Automatic Loop Transformations and Parallelization for 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
Yorktown Heights, NY 10598-0218

ABSTRACT
From a software engineering perspective, the Java programming language provides an attractive platform for writing numerically intensive applications. A major drawback hampering its widespread adoption in this domain has been its poor performance on numerical codes. This paper describes a prototype Java compiler which demonstrates that it is possible to achieve performance levels approaching those of current state-of-the-art C, C++ and Fortran compilers on numerical codes. We describe 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 (for better data locality) and parallelization completely automatically. We believe that our compiler is the first to have such capabilities of optimizing numerical Java codes. We achieve, with Java, between 80 and 100% of the performance of highly optimized Fortran code in a variety of benchmarks. Furthermore, the automatic parallelization achieves speedups of up to 3.8 on four processors. Combining this compiler technology with packages containing the features expected by programmers of numerical applications would enable Java to become a serious contender for implementing new numerical applications.

1. INTRODUCTION
Java[TM] (Java is a trademark of Sun Microsystems, Inc.) 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 [22] to the code. These transformations include, but are not limited to, loop fusion, unimodular transformations, loop tiling and loop parallelization [2, 21, 26]. 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 [1, 17], we presented techniques that mitigate the impact of the precise exception model and mandatory run-time checks for **null**-pointer and array accesses. In [20], 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 [1, 17], 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 technique, 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 various optimization techniques, and Section 6 provides experimental evidence to support our claim that Java can be competitive with Fortran for numerical codes. Section 7 discusses related work. Finally, Section 8 presents our conclusions. For completeness, Appendix A shows detailed examples of the transformations performed to achieve the results discussed in Section 6.

2. SAFE REGION CREATION
As shown in [1, 17, 19], 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 that limits the applicability of most existing high order transformations to Java code. It is 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 [1, 19]. This approach is applied automatically by the compiler described in this paper.

To create an exception-free region (in our case, a loop nest), 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 (which contains the exception checks) 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. These checks are implicit at the Java and bytecode levels, but they become explicit at lower levels of representation. For clarity, we use the notation A[σ_1,σ_2], instead of the A.get(σ_1,σ_2) or A.set(σ_1,σ_2) notation used to represent array accesses in the Array package.

```
    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) = array X size 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 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 [18] (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. Therefore, it 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.

```
// 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]);
```

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

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 [1, 19], the general case of computing alias information for Java multidimensional arrays (which are arrays of 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 [1, 20]. We note that the Array package is written entirely in Java. Applications based on the Array package are fully portable and Java compliant.

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. Java supports relatively restrictive pointers compared to C/C++. In particular, Java does not allow setting a pointer to an arbitrary element 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 multidimensional Arrays from the Array package.

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

$$A \rightarrow \{A,C\}$$
$$B \rightarrow \{B,C\}$$
$$C \rightarrow \{A,B,C\}$$

(b) Alias sets for A, B, and C

Figure 3: An example of alias sets.

4. ALIAS VERSIONING FOR JAVA

Our compiler infrastructure (more specifically, the *Toronto Portable Optimizer*, as described in Section 5) employs context-sensitive interprocedural pointer analysis [11, 14, 25] for Fortran 90 and C to develop more precise alias information. However, this interprocedural analysis is not currently available for Java because of 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. 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.

Since the alias-free region is obtained by construction within a procedure, precise alias information – potentially better than that obtained by more complex compile-time 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 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 [1] 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 [20] 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 4). 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. Figure 5 shows the result of applying this transformation to the safe region of the program fragment in Figure 1.

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.

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

Figure 4: Array package Array object description.

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

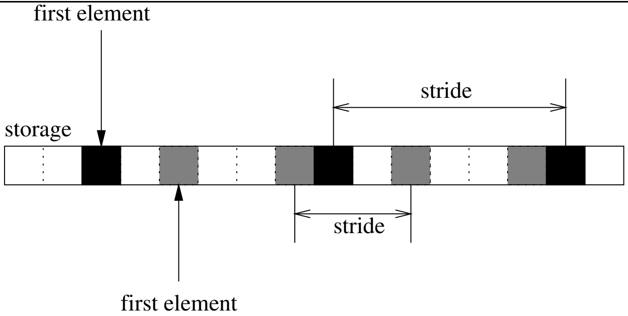
Figure 5: Program of Figure 1(b) after transformation to create alias-free regions.

4.2 Array package intersection test

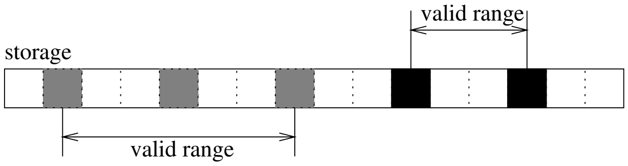
The Array sectioning operations create new Arrays that provide new views of the data in the original Array. These Arrays share memory storage with the original Array, but may not share any data elements. The data storage pointer test fails to provide alias disambiguation in this situation because the data storage pointers are equal – it is the elements within the data storage that differ. To disambiguate Array package Arrays in this situation, we now describe an intersection test (our current implementation of the versioning transformation does not yet exploit this test).

```
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 6: Example of intersection operation in the Array package.



(a) *GCD* test example



(b) *range* test example

Figure 7: Examples of intersection tests in the Array package.

Figure 6 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 [3] 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. 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 7(a) presents an example where the *GCD* test succeeds in proving that two Arrays do not share elements. Figure 7(b) is a similar example for the *range* test. Note that the *range* test fails when applied to the Arrays in Figure 7(a) because the ranges of elements in the data storage region overlap. Similarly, the *GCD* test fails for the Arrays in Figure 7(b) because the stride and the initial element of both Arrays are compatible.

4.3 Dynamic dependence test

The Array package intersection test results in execution of the alias-free region only when two arrays do not have any common elements. If two Arrays share elements, it may still be possible to execute the alias-free region if no dependence exists on the referenced elements. By using a run-time dependence test, it might be possible to 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 a particular optimizing transformation. In the first two scenarios, a single general test is sufficient. In the latter scenario, it might be necessary to modify the test conditions with each transformation.

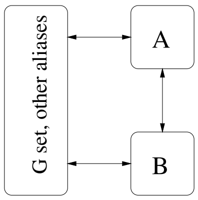
4.4 Propagating precise alias information to later compiler phases

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 dataflow 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 1 is shown in Figure 8.

In the original region of Figure 1, 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, $\mathtt{A}'$ and $\mathtt{B}'$. Let $\mathcal{A}(\alpha)$ denote the alias set of α. The alias set of a new symbol α' is constructed by the following algorithm:

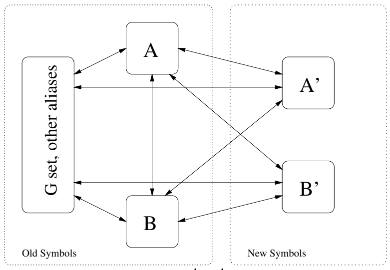
1. $\mathcal{A}(\alpha') = \{\alpha'\}$. The alias set of α' is initialized to α' to honor the reflexivity requirement.
2. $\mathcal{A}(\alpha') = \mathcal{A}(\alpha') \cup \{\sigma | \sigma \in \mathcal{A}(\alpha)\}$. All symbols aliased to α are added to the alias set of α'. This includes α because 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 8(a) leads to the alias sets of Figure 8(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 $\mathtt{A}'$ ($\mathtt{B}'$). Note



$\mathcal{A}(\mathtt{A})$: $\{\mathtt{A},\mathtt{B}\} \cup G$
$\mathcal{A}(\mathtt{B})$: $\{\mathtt{A},\mathtt{B}\} \cup G$

(a) The original aliasing relations



$\mathcal{A}(\mathtt{A})$: $\{\mathtt{A},\mathtt{B},\mathtt{A}',\mathtt{B}'\} \cup G$
$\mathcal{A}(\mathtt{B})$: $\{\mathtt{A},\mathtt{B},\mathtt{A}',\mathtt{B}'\} \cup G$
$\mathcal{A}(\mathtt{A}')$: $\{\mathtt{A},\mathtt{B},\mathtt{A}'\} \cup G$
$\mathcal{A}(\mathtt{B}')$: $\{\mathtt{A},\mathtt{B},\mathtt{B}'\} \cup G$

(b) Aliasing relations after versioning

Figure 8: An example of alias set construction to propagate alias information.

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.

5. IMPLEMENTATION

Our implementation is based on the IBM *xl* family of compilers. Figure 9 shows the high-level organization of those compilers. The alias versioning and safe region creation are implemented in the *Toronto Portable Optimizer* (TPO), the component that performs aggressive dataflow analysis and high level loop transformations. Semantic expansion of the Array package methods [1] is implemented within the IBM *High Performance Compiler for Java* [24] (HPCJ). 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, and the transformations for locality and automatic parallelization that are thereby enabled.

5.1 Alias versioning implementation

The alias versioning transformation can be applied to any program region, but there is little to be gained from applying it to program regions where further high order transformations are not possible. Therefore, alias versioning is only applied to safe regions. Our implementation compares base pointers to disambiguate potentially aliased arrays, and does not use the more sophisticated intersection tests described in Section 4. Although this test is the simplest of the three, it results in good performance as shown in Section 6.

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 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.) To avoid this explosive code growth, the compiler attempts to statically evaluate the expressions involved in the tests. If this evaluation shows that one form of the region is never executed, 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 can have a negative impact on performance. Note that some high order transformations, such as loop unrolling, will lead to further code growth.

5.2 Loop transformations to improve data locality

The Toronto Portable Optimizer 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 [2, 22, 26]. 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. Finally, after these high-order transformations and parallelization, TPO applies additional local transformations like inner loop unrolling and loop interleaving. Appendix A describes, in further detail, the effect of some of the loop transformations on two examples.

Currently, 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 [1]. In the future, we shall investigate using more aggressive forms of safe region creation which will enable more global program transformations.

5.3 Loop Parallelization

We rely on the automatic loop parallelization capabilities of TPO to parallelize Java code. The safe region creation and alias versioning transformations, which enable loop transformations for locality, also enable loop parallelization. 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 (by default, the run-time uses static scheduling).

As a consequence of the program being automatically parallelized, instead of being explicitly parallelized at the Java source level, the threads used in parallel execution are internal operating system threads, not Java threads. The parallelism is completely transparent from a Java perspective. In particular, no garbage collection is per-

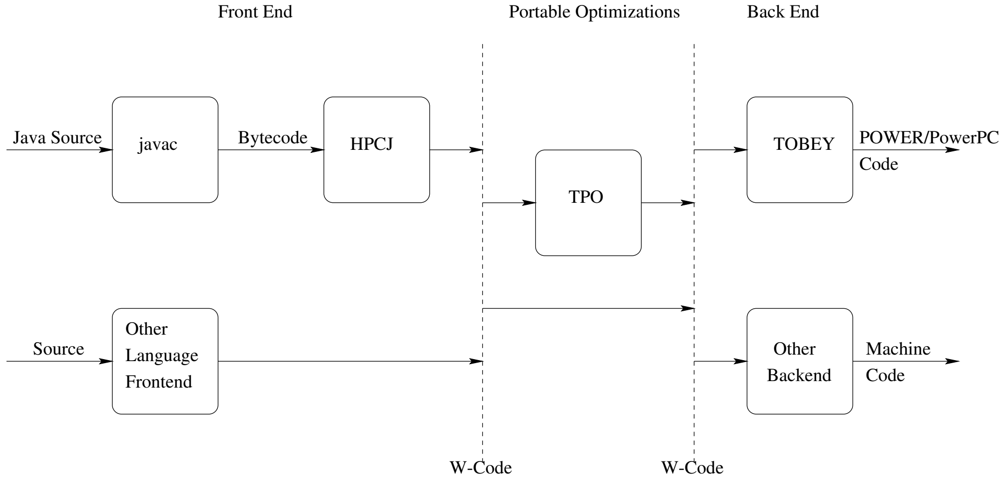


Figure 9: Architecture of the IBM *xl* compilers.

formed on these internal operating system threads used for parallel execution. This does not create a problem for our current implementation, because our parallel regions cannot contain an explicit memory allocation statement. This is because all parallel loops are obtained from safe (exception free) and alias-free regions. Memory allocation statements may throw an exception, necessarily making the region unsafe.

6. RESULTS

We used a suite of eight benchmarks to evaluate the performance impact of our techniques. These benchmarks are described further in [1] and [20]. 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. The translator adjusts the array accesses so that multidimensional arrays are accessed according to their preferred dimension – column-major for Fortran and row-major for Java. Results for serial execution were obtained on a 200 MHz POWER 3 machine (IBM RS/6000 model 260), while 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 *xlf90* 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.

6.1 Sequential 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 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 [1]. 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 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. Java results were obtained with a common set of compiler flags for all benchmarks (similar to SPEC baseline performance). We will indicate in the text when we can obtain better performance by specializing compiler options for each benchmark. 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. This compiler produces better results when the Array package is used [1, 20]. (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 [16].) 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.

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

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	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	1.5	40%
FFT	191	101	104	1.03	54%

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. 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. In particular, for MATMUL we force unrolling of the innermost loop and for MICRODC we ignore the limit on maximum simultaneous data streams. (Appendix A has more details.) The two benchmarks in which our prototype compiler does not do so well (FFT and TOMCATV) deserve a more detailed analysis.

6.1.1 FFT *performance*

The superior Fortran performance for FFT is the result of interprocedural optimizations, interprocedural constant propagation and procedure cloning, performed by the Fortran compiler. As we mentioned earlier, interprocedural analysis in TPO does not currently work with Java. Because the cloned version of a major procedure in this benchmark 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 interprocedural optimization), can be used intraprocedurally 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.

6.1.2 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 automatically 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.

6.2 Parallel results

Loop parallelization is another important transformation enabled by alias versioning. In this section, we report speedup results from loop parallelization of our Java benchmarks. All experiments were conducted using the Array package version of the benchmarks, compiled with our prototype compiler with automatic parallelization enabled.

Speedup results, relative to the single processor performance of the parallel code, are shown in Figure 10, both in tabular and graphical forms. 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. (There was a small degradation for CHOLESKY on two processors.) The cost models built into the compiler prevented applying loop parallelization when it was likely to degrade performance. Due to some current compiler limitations, not all loop transformations could be applied when automatic parallelization is enabled. The optimization engine we have used is still in a development stage, and subject to constant modifications and enhancements.

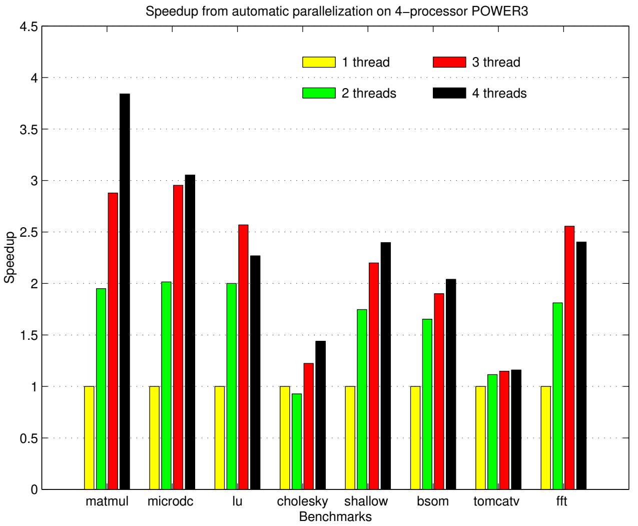


Figure 10: Speedups on a 4-way POWER3 SMP.

6.3 Code expansion

Table 3 reports the size of the executable for each benchmark, under different conditions. Column "baseline" shows the size, in bytes, for the code compiled without any safe region or aliasing versioning. Column "versioning" shows the size and growth (as a percentage of baseline) when the code is compiled with the safe region and aliasing versioning, and loop transformations are applied. Finally, column "parallelization" shows the corresponding results when automatic parallelization is further applied.

We observe that code growth can indeed be substantial, such as for SHALLOW and TOMCATV. For all other benchmarks the code expansion, without parallelization, is less than 50%. Given the fact that numerical applications typically run on large memory systems, and that their data sets are much larger than the code, we believe that code expansion is a small price to pay for the enormous boost in performance.

7. RELATED WORK

The analysis of aliasing relationships in programs is well studied [10, 11, 14, 15, 25, 27]. Our work is complementary to this pre-

Table 3: Code expansion for the eight benchmarks.

benchmark	Executable size				
	baseline	versioning		parallelization	
	(bytes)	(bytes)	(growth)	(bytes)	(growth)
MATMUL	11986	16858	41%	17950	50%
MICRODC	15978	18098	13%	17950	12%
LU	14160	14656	4%	15328	8%
CHOLESKY	11836	13428	13%	14628	24%
BSOM	24636	34684	41%	42008	71%
SHALLOW	28252	63476	125%	73168	159%
TOMCATV	16006	30286	89%	35130	119%
FFT	19744	29200	48%	32740	66%

vious 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.

Versioning is a long-standing technique for optimizing compilers to deal with the lack of complete information at compile time. In particular, [7] discusses versioning in the context of vectorization and parallelization of loops. Run-time tests of loop increments, data dependences, and loop bounds are used to select the best version (sequential, vector, parallel, vector/parallel) of a loop nest for execution. 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. In particular, [5, 6, 23] 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 [4, 8, 13]. 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 [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. Alias analysis (at compile- or run-time) is facilitated in this case. 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, which is more appropriate for alias and dependence analysis.

Marmot [12] 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 [12]. Marmot shows significant speedups relative to other Java environments in several benchmarks, but its performance is only average for the numerical benchmarks listed in [12].

8. 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, and optimization components, 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.

9. REFERENCES

[1] P. V. Artigas, M. Gupta, S. P. Midkiff, and J. E. Moreira. High performance numerical computing in Java: Language and compiler issues. In J. Ferrante et al., editors, *12th International Workshop on Languages and Compilers for Parallel Computing*. Springer Verlag, August 1999. IBM Research Division report RC21482.

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

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

[4] 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.

[5] 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.

[6] R. F. Boisvert, J. J. Dongarra, R. Pozo, K. A. Remington, and G. W. Stewart. Developing numerical libraries in Java. *Concurrency, Pract. Exp. (UK)*, 10(11-13):1117–29, September-November 1998. ACM 1998 Workshop on Java for High-Performance Network Computing. URL: `http://www.cs.ucsb.edu/conferences/java98`.

[7] 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.

[8] 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.

[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. Java for Computational Science and Engineering - Simulation and Modeling II, Las Vegas, NV, June 21, 1997.

[10] K. D. 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.

[11] 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.

[12] 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%discretionary//default.htm`.

[13] 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`.

[14] 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. available as IBM Research Report RC28752.

[15] J. Hummel, L. J. Hendren, and A. Nicolau. A general data dependence test for dynamic, pointer-based data structures. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 242–256, June 1994.

[16] Java Grande Forum. *Java Grande Forum Report: Making Java Work for High-End Computing*, November 1998. Java Grande Forum Panel, SC98, Orlando, FL. URL: `http://www.javagrande.org/reports.htm`.

[17] 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.

[18] 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.

[19] 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.

[20] J. E. Moreira, S. P. Midkiff, M. Gupta, P. V. Artigas, M. Snir, and R. D. Lawrence. Java programming for high performance numerical computing. *IBM Systems Journal*, 39(1):21–56, 2000. IBM Research Division report RC21481.

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

[22] 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.

[23] 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`.

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

[25] 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.

[26] 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.

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

APPENDIX

A. TRANSFORMATIONS APPLIED TO MATMUL AND MICRODC

In order to illustrate the application of high order transformations by our optimizing compiler, we describe in detail the transformations applied to two simple kernels extracted from our benchmark suite. The kernels are taken from the MATMUL and MICRODC benchmarks.

Figure 11(a) shows the safe region of the main kernel for MATMUL. 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`. 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.

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 this unrolling if it enhances the reuse of data brought to the cache from improvements in either temporal or spatial locality. As no dependences exist in the loop nest, both `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,

```
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];
```

(a) original MATMUL kernel

```
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];
    }
  }
}
```

(b) MATMUL after loop transformations

Figure 11: Optimizing MATMUL kernel: (a) original code, (b) after loop interchange, outer loop unrolling (fix up code omitted), and invariant code motion.

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.

The last transformation with a significant impact on MATMUL performance is invariant code motion. The optimizer is able to detect that loads to elements of the array `A` are invariant with respect to the `j`-loop and removes those loads from the loop body. The MATMUL code after all of these transformations is shown in Figure 11(b).

Figure 12(a) shows the safe region of the original MICRODC kernel code. The first transformation applied to this kernel is not a high order transformation, but is essential for good performance. 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 being written. This enables the hardware prefetcher in the POWER 3 (which only prefetches data being read, not data being written) to prefetch data from the array `A`.

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 (four loads of `B`, plus the dummy load of `A`) are in cache, having been brought by the first half of the unrolled loop body. See Figure 13 for an illustration.

The final transformation is loop iteration interleaving, which enables better instruction scheduling by reordering loads and stores inside a dependence-free unrolled loop. It creates opportunities for the use of multi-word 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

```
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) original MICRODC kernel

```
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]);
  }
}
```

(b) MICRODC kernel after loop transformations

Figure 12: Optimizing MICRODC: (a) original code, (b) after dummy load insertion, outer loop unrolling, and loop iteration interleaving.

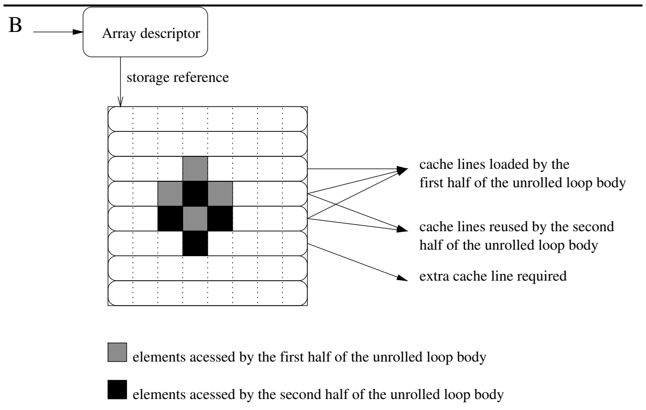


Figure 13: Cache line reuse opportunities in MICRODC kernel.

exist in the loop nest, loop iteration interleaving may be legally applied. The transformed program segment is shown in Figure 12(b).

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 were obtained for most cases.