moreira. High performance numerical computing in Java: Language and compiler issues. In J. Ferrante et al., editors, *12th International Workshop on Languages and Compilers for Parallel Computing*, volume 1863 of *Lecture Notes in Computer Science*, pages 1–17. Springer Verlag, August 1999. IBM Research Report RC21482.

[2] 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`.

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

[4] S. Chatterjee, V. V. Jain, A. R. Lebeck, S. Mundhra, and M. Thottethodi. Nonlinear array layouts for hierarchical memory systems. In *Proceedings of the 1999 International Conference on Supercomputing*, pages 444–453, Rhodes, Greece, 1999.

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

[6] J. Gosling, B. Joy, and G. Steele. *The Java(TM) Language Specification*. Addison-Wesley, 1996.

[7] 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.

[8] 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.

[9] 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.

[10] M. J. Serrano, R. Bordawekar, S. P. Midkiff, and M. Gupta. Quicksilver: a quasi-static compiler for Java. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'00)*, pages 66 – 82, Minneapolis, MN, USA, Oct. 2000.

[11] V. Seshadri. IBM high performance compiler for Java. AIXpert Magazine, September 1997. URL: `http://www.developer.ibm.com/library/aixpert`.

[12] M. J. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 2000.

A Sidebar: The Array package for Java

The Array package for Java (provisionally named `com.ibm.math.array`) provides the functionality and performance associated with true multidimensional arrays. The difference between arrays of arrays, directly supported by the Java Programming Language and Java Virtual Machine, and true multidimensional arrays is illustrated in Figure 1. Multidimensional arrays (Arrays) are rectangular collections of elements characterized by three immutable properties: *type*, *rank*, and *shape*. The type of an Array is the type of its elements (*e.g.*, `int`, `double`, or `Complex`). The rank (or dimensionality) of an Array is its number of axes. For example, the Arrays in Figure 1 are two-dimensional. The shape of an Array is determined by the extent of its axes. The dense and rectangular shape of Arrays facilitate the application of automatic compiler optimizations.

Figure 5 illustrates the class hierarchy for the Array package. The root of the hierarchy is an `Array` abstract class (not to be confused with the Array *package*). From the `Array` class we derive type-specific abstract classes. The leaves of the hierarchy correspond to final concrete classes, each implementing an Array 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.

Arrays can be manipulated element-wise or as aggregates. For instance, if one wants to compute a two-dimensional Array C of shape $m \times n$ in which each element is the sum of the corresponding elements of Arrays A and B, also of shape $m \times n$, then one can write either

```
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));
```

or

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

There are subtle differences between the two forms. The latter (aggregate) form has *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 A and/or B, the resulting values of elements of C may differ from the aggregate form.

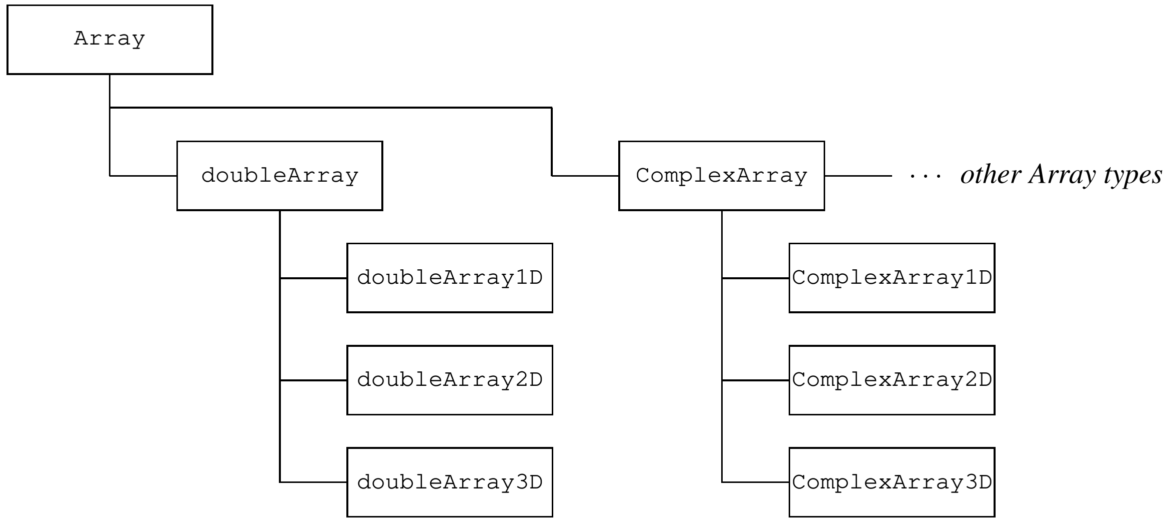


Figure 5: Simplified partial class hierarchy chart for the Array package.

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.

The code snipets above also show that syntactic support for the multidimensional arrays in the Array package would increase their usability. For example, it would be clearer to write

```
C[i,j] = A[i,j] + B[i,j];
```

for the body of the loop and

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

for the aggregate form. These issues are orthogonal to the usefulness of the library for enabling compiler optimizations, but will increase programmers acceptance of the package.

The Array package for Java is currently going through a standardization process through the Java Community Process (JSR 083 - `http://java.sun.com/aboutJava/communityprocess/jsr/jsr_083_multiarray.html`). The standardization is an important step in making Java practical for numerical computing. We note that the current naming conventions for the Array package do not follow recommended Java practice (*e.g.*, some classes start with lower case letters). We expect this will change with the standardization process.

B Sidebar: Numerical linear algebra in Java

Numerical linear algebra operations are important building blocks for scientific and engineering applications. Many problems in those domains can be expressed as a system of linear equations. Much work has been done, by industry, academia, and government, to develop libraries of routines that manipulate and solve these diverse systems of equations using numerical linear algebra. The Basic Linear Algebra Subprograms (BLAS) and the Linear Algebra Package (LAPACK) are two popular examples of such libraries available to Fortran and C programmers [5]. Part of our work in optimizing Java performance for numerically intensive computing involved the development of a linear algebra library for Java. This library is part of the Array package for Java. We call it *Java BLAS*.

We chose to develop this library entirely in Java, with no native code components. We took advantage of Java's object oriented features to arrive at a design that is easy to maintain, portable, and achieves high performance. In fact, the implementation of our linear algebra library in Java allowed us to pursue new optimization techniques.

Linear algebra algorithms (*e.g.*, solving for vector x in the equation $Ax = b$) are expressed in terms of vector and matrix operations. For that reason, we defined two interfaces, `BlasVector` and `BlasMatrix` that define the behavior of vectors and matrices, respectively. For example, any implementation of the `BlasMatrix` interface must provide methods `gemm` (for matrix multiplication), `trsm` (for solution of triangular systems), and `syrk` (for update of symmetric matrices). Linear algebra algorithms are then expressed strictly in terms of the methods defined by the `BlasVector` and `BlasMatrix` interfaces. This approach is particularly appropriate for the implementation of linear algebra algorithms in recursive form [7].

The one- and two-dimensional floating-point Arrays in the Array package (namely `floatArray1D`, `floatArray2D`, `doubleArray1D`, `doubleArray2D`, `ComplexArray1D`, `ComplexArray2D`) implement the `BlasVector` and `BlasMatrix` interfaces, respectively. Therefore, a single instance of a linear algebra algorithm works for single precision, double precision, and complex floating-point numbers. This results in our linear algebra library being much smaller than equivalent implementations in C and Fortran. We have been able to achieve very respectable performance with our all-Java implementation. Figure 6 compares the performance of our Java BLAS library and the highly tuned ESSL product when performing the `SGEMM` BLAS operation (*i.e.*, computing $C = \beta C + \alpha A \times B$ for single precision floating-point matrices A, B, and C). We observe that the Java BLAS version achieves 80% of ESSL performance and 75% of the machine peak performance (800 Mflops).

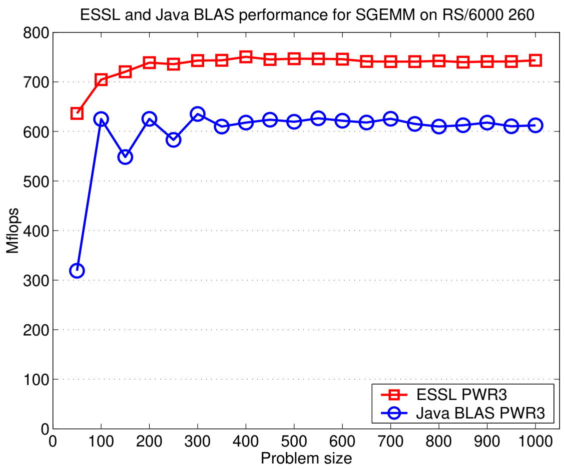


Figure 6: Performance results for ESSL and Java BLAS for SGEMM operation.

The area where Java allowed us to pursue new optimization techniques is in the exploitation of memory hierarchies, the multilevel cache structure of most current machines. It has been known for a while that neither the column major format of Fortran nor the row major format of C for storing multidimensional arrays is optimal for linear algebra algorithms. Java in general, and the Array package in particular, hide the specific memory layout of an array. Therefore, we are free to organize arrays in any form that we find convenient, totally transparent to the application programmer. In particular, we have experimented with a *block recursive* storage format [4]. The idea behind block recursive formats is illustrated in Figure 7. We start by dividing the array into two blocks and laying each block contiguous in memory. We repeat the partitioning for each block until we arrive at some convenient block size (*e.g.*, that fits into level 1 data cache). Our experiments with a block recursive storage format have shown performance improvements of approximately 10% above and beyond what is achieved by already highly optimized code.

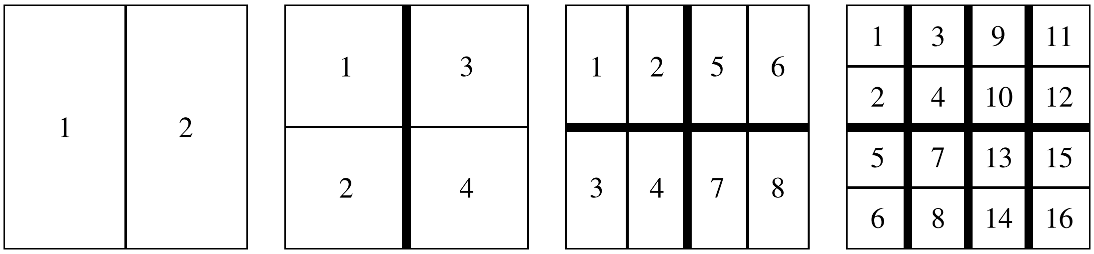


Figure 7: Illustration of the recursive block format.