RC 21609 (97438) 24NOV99 Computer Science/Mathematics

Research Report

Applying High Order Transformations to Java

P. V. Artigas, M. Gupta, S. P. Midkiff, J. E. Moreira
IBM T. J. Watson Research Center
P. O. Box 218
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Delhi - Haifa - T. J. Watson - Tokyo - Zurich

Applying High Order Transformations to Java

Pedro V. Artigas Manish Gupta Samuel P. Midkiff José E. Moreira
artigas@lsi.usp.br {mgupta,smidkiff,jmoreira}@us.ibm.com

IBM T. J. Watson Research Center
P. O. Box 218
Yorktown Heights, NY 10598-0218

Abstract

The Java programming language is well suited to the domain of numerically intensive applications: it is easy to use; it supports an object oriented programming style that simplifies the development of large numerical programs; it has an extensive set of libraries; it supports interfaces with existing programs written in other languages; its portability makes it ideal for generating codes operating in heterogenous environments; and it boasts of a growing supply of programmers. This paper describes a prototype Java compiler that demonstrates that it is possible to produce Java compilers that approach the performance of current state-of-the-art C, C++ and Fortran compilers. In order to approach these performance levels, it is necessary to integrate the Java compiler with a high-quality loop transformations optimizer and a back-end capable of machine specific optimizations. Combining such a compiler with packages containing the features expected by programmers of numerical applications enables Java to become a serious contender for implementing new numerical applications.

1. Introduction

Java is widely recognized as being a programming language well suited to the development of large complex applications. Nevertheless, the performance of existing Java compilers on numerical codes has prevented the 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. We know of no other Java compiler that is so effective in optimizing numerical codes.

Current successful Fortran compilers for high performance numerical applications include optimizers capable of applying high order transformations [26] to the code. These transformations include, but are not limited to, loop unrolling, unimodular transformations, loop tiling and loop parallelization [24, 4, 31]. There is nothing fundamental in the Java language that prevents the utilization of these same techniques. However, 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, 20] we presented techniques that mitigate the impact of the precise exception model and mandated run-time checks for **null**-pointer and array accesses. In [23] we 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 has been implemented in our high performance prototype compiler for Java. This transformation 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 [3, 20], enable high order transformations, including automatic parallelization, to be applied to Java programs. By doing so, we achieve levels of performance that approach those of tuned Fortran programs.

The rest of the paper is organized as follows. Section 2 briefly discusses the creation of safe regions, which is a precursor to the main contribution of this paper. Section 3 describes the interplay between alias analysis and high order transformations. It motivates the transformation, described in Section 4, for the creation of alias-free regions of the program. This constitutes the main contribution of this paper. Section 5 describes our implementation of these techniques, and Section 6 provides experimental evidence to support our claim that Java can be competitive with Fortran for numerical codes. Section 7 describes some initial work on automatic parallelization of Java programs, and Section 9 presents our conclusions. Finally, Appendix A shows detailed examples of the transformations performed to achieve the results discussed in Sections 6 and 7.

2. Safe region creation

As shown in [3, 22, 20], the Java requirements for array bounds checks and **null**-pointer checks, coupled with its precise exception model, prevent any transformations that change the order of array references. This is a crippling restriction, which limits the applicability of most existing high order transformations to Java code. It becomes critically important to identify or create regions of code that are free of exceptions. One approach to obtain large, exception-free regions where high order transformations can be profitably applied is presented in [3, 22]. This approach is applied automatically by the compiler described in this paper.

For each region (loop nest) to be made safe, the algorithm duplicates the region and removes exception checks from one copy of the region (called the *safe* region or version). The compiler then builds a run-time test that, during execution of the program, selects the safe version of the region to execute if no exceptions can be thrown, or the unsafe version otherwise. Thus, the transformed code always produces the same results as the original code. Although a significant performance increase results directly from eliminating individual exception checks, the greater benefit accrues from being able to optimize the safe region using transformations that change the order of data accesses. An example of this transformation to create safe regions is shown in Figure 1. In that Figure, `CHKn` indicates a check for **null**-pointer and `CHKb` indicates a check for out-of-bounds access.

```
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)]);
  }
}
```

(a) original code

```
/* 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 */
  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)]);
    }
  }
}
```

(b) code after safe region creation

Figure 1. Creation of safe regions.

3. Aliasing and high order transformations

Data dependence analysis is a fundamental technique to ensure that applying high order transformations to a particular program will not change its semantics. A requirement for effective data dependence analysis is the availability of precise alias information – conservative alias information will reduce the accuracy of the dependence analysis, and constrain the transformations that can be performed by the compiler.

3.1. The importance of aliasing information

Consider the loop nest shown in Figure 2 from the kernel of the MICRODC benchmark [21] (with a changed ordering of loops, for the purpose of illustration). That loop nest displays poor data locality if the arrays are stored in row-major order. A good optimizing compiler will, whenever possible, interchange the two loops to improve data locality. If alias information is not available or is overly conservative, the compiler is forced to conclude that the elements of the `A` and `B` arrays may be aliased to each other other, and that true dependences may be carried by both the `i` and `j` loops. Therefore, the compiler will regard loop interchange or any other transformation that alters the order in which elements of `A` and `B` are accessed, as illegal. If precise alias information is available and it shows that no aliasing exists between the elements of `A` and `B`, loop interchange can be applied.

In a loop region where multiple arrays are read and written, it is possible to compute precise dependence information only if the information on array aliasing is known. As discussed in [3, 22], the general case of computing alias in-

```
/* A and B are (w+1,n+1) matrices */
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]);
  }
}
```

Figure 2. A kernel from MICRODC**, illustrating the importance of good aliasing information.**

formation for Java arrays is extremely difficult at compile-time and very expensive at run-time. In this paper we focus on resolving the issue of providing precise alias information for Java one-dimensional arrays and for multidimensional arrays from the Array package for Java [3, 23].

3.2. Impact of language design

In some programming languages like Fortran, the language definition requires subroutine parameters not be aliased (although many real-life programs break this requirement). Languages like C and C++ support unrestricted pointers, which makes alias analysis difficult and often quite conservative. Some C/C++ compilers provide pragmas for specifying additional alias information. Java supports relatively restrictive pointers, compared with C/C++. In particular, Java does not allow getting a pointer to an arbitrary element in the middle of a one-dimensional array. Hence, given two references to one-dimensional arrays, they either refer to identical arrays or to completely non-overlapping arrays. 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 Array package Arrays, which support true multidimensional arrays in Java.

3.3. Our aliasing representation

We now introduce the representation used to describe the aliasing information maintained by our compiler. Each symbol α in the symbol table is associated with a separate alias set $\mathcal{A}(\alpha)$, which consists of all symbols which may be aliased with α (as inferred conservatively using compiler analysis). Alias sets are reflexive: for every symbol α, $\{\alpha\}$ is the minimum alias set. Alias sets are also symmetric: $\beta \in \mathcal{A}(\alpha)$ implies $\alpha \in \mathcal{A}(\beta)$. However, alias sets are *not* transitive. This representation provides sufficient expressiveness to represent complex alias relationships, as in the (C or C++) code fragment of Figure 3. 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.

```
union X {
  long C;
  struct {
    short A;
    short B;
  };
};
```

(a) A C/C++ code fragment

$$
\begin{aligned}
\mathtt{A} &\rightarrow \{\mathtt{A},\mathtt{C}\}\\
\mathtt{B} &\rightarrow \{\mathtt{B},\mathtt{C}\}\\
\mathtt{C} &\rightarrow \{\mathtt{A},\mathtt{B},\mathtt{C}\}
\end{aligned}
$$

(b) The alias sets for the symbols of (a)

Figure 3. An example of alias sets.

4. Alias versioning for Java

Our compiler infrastructure (more specifically, the *Toronto Portable Optimizer*, described further in Section 5) employs context-sensitive interprocedural pointer analysis [18, 15, 30] for Fortran 90 and C to develop more precise alias information. However, this interprocedural analysis is not currently available for Java, due to difficulties associated with handling the dynamic features of the language and ensuring binary compatibility of the generated code (binary compatibility is currently achieved by *not* incorporating the effect of other classes on code generated for a given class). Hence, we propose a different approach which exploits the fact that Java has simpler pointers. We construct a region free of aliases and use a run-time test to determine if the alias-free region can be safely executed. If the test fails, a copy of the region with more conservative alias information is executed. The example in Figure 4 demonstrates this technique. Although we use this technique in lieu of interprocedural alias analysis, in principle, this approach is complementary to alias analysis by the compiler. It can be used to deal with the cases where compiler analysis is too conservative.

Since the alias-free region is obtained by construction within a procedure, precise alias information – potentially better than that obtained by more complex techniques – is available for use by later compilation phases. The cost of this technique is that the test to detect aliases is performed at run-time. No performance benefit will accrue if the test fails, because the original region with very conservative aliasing information is executed. Although a precise run-time test is desirable, we note that it must not be overly

```
    /* region of code containing possibly */
    /* aliased array references          */
    S_1, S_2, ..., S_n
```

(a) before transformation

```
if ( safe to execute alias-free region ) {
  /* same code, but free of aliases */
  S_1, S_2, ..., S_n
} else {
  /* region with original alias information */
  S_1, S_2, ..., S_n
}
```

(a) after transformation

Figure 4. A simple example showing versioning to remove aliases.

complex, to avoid excessive run-time performance penalties. Our benchmark results show that the cost of the test is more than offset by the benefits of the enabled optimizations. In numerically intensive programs, the alias-free regions (loop nests) tend to dominate execution time.

4.1. Array alias disambiguation in Java

With Java, a very simple but precise test may be used for one-dimensional arrays. Because no two one-dimensional Java arrays can partially overlap, only two possibilities exist for aliasing: (i) the two arrays are unrelated, or (ii) the two arrays are the same. Therefore, comparing array object references is sufficient to disambiguate any two one-dimensional Java arrays. This property does not hold for Java arrays of rank greater than one, because they are implemented as arrays of arrays (see [3] for a detailed explanation.) Thus, even if the base pointers of two two-dimensional arrays differ, they may share one or more rows.

Array package [23] Arrays (denoted with a capital “A” in this paper) of any rank can also be disambiguated with a very simple test. For Arrays, the test compares the `data` storage pointer, which is a field in the Array object (see Figure 5). The compiler is able to construct run-time tests that are able to disambiguate Array package Arrays and one-dimensional Java arrays with a simple pointer comparison. An example is shown in Figure 6.

4.2. Array package intersection test

Array package Arrays may share storage with one another, and still not share any data elements. Array sectioning operations create new Arrays that provide new views of the data in the original Array. These Arrays share memory storage with the original Array. Thus two Arrays representing different views of the same underlying storage may contain distinct elements, and therefore not be aliased. The `data` storage pointer test fails in this situation because the `data` storage pointers are equal – it is the elements within the data storage that differ. To disambiguate Array package Arrays in this situation, an intersection test is used to determine if the two Arrays sharing storage share data elements.

Figure 7 shows an example of two Arrays that share storage but are not aliased.. Array `A` shares storage and Array elements with Arrays `A1` and `A2`, but `A1` and `A2` do not share any Array elements. The Array package intersection test includes a *GCD* test [5] and a *range* test. The *GCD* test determines when the sequence of elements in the Arrays do not intersect, without regard to the bounds of the Arrays. If the *GCD* test succeeds, then no Array elements are shared.

```
class doubleArray2D extends doubleArray {
   double[] data; // storage for elements
   int n0, n1; // shape of Array
   int w0, w1, w2; // indexing weights
}
```

Figure 5. Array package Array object description.

```
/* A and B are (w+1,n+1) array package arrays */
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]);
  }
}
```

(a) before alias versioning (note that `A[,]` notation is used instead of `A.get(...)`, `A.set(...)`)

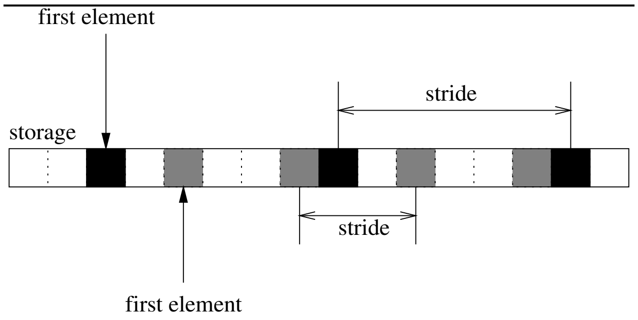
```
if (A.data != B.data) {
  /* A NOT aliased to B */
  for(i=1;i<w;i++) {
    for(j=1;j<n;j++) {
      A[i,j]=0.25*(B[i+1,j]+B[i-1,j]+
                  B[i,j+1]+B[i,j-1]);
    }
  }
} else {
  /* A aliased to B */
  for(i=1;i<w;i++) {
    for(j=1;j<n;j++) {
      A[i,j]=0.25*(B[i+1,j]+B[i-1,j]+
                  B[i,j+1]+B[i,j-1]);
    }
  }
}
```

(b) after alias versioning

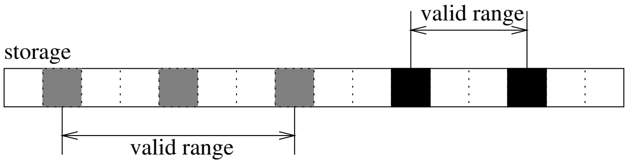
Figure 6. Example of alias versioning for Array package Arrays.

```
doubleArray2D A = new doubleArray2D(size+1,size+1);

/* array A1 shares data storage with A and contains first half of rows of A */
doubleArray2D A1 = A.section(new Range(0,size/2),new Range(0,size));

/* array A2 shares data storage with A and contains second half of rows of A */
doubleArray2D A2 = A.section(new Range(size/2+1,size),new Range(0,size));

boolean A1_intersects_A2 = A1.intersects(A2);
```

Figure 7. Example of intersection operation in the Array package.



(a) GCD test example



(b) range test example

Figure 8. Examples of intersection tests in the Array package.

If the *GCD* shows that elements may be shared, the range test determines if the Array ranges overlap in the data storage region. If they do not overlap, the range test succeeds, indicating that no elements are shared between the two arrays.

Figure 8(a) presents an example where the *GCD* test succeeds in proving that two Arrays do not share elements. Figure 8(b) is a similar example for the *range* test. Note that the *range* test fails when applied to the Arrays in Figure 8(a) because the ranges of elements in the data storage region overlap. Similarly, the *GCD* test fails for the Arrays in Figure 8(b) because the stride and the initial element of both Arrays are compatible.

We emphasize that the technique of the intersection test is applicable only to Array Package Arrays, and not to Java multi-dimensional arrays. Since Java multidimensional arrays are arrays-of-arrays, it is not necessarily true that element $i_1, i_2, ...i_n$ occupies different storage from element $j_1, j_2, ...j_n$ if $j_k \neq i_k$ for some k. Thus, the results of the *GCD* test are not applicable. For the same reason, we cannot conclude that because the coordinates of elements do not overlap the elements themselves do not overlap, which invalidates the range test.

Currently, the Array package intersection test is performed on a linearized form of the indexing expressions. It could be extended so that the test is performed in an axis-by-axis manner. The two tests are analogous to dependence testing of Array references using linearized or dimension-by-dimension subscript expressions. Further refinement of the test is discussed in the next section.

4.3. Dynamic dependence test

The Array package intersection test allows a compiler to use versioning to provide precise alias information to later compilation phases only when two arrays do not have any common elements. If two Arrays share elements, it may still be possible to perform high order transformations and parallelizations if no dependence exists on the aliased elements. By using a run-time dependence test, the compiler might prove that even though the Arrays are aliased: (i) the set of elements accessed in the region being analyzed are not; (ii) only read accesses to aliased elements are performed in the region, and therefore the aliasing relationship can be ignored, (iii) both read and write operations exist on the aliased elements, but they do not preclude the transformation under consideration. In the first two scenarios, a single test is sufficient. In the latter scenario, it might be necessary to modify the test conditions with each transformation.

4.4. 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. The alias set of these new symbols 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 example to illustrate this idea is shown in Figure 9.

In the original region of Figure 9(a), 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. $\mathcal{A}(\alpha') = \mathcal{A}(\alpha') + \sigma, \forall\sigma \in \mathcal{A}(\alpha)$. All symbols aliased to α are added to the alias set of α'. This includes α because of reflexivity.

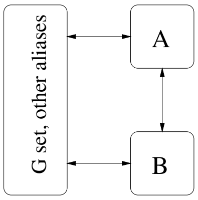
3. $\mathcal{A}(\sigma) = \mathcal{A}(\sigma) + \alpha', \forall\sigma \in \mathcal{A}(\alpha)$. α' is added to the alias set of all symbols in the alias set of symbol α. This operation enforces the symmetry of the aliasing relations.

Applying this algorithm to A and B in Figure 9 leads to the alias sets of Figure 9(c).

After creating the new symbols A′ and B′, the region of code containing the original references is duplicated, as shown in Figure 9(d). A run-time test is constructed to select one of the two regions. In the "alias-free" region, 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.

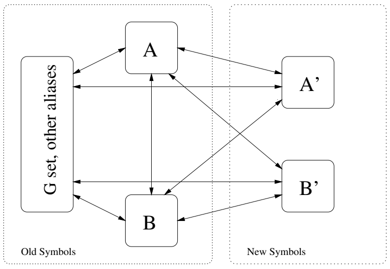
```
/* Arrays A and B are accessed */
for(i=1;i<w;i++) {
  for(j=1;j<n;j++) {
    A[i,j]=0.25*(B[i+1,j]+B[i-1,j]+
                 B[i,j+1]+B[i,j-1]);
  }
}
```

(a) Program region



A: $\{A, B\} \cup G$
B: $\{A, B\} \cup G$

(b) The original aliasing relations



A: $\{A, B\ A', B'\} \cup G$
B: $\{A, B\ A', B'\} \cup G$
A′: $\{A, B, A'\} \cup G$
B′: $\{A, B, B'\} \cup G$

(c) Aliasing relations after versioning

```
if (no alias) /* run-time test */ {
  /* Arrays A' and B' are accessed */
  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 {
  /* Arrays A and B are accessed */
  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]);
    }
  }
}
```

(d) Transformed program of from (a)

Figure 9. An example of alias set construction to propagate alias information

5. Implementation

Our implementation is based on the IBM *xl* family of compilers. Figure 10 shows the high-level organization of those compilers. Alias versioning and safe region creation are implemented in the *Toronto Portable Optimizer* (TPO), the same component that performs aggressive dataflow analysis and high level loop transformations. Semantic expansion of the Array package methods [3] is implemented in

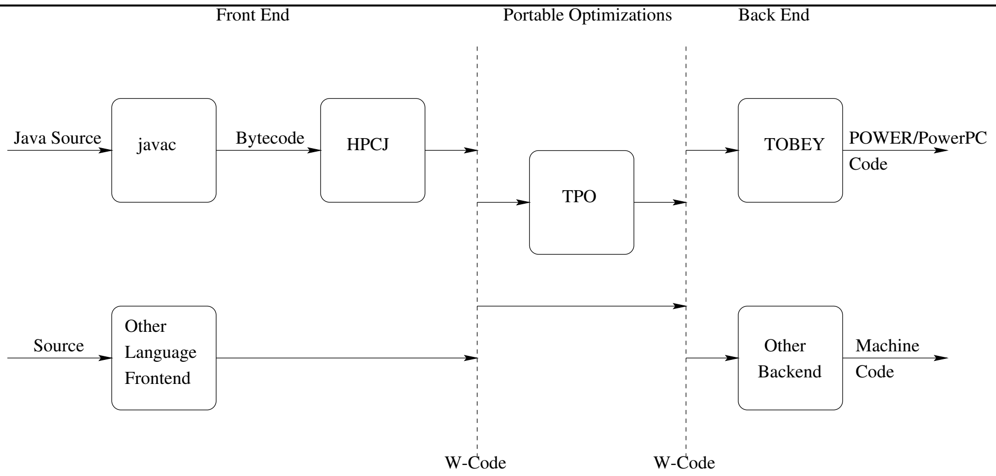


Figure 10. Architecture of the IBM *xl* compilers.

the IBM *High Performance Compiler for Java* (HPJC) [28]. The Fortran compiler we use for comparison has exactly the same organization, with the appropriate language front-end. The same code generation back-end (TOBEY) is used for Java and Fortran. The remainder of this section will describe, in more detail, the implementation of the transformation to construct alias-free regions.

The alias versioning algorithm can be applied to any given program region, but because of the overheads associated with the run-time tests, applying the transformation to every program region is unlikely to maximize performance. The alias versioning transformation enables high order transformations on the alias-free region. These high order transformations, in turn, increase the performance of code in that region. Therefore, there is little or no benefit to applying the alias versioning transformation to program regions where further high order transformations are not possible. Because of constraints introduced by the precise exception model in Java (discussed earlier), regions containing possible exceptions are not considered as candidates for the application of high order transformations. Therefore, the alias versioning transformation is only applied to safe regions. Currently, the versioning transformation is applied to all safe regions.

Our implementation compares base pointers to disambiguate potentially aliased arrays, and it does not use the more sophisticated intersection tests described above. Although this test is the simplest of the three, it results in good performance, as shown in Section 6. We note that this test is valid only for Java one-dimensional arrays and Array package Arrays of any rank.

We now work through an extended example to make concrete the application of the alias versioning transformation. We use the simple kernel from the MICRODC benchmark shown in Figure 1(a). For clarity, exception checks are made explicit by the `CHKn` (**null**-pointer check) and `CHKb` (bounds check) functions. The first optimization step is the creation of safe regions, as shown in Figure 1(b). For this example the safe region creation pass is able to compute the range of indices accessed in the two-deep loop nest of the original program. Therefore, to maximize the size of the created safe region, it duplicates the outermost loop. As shown in Figure 1(b), exception checks are removed from one of the regions (the safe region) and a run-time test is constructed to ensure that the safe region is executed only when a bounds or **null**-pointer exception will not occur.

Figure 11 shows the additional transformation of the code to create alias-free regions. As previously mentioned, the alias versioning transformation is applied to each newly created safe region. In the code of Figure 11, the original safe region is selected for the alias versioning transformation and transformed into two regions: (i) one with new, unaliased, symbols replacing all array references, and (ii) another with the original, potentially aliased, symbols. A run-time test using the `data` storage pointers is constructed.

5.1. Code growth

The application of the safe region creation and alias versioning transformation leads to a potentially large increase in the code size. 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 illustrates, complete coverage of a program can lead to a factor of three increase in code size. To avoid this explosive code growth the compiler attempts to evaluate, statically, the expressions involved in the tests. If this evaluation shows that one form of the region is never ex-

```
/* 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 not free from 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 */
  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)]);
    }
  }
}
```

Figure 11. Program of Figure 1(b) after transformation to create alias-free regions

ecuted, that form becomes dead code and it is eliminated. For example, if a region references only a single array, then it is known at compile-time that no aliasing can occur. The code growth can be further limited (although we have not implemented it) by creating a single safe/alias-free region and a single unsafe/potentially aliased region, with the appropriate tests. This reduces code growth to a factor of two, but may have a negative impact on performance.

Alias versioning is done to enable high order transformations. Some high order transformations, such as loop unrolling, may also lead to code growth, and exert a multiplicative effect on the total code size. (We note that inner loop unrolling is always legal, and can lead to a large code growth.) Applying high order transformations to the unsafe or non-alias-free versions of a region is unlikely to increase performance for two reasons. First, the presence of potential aliasing or exceptions will prevent most high order transformations from happening. Second, for most numerical programs, the alias-free region will dominate the execution time. We mark the unsafe and non-alias-free regions as unsuitable for high order transformations, which reduces code growth as well as the compilation time. Because these regions are necessary for correctness, but are rarely executed and rarely affect performance, we believe this design decision is reasonable.

6. Results

An important consideration in comparing the performance of programs written in different languages is how similar the programs are. For the Fortran and Java results below, the benchmarks were first written in an internally developed third language, *z*-code. An automatic translator then generates Fortran and Java versions of the *z*-code programs, which we then run on an IBM RS/6000-260 workstation. This workstation uses an IBM POWER 3 200 MHz processor, with 32 Kbyte of L1 I-cache, 64 Kbyte of L1 D-cache and a 4 Mbyte unified secondary cache. All benchmarks discussed in this section are strictly sequential (single-threaded). The Fortran compiler used was an internal development version of the IBM *xlf90* 7.1[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 differences.

The eight benchmarks used are: MATMUL, BSOM, MICRODC, TOMCATV, LU, CHOLESKY, SHALLOW, and FFT. MATMUL computes $C = C + A \times B$, where C, A, and B are matrices of size 500×500. We use a dot-product version of matrix multiplication. This benchmark *does not* use the BLAS routines in the Array package. BSOM (Batch Self-Organizing Map) benchmark is a data-mining kernel. We time the execution of the *training* phase of this algorithm, which consists of 25 *epochs*. Each epoch updates a neural network with 16 nodes using 256 records of 256 fields each. MICRODC solves the equation $\nabla^2\Phi = -\frac{\rho}{\epsilon}$ on a discretized domain using Jacobi relaxation [25]. For this benchmark, we use the problem configuration described in [21], with a 1000×1000 grid and four parallel microstrips of cross section 100×10 each. TOMCATV is part of the SPECfp95 suite (`www.spec.org`). For our experiments, we use a problem size $n = 513$. LU computes the L and U factors of a square matrix A such that $PA = LU$, where P is a permutation matrix. The benchmark computes the factors of a 500×500 matrix in place. CHOLESKY computes the Cholesky factor L of a symmetric positive definite matrix A such that $LL^T = A$. The benchmark computes

[1]Because this compiler is an internal compiler, no assumptions should be made about its possible release as a product. No conclusions should be drawn about the performance of future HPCJ based static Java compilers or IBM Fortran compilers based on the results contained in this paper.

the factor of a 1000×1000 matrix in place. SHALLOW is a shallow water simulation kernel derived from an original code from the National Center for Atmospheric Research (NCAR). It operates on 14 matrices of size 256×256. Finally, FFT computes the two-dimensional FFT of 256×256 real samples.

Results for the eight benchmarks 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 1(a) 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. We remind the reader that 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 1(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. We note that the peak performance of a 200 MHz POWER 3 is 800 Mflops. 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 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. This compiler produces better results when the Array package is used [3, 23]. (The Array package version also uses the POWER/PowerPC fused-multiply-add `fma` instruction. This instruction is not currently legal in Java but proposals are being considered to allow its use [1].) 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.

Table 2. Performance results for the eight benchmarks.

benchmark	Mflops			speedup	% F90
	F90	J116	Ninja		
MATMUL	403	6.63	340	51.1	84%
MICRODC	205	52.6	210	4.1	102%
LU	165	44.8	154	3.4	93%
CHOLESKY	167	4.95	172	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	1.5	40%
FFT	191	101	104	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. Particularly remarkable is the result for MICRODC, for which the Java version slightly outperforms Fortran. For these benchmarks, the loops covered by the safe region and alias versioning transformations (Table 1) clearly represent the bulk of the computation. The two benchmarks in which our prototype compiler does not do so well (FFT and TOMCATV) deserve a more detailed analysis.

FFT **performance:** The superior Fortran performance for FFT is the result of inter-procedural analysis being performed by the Fortran compiler. The Java front-end uses calls through function pointers to implement all method calls, and therefore inter-procedural analysis is ineffective with the Java program. This in turn prevents TPO from applying inter-procedural optimizations to Java methods. In the Fortran version of FFT, procedure cloning and inter-procedural constant propagation are performed. Moreover, because the cloned version of a major procedure has some constant parameters, the Fortran compiler is able to perform full loop unrolling on some loops. These transformations are responsible for the performance difference observed between the Fortran and the Java versions. We note that the more refined information obtained after procedure cloning (an inter-procedural optimization), can be used intra-procedurally to perform further optimizations in the cloned procedure. This result indicates that having a tight linkage between inter- and intra-procedural optimizations is beneficial in terms of performance.

We note that the approach taken by the Java front-end to perform all method invocations through function pointers is not required by the Java language. In order to enable the same transformations for Java code, the Java front-end could be altered to allow static methods of classes to be called directly, thus enabling inter-procedure analysis. The same inter-procedural analysis can often also be applied to nonstatic methods if function devirtualization [29] is performed by the front-end. Proper handling of static methods and function devirtualization would allow the same intra-procedural optimizations currently available for Fortran programs to be applied to Java.

TOMCATV **performance:** Performance of the Java version of TOMCATV is significantly lower than its Fortran counterpart 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. In the Fortran version of the benchmark, an unroll-and-jam transformation is applied to this loop. This transformation accounts for most of the performance difference observed between the Java and Fortran versions of the program.

We note that the coverage of loops (in terms of loops covered by safe regions) in TOMCATV is as high as in most of the other benchmarks, but the missed loop performs a large amount of computation. The lost optimization opportunity results in the performance difference. In the other benchmarks no significant optimization opportunity was lost due to the less than perfect coverage of loops.

7. Parallelization

Loop parallelization is another important transformation enabled by alias versioning. To determine how well the automatic parallelizer in TPO works with Java, we conducted experiments on an IBM RS/6000 F50 workstation. This workstation contains 4 PowerPC 604e 332 MHz processors. Each processor has a 32Kbyte instruction cache and a 32Kbyte data cache. A secondary 256Kbyte unified instruction and data cache is shared by all processors. 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, are shown in Table 3. In six of the benchmarks (MATMUL, MICRODC, LU, CHOLESKY, TOMCATV, and FFT) the compiler was able to parallelize some loops. In four benchmarks (MATMUL, MICRODC, LU, and CHOLESKY) significant speedups were obtained. Although the results were not as good for TOMCATV and FFT, and the compiler was not able to parallelize any loops in BSOM, we note that no performance degradation due to parallelization was observed. The cost models built into the compiler prevented applying loop parallelization when it was likely to degrade performance. Due to a compiler bug, we were not able to compile SHALLOW with automatic parallelization enabled. We also note that, due to some other compiler bugs, not all loop transformations can be applied when automatic parallelization is enabled. We want to emphasize that the optimization engine we use is still in a development stage, and subject to constant modifications and enhancements.

Table 3. Parallelization results (speedups) for a 4-way 332 MHz PPC604e SMP.

benchmark	Number of processors			
	1	2	3	4
MATMUL	1.00	1.72	2.52	3.25
MICRODC	1.00	1.87	2.36	2.63
LU	1.00	1.90	2.73	3.43
CHOLESKY	1.00	1.30	1.69	2.01
BSOM	1.00	1.00	1.00	1.00
SHALLOW	–	–	–	–
TOMCATV	1.00	1.20	1.28	1.30
FFT	1.00	1.02	1.02	1.03

As a consequence of the program being automatically parallelized, as opposed to being explicitly parallelized at the Java source level, the parallel threads used in these experiments are internal operating system threads, not Java threads. The parallelism is completely transparent from a Java perspective. The only observable difference is an increase in performance for the Java programs. All parallel loops were obtained from safe and alias free regions. In these regions, no garbage collection may take place. Therefore, no interaction between Java threads and the parallel threads created by our compiler may currently exist.

8. Related work

The analysis of aliasing relationships in programs is well studied [13, 12, 10, 14, 15, 30, 18, 19]. Our work is complementary to this work. Where alias analysis successfully proves that a region is alias free, our techniques are not needed. In regions where this cannot be proved, our techniques allow run-time information to cooperate with static transformations to yield good program performance.

One approach to high performance numerical computing in Java is the use of high-performance libraries. In particular, [7, 8, 27] describe projects to develop such libraries entirely in Java. In addition, there are approaches in which access to existing numerical libraries (typically coded in C and Fortran) is provided to Java applications [6, 9, 17]. All of these approaches shift 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 [11], 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. This approach can lead to results that are as good as from using multidimensional Arrays in the Array package. We differ in that we do not need global analysis: Arrays from the Array package are intrinsically rectangular and can always be stored in a dense form.

The technique of semantic expansion we use is similar to the approach taken by Fortran compilers in implementing some *intrinsic* procedures [2]. Intrinsic procedures can be treated by the Fortran compiler as language primitives, just as we treated the `get` and `set` methods in various classes of the Array package. Semantic expansion differs from the handling of intrinsics in Fortran in that both new data types and operations on the new data types are treated as language primitives.

Marmot [16] is an optimizing static Java compiler being developed at Microsoft Research. It implements both standard scalar (Fortran and C style) optimizations and basic object oriented optimizations. It also performs bounds checking optimization by determining, through a form of range analysis, which checks are redundant. There is no discussion of versioning in [16]. Marmot shows significant speedups relative to other Java environments in several benchmarks, but its performance is only average for the numerical benchmarks listed in [16].

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 is not only competitive with Fortran for writing numerical programs, but by generating code that uses runtime information, it can exceed the performance of current production quality Fortran compilers on some applications.

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, and optimization components, to provide the same benefit to Java that they have to other languages. Because of this, 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.

A. Transformations applied to MATMUL and MICRODC

In order to illustrate the application of high order transformations by our optimizing compiler, we apply the transformations step by step to two simple kernels extracted from our benchmark suite. We also describe the benefits provided by each step. The kernels are taken from the MATMUL and MICRODC benchmarks. The main kernel for MATMUL is presented in Figure 12 as a Java code utilizing the Array package. For clarity, `A[`σ_1, σ_2`]` will be used instead of `A.get(`σ_1, σ_2`)` or `A.set(`σ_1, σ_2`)` to represent array accesses.

```
for(i=0;i<m;i++) {
  for(j=0;j<n;j++) {
    for(k=0;k<p;k++) {
      C[i,j] += A[i,k]*B[k,j];
    }
  }
}
```

Figure 12. Original MATMUL kernel.

The first high order transformation applied to MATMUL is to interchange the `j` and `k` loops to increase the locality of the references to `A` and `B`, as shown in Figure 13. In the case of Array `A` the smallest stride dimension is indexed by the `k` loop. Since the reference to `A[i,k]` is invariant with regard to the loop `j`, successive accesses to array `A` are along the smallest stride dimension. After the interchange all arrays are accessed along the smallest stride dimension.

```
for(i=0;i<m;i++) {
  for(k=0;k<p;k++) {
    for(j=0;j<n;j++) {
      C[i,j] += A[i,k]*B[k,j];
    }
  }
}
```

Figure 13. MATMUL after loop interchange.

Outer loop unrolling (unroll-and-jam) is next applied to the loop nest. TPO only applies outer loop unrolling when no dependences that would prevent the jam phase of outer loop unrolling exist. Also, the compiler will only perform the unrolling if reuse of data brought to the cache increases, either from improvements of temporal or spatial locality. As no dependences exist in the loop nest, both the `i` and `k` loops are candidates for outer loop unrolling. Note that if the `i` loop is unrolled the array reference `B[k,j]` is reused for multiple iterations. Similarly if the `k` loop is unrolled, the array reference `C[i,j]` is reused. Therefore, temporal locality is increased and both the `i` and `k` loops are unrolled by a factor of two, as seen in Figure 14. (We note that this unrolling did not occur with the optimization settings used to produce the results in Table 2 and Table 3, but required passing the `unrlsxxl` option to the compiler. Performance increased from 340 to 369 MFlops.)

```
for(i=0;i<m;i+=2) {
  for(k=0;k<p;k+=2) {
    for(j=0;j<n;j++) {
      C[i,j] += A[i,k]*B[k,j];
      C[i,j] += A[i,k+1]*B[k+1,j];
      C[i+1,j] += A[i+1,k]*B[k,j];
      C[i+1,j] += A[i+1,k+1]*B[k+1,j];
    }
  }
}
```

Figure 14. MATMUL **after outer loop unrolling (fix-up code is omitted).**

The last transformation with significant impact on MATMUL performance is invariant code motion. The TOBEY back-end is able to detect that loads to elements of the array `A` are invariant with regard to the `j`-loop and removes those loads from the loop body. The MATMUL code after invariant code motion is shown in Figure 15.

```
for(i=0;i<m;i+=2) {
  for(k=0;k<p;k+=2) {

    TEMP0 = A[i,k];
    TEMP1 = A[i,k+1];
    TEMP2 = A[i+1,k];
    TEMP3 = A[i+1,k+1];

    for(j=0;j<n;j++) {
      C[i,j] += TEMP0*B[k,j];
      C[i,j] += TEMP1*B[k+1,j];
      C[i+1,j] += TEMP2*B[k,j];
      C[i+1,j] += TEMP3*B[k+1,j];
    }
  }
}
```

Figure 15. MATMUL **after invariant code motion.**

Figure 16 shows the original MICRODC kernel code. The first transformation applied to the MICRODC kernel is not a high order transformation, but is essential for good performance. We present it here to illustrate the benefits that are obtained with Java code when compiled with a high quality code optimizer. The transformation is *dummy load insertion*, which helps the IBM POWER 3 processor hardware prefetch unit. Dummy load insertion places in the loop body a dummy load to an element of the array `A` that is to be written. The dummy load increases performance because it enables the hardware prefetcher in the POWER 3 to prefetch data from the array `A`. Hardware design decisions related to the memory system cause the POWER 3 to only prefetch data that is accessed by load instructions. In the original MICRODC kernel loop data from array `A` is only written. Therefore, no hardware prefetch of `A` is performed. When a dummy load (`JUNK = A[i,j];`) is inserted as shown in Figure 17, prefetching does occur. Elements of `A` are loaded several cycles in advance of being written, allowing the array element to be in the processor prefetch buffer when the write occurs. This decreases the effect of memory latency and bandwidth constraints. These effects are severe enough that the cost of wasting an issue slot for the additional load is negligible.

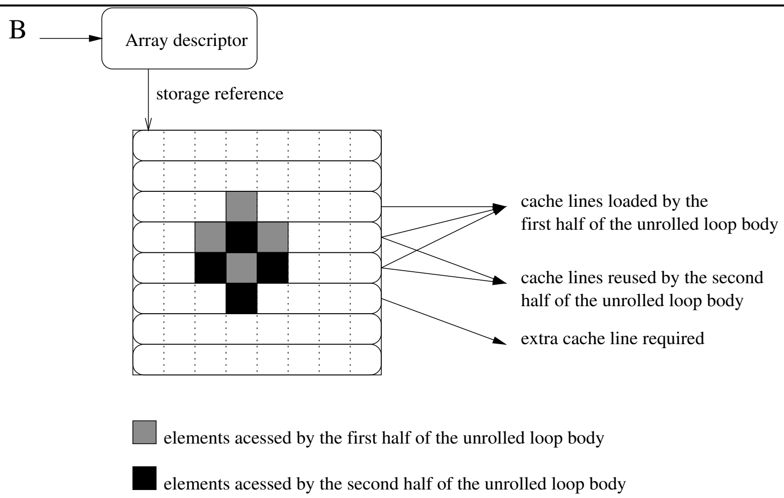
```
for(i=1;i<w;i++) {
  for(j=1;j<n;j++) {
    A[i,j] = 0.25*(B[i+1,j]+B[i-1,j]+
                   B[i,j+1]+B[i,j-1]);
  }
}
```

Figure 16. Original MICRODC **kernel code.**

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

Figure 17. MICRODC **kernel after dummy load insertion.**

The next transformation applied to the MICRODC kernel is outer loop unrolling. The compiler unrolls the outer loop of the nest, which is legal since no dependences exist. This increases spatial locality since in the second half of the unrolled loop body the cache lines accessed by three of the five loads executed are in cache, having been brought by the first half of the unrolled loop body. See Figure 18 for an illustration.




```
for(i=1;i<w;i+=2) {
  for(j=1;j<n;j++) {
    JUNK = A[i,j];
    A[i,j] = 0.25*(B[i+1,j]+B[i-1,j]+B[i,j+1]+B[i,j-1]);
    JUNK = A[i+1,j];
    A[i+1,j] = 0.25*(B[i+2,j]+B[i,j]+B[i+1,j+1]+B[i+1,j-1]);
  }
}
```

Figure 18. MICRODC **kernel after outer loop unrolling.**

Finally, Figure 19 shows the MICRODC kernel after loop iteration interleaving. Loop iteration interleaving is a transformation that enables better instruction scheduling by reordering loads and stores inside a dependence free unrolled loop. It creates opportunities for the use of multiword load instructions. These instructions, present in the POWER 2 and IA-64 architectures, allow two adjacent floating point registers to be loaded through a single instruction. The loop unrolling transformations expose loads to adjacent floating point operands within the same basic block. Since no dependences exist in the loop nest, loop iteration interleaving may be legally applied.

It is important to note that all loop transformations discussed above were applied to both Java and Fortran versions of the benchmarks, and similar levels of performance where obtained for most cases.

```
for(i=1;i<w;i+=2) {
  for(j=1;j<n;j++) {
    (JUNK)=A[i,j];
    (JUNK)=A[i+1,j];
    TEMP0 = B[i,j];
    TEMP1 = B[i,j+1]; load neighbor address
    TEMP2 = B[i+1,j];
    TEMP3 = B[i+1,j+1]; load neighbor address
    A[i,j] = 0.25*(TEMP2+B[i-1,j]+
                   TEMP1+B[i,j-1]);
    A[i+1,j] = 0.25*(B[i+2,j]+TEMP0+
                     TEMP3+B[i+1,j-1]);
  }
}
```

Figure 19. MICRODC **after loop iteration interleaving.**

References

[1] Java Grande Forum report: Making Java work for high-end computing. Java Grande Forum Panel, SC98, Orlando, FL, November 1998. URL: http://www.javagrande.org/reports.htm.

[2] J. C. Adams, W. S. Brainerd, J. T. Martin, B. T. Smith, and J. L. Wagener. *Fortran 90 Handbook: Complete ANSI/ISO Reference*. McGraw-Hill, 1992.

[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*. Springer Verlag, August 1999. IBM Research Division report RC21482.

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

[5] U. Banerjee. *Dependence Analysis*. Loop Transformations for Restructuring Compilers. Kluwer Academic Publishers, Boston, MA, 1997.

[6] A. J. C. Bik and D. B. Gannon. A note on native level 1 BLAS in Java. *Concurrency, Pract. Exp. (UK)*, 9(11):1091–1099, November 1997.

[7] B. Blount and S. Chatterjee. An evaluation of Java for numerical computing. In *Proceedings of ISCOPE'98*, volume 1505 of *Lecture Notes in Computer Science*, pages 35–46. Springer Verlag, 1998.

[8] R. F. Boisvert, J. J. Dongarra, R. Pozo, K. A. Remington, and G. W. Stewart. Developing numerical libraries in Java. In *ACM 1998 Workshop on Java for High-Performance Network Computing*. ACM SIGPLAN, 1998. URL: `http://www.cs.ucsb.edu/conferences/java98`.

[9] 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.

[10] J.-D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *20th Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, pages 232–245, January 1993.

[11] M. Cierniak and W. Li. Just-in-time optimization for high-performance Java programs. *Concurrency, Pract. Exp. (UK)*, 9(11):1063–73, November 1997. Java for Computational Science and Engineering - Simulation and Modeling II, Las Vegas, NV, June 21, 1997.

[12] K. Cooper and K. Kennedy. Faster interprocedural alias analysis. In *Conference Record of the 16'th ACM Symposium on Principles of Programming Languages*, pages 49–59, January 1989.

[13] D. Coutant. Retargetable high level alias analysis. In *Conference Record of the 13'th ACM Symposium on Principles of Programming Languages*, pages 110–118, January 1986.

[14] A. Deutch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*, pages 230–241, Orlando, Florida, June 1994. ACM Press.

[15] M. Emami, R. Ghiya, and L. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 242–256, June 1994.

[16] R. Fitzgerald, T. B. Knoblock, E. Ruf, B. Steensgaard, and D. Tarditi. Marmot: an optimizing compiler for Java. Technical report, Microsoft Research, October 1998. URL: `http://research.microsoft.com/apl/default.htm`.

[17] V. Getov, S. Flynn-Hummel, and S. Mintchev. High-performance parallel programming in Java: Exploiting native libraries. In *ACM 1998 Workshop on Java for High-Performance Network Computing*. ACM SIGPLAN, 1998. URL: `http://www.cs.ucsb.edu/conferences/java98`.

[18] M. Hind, M. Burke, P. Carini, and J.-D. Choi. Interprocedural pointer alias analysis, 1997. to appear, ACM TOPLAS, available as IBM Research Report RC8752.

[19] B. R. J-S Yur and W. Landi. An incremental flow- and context-sensitive pointer aliasing analysis. In *Proceedings of the 1999 International Conference on Software Engineering*, pages 442–451, 1999.

[20] S. P. Midkiff, J. E. Moreira, and M. Snir. Optimizing array reference checking in Java programs. *IBM Systems Journal*, 37(3):409–453, August 1998.

[21] J. E. Moreira and S. P. Midkiff. Fortran 90 in CSE: A case study. *IEEE Computational Science & Engineering*, 5(2):39–49, April-June 1998.

[22] J. E. Moreira, S. P. Midkiff, and M. Gupta. From flop to megaflops: Java for technical computing. *ACM Transactions on Programming Languages and Systems*, 1999. (to appear); Also available as IBM Research Report RC 21166.

[23] J. E. Moreira, S. P. Midkiff, M. Gupta, P. V. Artigas, M. Snir, and R. D. Lawrence. Java programming for high performance numerical computing. Technical Report 21481, IBM Research Division, 1999. To appear in IBM Systems Journal, 2000.

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

[25] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in FORTRAN: The Art of Scientific Computing*. Cambridge University Press, 1992.

[26] 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.

[27] M. Schwab and J. Schroeder. Algebraic Java classes for numerical optimization. In *ACM 1998 Workshop on Java for High-Performance Network Computing*. ACM SIGPLAN, 1998. URL: `http://www.cs.ucsb.edu/conferences/java98`.

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

[29] F. Tip, C. Laffra, and P. F. Sweeny. Practical experience with an application extractor for Java. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'99)*, pages 292–305, Denver, CO, November 1999.

[30] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 1–12, June 1995.

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