RC 21184 (94652) 05/18/98
Computer Science/Mathematics

IBM Research Report

Optimizing Array Reference Checking in Java Programs

Samuel P. Midkiff, José E. Moreira, Marc Snir

IBM Research Division
T.J. Watson Research Center
Yorktown Heights, New York

IBM Research Division
Almaden · T.J. Watson · Tokyo · Zurich

This page intentionally left blank.

Optimizing Array Reference Checking in Java Programs

Samuel P. Midkiff José E. Moreira Marc Snir
{smidkiff,jmoreira,snir}@us.ibm.com

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

Abstract

The Java language specification requires that all array references be checked for validity. If a reference is invalid, an exception must be thrown. Furthermore, the environment at the time of the exception must be preserved and made available to whatever code handles the exception. Performing the checks at run-time incurs a large penalty in execution time. In this paper we describe a collection of transformations that can dramatically reduce this overhead in the common case (when the access is valid) while preserving the program state at the time of an exception. The transformations allow trade-offs to be made in the efficiency and size of the resulting code, and are fully compliant with the Java language semantics. Preliminary evaluation of the effectiveness of these transformations show that performance improvements of 10 times and more can be achieved for array-intensive Java programs.

1 Introduction and motivation

Three goals of the Java(TM)[1] programming language [1] and execution environment are bit-for-bit reproducibility of results on different platforms, safety of execution, and ease of programming and testing. A crucial component of the Java language specification for achieving the latter two goals is that a program only be allowed to access objects via a valid pointer, and that programs only be allowed to access elements of an array that are part of the defined extent of the array. A naive implementation of this specification component requires every access to every element of an array to check the validity of all base pointers and indices involved in the access.

Figure 1 shows a typical representation of a four-dimensional array in Java. A naive checking of a reference to element $A[1][2][3][4]$, indicated in the figure, would require four base pointer tests and four range tests: A, $A[1]$, $A[1][2]$, and $A[1][2][3]$ must all be tested as valid pointers; 1, 2, 3, and 4 must all be tested as valid indices along the corresponding axes of the array. If the entire array is accessed, a total of $4N$ base pointer tests and $4N$ range tests will be necessary, where N is the total number of elements in the array. In this paper, we present a suite of techniques to greatly reduce the number of run-time tests. In many cases, the run-time tests can be completely eliminated.

The above mentioned goal of bit-for-bit reproducibility of results, combined with the ability to catch exceptions and examine the state of the program at the time the exception was thrown, imply that it is not sufficient to determine that an invalid reference occurred. Rather, any optimizations of valid reference checking must cause all exceptions that would have been thrown in the original program to still be thrown. The exceptions must be thrown in precisely the same order, with respect to the rest of the computation,

[1] Java is a trademark of Sun Microsystems, Inc.

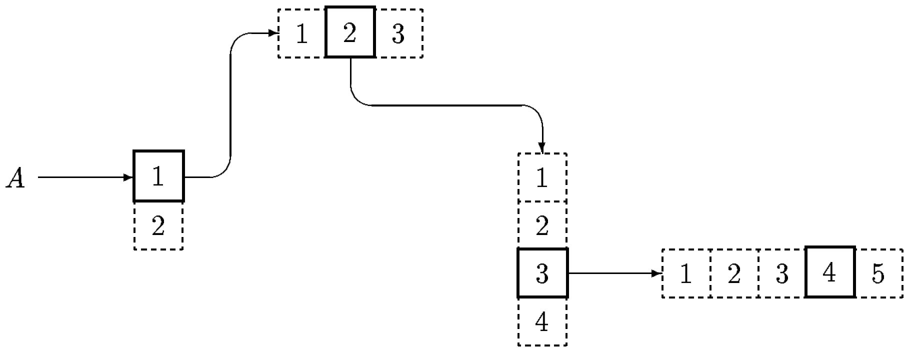


Figure 1: Access of an element in a four-dimensional array.

dictated by the original program semantics. The techniques we describe fulfill this requirement. Our methods also handle loops that catch exceptions within their bodies (*i.e.*, with nested **try** blocks).

We note that the techniques and methods described here are not restricted to Java. They can be applied to any language in which the bounds of an array can be determined at run-time. They could be used, for example, in C and Fortran compilers that want to provide an option to check array references.

The rest of this paper is organized as follows. Section 2 presents an informal overview of our optimizations. Section 3 introduces the concept of safe bounds, used throughout the paper. Section 4 presents the first of our optimization techniques, the exact method. Sections 5, 6, and 7 present other techniques, the general, compact, and restricted methods, respectively. Section 8 discusses an inspector/executor variant of our methods, to handle more complex cases. Section 9 explains our optimizations in the context of multithreaded execution. Section 10 presents some experimental results, and Section 11 discusses related work. Finally, Section 12 presents our conclusions.

2 Overview of the optimizations

The main goal of our work is to develop techniques that minimize the number of run-time tests that must be performed for array reference operations in Java. An array A is defined by a lower bound $\mathrm{lo}(A)$ and an upper bound $\mathrm{up}(A)$. We use a Java-like notation for array declarations:

$$\mathsf{double}\ A[] = \mathbf{new}\ \mathsf{double}[\mathrm{lo}(A) : \mathrm{up}(A)]$$

declares a one-dimensional array A of doubles. Note that we allow an explicit declaration of the lower bound $\mathrm{lo}(A)$ of array A. In Java, the lower bound is always zero. Also, Java array declarations specify the number of elements of the array. Therefore, the Java declaration

```
double A[] = new double[n]
```

would correspond to the declaration

$$\mathsf{double}\ A[] = \mathbf{new}\ \mathsf{double}[0 : n-1]$$

in our notation.

An array element reference is denoted by $A[\sigma]$, where σ is an *index* into the array. For the reference to be valid, A must not be **null**, and σ must be within the *valid range* of indices for A: $\mathrm{lo}(A), \mathrm{lo}(A)+1, \ldots, \mathrm{up}(A)$. If A is **null**, then the reference causes a *null pointer* violation. In Java, this corresponds to the throwing of a `NullPointerException`. If A is a valid pointer and σ is less than the lower bound $\text{lo}(A)$ of the array, then the reference causes a *lower bound* violation. If A is a valid pointer and σ is greater than the upper bound $\text{up}(A)$ of the array, then the reference causes an *upper bound* violation. Java does not distinguish between lower and upper bound violations. A bound violation (lower or upper) in Java causes an `ArrayIndexOutOfBoundsException` to be thrown.

Multi-dimensional arrays are treated as arrays of arrays:

$$\mathsf{double}\ M[][] = \mathbf{new}\ \mathsf{double}[l_1 : u_1][l_2 : u_2]$$

declares an array M with lower bound l_1 and upper bound u_1. Each element of M is an array of doubles, with lower bound l_2 and upper bound u_2. M is an example of a *rectangular array*. Rectangular arrays have uncoupled bounds along each axis. As in Java, we also allow *ragged arrays*, where the bounds for one axis depend on the coordinate along another axis. A triangular array is an example of a ragged array:

$$\mathsf{double}\ T[][] = \mathbf{new}\ \mathsf{double}[1 : n][]$$
$$T[i] = \mathbf{new}\ \mathsf{double}[1 : i],\ i = 1, \ldots, n$$

Even though ragged arrays are allowed and treated by our techniques, we do have optimizations that take advantage of rectangular arrays.

Arbitrary array lower bounds, the distinction between lower bound and upper bound violations, and treatment for rectangular arrays are features of our work that are not strictly necessary for Java applications. Our motivation to include them is two-fold: First, we want our methods to be applicable to other programming languages, in particular C, C++, and Fortran 90. Second, we want to be prepared to handle extensions to Java that are being considered to efficiently support numerical computing, as described in [2].

Array accesses typically occur in the body of loops. Our optimizations operate on **do**-loops, which are **for**-loops of the form:

$\mathbf{for}\ (i = l;\ i \le u;\ i{+}{+})\ \{$
$\quad B(i)$
$\}$

where i is the index variable of the loop, l defines the lower bound of the loop, and u defines the upper bound of the loop. The iteration space of the loop is defined by the range of values that i takes: $l, l+1, \ldots, u$. All loops have a unit step, and therefore a loop with $l > u$ has an empty iteration space. $B(i)$ is the body of the loop, which typically contains references to the loop index variable i. As a shorthand, we represent the above loop structure by the notation $L(i, l, u, B(i))$. Note that loops with positive nonunity strides can be normalized by the transformation

$$\boxed{\begin{array}{l}\mathbf{for}\ (i = l;\ i \le u;\ i = i + s)\ \{\\ \quad B(i)\\ \}\end{array}} \Longrightarrow \boxed{\begin{array}{l}\mathbf{for}\ (i = 0;\ i \le \left\lfloor \frac{u-l}{s} \right\rfloor;\ i{+}{+})\ \{\\ \quad B(l + is)\\ \}\end{array}}$$

A loop with negative stride can be first transformed into a loop with a positive stride:

$$\boxed{\begin{array}{l}\mathbf{for}\ (i = u;\ i \ge l;\ i = i - s)\ \{\\ \quad B(i)\\ \}\end{array}} \Longrightarrow \boxed{\begin{array}{l}\mathbf{for}\ (i = l;\ i \le u;\ i = i + s)\ \{\\ \quad B(u + l - i)\\ \}\end{array}}$$

Loops are often nested within other loops. Standard control-flow and data-flow techniques [3] can be used to recognize many `for`, `while`, and `do-while` loops, which occur in Java and C, as **do**-loops. Many `goto` loops, occurring in C and Fortran, can be recognized as **do**-loops as well.

Four different tests can be applied to an array reference $A[i]$. A *null test* verifies whether A is a **null** pointer. An *lb test* verifies whether $i \geq \mathrm{lo}(A)$, while a *ub test* verifies whether $i \leq \mathrm{up}(A)$. Finally, an *all tests* verifies whether $\mathrm{lo}(A) \leq i \leq \mathrm{up}(A)$. Tests for bounds subsume **null** tests, since a **null** array can be set to have an empty extent. Furthermore, we show in the next section that for **do**-loops only one of three cases can occur for each iteration and each array reference: (i) an *lb test* is required, (ii) a *ub test* is required, or (iii) no test is required. In many situations it is possible to precisely determine which case occurs. In other situations, one has to be conservative and adopt a stronger test than absolutely required. In particular, an *all tests* can be used to detect any possible violations.

We introduce in Section 4 a method for deriving the minimum set of tests required at each iteration of a simple loop with no nesting. This method will be referred to as the *exact* method. It provides the general framework that is used by the subsequent methods, which handle nested loops. The exact method specifies, for each iteration and each array reference, which test is required, if any: *lb test*, *ub test*, or *all tests*. A loop with ρ array references in its body is split at compile time into up to 3^ρ consecutive regions, each with a specialized version of the loop body. At run time, no more than $2\rho + 1$ regions are actually executed.

This level of code replication is not practical in general. However, one can reduce the number of code versions by merging together regions, at the expense of performing superfluous tests. This is shown in Section 5. In practice, this does not increase execution time. The regions where tests are required will usually be found to be empty at run time so that the tests in these regions are never normally executed. Reducing the number of distinct regions will not only decrease code size, but may also decrease execution time. A practical version of the exact method splits an iteration space $i = l, l+1, \ldots, u$ into three regions:

1. A region of the iteration space where no tests are needed. This region is defined by a *safe lower* bound l^s and a *safe upper* bound u^s. The range of values of i in this region is $l^s, l^s + 1, \ldots, u^s$.

2. A region of the iteration space where the index variable has values smaller than the safe lower bound l^s. For this region $i = l, l+1, \ldots, l^s - 1$.

3. A region of the iteration space where the index variable has values greater than the safe upper bound u^s. For this region $i = u^s + 1, u^s + 2, \ldots, u$.

The loop is split into three loops, each associated with a different code version. As a simpler alternative, the number of code versions can be further reduced to two: one with no tests, and one with all tests enabled.

We extend this method to nested loops in Section 5, recursively splitting each iteration space into three regions. When applied to a perfect d-dimensional loop nest, this method leads to 3^d distinct loop nests, each potentially executing a different code version. One can reduce the number of distinct code versions by merging different versions. In the extreme case, it is possible to use only two versions. This method will be referred to as the *general* method.

We next present in Section 6 a method that avoids this exponential blow-up in static code size. This method is referred as the *compact* method. When applied to an perfect d-dimensional loop nest, it results in $2d + 1$ loop nests. Depending on the simplifications adopted, it can use from 2 up to $2d + 1$ different code versions.

Finally we describe in Section 7 a tiling method that can be applied to certain types of loop nests. It effectively implements the *compact* method through generated code of a very simple form. The method uses two versions for each loop body (no tests and all tests enabled). This method is referred to as the *restricted* method.

These four methods we describe apply to Java as well as to Fortran or C, and work for all machine architectures. Additional optimizations are possible in important special cases. We briefly discuss two of these optimizations now: how to determine a valid index with a single test and how to test for **null** pointers. We do not discuss special optimization techniques any further in this paper.

In the particular case of Java an *all tests* can be implemented with a single comparison. Because Java does not distinguish between lower bound and upper bound violations, and because $\text{lo}(A) = 0$ always, it suffices to test if $i < \text{up}(A) + 1$ using unsigned arithmetic. A negative number appears, in unsigned arithmetic, as a very large positive number which, by the Java language specification, is always larger than the largest possible array extent. This technique is used, for example, in the IBM HPCJ project [4].

The optimization of checks for **null** pointer violations in array references is a direct consequence of the optimization of checks for bound violations, as discussed in Section 3. There are also many *ad hoc* solutions to the optimization of **null** pointer violations. For machines with segmented memory architecture, such as the IBM POWER, **null** pointers can be represented by a pointer to a protected segment. For machines with linear, but paged, memory architecture, a protected segment can be simulated with a region of protected pages. Any attempts of access through a **null** pointer will immediately cause a hardware exception that can then be translated into the appropriate software exception. This implementation completely eliminates the need for any explicit checks for **null** pointer violations in the code. It is also applicable to all pointer dereferences, not just array accesses. Despite the complications in properly translating a hardware exception, this technique has been successfully used, for example, in the CACAO project [5].

3 Computing safe bounds

The basic idea of our methods is to partition the iteration space of a loop into *regions*. A region is a contiguous subset of the iteration space. It is characterized by a collection of tests that need to be performed for array references in that subset. We are particularly interested in finding *safe regions*, where we know there can be no violations in array references. If an array reference is guaranteed not to generate a violation, explicit tests are unnecessary.

We illustrate the computation of safe regions with a simple example. We then proceed to formalize it to more general cases. Consider the simple loop:

for $(i = l;\ i \leq u;\ i{+}{+})\ \{$
$\quad B(i)$
$\}$

which we represent as $L(i, l, u, B(i))$. $B(i)$ is the body of the loop and, in general, contains references to the loop index variable i. The iteration space of this loop consists of the range of values $i = l, l + 1, \ldots, u$.

Let there be a single array reference $A[i]$ in the loop body $B(i)$. We denote the lower bound of A by $\text{lo}(A)$ and the upper bound of A by $\text{up}(A)$. Therefore, for the array reference $A[i]$ to be valid we must have $\text{lo}(A) \leq i \leq \text{up}(A)$. If A is **null**, we define $\text{lo}(A) = 0$ and $\text{up}(A) = -1$. This guarantees that $A[i]$ is never valid if A is **null**. We can split the iteration space of the loop into three regions $\mathcal{R}[1]$, $\mathcal{R}[2]$, and $\mathcal{R}[3]$, defined as follows:

$$\begin{aligned}
\mathcal{R}[1] &\ : \ (l \leq i \leq u) \wedge (i < \text{lo}(A)), && (1)\\
\mathcal{R}[2] &\ : \ (l \leq i \leq u) \wedge (\text{lo}(A) \leq i \leq \text{up}(A)), && (2)\\
\mathcal{R}[3] &\ : \ (l \leq i \leq u) \wedge (i > \text{up}(A)). && (3)
\end{aligned}$$

Region $\mathcal{R}[1]$ corresponds to those iterations of the loop for which the index i into array A is too small. Therefore an *lb test* is required before each array reference in this region. No tests are required in region $\mathcal{R}[2]$, since the index i falls within the bounds of A. Finally, a *ub test* is required in region $\mathcal{R}[3]$, because the values of i are too large to index A.

Using Equation (2) we can compute the lower and upper bounds $\mathcal{L}$ and $\mathcal{U}$ of the safe region:

$$\begin{aligned}
\mathcal{L} &= \max(l, \mathrm{lo}(A)), && (4)\\
\mathcal{U} &= \min(u, \mathrm{up}(A)), \text{ and} && (5)\\
\mathcal{R}[2] &: i = \mathcal{L}, \ldots, \mathcal{U}. && (6)
\end{aligned}$$

Similarly, we use Equation (1) and Equation (3) to compute the lower and upper bounds of regions $\mathcal{R}[1]$ and $\mathcal{R}[3]$, respectively:

$$\begin{aligned}
\mathcal{R}[1] &: i = l, \ldots, \min(u+1, \mathrm{lo}(A)) - 1, \text{ and} && (7)\\
\mathcal{R}[3] &: i = \max(l-1, \mathrm{up}(A)) + 1, \ldots, u. && (8)
\end{aligned}$$

Note that $\min(u+1, \mathrm{lo}(A)) = \mathcal{L}$, except when $\mathrm{lo}(A) < l$ or $\mathrm{lo}(A) > u+1$. In the former case $\mathcal{R}[1]$ is empty; in the latter case $\mathcal{R}[2]$ is empty. To handle these cases, and the symmetric upper bound cases, we redefine

$$\begin{aligned}
\mathcal{L} &= \min(u+1, \max(l, \mathrm{lo}(A))), && (9)\\
\mathcal{U} &= \max(l-1, \min(u, \mathrm{up}(A))). && (10)
\end{aligned}$$

We can now express the bounds of each of the regions just in terms of l, u, $\mathcal{L}$, and $\mathcal{U}$:

$$\begin{aligned}
\mathcal{R}[1] &: i = l, \ldots, \mathcal{L} - 1, && (11)\\
\mathcal{R}[2] &: i = \mathcal{L}, \ldots, \mathcal{U}, \text{ and} && (12)\\
\mathcal{R}[3] &: i = \mathcal{U} + 1, \ldots, u. && (13)
\end{aligned}$$

Equations (11)-(13) define the same regions as Equations (6)-(8). The values of regions bounds are different only for empty regions, but they are still empty.

We illustrate in Figure 2 the values of $\mathcal{L}$ and $\mathcal{U}$ for different relative positions of iteration bounds and array bounds. Figure 2(A) has empty regions $\mathcal{R}[1]$ and $\mathcal{R}[3]$, while region $\mathcal{R}[2]$ comprises the entire iteration space. Region $\mathcal{R}[1]$ is empty in Figure 2(B), while region $\mathcal{R}[3]$ is empty in Figure 2(C). All three regions are nonempty in Figure 2(D). Regions $\mathcal{R}[2]$ and $\mathcal{R}[3]$ are empty in Figure 2(E), because all values of i fall below the lower bound of A. Finally, regions $\mathcal{R}[1]$ and $\mathcal{R}[2]$ are empty in Figure 2(F), since all values of i fall above the upper bound of A.

In the general case, we have an array reference of the form $A[f(i)]$ in the body $B(i)$ of a loop. Depending on the behavior of $f(i)$, we can compute safe bounds $\mathcal{L}$ and $\mathcal{U}$ that partition the iteration space into three regions, as defined by Equations (11)-(13). Region $\mathcal{R}[2]$ is safe, and no test is defined in it. We define tests $\tau[1]$ and $\tau[3]$ that are sufficient for regions $\mathcal{R}[1]$ and $\mathcal{R}[3]$ respectively. Expressions for $\mathcal{L}$, $\mathcal{U}$, and τ for various forms of $f(i)$ are described in Appendix A. The safe bounds are computed so that we always have $\mathcal{U} \geq \mathcal{L} - 1$ and $\mathcal{L} \leq \mathcal{U} + 1$. These properties are important for the correct functioning of our methods.

4 An exact method

We consider in this section the case of a simple (depth one) loop, with multiple references. The *exact* method described here performs only the tests strictly necessary (as specified in Appendix A) to detect the array reference violations that occur during the execution of a loop. It is, in general, of limited practicality because up to 3^ρ versions of the loop body must be instantiated in the code, where ρ is the number of array references in the loop body. In some situations, when enough information is available to the compiler, it is possible

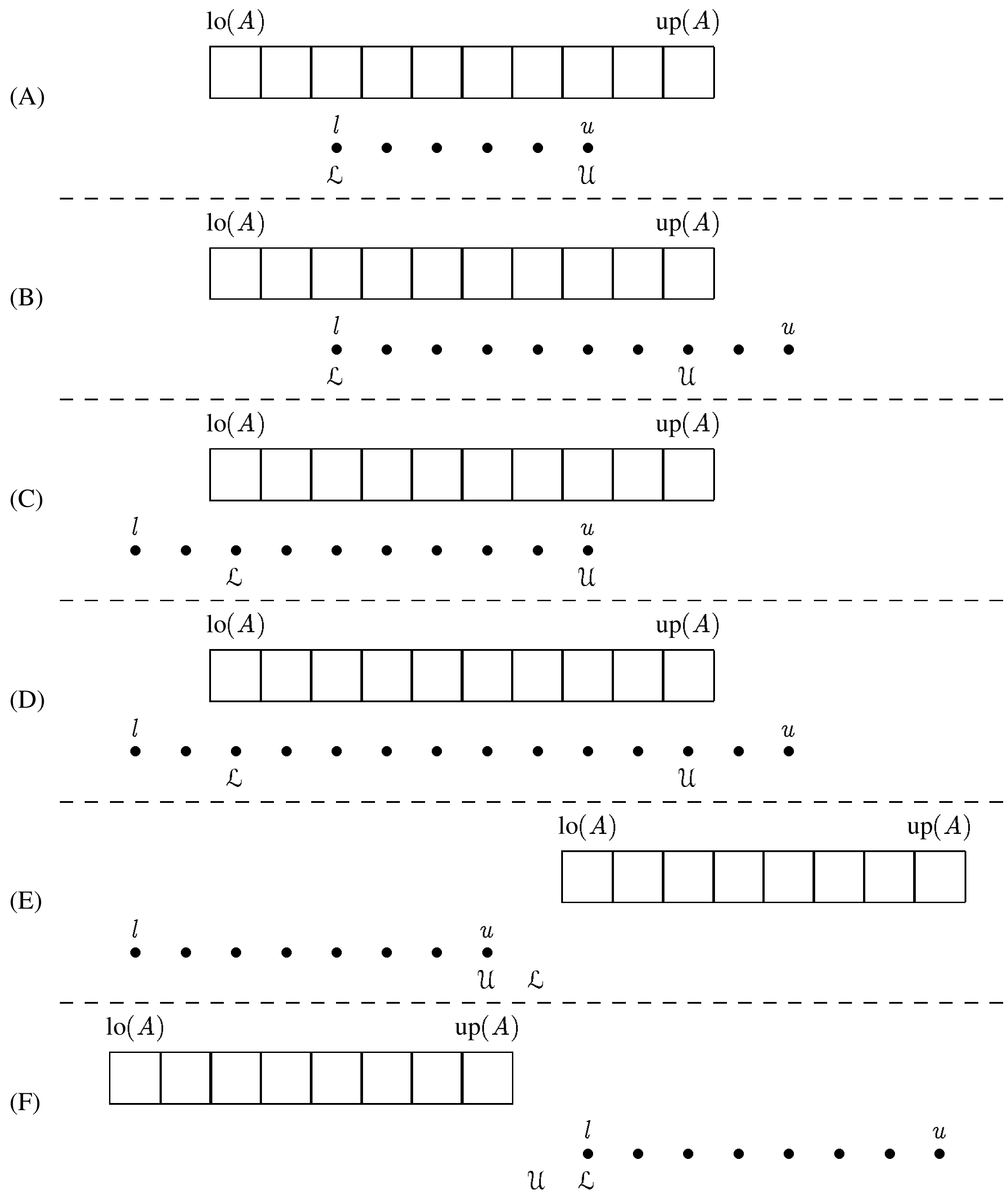


Figure 2: Relationship between loop bounds and array bounds.

to generate efficient code using this technique. Our main interest in studying this method is that the other, more practical, transformations are derived from this technique.

In this section we treat references of the form $A[f(i)]$ that allow the computation of safe bounds as described in Section 3. Section 8 discusses how to handle more complex subscripts. We first describe how the method generates optimized code, and then we illustrate the method with an example.

4.1 Code generation

The method works as follows. Let $L(i, l, u, B(i))$ be a loop on index variable i and body $B(i)$. Let there be ρ references of the form $A_j[f_j(i)]$, $j = 1, \ldots, \rho$, in body $B(i)$. Each array reference $A_j[f_j(i)]$ partitions the iteration space into three (each possibly empty) regions:

$$\begin{aligned} \mathcal{R}^j[1] &: \quad i = l, \ldots, \mathcal{L}^j - 1, \\ \mathcal{R}^j[2] &: \quad i = \mathcal{L}^j, \ldots, \mathcal{U}^j, \\ \mathcal{R}^j[3] &: \quad i = \mathcal{U}^j + 1, \ldots, u, \end{aligned}$$

as defined by Equations (11)-(13). We call these regions defined by a single array reference *simple regions*. Also, each region $\mathcal{R}^j[k]$ has a test $\tau^j[k]$ associated with it, which describes what kind of test has to be performed for reference $A_j[f_j(i)]$ in that region. The test $\tau^j[2]$ for region $\mathcal{R}^j[2]$ always specifies no test, because $\mathcal{R}^j[2]$ is a safe region with respect to this reference. The tests $\tau^j[1]$ and $\tau^j[3]$, for regions $\mathcal{R}^j[1]$ and $\mathcal{R}^j[3]$ respectively, can be any one of *ub test* (an upper bound test), *lb test* (a lower bound test), or *all tests* (both a lower and an upper bound tests). The exact choice depends on characteristics of the array reference $A_j[f_j(i)]$. This issue is discussed in detail in Appendix A. For each region $\mathcal{R}^j[k]$ we denote its lower bound by $\mathcal{R}^j[k].l$ and its upper bound by $\mathcal{R}^j[k].u$.

Two references $A_j[f_j(i)]$ and $A_k[f_k(i)]$ can be thought of as partitioning the iteration space into 5 regions defined by the 4 points $\mathcal{L}^j, \mathcal{U}^j, \mathcal{L}^k$, and $\mathcal{U}^k$. We illustrate this in Figure 3 for two references $A_1[f_1(i)]$ and $A_2[f_2(i)]$. Note that some of the resulting regions may be empty, in general. The resulting regions are a subset of the $3 \cdot 3 = 9$ regions formed by all combinations of intersections of two simple regions, one from each array reference.

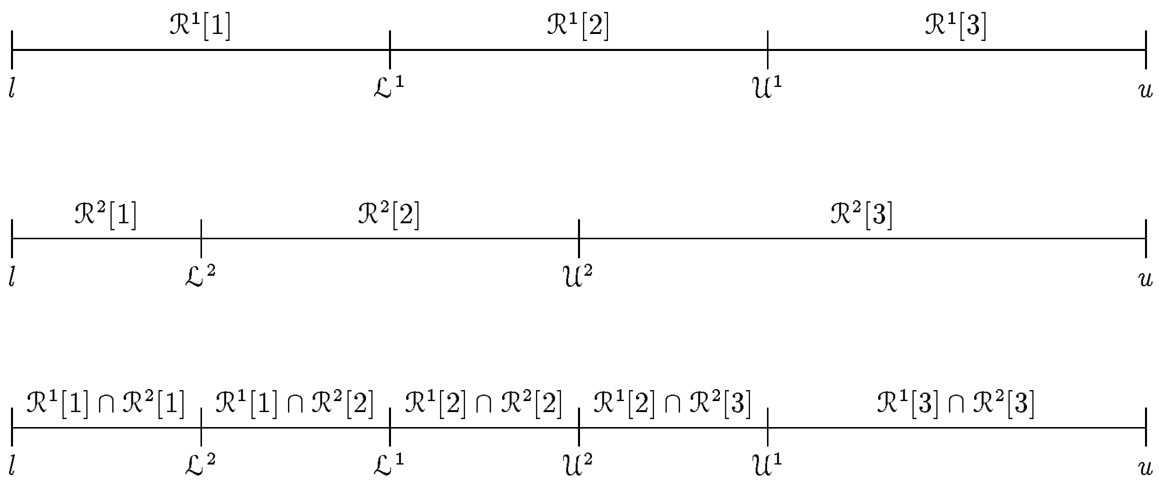


Figure 3: Regions generated by the intersection of simple regions.

In general, given the ρ references $A_j[f_j(i)]$ we can create a vector of all 3^ρ regions formed by intersecting the simple regions from different array references:

$$\mathcal{R}[x_1 \cdot x_2 \cdot \ldots \cdot x_\rho] = \mathcal{R}^1[x_1] \cap \mathcal{R}^2[x_2] \cap \ldots \cap \mathcal{R}^\rho[x_\rho]. \tag{14}$$

Each index x_k is 1, 2, or 3. The lower bound of a region $\mathcal{R}[x_1 \cdot x_2 \cdot \ldots \cdot x_\rho]$ is the maximum of the lower bounds of the forming simple regions. Correspondingly, its upper bound is the minimum of the upper bounds of the forming simple regions:

$$\begin{aligned} \mathcal{R}[x_1 \cdot x_2 \cdot \ldots \cdot x_\rho].l &= \max(\mathcal{R}^1[x_1].l, \mathcal{R}^2[x_2].l, \ldots, \mathcal{R}^\rho[x_\rho].l), && (15)\\ \mathcal{R}[x_1 \cdot x_2 \cdot \ldots \cdot x_\rho].u &= \min(\mathcal{R}^1[x_1].u, \mathcal{R}^2[x_2].u, \ldots, \mathcal{R}^\rho[x_\rho].u). && (16) \end{aligned}$$

Finally, the tests that characterize region $\mathcal{R}[x_1 \cdot x_2 \cdot \ldots \cdot x_\rho]$ can be described by:

$$\tau[x_1 \cdot x_2 \cdot \ldots \cdot x_\rho] = \{\tau^j[x_j],\ j = 1, \ldots, \rho\}. \tag{17}$$

That is, the combination of the tests for each simple region $\mathcal{R}^j[x_j]$.

Using this partitioning of the iteration space into 3^ρ regions, the loop

$$\begin{array}{l} \textbf{for}\ (i = l;\ i \le u;\ i\text{++})\ \{ \\ \quad B(i) \\ \} \end{array} \tag{18}$$

can be transformed into 3^ρ loops, each one implementing one of the regions $\mathcal{R}[x_1 \cdot x_2 \cdot \ldots \cdot x_\rho]$. The body of each region can be specialized to perform exactly the tests described by $\tau[x_1 \cdot x_2 \cdot \ldots \cdot x_\rho]$. We use $B_{x_1 x_2 \ldots x_\rho}(i)$ to denote the body $B(i)$ specialized with the tests described by $\tau[x_1 \cdot x_2 \cdot \ldots \cdot x_\rho]$:

$$\boxed{\begin{array}{l} \textbf{for}\ (i = \mathcal{R}[1 \cdot 1 \cdot \ldots \cdot 1].l;\ i \le \mathcal{R}[1 \cdot 1 \cdot \ldots \cdot 1].u;\ i\text{++})\ \{B_{11\ldots1}(i)\ \} \\ \textbf{for}\ (i = \mathcal{R}[1 \cdot 1 \cdot \ldots \cdot 2].l;\ i \le \mathcal{R}[1 \cdot 1 \cdot \ldots \cdot 2].u;\ i\text{++})\ \{B_{11\ldots2}(i)\ \} \\ \textbf{for}\ (i = \mathcal{R}[1 \cdot 1 \cdot \ldots \cdot 3].l;\ i \le \mathcal{R}[1 \cdot 1 \cdot \ldots \cdot 3].u;\ i\text{++})\ \{B_{11\ldots3}(i)\ \} \\ \hfil\vdots\hfil \\ \textbf{for}\ (i = \mathcal{R}[1 \cdot 2 \cdot \ldots \cdot 1].l;\ i \le \mathcal{R}[1 \cdot 2 \cdot \ldots \cdot 1].u;\ i\text{++})\ \{B_{12\ldots1}(i)\ \} \\ \textbf{for}\ (i = \mathcal{R}[1 \cdot 2 \cdot \ldots \cdot 2].l;\ i \le \mathcal{R}[1 \cdot 2 \cdot \ldots \cdot 2].u;\ i\text{++})\ \{B_{12\ldots2}(i)\ \} \\ \textbf{for}\ (i = \mathcal{R}[1 \cdot 2 \cdot \ldots \cdot 3].l;\ i \le \mathcal{R}[1 \cdot 2 \cdot \ldots \cdot 3].u;\ i\text{++})\ \{B_{12\ldots3}(i)\ \} \\ \hfil\vdots\hfil \\ \textbf{for}\ (i = \mathcal{R}[3 \cdot 3 \cdot \ldots \cdot 1].l;\ i \le \mathcal{R}[3 \cdot 3 \cdot \ldots \cdot 1].u;\ i\text{++})\ \{B_{33\ldots1}(i)\ \} \\ \textbf{for}\ (i = \mathcal{R}[3 \cdot 3 \cdot \ldots \cdot 2].l;\ i \le \mathcal{R}[3 \cdot 3 \cdot \ldots \cdot 2].u;\ i\text{++})\ \{B_{33\ldots2}(i)\ \} \\ \textbf{for}\ (i = \mathcal{R}[3 \cdot 3 \cdot \ldots \cdot 3].l;\ i \le \mathcal{R}[3 \cdot 3 \cdot \ldots \cdot 3].u;\ i\text{++})\ \{B_{33\ldots3}(i)\ \} \end{array}} \tag{19}$$

Note that the order of the resulting loops is important, although there are many correct orders. The requirement is that, for any value of j, region $\mathcal{R}[x_1 \cdot \ldots \cdot x_{j-1} \cdot 1 \cdot x_{j+1} \cdot \ldots \cdot x_\rho]$ has to precede $\mathcal{R}[x_1 \cdot \ldots \cdot x_{j-1} \cdot 2 \cdot x_{j+1} \cdot \ldots \cdot x_\rho]$ which in turn has to precede $\mathcal{R}[x_1 \cdot \ldots \cdot x_{j-1} \cdot 3 \cdot x_{j+1} \cdot \ldots \cdot x_\rho]$.

Out of the 3^ρ possible regions formed by the intersection of the simple regions, no more than $2\rho + 1$ are nonempty and are actually executed at run-time. Which of the 3^ρ regions are nonempty depends on the relative positions of the 2ρ safe bounds $\mathcal{L}^1, \ldots, \mathcal{L}^\rho, \mathcal{U}^1, \ldots, \mathcal{U}^\rho$. Let $p_1, \ldots, p_{2\rho}$ be the the list obtained by sorting $\mathcal{L}^1, \ldots, \mathcal{L}^\rho, \mathcal{U}^1 + 1, \ldots, \mathcal{U}^\rho + 1$ in ascending order. Define $p_0 = l$ and $p_{2\rho+1} = u + 1$. The safe bounds $\mathcal{L}^1, \ldots, \mathcal{L}^\rho, \mathcal{U}^1, \ldots, \mathcal{U}^\rho$ partition the iteration space into $2\rho + 1$ (each possibly empty) regions $\mathcal{P}[k]$, $k = 0, 1, \ldots, 2\rho$ defined by

$$\mathcal{P}[k] \quad : \quad i = p_k, \ldots, p_{k+1} - 1. \tag{20}$$

Each region $\mathcal{P}[k]$ corresponds to a region $\mathcal{R}[x_1 \cdot x_2 \cdot \ldots \cdot x_\rho]$ defined by

$$x_j = \begin{cases} 1 & \text{if } \mathcal{L}^j \geq p_{k+1}, \\ 3 & \text{if } \mathcal{U}^j < p_k, \\ 2 & \text{if } (\mathcal{L}^j < p_{k+1}) \wedge (\mathcal{U}^j \geq p_k). \end{cases} \tag{21}$$

(It can occur that both $\mathcal{L}^j \geq p_{k+1}$ and $\mathcal{U}^j < p_k$. In that case, region $\mathcal{P}[k]$ is necessarily empty and the choice of x_j is irrelevant.) Let $M(k)$ be the function that defines the correspondence $\mathcal{R}[M(k)] \equiv \mathcal{P}[k]$ for $k = 0, \ldots, 2\rho$. Then the loop

$$\begin{array}{l} \textbf{for } (i = l;\ i \leq u;\ i{+}{+})\ \{ \\ \quad B(i) \\ \} \end{array} \tag{22}$$

can be transformed to

$$\begin{array}{l} \textbf{for } (k = 0;\ k \leq 2\rho;\ k{+}{+})\ \{ \\ \quad \textbf{for } (i = p_k;\ i < p_{k+1};\ i{+}{+})\ \{ \\ \quad\quad B_{M(k)}(i) \\ \} \end{array} \tag{23}$$

If the order of the safe bounds $\mathcal{L}^1, \ldots, \mathcal{L}^\rho, \mathcal{U}^1, \ldots, \mathcal{U}^\rho$ is known at compile time, then the mapping function $M(k)$ can also be determined. In this case, only the loop body versions that are actually executed need to be generated. In general, the order of the safe bounds is not known, and $B_{M(k)}(i)$ has to be selected at run-time from the set of all possible 3^ρ body versions.

4.2 An example

In the interest of conciseness, we use a very simple code fragment to illustrate this method. Figure 4(A) illustrates the original loop to be transformed. It has two array references, $A_1[i-2]$ and $A_2[i+2]$, each generating three simple regions: $\mathcal{R}^1[1:3]$ and $\mathcal{R}^2[1:3]$, respectively. The straightforwardly transformed code, using the exact method, is shown in Figure 4(B). We note that:

$$\mathcal{L}^1 = \min(u+1, \max(l, 3)), \tag{24}$$
$$\mathcal{L}^2 = \min(u+1, \max(l, -1)), \tag{25}$$
$$\mathcal{U}^1 = \max(l-1, \min(u, n+2)), \tag{26}$$
$$\mathcal{U}^2 = \max(l-1, \min(u, n-2)), \tag{27}$$
$$\tau^1[1] = \tau^2[1] = \textit{lb test}, \tag{28}$$
$$\tau^1[3] = \tau^2[3] = \textit{ub test}, \tag{29}$$

which does not provide us with enough information to order the safe bounds $\mathcal{L}^1$, $\mathcal{L}^2$, $\mathcal{U}^1$, and $\mathcal{U}^2$. We implement the code in Figure 4(B) according to Equation (19). We could have used the method of Equation (23), but it would still have required all 3^ρ body versions to be generated.

Now let $n = 10$, as shown in Figure 5(A). In this case, the expressions for the safe bounds become:

$$\mathcal{L}^1 = \min(u+1, \max(l, 3)) \tag{30}$$
$$\mathcal{L}^2 = \min(u+1, \max(l, -1)) \tag{31}$$
$$\mathcal{U}^1 = \max(l-1, \min(u, 12)) \tag{32}$$
$$\mathcal{U}^2 = \max(l-1, \min(u, 8)) \tag{33}$$

```
double A_1[] = new double[1 : n]
double A_2[] = new double[1 : n]

for (i = l; i ≤ u; i++) {
   A_2[i + 2] = A_1[i − 2]    // B(i)
}
```

(A)

```
for (i = l; i ≤ min(ℒ^1 − 1, ℒ^2 − 1); i++) {
   B_11(i)  // A_1, A_2 lb check
}
for (i = max(l, ℒ^2); i ≤ min(ℒ^1 − 1, 𝒰^2); i++) {
   B_12(i)  // A_1 lb check, no A_2 check
}
for (i = max(l, 𝒰^2 + 1); i ≤ min(ℒ^1 − 1, u); i++) {
   B_13(i)  // A_1 lb check, A_2 ub check
}
for (i = max(ℒ^1, l); i ≤ min(𝒰^1, ℒ^2 − 1); i++) {
   B_21(i)  // no A_1 check, A_2 lb check
}
for (i = max(ℒ^1, ℒ^2); i ≤ min(𝒰^1, 𝒰^2); i++) {
   B_22(i)  // no A_1, A_2 check
}
for (i = max(ℒ^1, 𝒰^2 + 1); i ≤ min(𝒰^1, u); i++) {
   B_23(i)  // no A_1 check, A_2 ub check
}
for (i = max(𝒰^1 + 1, l); i ≤ min(u, ℒ^2 − 1); i++) {
   B_31(i)  // A_1 ub check, A_2 lb check
}
for (i = max(𝒰^1 + 1, ℒ^2); i ≤ min(u, 𝒰^2); i++) {
   B_32(i)  // A_1 ub check, no A_2 check
}
for (i = max(𝒰^1 + 1, 𝒰^2 + 1); i ≤ u; i++) {
   B_33(i)  // A_1, A_2 ub check
}
```

(B)

Figure 4: A program optimized using the exact method: (A) original code, (B) transformed code.

and we can order $\mathcal{L}^2 \leq \mathcal{L}^1 \leq \mathcal{U}^2 \leq \mathcal{U}^1$. In particular, this is the ordering shown in Figure 3. We only need to generate code for five loops, as shown in Figure 5(B). In some cases, a compiler with appropriate symbolic analysis can even eliminate some of these five loops. For example, if the compiler could prove that $l > 2$, then neither of the first two loops (body versions $B_{11}(i)$ and $B_{12}(i)$) would be necessary.

4.3 Summary of the exact method

The exact method transforms code so that, for each iteration of the original loop, only those tests that cannot be shown to be unnecessary are performed. In the straightforward application of the method to a loop with ρ array references in its body, 3^ρ new loops are generated, each with a slightly different body. No more than $2\rho + 1$ of these loops are actually executed at run-time. In some situations, compile-time analysis can show that some of these loops implement empty regions of the iteration space, and therefore can be discarded.

```
double A_1[] = new double[1 : 10]
double A_2[] = new double[1 : 10]

for (i = l; i ≤ u; i++) {
   A_2[i + 2] = A_1[i − 2]    // B(i)
}
```

(A)

```
for (i = l; i < ℒ^2; i++) {
   B_11(i)  // A_1, A_2 lb check
}
for (i = ℒ^2; i < ℒ^1; i++) {
   B_12(i)  // A_1 lb check, no A_2 check
}
for (i = ℒ^1; i < 𝒰^2 + 1; i++) {
   B_22(i)  // no A_1, A_2 check
}
for (i = 𝒰^2 + 1; i < 𝒰^1 + 1; i++) {
   B_23(i)  // no A_1 check, A_2 ub check
}
for (i = 𝒰^1 + 1; i < u; i++) {
   B_33(i)  // A_1, A_2 ub check
}
```

(B)

Figure 5: A program optimized using the exact method: (A) original code, (B) transformed code.

5 The general method

The general method works by partitioning the iteration space of a loop always into three regions, independent of the number of array references in the body of the loop. One of the regions is a safe region: no array reference in that region can cause a violation. Another region precedes the safe region, in iteration order. Finally, the third region succeeds the safe region, in iteration order. This method can be applied to each and every loop of an arbitrary loop nest individually. The general method does not specialize the tests as much as the exact method, but it does identify the same safe regions that can be executed without any tests.

5.1 Transforming a single loop

Consider the loop $L(i, l, u, B(i))$, with ρ references of the form $A_j[f_j(i)]$ in its body. Using the concepts developed in Sections 3 and 4, we can compute its safe region as the intersection of all simple safe regions. This safe region is defined by the range of values of $i = l^s, l^s + 1, \ldots, u^s$ where:

$$l^s = \max(\mathcal{L}^1, \mathcal{L}^2, \ldots, \mathcal{L}^\rho), \tag{34}$$

$$u^s = \max(l^s - 1, \min(\mathcal{U}^1, \mathcal{U}^2, \ldots, \mathcal{U}^\rho)). \tag{35}$$

The $l^s - 1$ term in the expression for u^s is necessary to handle cases where the safe region is empty. We can then partition the iteration space of the loop into three (each possibly empty) regions:

$$\mathcal{R}[1] \ : \ i = l, \ldots, l^s - 1, \tag{36}$$

$$\mathcal{R}[2] \ : \ i = l^s, \ldots, u^s, \tag{37}$$

$$\mathcal{R}[3] \ : \ i = u^s + 1, \ldots, u. \tag{38}$$

This partitioning is not very different from our very first example in Section 3, except that now it applies to a loop with an arbitrary number of array references in its body. We illustrate this for two references

$A_1[f_1(i)]$ and $A_2[f_2(i)]$ in Figure 6. Region $\mathcal{R}[2]$ is the intersection of the safe simple regions $\mathcal{R}^1[2]$ and $\mathcal{R}^2[2]$. Correspondingly, region $\mathcal{R}[1]$ is the union of the unsafe simple regions $\mathcal{R}^1[1]$ and $\mathcal{R}^2[1]$. Region $\mathcal{R}[3]$ is the union of the unsafe simple regions $\mathcal{R}^1[3]$ and $\mathcal{R}^2[3]$. With reference to Figure 3, $\mathcal{R}[1]$ is formed by merging all regions preceding $\mathcal{R}[2 \cdot 2]$ and region $\mathcal{R}[3]$ is formed by merging all regions succeeding region $\mathcal{R}[2 \cdot 2]$.

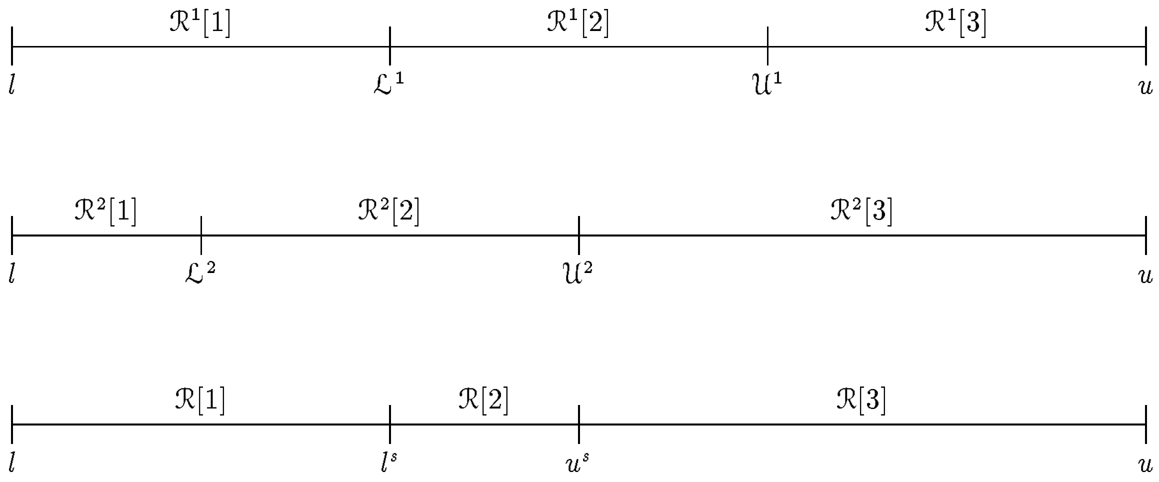


Figure 6: Partitioning of an iteration space into three regions.

Region $\mathcal{R}[2]$ is the safe region, and its body can be implemented without any tests. We denote this version of the loop body by $B(\textbf{notest}(i))$. Conversely, the bodies of regions $\mathcal{R}[1]$ and $\mathcal{R}[3]$ need at least some tests. For simplicity, we can implement both with a version $B(\textbf{test}(i))$ of body $B(i)$. This version performs an *all tests* test before each and every array reference.

The implementation of the general method consists of the transformation:

$$\boxed{\begin{array}{l} \textbf{for}\ (i = l;\ i \leq u;\ i{+}{+})\ \{ \\ \quad B(i) \\ \} \end{array}} \Longrightarrow \boxed{\begin{array}{l} \textbf{for}\ (i = l;\ i \leq l^s - 1;\ i{+}{+})\ \{ \\ \quad B(\textbf{test}(i)) \\ \} \\ \textbf{for}\ (i = l^s;\ i \leq u^s;\ i{+}{+})\ \{ \\ \quad B(\textbf{notest}(i)) \\ \} \\ \textbf{for}\ (i = u^s + 1;\ i \leq u;\ i{+}{+})\ \{ \\ \quad B(\textbf{test}(i)) \\ \} \end{array}} \tag{39}$$

or, in shorthand:

$$L(i, l, u, B(i)) \Longrightarrow \boxed{\begin{array}{l} L_1(i, l, l^s - 1, B(\textbf{test}(i))) \\ L_2(i, l^s, u^s, B(\textbf{notest}(i))) \\ L_3(i, u^s + 1, u, B(\textbf{test}(i))) \end{array}}\,. \tag{40}$$

The code expansion in this case is only two-fold, since the same code for $B(\textbf{test}(i))$ can be used twice. Methods to realize all three resulting loops using only two instances of $B(i)$ are discussed in Appendix B.

5.2 Recursive application of the transformation

If the body B of a loop contains other loops, then the same transformation can be applied to each of the loops in B. The transformation can be applied individually to each and every loop in a general loop nest. In fact, the final result is independent of the order in which the individual loops are transformed.

Consider the case of the two-dimensional loop nest $L(i, l_i, u_i, L'(j, l_j, u_j, B(i, j)))$. By first applying the transformation to the L loop, we generate a region without any tests on array references indexed by i:

$$L(i, l_i, u_i, L'(j, l_j, u_j, B(i, j))) \Longrightarrow \boxed{\begin{array}{l} L_1(i, l_i, l_i^s - 1, L'(j, l_j, u_j, B(\textbf{test}(i), j))) \\ L_2(i, l_i^s, u_i^s, L'(j, l_j, u_j, B(\textbf{notest}(i), j))) \\ L_3(i, u_i^s + 1, u_i, L'(j, l_j, u_j, B(\textbf{test}(i), j))) \end{array}}.$$

We can now apply the transformation to each of the three L' loops. This will generate two regions without any tests on array references indexed by j and one region without any tests on array references indexed by either i or j. The complete transformation is shown in Figure 7. This transformation partitions the iteration space into nine regions, and requires four different versions of the loop body $B(i)$.

$$L(i, l_i, u_i, L'(j, l_j, u_j, B(i, j))) \Longrightarrow \boxed{\begin{array}{l} L_1(i, l_i, l_i^s - 1, \boxed{\begin{array}{l} L_1'(j, l_j, l_j^s - 1, B(\textbf{test}(i), \textbf{test}(j))) \\ L_2'(j, l_j^s, u_j^s, B(\textbf{test}(i), \textbf{notest}(j))) \\ L_3'(j, u_j^s + 1, u_j, B(\textbf{test}(i), \textbf{test}(j))) \end{array}}) \\ L_2(i, l_i^s, u_i^s, \boxed{\begin{array}{l} L_1'(j, l_j, l_j^s - 1, B(\textbf{notest}(i), \textbf{test}(j))) \\ L_2'(j, l_j^s, u_j^s, B(\textbf{notest}(i), \textbf{notest}(j))) \\ L_3'(j, u_j^s + 1, u_j, B(\textbf{notest}(i), \textbf{test}(j))) \end{array}}) \\ L_3(i, u_i^s + 1, u_i, \boxed{\begin{array}{l} L_1'(j, l_j, l_j^s - 1, B(\textbf{test}(i), \textbf{test}(j))) \\ L_2'(j, l_j^s, u_j^s, B(\textbf{test}(i), \textbf{notest}(j))) \\ L_3'(j, u_j^s + 1, u_j, B(\textbf{test}(i), \textbf{test}(j))) \end{array}}) \end{array}}$$

Figure 7: Applying the general method to a loop nest.

5.3 An example of the general method

For simplicity, we give an example of single threaded code with rectangular arrays. Section 9 will discuss the application of these optimizations to multithreaded code. The program of Figure 8 will be used in this example. It implements a step of a two-dimensional *Jacobi relaxation* [6]. We chose it because it is a well-known operation that illustrates a loop nest with various array references in its body. Also, the array references all use slightly different indices, making our example more interesting.

Figure 9 shows the loop implemented with naive tests. The statement

$$\text{ifAccessViolation}(A, i)\ \textbf{throwException}$$

throws the appropriate exception if A is a **null** pointer, i is below the lower bound of A, or i is above the upper bound of A. In the actual executable code, the tests would be interweaved with the individual array

```
double a[][] = new double[l_1 : u_1][l_2 : u_2]
double b[][] = new double[l_1 : u_1][l_2 : u_2]
for (i = l_i; i ≤ u_i; i++) {
   for (j = l_j; j ≤ u_j; j++) {
      b[i][j] = (a[i][j + 1] + a[i][j − 1] + a[i + 1][j] + a[i − 1][j])/4
   }
}
```

Figure 8: Program fragment to be optimized.

references, and the order of tests would be slightly different, but this example gives an idea of the cost of naive testing. Figure 10 shows the loop nest optimized for testing using the general method. In each of the resulting loops, we list the violations that can occur in its body. The code with explicit tests for the different versions of the loop body are shown in Figure 11. The minimal tests needed for the i_1 and i_3 loops differ in that the i_1 loop only needs to test for lower bounds violations, and the i_3 loop only needs to test for upper bounds violations. Using the transformation of Equation (39), the version for both loops can be the same, as indicated in Figure 10.

```
double a[][] = new double[l_1 : u_1][l_2 : u_2]
double b[][] = new double[l_1 : u_1][l_2 : u_2]

for (i = l_i; i ≤ u_i; i++) {
   for (j = l_j; j ≤ u_j; j++) {
      ifAccessViolation(a, i) throwException
      ifAccessViolation(a[i], j + 1) throwException
      ifAccessViolation(a[i], j − 1) throwException
      ifAccessViolation(a, i + 1) throwException
      ifAccessViolation(a[i + 1], j) throwException
      ifAccessViolation(a, i − 1) throwException
      ifAccessViolation(a[i − 1], j) throwException
      ifAccessViolation(b, i) throwException
      ifAccessViolation(b[i], j) throwException
      b[i][j] = (a[i][j + 1] + a[i][j − 1] + a[i + 1][j] + a[i − 1][j])/4
   }
}
```

Figure 9: A program with explicit violation tests.

double $a[][] = \textbf{new}$ double$[l_1 : u_1][l_2 : u_2]$
double $b[][] = \textbf{new}$ double$[l_1 : u_1][l_2 : u_2]$

for $(i_1 = l_i;\ i_1 \leq l_i^s - 1;\ i_1\text{++})\ \{$ *// i lower bound violations*
  for $(j_{1,1} = l_j;\ j_{1,1} \leq l_j^s - 1;\ j_{1,1}\text{++})\ \{$ *// j lower bound violations*
    $B(\textbf{test}(i_1), \textbf{test}(j_{1,1}))$
  $\}$
  for $(j_{1,2} = l_j^s;\ j_{1,2} \leq u_j^s;\ j_{1,2}\text{++})\ \{$ *// no j violations*
    $B(\textbf{test}(i_1), \textbf{notest}(j_{1,2}))$
  $\}$
  for $(j_{1,3} = u_j^s + 1;\ j_{1,3} \leq u_j;\ j_{1,3}\text{++})\ \{$ *// j upper bound violations*
    $B(\textbf{test}(i_1), \textbf{test}(j_{1,3}))$
  $\}$
$\}$
for $(i_2 = l_i^s;\ i_2 \leq u_i^s;\ i_2\text{++})\ \{$ *// no i violations*
  for $(j_{2,1} = l_j;\ j_{2,1} \leq l_j^s - 1;\ j_{2,1}\text{++})\ \{$ *// j lower bound violations*
    $B(\textbf{notest}(i_2), \textbf{test}(j_{2,1}))$
  $\}$
  for $(j_{2,2} = l_j^s;\ j_{2,2} \leq u_j^s;\ j_{2,2}\text{++})$ *// no j violations*
    $B(\textbf{notest}(i_2), \textbf{notest}(j_{2,2}))$
  $\}$
  for $(j_{2,3} = u_j^s + 1;\ j_{2,3} \leq u_j;\ j_{2,3}\text{++})$ *// j upper bound violations*
    $B(\textbf{notest}(i_2), \textbf{test}(j_{2,3}))$
  $\}$
$\}$
for $(i_3 = u_i^s + 1;\ i_3 \leq u_i, i_3\text{++})\ \{$ *// i upper bound violations*
  for $(j_{3,1} = l_j;\ j_{3,1} \leq l_j^s - 1;\ j_{3,1}\text{++})\ \{$ *// j lower bound violations*
    $B(\textbf{test}(i_3), \textbf{test}(j_{3,1}))$
  $\}$
  for $(j_{3,2} = l_j^s;\ j_{3,2} \leq u_j^s;\ j_{3,2}\text{++})$ *// no j violations*
    $B(\textbf{test}(i_3), \textbf{notest}(j_{3,2}))$
  $\}$
  for $(j_{3,3} = u_j^s + 1;\ j_{3,3} \leq u_j;\ j_{3,3}\text{++})$ *// j upper bound violations*
    $B(\textbf{test}(i_3), \textbf{test}(j_{3,3}))$
  $\}$
$\}$

Figure 10: An example of the general method.

5.3.1 Transforming the loops

The optimized code is the result of the following transformations. First, the outer i loop is split into three loops (i_1, i_2, and i_3), as shown in Figure 10. The middle loop, i_2, corresponds to those iterations of the original i loop that cannot cause bound violations on array references indexed by i. Within this loop, these array references need not be tested. This is the *safe region* for loop i.

$B(\textbf{test}(i), \textbf{test}(j))$:	ifAccessViolation(a, i) **throwException**
	ifAccessViolation$(a[i], j+1)$ **throwException**
	ifAccessViolation$(a[i], j-1)$ **throwException**
	ifAccessViolation$(a, i+1)$ **throwException**
	ifAccessViolation$(a[i+1], j)$ **throwException**
	ifAccessViolation$(a, i-1)$ **throwException**
	ifAccessViolation$(a[i-1], j)$ **throwException**
	ifAccessViolation(b, i) **throwException**
	ifAccessViolation$(b[i], j)$ **throwException**
	$b[i][j] = (a[i][j+1] + a[i][j-1] + a[i+1][j] + a[i-1][j])/4$
$B(\textbf{test}(i), \textbf{notest}(j))$:	ifAccessViolation(a, i) **throwException**
	ifAccessViolation$(a, i+1)$ **throwException**
	ifAccessViolation$(a, i-1)$ **throwException**
	ifAccessViolation(b, i) **throwException**
	$b[i][j] = (a[i][j+1] + a[i][j-1] + a[i+1][j] + a[i-1][j])/4$
$B(\textbf{notest}(i), \textbf{test}(j))$:	ifAccessViolation$(a[i], j+1)$ **throwException**
	ifAccessViolation$(a[i], j-1)$ **throwException**
	ifAccessViolation$(a[i+1], j)$ **throwException**
	ifAccessViolation$(a[i-1], j)$ **throwException**
	ifAccessViolation$(b[i], j)$ **throwException**
	$b[i][j] = (a[i][j+1] + a[i][j-1] + a[i+1][j] + a[i-1][j])/4$
$B(\textbf{notest}(i), \textbf{notest}(j))$:	$b[i][j] = (a[i][j+1] + a[i][j-1] + a[i+1][j] + a[i-1][j])/4$

Figure 11: The four different body versions used in Figure 10.

The first loop, i_1, executes those iterations of the original i loop whose values of i precede the safe region. Within this loop, all array references indexed by i need to be tested. Because the subscript expressions on i are monotonically increasing across the iteration space of the i loop, only lower bound violations can occur in the i_1 region.

The third loop, i_3, executes those iterations of the original i loop whose values of i succeed the safe region. Again, within this loop, all array references indexed by i need to be tested. Because the subscript expressions on i are monotonically increasing across the iteration space of the i loop, only upper bound violations can occur in the i_3 region.

Within each of the resulting loops i_1, i_2, and i_3 the j loop is similarly split. For example, the body of resulting loop $j_{1,3}$ executes all iterations of the nest that attempt to reference elements of a or b that are below the corresponding lower bound, and elements of $b[i]$, $a[i]$, $a[i+1]$, and $a[i-1]$that are above the corresponding upper bound. In a typical correct numerical code, where all references are in bounds, only the body of loop $j_{2,2}$ will execute. This body is represented by $B(\textbf{notest}(i), \textbf{notest}(j))$, and because it has no tests it executes faster.

5.3.2 Computing the bounds of the split loops

To compute the bounds for the resulting i_1, i_2, and i_3 loops, we first have to compute the safe bounds for loop i: l_i^s and u_i^s. In the body of the i loop there are five array references indexed by i: $b[i]$, $a[i]$ (appearing twice), $a[i+1]$ and $a[i-1]$. The values of $\mathcal{L}_i^k$ and $\mathcal{U}_i^k$ are shown in Figure 12(A). They are computed using Equation (64) of Appendix A. The values of l_i^s and u_i^s can be compute according to Equations (34) and (35):

$$\begin{aligned} l_i^s &= \max(\mathcal{L}_i^1, \mathcal{L}_i^2, \mathcal{L}_i^3, \mathcal{L}_i^4, \mathcal{L}_i^5), \\ u_i^s &= \max(l_i^s - 1, \min(\mathcal{U}_i^1, \mathcal{U}_i^2, \mathcal{U}_i^3, \mathcal{U}_i^4, \mathcal{U}_i^5)). \end{aligned}$$

reference	$f_k(i)$	$\mathcal{L}_i^k$	$\mathcal{U}_i^k$
$a[i]$	$f_1(i) = i$	$\mathcal{L}_i^1 = \min(u_i + 1, \max(l_i, l_1))$	$\mathcal{U}_i^1 = \max(l_i - 1, \min(u_i, u_1))$
$a[i]$	$f_2(i) = i$	$\mathcal{L}_i^2 = \min(u_i + 1, \max(l_i, l_1))$	$\mathcal{U}_i^2 = \max(l_i - 1, \min(u_i, u_1))$
$a[i+1]$	$f_3(i) = i + 1$	$\mathcal{L}_i^3 = \min(u_i + 1, \max(l_i, l_1 - 1))$	$\mathcal{U}_i^3 = \max(l_i - 1, \min(u_i, u_1 - 1))$
$a[i-1]$	$f_4(i) = i - 1$	$\mathcal{L}_i^4 = \min(u_i + 1, \max(l_i, l_1 + 1))$	$\mathcal{U}_i^4 = \max(l_i - 1, \min(u_i, u_1 + 1))$
$b[i]$	$f_5(i) = i$	$\mathcal{L}_i^5 = \min(u_i + 1, \max(l_i, l_1))$	$\mathcal{U}_i^5 = \max(l_i - 1, \min(u_i, u_1))$

(A) Array references indexed by i

reference	$f_k(j)$	$\mathcal{L}_j^k$	$\mathcal{U}_j^k$
$a[i][j+1]$	$f_1(j) = j + 1$	$\mathcal{L}_j^1 = \min(u_j + 1, \max(l_j, l_2 - 1))$	$\mathcal{U}_j^1 = \max(l_j - 1, \min(u_j, u_2 - 1))$
$a[i][j-1]$	$f_2(j) = j - 1$	$\mathcal{L}_j^2 = \min(u_j + 1, \max(l_j, l_2 + 1))$	$\mathcal{U}_j^2 = \max(l_j - 1, \min(u_j, u_2 + 1))$
$a[i+1][j]$	$f_3(j) = j$	$\mathcal{L}_j^3 = \min(u_j + 1, \max(l_j, l_2))$	$\mathcal{U}_j^3 = \max(l_j - 1, \min(u_j, u_2))$
$a[i-1][j]$	$f_4(j) = j$	$\mathcal{L}_j^4 = \min(u_j + 1, \max(l_j, l_2))$	$\mathcal{U}_j^4 = \max(l_j - 1, \min(u_j, u_2))$
$b[i][j]$	$f_5(j) = j$	$\mathcal{L}_j^5 = \min(u_j + 1, \max(l_j, l_2))$	$\mathcal{U}_j^5 = \max(l_j - 1, \min(u_j, u_2))$

(B) Array references indexed by j

Figure 12: Lower and upper safe bounds for each reference.

The bounds for the resulting j loops are computed similarly. In the body of the j loop there are five array references indexed by j: $(a[i])[j+1]$, $(a[i])[j-1]$, $(a[i+1])[j]$, $(a[i-1])[j]$, and $(b[i])[j]$. Note that $b[i]$, $a[i]$, $a[i+1]$, and $a[i-1]$ are the arrays being accessed. The values of l_j^s and u_j^s can be computed by Equations (34) and (35), using values of $\mathcal{L}_j^k$ and $\mathcal{U}_j^k$ obtained through Equation (64) and shown in Figure 12(B). In this example, the arrays are rectangular, and the lower and upper bounds for $a[i]$, $a[i+1]$, $a[i-1]$ and $b[i]$ do not depend on the value of i. (Note that, in general, l_j^s and u_j^s would be functions of i.)

5.3.3 Checks that must still be done

The transformation just discussed creates regions of the loop that are free from violations on array references indexed by either i, j, or both. Figure 10 lists the specific violations that can occur in each region. Note that this particular behavior is specific to this loop. We chose to implement the loop using versions of the body that perform tests covering more than the strictly necessary violations. This allowed us to use only four versions of the loop body, as shown in Figure 10 and detailed in Figure 11. We perform an *all tests* on any i reference when outside the safe region for i. Correspondingly, we perform an *all tests* on any j reference when outside the safe region for j.

5.4 Summary of the general method

The exact method of Section 4 formed a different region of the iteration space of a loop for each possible combination of necessary array reference tests. The general method always partitions the iteration space of a loop into three regions: (i) a region for those iterations that need no test (the safe region), (ii) a region for those iterations that occur prior to the safe region, and (iii) a region for those iterations that occur after the safe region. In each of the non-safe loop regions, any test that might be needed for a reference in any iteration of that region is performed in all iterations of that region. This means that some additional tests might be executed compared to the exact method. For a nest of loops, the general method can be applied to each and every loop recursively and independently. Thus, for a loop nest of depth d, 3^d loop regions are created.

When generating code for the regions, several approaches can be taken. One approach generates a different version of the loop body for each region. When applied to to the example of Figure 10, this would generate only lower bound tests on arrays indexed by i in the i_1 loop, and only upper bound tests in the i_3 loop. This approach leads to 3^d versions of the loop body for a loop nest of depth d. A second approach, actually used in our example of Figures 10 and 11, uses only two versions of the loop body for each loop index: one with no tests on that index and one with all tests. This leads to 2^d versions of the loop body for a loop nest of depth d. A third approach would be to generate only two versions of the loop body, independent of the number of nested loops: one with all tests on all indices and one with no tests on any index. Obviously, the version with no tests can only be used in that region where no violations in any array reference can occur.

6 The compact method

The exponential expansion on the number of loops caused by the application of the general method can be highly undesirable in some cases. In this section we propose an alternative method that results in only a linear increase in the number of regions. The difference resides on how the transformation is applied to loop nests. In the compact method, the partitioning into three regions is always applied in outermost to innermost loop order. Furthermore, it is only applied to inner loops in the untested version of an outer loop body.

Again, consider the case of the two-dimensional loop nest $L(i, l_i, u_i, L'(j, l_j, u_j, B(i, j)))$. By first applying the transformation to the outer i loop, we generate a region without tests on array references indexed by i:

$$L(i, l_i, u_i, L'(j, l_j, u_j, B(i, j))) \Longrightarrow \boxed{\begin{array}{l} L_1(i, l_i, l_i^s - 1, L'(j, l_j, u_j, B(\mathbf{test}(i), j))) \\ L_2(i, l_i^s, u_i^s, L'(j, l_j, u_j, B(\mathbf{notest}(i), j))) \\ L_3(i, u_i^s + 1, u_i, L'(j, l_j, u_j, B(\mathbf{test}(i), j))) \end{array}} \tag{41}$$

We now apply the transformation to the instance of L' in the L_2 region. This results in a region without any tests on array references indexed by either i or j. The complete transformation is shown in Figure 13. This transformation partitions the two-dimensional iteration space into five regions, and requires three different versions of the loop body $B(i)$. It still generates the same region safe on i and j as the general method.

6.1 An example of the compact method

As an example, we apply the compact method to the two-dimensional loop nest of Figure 8. The resulting code is shown in Figure 14. Note that only loop i_2 has its j loop partitioned into three regions. When applied to a d-dimensional loop nest, the compact method generates $2d + 1$ regions of the loop iteration space. Because the two regions preceding and succeeding a safe region are similar, only $d + 1$ versions of the loop body must be generated.

$$L(i, l_i, u_i, L'(j, l_j, u_j, B(i,j))) \Longrightarrow \boxed{\begin{array}{l} L_1(i, l_i, l_i^s - 1, L'(j, l_j, u_j, B(\mathbf{test}(i), \mathbf{test}(j)))) \\ L_2(i, l_i^s, u_i^s, \boxed{\begin{array}{l} L'_1(j, l_j, l_j^s - 1, B(\mathbf{notest}(i), \mathbf{test}(j))) \\ L'_2(j, l_j^s, u_j^s, B(\mathbf{notest}(i), \mathbf{notest}(j))) \\ L'_3(j, u_j^s + 1, u_j, B(\mathbf{notest}(i), \mathbf{test}(j))) \end{array}}) \\ L_3(i, u_i^s + 1, u_i, L'(j, l_j, u_j, B(\mathbf{test}(i), \mathbf{test}(j)))) \end{array}}$$

Figure 13: Applying the compact method to a loop nest.

```
double a[][] = new double[l1 : u1][l2 : u2]
double b[][] = new double[l1 : u1][l2 : u2]

for (i1 = li; i1 ≤ li^s − 1; i1++) {                  // i lower bound violations
    for (j1,1 = lj; j1,1 ≤ uj; j1,1++) {             // j lower and upper bound violations
        B(test(i1), test(j1,1))
    }
}
for (i2 = li^s; i2 ≤ ui^s; i2++) {                    // no i violations
    for (j2,1 = lj; j2,1 ≤ lj^s − 1; j2,1++) {       // j lower bound violations
        B(notest(i2), test(j2,1))
    }
    for (j2,2 = lj^s; j2,2 ≤ uj^s; j2,2++) {          // no j violations
        B(notest(i2), notest(j2,2))
    }
    for (j2,3 = uj^s + 1; j2,3 ≤ uj; j2,3++) {       // j upper bound violations
        B(notest(i2), test(j2,3))
    }
}
for (i3 = ui^s + 1; i3 ≤ ui; i3++) {                  // i upper bound violations
    for (j3,1 = lj; j3,1 ≤ uj; j3,1++) {             // j lower and upper bound violations
        B(test(i3), test(j3,1))
    }
}
```

Figure 14: Structure of generated code for the compact method.

6.2 Summary of the compact method

Like the general method of Section 5, the compact method divides each loop into one safe and two non-safe regions. The differences between the two methods are manifested only when applying the transformation to a loop nest. The general method splits all nested loops into three regions, whereas the compact method splits only loops nested within the version without tests. This implies that within the non-safe versions of an outer loop the compact method may perform more tests than the general method. The benefit is that only a linear number of loop regions ($2d + 1$ for a loop nest of depth d) are necessary with the compact method, as opposed to an exponential number with the general method. The number of versions of the loop needed is a linear function of the nesting depth of the loop ($d+1$ for a loop nest of depth d).

The number of versions of code can be further reduced to two by using a fully tested version on any region that needs tests. Also, if the loop nesting is perfect, the structure of the generated code can be greatly simplified when only two versions are used. These observations provide the motivation for the restricted method of Section 7.

7 The restricted method

We now consider a transformation that can be applied to perfect loop nests, or any loop nest that can be transformed to a perfect loop nest via compiler techniques. The restricted method partitions the loop into multiple regions. Each region executes either a **test** (all array references are fully tested) or a **notest** (no array references are tested) version of the loop body. Having a perfect loop nest allows us to map every iteration of the loop nest onto a single point in a d-dimensional space, where d is the depth of the loop nest. Partitioning the loop into **test** and **notest** regions is equivalent to tiling this iteration space.

The advantage of the restricted method, when applied to a perfect d-dimensional loop nest, is that it can be implemented with only one instance of each version of the loop body. Also, the instances can be generated in-place in a fixed loop structure. The previous methods, when implemented with only two versions of code, require a specialized loop structure that invokes those versions from more than one point in the program.

We start by considering a d-dimensional rectangular loop nest:

$$L_1(i_1, l_{i_1}, u_{i_1}, L_2(i_2, l_{i_2}, u_{i_2}, \ldots, L_d(i_d, l_{i_d}, u_{i_d}, B(i_1, i_2, \ldots, i_d))\ldots)). \tag{42}$$

That is, the bounds l_{i_k} and u_{i_k} of loop i_k do not depend on the values of $i_1,\ldots,i_{k-1}$. The iteration space of this loop can be partitioned into consecutive regions described by a vector $\mathcal{R}[1:u_\delta]$. Each entry in this vector has the form:

$$\begin{aligned}
\mathcal{R}[\delta] &= (\mathbf{l}(\delta), \mathbf{u}(\delta), \tau), && (43)\\
\mathbf{l}(\delta) &= (l_{i_1}(\delta), l_{i_2}(\delta), \ldots, l_{i_d}(\delta)), && (44)\\
\mathbf{u}(\delta) &= (u_{i_1}(\delta), u_{i_2}(\delta), \ldots, u_{i_d}(\delta)), && (45)\\
\tau &= \{\textbf{test}\ |\textbf{notest}\}. && (46)
\end{aligned}$$

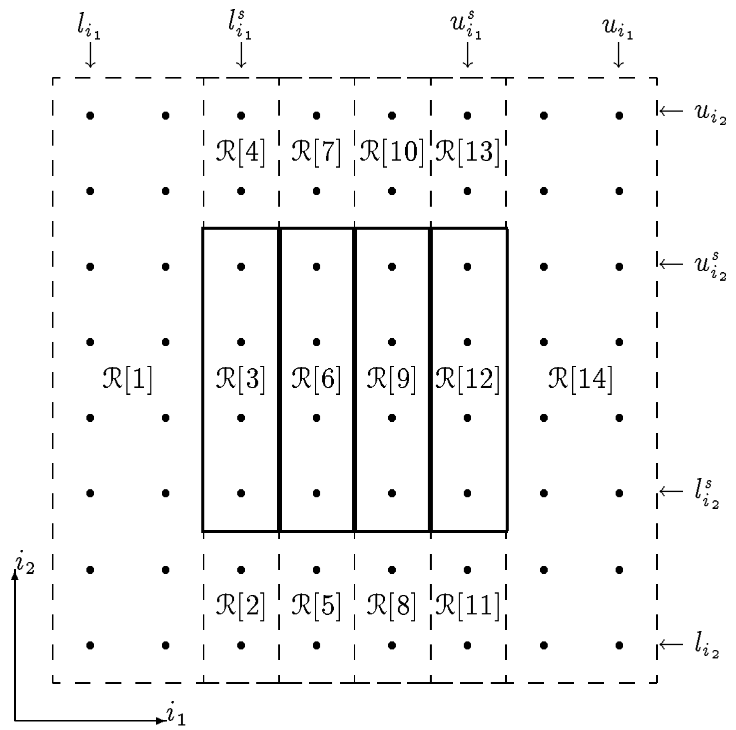
The vectors $\mathbf{l}$ and $\mathbf{u}$ define the lower and upper bounds for each loop in the region, respectively. The elements $l_{i_j}(\delta)$ and $u_{i_j}(\delta)$ denote the lower and upper bounds, respectively, of loop i_j in the region $\mathcal{R}[\delta]$. The definition of a region also contains a flag, τ, that has the value **test** if the region requires any array reference to be tested and **notest** otherwise.

Using the partitioning, the original loop can be executed by a driver loop that iterates over the regions:

for ($\delta = 1;\ \delta \leq u_\delta;\ \delta$++) {
  $L_1(i_1, l_{i_1}(\delta), u_{i_1}(\delta), L_2(i_2, l_{i_2}(\delta), u_{i_2}(\delta), \ldots, L_d(i_d, l_{i_d}(\delta), u_{i_d}(\delta), B(i_1, i_2, \ldots, i_d))\ldots)).$
}

Note that in previous sections single-dimensional regions were described by a vector containing the upper and lower bounds for the regions. Here a multi-dimensional region is described by vectors with upper and lower bounds for every index variable in the the loop nest. The techniques of the previous sections formed regions loop by loop, whereas the techniques of this section form regions for the entire loop nest.

Our goal is to partition the iteration space into regions that we can identify as either requiring tests on array references or not. Those regions that do not require tests can then be executed with a **notest** version of the loop body. We use the same kind of partitioning described in the compact method. First, an outer loop is divided into three regions. Then, the inner loop in the region without tests is divided again. This partitioning is illustrated in Figure 15, for a two-dimensional iteration space.



(A) Partitioning the iteration space into regions

- $\mathcal{R}[1]$: tests in i_1 and i_2
- $\mathcal{R}[2], \mathcal{R}[5], \mathcal{R}[8], \mathcal{R}[11]$: tests in i_2
- $\mathcal{R}[3], \mathcal{R}[6], \mathcal{R}[9], \mathcal{R}[12]$: no tests
- $\mathcal{R}[4], \mathcal{R}[7], \mathcal{R}[10], \mathcal{R}[13]$: tests in i_2
- $\mathcal{R}[14]$: tests in i_1 and i_2

(B) Mandatory tests in each region

Figure 15: Iteration space for a perfectly nested two-dimensional loop.

7.1 Computing vector $\mathcal{R}$

Vector $\mathcal{R}$ can be computed by procedure *regions* in Figure 16. Procedure *regions* takes seven parameters. The first five are input parameters, and describe the loop nest being optimized. They are:

1. The index j indicating that region extents along index variable i_j are being computed.
2. The vector $(\alpha_1, \alpha_2, \ldots, \alpha_{j-1})$, where α_k is the lower bound for loop index i_k in the regions to be computed.
3. The vector $(\omega_1, \omega_2, \ldots, \omega_{j-1})$, where ω_k is the upper bound for loop index i_k in the regions to be computed.
4. The dimensionality d of the loop nest.
5. The vector $\mathcal{B}[1:d]$, where $\mathcal{B}[k] = (l_{i_k}, u_{i_k}, l^s_{i_k}, u^s_{i_k})$ contains the full and safe bounds for loop i_k.

The next two parameters are the output of the procedure. The first output parameter is the vector of regions, $\mathcal{R}$, described in Equations 43 through 46, for the loop. The second is u_δ, the count of those regions. Note that u_δ is also used as an input, which gets incremented each time a new region is created.

An invocation of procedure *regions* for a given value of j partitions the loop nest formed by loops $i_j, i_{j+1}, \ldots, i_d$. It is only performed for safe values of $i_1, i_2, \ldots, i_{j-1}$. Procedure *regions* partitions loop i_j into three parts. The first part consists of iterations of i_j that precede the safe region of i_j. The inner loops of i_j are not partitioned. These regions necessarily require testing of the array references, since they are unsafe on references indexed by i_j. Regions corresponding to this part of the iteration space are computed in statements $S1$ and $S2$, and correspond to the regions $\mathcal{R}[1]$ (for $j = 1$), $\mathcal{R}[2]$, $\mathcal{R}[5]$, $\mathcal{R}[8]$, and $\mathcal{R}[11]$ (for $j = 2$) in the two-dimensional iteration space of Figure 15. The third part consists of the iterations of i_j that succeed the safe region of i_j. Again, the inner loops are not partitioned, and testing is required. Regions corresponding to this part of the iteration space are computed in statements $S11$ and $S12$, and correspond to the regions $\mathcal{R}[4]$, $\mathcal{R}[7]$, $\mathcal{R}[10]$, $\mathcal{R}[13]$, (for $j = 2$) and $\mathcal{R}[14]$ (for $j = 1$) in Figure 15. The middle (second) part consists of the iterations of i_j that are within its safe region. If i_j is the innermost loop, this is computed by statements $S4$ and $S5$, and the partitioning of loop i_j is complete. No tests are required. If i_j is not the innermost loop, then the partitioning is applied recursively to loop i_{j+1} for each iteration of i_j in its safe region, as shown in lines $S7$ and $S8$.

To compute the entire vector of regions, the invocation *regions*$(1, (), (), d, \mathcal{B}, \mathcal{R}, u_\delta = 0)$ should be performed. At the end of the computation, vector $\mathcal{R}$ contains the description of the regions, and the value of u_δ is the total number of regions in $\mathcal{R}$.

Although the example in Figure 15 and the notation imply that the iteration space passed to *regions* in $\mathcal{B}$ is rectangular, the algorithm is not restricted to rectangular loop nests. In particular, if the expressions for l_{i_j}, u_{i_j}, $l^s_{i_j}$, and $u^s_{i_j}$ are functions of $i_k, 1 \le k < j$, the computation performed by *regions* is correct.

---

procedure *regions*$(j, (\alpha_1, \alpha_2, \ldots, \alpha_{j-1}), (\omega_1, \omega_2, \ldots, \omega_{j-1}), d, \mathcal{B}, \mathcal{R}, u_\delta)$ $\{$
$S1$ $u_\delta = u_\delta + 1$
$S2$ $\mathcal{R}[u_\delta] = ((\alpha_1, \ldots, \alpha_{j-1}, l_{i_j}, l_{i_{j+1}}, \ldots, l_{i_d}), (\omega_1, \ldots, \omega_{j-1}, l^s_{i_j} - 1, u_{i_{j+1}}, \ldots, u_{i_d}),$ **test**$)$
$S3$ **if** $(j == d)$ $\{$
$S4$ $u_\delta = u_\delta + 1$
$S5$ $\mathcal{R}[u_\delta] = ((\alpha_1, \ldots, \alpha_{d-1}, l^s_{i_d}), (\omega_1, \ldots, \omega_{d-1}, u^s_{i_d}),$ **notest**$)$
$S6$ $\}$ **else** $\{$
$S7$ **for** $(k = l^s_{i_j};\ k \le u^s_{i_j};\ k{+}{+})$ $\{$
$S8$ *regions*$(j+1, (\alpha_1, \alpha_2, \ldots, \alpha_{j-1}, k), (\omega_1, \omega_2, \ldots, \omega_{j-1}, k), d, \mathcal{B}, \mathcal{R}, u_\delta)$
$S9$ $\}$
$S10$ $\}$
$S11$ $u_\delta = u_\delta + 1$
$S12$ $\mathcal{R}[u_\delta] = ((\alpha_1, \ldots, \alpha_{j-1}, u^s_{i_j} + 1, l_{i_{j+1}}, \ldots, l_{i_d}), (\omega_1, \ldots, \omega_{j-1}, u_{i_j}, u_{i_{j+1}}, \ldots, u_{i_d}),$ **test**$)$
$\}$

Figure 16: Procedure to compute the regions for a loop nest.

We can optimize the execution of the loop nest by using two versions of code inside the the driver loop: (i) a version $B($**notest**$(i_1),$ **notest**$(i_2), \ldots,$ **notest**$(i_d))$ that does not perform any of the array reference tests, and (ii) a version $B($**test**$(i_1),$ **test**$(i_2), \ldots,$ **test**$(i_d))$ that performs tests on all array references. Version (i) is used only for regions where $\mathcal{R}[\delta].\tau$ is **notest**, while version (ii) is used for all other regions. This corresponds to forcing all regions with *any* potential reference violation to perform *all* reference tests. The optimized loop nest can be implemented by the following construct:

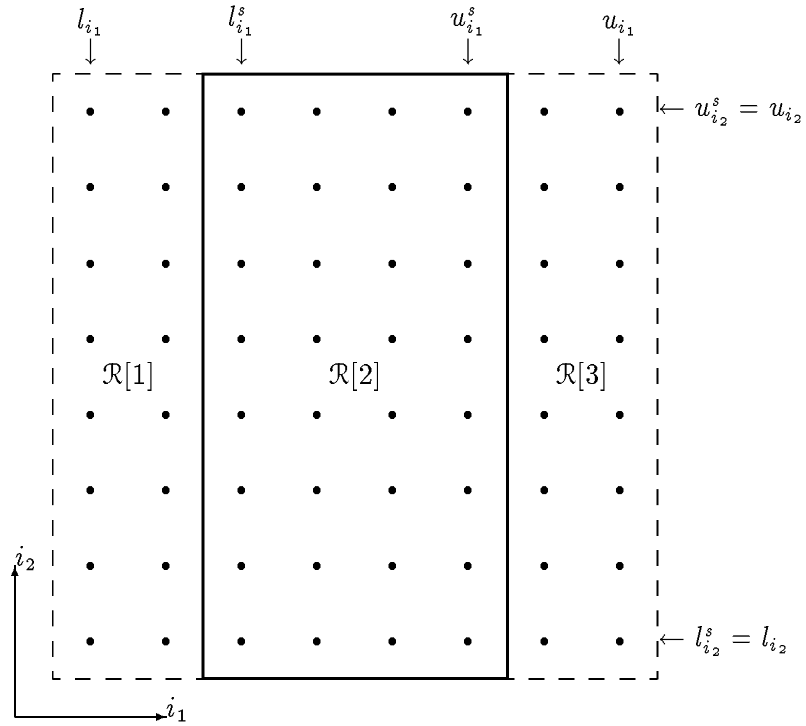
$$
\begin{array}{l}
\textbf{for}\ (\delta = 1;\ \delta \leq u_\delta;\ \delta\text{++})\ \{ \\
\quad \textbf{if}\ (\mathcal{R}[\delta].\tau == \textbf{test})\ \{ \\
\quad\quad L_1(i_1, l_{i_1}(\delta), u_{i_1}(\delta), \\
\quad\quad\quad L_2(i_2, l_{i_2}(\delta), u_{i_2}(\delta), \\
\quad\quad\quad\quad \ldots, \\
\quad\quad\quad\quad\quad L_d(i_d, l_{i_d}(\delta), u_{i_d}(\delta), B(\textbf{test}(i_1), \textbf{test}(i_2), \ldots, \textbf{test}(i_d))) \\
\quad\quad\quad\quad \ldots \\
\quad\quad\quad) \\
\quad\quad) \\
\quad \}\ \textbf{else}\ \{ \\
\quad\quad L_1(i_1, l_{i_1}(\delta), u_{i_1}(\delta), \\
\quad\quad\quad L_2(i_2, l_{i_2}(\delta), u_{i_2}(\delta), \\
\quad\quad\quad\quad \ldots, \\
\quad\quad\quad\quad\quad L_d(i_d, l_{i_d}(\delta), u_{i_d}(\delta), B(\textbf{notest}(i_1), \textbf{notest}(i_2), \ldots, \textbf{notest}(i_d))) \\
\quad\quad\quad\quad \ldots \\
\quad\quad\quad) \\
\quad\quad) \\
\quad \} \\
\}
\end{array}
\tag{47}
$$

7.2 An important optimization

If, for a particular value of j, $l^s_{i_j} = l_{i_j}$ and $u^s_{i_j} = u_{i_j}$, then the safe region along axis i_j of the iteration space corresponds to the entire extent of the axis. If $l^s_{i_k} = l_{i_k}$ and $u^s_{i_k} = u_{i_k}$ for $k = j, \ldots, d$, then axis i_{j-1} can be partitioned into only three regions: (i) one region from l_{j-1} to $l^s_{i_{j-1}} - 1$, (ii) one region from $l^s_{i_{j-1}}$ to $u^s_{i_{j-1}}$, and (iii) one region from $u^s_{i_{j-1}} + 1$ to $u_{i_{j-1}}$. Each of these regions spans the entire iteration space along axes $i_j, i_{j+1}, \ldots, i_d$. This situation for a two-dimensional iteration space is illustrated in Figure 17. Note that the new partitioning results in only three regions. Collapsing multiple regions into a single region reduces the cost of computing the regions, the total number of iterations in the driver loop (47), and, consequently, reduces the run-time overhead of the restricted method. To incorporate this optimization in the computation of regions, procedure *regions* is modified as shown in Figure 37 of Appendix F.

7.3 An example of the restricted method

We apply the restricted method to the two-dimensional loop nest of Figure 8. The logical structure of the generated code is the same as that generated by the compact method in Figure 14. The actual implementation with the driver loop and the two versions of code is shown in Figure 18. The driver loop (with index δ) of Figure 18 executes u_δ iterations, where u_δ is the number of regions computed by procedure *regions*. On each iteration, either the version of the loop with run-time tests or the version with no run-time tests is executed. The code in Figure 18 instantiates the regions dynamically. This contrasts with the static instantiation of Figure 14.



- $\mathcal{R}[1]$: tests in i_1 and i_2
- $\mathcal{R}[2]$: no tests
- $\mathcal{R}[3]$: tests in i_1 and i_2

(A) Partitioning of the iteration space into regions (B) Mandatory tests in each region

Figure 17: Iteration space for a perfectly nested two-dimensional loop.

7.4 Summary of the restricted method

The restricted method works on perfect loop nests. It partitions the iteration space into multidimensional regions and generates two versions of the loop body. Each region is executed with either the **test** version of the body or the **notest** version. Only regions that are free from any possible access violation can use the **notest** version.

8 Iterative computation of loop bounds

Section 3 and Appendix A discuss how to partition an iteration space into regions for a variety of common forms of subscript functions. In this section, we discuss how regions-based testing optimization can often be performed even when the subscript expressions are not one of the forms discussed in those sections.

The technique is similar to that of the *inspector/executor* method for parallelizing loops [7]. We decompose the computation and execution of the regions into an *inspector* phase and an *executor* phase. The *inspector* phase examines the references within the iteration space of a loop and computes a list of iteration subspaces (regions). Some regions need run-time tests, whereas others do not. The *inspector* phase is analogous to procedure *regions* of Section 7. The *executor* phase traverses the list of iteration subspaces, and executes them using different versions of the loop body. Thus the*executor* phase corresponds to the driver loop of Section 7.

```
double a[][] = new double[l_1 : u_1][l_2 : u_2]
double b[][] = new double[l_1 : u_1][l_2 : u_2]
...
regions(1, (), (), 2, ((l_i, u_i, l_i^s, u_i^s), (l_j, u_j, l_j^s, u_j^s)), R, u_δ = 0)
for (δ = 1; δ ≤ u_δ; δ++) {
   if (R[δ].τ == test) {
      for (i = l_i(δ); i ≤ u_i(δ); i++) {
         for (j = l_j(δ); j ≤ u_j(δ); j++) {
            // all out-of-bounds tests are performed
            B(test(i), test(j))
         }
      }
   } else {
      for (i = l_i(δ); i ≤ u_i(δ); i++) {
         for (j = l_j(δ); j ≤ u_j(δ); j++) {
            // no out-of-bounds tests are performed
            B(notest(i), notest(j))
         }
      }
   }
}
```

Figure 18: Generated code for the restricted method.

8.1 Construction of the *inspector* phase

We show how to construct an inspector for a singly nested loop. This method can then be recursively applied to a loop nest. Let $L(i, l, u, B(i))$ be a loop on i, where $B(i)$ contains a series of ρ references $A_1[\sigma_1], A_2[\sigma_2], \ldots, A_\rho[\sigma_\rho]$, where σ_j is a function defined in the iteration space of L. A reference can be of the form $A_j[A_k[\sigma_k]]$, where A_k is also an array. In general, $A_k[\sigma_k]$ may be present as a term in σ_j. We label the array references so that $A_j[\sigma_j]$ executes before $A_{j+1}[\sigma_{j+1}]$.

The inspector constructed takes the form as shown in Figure 19(A). The first argument to procedure *regions* is the output $\mathcal{R}$, a vector of regions. An element $\mathcal{R}[\delta]$ of the region vector consists of:

$$\mathcal{R}[\delta] = (l(\delta), u(\delta), \tau)$$

where $l(\delta)$ and $u(\delta)$ are the bounds of region $\mathcal{R}[\delta]$, and τ is its test flag. The next argument, u_δ, is also an output: the number of regions in vector $\mathcal{R}$. The other arguments for procedure *regions* are the input of the procedure. They are: (i) the lower and upper bounds, l and u, of the loop, and (ii) the list of array references $(A_1[\sigma_1], A_2[\sigma_2], \ldots, A_\rho[\sigma_\rho])$.

The inspector enumerates all iterations from the loop, by executing a **for** $(i = l;\ i \leq u;\ i{+}{+})$ loop. For each iteration i, it checks all array references $A_j[\sigma_j]$ and determines if a test is necessary. If the evaluation of σ_j itself causes a violation, we define $\sigma_j = -\infty$. If σ_j is invariant with respect to i, optimizations can be performed to reduce the number of evaluations in the inspector. The inspector marks in a flag *check* if a test is required for iteration i. If *check* is the same as *oldcheck*, this iteration i belongs to the same region as the previous iterations, and we update the upper bound of the current region. Otherwise, it is the first iteration of a new region.

```
procedure regions(ℛ, u_δ, l, u, (A_j[σ_j], j = 1, ..., ρ)) {
   u_δ = 0
   oldcheck = undefined
   for (i = l; i ≤ u; i++) {
      check = notest
      for (j = 1; j ≤ ρ; j++) {
         if ((σ_j < lo(A_j)) ∨ (σ_j > up(A_j))) {
            check = test
         }
      }
      if (check == oldcheck) {
         u(u_δ) = i
      } else {
         u_δ = u_δ + 1
         l(u_δ) = u(u_δ) = i
         ℛ[u_δ].τ = check
         oldcheck = check
      }
   }
}
```

(A) Inspector

```
for (δ = 1; δ ≤ u_δ; δ++) {
   if (ℛ[δ].τ == test ) {
      for (i = l(δ); i ≤ u(δ); i++) {
         B(test(i))
      }
   } else {
      for (i = l(δ); i ≤ u(δ); i++) {
         B(notest(i))
      }
   }
}
```

(B) Executor

Figure 19: Template for constructing an inspector/executor.

8.2 Construction of the *executor* phase

The *executor*, shown in Figure 19(B), consists of a driver loop (indicated by an index variable δ), that iterates over the regions. Each region is executed with one of the two different versions of the loop body, one with tests and one without.

8.3 Optimizations and alternate constructions

The inspector/executor technique can be applied to each loop in a loop nest independently, as we did for the general method in Section 5. Alternatively, it can be directly applied to a d-dimensional perfect loop nest. In this case, the inspector has to enumerate all iterations in the d-dimensional iteration space, and the executor consists of a driver loop around different versions of the loop nest.

For the inspector/executor method to be effective, the inspector phase has to be hoisted out of a loop nest. When that is possible, the same vector of regions $\mathcal{R}[1 : u_\delta]$ can be used in multiple instances of the executor. The number of checks is reduced by the number of iterations in the loops from which the inspector was hoisted. We show such an example in Section 8.4.

Finally, it is also possible to build an inspector/executor pair that uses more refined versions of the loop body. For example, one could use all 3^ρ versions generated in the exact method.

8.4 An example

In this example, the loop of Figure 20(A) is transformed. The inspector for this loop is shown in Figure 20(B) and the executor in Figure 20(C). Note that we have optimized the loop by moving the *inspector* phase out of the j loop. This is possible because the array reference patterns are not dependent on j. Thus, even though there are $O(u_i u_j)$ computations being performed, only $O(u_i)$ tests are necessary.

```
double a[] = new double[l_1 : u_1]
integer b[] = new integer[l_2 : u_2]
for (j = 1; j ≤ u_j; j++) {
   for (i = 1; i ≤ u_i; i++) {
      a[b[i]] += h(i, j)
   }
}
```

(A) Original loop

```
u_δ = 0
oldcheck = undefined
for (i = 1; i ≤ u_i; i++) {
   check = notest
   if ((i < l_2) ∨ (i > u_2)) {
      check = test
   } elsif ((b[i] < l_1) ∨ (b[i] > u_1)) {
      check = test
   }
   if (check == oldcheck) {
      u(u_δ) = i
   } else {
      u_δ = u_δ + 1
      l(u_δ) = u(u_δ) = i
      R[u_δ].τ = check
      oldcheck = check
   }
}
```

(B) Inspector

```
for (j = 1; j ≤ u_j; j++) {
   for (δ = 1; δ ≤ u_δ; δ++) {
      if (R[δ].check == test) {
         for (i = l_i(δ); i ≤ u_i(δ); i++) {
            ifAccessViolation(b, i) throwException
            ifAccessViolation(a, b[i]) throwException
            a[b[i]] += h(i, j)
         }
      } else {
         for (i = l_i(δ); i ≤ u_i(δ); i++) {
            a[b[i]] += h(i, j)
         }
      }
   }
}
```

(C) Executor

Figure 20: A program optimized using inspector/executor.

8.5 Summary of inspector based methods

The inspector-based methods differ from the exact, general, compact, and restricted methods in how regions are formed. In the former methods, regions are computed analytically using the formulas of Section 3 and Appendix A. In the inspector based methods the regions are computed by executing the code that generates the subscripts for the references being optimized. This allows references whose subscripts preclude an analytic computation of regions to be optimized. There are two caveats, however. First, because the inspector overhead is proportional within a constant factor to the cost of executing an instance of the loop, the methods are most effective when the inspector for a loop can be hoisted out of that loop. Second, there are still some loops which cannot be optimized with inspector-based methods. Figure 21 shows such a loop. Because the execution of an inspector for this loop would have side effects (other than region computation) that live beyond the life of the inspector, forming an inspector with the methods we have discussed would alter the outcome of the program. This loop can be optimized using *speculative* methods. Because they are not region based, they are beyond the scope of this paper. They are discussed in [8].

$$
\begin{array}{l}
\textbf{for } (i = l; i \leq u; i{+}{+}) \ \{ \\
\quad \sigma = a[i] \\
\quad a[\sigma] = f(i) \\
\}
\end{array}
$$

Figure 21: A loop that cannot be optimized by inspector-based methods.

9 Multithreading considerations

Up to now, the problem of other threads of execution and optimized code being active at the same time has been ignored. If arrays had static shape, that is, if it were not possible for an array to change shape during the course of execution of a routine, multithreading would not be a problem. Multithreading is beyond the scope of the current Fortran, C, and C++ language definitions. Java, however, has made multithreading an integral part of the language. Furthermore, it is possible for one thread of a Java program to alter the shape of a multidimensional array that is being accessed by other threads.

The Java memory model enables a solution to this problem. The coherent copy of a shared variable resides in *main memory*. A local copy of any shared variable can be copied by a thread from main memory to *working memory*. The working memory is accessible only by the thread, and therefore is not alterable by any other thread. Java requires that:

1. No synchronization can occur within a thread between time t_m, when a variable is copied from main to local memory, and time t_a, when the variable is accessed by the thread. This can be enforced by refreshing the shared variable in local memory after a synchronization and before the access.

2. Before a synchronization within a thread is finished, all shared variables that have been modified since the last synchronization point in that thread must be written back to the main memory.

3. If a shared variable has been written by a thread, it must be written to main memory before its value can be read from main memory by the same thread. This prevents locally written values to shared variables from being lost.

4. As long as item 3 is obeyed, the value of a shared variable can be updated from global memory at any time prior to an access by a thread.

The consequences of the Java memory model rules to our work are three-fold. First, it is legal and valid within a thread to cache in local memory the values of a variable – even if the thread is not synchronized. Second, if the thread is not synchronized, the cached values need not be written back. Third, if a synchronization point exists within a loop nest being optimized, the shapes of shared arrays have to be conservatively assumed to have changed at the synchronization point. To prove that the shape of a shared array does not change across a synchronization requires proving that none of the threads that have access to the array at that time can change its shape.

We describe how to handle the case in which a body of code without synchronization accesses a (potentially) shared array. We can divide arrays into two distinct parts. The first part is comprised of *descriptors*, those parts of an array that contain pointers to other descriptors or to rows of data. The second part is comprised of data, the one-dimensional rows that contain the actual elements. In Figure 22, the dashed boxes indicate descriptors and the solid boxes indicate data. In general, a d-dimensional rectangular array $A[1 : n_1][1 : n_2] \ldots [1 : n_d]$ consists of one level-0 descriptor that points to a vector of n_1 level-1 descriptors. Each level-1 descriptor points to a vector of n_2 level-2 descriptors, for a total of $n_1 n_2$ level-2 descriptors. There are a total of $O(n_1 n_2 \ldots n_{d-1})$ descriptors. The last axis of the array contains the actual data, in the form of vectors of n_d elements, pointed to by the level-$(d-1)$ descriptors.

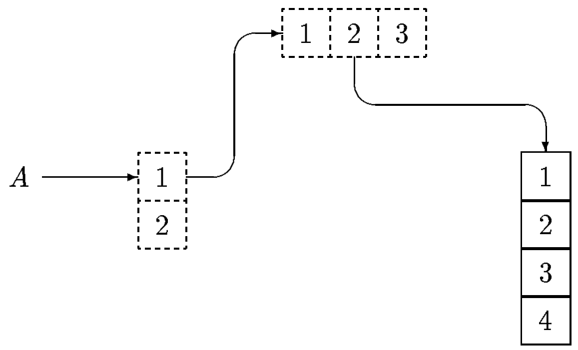


Figure 22: Multidimensional array organization, showing the separation between descriptors and data.

The following then ensures the thread safety of our transformations: before every optimized body of code, a copy of the descriptors part of each shared array A is cached in working memory. Note that if a cached descriptor for A that is valid by the Java thread semantics is already present, it may be used for the purposes we describe. This action insulates the thread executing the optimized code from any changes to the shared array shape caused by other threads. Also note that only the array shape is relevant in computing the safe and unsafe regions for our transformations. The data part of the array can be modified by other threads without any effects on those regions. Using this caching of descriptors, the only places that shape changes become visible are at synchronization points. The shape of a shared array should be marked as variant across those points. This, in turn, can prevent many optimizations.

At first glance the caching of descriptors may seem expensive. In general, given a d-dimensional rectangular array $A[1 : n_1][1 : n_2] \ldots [1 : n_d]$ array, only $O(n_1 n_2 \ldots n_{d-1})$ storage needs to be cached. Thus, a one-dimensional array needs only a constant amount of storage, and and $n_1 \times n_2$ array needs only n_1 words of storage. Therefore, if most of the array is accessed within the loop, the amount of storage cached is approximately a factor of n_d less that the number of references.

10 Experimental results

To test the effectiveness of our compiler transformations, we developed a prototype framework. This prototype framework currently implements only the restricted method, and only handles perfectly nested rectangular loops. The framework consists of a Java-to-Java translator that produces the two versions of code necessary for the restricted method. These versions are then compiled into executable object code using the IBM *High Performance Compiler for Java* (HPCJ) [4]. HPCJ has a switch to generate code without any run-time checks, on a class basis.

10.1 Benchmarks

Using the prototype framework, we applied the transformations in the restricted method to Java programs in a benchmark suite. This suite consists of two numerical kernels and three application kernels, all with array-intensive computations. The two numerical kernels are a vector dot-product operation (DDOT), and a matrix-multiply operation (MATMUL). The three application kernels are a shallow-water simulation kernel (SHALLOW), a data-mining kernel (BSOM), and a cryptography kernel (CBC). We compare the performance of a Java implementation of each benchmark with a corresponding version written in either C (DDOT, MATMUL, and CBC) or C++ (SHALLOW and BSOM). The C and C++ versions were written and compiled according to what we call *Java rules:* (i) two-dimensional arrays are represented as vectors of pointers to one-dimensional arrays, and (ii) any compiler optimizations that violate the Java IEEE 754 floating-point semantics (exact reproducibility of results) are disabled. The impact of these rules on the performance of C or C++ programs is beyond the scope of this paper. We discuss each of the benchmarks in more detail below.

10.1.1 DDOT

The DDOT benchmark computes the dot-product $\sum_{i=1}^{n} x_i y_i$ of two vectors x and y of n double-precision floating-point numbers. The Java code that implements DDOT is straightforward and shown in Figure 23. The C code is very similar. The computation of DDOT requires $2n$ floating-point loads and $2n$ floating-point operations (n multiplies and n adds). For our measurements we use $n = 10^6$. We report the performance of DDOT in Mflops.

```
static double[] x = new double[n];
static double[] y = new double[n];

double sum = 0;
for (int i=0; i<n; i++) {
  sum += x[i] * y[i];
}
```

Figure 23: Java code that implements DDOT.

10.1.2 MATMUL

The MATMUL benchmark computes the matrix operation $C = C + A \times B$, where C is an $m \times p$ matrix, A is an $m \times n$ matrix, and B is an $n \times p$ matrix. The elements are double-precision floating-point numbers. The Java implementation of this matrix operation is illustrated in Figure 24. The C implementation is very similar, with the matrices implemented as vectors of pointers to rows of elements. This is in accordance with the previously mentioned *Java rules* for C. The computation of MATMUL requires $2mnp$ floating-point operations. (mnp adds and mnp multiplies.) For our measurements, we used $m = n = p = 64$. We also report the performance of MATMUL in Mflops.

```java
static double[][] A = new double[m][n];
static double[][] B = new double[n][p];
static double[][] C = new double[m][p];

static void matmul() {

  int i;
  int j;
  int k;

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

Figure 24: Source code for $C = C + A \times B$ in Java.

10.1.3 SHALLOW

The SHALLOW benchmark is a computational kernel from a shallow water simulation code from the National Center for Atmospheric Research (NCAR). It consists of approximately 200 lines of code, in either the C++ or Java version. The data structures in SHALLOW consist of 14 matrices (2-dimensional arrays) of size $n \times m$ each. The computational part of the code is organized as a time step loop, with several array operations executed in each time step (iteration), as shown in Figure 25. In that figure we indicate the number of occurrences for each kind of array operation. The notations $A + A$, $A * A$, and A/A denote addition, multiplication, and division of all *corresponding array elements*. The notation $s * A$ denotes multiplication of a scalar value by each of the elements of an array. Therefore, each iteration of the time step loop executes $65mn$ floating-point operations. The matrix operations do not appear explicitly. Instead, they are fused into three double-nested loops.

$$
\begin{array}{l}
\textbf{for } t = 1, \ldots, T \; \{ \\
\quad \begin{array}{rcccccc}
39 & \times & (A & + & A) \\
8 & \times & (A & * & A) \\
17 & \times & (s & * & A) \\
1 & \times & (A & / & A)
\end{array} \\
\}
\end{array}
$$

Figure 25: General structure of the SHALLOW benchmark.

Just as in the MATMUL benchmark, the matrices in the C++ code are implemented as vectors of pointers to rows of elements. For our measurements we fix the number of time steps $T = 20$ and use $n = m = 256$.

Once again, performance for SHALLOW is reported in Mflops.

10.1.4 BSOM

BSOM (Batch Self-Organizing Map) is a data-mining kernel being incorporated into Version 2 of the IBM Intelligent Miner. It implements a neural-network-based algorithm to determine clusters of input records that exhibit similar attributes of behavior. The simplified kernel used for this study consists of approximately 300 lines of code, in either the C++ or Java versions.

We time the execution of the *training* phase of this algorithm, which actually builds the neural network. The training is performed in multiple passes over the training data. Each pass is called an *epoch*. Let e be the number of epochs, let n be the number of neural nodes, let r be the number of records in the training data, and let m be the number of fields in each input record. For each epoch and each input record, the training algorithm performs nm connection updates in the neural network. Each update requires 5 floating-point operations.

For our measurements, we use $e = 25$, $n = 16$, and $r = m = 256$. We report the performance of BSOM in millions of connection updates per second, or MCUP/s, as is usually done in the literature for neural-network training.

10.1.5 CBC

Our last benchmark is an implementation of CBC, or *cipher block chaining* [9]. CBC is a block cipher mode in which a feedback mechanism is used to encrypt a vector of n blocks of data. Each block is encrypted in sequence. The result of encrypting a block is XORed with the next block before encrypting this next block. Let $P = (P_1, \ldots, P_n)$ be the vector of plaintext blocks and let $C = (C_1, \ldots, C_n)$ be the vector of ciphertext blocks. Then

$$C_i = E_K(P_i \oplus C_{i-1}),\ i = 1, \ldots, n$$

where $E_K(\cdot)$ is the encrypting function with key K and C_0 is the *initial chaining value*. The CBC benchmark uses the *data encryption standard* (DES) algorithm [9] to encrypt the blocks. DES transforms each 64-bit plaintext block into a 64-bit ciphertext block, using a 56-bit key.

We use our CBC benchmark to encrypt a vector of 128k blocks (1 MB) and we report its performance in millions of bytes encrypted per second (MB/s). We measure both C and Java versions of this benchmark. The CBC benchmark performs integer and logical operations on array elements. This contrasts with the mostly floating-point operations that the other four benchmarks perform.

10.2 Execution environment

We performed our experiments on an IBM RS/6000 model 590 workstation, running AIX 4.2. This workstation has a 66 MHz POWER2 processor, and is configured with 256 kB of level-1 cache and 512 MB of main memory. The C and C++ programs were compiled with C Set++ V3 version 3.1.4, and we used version βA9 of HPCJ to compile the Java programs.

10.3 Results

Table 1 summarizes the results from our experiments. For each benchmark and code version, we list the measured performance in the appropriate units. The first row (C or C++ with Java rules) of the table lists the results for the C or C++ version of the benchmarks. The second row (HPCJ no checks) lists the results for the Java version compiled with HPCJ, with *all run-time checks disabled*. Note that this version is not Java-compliant, since it will not detect any indexing violations that may occur. The third row (Transformations) lists the results for the Java version with our transformations applied. This version is Java-compliant, since all necessary tests are performed. When it is necessary to make private copies of the descriptors of an array (as described in Section 9), the time for copying is included in computing the performance. Finally, the last row (HPCJ with all checks) of the table lists the performance measured for Java compiled with HPCJ, with the run-time checks enabled. Again, this version is Java-compliant.

Table 1: Summary of results from applying our transformations.

Code version	Performance on RS/6000 590				
	DDOT (Mflops)	MATMUL (Mflops)	SHALLOW (Mflops)	BSOM (MCUP/s)	CBC (MB/s)
C or C++ with Java rules	46.3	33.3	34.3	10.1	1.28
HPCJ no checks	39.1	33.3	39.2	9.2	0.95
Transformations (100% Java)	**38.5**	**30.8**	**39.1**	**7.6**	**0.70**
HPCJ with all checks (100% Java)	5.9	2.2	3.1	0.6	0.19

From Table 1, we observe that HPCJ can produce executable code for Java that is very competitive with code produced for C or C++, *if the run-time checks are disabled*. However, when the checks are present, the performance of Java code is degraded by as much as 15 times (MATMUL). This indicates that HPCJ already implements many of the optimizations necessary to make Java competitive with C or C++ in performance. In fact, the Java version of SHALLOW achieves higher performance than the C++ version, because of better pointer disambiguation in Java than in C++. The executable code produced by HPCJ is hampered only by the need for run-time tests. When our transformations are applied to the code, part of the execution can be performed without any run-time tests. That is the reason why the performance of the transformed code is so much better than that of HPCJ with all tests.

We compare the performance achieved for Java with our transformations to the performance of the corresponding C or C++ code. That comparison is shown in Table 2. The '%' row in that table indicates the fraction of C or C++ performance that the Java code with transformation achieves. We observe that we can achieve between 55% and 114% of C or C++ code performance with code that is entirely legal Java.

Table 2: Comparing the performance of C and Java with transformations.

Code version	Performance on RS/6000 590				
	DDOT (Mflops)	MATMUL (Mflops)	SHALLOW (Mflops)	BSOM (MCUP/s)	CBC (MB/s)
C with Java rules	46.3	33.3	34.3	10.1	1.28
Transformations (100% Java)	38.5	30.8	39.1	7.6	0.70
% (Java/C)	**83%**	**92%**	**114%**	**75%**	**55%**

Finally, we compare the performance of code produced with our transformations to the performance of code generated by HPCJ with all run-time checks enabled. Those results are shown in Table 3. The 'Improvement' row lists the performance ratio between the version with transformations and the version without. We observe that we achieve performance improvements of up to 14 times when we apply our transformations.

Table 3: Comparing the performance of Java with and without transformations.

Code version	Performance on RS/6000 590				
	DDOT (Mflops)	MATMUL (Mflops)	SHALLOW (Mflops)	BSOM (MCUP/s)	CBC (MB/s)
Transformations (100% Java)	38.5	30.8	39.1	7.6	0.70
HPCJ with all checks	5.9	2.2	3.1	0.6	0.19
Improvement (100% Java)	**6.5** ×	**14.0** ×	**12.6** ×	**12.7** ×	**3.7** ×

11 Related work

A great deal of work has been done in the area of optimizing bounds checking for arrays. Most of this work has been done in the context of programming languages with no requirements as to the execution state at the time the bounds violation occurs. In Java, a violation must generate an exception at a very precise point in the execution of the program. The bounds checking work for Ada [10] confronts a problem similar to the one we face, but takes a less aggressive approach than we do.

There are two main approaches in the literature to optimizing array bounds checks: (i) the use of static data-flow analysis information to determine that a test is unnecessary [11, 12, 13, 14, 10], and (ii) the use of data-flow information and symbolic analysis at compile time to reduce the dynamic number of tests remaining in the program [15, 16, 17, 18, 19].

Work in the first group uses data-flow information to prove at compile time that an array bounds violation cannot occur at run-time, and therefore, that the test for the violation is unnecessary. Using the terms of our discussion, the goal of this work is to identify loops that are safe regions. In contrast, the goal of our work is to transform loops to place the maximum possible number of iterations in safe regions (with constraints on the number of loop versions or regions). Methods of the first group use information about loop and array bounds, and about subscript expressions, to prove that either no access violation will occur in a loop execution, or that an access violation might occur. In the former case, no tests are generated. In the latter case, tests are generated as needed. The difficulty of this approach is that the information needed to prove that no violation will occur might not be available at compile time. These techniques would have better results with *just-in-time* compilation, but the analysis overhead then becomes problematic.

Work in the second group attempts to reduce the dynamic and static number of bounds tests. It also attempts to reduce the overhead induced by a test even if it cannot be eliminated. This is done (i) by hoisting tests out of loops when possible [19] and (ii) by also determining that a test is covered by another test [18, 17, 15, 16] and can be eliminated.

The optimization method of Markstein, Cocke, and Markstein [19] is closely related to our computation of l^s and u^s for a single linear reference. It is implemented by three changes to the loop. First, before entering the loop a test is made to determine if a bounds violation occurs on the first iteration. If so, the loop is exited immediately. Second, the loop exit conditions are modified so that the loop terminates before the access violation occurs. Finally, at loop exit, the final iterate value is examined. If it is less than or equal to the upper loop bound in the original program, an access violation occurs and an exception (or *trap*, in their terminology) is thrown. The application of the method to the four point stencil problem of Figure 8 is shown in Figure 26.

To see how the removal of redundant (covered) tests works, consider the naive sequence of tests in Figure 9. If $a[i-1]$ and $a[i+1]$ are both legal references, then $a[i]$ is also legal; therefore the explicit test ifAccessViolation(a, i) is redundant. Similarly, considering references indexed by j, if $a[i][j+1]$ and $a[i][j-1]$ are both legal, then $a[i+1][j]$ and $a[i-1][j]$ are also legal. (Note that we need to know that the array

```
double a[][] = new double[l_1 : u_1][l_2 : u_2]
double b[][] = new double[l_1 : u_1][l_2 : u_2]
if ((l_i < max(l_1, l_1 + 1, l_1 − 1)) ∨ (l_i > min(u_1, u_1 + 1, u_1 − 1))) throwException
else {
   for (i = l_i; (i ≤ u_i) ∧ (i ≥ max(l_1, l_1 + 1, l_1 − 1)) ∧ (i ≤ min(u_1, u_1 + 1, u_1 − 1)); i++) {
      if ((l_j < max(l_2, l_2 + 1, l_2 − 1)) ∨ (l_j > min(u_2, u_2 + 1, u_2 − 1))) throwException
      else {
         for (j = l_j; (j ≤ u_j) ∧ (j ≥ max(l_2, l_2 + 1, l_2 − 1)) ∧ (j ≤ min(u_2, u_2 + 1, u_2 − 1)); j++) {
            b[i][j] = (a[i][j + 1] + a[i][j − 1] + a[i + 1][j] + a[i − 1][j])/4
         }
         if (j ≠ u_j + 1) throwException
      }
   }
   if (i ≠ u_i + 1) throwException
}
```

Figure 26: Optimization of the code of Figure 8 by the Markstein, Cocke, and Markstein method.

a is rectangular in order to make this statement.) Therefore, the explicit tests ifAccessViolation($a[i+1], j$) and ifAccessViolation($a[i-1], j$) are redundant. Finally, because b and a are arrays with the same shape, the tests ifAccessViolation(b, i) and ifAccessViolation($b[i], j$) are also redundant. After these optimizations we are left with the four explicit tests shown in Figure 27. Note that the remaining tests must be ordered appropriately.

Neither of these optimizations are usable with Java in general because the Java semantics requiring precise exceptions make the hoisting and reordering of tests illegal in many situations. Also, when an access violation exception is caught by a **try** block in the loop body, the loop should not be terminated (as would occur when hoisting tests). Nevertheless, the techniques for eliminating redundant tests can be used to optimize the performance of inspectors.

Some instruction set architectures provide special support for index checking, which have been used to speedup bounds checking in various implementations of Java. This is exemplified by the `bound` instruction in the Intel x86 architecture [20], used in the *jx* virtual machine, and the `trap` instructions of the IBM POWER architecture [21], used in the IBM Java JIT Compiler [22]. These instructions generate interrupts in the case of violations, which inhibit a wide range of compiler optimizations [3]. Optimizations that are inhibited include code motion, reordering, and prescient stores [1].

12 Conclusions

We have developed a set of optimizations to reduce the number of run-time tests that need to be performed in Java programs. All of the optimizations work by transforming a loop nest, which implements an iteration space, into code that partitions this iteration space into multiple regions. In some regions, run-time tests are performed, whereas in other regions they are not necessary. The optimizations differ on their level of refinement and practicality. The more refined methods generate code versions that are more specialized for the different characteristics of regions. They can cause an unacceptable code expansion. The less refined methods use fewer code versions and merge regions with similar characteristics. They cause a smaller code expansion and are more practical. All methods can create one or more regions that are completely free

```
double a[][] = new double[l_1 : u_1][l_2 : u_2]
double b[][] = new double[l_1 : u_1][l_2 : u_2]

for (i = l_i; i ≤ u_i; i++) {
   for (j = l_j; j ≤ u_j; j++) {
      ifAccessViolation(a, i + 1) throwException
      ifAccessViolation(a, i − 1) throwException
      ifAccessViolation(a[i], j + 1) throwException
      ifAccessViolation(a[i], j − 1) throwException
      b[i][j] = (a[i][j + 1] + a[i][j − 1] + a[i + 1][j] + a[i − 1][j])/4
   }
}
```

Figure 27: Optimization of the tests in Figure 9 by test coverage.

of run-time tests. In correct array-intensive programs, these regions are expected to perform all or almost all of the computations of a loop nest. In these cases, the less refined methods can deliver most of the improvements to be gained.

We want to emphasize the importance of creating regions without any possible array bounds and **null** pointer violations. The benefits of creating these regions are three-fold: First, run-time tests can be eliminated from the regions, which results in a direct performance improvement. Second, **try** blocks that catch bounds exceptions can be removed from the loop versions implementing the safe regions. Third and most importantly, as a consequence of the first two, the resulting regions have simpler code, which enables optimizations that would otherwise be hampered by the explicit run-time tests. In general, the ability to perform optimizations is enhanced by maximizing the extent of violation-free regions. Many forms of operation reordering and parallelization can be performed in these regions.

We implemented a prototype framework for performing our more practical optimization, which only requires two versions of code to be generated. We have measured the effectiveness of this optimization on a suite of array-intensive benchmarks. Our results indicate that Java code can achieve performance that is within 55 to 100% or more of the performance of C or C++ code, when the C or C++ code follows Java rules. C or C++ code that follows Java rules implements two-dimensional arrays as a vector of pointers to one-dimensional arrays, and does not perform optimizations that violate Java floating-point semantics.

Our benchmark suite currently consists of various array-intensive programs, including numerical, data-mining, and cryptography kernels. The array operations in these benchmarks are representative of the behavior of many applications. We intend to extend our suite to include graphics and image processing applications.

In all our discussion we have ignored the use of control-flow analysis. This analysis can be a powerful tool to prove that array references only occur in certain combinations (because of, for example, different paths through the body of a loop), thus reducing some of our code replication. We intend to investigate the usefulness of control-flow analysis as part of our future work.

Finally, our next step is to implement and integrate our transformations into IBM Java-related products. We will work with technical and data-mining application groups to help ensure the success of large-scale, array-intensive Java applications that depend on high performance.

Acknowledgements

The authors wish to thank George Almasi and Rick Lawrence for providing and assisting with the BSOM benchmark, David Edelsohn for providing and assisting with the CBC benchmark, and Yurij Baransky and Murthy Devarakonda for leading the Java performance efforts that initially shed light on this problem.

A Computing safe bounds

We describe how to compute the safe bounds $\mathcal{L}$ and $\mathcal{U}$ of an iteration space $i = l, \ldots, u$ with respect to an array reference. The safe region of the iteration space of i with respect to an array reference $A[f(i)]$ is the set of values of i that make $f(i)$ fall within the bounds of A. The safe region for this operation is defined by

$$(\text{lo}(A) \leq f(i) \leq \text{up}(A)) \wedge (l \leq i \leq u). \tag{48}$$

We first consider the case of $f(i)$ monotonically increasing. The iteration space is partitioned into three regions defined as follows:

$$\begin{aligned}
\mathcal{R}[1] &: (l \leq i \leq u) \wedge (f(i) < \text{lo}(A)), &(49)\\
\mathcal{R}[2] &: (l \leq i \leq u) \wedge (\text{lo}(A) \leq f(i) \leq \text{up}(A)), &(50)\\
\mathcal{R}[3] &: (l \leq i \leq u) \wedge (f(i) > \text{up}(A)). &(51)
\end{aligned}$$

Using the fact that

$$\begin{aligned}
\lceil f^{-1}(\text{lo}(A)) \rceil \leq i &\Rightarrow \text{lo}(A) \leq f(i) &(52)\\
\lfloor f^{-1}(\text{up}(A)) \rfloor \geq i &\Rightarrow \text{up}(A) \geq f(i) &(53)
\end{aligned}$$

the safe region $\mathcal{R}[2]$ can be defined by

$$\mathcal{R}[2] \; : \; i = \max\left(l, \lceil f^{-1}(\text{lo}(A)) \rceil\right), \ldots, \min\left(u, \lfloor f^{-1}(\text{up}(A)) \rfloor\right) \tag{54}$$

No test is required in this region.

Correspondingly, we can define region $\mathcal{R}[1]$, which requires a *lb test*, by

$$\mathcal{R}[1] \; : \; i = l, \ldots, \min\left(u+1, \lceil f^{-1}(\text{lo}(A)) \rceil\right) - 1, \tag{55}$$

and region $\mathcal{R}[3]$, which requires an *ub test*, by

$$\mathcal{R}[3] \; : \; i = \max\left(l-1, \lfloor f^{-1}(\text{up}(A)) \rfloor\right) + 1, \ldots, u. \tag{56}$$

To combine all three definitions, we can compute

$$\begin{aligned}
\mathcal{L} &= \min\left(u+1, \max\left(l, \lceil f^{-1}(\text{lo}(A)) \rceil\right)\right), &(57)\\
\mathcal{U} &= \max\left(l-1, \min\left(u, \lfloor f^{-1}(\text{up}(A)) \rfloor\right)\right) &(58)
\end{aligned}$$

and write

$$\begin{aligned}
\mathcal{R}[1] &: i = l, \ldots, \mathcal{L} - 1, &(59)\\
\mathcal{R}[2] &: i = \mathcal{L}, \ldots, \mathcal{U}, &(60)\\
\mathcal{R}[3] &: i = \mathcal{U} + 1, \ldots, u. &(61)
\end{aligned}$$

As previously discussed, $\tau[1]$, the test for $\mathcal{R}[1]$, is *lb test* and $\tau[3]$, the test for $\mathcal{R}[3]$, is *ub test*.

Similarly, if $f(i)$ is monotonically decreasing we can compute

$$\mathcal{L} = \min\left(u+1, \max\left(l, \lceil f^{-1}(\mathrm{up}(A))\rceil\right)\right), \tag{62}$$
$$\mathcal{U} = \max\left(l-1, \min\left(u, \lfloor f^{-1}(\mathrm{lo}(A))\rfloor\right)\right). \tag{63}$$

$\mathcal{R}[1]$, $\mathcal{R}[2]$ and $\mathcal{R}[3]$ are defined as in Equations (59)-(61). In this case, $\tau[1]$, the test for $\mathcal{R}[1]$, is *ub test* and $\tau[3]$, the test for $\mathcal{R}[3]$, is *lb test*.

Note that it is always legal to set either $\tau[1]$ or $\tau[3]$ (or both) to *all tests*, so as to reduce the number of distinct code versions. We actually use this simplification in some of our methods.

A.1 Linear subscripts

In the particular case of a linear subscript function of the form $f(i) = ai + b$, the inverse function $f^{-1}(i)$ can be easily computed:

$$f^{-1}(i) = \frac{i-b}{a}. \tag{64}$$

Also, the monotonicity of $f(i)$ is determined from the value of a: if $a > 0$ then $f(i)$ is monotonically increasing, and if $a < 0$ then $f(i)$ is monotonically decreasing. Note that the values of a and b need not be known at compile time, since $\mathcal{L}$ and $\mathcal{U}$ can be efficiently computed at run-time.

A.2 Affine subscripts

Consider the d-dimensional loop nest

for $(i_1 = l_{i_1};\ i \le u_{i_1};\ i_1{+}{+})\ \{$
  for $(i_2 = l_{i_2};\ i_2 \le u_{i_2};\ i_2{+}{+})\ \{$
    $\ldots$
      for $(i_d = l_{i_d};\ i_d \le u_{i_d};\ i_d{+}{+})\ \{$
        $B(i_1, i_2, \ldots, i_d)$
      $\}$
    $\ldots$
  $\}$
$\}$

and let there be an array reference $A[f(i_1, i_2, \ldots, i_d)]$ in the body $B(i_1, i_2, \ldots, i_d)$. Let the subscript be an affine function of the form $f(i_1, i_2, \ldots, i_d) = a_1 i_1 + a_2 i_2 + \ldots + a_d i_d + b$, where $i_1, \ldots, i_d$ are the loop index variables, and $a_1, \ldots, a_d, b$ are loop invariants. At the innermost (i_d) loop the values of $i_1, \ldots, i_{d-1}$ are fixed, and f can be treated as linear on i_d. Determination of safe bounds for the $i_1, \ldots, i_{d-1}$ loops can be done using the inspector/executor method described in Section 8. Alternatively, these safe bounds can be approximated. Replacing true safe bounds $\mathcal{L}$ and $\mathcal{U}$ by approximated safe bounds $\tilde{\mathcal{L}}$ and $\tilde{\mathcal{U}}$ does not introduce any hazards as long as $\tilde{\mathcal{L}} \ge \mathcal{L}$ and $\tilde{\mathcal{U}} \le \mathcal{U}$. Techniques for approximating the iteration subspace of a loop that accesses some range of an affinely subscripted array axis are described in [23, 24].

A.3 Constant subscripts

For an array reference $A[f(i)]$ where $f(i) = k$ (a constant), $f(i)$ is neither monotonically increasing nor monotonically decreasing. Nevertheless, we can treat this special case by defining

$$\begin{aligned} \tilde{\mathcal{L}} = l \text{ and } \tilde{\mathcal{U}} = u &\quad \text{if} \quad \text{lo}(A) \le k \le \text{up}(A), \\ \tilde{\mathcal{L}} = l \text{ and } \tilde{\mathcal{U}} = l - 1 &\quad \text{if} \quad k > \text{up}(A), \\ \tilde{\mathcal{L}} = u + 1 \text{ and } \tilde{\mathcal{U}} = u &\quad \text{if} \quad k < \text{lo}(A). \end{aligned}$$

and then computing

$$\mathcal{L} = \min\left(u + 1, \max\left(l, \tilde{\mathcal{L}}\right)\right), \tag{65}$$

$$\mathcal{U} = \max\left(l - 1, \min\left(u, \tilde{\mathcal{U}}\right)\right). \tag{66}$$

(This last step is necessary to handle empty loops.) The safe region for reference $A[k]$ is either the whole iteration space, if k falls within the bounds of A, or empty otherwise. Only region $\mathcal{R}[1]$ is nonempty if k is too small, and only region $\mathcal{R}[3]$ is nonempty if k is too large. We also define $\tau[1] =$ *lb test* and $\tau[3] =$ *ub test*.

A.4 Modulo-function subscripts

Another common form of array reference is $A[f(i)]$ where $f(i) = g(i) \bmod m + ai + b$. In general, this is not a monotonic function. However, we know that the values of $f(i)$ are within the range described by $ai + b + j$, for $i = l, l + 1, \ldots, u$ and $j = 0, 1, \ldots, m - 1$. We define a function $h(i, j) = ai + b + j$. Let $h_{\max}$ be the maximum value of $h(i, j)$ in the domain $i = l, l + 1, \ldots, u$ and $j = 0, 1, \ldots, m - 1$. Let $h_{\min}$ be the minimum value of $h(i, j)$ in the same domain. These *extreme values* of $h(i, j)$ can be computed using the techniques described in [25]. Then we can define

$$\begin{aligned} \tilde{\mathcal{L}} &= \begin{cases} l & \text{if } ((\text{lo}(A) \le h_{\min}) \wedge (\text{up}(A) \ge h_{\max})), \\ u + 1 & \text{otherwise,} \end{cases} \\ \tilde{\mathcal{U}} &= u, \end{aligned}$$

and compute

$$\mathcal{L} = \min\left(u + 1, \max\left(l, \tilde{\mathcal{L}}\right)\right), \tag{67}$$

$$\mathcal{U} = \max\left(l - 1, \min\left(u, \tilde{\mathcal{U}}\right)\right). \tag{68}$$

(Again, this is necessary to handle empty loops.) That is, the safe region is the whole iteration space if we can guarantee that $g(i) \bmod m + ai + b$ is always within the bounds of A, or empty otherwise. Region $\mathcal{R}[3]$ is always empty and we make $\tau[1] =$ *all tests* to catch all violations when region $\mathcal{R}[1]$ is not empty.

If the subscript function is neither one of the described cases, then the more general inspector/executor method described in Section 8 should be used, if possible. This method also lifts the restriction on function $f(i)$ being monotonically increasing or decreasing.

B Auxiliary transformations

In Section 5 we used the transformation

$$
\boxed{\begin{array}{l}\textbf{for}\ (i=l;\ i\le u;\ i\text{++})\ \{\\ \quad B(i)\\ \}\end{array}}
\Longrightarrow
\boxed{\begin{array}{l}\textbf{for}\ (i=l;\ i\le l^s-1;\ i\text{++})\ \{\\ \quad B(\textbf{test}(i))\\ \}\\ \textbf{for}\ (i=l^s;\ i\le u^s;\ i\text{++})\ \{\\ \quad B(\textbf{notest}(i))\\ \}\\ \textbf{for}\ (i=u^s+1;\ i\le u;\ i\text{++})\ \{\\ \quad B(\textbf{test}(i))\\ \}\end{array}}
\tag{69}
$$

to partition the iteration space of a loop into three regions, using two different versions of the loop body: one with tests ($B(\textbf{test}(i))$, which appears twice) and one without tests ($B(\textbf{notest}(i))$, which appears once). The following equivalent construct contains only one instance of each loop body version:

$$
\boxed{\begin{array}{l}\textbf{for}\ (\delta=1;\ \delta\le n;\ \delta\text{++})\ \{\\ \quad \textbf{for}\ (i=l(\delta);\ i\le u(\delta);\ i\text{++})\ \{\\ \quad\quad B(\textbf{test}(i))\\ \quad \}\\ \quad \textbf{for}\ (i=l^s(\delta);\ i\le u^s(\delta);\ i\text{++})\ \{\\ \quad\quad B(\textbf{notest}(i))\\ \quad \}\\ \}\end{array}}
\tag{70}
$$

A driver loop, on index variable δ, iterates over the two instances of the loop, one with the tested body version and the other with the untested version. The bounds $l(\delta)$, $u(\delta)$, $l^s(\delta)$, and $u^s(\delta)$ for each iteration of the driver loop must be properly computed to guarantee that the semantics of the original loop are preserved. The specifications of these bounds as a function of l, u, l^s, and u^s for the most practical choices of n ($n = 2$ and $n = 3$) are shown in Figure 28.

for ($\delta = 1;\ \delta \le 3;\ \delta$++)

δ	$l(\delta)$	$u(\delta)$	$l^s(\delta)$	$u^s(\delta)$
1	l	l^s-1	1	0
2	1	0	l^s	u^s
3	u^s+1	u	1	0

for ($\delta = 1;\ \delta \le 2;\ \delta$++)

δ	$l(\delta)$	$u(\delta)$	$l^s(\delta)$	$u^s(\delta)$
1	l	l^s-1	l^s	u^s
2	u^s+1	u	1	0

Figure 28: Bounds for the two loops that appear the δ driver loop.

C Applying the general method to arbitrary loop nests

As a more complex example of the general method, consider the loop nest of Figure 29. It consists of an outermost loop i. The body of loop i contains two inner loops (j_1 and j_2) and three segments of straight-line code (S_1, S_3, and S_5). The body of loop j_1 contains two innermost loops (k_1 and k_2) and a segment of straight-line code (S_2). The body of loop j_2 also contains two innermost loops (k_3 and k_4) and a segment of straight-line code (S_4). The pseudo-code for the loop nest is shown in Figure 29(A) and a schematic representation is shown in Figure 29(B). We use the notation $i(B$ (curved braces) to represent a loop on index variable i and body B that needs bounds testing on array references indexed by i. We use the notation $i[B$ (square braces) to represent a loop on index variable i and body B that does not need testing on array references indexed by i. In the original loop, all array accesses have to be tested for valid indices, and this is represented in the diagram with curved braces for all the loops.

```
for (i = l_i; i <= u_i; i++) {
   S_1
   for (j_1 = l_{j_1}; j_1 <= u_{j_1}; j_1++) {
      for (k_1 = l_{k_1}; k_1 <= u_{k_1}; k_1++) {
         B_1
      }
      S_2
      for (k_2 = l_{k_2}; k_2 <= u_{k_2}; k_2++) {
         B_2
      }
   }
   S_3
   for (j_2 = l_{j_2}; j_2 <= u_{j_2}; j_2++) {
      for (k_3 = l_{k_3}; k_3 <= u_{k_3}; k_3++) {
         B_3
      }
      S_4
      for (k_4 = l_{k_4}; k_4 <= u_{k_4}; k_4++) {
         B_4
      }
   }
   S_5
}
```

(A)

$$
i \left(\begin{array}{l} \quad S_1 \\ j_1 \left(\begin{array}{l} k_1\,(B_1 \\ \quad S_2 \\ k_2\,(B_2 \end{array} \right. \\ \quad S_3 \\ j_2 \left(\begin{array}{l} k_3\,(B_3 \\ \quad S_4 \\ k_4\,(B_4 \end{array} \right. \\ \quad S_5 \end{array} \right.
$$

(B)

Figure 29: Example of a loop nest to be transformed.

We first apply the transformation to the four innermost loops (k_1, k_2, k_3, and k_4). Each of these loops is transformed into a sequence of three loops, with the middle one not needing tests on the loop index. These transformations are illustrated in Figure 30.

In the next step, we apply the transformation to the middle loops j_1 and j_2. Again, each of these loops is transformed into a sequence of three loops. The resulting loop nest is shown in Figure 31. For clarity, we represent each of the transformed (expanded) loops, j_1 and j_2, by $\boxed{\mathcal{E}(j_1)}$ and $\boxed{\mathcal{E}(j_2)}$ respectively.

Finally, we complete the operation by applying the transformation to the outermost i loop. This generates

$$
k_1\,(B_1 \Longrightarrow \begin{array}{l} k_1\,(B_1 \\ k_1\,[B_1 \\ k_1\,(B_1 \end{array}
\;\Bigg|\;
k_2\,(B_2 \Longrightarrow \begin{array}{l} k_2\,(B_2 \\ k_2\,[B_2 \\ k_2\,(B_2 \end{array}
\;\Bigg|\;
k_3\,(B_3 \Longrightarrow \begin{array}{l} k_3\,(B_3 \\ k_3\,[B_3 \\ k_3\,(B_3 \end{array}
\;\Bigg|\;
k_4\,(B_4 \Longrightarrow \begin{array}{l} k_4\,(B_4 \\ k_4\,[B_4 \\ k_4\,(B_4 \end{array}
$$

Figure 30: The four innermost loops transformed.

$$
j_1 \left(\begin{array}{c} k_1\,(B_1 \\ k_1\,[B_1 \\ k_1\,(B_1 \\ S_2 \\ k_2\,(B_2 \\ k_2\,[B_2 \\ k_2\,(B_2 \end{array} \right. \Rightarrow
\begin{array}{l}
j_1 \left(\begin{array}{c} k_1\,(B_1 \\ k_1\,[B_1 \\ k_1\,(B_1 \\ S_2 \\ k_2\,(B_2 \\ k_2\,[B_2 \\ k_2\,(B_2 \end{array} \right. \\
j_1 \left[\begin{array}{c} k_1\,(B_1 \\ k_1\,[B_1 \\ k_1\,(B_1 \\ S_2 \\ k_2\,(B_2 \\ k_2\,[B_2 \\ k_2\,(B_2 \end{array} \right. \\
j_1 \left(\begin{array}{c} k_1\,(B_1 \\ k_1\,[B_1 \\ k_1\,(B_1 \\ S_2 \\ k_2\,(B_2 \\ k_2\,[B_2 \\ k_2\,(B_2 \end{array} \right.
\end{array}
= \boxed{\mathcal{E}(j_1)}
\quad\Bigg|\quad
j_2 \left(\begin{array}{c} k_3\,(B_3 \\ k_3\,[B_3 \\ k_3\,(B_3 \\ S_4 \\ k_4\,(B_4 \\ k_4\,[B_4 \\ k_4\,(B_4 \end{array} \right. \Rightarrow
\begin{array}{l}
j_2 \left(\begin{array}{c} k_3\,(B_3 \\ k_3\,[B_3 \\ k_3\,(B_3 \\ S_4 \\ k_4\,(B_4 \\ k_4\,[B_4 \\ k_4\,(B_4 \end{array} \right. \\
j_2 \left[\begin{array}{c} k_3\,(B_3 \\ k_3\,[B_3 \\ k_3\,(B_3 \\ S_4 \\ k_4\,(B_4 \\ k_4\,[B_4 \\ k_4\,(B_4 \end{array} \right. \\
j_2 \left(\begin{array}{c} k_3\,(B_3 \\ k_3\,[B_3 \\ k_3\,(B_3 \\ S_4 \\ k_4\,(B_4 \\ k_4\,[B_4 \\ k_4\,(B_4 \end{array} \right.
\end{array}
= \boxed{\mathcal{E}(j_2)}
$$

Figure 31: The two middle loops transformed.

three versions of the i loop, with the middle one needing no tests on array references indexed by i. The final result is shown in Figure 32. The original loop nest of Figure 29(B) is transformed into a sequence of three i loops. Inside the middle i loop there are regions that do not need any tests. The transformed code will execute efficiently if all or almost all the iterations are executed in these regions.

$$
i\left(\begin{array}{c} S_1\\ \boxed{\mathcal{E}(j_1)}\\ S_3\\ \boxed{\mathcal{E}(j_2)}\\ S_5\end{array}\right.
\Longrightarrow
\begin{array}{l}
i\left(\begin{array}{c} S_1\\ \boxed{\mathcal{E}(j_1)}\\ S_3\\ \boxed{\mathcal{E}(j_2)}\\ S_5\end{array}\right.\\
i\left[\begin{array}{c} S_1\\ \boxed{\mathcal{E}(j_1)}\\ S_3\\ \boxed{\mathcal{E}(j_2)}\\ S_5\end{array}\right.\\
i\left(\begin{array}{c} S_1\\ \boxed{\mathcal{E}(j_1)}\\ S_3\\ \boxed{\mathcal{E}(j_2)}\\ S_5\end{array}\right.
\end{array}
$$

Figure 32: Result of applying the transformation to all loops.

D Applying the compact method to arbitrary loop nests

As a more complex example of the compact method, consider the loop nest of Figure 29. The application of the transformation to the outermost i loop is illustrated in Figure 33. It generates a driver loop around two instances of the loop nest (as described in Appendix B). In our schematic notation, driver loops are represented by curly braces ('{'). One of the instances (with the square bracket for i) does not need any tests on references indexed by i.

$$
i \left(\begin{array}{l} S_1 \\ j_1 \left(\begin{array}{c} k_1\,(B_1 \\ S_2 \\ k_2\,(B_2 \end{array} \right. \\ S_3 \\ j_2 \left(\begin{array}{c} k_3\,(B_3 \\ S_4 \\ k_4\,(B_4 \end{array} \right. \\ S_5 \end{array} \right.
\Longrightarrow
\left\{ \begin{array}{l}
i \left(\begin{array}{l} S_1 \\ j_1 \left(\begin{array}{c} k_1\,(B_1 \\ S_2 \\ k_2\,(B_2 \end{array} \right. \\ S_3 \\ j_2 \left(\begin{array}{c} k_3\,(B_3 \\ S_4 \\ k_4\,(B_4 \end{array} \right. \\ S_5 \end{array} \right. \\
i \left[\begin{array}{l} S_1 \\ j_1 \left(\begin{array}{c} k_1\,(B_1 \\ S_2 \\ k_2\,(B_2 \end{array} \right. \\ S_3 \\ j_2 \left(\begin{array}{c} k_3\,(B_3 \\ S_4 \\ k_4\,(B_4 \end{array} \right. \\ S_5 \end{array} \right.
\end{array} \right.
$$

Figure 33: Applying the transformation to the outermost loop.

The transformation can then be applied to the middle loops j_1 and j_2 in the unchecked version of the i loop. This is illustrated in Figure 34. It results in the creation of a loop nest without any tests for i or j_1 and another loop nest without any tests for i or j_2.

Finally, we apply the transformation to the innermost k_1, k_2, k_3, and k_4 loops, in the regions already without i, j_1, and j_2 tests. As illustrated in Figure 35, this creates four regions of code (the bodies of the k_1, k_2, k_3, and k_4 loops) that do not need any tests on the array references.

$$
\left\{\begin{array}{l}
i \left(\begin{array}{c}
S_1 \\
j_1 \left(\begin{array}{c} k_1\,(B_1 \\ S_2 \\ k_2\,(B_2 \end{array}\right. \\
S_3 \\
j_2 \left(\begin{array}{c} k_3\,(B_3 \\ S_4 \\ k_4\,(B_4 \end{array}\right. \\
S_5
\end{array}\right. \\
i \left[\begin{array}{c}
S_1 \\
j_1 \left(\begin{array}{c} k_1\,(B_1 \\ S_2 \\ k_2\,(B_2 \end{array}\right. \\
S_3 \\
j_2 \left(\begin{array}{c} k_3\,(B_3 \\ S_4 \\ k_4\,(B_4 \end{array}\right. \\
S_5
\end{array}\right.
\end{array}\right.
\Longrightarrow
\left\{\begin{array}{l}
i \left(\begin{array}{c}
S_1 \\
j_1 \left(\begin{array}{c} k_1\,(B_1 \\ S_2 \\ k_2\,(B_2 \end{array}\right. \\
S_3 \\
j_2 \left(\begin{array}{c} k_3\,(B_3 \\ S_4 \\ k_4\,(B_4 \end{array}\right. \\
S_5
\end{array}\right. \\
i \left[\begin{array}{c}
S_1 \\
\left\{\begin{array}{l}
j_1 \left(\begin{array}{c} k_1\,(B_1 \\ S_2 \\ k_2\,(B_2 \end{array}\right. \\
j_1 \left[\begin{array}{c} k_1\,(B_1 \\ S_2 \\ k_2\,(B_2 \end{array}\right.
\end{array}\right. \\
S_3 \\
\left\{\begin{array}{l}
j_2 \left(\begin{array}{c} k_3\,(B_3 \\ S_4 \\ k_4\,(B_4 \end{array}\right. \\
j_2 \left[\begin{array}{c} k_3\,(B_3 \\ S_4 \\ k_4\,(B_4 \end{array}\right.
\end{array}\right. \\
S_5
\end{array}\right.
\end{array}\right.
$$

Figure 34: Applying the transformation to the two middle loops.

$$
\left\{\begin{array}{l}
i\left(\begin{array}{l}
S_1\\
j_1\left(\begin{array}{l} k_1\,(B_1\\ \quad S_2\\ k_2\,(B_2\end{array}\right.\\
S_3\\
j_2\left(\begin{array}{l} k_3\,(B_3\\ \quad S_4\\ k_4\,(B_4\end{array}\right.\\
S_5
\end{array}\right.\\
i\left[\begin{array}{l}
S_1\\
\left\{\begin{array}{l}
j_1\left(\begin{array}{l} k_1\,(B_1\\ \quad S_2\\ k_2\,(B_2\end{array}\right.\\
j_1\left[\begin{array}{l} k_1\,(B_1\\ \quad S_2\\ k_2\,(B_2\end{array}\right.
\end{array}\right.\\
S_3\\
\left\{\begin{array}{l}
j_2\left(\begin{array}{l} k_3\,(B_3\\ \quad S_4\\ k_4\,(B_4\end{array}\right.\\
j_2\left[\begin{array}{l} k_3\,(B_3\\ \quad S_4\\ k_4\,(B_4\end{array}\right.
\end{array}\right.\\
S_5
\end{array}\right.
\end{array}\right.
\Longrightarrow
\left\{\begin{array}{l}
i\left(\begin{array}{l}
S_1\\
j_1\left(\begin{array}{l} k_1\,(B_1\\ \quad S_2\\ k_2\,(B_2\end{array}\right.\\
S_3\\
j_2\left(\begin{array}{l} k_3\,(B_3\\ \quad S_4\\ k_4\,(B_4\end{array}\right.\\
S_5
\end{array}\right.\\
i\left[\begin{array}{l}
S_1\\
\left\{\begin{array}{l}
j_1\left(\begin{array}{l} k_1\,(B_1\\ \quad S_2\\ k_2\,(B_2\end{array}\right.\\
j_1\left[\begin{array}{l}
\left\{\begin{array}{l} k_1\,(B_1\\ k_1\,[B_1\end{array}\right.\\
\quad S_2\\
\left\{\begin{array}{l} k_2\,(B_2\\ k_2\,[B_2\end{array}\right.
\end{array}\right.
\end{array}\right.\\
S_3\\
\left\{\begin{array}{l}
j_2\left(\begin{array}{l} k_3\,(B_3\\ \quad S_4\\ k_4\,(B_4\end{array}\right.\\
j_2\left[\begin{array}{l}
\left\{\begin{array}{l} k_3\,(B_3\\ k_3\,[B_3\end{array}\right.\\
\quad S_4\\
\left\{\begin{array}{l} k_4\,(B_4\\ k_4\,[B_4\end{array}\right.
\end{array}\right.
\end{array}\right.\\
S_5
\end{array}\right.
\end{array}\right.
$$

Figure 35: Applying the transformation to the four inner loops.

E Computing the number of regions when using the compact method

In this section we derive an expression for the number of regions a loop nest is partitioned into when using the compact method of Section 6. If we apply the compact method to loop L_1 of the loop nest shown in Equation (42), we partition the iteration space into three regions, according to Equation (41):

$$\boxed{\begin{array}{l} L_1(i_1, l_{i_1}, l^s_{i_1} - 1, L_2(i_2, l_{i_2}, u_{i_2}, \ldots, L_d(i_d, l_{i_d}, u_{i_d}, B(i_1, i_2, \ldots, i_d)) \ldots)) \\ L_1(i_1, l^s_{i_1}, u^s_{i_1}, L_2(i_2, l_{i_2}, u_{i_2}, \ldots, L_d(i_d, l_{i_d}, u_{i_d}, B(i_1, i_2, \ldots, i_d)) \ldots)) \\ L_1(i_1, u^s_{i_1} + 1, u_{i_1}, L_2(i_2, l_{i_2}, u_{i_2}, \ldots, L_d(i_d, l_{i_d}, u_{i_d}, B(i_1, i_2, \ldots, i_d)) \ldots)). \end{array}}$$

(At this point, we are not concerned with the run-time tests necessary in each region.) Applying the method recursively to the L_2 loop in the safe region of L_1 generates $2 + 3n^s_1$ regions, where $n^s_j = u^s_{i_j} - l^s_{i_j} + 1$ is the number of iterations in the safe region of loop L_j. This is shown in Figure 36.

$$\boxed{\begin{array}{ll} \multicolumn{2}{l}{L_1(i_1, l_{i_1}, l^s_{i_1} - 1, L_2(i_2, l_{i_2}, u_{i_2}, \ldots, L_d(i_d, l_{i_2}, u_{i_d}, B(i_1, i_2, \ldots, i_d)) \ldots))} \\ L_1(i_1, l^s_{i_1}, u^s_{i_1}, & \boxed{\begin{array}{l} L_2(i_2, l_{i_2}, l^s_{i_2} - 1, \ldots, L_d(i_d, l_{i_d}, u_{i_d}, B(i_1, i_2, \ldots, i_d)) \ldots) \\ L_2(i_2, l^s_{i_2}, u^s_{i_2}, \ldots, L_d(i_d, l_{i_d}, u_{i_d}, B(i_1, i_2, \ldots, i_d)) \ldots) \\ L_2(i_2, u^s_{i_2} + 1, u_{i_2}, \ldots, L_d(i_d, l_{i_2}, u_{i_d}, B(i_1, i_2, \ldots, i_d)) \ldots) \end{array}}) \\ \multicolumn{2}{l}{L_1(i_1, u^s_{i_1} + 1, u_{i_1}, L_2(i_2, l_{i_2}, u_{i_2}, \ldots, L_d(i_d, l_{i_d}, u_{i_d}, B(i_1, i_2, \ldots, i_d)) \ldots))} \end{array}}$$

Figure 36: Applying the compact method to the two outermost loops generates $2 + 3n^s_1$ regions.

In general, applying the compact method to the perfect loop nest of Equation (42) results in

$$n = 2 + \sum_{i_1 = l^s_{i_1}}^{u_{i_1}} \left(2 + \sum_{i_2 = l^s_{i_2}}^{u^s_{i_2}} \left(\ldots \left(2 + \sum_{i_{d-1} = l^s_{i_{d-1}}}^{u^s_{i_{d-1}}} 3 \right) \ldots \right) \right) \tag{71}$$

regions. The summations over i_j add the number of regions for each value of i_j in its safe region. Note that the values of $l^s_{i_j}$ and $u^s_{i_j}$ can depend on the values of $i_1, i_2, \ldots, i_{j-1}$. If the loops are *rectangular* (*i.e.*, $l^s_{i_j}$ and $u^s_{i_j}$ do not depend on the values of other index variables), then the expression for the number of regions simplifies to:

$$\begin{aligned} n &= 2 + n^s_1(2 + n^s_2(\ldots(2 + n_{d-1}3)\ldots)) \\ &= 2 + 2n^s_1 + 2n^s_1 n^s_2 + \ldots + 2n^s_1 n^s_2 \ldots n^s_{d-2} + 3n^s_1 n^s_2 \ldots n^s_{d-1} \end{aligned} \tag{72}$$

or, in more compact form:

$$n = 2\left(1 + \sum_{i=1}^{d-2} \prod_{j=1}^{i} n^s_j\right) + 3\prod_{i=1}^{d-1} n^s_i. \tag{73}$$

To make Equation (73) correct for $d = 1$ we define $\prod_{i=1}^{0} n^s_i = \frac{1}{3}$. The partitioning of a two-dimensional iteration space into regions is shown in Figure 15. Note that the regions have different characteristics as to which run-time checks have to be performed.

F Algorithm to optimize the number of regions computed for the restricted method

Figure 37 shows the algorithm for performing the optimization described in Section 7 that reduces the number of regions. The main difference, with respect to procedure *regions* of Figure 16, is when building the safe region for loop i_j.

---

procedure *regions*$(j, (\alpha_1, \alpha_2, \ldots, \alpha_{j-1}), (\omega_1, \omega_2, \ldots, \omega_{j-1}), d, \mathcal{B}, \mathcal{R}, u_\delta)$ {
  if $(l_{i_j} < l^s_{i_j})$ {
    $u_\delta = u_\delta + 1$
    $\mathcal{R}[u_\delta] = ((\alpha_1, \ldots, \alpha_{j-1}, l_{i_j}, l_{i_{j+1}}, \ldots, l_{i_d}), (\omega_1, \ldots, \omega_{j-1}, l^s_{i_j} - 1, u_{i_{j+1}}, \ldots, u_{i_d}),$ **test**$)$
  }
  if $(j == d)$ {
    $u_\delta = u_\delta + 1$
    $\mathcal{R}[u_\delta] = ((\alpha_1, \ldots, \alpha_{d-1}, l^s_{i_d}), (\omega_1, \ldots, \omega_{d-1}, u^s_{i_d}),$ **notest**$)$
  } **elsif** $($*nochecks*$((l_{i_{j+1}}, l_{i_{j+2}}, \ldots, l_{i_d}), (l^s_{i_{j+1}}, l^s_{i_{j+2}}, \ldots, l^s_{i_d}), (u_{i_{j+1}}, u_{i_{j+2}}, \ldots, u_{i_d}), (u^s_{i_{j+1}}, u^s_{i_{j+2}}, \ldots, u^s_{i_d}))$ {
    $u_\delta = u_\delta + 1$
    $\mathcal{R}[u_\delta] = ((\alpha_1, \ldots, \alpha_{j-1}, l^s_{i_j}, l_{i_{j+1}}, \ldots, l_{i_d}), (\omega_1, \ldots, \omega_{j-1}, u^s_{i_j}, u_{i_{j+1}}, \ldots, u_{i_d}),$ **notest**$)$
  } **else** {
    for $(k = l^s_{i_j};\ k \leq u^s_{i_j};\ k{+}{+})$ {
      regions$(\mathcal{R}, j + 1, (\alpha_1, \alpha_2, \ldots, \alpha_{j-1}, k), (\omega_1, \omega_2, \ldots, \omega_{j-1}, k), d, \mathcal{B}, \mathcal{R}, u_\delta)$
    }
  }
  if $(u_{i_j} > u^s_{i_j})$ {
    $u_\delta = u_\delta + 1$
    $\mathcal{R}[u_\delta] = ((\alpha_1, \ldots, \alpha_{j-1}, u^s_{i_{j+1}}, l_{i_{j+1}}, \ldots, l_{i_d}), (\omega_1, \ldots, \omega_{j-1}, u_{i_j}, u_{i_{j+1}}, \ldots, u_{i_d}),$ **test**$)$
  }
}

boolean function *nochecks*$((l_{i_1}, \ldots, l_{i_m}), (l^s_{i_1}, \ldots, l^s_{i_m}), (u_{i_1}, \ldots, u_{i_m}), (u^s_{i_1}, \ldots, u^s_{i_m}))$ {
  if $(((l_{i_j} = l^s_{i_j}) \wedge (u_{i_j} = u^s_{i_j}))\forall j = 1, \ldots, m)$
    return true
  else
    return false
}

Figure 37: Optimized procedure to compute the regions for a loop nest.

In Figure 37, instead of directly applying procedure *regions* to each iteration of the safe region i_j, a test is performed using function *nochecks*. The test verifies whether the safe lower and upper bounds of all loops inside loop i_j (loops $i_{j+1}, i_{j+2}, \ldots, i_d$) are equal to the corresponding full bounds. If that is the case (function *nochecks* returns **true**), then a single multidimensional safe region can be created at this point. If function *nochecks* returns **false**, then procedure *regions* is applied recursively as in Figure 16. We also put guards to generate the regions preceding and succeeding the safe region of loop i_j only if they are nonempty.

References

[1] James Gosling, Bill Joy, and Guy Steele. *The Java(TM) Language Specification*. Addison-Wesley, 1996.

[2] James Gosling. The evolution of numerical computing in Java. Document available at Web-page `http://java.sun.com/people/jag/FP.html`. Sun Microsystems.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1985.

[4] V. Seshadri. IBM high performance compiler for Java. *AIXpert Magazine*, September 1997. Electronic publication available at URL `http://www.developer.ibm.com/library/aixpert`.

[5] A. Krall and R. Grafl. CACAO – A 64-bit JavaVM just in time compiler. *Concurrency, Pract. Exp.*, 9(11):1017–30, Nov 1997. Java for Computational Science and Engineering - Simulation and Modeling II Las Vegas, NV, USA 21 June 1997.

[6] 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.

[7] D. Baxter, R. Mirchandaney, and J. H. Saltz. Run-time parallelization and scheduling of loops. In *Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures*, pages 303–312, 1989.

[8] S. P. Midkiff, J. E. Moreira, and M. Gupta. Method for optimizing array bounds checks in programs. Patent pending, IBM Docket #YO-998-052, filed with U. S. Patent Office, April 24th, 1998.

[9] Bruce Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, Inc, 1994.

[10] B. Schwarz, W. Kirchgassner, and R. Landwehr. An optimizer for Ada – design, experience and results. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 175–185, June 1988.

[11] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4'th ACM Symposium on Principles of Programming Languages*, pages 238–252, January 1977.

[12] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the 5'th ACM Symposium on Principles of Programming Languages*, pages 84–96, January 1978.

[13] P. Cousot and N. Halbwachs. Automatic proofs of the absence of common runtime errors. In *Conference Record of the 5'th ACM Symposium on Principles of Programming Languages*, pages 105–118, January 1978.

[14] W.H. Harrison. Compiler analysis for the value ranges for variables. *IEEE Transactions on Software Engineering*, SE3(3):243–250, May 1977.

[15] J.M. Asuru. Optimization of array subscript range checks. *ACM Letters on Programming Languages and Systems*, 1(2):109–118, June 1992.

[16] R. Gupta. A fresh look at optimizing array bounds checking. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 272–282, June 1990.

[17] R. Gupta. Optimizing array bound checks using flow analysis. *ACM Letters on Programming Languages and Systems*, 2(1-4):135–150, March–December 1993.

[18] P. Kolte and M. Wolfe. Elimination of redundant array subscript range checks. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 270–278, Jun 1995.

[19] V. Markstein, J. Cocke, and P. Markstein. Elimination of redundant array subscript range checks. In *Proceedings of the ACM SIGPLAN '82 Conference on Programming Language Design and Implementation*, pages 114–119, June 1982.

[20] Intel Corporation. *Pentium Processor Family Developer's Manual, Volume 3: Architecture and Programming Manual*, 1995.

[21] International Business Machines Corporation. *AIX Version 3.2 Assembler Language Reference*, third edition, October 1993.

[22] International Business Machines Corporation. Java(TM) JIT compiler project home page. Available at URL `http://www.trl.ibm.co.jp/projects/s7210/java_jit/index_e.htm`.

[23] S. P. Midkiff. Computing the local iteration set of a block-cyclically distributed reference with affine subscripts. In *Sixth Workshop on Compilers for Parallel Computing*, 1996.

[24] K. van Reeuwijk, W. Denissen, H. J. Sips, and E. M. R. M. Paalvast. An implementation framework for HPF distributed arrays on message-passing parallel computer systems. *IEEE Transactions on Parallel and Distributed Systems*, 7(9):897–914, Sep 1996.

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