Subregion Analysis and Bounds Check Elimination for High Level Arrays

Mackale Joyner[1], Zoran Budimlić[2], and Vivek Sarkar[2]

[1] Texas Instruments, Dallas, TX 75243
[2] Rice University, Houston, TX 77054

Abstract. For decades, the design and implementation of arrays in programming languages has reflected a natural tension between productivity and performance. Recently introduced HPCS languages (Chapel, Fortress and X10) advocate the use of *high-level arrays* for improved productivity. For example, high-level arrays in the X10 language support rank-independent specification of multidimensional loop and array computations using *regions* and *points*. Three aspects of X10 high-level arrays are important for productivity but pose significant performance challenges: high-level accesses are performed through point objects rather than integer indices, variables containing references to arrays are rank-independent, and all subscripts in a high-level array access must be checked for bounds violations. The first two challenges have been addressed in past work. In this paper, we address the third challenge of optimizing the overhead of array bounds checks by developing a novel *region-based* interprocedural array bounds analysis to automatically identify redundant checks. Elimination of redundant checks reduces the runtime overhead of bounds checks, and also enables further optimization by removing constraints that arise from precise exception semantics. We have implemented an array bounds check elimination algorithm that inserts special annotations that are recognized by a modified JVM.
We also introduce *array views*, a high-level construct that improves productivity by allowing the programmer to access the underlying array through multiple views. We describe a technique for optimizing away the overhead of many common cases of array views in X10. Our experiments show that eliminating bounds checks using the results of the analysis described in this paper improves the performance of our benchmarks by up to 22% over JIT compilation.

1 Introduction

Since the dawn of computing, arrays have played an important role in programming languages as a central data structure used by application and library developers. However, the design and implementation of array operations have been subject to a natural tension between productivity and performance. Languages such as APL and MATLAB have demonstrated the productivity benefits of *high-level arrays* that support a powerful set of array operations, but the usage of high-level arrays is typically restricted to prototyping languages because it has proved very challenging to deliver production-strength performance for these operations. Conversely, languages such as C include *low-level arrays* with a very restricted set of array operations (primarily, subscripting to access

individual array elements) that are amenable to efficient production-strength implementations. Some languages, such as FORTRAN 90 and its successors, augment low-level arrays with a few high-level operations that operate on entire arrays; in some cases, efficient code can be generated through optimized *scalarization* [30] for these high-level operations, but it is often necessary for users of these languages to replace high-level array operations by low-level equivalents when hand-tuning their code.

Chapel, Fortress and X10, initially developed within DARPA's High Productivity Computing System (HPCS) program, are all parallel high-level object-oriented languages designed to deliver both high productivity and high performance. These languages offer abstractions that enable programmers to develop applications for parallel environments without having to explicitly manage many of the details encountered in low level parallel programming. Unfortunately, runtime performance usually suffers when programmers use early implementations of these languages. Compiler optimizations are crucial to reducing performance penalties resulting from their abstractions.

For example, high-level arrays in the X10 language support rank-independent specification of multidimensional loop and array computations using *regions* and *points*. Three aspects of X10 high-level arrays are important for productivity but also pose significant performance challenges: high-level accesses are performed through point objects instead of integer indices, variables containing references to arrays are rank-independent, and all high-level array accesses must be checked for bounds violations.

The first two challenges have been addressed in the past [17, 18]. In this paper, we address the optimizing the overhead of array bounds checks by developing a novel *region-based* interprocedural array bounds analysis to automatically identify redundant checks. Elimination of redundant checks reduces the runtime overhead of bounds checks, and also enables further optimization by removing constraints that arise from precise exception semantics. A single high-level array access may result in multiple bounds checks, one per dimension in general. We have implemented an array bounds check elimination algorithm that eliminates these per-dimension checks from the X10 program when legal to do so, and inserts special annotations to enable the underlying JVM to eliminate the last remaining bounds check for the underlying linearized array. An unusual aspect of our analysis is the use of *must* information (the sub-region relationship) to eliminate bounds checks, while most of the traditional analysis use *may* information for bounds check elimination.

We also introduce *array views*, a high-level construct that improves productivity by allowing the programmer to access the underlying array through multiple views. We present a method for optimizing away the overhead of some common cases of array views in X10. Our experiments show that a bounds check elimination optimization alone using the results of the analysis described in this paper improves the performance of our benchmarks by up to 22% over JIT compilation and approaches the performance of the code in which all runtime checks (bounds checks, null checks, cast checks, etc...) are turned off.

2 Points, Regions and Arrays

High level arrays are embodied in a number of languages including the recent HPCS languages and earlier languages such as Titanium [29]. Both Chapel and X10 build

on ZPL's foundational concepts of *points* and *regions* [24]. In this paper, we focus on X10's embodiment of high level arrays.

A *point* is an element of an n-dimensional Cartesian space ($n \geq 1$) with integer-valued coordinates, where n is the *rank* of the point. A *region* is a set of points, and can be used to specify an array allocation or iteration constructs such as point-wise sequential and parallel loops. For instance, the region `[0:200,1:100]` specifies a collection of two-dimensional points `(i,j)` with `i` ranging from `0` to `200` and `j` ranging from `1` to `100`.

X10's high level arrays can be used to express rank-independent loop and array computations as shown in Figure 1[3]. For simplicity, an additional loop is introduced to compute the weighted sum using the elements in the stencil, but this loop could be replaced by a high level array `sum()` operation as well. Note that the code in this example can be executed on arrays with different ranks, by setting `R_inner` and `stencil` appropriately.

```
region R_inner = ... ; // Inner region
region stencil = ... ; // Set of points in stencil
double omega_factor = ...;// Weight used for stencil points
for (int p=0; p<num_iterations; p++) {
  for (point t : R_inner) { //
    double sum = one_minus_omega * G[t];
    for (point s : stencil) sum += omega_factor * G[t+s];
    G[t] = sum;
  }
}
```

Fig. 1. Rank-independent version of Java Grande SOR benchmark

Points and regions are first-class value types — a programmer can declare variables and create expressions of these types using the operations listed in Figure 2 — in X10 [8]. In addition, X10 supports a special syntax for point construction — the expression, "`[a,b,c]`", is implicit syntax for a call to a three-dimensional point constructor, "`point.factory(a,b,c)`" — and also for variable declarations. The declaration, "`point p[i,j]`" is exploded syntax for declaring a two-dimensional point variable `p` along with integer variables `i` and `j` which correspond to the first and second elements of `p`. Further, by requiring that points and regions be value types, the X10 language ensures that individual elements of a point or a region cannot be modified after construction.

A summary of array operations in X10 can be found in Figure 2. Note that the X10 array allocation expression, "`new double[R]`", directly allocates a multi-dimensional array specified by region `R`. The region associated with an array is available through the `.region` field. In its full generality, an array allocation expression in X10 takes a *distribution* instead of region. However, we will ignore distributions in this paper and limit our attention to single-place executions although it is straightforward to extend the subregion analysis and bounds check elimination algorithms in this paper to handle distributed arrays.

[3] This paper uses the old Java-based syntax from X10 v1.5 [8]. The latest version of X10 has a different syntax and an updated type system, but retains the basic structure of high level arrays from v1.5.

Region operations:

```
R.rank ::= # dimensions in region;
R.size() ::= # points in region
R.contains(P) ::= predicate if region R contains point P
R.contains(S) ::= predicate if region R contains region S
R.equal(S) ::= true if region R and S contain same set of points
R.rank(i) ::= projection of region R on dimension i
R.rank(i).low() ::= lower bound of i-th dimension of region R
R.rank(i).high() ::= upper bound of i-th dimension of region R
R.ordinal(P) ::= ordinal value of point P in region R
R.coord(N) ::= point in region R with ordinal value = N
R1 && R2 ::= region intersection
R1 || R2 ::= union of regions R1 and R2
R1 - R2 ::= region difference
```

Array operations:

```
A.rank    ::= # dimensions in array
A.region ::= index region (domain) of array
A[P] ::= element at point P, where P belongs to A.region
A | R ::= restriction of array onto region R
A.sum(), A.max() ::= sum/max of elements in array
A1 <op> A2 ::= result of applying point-wise op on A1 and A2,
               when A1.region = A2. region
               (<op> can include +, -, *,  and / )
A1 || A2 ::= disjoint union of arrays A1 and A2
             (A1.region and A2.region must be disjoint)
A1.overlay(A2) ::= array with region, A1.region || A2.region,
                   with element value A2[P] for all points P in
                   A2.region and A1[P] otherwise.
```

Fig. 2. Region and array operations in X10

3 Region Analysis

3.1 Intraprocedural Region Analysis

Our static bounds analysis first runs a local pass over each method after we translate the code into a static single assignment (SSA) form. Using a dominator based value numbering technique [7], we assign value numbers to each point, region, array, and array access inside the method body. These value numbers represent region association. Upon completion of local bounds analysis, we map region value numbers back to the source using the source code position as the unique id. Algorithm 1 shows the algorithm for the intraprocedural region analysis.

To perform the analysis and transformation techniques described above, we use the Matlab D framework developed at Rice University [9, 11]. We generate an XML file from the AST of the X10 program, then read this AST within the Matlab D compiler, convert it into SSA, perform the value numbering based algorithms presented in this chapter to infer the regions associated with arrays, points and regions in the program, then use the unique source code position to map the analysis information back into the X10 compiler.

We build both array region and value region relationships during the local analysis pass. Discovering an array's range of values exposes additional code optimization opportunities. Barik and Sarkar's [4] enhanced bit-aware register allocation strategy uses array value ranges to precisely determine how many bits a scalar variable requires when it is assigned the value of an array element. In the absence of sparse data structures [19], sparse matrices in languages like Fortran, C, and Java are often represented by a set of

1-D arrays that identify the indices of non-zero values in the matrix. This representation usually inhibits standard array bounds elimination analysis because array accesses often appear in the code with subscripts that are themselves array accesses. We employ value range analysis to infer value ranges for arrays. Specifically, our array value range analysis tracks all assignments to array elements. We ascertain that when program execution assigns an array's element a value using the *mod* function, a loop induction variable, a constant, or array element value, we can analyze the assignment and establish the bounds for the array's element value range. [4]

In Figure 3, assuming that the assignment of array values for *row* is the only *row* update, analysis will conclude that *row*'s value region is *reg1*. Our static bounds analysis establishes this value region relationship because the *mod* function inherently builds the region [0:*reg1.high()*]. Figure 4 shows this analysis code view update for array element assignments to *row* and *col*.

```
//code fragment is used to highlight
//interprocedural array element value
//range analysis
...
region reg1 = [0:dm[size]-1];
region reg2 = [0:dn[size]-1];
region reg3 = [0:dp[size]-1];
double[.] x = randVec(reg2);
double[.] y = new double[reg1];
int[.] val = new double[reg3]
int[.] col = new double[reg3];
int[.] row = new double[reg3];
Random R;...
for (point p1 : reg3) {
  //array row has index set in reg3 and value range in reg1
  row[p1] = Math.abs(R.Int()) % (reg1.high()+1);
  col[p1] = Math.abs(R.Int()) % (reg2.high()+1);...
}
kernel(x,y,val,col,row,..);

double[.] randVec(region r1){
   double[.] a = new double[r1];
   for (point p2: r1)
     a[p2] = R.double();
    return a;
}

kernel(double[.]x,double[.]y,int[.]val,int[.]col,int[.]row,..){
   for (point p3 : col)
     y[row[p3]]+= x[col[p3]]*val[p3];
}
```

Fig. 3. Java Grande Sparse Matrix Multiplication (source view).

We use an implicit, infinitely wide type lattice to propagate the values of the regions through the program. The lattice is shown on Figure 5. In the Matlab D compiler [11], a ϕ function performs a *meet* operation $(\wedge)$ of all its arguments, where each argument is a lattice value, and assigns the result to the target of the assignment.

[4] Note: when array *a1* is an alias of array *a2* (e.g. via an array assignment), we assign both *a1* and *a2* a value range of $\perp$, even if *a1* and *a2* share the same value range, in order to eliminate the need for interprocedural alias analysis. In the future, value range alias analysis can be added to handle this case.

```
//code fragment is used to highlight
//interprocedural array element value
//range analysis
...
reg1 = [0:dm[size]-1];
reg2 = [0:dn[size]-1];
reg3 = [0:dp[size]-1];
x = reg2; //replaced call with region argument reg2
y = reg1;
col = reg3;
row = reg3;
Random R;...
p1 = reg3; //replaced \Xten{} loop with assignment to p1
row[p1] = [0:reg1.high()]; //followed by loop body
col[p1] = [0:reg2.high()];//created value range from mod
kernel(x,y,col,row,..);
...
region randVec(region r1){
  a = r1;
  p2 = r1; //replaced \Xten{} loop with assignment to p2
  a[p2] = R.double(); //followed by loop body
  return r1; //returns formal parameter
}
kernel(double[.]x,double[.]y,int[.]col,int[.] row,..){
  p3 = col; //replaced \Xten{} loop with assignment to p3
  y[row[p3]]+= x[col[p3]]*val[p3]; //followed by loop body
}
```

Fig. 4. Java Grande Sparse Matrix Multiplication (analysis view).

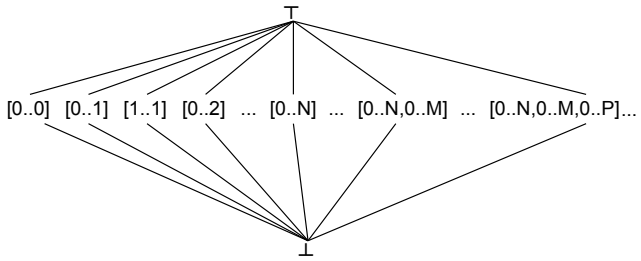

Fig. 5. Type lattice for region equivalence

3.2 Interprocedural Region Analysis

If a method returns an expression with a value number that is the same as a formal parameter value number, analysis will give the array assigned the result of the method call the value number of the corresponding actual argument at the call site.

Static interprocedural analysis commences once intraprocedural analysis completes. During program analysis, we work over two different views of the code. The first is the standard source view which affects code generation. The second is the analysis view. Changes to the analysis view of the code do not impact code generation. In Figure 3,

Algorithm 1: Intraprocedural region analysis algorithm builds local region relationships.

Input: CFG of X10 program
Output: $regmap$, a local mapping of each variable with type X10 array, region or point to its region value number

begin
  `// initialization`
  foreach $n \in$ *Region, Point, Array* **do**
    $regmap(n) = \top$;
  `// infer X10 region mapping`
  foreach $a \in$ *assign* **do**
    if $a \in \phi$ *function* **then**
      $regmap(a.def) \leftarrow \bigwedge_{i=0}^{a.numargs} regmap(a.arg(i))$
    else if *a.rhs* $\in$ *constant* **then**
      regmap(a.lhs) = a.rhs
    else
      regmap(a.lhs) = regmap(a.rhs)

program execution assigns array *x* the result of invoking method *RandomVector*. Because our analysis determines that the method will return the region the program passes as an argument (assuming region has a lower bound of 0), we will modify the analysis view by replacing the method call with an assignment to the argument (*reg2* in our example). Figure 4 shows this update. When encountering method calls which our interprocedural regions analysis is not currently analyzing, we assign each formal argument to the actual argument if and only if the actual argument has a region association. Each actual argument can have one of the following three region states:

- If the method argument is a X10 array, region, or point, then the argument will be in the full region state.
- The method argument has a partial region state when it represents the high or low bound of a linear region.
- If the method argument does not fall within the first two cases, then we assign $\perp$ to the argument (no region association). This distinction minimizes the number of variables that we need to track during region analysis.

In addition to analyzing the code to detect region equivalence, we augment the analysis with extensions to support sub-region relationships. Inferring sub-region relationships between arrays, regions and points is similar in structure to region equivalence inference analysis, but is different enough to warrant a separate discussion. As with the interprocedural region equivalence analysis, there is an implicit type lattice, but this time the lattice is unbounded in height as well as in width. The lattice is defined as follows:

- There $\exists$ an edge between regions A and B in the lattice *L iff* the two regions are of the same dimensionality and region A is completely contained within region B.
- The lattice *L* meet operator $\wedge$ is idempotent, commutative, and associative for all $i \in L$.
- Given the lattice *L* and $i, j \in L$, $i < j$ iff $i \wedge j = i$ and $i \neq j$

During our analysis, we compute on demand an approximation of the sub-region lattice relation — if we cannot prove that a region A is a sub-region of region B, then $A \wedge B = \perp$. It is important to point out that, even though the subregion lattice is unbounded in height and width, this does not adversely affect the running time or correctness of our algorithm. The lattice is implicit — it is never completely computed or traversed in the algorithm. Checking whether there is a sub-region relation between two regions takes constant amount of time, regardless of their position in the lattice.

In addition, our analysis is flow-insensitive for global variables. When static analysis determines that a global variable might be involved in multiple region assignments involving different regions, the region for the variable becomes $\perp$. In the future, we can extend the algorithm to assign the variable the region intersection instead of $\perp$. Algorithm 2 presents pseudo code for the static interprocedural region analysis algorithm. The interprocedural region analysis algorithm can be implemented to run in $O(|V|+|E|)$ time for a graph G, where V is the number of array, point, and region variables in the whole program and E is the number of edges between them. An edge exists between two variables if one defines the other. Theorem 1 shows that this algorithm has complexity $O(|V| + |E|)$ and preserves program correctness:

Definition 1. *Given a program P, let T be the set containing point, region and array types in P and N be the set of variables in P with type t ∈ T such that for all m ∈ N:*

(1) DEF(m) is the set of variables in P defined by m.

(2) REG(i) is the region associated with i. There ∃ precise region for i iff i ∈ V and REG(i) ≠ ⊤ or ⊥.

Definition 2. *Given a directed graph G where V is the set of program variables of type array or region, there exists an edge E between i, j ∈ V where i is the source and j is the sink iff j ∈ DEF(i).*

Theorem 1. *The region analysis algorithm runs in time O(V+E) and preserves program correctness.*

Proof. Initially each node $n \in$ V is placed on the worklist with lattice value $\top$. Once node *n* is taken off the worklist, *n* can only be put back on the list iff $n \in$ DEF(m) and $m < n$ or there $\exists$ precise regions for both *n* and *m* and REG(n) $\neq$ REG(m). In the latter case $n \leftarrow \perp$ before we place *n* back on the worklist. Since the lattice is bounded, all regions in the lattice have finite size k, and a node *n* can only have its lattice value lowered, each node can only be placed on the worklist a maximum of k+2 times. Because we traverse source node edges when lattice value changes, each edge will be traversed a maximum of k+1 times. Therefore, because *V* is a finite set of nodes, the algorithm must eventually halt. Since each node *n* is placed on the worklist a maximum of k+2 times and its edges are traversed a maximum of k+1 times, the complexity is *O*(*V*+*E*). Assuming the whole

program is available to the region analysis algorithm, the algorithm preserves program correctness. The region algorithm will produce an incorrect program iff the algorithm assigns an incorrect precise region to a program variable with type array or region. This would only occur when the variable can have multiple regions. However, when a variable has multiple regions, the region analysis algorithm assigns the variable $\perp$. Therefore, the region analysis algorithm produces a correct program.

Algorithm 2: Interprocedural region analysis, maps variables of type X10 array, point, and region to a concrete region.

```
Input: X10 program
Output: regmap, a mapping of each variable with type X10 array, region or point
        to its region
begin
    // initialization
    worklist = ∅, def = ∅;
    foreach n ∈ Region, Point, Array do
        regmap(n) = ⊤;
        worklist = worklist + n
    foreach assign a do
        if a.rhs ∈ constant then
            regmap(a.lhs) = a.rhs;
            use(a.rhs) = use(a.rhs) ∪ a.lhs
    foreach call arg c → param f do
        if c ∈ constant then
            regmap(f) = c;
            use(c) = use(c) ∪ f
    // infer X10 region mapping
    while worklist ≠ ∅ do
        worklist = worklist - n;
        foreach v ∈ use(n) do
            if regmap(n) < regmap(v) in lattice then
                regmap(v) = regmap(n);
                worklist = worklist + v;
            else if regmap(n) ≱ regmap(v) in lattice then
                regmap(v) = ⊥;
                worklist = worklist + v;
```

3.3 Rectangular Region Algebra

In this paper, we refer to sub-arrays (products of intervals) as *regions*, as it has been the common practice in recent literature. This is in contrast to the *region* concept originally

introduced by Triolet et al. [27] to describe convex sets of array elements, which are strictly more powerful.

Often in scientific codes, loops iterate over the interior points of an array. If through static analysis we can prove that loops are iterating over sub-regions of an array, we can identify the bounds checks for those array references as superfluous. We use the example on Figure 6 to highlight the benefits of employing region algebra to build variable region relationships. Algorithm 3 shows the algorithm for region algebra analysis.

When our static region analysis encounters the *dgefa* method call with a region high bound argument in Figure 6, analysis will assign *dgefa*'s formal parameter *n* the high bound of *region1*'s second dimension and *a* the region *region1*. We shall henceforth refer to the region representing *region1*'s second dimension as *region1_2dim*. Inside *dgefa*'s method body, analysis will categorize *nm1* as a region bound and *region3* as a sub-region of *region1_2dim* when inserting it in the region tree.

Next, we assign array *col_k* the region *region1_2dim* and categorize *kp1* as a sub-region of *region1_2dim*. When static region analysis examines the binary expression *n-kp1* on the right hand side of the assignment to *var1*, it discovers that the *n* is *region1_2dim.hbound()* and kp1 is a sub region of *region1_2dim*. As a result, we can use region algebra to prove that this region operation will return a region *r* where: *r.lbound()* $\geq$ *region1_2dim.lbound()* and *r.bound()* $\leq$ *region1_2dim.hbound()*. Consequently, *var1* will be assigned *region1_2dim*.

Finally, analysis determines that *var2*'s region is a sub-region of *region1_2dim*. As a result, when analysis encounters the *daxpy* call it will assign *daxpy* formal parameter *dx* the region *region1_2dim* and formal parameter *dax_reg* the same region as *var2* enabling us to prove and signal to the VM that the bounds check for the array access *dx[p2]* in *daxpy*'s method body is unnecessary.

```
//code fragment is used to highlight
//interprocedural region analysis using region algebra
int n = dsizes[size];
int ldaa = n;
int lda = ldaa + 1;
...
region region1 = [0:ldaa-1,0:lda-1];...
double[.] a = new double[region1]...
info = dgefa(a, region1.rank(1).high(), ipvt);
//dgefa method, lufact kernel
int dgefa(double[.] a, int n, int[.] ipvt){...
  nm1 = n - 1;...
  region region3 = [0:nm1-1];...
  for (point p1[k] : region3) {
    col_k = RowView(a,k);...
     kp1 = k + 1...
     int var1 = n-kp1;
     region var2 = [kp1:n];...
     daxpy(var1,col_k,kp1,var2,...);...
  }
}
...
//daxpy method
void daxpy(int n,double[.]dx,int dx_off,region dax_reg,..){..
  for (point p2 : dax_reg)
    dy[p2]+= da*dx[p2];...
}
```

Fig. 6. Java Grande LU factorization kernel.

Algorithm 3: Region algebra algorithm discovers integers and points that have a region association.

```
Input: X10 program
Output: regAssoc, a mapping of each variable of type X10 array, region, point or
        int to its region association
begin
    // initialization
    worklist = ∅, def = ∅;
    foreach n ∈ Region, Point, Array, int do
        regAssoc(n) = ⊤;
        worklist = worklist + n
    foreach assign a do
        if a.rhs ∈ constant ∨ bound then
            regAssoc(a.lhs) = a.rhs;
            use(a.rhs) = use(a.rhs) ∪ a.lhs
    foreach call arg c → param f do
        if c ∈ constant ∨ bound then
            regAssoc(f) = a.rhs;
            use(c) = use(c) ∪ f
    // infer X10 region mapping
    while worklist ≠ ∅ do
        worklist = worklist − n;
        foreach v ∈ use(n) do
            if regAssoc(n) < regAssoc(v) in lattice then
                regAssoc(v) = regAssoc(n);
                worklist = worklist + v;
            else if regAssoc(n) ≱ regAssoc(v) in lattice then
                regAssoc(v) = ⊥;
                worklist = worklist + v;
```

3.4 Interprocedural Linearized Array Bounds Analysis

Our array bounds analysis algorithm as described in Section 3.1 and Section 3.2 makes heavy use of X10 points and regions to discover when bounds checks are superfluous. In general, the programmer iterates through the elements in an array by implementing an X10 *for* loop whose header contains both a point declaration *p1* and the region *r1* containing the set of points defining *p1*. As a result, when encountering an array access with subscript *p1*, if our array bounds analysis can establish a subset relationship between the array's region and region *r1*, our analysis can signal the VM that a bounds check for this array access is superfluous.

Figure 7 illustrates an MG code fragment where the application developer linearizes a 3-dimensional array to boost runtime performance. This example shows why our current array bounds analysis cannot rely on the compiler automatically converting lin-

earized arrays to X10 multi-dimensional arrays because the range for each dimension in this case cannot be established. As a result, our bounds analysis must be extended if we want to analyze linearized array accesses to discover useless bound checks. Figure 7 highlights another extension to the array bounds analysis we previously described in Section 3.1 and Section 3.2. Studying the MG code fragment reveals that all the accesses to array *r* inside method *psinv* are redundant. Our array bounds analysis adds the following requirements to prove that *r's* bounds checks are redundant:

- The array region summary for *psinv's* formal parameter *r* is a subset of the region summary for *zero3's* formal parameter *z* . The region summary for a given array and procedure defines the valid array index space inside the procedure for which a bounds check is useless. The region summary contains only an index set that must execute when the programmer invokes this method. We do not include array accesses occurring inside conditional statements in the region summary.
- The region representing the actual argument of *psinv's* formal parameter *r* is a subset of the region representing the actual argument for *zero3's* formal parameter *z*.
- The program must call *zero3* before calling *psinv*.
- Since our analysis modifies *psinv's* actual method body, the previous requirements must hold on all calls to *psinv*.

These requirements enable our interprocedural region analysis to de-linearize array accesses into region summaries and to propagate the region summary information to discover redundant bounds checks. Note: The algorithm does not reanalyze recursive functions in a call chain. The algorithm can be extended to take advantage of recursion by splitting the recursive function *A* into *A* and *A'* and subsequently eliminating the checks in *A'*.

4 Bounds Check Elimination

The results of the interprocedural region analysis described in Section 3 can be used in different contexts in an optimizing compiler. In this section, we describe how the subregion relationship computed in Section 3 can be used to eliminate unnecessary array bounds checks in an X10 program.

Many high-level languages perform automatic array bounds checking to improve both safety and correctness of the code, by eliminating the possibility of an incorrect (or malicious) code randomly "poking" into memory through an out of bounds array access or buffer overflow. While these checks are beneficial for safety and correctness, performing them at run time can significantly degrade performance especially in array-intensive codes. Two main ways that bounds checks can affect performance are:

1. *The Cost of Checks*. The runtime may need to check the array bounds when program execution encounters an array access.
2. *Constraining Optimizations*. The compiler may be forced to constrain or disable code optimizations in code region containing checks, in the presence of precise exception semantics.

```
//MG code fragment highlights opportunity to eliminate
//bound checks with procedure array bound summaries
int nm = 2+(1<<lm);
int nv = (2+(1<<ndim1))*(2+(1<<ndim2))*(2+(1<<ndim3));
int nr = (8*(nv+nm*nm+5*nm+7*lm))/7;
region reg_nr = [0:nr-1];
double[.]u=new double[reg_nr]; //create linearized array
zero3(u, 0, n1, n2, n3);
psinv(u, 0, n1, n2, n3);
...
void zero3(double[.] z, int off, int n1, int n2, int n3) {
   for (point p1[i3,i2,i1]: [0:n3-1,0:n2-1,0:n1-1])
      z[off+i1+n1*(i2+n2*i3)] = 0.0;
}...
void psinv(double[.] r, int roff, int n1, int n2, int n3) {...
   for (point p40[i3,i2]: [1:n3-2,1:n2-2]) {
      for (point p41[i1] : [0:n1-1]) {
         r1[p41] = r[roff+i1+n1*(i2-1+n2*i3)]
            + r[roff+i1+n1*(i2+1+n2*i3)]
            + r[roff+i1+n1*(i2+n2*(i3-1))]
            + r[roff+i1+n1*(i2+n2*(i3+1))];
         r2[p41] = r[roff+i1+n1*(i2-1+n2*(i3-1))]
            + r[roff+i1+n1*(i2+1+n2*(i3-1))]
            + r[roff+i1+n1*(i2-1+n2*(i3+1))]
            + r[roff+i1+n1*(i2+1+n2*(i3+1))];
      }...
   }}
```

Fig. 7. This MG code fragment shows an opportunity to remove all array *r* bounds checks inside the *psinv* method because those checks are all redundant since the programmer must invoke method *zero3* prior to method *psinv*.

In our approach, we insert a special *noBoundsCheck* annotation [5] around an array subscript to signal to a modified version of the IBM J9 Java Virtual Machine [6] that it can skip the array bounds check for that particular array access. These annotations can be inserted if the compiler can establish one of the following properties:

1. *Array Subscript within Region Bound*. If the array subscript is a point that the programmer is using to iterate through region *r1* and *r1* is a subregion of the array's region, then the bounds check is unnecessary.
2. *Subscript Equivalence*. Given two array accesses, one with array *a1* and subscript *s1* and the second with array *a2* and subscript *s2*: if subscript *s1* has the same value number as *s2*, *s1* executes before subscript *s2* and array *a1*'s region is a subregion of *a2*'s region, then the bounds check for *a2[s2]* is unnecessary.

We use a dominator-based value numbering technique [7] to find redundant array accesses. We annotate each array access in the source code with two value numbers. The first value number represents a value number for the array access. We derive a value number for the array access by combining the value numbers of the array reference and the subscript. The second value number represents the array's element value range. By maintaining a history of these array access value numbers we can discover redundant array accesses and eliminate them.

[5] The "annotation" is simply a call to an empty *noBoundsCheck* method that is recognized by the JVM as a *NOP* and causes the JVM to skip the bounds check for the enclosed subscript

[6] Any JVM can be extended to recognize the noBoundsCheck annotation, but in this paper we report our experiences with a version of the IBM J9 JVM that was modified with this capability.

5 Array Views

```
//code fragment highlights array view productivity benefit
region reg_mex = [0:MESH_EXT-1];
region reg_mex_linear=[0:MESH_EXT*MESH_EXT*MESH_EXT-1];
double[.] x = new double[reg_mex_linear-];
double[.] y = new double[reg_mex_linear];
double[.] z = new double[reg_mex_linear];...
for (point pt3[pz] : reg_mex) {...
  for (point pt2[py] : reg_mex) {...
    for (point pt1[px] : reg_mex) {
      //using less productive linearized array access
      x[px+MESH_EXT*(py + MESH_EXT*pz)] = tx;
      y[px+MESH_EXT*(py + MESH_EXT*pz)] = ty;
      z[px+MESH_EXT*(py + MESH_EXT*pz)] = tz;
      tx += ds;
    }
    ty += ds;
  }
  tz += ds;
}...
region reg_br = [0:MESH_EXT-2];
region reg_br_3D = [reg_br, reg_br, reg_br];
int[.] p1,p2 = new int[reg_br_3D];...
//would be invalid if x, y, and z were 3-D arrays
for (point pt7 : reg_br_3D) {
  ux = x[p2[pt7]] - x[p1[pt7]];
  uy = y[p2[pt7]] - y[p1[pt7]];
  uz = z[p2[pt7]] - z[p1[pt7]]; ...
}
```

Fig. 8. Hexahedral cells code showing that problems arise when representing arrays *x*, *y*, and *z* as 3-dimensional arrays due to programmers indexing into these arrays using an array access returning integer value instead of a triplet.

Though multidimensional high-level arrays provide significant productivity benefits compared to their low-level counterparts, there are many cases when a programmer wishes to "view" a subset of array elements using a different index set from that of the original array *e.g.,* when a subarray needs to be passed by reference to a procedure or when a multi-dimensional array needs to be accessed as a one-dimensional array. In contrast, languages such as APL and Matlab provide restructuring operations that are value-oriented with copying semantics by default.

In this paper, we introduce *array views* for high-level arrays to address this limitation. Array views give the programmer the opportunity to work with multiple views of an array, with well defined bounds checks that are performed on the region associated with the view. A programmer can exploit the array's view to traverse an alternative representation of the array. Prevalent in scientific codes is the expression of the form *a = b[i]* which often assigns the variable *a* row *i* of array *b* when *b* is a two-dimensional array. Array views can extend this idea by providing an alternate view for the entire array. The following code snippet shows an array view example:

```
double[.] ia = new double[[1:10,1:10]];
double[.] v = ia.view([10,10],[1:1]);
v[1] = 42;
print(ia[10,10]);
```

```
//code fragment highlights \Xten{} to Java array translation
region reg_mex = [0:MESH_EXT-1];
region reg_mex_3D = [reg_mex,reg_mex,reg_mex];
double[.] x,y,z = new double[reg_mex_3D];...
for (point pt3[pz] : reg_mex) {...
  for (point pt2[py] : reg_mex) {...
     for (point pt1[px] : reg_mex) {
       x[pz,py,px] = tx; //use productive multi-D
       y[pz,py,px] = ty; //access with array views
       z[pz,py,px] = tz;
       tx += ds;
     }
     ty += ds;
   }
   tz += ds;
}...
region reg_br = [0:MESH_EXT-2];
region reg_br_3D = [reg_br, reg_br, reg_br];
int[][][] p1,p2 = new int[reg_br_3D];...
region reg_linear=[0:MESH_EXT*MESH_EXT*MESH_EXT-1];
double[.] xv = x.view([0,0],[0:reg_linear];
double[.] yv = y.view([0,0],[0:reg_linear];
double[.] zv = z.view([0,0],[0:reg_linear];
for (point pt7[i,j,k]: reg_br_3D) {
   ux = xv[p2[i][j][k]] - xv[p1[i][j][k]] ;
   uy = yv[p2[i][j][k]] - yv[p1[i][j][k]] ;
   uz = zv[p2[i][j][k]] - zv[p1[i][j][k]] ;...
}
```

Fig. 9. Array views *xv*, *yv*, and *zv* enable the programmer to productively implement 3-dimensional array computations inside the innermost loop. We highlight the array transformation of X10 arrays into Java arrays to boost runtime performance. In this hexahedral cells volume calculation code fragment, our compiler could not transform X10 arrays *x*, *y*, *z*, *xv*, *yv*, *zv* into Java arrays because the Java language doesn't have an equivalent array view operation.

```
//code fragment shows opt with array linearization
region reg_mex = [0:MESH_EXT-1];
region reg_mex_3D = [reg_mex,reg_mex,reg_mex];
double[.] x,y,z = new double[LinearViewAuto(reg_mex_3D)];...
for (point pt3[pz] : reg_mex) {...
   for (point pt2[py] : reg_mex) {...
      for (point pt1[px] : reg_mex) {
        x[pz,py,px] = tx; //use productive multi-D
        y[pz,py,px] = ty; //access with array views
        z[pz,py,px] = tz;
        tx += ds;
      }
      ty += ds;
   }
   tz += ds;
}...
region reg_br = [0:MESH_EXT-2];
region reg_br_3D = [reg_br, reg_br, reg_br];
int[] p1,p2 = new int[LinearViewAuto(reg_br_3D)];...
region reg_linear=[0:MESH_EXT*MESH_EXT*MESH_EXT-1];
double[] xv = x;
double[] yv = y;
double[] zv = z;
for (point pt7[i,j,k]: reg_br_3D) { //sub M for MESH_EXT
   ux=xv[p2[k+(M-1)(j+(M-1)*i)]-xv[p1[k+(M-1)(j+(M-1)*i)];
   uy=yv[p2[k+(M-1)(j+(M-1)*i)]-yv[p1[k+(M-1)(j+(M-1)*i)];
   uz=zv[p2[k+(M-1)(j+(M-1)*i)]-zv[p1[k+(M-1)(j+(M-1)*i)];..
}
```

Fig. 10. We show the final version for the Hexahedral cells code which demonstrates the compiler's ability to translate X10 arrays into Java arrays in the presence of array views.

The programmer declares array *ia* to be a 2-dimensional array. Next, the programmer creates the array view *v* to represent a view of array *ia*, with starting point `[10,10]` and region `[1:1]`. This essentially introduces a pointer to element ia[10,10]. Subsequently, when the programmer modifies the array *v*, array *ia* is also modified resulting in the print statement yielding the value 42. We will use a hexahedral cells code [12] as a running example to illustrate the productivity benefits of using array views in practice.

Figure 8 illustrates one problem that arises when programmers utilize an array access as a multi-dimensional array subscript. Since the subscript returns an integer, the developer cannot use the subscript for multi-dimensional arrays. As a result, the programmer must rewrite this code fragment by first replacing the 3-dimensional arrays *x*, *y* and *z* with linearized array representations. Subsequently, the developer needs to modify the array subscripts inside the innermost loop of Figure 8 with the more complex subscript expression for the linearized arrays. While this solution is correct, we can implement a more productive solution using X10 array views as shown in Figure 9. This solution enables programmers to develop scientific applications with multi-dimensional array computations in the presence of subscript expressions returning non-tuple values.

Figure 9 shows the result of converting the 3-D X10 arrays into 3-D Java arrays when analysis determines it is safe to do so. This compilation pass does not transform the X10 arrays *x*, *y*, *z*, *xv*, *yv*, and *zv* because of their involvement in the X10 *array.view()* method call. There is not a semantically-equivalent Java method counterpart for the X10 *array.view()* method. One drawback of array views as presented is that safety analysis marks the view's target array as unsafe to transform. Our compiler does convert the X10 general arrays *p1* and *p2* in Figure 9 into 3-D Java arrays. Although 3-D array accesses in Java are inefficient, this transformation still delivers more than a factor of 3 speedup over the code version with only X10 general arrays. Finally, we can achieve even better performance by linearizing the 3-D Java arrays, and optimizing away the array views by replacing them by assignments to the whole array. Figure 10 provides the final source output for the hexahedral cells code fragment.

6 Experimental Results

We performed our experiments on a single node of an IBM 16-way 4.7 GHz Power6 SMP with 186 GB main memory. The Java runtime environment used is the IBM J9 virtual machine (build 2.4, J2RE 1.6.0) which includes the IBM Testarossa (TR) Just-in-Time (JIT) compiler [25].

It is important to note that all our experiments are performed on a JVM that already performs dynamic bounds check elimination within a JIT compiler. Our improvements are relative to a state-of-the art dynamic compilation system that already removes as many runtime checks as it can.

We studied the X10 sequential ports of benchmarks from the Java Grande [16] suite. We compare two versions of each benchmark. The first version is the ported code. The second version, through static analysis, inserts *noBoundsCheck* calls around an array index when the bounds check is unnecessary.

In columns 2,3 and 4 of Table 1, we report the dynamic counts for the Java Grande, *hexahedral*, and 2 NAS parallel (cg, mg) X10 benchmarks. We compare dynamic counts for potential general X10 array bounds checks against omitted general X10 array bounds

checks using our static analysis techniques. We use the term "general X10 array" to refer to arrays the programmer declares using X10 regions. In several cases our static bounds analysis removes over 99% of potential bound checks.

In columns 5,6 and 8 of Figure 1, we report the execution times for the Java Grande, *hexahedral*, and 2 NAS parallel (cg, mg) X10 benchmarks. Column 5 shows the execution times of the baseline unoptimized code that is executed on a JIT with dynamic bounds check elimination. Column 6 shows the execution times with automatically generated *noBoundsCheck* annotations with runtime checks enabled. These annotations alert the IBM J9 VM when array bounds checking for an array access is unnecessary. Performing static array bounds analysis and subsequent automatic program transformation, we improve runtime performance by up to 22.3%. These results demonstrate that our static no bounds check analysis helps reduce the performance impact of programmers developing applications in type-safe languages. We experience a -2.4% decrease in performance for *raytracer* because the *noBoundsCheck* annotation cost was not amortized over all bounds checks along the "hot" path. This occurs when the "hot" path does not involve loops with *noBoundsCheck* annotations (hoisted out of loop). Column 8 of Table 1 shows the execution times that are obtained by running a version of the code with *all* runtime checks turned off (including bounds checks, null checks and cast checks) and illustrates how close does our optimization come to the theoretical limit on runtime checks elimination. We can conclude that we still may further improve runtime performance in some cases by eliminating other types of runtime checks such as null checks or cast checks. One interesting result to point out is the *mg* benchmark, where even though we eliminate only 5.8% of the total number of bounds checks, that does not have a large effect on performance, since only 6.1% of the execution time is spent on all runtime checks.

Benchmarks	*Array Bounds Checks Dynamic Counts*			*Sequential Runtime Performance in seconds*				
	total X10 *ABCs*	*total* X10 *ABCs eliminated*	*percent eliminated*	*baseline*	*optimized baseline*	*runtime improv.*	*all checks removed*	*max. theor. improv.*
sparsemm	2.51×10^9	2.51×10^9	100.0%	34.46	27.02	21.6%	24.01	30.3%
crypt	1.0×10^9	1.0×10^8	10.0%	9.11	9.1	0.1%	8.79	3.5%
lufact	5.43×10^9	5.37×10^9	99.1%	46.86	40.43	13.7%	39.59	15.5%
sor	4.81×10^9	4.80×10^9	99.8%	3.67	3.66	0.2%	3.66	0.2%
series	4.0×10^6	4.0×10^6	99.9%	1233.77	1226.61	0.6%	1218.39	1.2%
moldyn	5.95×10^9	4.02×10^9	67.6%	89.98	88.65	1.5%	75.21	16.4%
montecarlo	7.80×10^8	4.20×10^8	53.8%	24.64	24.41	0.9%	24.19	1.8%
raytracer	1.18×10^9	1.18×10^9	100.0%	34.79	35.73	-2.4%	33.11	4.8%
hexahedral	3.59×10^{10}	3.21×10^{10}	89.4%	15.31	12.03	22.3%	10.38	32.2%
cg	3.04×10^9	1.53×10^9	50.4%	9.73	9.34	4.0%	9.04	7.1%
mg	6.61×10^9	3.83×10^8	5.8%	31.3	30.4	2.9%	29.39	6.1%

Table 1. Dynamic counts for Array Bounds Checks and execution times for unoptimized and optimized versions of the code

Benchmark	*Sequential Runtime Performance*			
	Fortran Version	*Unoptimized* X10 *Slowdown*	*Optimized* X10 *Slowdown*	*Java Slowdown*
cg	2.58	10.43×	3.31×	1.60×
mg	2.02	46.72×	13.66×	9.53×

Table 2. Fortran, Unoptimized X10, Optimized X10, and Java raw sequential runtime performance comparison (in seconds) for 2 NAS Parallel benchmarks, obtained on the IBM 16-way SMP machine.

Finally, in Table 2, we compare Fortran, Unoptimized X10, Optimized X10, and Java execution times for the 2 NAS parallel (cg, mg) benchmarks. The Optimized X10 significantly reduces the slowdown factor that results from comparing Unoptimized X10 with Fortran. These results were obtained on the IBM 16-way SMP. Note: the 3.0 NAS Java mg version was run on a 2.16 GHz Intel Core 2 Duo with 2GB of memory due to a J9 JIT compilation problem with this code version. In the future, we will continue to extend our optimizations to further reduce the overhead of using high-level X10 array computations.

7 Related Work

Bodík et al. [6] reduce array bounds checks in the context of dynamic compilation in a JVM. They focus their optimization on program hot spots to maximize benefits and to amortize the cost of performing the analysis on a lightweight inequality graph. Though it was limited to the intraprocedural context, a dynamic interprocedural analysis framework such as [21] can be used to extend their work to the interprocedural context. Rather than modifying the JVM, our strategy is that the compiler communicates to the JVM the results of the static analysis. Suzuki and Ishihata [26] provide an intraprocedural array bounds checking algorithm based on theorem proving which can be prohibitively costly. Most JIT compilers also perform array bounds analysis to eliminate bounds checks. However, the analysis is generally intraprocedural, limiting the effectiveness. A key difference between our work and [6, 26] is that our work targets high level arrays and leverages subregion analysis for enhanced bounds check elimination.

Aggarwal and Randall [1] use related field analysis to eliminate bounds checks. They observe that an array a and an integer b may have an invariant relationship where $0 \leq b < a.length$ for every instance of class c. To find related fields, they analyze every pair *[a,b]* where a is a field with type array(1-Dimensional) and b is a field with type integer in class c. Heffner et al [14] extend this by addressing the overhead required to prove program invariants for field relations at each point in the program. By contrast, we examine every array, region, point, and integer variable. As a result, we can eliminate bound checks for multi-dimensional arrays that Aggarwal and Randall would miss.

Gupta [13] uses a data-flow analysis technique to eliminate both identical and subsumed bounds checks. Ishizaki et al. [15] extends Gupta's work by showing when bounds checks with constant index expressions can be eliminated. This algorithm relies on the assumption that all arrays have a lower bound of 0, which is generally not the

case in X10. FALCON [22], is a compiler for translating MATLAB programs into Fortran 90, that performs both static and dynamic inference of scalar (e.g. real, complex) or fixed array types. MAJIC [2], a MATLAB just-in-time compiler, compiles code ahead of time using speculation. Both the FALCON and MAJIC type inference schemes are limited compared to our precise type inference with type jump functions since neither uses symbolic variables to resolve types.

Some research in the verification community, such as the ASTREÉ static analyzer [5] and the memory safety analysis of OpenSSH [10] has focused on proving safety of array accesses, which could also be used for optimization. These analyses are much heavier weight than what we present in this paper.

The use of equivalence sets in our type analysis algorithm builds on past work on equivalence analysis [3] and constant propagation [28]. As in constant propagation, we have a lattice of height ≤ 3. By computing the meet-over-all-paths, our type inference may be more conservative than Sagiv's [23] algorithm for finding the meet-over-all-valid-paths solution. The idea of creating specialized method variants based on the calling context is related to specialized library variant generation derived from type jump functions [9]. McCosh's [20] type inference generates pre-compiled specialized variants for MATLAB. The context in which we apply our algorithm differs from McCosh since we perform type inference in an object-oriented environment on rank-independent type variables that must be mapped to rank-specific types. Without function cloning during rank analysis, formal parameters with multiple ranks resolve to $\perp$.

8 Conclusions

In this paper, we addressed the problem of subregion analysis and bounds check elimination for high level arrays. We describe a novel analysis technique that computes the subregion relation between arrays, regions and points. We used this analysis to implement an array bounds elimination optimization, which improves the performance of our benchmarks by up to 22% over JIT compilation and is very close to the performance of the code where *none* of the runtime checks are performed. We also introduced *array views*, a high-level construct that improves productivity by allowing the programmer to access the underlying array through multiple views, and described a technique for optimizing away the overhead resulting from this abstraction in some common cases.

References

1. A. Aggarwal and K. H. Randall. Related field analysis. In *PLDI '01*, pages 214–220, 2001.
2. G. Almási and D. Padua. MaJIC: compiling MATLAB for speed and responsiveness. In *PLDI '02*, pages 294–303.
3. B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *POPL '88*, pages 1–11. ACM Press.
4. R. Barik and V. Sarkar. Enhanced bitwidth-aware register allocation. In *CC*, pages 263–276, 2006.
5. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI'03)*, pages 196–207, San Diego, California, USA, June 7–14 2003. ACM Press.

6. R. Bodík, R. Gupta, and V. Sarkar. ABCD: Eliminating Array Bounds Checks on Demand. In *PLDI '00*, pages 321–333. ACM Press.
7. P. Briggs, K. Cooper, and T. L. Simpson. Value numbering. *Software Practice and Experience*, 27(6):701–724, June 1997.
8. P. Charles, C. Donawa, K. Ebcioglu, C. Grothoff, A. Kielstra, C. von Praun, V. Saraswat, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *OOPSLA 2005 Onward! Track*, 2005.
9. A. Chauhan, C. McCosh, K. Kennedy, and R. Hanson. Automatic type-driven library generation for telescoping languages. In *Supsercomputing '03*, Washington, DC, 2003.
10. I. Dillig, T. Dillig, and A. Aiken. Fluid updates: Beyond strong vs. weak updates. In *ESOP*, pages 246–266, 2010.
11. M. Fletcher, C. McCosh, G. Jin, and K. Kennedy. Compiling Parallel Matlab for General Distributions Using Telescoping Languages. In *ICASSP: Proceedings of the 2007 International Conference on Acoustics, Speech and Signal Processing*, Honolulu, Hawai'i, USA.
12. J. Grandy. Efficient computation of volume of hexahedral cells. Technical Report UCRL-ID-128886, Lawrence Livermore National Laboratory, October 1997.
13. R. Gupta. Optimizing array bound checks using flow analysis. *ACM Lett. Program. Lang. Syst.*, 2(1-4):135–150, 1993.
14. K. Heffner, D. Tarditi, and M. D. Smith. Extending object-oriented optimizations for concurrent programs. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*, pages 119–129, 2007.
15. K. Ishizaki et al. Design, implementation, and evaluation of optimizations in a just-in-time compiler. In *Proceedings of the ACM 1999 conference on Java Grande*, pages 119–128.
16. The Java Grande forum benchmark suite. http://www.epcc.ed.ac.uk/javagrande.
17. M. Joyner, Z. Budimlić, and V. Sarkar. Optimizing array accesses in high productivity languages. In *Proceedings of the High Performance Computation Conference (HPCC)*, Houston, Texas, September 2007.
18. M. Joyner, Z. Budimlić, V. Sarkar, and R. Zhang. Array optimizations for parallel implementations of high productivity languages. In *Workshop on Performance Optimization for High-Level Languages and Libraries (POHLL)*, Miami, Florida, April 2008.
19. N. Mateev, K. Pingali, P. Stodghill, and V. Kotlyar. Next-generation generic programming and its application to sparse matrix computations. In *ICS '00: Proceedings of the 14th international conference on Supercomputing*, pages 88–99, 2000.
20. C. McCosh. Type-Based Specialization in a Telescoping Compiler for ARPACK. Master's thesis, Rice University, Houston, Texas, 2002.
21. I. Pechtchanski and V. Sarkar. Dynamic optimistic interprocedural analysis: a framework and an application. In *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '01, pages 195–210, New York, NY, USA, 2001. ACM.
22. L. D. Rose and D. Padua. Techniques for the translation of MATLAB programs into Fortran 90. *ACM Trans. Program. Lang. Syst.*, 21(2):286–323, 1999.
23. M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theor. Comput. Sci.*, 167(1-2):131–170, 1996.
24. L. Snyder. *A Programmer's Guide to ZPL*. MIT Press, Cambridge, MA, USA, 1999.
25. V. Sundaresan et al. Experiences with multi-threading and dynamic class loading in a java just-in-time compiler. In *CGO '06*, pages 87–97, Washington, DC, USA, 2006.
26. N. Suzuki and K. Ishihata. Implementation of an Array Bound Checker. In *POPL '77*, pages 132–143.
27. R. Triolet, F. Irigoin, and P. Feautrier. Direct parallelization of call statements. *SIGPLAN Not.*, 21:176–185, July 1986.

28. M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.*, 13(2):181–210, 1991.
29. K. Yelick et al. Titanium: A High-Performance Java Dialect. *Concurrency: Practice and Experience*, 10(11), Sept. 1998.
30. Y. Zhao and K. Kennedy. Scalarizing Fortran 90 Array Syntax. Technical Report TR01-373, Department of Computer Science, Rice University, 2001.