Scientific Programming 10 (2002) 19–33
IOS Press

NINJA: Java for high performance numerical computing

José E. Moreira[a], Samuel P. Midkiff[a], Manish Gupta[a], Peng Wu[a], George Almasi[a] and Pedro Artigas[b]
[a]*IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598-0218, USA*
Tel.: +1 914 945 3018; Fax: +1 914 945 4270;
E-mail: {jmoreira,smidkiff,mgupta,pengwu,gheorghe}@us.ibm.com
[b]*School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213-3891, USA*
E-mail: artigas@cs.cmu.edu

Abstract: When Java was first introduced, there was a perception that its many benefits came at a significant performance cost. In the particularly performance-sensitive field of numerical computing, initial measurements indicated a hundred-fold performance disadvantage between Java and more established languages such as Fortran and C. Although much progress has been made, and Java now can be competitive with C/C++ in many important situations, significant performance challenges remain. Existing Java virtual machines are not yet capable of performing the advanced loop transformations and automatic parallelization that are now common in state-of-the-art Fortran compilers. Java also has difficulties in implementing complex arithmetic efficiently. These performance deficiencies can be attacked with a combination of class libraries (*packages*, in Java) that implement truly multidimensional arrays and complex numbers, and new compiler techniques that exploit the properties of these class libraries to enable other, more conventional, optimizations. Two compiler techniques, *versioning* and *semantic expansion*, can be leveraged to allow fully automatic optimization and parallelization of Java code. Our measurements with the NINJA prototype Java environment show that Java can be competitive in performance with highly optimized and tuned Fortran code.

1. Introduction

When Java(TM) was first introduced, there was a perception (properly founded at the time) that its many benefits, including portability, safety and ease of development, 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. The JavaGrande forum [11] was organized to facilitate cooperation and the dissemination of information among those researchers and applications writers wanting to improve the usefulness of Java on these environments.

Much has changed since those early days. More attention to optimization techniques in the just-in-time (JIT) compilers of modern virtual machines has resulted in performance that can be competitive with popular C/C++ compilers [4]. Figure 1(a), with data from a study described in [4], shows the performance of a particular hardware platform (a 333 MHz Sun Sparc-10) for different versions of the Java Virtual Machine (JVM). The results reported are the aggregate performance for the SciMark [16] benchmark. We note that performance has improved from 2 Mflops (with JVM version 1.1.6) to better than 30 Mflops (with JVM version 1.3). However, as Fig. 1(b) with data from the same study shows, the performance of Java is highly dependent on the platform. Often, the better hardware platform does not have a virtual machine implementing the more advanced optimizations.

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-

ISSN 1058-9244/02/$8.00

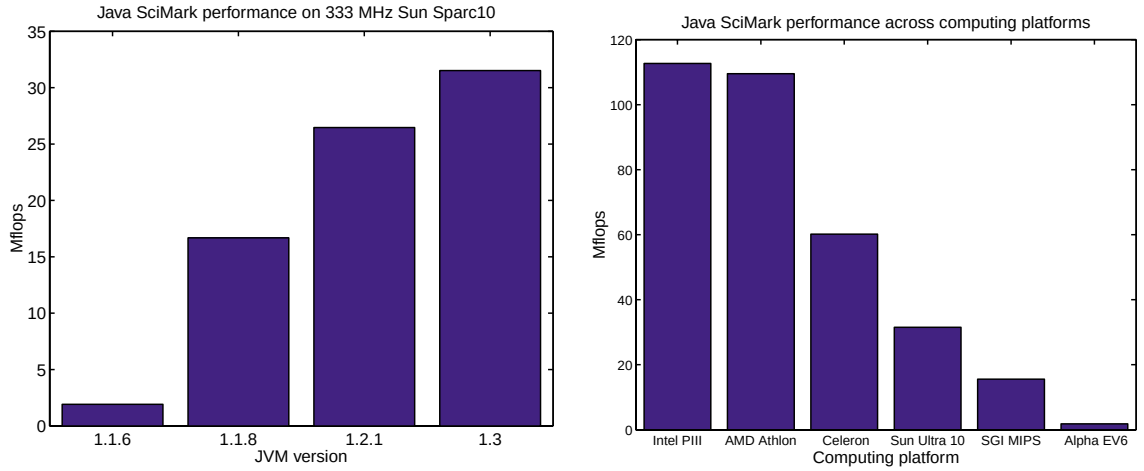


Fig. 1. Although Java performance on numerical computing has improved significantly in the past few years (a), that performance is inconsistent across platforms (b) and still not up to par with state-of-the-art C and Fortran compilers. (Data courtesy of Ron Boisvert and Roldan Pozo, of the National Institute of Standards and Technology.)

art Fortran and C compilers. Programs using complex arithmetic exhibit particularly bad performance [21]. Furthermore, current Java platforms are incapable of automatically applying important optimizations for numerical code, such as loop transformations and automatic parallelization [20]. 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 that 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 [17], lowering the economic barriers to Java's acceptance.

The primary goal of this paper is to convince virtual machine and application developers alike that Java can deliver both on the software engineering and performance fronts. The technology is available to make Java perform as well for numerical computing as highly tuned Fortran or C code. 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 rest of this paper is organized as follows. Section 2 describes the main sources of difficulties in optimizing Java performance for numerical computing. Section 3 covers the solutions that we have developed to overcome those difficulties. Section 4 discusses how those solutions were implemented in our prototype Java environment and provides various results that validate our approach to deliver high performance in numerical computing with Java. Finally, Section 5 presents our conclusions. Two appendices provide further detail on technologies of importance to numerical computing in Java: Appendix A gives the flavor of a multidimensional array package and Appendix B discusses a library for numerical linear algebra.

A note about the examples in this paper. The Java compilation model involves a Java source code to Java bytecode translation step, with the resulting bytecode typically compiled into native, or machine code using a dynamic (i.e. *just-in-time*) compiler. The NINJA compiler performs its optimizations during this bytecode to machine code compilation step, but we present our examples using source code for readability.

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.

2.1. 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 [8]. 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.

2.2. 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 2(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 Fig. 2(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 Fig. 2(c). Knowing the shape of a multidimensional array is necessary to enable some key optimizations that we discuss below. As can be seen in Fig. 2(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 Fig. 2(c), only requires looking at a small number of parameters.

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 Fig. 2. For the arrays of arrays shown in Fig. 2(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 Fig. 2(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 [20].

2.3. Complex numbers in Java

From a numerical perspective, Java only has direct support for real numbers. Fortran has direct support for complex numbers also. For even more versatility, both Fortran and C++ provide the means for efficiently supporting other arithmetic systems. Efficient support for complex numbers and other arithmetic systems 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.

We note that an array of n complex numbers requires the creation of n objects of type `Complex`, further complicating alias analysis and putting more pres-

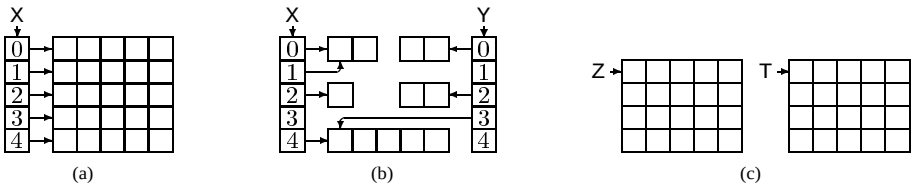


Fig. 2. Examples of (a) array of arrays simulating a two-dimensional array, (b) array of arrays in a more irregular structure, and (c) rectangular two-dimensional array.

sure on the memory allocation and garbage collection system. 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 to hold the result of 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. In that case, even modern virtual machines may perform a hundred times slower than equivalent Fortran code.

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 performance results on a particular virtual machine, however, depends on the extent to which that virtual machine (more precisely, its Java bytecode to machine code compiler) implements the automatic optimizations we describe below.

3.1. 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 ranks (*e.g.*, `doubleArray2D`, `ComplexArray3D`, `ObjectArray1D`). This Array package introduces true multidimensional arrays in Java through a class library. See Appendix A, *The Array package for Java*, for further discussion.

Element accessor methods (**get** and **set** methods for individual array elements), sectioning operations, gather and scatter operations, and basic linear algebra subroutines (BLAS) are some of the operations defined for the Array data types. By construction, 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. This allows programs using the Array package to have high performance when executed on JVM that recognize the Array package methods.

As an example, consider the operation of computing $C_{ij} = A_{ij} + B_{ji}$ for all elements of $n \times n$ Arrays A, B, and C. The code for that operation would look something like:

```
doubleArray2D A, B, C;
...
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++) {
    C.set(i, j, A.get(i, j) + B.get(j, i));
  }
}
```

which requires three method calls (two **get**s and one **set**) in every loop iteration. If the compiler knows that A, B, and C are multidimensional arrays, it can generate code that directly accesses the elements of the Arrays, much like a Fortran compiler generates code for the source fragment

do $i = 1, n$
  do $j = 1, n$
    $C(i, j) = A(i, j) + B(j, i)$
  end do
end do

Note that this is different from the important, but more conventional, optimization of inlining. The compiler does not replace the invocation of **get** and **set** by their library code. Instead, the compiler *knows* about them: it knows the semantics of the classes and of the methods. Semantic expansion is an escape mechanism for efficiently extending a programming language through standard class libraries.

3.2. 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. (See Fig. 3.) 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 full Java object representation).

Figure 3 illustrates the differences between value-objects and regular objects. A value-object version of `Complex` contains only fields for the real and imaginary parts of the complex number represented, as shown in Fig. 3(b). It is akin to a C `struct`, and can be easily allocated on the stack and even on registers. For `Complex` to behave as a true Java object, a different representation is necessary, shown in Fig. 3(c). In particular, every Java object requires an object header, which can represent a significant fraction of the object size. (For example, a `Complex` object of double-precision real and imaginary parts occupies 32 bytes in modern virtual machines, even though only 16 bytes are dedicated to the numerical fields.) Even worse is the overhead of creating and destroying objects, which typically are allocated on the heap.

Any computation involving the arithmetic methods can be semantically expanded to use complex values. Conversion to `Complex` objects is 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.

We illustrate those concepts with an example. Consider the computation of $y_i = ax_i$ for all n elements of arrays x and y of complex numbers. This operation would typically be coded as

`ComplexArray1D` x, y;
`Complex` a;
...
for $(i = 0; i < n; i$++$)$ {
  y.**set**$(i, a$.**times**$(x$.**get**$(i)))$;
}

A straightforward execution of this code would require the creation of $2n$ temporary objects. For every iteration, an object has to be created to represent x_i. A second object is created to hold the result of ax_i. The cost of creating and destroying these objects completely dominates execution. If the compiler knows the semantics of `Complex` and `ComplexArray`s, it can replace the method calls by code that simply manipulates values. Only the values of the real and imaginary parts of x_i are generated by x.**get**(i). Only the values of the real and imaginary parts of ax_i are computed by a.**times**$(x$.**get**$(i))$. Finally, those values are used to update y_i. As a result, the object code generated would not be significantly different from that produced by a Fortran compiler for the source fragment

`complex*16` $x(n), y(n)$
`complex*16` a
...
do $i = 1, n$
  $y(i) = a * x(i)$
end do

3.3. 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 [20]. 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

```
public final class Complex {

    private double re, im;

    public Complex(double r, double i) {
        re = r; im = i;
    }

    public Complex plus(Complex z) {
        return new Complex(re+z.re,im+z.im);
    }

    public Complex minus(Complex z) {
        return new Complex(re-z.re,im-z.im);
    }

    public Complex times(Complex z) {
        return new Complex(re*z.re-im*z.im,im*z.re+re*z.im);
    }
}
```

(a) partial code for `Complex` class

`re`	`im`
0.0	0.0

(b) `Complex` value-object representation

descriptor	`re`	`im`
	0.0	0.0

(c) `Complex` object representation

Fig. 3. A Java class for complex numbers.

inside the loop will generate in-bound accesses. If the tests pass, 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 kind 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.

We note that the "no aliasing" property between two Arrays is invariant to garbage collection activity. Garbage collection may remove aliasing, but it will never introduce it. Therefore, it is enough to verify once that two Arrays are not aliased to each other. We have to make sure, however, that there are no assignments to Array references (*e.g.*, $A = B$) in a safe and alias-free region, as that can introduce new aliasing. Assignments to the elements of an Array (*e.g.*, $A[i] = B[j]$) never introduce aliasing.

An example of the versioning transformation to create safe and alias-free regions is shown in Fig. 4. Figure 4(a) illustrates the original code for computing $A_i = \mathcal{F}(B_{i+1})$ for n-element arrays A and B. Figure 4(b) explicitly shows all `null` pointer and array bounds runtime checks that are performed when the code is executed by a Java virtual machine. The check **chknull**(A) verifies that Array reference A is not a `null`-pointer, whereas check **chkbounds**(i) verifies that the index i is valid for that corresponding Array. Figure 4(c) 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

```
for (i = 0; i < n; i++) {
  A[i] = F(B[i + 1])
}
```

(a) original code

```
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 with explicit runtime checks

```
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)])
  }
}
```

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

Fig. 4. Creation of safe and alias-free regions.

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. We denote these new, alias-free symbols, by A' and B'. This information is later used to enable the various loop transformations. We note that the representation shown in Fig. 4(c) only exists as a compiler internal intermediate representation, after the versioning is automatically performed and before object code is generated. Neither the Java language, nor the Java bytecode, can directly represent that information.

The concepts illustrated by the example of Fig. 4 can be extended to loop nests of arbitrary depth operating on multidimensional arrays. The tests for safety and aliasing are much simpler (and cheaper) if the arrays are known to be truly multidimensional (rectangular), as in Fig. 2(c). The Arrays from the Array package have this property.

3.4. 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) [5]. In the other approach, new libraries are developed entirely in Java [3]. Both approaches have their merits, with the right choice depending on the specific goals and constraints of an application.

Using existing native libraries through JNI is very appealing. First, it provides access to a large body of existing code. Second, that code has already been debugged and its performance tuned by previous pro-

grammers. Third, in many cases (*e.g.*, BLAS, MPI, LAPACK, ...) the same native library is available for a variety of platforms, properly tuned by the vendor of each platform.

However, using libraries that are themselves written in Java also has its advantages. First, those libraries are truly portable, and one does not have to worry about ideosyncrasies that typically occur in versions of a native library for different platforms, such as maintaining Java floating point semantics. Second, Java libraries typically fit better with Java applications. One does not have to worry about parameter translation and data representations that can cause performance problems and/or unexpected behavior. Third, and perhaps most importantly, by writing the libraries in Java the more advanced optimization and programming techniques that are being developed, and will be developed, for Java will be exploited in the future without the additional work of performing another port. The discussion of Appendix B describes one technique which is easier to implement with Java, that can lead to improved performance.

The Array package itself is a library for numerical computing. In addition to focusing on properties that enable compiler optimizations, we also designed the Array package so that most operations could be performed in parallel. We have implemented a version of the Array package which uses multiple Java threads to exploit multiprocessor parallelism inside some key methods. This is a convenient approach for the application developer. The application code itself can be kept sequential, and parallelism is exploited transparently inside the methods of the Array package. We report results with this approach in the next section. For further information on additional library support for numerical computing in Java, see Appendix B, *Numerical linear algebra in Java*.

3.5. A comment on our optimization approaches

We want to close this section by emphasizing that the class libraries and compiler optimizations that we presented are strictly Java compliant. They do not require any changes to the base language or the virtual machines, and they do not change existing semantics. The Array and complex classes are just tools for developing numerical applications in a style that is familiar to scientific and technical programmers. The compiler optimizations (versioning and semantic expansion) are exactly that: optimizations that can improve performance of code significantly (by orders of magnitude as we will see in the next section) without changing the observed behavior.

4. Implementation and results

We have implemented our ideas in the NINJA prototype Java environment, based on the IBM *XL* family of compilers. Figure 5 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. TPO can also perform both directive-assisted and automatic parallelization of loops and other constructs. Finally, the transformed W-Code is converted into optimized machine code by an architecture-specific back-end.

The particular compilation path for Java programs is illustrated in the top half of Fig. 5. Java source code is compiled by a conventional Java compiler (*e.g.*, *javac*) into bytecode for the Java Virtual Machine. We then use the IBM *High Performance Compiler for Java* [19] (HPCJ) to statically translate bytecode into W-Code. In other words, HPCJ plays the role of front-end for bytecode. Once W-Code for Java is generated, it follows the same path through TPO and back-ends as W-Code generated from other source languages. Semantic expansion of the Array package methods [2] is implemented within HPCJ, as it is Java specific. Safe region creation and alias versioning have been implemented in TPO and those techniques can be applied to W-Code from any other language.

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 [18], the more expensive optimization and analysis techniques employed by TPO can be done off-line, sharply reducing the impact of compilation overhead. We should also mention that our particular implementation is based on IBM products for the RS/6000 family of machines and the AIX operating system. However, the organization of our implementation is representative of typical high-performance compilers [15] and it is adopted by other vendors. Obviously, a reimplementation effort is necessary for each different platform, but the approach we followed serves as a template for delivering high-performance solutions for Java.

We used a suite of eight real and five complex arithmetic benchmarks to evaluate the performance impact

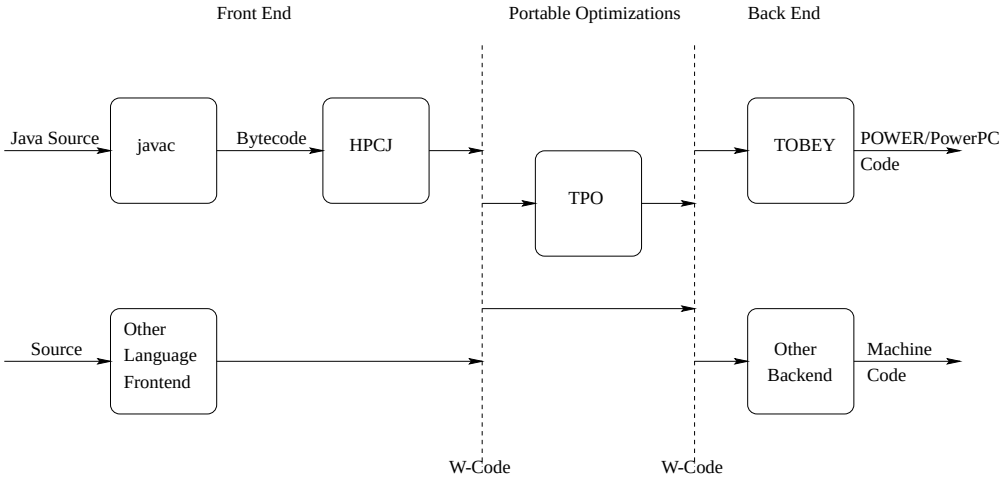


Fig. 5. Architecture of the IBM *XL* compilers.

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 [2,13,14]. 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.

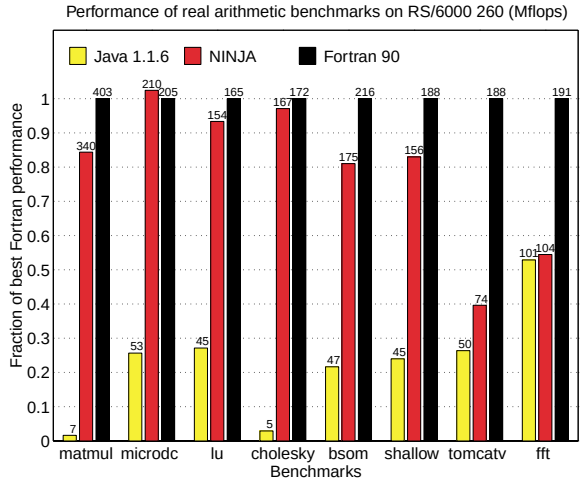
4.1. Sequential execution results

The eight real arithmetic benchmarks are matmul (matrix multiply), microdc (electrostatic potential computation), lu (LU factorization), cholesky (Cholesky factorization), shallow (shallow water simulation), bsom (neural network training), tomcatv (mesh generation and solver), and fft (FFT with explicit real arithmetic). Results for these benchmarks, when running in strictly sequential (single-threaded) mode, are summarized in Fig. 6(a). Measurements were made on an RS/6000 model 260 machine, with a 200 MHz POWER3 processor. 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.

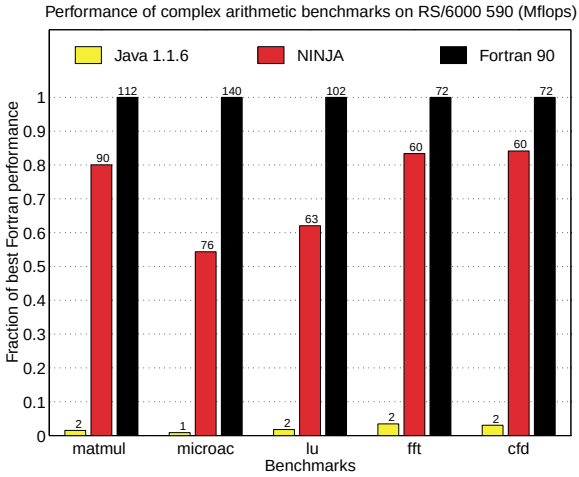
For six of the benchmarks (matmul, microdc, lu, cholesky, bsom, and shallow) the performance of the Java version (with the Array package and our compiler) is 80% or more of the performance of the Fortran version. This high performance is due to well-known loop transformations, enabled by our techniques, 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, which has a big impact in the optimization of the Fortran code.

4.2. Results for complex arithmetic benchmarks

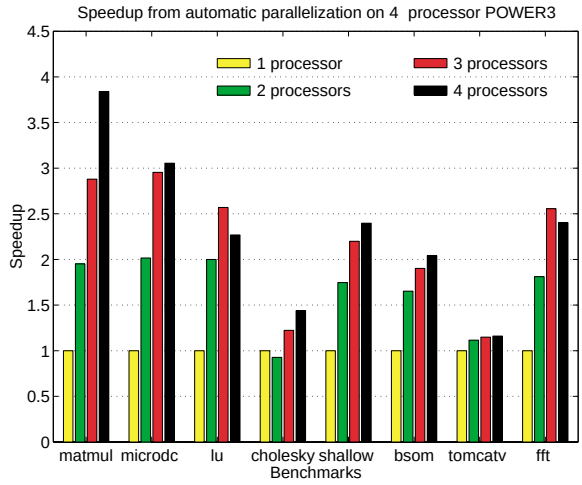
The five complex benchmarks are matmul (matrix multiply), microac (electrodynamic potential computation), lu (LU factorization), fft (FFT with complex arithmetic), and cfd (two-dimensional convolution). Results for these benchmarks are summarized in Fig. 6(b). Measurements were made on an RS/6000 model 590 machine, with a 67 MHz POWER2 processor. 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.



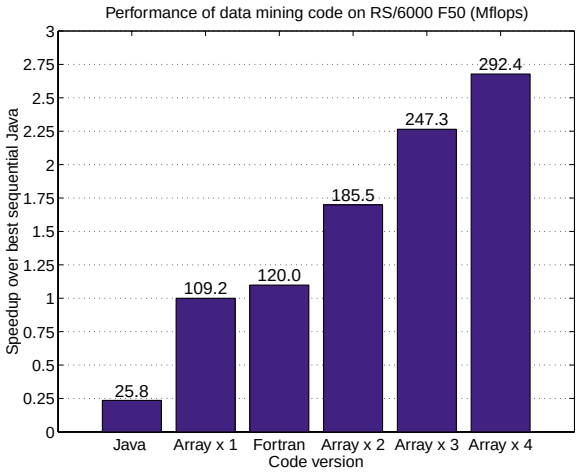
(a) real arithmetic benchmarks



(b) complex arithmetic benchmarks



(c) speedup from automatic parallelization



(d) applying the Array package to data mining

Fig. 6. Performance results of applying our Java optimization techniques to various cases.

4.3. 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 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 Fig. 6(c). Measurements were made in a machine with four 200 MHz POWER3 processors. 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).

4.4. Results for parallel libraries

We further demonstrate the effectiveness of our solutions by applying NINJA to a production data mining code [14]. 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 Fig. 6(d). Measurements were made with a RS/6000 model F50

machine, with four 332 MHz PowerPC 604e processors. 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. The Java-specific optimizations are relatively simple and most of the benefits accrue from leveraging well understood language-independent optimizations that are already implemented in current compilers. Moreover, Java has many features like simpler pointers and flexibility in choosing object layouts, which facilitate application of the optimization techniques we have developed.

The impediments to high-performance computing in Java are instead 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.

We have already engaged with companies that are interested in doing numerical computing in Java, which represents a first step towards wider adoption of Java in that field. Java already has a strong user base in commercial computing. For example, IBM's Websphere suite is centered around Java and is widely used in the industry. However, the characteristics of the commercial computing market are significantly different, in both size and requirements, from the technical computing market. It is our hope that the concepts and results presented in this paper will help overcome the difficulties of establishing Java as a viable platform for numerical computing and accelerate the acceptance of Java, positively impacting the technical computing community in the same way that Java has impacted the commercial computing community.

Appendix A. 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 Fig. 2. 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 Fig. 2 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 7 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

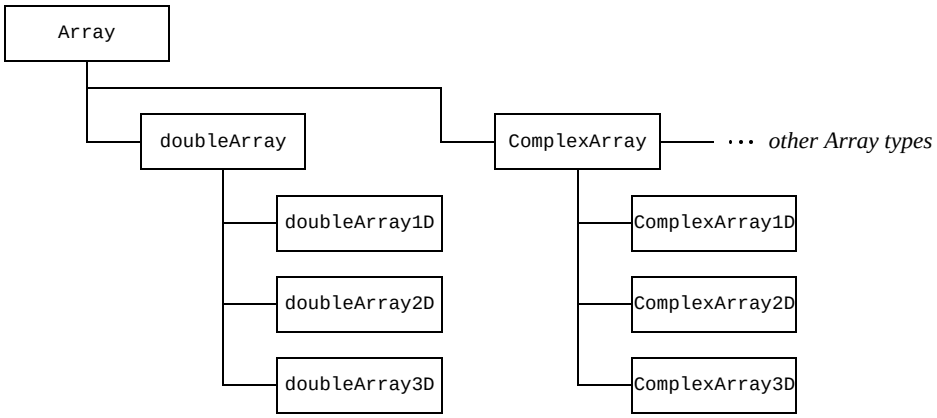


Fig. 7. Simplified partial class hierarchy chart for the Array package.

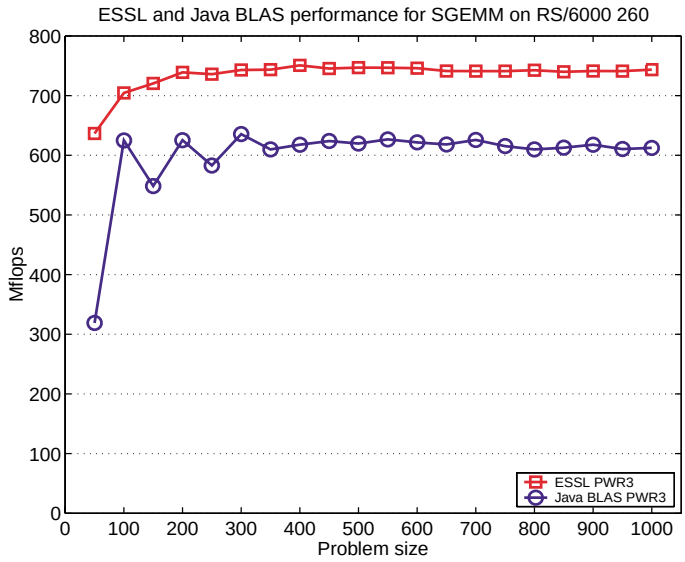


Fig. 8. Performance results for ESSL and Java BLAS for `SGEMM` operation.

```
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. 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 snippets 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 programmer acceptance of the package.

The Array package for Java is currently going through a standardization process through the Java

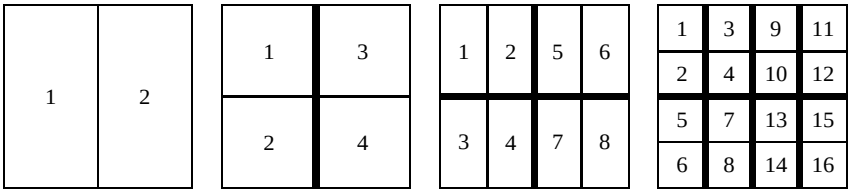


Fig. 9. Illustration of the block recursive layout.

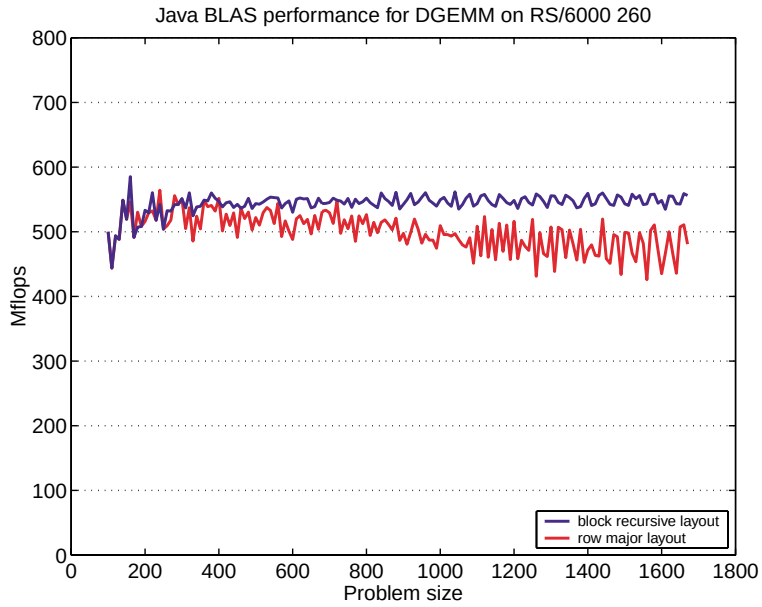


Fig. 10. Performance results for Java `DGEMM` with two array layouts.

Community Process [12]. 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. It is also likely that the class hierarchy of the standardized package will be somewhat different. Nevertheless, the key properties of truly rectangular multidimensional arrays, important for enabling compiler optimizations, will be preserved.

Appendix B. 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 [7]. 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 [1]. The implementation of our linear algebra library in Java also 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 [9].

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 8 compares the performance of our Java BLAS library and the highly tuned ESSL product [10] 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). In those measurements, all three matrices are of size $n \times n$, where n is the problem size. We observe that the Java BLAS version achieves 80% of ESSL performance and 75% of the machine peak performance (800 Mflops).

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 layout of Fortran nor the row major layout 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 layout [6]. The idea behind block recursive layouts is illustrated in Fig. 9. 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 layout have shown significant performance improvements above and beyond what is achieved by already highly optimized code. The performance impact of the recursive blocked layout can be observed in Fig. 10. The bottom (lighter) plot in that figure shows the performance of the BLAS `DGEMM` operation (*i.e.*, the double-precision version of `SGEMM`), as a function of problem size, for an optimized code operating on an array with row major layout. The top (darker) plot shows the performance for the same code operating on an array with block recursive layout. For large problem sizes, the Mflops rate for the block recursive layout can be up to 30% higher. Furthermore, we observe that the performance of the block recursive layout to be more stable with the problem size.

References

[1] G. Almasi, F.G. Gustavson and J.E. Moreira, Design and Evaluation of a Linear Algebra Package for Java, in: *Proceedings of the ACM 2000 Conference on Java Grande*, ACM, June 3–4, 2000, pp. 150–159.

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

[3] 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) (September–November 1998), 1117–1129. ACM 1998 Workshop on Java for High-Performance Network Computing, URL: http://www.cs.ucsb.edu/conferences/java98.

[4] R.F. Boisvert, J.E. Moreira, M. Philippsen and R. Pozo , Java and Numerical Computing, *Computing in Science and Engineering* **3**(2) (March/April 2001), 18–24.

[5] H. Casanova, J. Dongarra and D.M. Doolin, Java Access to Numerical Libraries, *Concurrency, Pract. Exp.* (*UK*) **9**(11) (November 1997), 1279–1291. Java for Computational Science and Engineering – Simulation and Modeling II Las Vegas, NV, USA, 21 June 1997.

[6] 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*, Rhodes, Greece, 1999, pp. 444–453.

[7] 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.

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

[9] F.G. Gustavson, Recursion Leads to Automatic Variable Blocking For Dense Linear Algebra Algorithms, *IBM Journal of Research and Development* **41**(6) (November 1997), 737–755.

[10] International Business Machines Corporation, *IBM Parallel Engineering and Scientific Subroutine Library for AIX – Guide and Reference*, December 1997.

[11] Java Grande Charter, http://www.javagrande.org/public.htm.

[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, M. Gupta, P.V. Artigas, M. Snir and R.D. Lawrence, Java Programming for High Performance Numerical Computing, *IBM Systems Journal* **39**(1) (2000) 21–56, IBM Research Report RC21481.
[14] J.E. Moreira, S.P. Midkiff, M. Gupta and R.D. Lawrence, Parallel Data Mining in Java, in: *Proceedings of SC '99*, Also available as IBM Research Report 21326, Nov. 1999.
[15] S.S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufmann, San Francisco, California, 1997.
[16] R. Pozo and B. Miller, SciMark: A Numerical Benchmark for Java and C/C++, National Institute of Standards and Technology, Gaithersburg, MD, http://math.nist.gov/SciMark.
[17] V. Sarkar, Automatic selection of high-order transformations in the IBM XL Fortran compilers, *IBM Journal of Research and Development* **41**(3) (May 1997), 233–264.
[18] 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*), Minneapolis, MN, USA, Oct. 2000, pp. 66–82.
[19] V. Seshadri, IBM High Performance Compiler for Java, AIXpert Magazine, September 1997, URL: http://www.developer.ibm.com/library/aixpert.
[20] M.J. Wolfe, *High Performance Compilers for Parallel Computing*, Addison-Wesley, 2000.
[21] 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*, IBM Research Report RC21393, 1999, pp. 109–118.