

AUTOMATIC LOOP TRANSFORMATIONS AND PARALLELIZATION FOR JAVA

PEDRO V. ARTIGAS

*School of Computer Science, Carnegie Mellon University
Pittsburgh, Pennsylvania, 15213-3891, USA*

and

MANISH GUPTA

SAMUEL P. MIKIFF

JOSÉ E. MOREIRA

*IBM T.J. Watson Research Center, P.O. Box 218
Yorktown Heights, New York 10598-0218, USA*

Received (received date)

Revised (revised date)

Communicated by (Name of Editor)

ABSTRACT

This paper describes a prototype Java compiler that achieves performance levels approaching those of current state-of-the-art Fortran compilers on numerical codes. We present a new transformation called alias versioning that takes advantage of the simplicity of pointers in Java. This transformation, combined with other techniques that we have developed, enables the compiler to perform high order loop transformations and parallelization completely automatically. We believe that our compiler is the first to have such capabilities of optimizing numerical Java codes. By exploiting synergies between our compiler and the Array package for Java, we achieve between 80 and 100% of the performance of highly optimized Fortran code in a variety of benchmarks. Furthermore, automatic parallelization achieves speedups of up to 3.8 on four processors. Our compiler technology makes Java a serious contender for implementing numerical applications.

1. Introduction

JavaTM is well suited to the development of large complex applications. However, the poor performance of existing Java compilers (both static and dynamic) on numerical codes has prevented widespread adoption of Java as a language for the development of scientific and engineering applications. This paper describes a prototype Java compiler that can match the performance of well-tuned Fortran applications, and that is more effective in optimizing numerical codes than any other Java compiler we know of.

Current Fortran compilers are capable of automatically applying high order transformations [1] to the code. These transformations include loop fusion, unimodular transformations, loop tiling and loop parallelization [2]. Nothing in the Java language prevents the utilization of these same techniques, although Java does have

some unique features that are detrimental to performance when compiled naively. Among these features are: (i) exceptions checks for **null**-pointer and out-of-bounds array accesses, (ii) a precise exception model, and (iii) increased opportunities for aliasing among array rows and elements. In [3], we presented techniques that mitigate the impact of the precise exception model and mandatory run-time checks for **null**-pointer and array accesses. We also describe an Array package that adds true multidimensional arrays (similar to those of Fortran 90) to the Java language. The Array package addresses problems caused by Java's overly general array representation.

In this paper, we describe a transformation technique that creates alias-free regions on which aggressive optimizations can take place. Run-time tests determine if it is legal to execute this alias-free region, or if a region with more conservative aliasing properties must be executed. These alias-free regions, combined with the safe regions described in [4], enable high order transformations, including automatic parallelization, to be applied to Java programs.

The rest of the paper is organized as follows. We start with an overview of the Array package for Java. We proceed to briefly discuss the creation of safe regions, which is a precursor to the compiler optimizations discussed in this paper. We then describe the interplay between alias analysis and high order transformations, and motivate the technique for the creation of alias-free regions in the program. Next we describe our implementation of various optimization techniques, and provide experimental evidence to support our claim that Java can be competitive with Fortran for numerical codes. Finally, we discuss related work and present our conclusions.

2. The Array Package for Java

The Java Programming LanguageTM and the Java Virtual MachineTM directly support only one-dimensional arrays. Figure 1(a) shows how higher dimensional arrays are simulated in Java using *arrays of arrays*. This representation gives great flexibility in building data structures, but that flexibility reduces the ability of compilers to perform aggressive optimizations, and is rarely exploited by numerical programs. The difficulties for compilers fall mainly into three areas: (i) nonrectangular shape, (ii) shape volatility, and (iii) inter- and intra-array aliasing.

The nonrectangular shape results in difficulties with optimizations requiring information about array bounds [4]. Moreover, the shape of the array can change dynamically (e.g., by executing the statement `X[4] = new double[3]`). It is also possible for rows within an array to be aliased (e.g., `X[0]` and `X[1]`), for rows to be shared across arrays (e.g., `X[4]` and `Y[3]`), and for these aliasing relationships to change with time.

Fortran 90 supports true multidimensional arrays, as shown in Figure 1(b). Distinct arrays like `Z` and `T` do not share storage. Even if two arrays do share storage, as in two sections of the same base array, an aliasing test can be performed based on the base addresses and extents of both arrays. Also, Fortran 90 multidimensional arrays are rectangular: all rows have the same extent.

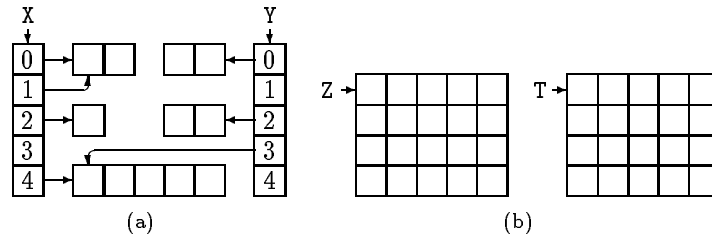


Fig. 1. Examples of (a) array of arrays, (b) rectangular two-dimensional array.

To bring these desirable properties of Fortran to Java, we have developed the Array package for Java [3]. This package is written entirely in Java, so that programs utilizing it are fully Java compliant. In this paper, we will refer to Java array objects as *arrays* and Array package array objects as *Arrays* (with a capital *A*). For clarity, accesses to elements of an Array will use the notation $A[i, j]$ instead of the more cumbersome method notation (i.e., $A.get(i, j)$ and $A.put(i, j)$) defined in the Array package. As an example, we shown in Figure 2 a kernel from the MICRODC benchmark [4].

```
// A, B are (w + 1, n + 1) two-dimensional Arrays
doubleArray2D A = new doubleArray2D(w + 1, h + 1);
doubleArray2D B = new doubleArray2D(w + 1, h + 1);

for (j = 1; j < n; j++)
  for (i = 1; i < w; i++)
    A[i, j] = 0.25 * (B[i + 1, j] + B[i - 1, j] + B[i, j + 1] + B[i, j - 1]);
```

Fig. 2. A kernel from MICRODC, used as an example in this paper.

3. Safe Region Creation

Java requirements for array bounds checks and **null**-pointer checks, coupled with its precise exception model, typically prevent all transformations that change the order of array references. Such transformations are legal only if it can be proven that the problematic exceptions cannot be thrown – that is, that a region of code is exception free, or *safe*. Safe regions can be formed by cloning an original code region (in our case, a loop nest) into two versions [4]. One version is optimistically formed without run-time checks for exceptions, and the other contains all mandated checks. A test is formed to determine, at run-time, which version to execute. If the test shows that no exception can possibly be thrown, the safe version is exe-

cuted, otherwise the unsafe version is executed. Figure 3 gives an example of this transformation for the MICRODC kernel of Figure 2, with `CHKn` indicating a check for `null`-pointer and `CHKb` indicating a check for out-of-bounds access. These checks are implicit at the Java and bytecode levels, but they become explicit at lower levels of representation.

<pre> for(i = 1; i < w; i++) for(j = 1; j < n; j++) CHKn(A)[CHKb(i),CHKb(j)] = 0.25*(CHKn(B)[CHKb(i + 1),CHKb(j)] + CHKn(B)[CHKb(i - 1),CHKb(j)] + CHKn(B)[CHKb(i),CHKb(j + 1)] + CHKn(B)[CHKb(i),CHKb(j - 1)]); </pre>	<pre> // X.size(y) = size of Array X along axis y if ((A != null) && (B != null) && ((w - 1) < A.size(0)) && ((n - 1) < A.size(1)) && (w < B.size(0)) && (n < B.size(1))) { // safe region for (i = 1; i < w; i++) for(j = 1; j < n; j++) A[i,j]=0.25*(B[i + 1,j]+B[i - 1,j]+ B[i,j + 1]+B[i,j - 1]); } else { // unsafe region, as in Figure 3(a) } </pre>
(a) original code	(b) code after safe region creation

Fig. 3. Creation of safe regions.

4. Aliasing and Transformations

Conservative aliasing information reduces the freedom a compiler has in moving references relative to one another, and thereby reduces the range of program transformations that can be applied. Consider the loop nest of Figure 2. The loop order shown results in poor data locality. An optimizing compiler will, whenever possible, interchange the two loops to improve data locality. If sufficient alias information is not available, the compiler must conclude that the elements of the *A* and *B* Arrays are aliased to each other, and that it cannot reorder the loops without changing the program semantics. If alias information is available that shows that elements of *A* and *B* are not aliased, loop interchange can be applied.

Impact of language design: In some programming languages like Fortran, the language definition requires subroutine parameters not to be aliased. Languages like C and C++ support unrestricted pointers, which makes alias analysis difficult and often quite conservative. Java supports relatively restrictive pointers compared to C/C++. In particular, Java does not allow a pointer to an arbitrary field of an object or an element of a one-dimensional array. Hence, given two references to objects (including one-dimensional arrays), they either refer to identical objects or to completely nonoverlapping objects. We exploit this key property, as discussed in the next section, to obtain a run-time test for alias disambiguation of one-dimensional Java arrays and multidimensional Arrays from the Array package.

Our aliasing representation: We use the following representation of alias relationships. Each symbol α in the symbol table is associated with a separate alias set $\mathcal{A}(\alpha)$, which contains all symbols that may be aliased with α (as inferred conservatively using compiler analysis). Alias sets are reflexive: for every symbol α , $\{\alpha\}$ is the minimal alias set. Alias sets are also symmetric: $\beta \in \mathcal{A}(\alpha)$ implies $\alpha \in \mathcal{A}(\beta)$. Alias sets are *not* transitive, allowing complex alias relationships, as in the (C or C++) code fragment of Figure 4. In this example, the symbol representing member C of union X is aliased with symbols A and B, but A is not aliased with B. This is expressible only because alias sets are not transitive. This property is essential for obtaining better aliasing information for Java through alias versioning, discussed next.

<pre> union X { long C; struct { short A; short B; }; }; </pre>	<pre> A → {A, C} B → {B, C} C → {A, B, C} </pre>
(a) A C/C++ code fragment	(b) Alias sets for A, B, and C

Fig. 4. An example of alias sets.

5. Alias Versioning for Java

Our compiler infrastructure employs context-sensitive interprocedural pointer analysis for Fortran 90 and C to develop more precise alias information. However, this interprocedural analysis is not available for Java because of difficulties associated with handling the dynamic features of the language and ensuring binary compatibility of the generated code. Hence, we use versioning to create a region free of aliases, and use an efficient run-time test (enabled by Java’s simple pointer semantics) to determine if the alias-free regions can be safely executed. Although we use this technique in lieu of interprocedural alias analysis, this approach is, in principle, complementary to alias analysis by the compiler. It can be used to deal with the cases where compiler analysis is too conservative.

5.1. Array Alias Disambiguation in Java

Because one-dimensional Java arrays cannot partially overlap, two arrays are either unrelated or are the same. The relationship can be determined by comparing the values of reference pointers. As seen in Figure 1, this is not true for Java multi-dimensional arrays of arrays [3], but is true for (Array package) Arrays, which are Java objects. Arrays can be disambiguated by comparing the data storage pointer in the Array object. The compiler can use this `data` field in a simple run-time

test to determining if two Arrays use the same data storage, as shown in Figure 5, which extends the versioning in Figure 3. Even though two Arrays share storage, they may not share elements. This can happen, for example, if two Arrays are both sections of the same base Array, one containing only the even elements and the other only the odd elements. The base pointer test shown in Figure 5 can be refined with dependence analysis techniques for analysis of Array sections [5].

```

// X.size(y) = size of Array X along axis y
if ( (A != null) && (B != null) &&
      ((w - 1) < A.size(0)) && ((n - 1) < A.size(1)) &&
      (w < B.size(0)) && (n < B.size(1))) {
  // safe region
  if (A.data != B.data) {
    // alias-free region
    for (i = 1; i < w; i++)
      for (j = 1; j < n; j++)
        A'[i, j] = 0.25 * (B'[i + 1, j] + B'[i - 1, j] + B'[i, j + 1] + B'[i, j - 1]);
  } else {
    // region potentially having aliases
    for (i = 1; i < w; i++)
      for (j = 1; j < n; j++)
        A[i, j] = 0.25 * (B[i + 1, j] + B[i - 1, j] + B[i, j + 1] + B[i, j - 1]);
  }
} else { // unsafe region, as in Figure 3(a) }

```

Fig. 5. Program of Figure 3(b) after transformation to create alias-free regions.

The alias disambiguation test requires $O(N_w * N_{wr})$ pointer comparisons, where N_w and N_{wr} are respectively the number of distinct arrays (which require alias disambiguation) being written and referenced (written or read) in a safe region. The complexity of analysis is also bounded by $O(N_w * N_{wr})$ (*i.e.*, by $O(N_{wr}^2)$). In practice, N_{wr} is often a small number, making our test quite efficient. The required compiler analysis is efficient as well, making it suitable for both dynamic and static compilers.

5.2. Propagating Precise Alias Information to Later Compiler Phases

The alias versioning technique creates new regions which are, by construction, free of aliasing between arrays. A run-time test verifies this property before executing the alias-free region. It is still necessary to represent and propagate the fact that this region is alias-free, and to do so in a manner that is consistent with the data-flow representation in the rest of the compiler. To achieve this, the alias versioning transformation creates new symbols, one for each existing array symbol in the original region. All old array symbols are replaced with these new symbols in the alias-free region. The alias set of a new symbol is constructed such that all of the alias information for the original symbol is added to the new symbol, but the new symbols themselves are not aliased to each other. An illustration for the

example from Figure 3 is shown in Figure 6.

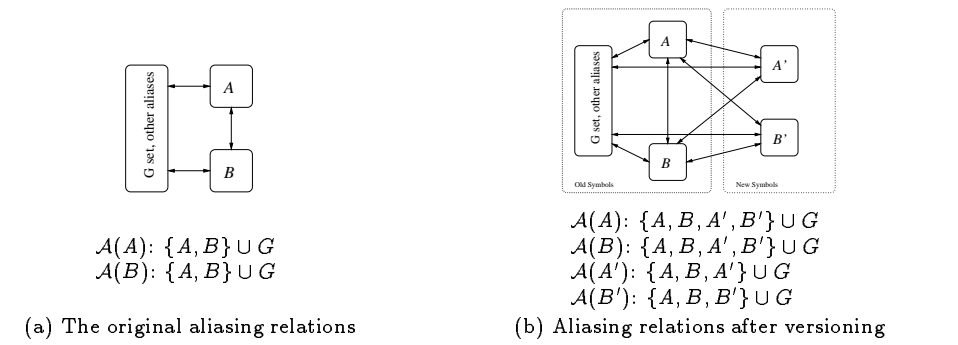


Fig. 6. An example of alias set construction to propagate alias information.

In the original region of Figure 3, Arrays A and B are written and read, respectively. Initially, let A and B be aliased to one another, and to the symbols in G . (G here represents a set of other program symbols, not referenced in this code fragment.) We introduce two new symbols, A' and B' . Let $\mathcal{A}(\alpha)$ denote the alias set of α . The alias set of a new symbol α' is constructed by the following algorithm:

1. $\mathcal{A}(\alpha') = \{\alpha'\}$. The alias set of α' is initialized to α' to honor the reflexivity requirement.
2. $\mathcal{A}(\alpha') = \mathcal{A}(\alpha') \cup \{\sigma \mid \sigma \in \mathcal{A}(\alpha)\}$. All symbols aliased to α are added to the alias set of α' . This includes α because of reflexivity.
3. $\mathcal{A}(\sigma) = \mathcal{A}(\sigma) \cup \{\alpha'\}, \forall \sigma \in \mathcal{A}(\alpha)$. α' is added to the alias set of all symbols in the alias set of symbol α . This operation enforces the symmetry of the aliasing relations.

Applying this algorithm to A and B in Figure 6(a) leads to the alias sets of Figure 6(b). Consider the transformed code shown in Figure 5. In the alias-free region (*i.e.*, the first loop nest in that figure), all references to A (B) are replaced by references to A' (B'). Note that these new symbols do not imply the allocation of any additional storage – they are only introduced to carry alias information. They share storage and all other characteristics (data type, data size, etc) with the original symbols. They are aliased with all symbols that the old symbols are aliased with, *but not with one another*. The net effect is that no aliasing exists between arrays in the newly created region, where the new symbols are introduced.

6. Implementation

Our implementation is based on the IBM *xl* family of compilers, which share a common high-level and low level optimization engines – the Toronto Portable

Optimizer (TPO) and TOBEY, respectively. The alias versioning and safe region creation are implemented in TPO, which performs aggressive dataflow analysis and high level loop transformations.

6.1. *Alias Versioning Implementation*

Code size for a region potentially doubles for each safe region creation. The safe region code size is then doubled again by the alias-free region transformation. As the example in Figure 5 illustrates, complete coverage of a program can lead to a factor of three increase in code size. (However, the safe and alias-free versions are typically simpler than the unsafe version, since they do not contain run-time tests.) By statically evaluating the tests to determine if a region should be executed, versions that will never be executed can be identified as dead code, and removed, helping to mitigate the increase in code size. Note that some high order transformations, such as loop unrolling, will lead to further code growth. Our measurements indicate that, most of the time, code growth is less than 50% [5].

6.2. *Loop Transformations to Improve Data Locality*

TPO applies several well-known high-order transformations to improve the data locality of the program. These transformations include (applied in this order) loop fusion, selective loop distribution, outer loop unrolling (also known as unroll-and-jam), and unimodular loop transformations [6]. All of these transformations would normally be inhibited for Java code, as explained earlier, but can now be applied to safe and alias-free regions created by our techniques. After these high-order transformations (and parallelization), TPO applies additional local transformations like inner loop unrolling and loop interleaving.

The safe region creation and alias versioning transformation are applied individually to, possibly imperfect, loop nests. The dependences from the unsafe region (generated as part of the safe region creation, unless it is optimized away at compile-time) prevent loop fusion from taking place across the original loop nests to which safe region creation is applied [3].

6.3. *Loop Parallelization*

The automatic loop parallelization capabilities of TPO are used to parallelize the Java code. The parallelization transformation is applied immediately after the unimodular loop transformations. The body of the parallel loop is converted into a subroutine using the *outlining* transformation, and the appropriate scheduling policy is used by the run-time system to assign work to various threads participating in parallel execution. The threads used in parallel execution are internal operating system threads, not Java threads. The parallelism is completely transparent from a Java perspective.

7. Results

We used a suite of eight benchmarks to evaluate the performance impact of our techniques. The benchmarks are: MATMUL(multiplication of 500×500 matrices), BSOM(16-node neural network training), MICRODC(Jacobi relaxation over a 1000×1000 grid), TOMCATV(from the SPECfp95 suite, with a problem size of 513), LU(LU decomposition of a 500×500 matrix), CHOLESKY(Cholesky decomposition of a 1000×1000 matrix), SHALLOW(shallow water simulation on a 256×256 grid), and FFT(two-dimensional FFT of 256×256 samples – complex arithmetic implemented explicitly with real number operations). The Java and Fortran version of each program were automatically generated from a single source in a internally developed third language. The translator adjusts the array accesses so that multidimensional arrays are accessed according to their preferred dimension for each language. Results for serial execution were obtained on a 200 MHz POWER 3 machine (IBM RS/6000 model 260), and parallel results are from an SMP system with four of those processors (IBM RS/6000 model N80). The Fortran compiler used was an internal development version of the IBM *xl90* 7.1 compiler, sharing the same optimization engine and back-end with our prototype Java compiler. We note that, because the Fortran and Java compiler share the same optimizers and machine-specific back-end, performance differences are due to language-specific issues.

7.1. Sequential Execution Results

Results for the eight benchmarks, when running in strictly sequential (single-threaded) mode, are summarized in Tables 1 and 2. Table 1 summarizes the static coverage of our optimizations. For each benchmark, column “loops” lists the total number of loop constructs in the program. (The nest of Figure 2 counts as two loops.) Column “covered loops” lists the total number of loops covered by the safe regions created by the compiler. Safe regions are not created for loops for which the compiler cannot compute array range information [3]. Column “safe regions” lists the number of safe regions actually created by the compiler. Alias versioning is applied in each safe region. We note that one safe region can encompass multiple loops. For example, the safe region of Figure 3(b) covers 2 loops. Finally, the column “% coverage” is the ratio, in percentage, of covered loops to total loops.

Table 1. Coverage results for the eight benchmarks.

benchmark	loops	covered loops	safe regions	% covered
MATMUL	14	11	5	79
MICRODC	11	9	5	82
LU	12	10	6	83
CHOLESKY	10	9	5	90
BSOM	25	21	13	84
SHALLOW	20	18	12	90
TOMCATV	18	15	10	83
FFT	19	11	7	58

Table 2 lists the performance achieved by each version of benchmark in our

test hardware. Column “F90” lists the performance for the Fortran version of the benchmark. Column “J116” lists the performance of the Java version under the IBM Development Kit (IBM DK) 1.1.6, which is generally recognized as a high performance Java environment. Results in this column are for versions of the benchmark with Java arrays, since they work better with the IBM DK. Column “Ninja” lists the performance of the Array package version of the benchmark under our prototype optimizing compiler. (The Array package version also uses the POWER/PowerPC fused-multiply-add `fma` instruction.) Two different sets of results are available for Ninja. The “base” results were obtained with a common set of compiler flags for all benchmarks (similar to SPEC baseline performance). The “peak” results were obtained by further specializing compiler options for each benchmark. Column “speedup” shows the speedup between the IBM DK and our prototype compiler, and column “% F90” shows the percentage of Fortran performance that we can accomplish with our prototype compiler for Java. Both speedup and % F90 results are with respect to the “base” measurements.

Table 2. Performance results for the eight benchmarks.

benchmark	Mflops				speedup	% F90
	F90	J116	Ninja			
			base	peak		
MATMUL	403	6.63	340	369	51.1	84%
MICRODC	205	52.6	210	223	4.1	102%
LU	165	44.8	154		3.4	93%
CHOLESKY	172	4.95	167		33.7	97%
BSOM	216	46.8	175		3.7	81%
SHALLOW	188	45.1	156		3.5	83%
TOMCATV	188	49.6	74.5	77.9	1.5	40%
FFT	191	101	104	106	1.03	54%

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. For these benchmarks, the loops covered by the safe region and alias versioning transformations (Table 1) clearly represent the bulk of the computation. We note that the performance of MATMUL and MICRODC can be increased to 369 and 223 Mflops, respectively, by further specializing the compiler flags for each of the benchmarks. For the other benchmarks, little or no benefit is gained by specializing the compiler flags. The superior performance of the Fortran FFT is the result of interprocedural optimizations, interprocedural constant propagation, and procedure cloning, which are performed by the Fortran compiler, but not on the Java code. Performance of the Java version of TOMCATV is significantly lower than the performance of its Fortran counterpart because one of the outer loops in the program is not covered by a safe region.

7.2. Parallel Execution Results

Speedup results, relative to the single processor performance of the parallel code,

are shown in Table 3. In all of the eight benchmarks the compiler was able to parallelize some loops. In six benchmarks (MATMUL, MICRODC, LU, SHALLOW, BSOM, and FFT) significant speedups were obtained (better than 50% efficiency on 4 processors). Although the results were not as good for CHOLESKY and TOMCATV, we note that no significant performance degradation due to parallelization was observed. Due to some compiler limitations, not all loop transformations could be applied when automatic parallelization is enabled. Further results, including details of the amount of code expansion due to program transformations, are described in [5].

Table 3. Speedups on a 4-way POWER3 SMP.

benchmark	Speedup			
	Number of processors			
	1	2	3	4
MATMUL	1.00	1.95	2.88	3.84
MICRODC	1.00	2.02	2.95	3.05
LU	1.00	2.00	2.57	2.27
CHOLESKY	1.00	0.93	1.22	1.44
SHALLOW	1.00	1.75	2.20	2.40
BSOM	1.00	1.65	1.90	2.04
TOMCATV	1.00	1.12	1.15	1.16
FFT	1.00	1.81	2.56	2.40

8. Related Work

Much work has been done on compile-time alias disambiguation (see, e.g. [7]). Our work is complementary to this, so that when static analysis does not prove a region to be alias free, we can construct alias free regions and exploit run-time information for enabling program transformations that lead to good performance.

Versioning is a long-standing technique for optimizing compilers to deal with the lack of complete information at compile time. In particular, [8] discusses versioning in the context of vectorization and parallelization of loops. Our work applies the versioning transformation for a different purpose, to create safe and alias-free regions, taking into account the Java semantics regarding exceptions and pointers.

One approach to high performance numerical computing in Java is the use of high-performance libraries, written both in Java and Fortran. This approach shifts the burden of delivering high performance entirely to the libraries. It is well known that programming with libraries has its limitations. For this reason, compiler techniques that allow the writing of applications with high performance, or the creation of new libraries using Java, is important.

In [9], Cierniak and Li describe a global analysis technique for determining that the shape of a Java array of arrays (*e.g.*, `double[][]`) is indeed rectangular and not modified. The array representation is then transformed to a dense storage and element access is performed through index arithmetic. The Array package differs in that it does not need global analysis to achieve its benefits, since Arrays are known to be dense and rectangular.

9. Conclusions

Although Java is widely recognized as a good programming language for non-numeric applications, there is a widespread belief that it is not suitable for programming high performance numerical applications. We have shown that Java programs can be optimized to the point where they are competitive, performance-wise, with equivalent Fortran code.

In order to achieve this performance, it is necessary to enable the use of the same optimization techniques that have been so successful with Fortran, C and C++. Fortunately, as we have shown in this paper, by utilizing two relatively straightforward transformations, safe and alias-free regions can be created that allow existing optimization techniques to provide the same benefit to Java that they have on other languages. With this approach, Java can become a major platform for the development of high performance numerical codes.

Acknowledgments

The technique of creating new symbols to provide more precise alias information was jointly developed with the TPO group of the IBM Toronto Lab. The compiler infrastructure used in this work was also developed by several compiler groups in the IBM Toronto Lab.

- [1] 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.
- [2] D. A. Padua and M. J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, December 1986.
- [3] 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.
- [4] J. E. Moreira, S. P. Midkiff, and M. Gupta. From flop to megaflops: Java for technical computing. *ACM Transactions on Programming Languages and Systems*, 22(2):265–295, March 2000.
- [5] P. V. Artigas, M. Gupta, S. P. Midkiff, and J. E. Moreira. Automatic loop transformations and parallelization for Java. In *International Conference on Supercomputing*, Santa Fe, NM, May 2000.
- [6] U. Banerjee. Unimodular transformations of double loops. In *Proc. Third Workshop on Programming Languages and Compilers for Parallel Computing*, Irvine, California, August 1990.
- [7] M. Hind, M. Burke, P. Carini, and J.-D. Choi. Interprocedural pointer alias analysis. *ACM Transactions on Programming Languages and Systems*, 21(4):847–893, July 1999. IBM Research Report RC28752.
- [8] M. Byler, J. R. B. Davies, C. Huson, B. Leasure, and M. Wolfe. Multiple version loops. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 312–318, August 17–21 1987.
- [9] M. Cierniak and W. Li. Just-in-time optimization for high-performance Java programs. *Concurrency, Pract. Exp. (UK)*, 9(11):1063–73, November 1997.